Snake: Player versus Player (PvP)

Jasper Angl[1], Unnas Hussain[2]
Northeastern University
angl.j@northeastern.edu[1], hussain.un@northeastern.edu[2]

1. Abstract

Snake is a popular arcade game with a simple structure. We altered it to make a more interesting adversarial problem space. This altered version of the is called "Snake PvP". In this version, there is the same classic Snake environment, but instead of having only one snake, two snakes compete for the same apple. The snakes will die more often, as they can run into each other. However, the snakes can also respawn a set amount of times, and have more opportunities to gain apples randomly as long as they survive. We developed the game's custom environment, along with different artificial intelligence algorithms to best compete in this game.

We decided to explore reinforcement learning algorithms to control one snake agent in Snake PvP, with the discrete goal of maximizing the score that it can get before losing all of its lives. We tried multiple reinforcement learning algorithms, and compared how well each algorithm accumulated points in Snake PvP. We started with an epsilon-greedy q-learning algorithm.We found that reinforcement learning can effectively inform the snake agent to get to the apple. But with basic information about the enemy's position, the agent undervalued the apple and overvalued not-dying to the enemy. The next approach was to use a neural network with a hidden layer to calculate approximate q-values so that memory storage was not an issue. This was the best approach that we found, as the snake had the exact right information to gain points. Finally, we used an Actor-Critic model to build a policy for the agent to follow. Previous work has been done with an actor-critic algorithm for the single-player snake. (Knagg 2019) However, in our training we were getting stuck at local optima and never reached promising results. Therefore, the Deep Q Learning agent seemed to be the most effective agent at Snake PvP.

2. Introduction

The original environment of Snake is a grid world with borders and 1 apple spawning at a random location. The snake agent consists of a list of orthogonally[1] connected positions in the grid, where the first position in the list is the snake's "head" and the rest of the snake is it's "tail". The agent can travel orthogonally along the grid from its head, and has a tail that follows it's head as it moves. As the snake eats more apples, the player's score increases and the snake's tail grows by one cell. Any apple that is eaten is immediately replaced in a random grid position. The failed state of the game(the death of the snake) occurs when the snake head runs into the border of the grid or its tail. The success state of the game is typically for the snake to grow to the point that it takes up the entire screen (DeMaria and Wilson 2004).

We are implementing a modified version of Snake called "Snake PvP". In this version, there will be 2 snake agents fighting for the same apple. If either snake hits itself or the other, that snake will die and lose a certain number of points. However, the snakes will have a set amount of lives and can respawn that many times. The environment is complete and can be fully observable (although in some cases, we found that CPU memory constraints led us to treat it as partially observable for certain algorithms). The environment is also discrete, due to the limited amount of lives and grid space. Although the number of possible outcomes (the number of scores that the snake can achieve) is very large. The adversarial component of Snake PvP makes it a competitive stochastic environment.

Solution Methods

There are many possible solutions for this problem space. At its face, it seems that a snake agent can simply search the environment for the apple and travel there with pathfinders such as A* or Dijkstra's Algorithm. However, as the snake moves in the environment, it leaves behind a tail, and it must not run into that tail. This is a common premise for traditional snake AI's: how to move in the environment so that the apples are collected, but also so that the snake doesn't run into itself (Yeh, et al 2016). With Snake PvP, there is an additional adversarial component as well. To solve this, we decided to experiment with multiple algorithms outside of the realm of searching. This is mainly because searching is already a problem that is

[1] Orthogonally here means that neighbors in the grid are only neighbors if their row and/or column are the same (so diagonal cells are not neighbors)

commonly used for original Snake, and our goal was to find unique approaches for this specific problem. Therefore, we pursued three different algorithms: Epsilon Greedy Q-Learning, Deep Neural Net Q Learning, and Actor-Critic algorithm.

Epsilon Greedy Q-Learning is normally applied to relatively simple problem spaces. It is a technique where the agent develops an optimal strategy for a task using the optimal policy from its history of interactions with the environment (van Hasselt, Guez, and Silver, D. 2015). The benefit of this technique is that it often doesn't take long to train. But the stored q-table can become really large when the algorithm needs to account for many variables resulting in many (action, state) pairs. Additionally, it can be challenging for the agent to explore all the states and therefore cannot make an informed decision in every scenario. This means that we cannot have a Q-Learning agent that observes more than two or three features[2]. Applying Q-Learning to an adversarial problem is likely not the most optimal solution, since it will be impossible to store every bit of information about the enemy/adversary in the q-table.

Deep Neural Nets (DNNs) provide a similar approach to Q-learning in that we can supply specific state features to the algorithm. However the DNN does not need to store a table of every (action, state) pair. Instead, it stores layers of weights and functions that can be applied to the input features, to receive a desired output (van Hasselt, Guez, and Silver, D. 2015). So we can input much more information about the environment and the enemy snake, and get outcomes that result in the player snake getting closer to an apple or avoiding the enemy snake.

