

Creating a Snake AI

Adrian Kawa

Harrisburg University

### **List of Figures**

Figure 1.1. Bellman equation used for Deep Q-Learning

Figure 1.2. Boolean value representation for an action

Figure 1.3. Boolean value representation for a state

Figure 1.4. My neural network/model used to predict actions

Figure 1.5. Pseudo-code for agent algorithm

Figure 1.6. matplotlib graph showing my AI results

Figure 1.7. pytorch code review

Figure 1.8. general algorithm code review

## **Introduction**

For my final project in this artificial intelligence course, I chose to make a version of snake where I can test and make an AI that plays snake for me using reinforcement learning and neural networks. Reinforcement learning is a machine learning training method that is based on a reward/punishment system. This means my AI will learn by doing the correct or incorrect action in my game which would be something I design in my code that gives the AI and reward/punishment with each action. Neural networks are a subset of machine learning that will essentially mimic a simple human brain to complete my goal of an AI that plays snake. Neural networks use a variety of input and output nodes with also inner layer nodes that will take in and then output data that the AI will use to make decisions in the game. To make my agent I chose to use python/pytorch to complete my project, as well as a couple more tool and libraries like gym and matplotlib that I found out through my research. My project relates to artificial intelligence because machine learning is a subset of AI, also reinforcement learning is a method of machine learning which is the type of machine learning I will be using. Through my research I quickly saw a method of learning for snake AI's which was deep q learning, I chose to focus on this method the most in my research. This project would tie into our fourth unit in the course which is about neural networks.

## **Motivation**

I believe my future career path should go towards game programming and while going into this course I wanted to learn more about video game AI so I could learn to create one for myself and potentially implement an agent into my future games. I've also been super interested

in how AI's learn and play video games on their own through video games and other forms of media that I enjoy taking part in. When playing video games during my younger years I loved to play a lot more competitive and so learning about game trees and AI intrigued me a lot in this course since I'm naturally looking to become extremely good at a video game whenever I play one, which I can implement through AI.

### **Reinforcement Learning**

A definition of reinforcement learning is learning what to do in a certain environment or learning to map situations to actions in order to maximize a reward for the learner (Sutton, R. S., & Barto, A. G. (2018) pg. 2). Learners aren't directly told how to play a game since that wouldn't be a way of reinforcement learning. Instead, they must discover on their own how much value their actions gain for them or learn to generate actions that will lead to greater rewards in future actions. This creates many trial and error situations for AIs that learn off this method of machine learning. One large challenge that reinforcement learning agents take on is the trade-off between exploration and exploitation. This is a dilemma that learners and mathematicians take on since, in order to receive large value through actions, the agent must exploit large value moves that it's learned in the past, but for the agent to learn these moves it must explore a large variety of moves to learn what's good or bad. This makes it tough to know when's the correct time to focus on exploiting or exploring. This can relate to the hill-climbing algorithm that we learned in class where if learners start focusing on exploiting a certain way of moving then there might be a better value from a different set of moves that could be learned in the future with more exploration (a larger hill that the learner doesn't see from its position). Elements of reinforcement learning are a key takeaway for building an AI, where the main elements were the agent and environment and sub-elements of policy, reward signal, value

function, and a model (Sutton, R. S., & Barto, A. G. (2018) pg. 6). The agent is the learner or AI that's given a situation with a goal in mind the entire time in order to learn through its environment. The environment is basically the game or situation that's given to the agent, where they learn through the given environment to obtain their goal. A policy is the general current rule/rules that it's currently following in its iteration of training. In my results, I found that there's always a learning phase where the agent explores many actions near the start of training. This can be an example of its behavior during the start of training. A reward signal is a goal that's given to the agent in order to select actions that get it closer to obtaining its goal. This comes in the form of giving your agent larger or smaller rewards depending on the action they chose when compared to their overall goal. This reward system will ultimately change the agents' policies since the policy is what chooses which action to take that either gave the agent a low or high-valued reward. The value function relates to the reward signal since it refers to the long-term reward estimation rather than the immediate reward from the reward signal. In other words, the value function finds the value of a set of moves found from the current state of the game where it's predicted to find the largest value, where the reward signal is the immediate reward given to the agent based on its action. This relates to humans where we receive pleasure (large reward)/pain (small reward) based on immediate actions like eating fast food or getting a poor grade on a test, we can also estimate getting more pleasure from taking the time to cook our own meal and perfect a dish. Estimating value in moves is such an important concept in reinforcement learning since it allows for much more effective algorithms that will find the best path to obtain the agent's goal. This compares to the best chess players in the world that look down the double-digit number of moves in order to achieve the best outcome for themselves. Finally, the model is a way of planning different actions given the state of the current environment. When the model is

given a state of the game, it uses its past experiences to predict an action for the agent to take.

During the early iterations, the model is almost picking random actions since it doesn't have past experience to go off of, this is a way of exploration that the agent can take part in by using a model.

### Reinforcement Learning Algorithms

Over my research, I came across many algorithms to use to tackle my Snake AI. A couple I came across were Monte Carlo which focuses on policy evaluation and policy improvement, and many different versions of Q-learning which uses Q values to estimate the reward of a certain action at a certain state