Actor-Critics are aiming to combine the benefits of policy and value based methods. They consist of an actor, which is a policy gradient (such as REINFORCE) and a critic which computes Values given the environment and the policies action. Actor-Critics usually perform best in large and continuous environments but have also shown decent performances at smaller discrete problems such as a Snake game on a 6x6 grid. (Patrenko 2018) However, the drawback of Actor-Critics and policy-based methods in general is the high variance in estimating the policy and that it often converges to a local optimal policy.

[2] This is the case where CPU memory limitations cause the environment to become "pseudo-partially observable" because we *can* observe the whole environment, but also *can't.*

3. Background

State Space Description

Snake PvP's core alterations are that the environment consists of 2 snake agents instead of 1 (a "player" snake and an "enemy" snake), and that the snakes will have lives. The basic inputs[3] to the problem is a state described with the following input parameters:

- `P_LIVES` : Number of lives that the player snake has
- `E_LIVES` : Number of lives that the enemy snake has
- `D_PENALTY` : Score penalty for losing a life
- `A_REWARD` : Score reward for an apple
- `SIZE` : the size of the world (which is always a `SIZE x SIZE` grid where `(0,0)` is the top, left corner)
- `Player` : the list of grid positions that the player snake occupies
- `Player[0]` : the grid position of the head of the player snake
- `Score`: the player's current score in the game
- `Enemy` : the list of grid positions that the player snake occupies
- `Enemy[0]` : the grid position of the head of the player snake
- `Food` : the grid position that the apple is at

Any other information about the game or snakes can be derived from these figures (for example, the length of the player snake is `len(Player)` in Python). The output (or action) of the problem is one of `LEFT, RIGHT, UP, DOWN`, which represents the global orthogonal direction that the snake head will travel in for any given state.

Algorithm Selection

In Reinforcement Learning there are two main types of algorithms. There are value-based methods and policy-based methods. Since this project is about applying Reinforcement Learning to an adversarial game, we wanted to be able to compare the two main types. The two Q-Learning algorithms mentioned before are value-based methods. We included the simplest RL and value-based algorithm (Epsilon Q-Learning) to get a good starting point that we want to build off of. As a more advanced value-based algorithm we implemented Deep Q-Learning(DQN) as it is one of the most popular algorithms when it comes to solving discrete state-action spaces. Policy-based methods in contrast try to directly find the optimal policy without needing the state-action

[3] We did not necessarily design our Python code with these exact input names. But the Python classes and methods all used these data structures or parameters.

pairs from q-values. These methods are better for continuous environments and converge faster but are less sample efficient and tend to have a high variance in training. We tried implementing REINFORCE (Monte Carlo Policy Gradient), which is the simplest policy gradient, but it ended up always converging to a policy that kills itself as fast as possible. Both policy-based and value-based methods bring out advantages and drawbacks and with the actor-critic method we wanted to take the advantage of both categories. Actor-Critic methods seemed to have worked well in normal snake games before (Patrenko 2018, Knagg 2019) so we wanted to see how good it performs in our environment.

4. Related Work

Snake is a simple arcade game, so it has had a lot of attention in the Artificial Intelligence community. One common approach has been to use genetic evolution algorithms to generate the best snake-playing policies (Binggeser 2017). While we considered this, it typically requires heavier equipment and longer periods of training. It also may not respond well to random adversarial movement, since genetic evolution tries to build a generalized policy, and generalizing a random enemy's movements would result in a large space of possible scenarios.

As previously mentioned, Snake (and even Snake PvP) could present a good opportunity for searching algorithms. There is one adversary, one target (the food), and a well defined environment. These factors could help in building a tree of all states, and using searching, minimax, alpha-beta pruning, or other techniques to determine every game's best course of actions. However, the added component of the snake having multiple lives means that as the game goes on, the snake may want to alter its general policy. This means that the search will start on one path in its search and may not be able to use its previous searching to know the optimal next step. Furthermore, these techniques have already been done with Snake, and yield smart agents. (Dattu Appaji 2020). However, we think Reinforcement Learning is an interesting field for games, as the computer is learning its own strategies for the game; compared to searching algorithms, where we would define the strategy. For these reasons, we decided to explore Reinforcement Learning methods, such as Q-Learning, DQN and Actor-Critics, instead.

5. Project Description

Using the pygame library, we developed a custom visual environment for Snake PvP in Python. The game consisted of a Snake and Enemy class, containing their own position values, functions for moving and observing the enviornment, and constants for the learning algorithms. We also developed a Food class that was similar to the Snake class, but only contained one position value.

Epsilon Q-Learning

Q-Learning is a technique where the agent develops an optimal strategy for a task using the optimal policy from its history of interactions with the environment, stored as a q-table with entries for every (action, state) pair (van Hasselt, Guez, and Silver, D. 2015). However, on standard devices, the q-table cannot be too large, and when dealing with a 2-dimensional grid, useful position data already requires 2 axes on the table. Therefore, we choose to store distance variables in the table, the distance from `Player[0]` to `Food` in the x and y axes, and the distance from `Player[0]` to `Enemy[0]` made up the states stored in the table. Each state has four, at the beginning arbitrarily chosen, reward q-values. One for each possible action to take in that state.

The algorithm then runs through a number of episodes with different observed states and received rewards, to further improve the q-values of the table. The algorithm balances exploration and exploitation with the epsilon-greedy exploration strategy which chooses random actions at a declining rate as training continues. At every step the table updates its q-values to make sure that in the future, the agent is well informed. Algorithm 1 shows the action selection.

```
1.   Init Q-table(state) -> action-values
2.   For a set number of episodes:
3.     obs := (Player[0]-Food,
4.                   Player[0]-Enemy[0])
5.     If randInt > epsilon:
6.        action := argmax(Q-table[obs])
7.     Else:
8.        action := random choice
```

Algorithm 1: Determining which action to take based on an epsilon value and the q-table

The if-statement from lines 5-8 in Algorithm 1 indicates a greedy approach that implements our desired balance between exploration and exploitation. Once the action to take is determined, the game state is updated based on that action. Now, the player may have been rewarded or penalized. So the q-value of the action taken for the observation that led to that action is updated with Algorithm 1 based on the Bellman Equation and two constants, **lr** and **γ**. lr is the learning rate that determines how quickly the algorithm will override its previous q-values with the learned q-values. **γ** is the discount factor, which models the fact that future rewards are worth less than immediate reward.

```
1.  old-q = Q-table[obs][action]
2.  new_obs = (player-food, player-enemy)
3.  new-q = (1 - lr) * old-q + lr *
4.      (Reward + γ * argmax(Q-table[new_obs])
5.  Q-table[obs][action]:= new-q
```

Algorithm 2: Updating the q-value of the taken action to reflect what happened when that action was taken

Notice that lines 3 and 4 of Algorithm 2 contain the standard equation of a q-learning algorithm (van Hasselt, Guez, and Silver, D. 2015). These two algorithms make up the core of the Epsilon Q-Learning that we explored for Snake PvP. We developed a *qlearning.py* script that trained the AI and outputted the filled out Q-table in a pickle file. We also developed a *play_agent.py* script that reads the pickle file and controlled a snake AI for a human to play against it. *qlearning.py* contained our implementations of Algorithm 1 and Algorithm 2 in Python.

Neural Net Q-Learning (NNQ or DQN)

The neural net that we used to implement the neural net q-learning was an unsupervised learner. It's output was not labeled, but rather made a logical classification of the data. If the output of the DNN got the snake head closer to the apple (including eating the apple) or farther from the enemy, the output was considered successful. This paradigm was used to develop the training data for the neural network. We used TensorFlow's tflearn library (http://tflearn.org/) to create the neural network. We experimented with different inputs for the input layer, and had a hidden layer that had a number of nodes equal to the square of the number of inputs[4]. The neural net assigns each of its node connections with a weight, and those weights are tuned to optimize the outputs for the training data (Korolev 2017). The "q-learning" aspect came from the fact that the TensorFlow neural net was actually developing weights for its own internal q-value equation. Essentially, instead of the q-values being stored in a table, the q-values are calculated with the neural network's hidden layer, and the output layer determines what action to choose based on that value.
For the tensor flow library, we developed a more compatible SnakeGame class that contained the Snake PvP elements and was able to "step" two game moves at a time. This was more compatible with the tensor flow library we used. For the neural network, we followed the TensorFlow tutorials to develop the deep_qlearning.py script (Korolev 2017) and modified it to work with our Snake PvP environment. The first key component that was developed was generating the (randomly acting) initial population for the training. As stated previously, if an action got the snake closer to the `Food`, farther from the `Enemy[0]`, or an increased `Score`, then the action was a success and was stored as a value of 1. If the action caused the snake to die (lose all lives, not just one), the action was stored as a -1. Otherwise, it was stored as a 0. This was an early decision, and after brief informal experimentation, it was determined to not store every loss of life as a -1. Algorithm 3 shows the training data building, where line 9 is defined by what we defined as the snake "succeeding" in its action..

```
1.   Init training_data
2.   For a set number of games:
3      Start a new game
4.     For a maximum number of game steps:
5.       game.step()
6.       obs := game.observe()
7.       If Player lost
8.         training_data.append(obs, -1)
9.       Elif Player does better
10.        training_data.append(obs, 1)
11.      Else
           training_data.append(obs, 0)
```

Algorithm 3: Developing the training data for the neural network

Notice in line 6 of Algorithm 3, we observe the game state. This observation is what comprises the input layer of the neural network, which was experimented with (see Section 6). The different possible inputs that we developed for the neural net's input are shown in Table 1.

[4] We only had one hidden layer in our neural network, therefore the neural network was not deep. However, our code base may sometimes refer to it as "deep", "DNN" or "DQN" to differentiate it from the non-neural-network q-learning algorithm or the actor-critic neural-network.

Obs #1	If there was a border to the left (True or False)
Obs #2	If there was a border to the right (True or False)
Obs #3	If there was a border to in `Player`'s current direction (True or False)
Obs #4	The angle between `Food` and `Player`'s current direction (float)
Obs #5	The angle the `Enemy[0]` was in relation to `Player`'s current direction (float)
Obs #6	The angle between the center of mass of `Enemy` (shown in Algorithm 4) and `Player`'s current direction (float)
Obs #7	The x,y distances between `Player[0]` and the center of mass of `Enemy` (float, float)

Table 1: The possible input observations for the neural network

```
1.  center_x := average(x-values in Enemy)
3.  center_2 := average(2-values in Enemy)
4.  Return (center_x, center_y)
```

Algorithm 4: Determining the "center of mass" of `Enemy`

Different combinations or permutations of these inputs represent the game observation used in line 6 of Algorithm 3. The neural net actions, as Korolev's TensorFlow tutorial (Korolev 2017) outlined, are relative to the thing being trained, `Player`. Therefore, the actions the neural network outputted represented "go straight, turn left, or turn right". These were converted to the actions that our environment expects (global `UP, DOWN, LEFT, RIGHT` commands).

Advantage Actor-Critic

As mentioned previously the advantage actor-critic algorithm consists of an actor and a critic. The critic estimates the value function which in our case is the state value V. The actor updates the policy distribution in the direction suggested by the Critic with a policy gradient. This algorithm makes use of an advantage value A, which determines how much better it is to take a specific action compared to the average, general action at the given state.(Yoon 2019) This algorithm builds off of the Policy Gradient equation and the Monte Carlo Policy Gradient, where $\nabla_\theta J(\pi_\theta)$ denotes the policy gradient. G_t, which denotes the cumulative reward of will be replaced in our actor-critic method by the advantage function. This advantage function will act as a baseline helping to limit the high variance and increase stability.

$$\nabla_\theta J(\theta) = \mathbb{E}_\tau[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) G_t]$$

$$A(s_t, a_t) = Q_w(s_t, a_t) - V_v(s_t)$$

Because of the relationship between Q and V values the advantage function can be rewritten as this:

$$A(s_t, a_t) = r_{t+1} + \gamma V_v(s_{t+1}) - V_v(s_t)$$

Leaving us with the final equation of:

$$\begin{aligned}\nabla_\theta J(\theta) &\sim \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t)(r_{t+1} + \gamma V_v(s_{t+1}) - V_v(s_t)) \\ &= \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) A(s_t, a_t)\end{aligned}$$

Algorithm 5: Actor-Critic Equation

For the actor-critic algorithm we modified a PyTorch tutorial code to develop our own actor_critic.py script. (Tabor 2019) The actor and the critic both share the same neural network structure as this is a convenient way to increase sample-efficiency. Instead of training two completely separate networks for the value and policy we combine the base layers and have different output layers. The neural net consists of two hidden layers and two separate output layers. One for the states' policy (actor) and one for the states' value (critic). The network learns based on the reward received at every state not only at the end of an episode, in contrast to the Monte Carlo Policy Gradient (REINFORCE). For updating our policy gradient we compute the advantage using future reward r, discount factor gamma γ, current Value V at state s and future Value V at state s + 1 of the critic:

```
adv = reward + gamma * critic_val(new_state)
        - critic_val(state)
actor_loss = log_prob * adv
critic_loss = adv * adv
ac_loss = actor_loss + critic_loss
Ac_loss.backward()
ac_optimizer.step()
```

Algorithm 6: Updating actor-critic

This pseudocode in PyTorch shows how we compute the advantage and the loss functions which are being used to backpropagate and update the neural net. Note how the `adv` variable correlates to A(s,a) from Algorithm 6 and how the log probability is multiplied to receive the actors gradients.

Similar to the previous algorithms we used two different observation spaces. The first, being a 5x5 grid surrounding the snake's head. Each cell within that grid will be represented by [0,1,2,3] depending on what occupies the cell. Nothing, the snake itself, the food or a game resetting block (Enemy or Border). With that observation space the agent would have a more difficult time finding the food than an agent who knows the food location, but it should be better at avoiding obstacles.

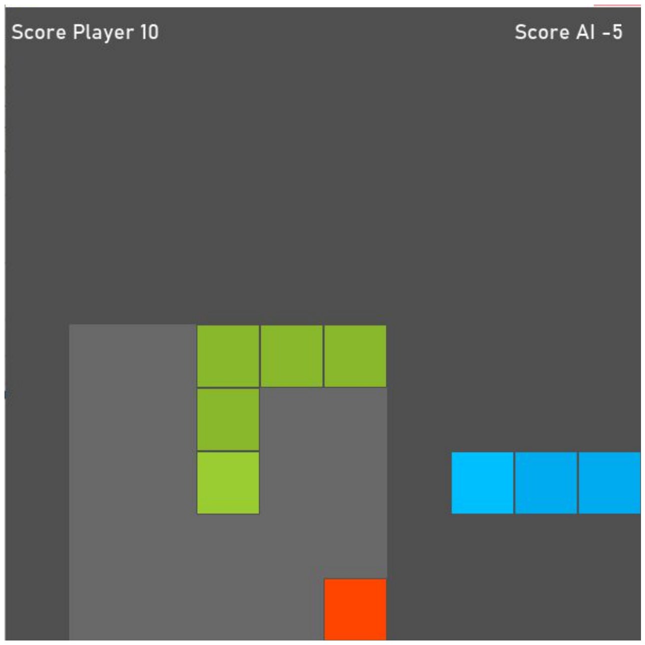

Figure 1: Bright cells represent observation space

```
EmptyCell = 0, SnakeCell = 1, FoodCell= 2,
DeathCell = 3
```

The second observation space is similar to the one from Deep Q-Learning and is limited to 6 observations which should help the training by creating a simpler state space. However, it will be limited in the complexity of its movements and therefore get stuck in its own or enemies body.

Obs #1	If there was a border to the left (True, False)
Obs #2	If there was a border to the right (True, False)
Obs #3	If there was a border in `Player`'s current direction (True, False)
Obs #4	The angle between `Food` and `Player`'s current direction (float)
Obs #5	The angle the `Enemy[0]` was in relation to `Player`'s current direction (float)
Obs #6	The relative distance from `Player[0]` to `Food`

Table 2: Input observations for the AC neural network

6. Experiments

In experimentation, we wanted to determine which of the different AI techniques best performed in Snake PvP, on average. We developed a simple enemy snake that followed Algorithm 6E (essentially always heading directly towards the `Food`, and sometimes making a random choice).

```
If random enemy > 0.5
    Enemy moves in random direction
Elif Food is above Enemy[0]:
    Enemy moves UP
Elif Food is below Enemy[0]:
    Enemy moves DOWN
Elif Food is to the right of Enemy[0]
    Enemy moves RIGHT
Else
    Enemy moves LEFT
```

Algorithm 6E: The test enemy's movement algorithm

Each trained algorithm ran as the `Player` 200 games against the Algorithm 6E `Enemy`[5]. Eating an apple gave the player a score increase of 5, and losing a life resulted in a score decrease of 10. The player had 5 lives, but the enemy had infinite lives (since it was running a very simple algorithm, it needed infinite lives to test the player's algorithm properly). Some training included small (sometimes fractional) penalties for moving to ensure that the player did not stall, but the 200 test games did not include that penalty

[5] The enemy snake running this algorithm is sometimes referred to as the "6E enemy" in this paper.

Epsilon Q-Learning Results

The basic q-learning agent had two different versions. The first version only observed how far the `Food` was, and the second observed how far the `Enemy[0]` was in addition. The first had a more concrete policy based on its inputs, because it's q-table was smaller, so the training was more complete. The second was larger, so the training took much longer. Figures 2a and 2b show the training of these agents.

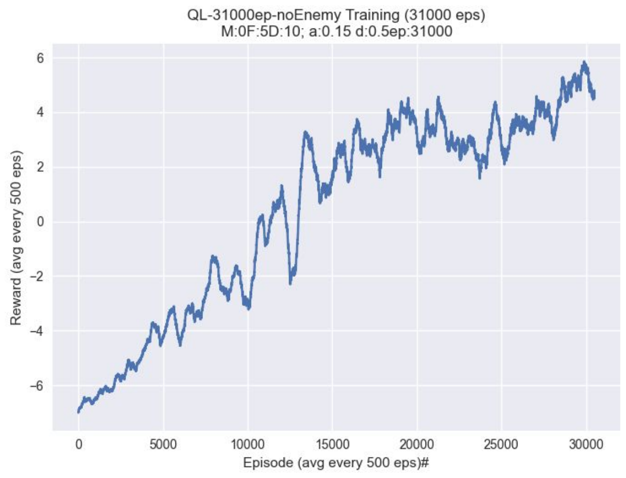

Figure 2a: The reward progression of the no-enemy Q-learning agent as trained over 31000 episodes

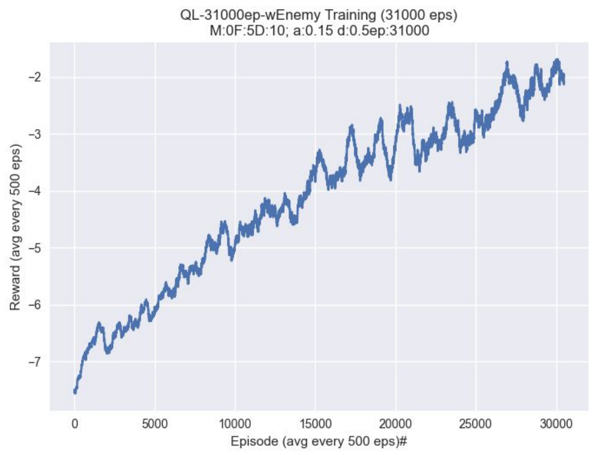

Figure 2b: The reward progression of the enemy-observing Q-learning agent as trained over 31000 episodes

The two agents trained simultaneously, however the agent with no observation of the enemy trained much better. Whereas the second agent took very long to get a positive valued reward. After training, we ran the agents against the 6E enemy in 200 games. Table 3 shows the results, and Figures 3a and 3b show those results as graphs.

Agent Description	Average Score	Max Score
Enemy-Observing Q-Learning	-15	10
Non-Enemy-Observing Q-Learning	-6.0	52

Table 3: The q-learning agents performance over 200 games

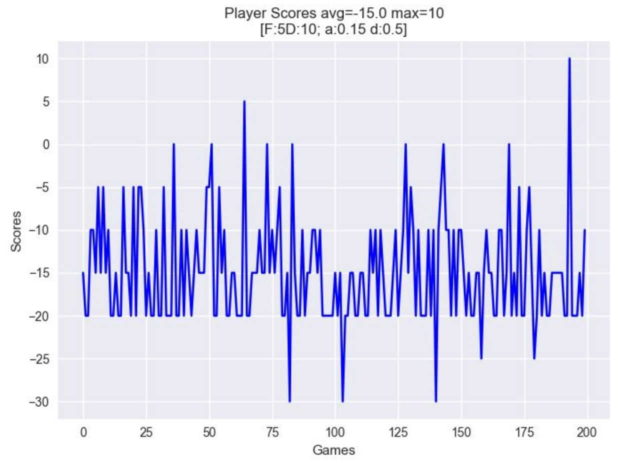

Figure 3a: The Q-learning Player's score over 200 games while observing the enemy

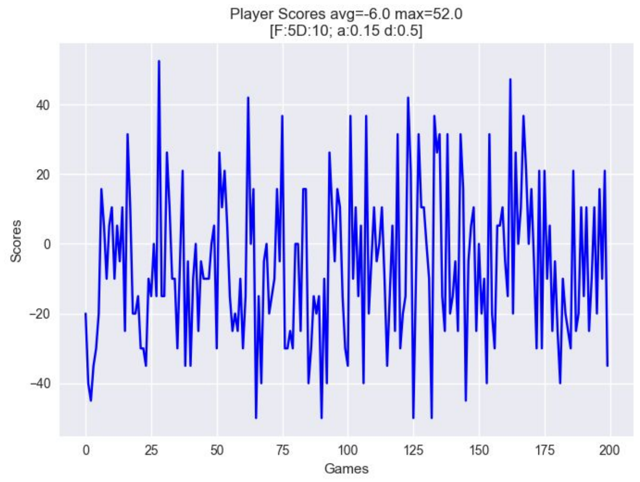

Figure 3b: The Q-learning Player's score over 200 games while ignoring the enemy

Interestingly, the Q-learning agent performed best when ignoring the enemy. This may be because the Q-learning algorithm punishes the player's reward function when they die, and the agent correctly observed that the enemy is the most common cause of death. However the player only knows it's distance to the enemy, since that is all the table can efficiently score. Therefore, if the enemy was too close to the apple, the player agent deduced that the apple was in a dangerous location, and it would not attempt to gain points. The Q-learning algorithm caused the player to overestimate the enemy's effect in succeeding in this game. This is further supported by the ignoring-enemy agent. As it was able to slightly increase its average score (and more often exceed at the game) because Q-learning was enough to inform the agent how to get to the apple in multiple scenarios, and this agent wasn't "afraid" to do so. This agent succeeded when the apple was close enough to it that the enemy agent was not effective, but commonly failed in all other scenarios.

Neural Network Q-Learning (DQN) Results

Similar to the Q-learning agent, there were experimentations made with what the agent actually observed. The neural net is not as space-intensive as the q-table is, therefore information about the enemy is always able to be inputted. However, we saw from the Q-learning agent that the type of information was important. The neural net's input layer always used the grid border and food angle (items 1-4 in Table 2) as inputs. In addition, we experimented with different ways to inform the agent about the enemy's position or length in the world. Table 4 shows the different permutations and results of the agent playing 10,000 games.

Additional Observations	**Average Player (Score, Length)**	**Max Player (Score, Length)**
Dist to Center of Enemy	(16.84, 3.64)	(59.6, 9)
Dist to Head of Enemy	(22.6, 3.79)	(93.0, 11)
Angle to Head & Dist to Center	(24, 3.97)	(62.4, 11)

Table 4: Results of training the neural net with different observations

As shown by the table, the best observation permutation for the snake, on average, was to observe the angle to the enemy head, and the distance from the enemy's center of mass. This is likely because the player can be cautious of the enemy, but still have the relevant information to *navigate around* the enemy, rather than trying to completely avoid it. This agent was the final neural net agent used in testing[6].

In the actual test, the neural net agent with the "Angle to head (of the enemy)" and "Dist to Center (of the enemy)" inputs was put against the 6E enemy in 200 games. Figure 4 shows the scores (the blue line) as well as the lengths[7] (the green line) of the player agent over those games. The red line in Figure 4 is the enemy's length.

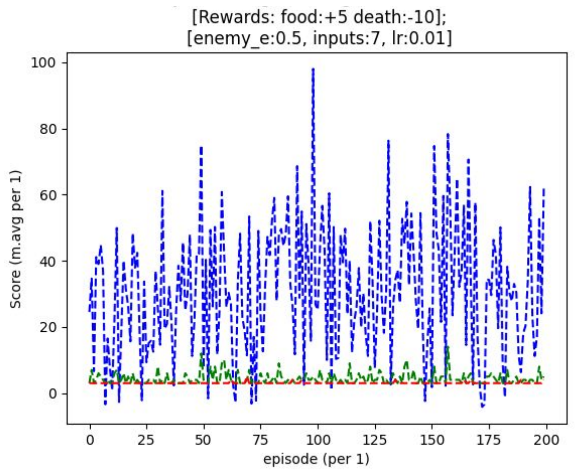

Figure 4: The NN Player's score (blue) and length (green) over 200 games

This was an extremely successful run for the neural net player. The neural network agent had the knowledge that the Q-learning algorithm had about getting to the apple. However, the neural network also provided the agent with observations about the enemy, and how to accurately avoid the enemy, and stay alive. Figure 5a shows a scenario where the enemy snake (the blue snake) is right next to the apple (the red square), but the player snake (the green snake) is still able to navigate to it.

[6] This final neural net had 7 nodes in its input layer and 49 nodes in its hidden layer.

[7] The "length" of a snake is the amount of positions it occupies. In Python it would be represented as `len(Player)` or `len(Enemy)`.

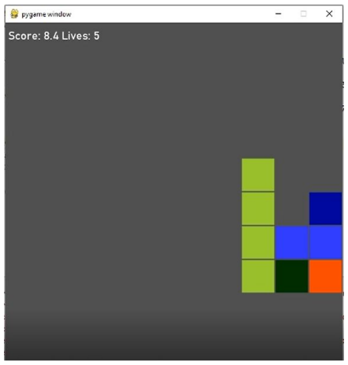

Figure 5a: The neural net agent (green snake) navigating to the apple in a "dangerous" scenario

In observing a few select games, it was also apparent that the snake has developed a sense of its lives. Figure 5b shows a scenario in which the snake has just collected an apple and is now very large. This means it will be difficult for it to navigate. The player snake then makes the choice to run into self, even though it doesn't have to. This may be due to the fact that it has all 5 of its lives left. Therefore, it could be advantageous for it to reset to a smaller size, and still have lives left over to gain points[8].

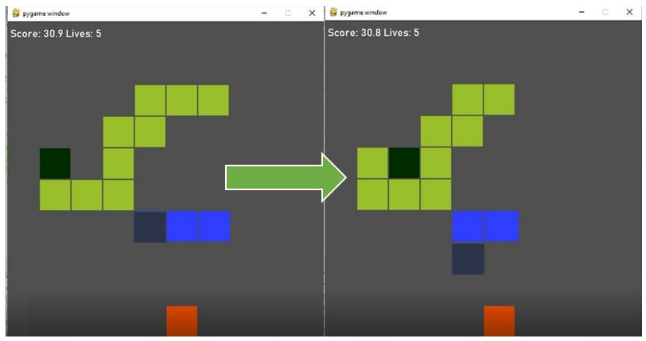

Figure 5b: A scenario in which the player just collected an apple, and then runs into self, despite having plenty of space to navigate.

Table 5 shows the formal result of this agent, comparable to the other algorithm agents.

Agent Description	**Average Score**	**Max Score**
7-input Neural Net	65.00	100

Table 5: The neural-net agent's performance over 200 games

[8] Of course it is impossible to know the exact reasoning of the agent. However, this is a likely explanation.

Actor Critic Results

We tested the advantage actor critic model using the two different observation spaces mentioned previously. (Figure 1 and Table 2) Both used the same amount of episodes, reward parameters and network structure of 2 fully connected hidden layers of size 128 and 64.

```
# of episodes: 5000
# reward for food: +1
# reward for death: -1
# move penalty: 0.01
```

The first observation space of a 5x5 grid performed worse than the agent with only 6 features. It appeared to have learned the policy of dying fast by simply turning right all the time, making it die within 7 steps for each life. This resulted in an average number of 34.8 moves and an average score of -49.1.

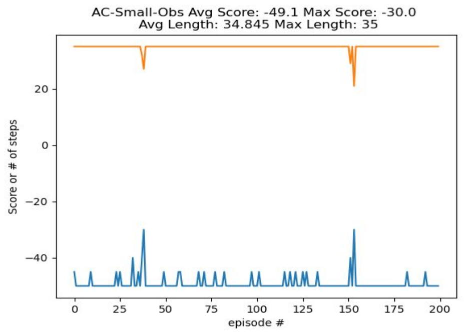

Figure 6a Showing results of a 5x5 Grid for Actor Critic

The second observation space of only 6 features performed mildly better than the first one due to a different learned policy. The fact that it almost never reaches a score of above 0 shows that it seems to have learned how to survive in the grid but doesn't really know how to reach the food.

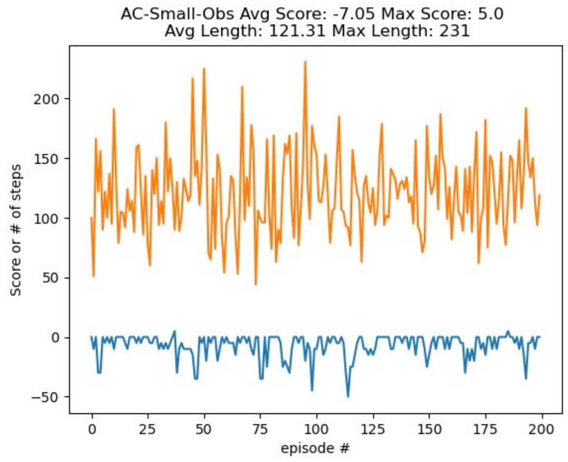

Figure 6b Results of 6 input features for Actor Critic

For both observation spaces we tried hyperparameter tuning the hidden layer number, hidden layer size, learning rate and discount factor gamma. All of them did not have an impact on the outcome of the training. Training the second observation space for a longer period of time also did not affect the trained policy.

The actor-critic had a difficult time learning which comes down to some predictable and some unpredictable issues. Since Actor Critics are not as sample efficient as value based methods like Q-Learning and Deep Q-Learning, it takes a lot of episodes to train them until they reach a good or optimal policy. We ran the algorithm for a maximum of 50.000 episodes, which didn't bring out any better results. Testing for 50.000 episodes took almost 12 hours. That is due to my slow laptop who can only train on the CPU and not GPU. For comparison, an article documented how Snake was implemented using A2C. They also observed either the partial grid with fully connected hidden layers or observed the full grid using convolutional layers. While they received decent results they also trained the environment for 5 million episodes.(Knagg 2019) With our hardware resources we were not able to get that much training data and therefore also did not receive good results. With the smaller observation space we tried to limit the amount of variables and increase learning that way, which has worked to some degree. However, it was still not performing to its expectations. The issue of policy gradients converging to a local optimal policy, instead of a global one, appeared to be the case for both of our actor critic methods, and we were to resolve that issue. It is still surprising that after some a few thousand iterations the policy either converges to dying directly or just trying to survive in the game and is unable to find a better way to navigate in the grid world.

7. Conclusion

Agent Algorithm Description	**Average Score (over 200 games)**	**Maximum Score (over 200 games)**
Enemy-Observing Q-Learning	-15.0	10.0
Non Enemy-Observing Q-Learning	-6.0	52.0
7-input Neural Network	65.00	100.0
Actor-Critic (Grid Network)	-49.0	-30.0
Actor-Critic (6-input Network)	-7.05	5.0

Table 6: A summary of how each algorithm performed in 200 games with 5 lives against the “6E enemy”

In this project we explored different reinforcement learning techniques to explore our unique PvP Snake environment. Epsilon-greedy Q-Learning performed well given its limited state space. Further improvement could have been made using different exploration algorithms and adding binary features similar to the DQN-algorithm (the 7-input neural network). DQN-learning performed by far the best out of the three algorithms. Given the environment it was predictable that DQN would be suited the best. Although Actor-Critics and Policy Gradients have their biggest benefits in continuous and large environments, it was surprising that the actor critic performed with average scores below 0. A more powerful hardware that supports training on the GPU would have been helpful to allow bigger state spaces and more training episodes.

8. References

Binggeser, P. 2017. Designing AI: Solving Snake with Evolution. *Becoming Human*. [Online]. Available: https://becominghuman.ai/designing-ai-solving-snake-with-evolution-f3dd6a9da867

Dattu Appaji, N.S. 2020. Comparison of Searching Algorithms in AI Against Human Agents in Snake Game.

Thesis, Department of Computer Science, Blekinge Institute of Technology, Sweden.

DeMaria, R., Wilson, J. 2004. *High Score!: The Illustrated History of Electronic Games*. Computer Games Series. McGraw-Hill/Osborne. https://books.google.co.jp/books?id=HJNvZLvpCEQC

van Hasselt, H.;Guez, A.; Silver, D. 2015. Deep Reinforcement Learning with Double Q-Learning. arXiv preprint. arXiv:1509.06461 [cs.LG]. Ithaca, NY: Cornell University Library

Knagg O. 2019. Learning to play snake at 1 million FPS. *Towards Data Science* [Online] Available: https://towardsdatascience.com/learning-to-play-snake-at-1-million-fps-4aae8d36d2f1

Korolev, S. 2017. Neural Network to Play a Snake Game. *Towards Data Science*. [Online]. Available: https://towardsdatascience.com/today-im-going-to-talk-about-a-small-practical-example-of-using-neural-networks-training-one-to-6b2cbd6efdb3

Patrenko A. 2018. Advantage Actor-Critic solves 6x6 Snake (Reinforcement Learning) *YouTube* [Online]. Available: https://www.youtube.com/watch?v=bh_5aIqVTUY

Tabor, P. 2019. PyTorch Actor-Critic Discrete. *Github.* [Online]. Available: https://github.com/philtabor/Youtube-Code-Repository/blob/master/ReinforcementLearning/PolicyGradient/actor_critic/torch_actor_critic_discrete.py

Yeh, J.;Su, P.;Huang, S.;Chiang, T. 2016. "Snake game AI: Movement rating functions and evolutionary algorithm-based optimization," Conference on Technologies and Applications of Artificial Intelligence (TAAI), Hsinchu, 2016, pp. 256-261, doi: 10.1109/TAAI.2016.7880166.

Yoon, C. 2019 Understanding Actor-Critic Methods. *Towards Data Science* [Online]. Available: https://towardsdatascience.com/understanding-actor-critic-methods-931b97b6df3f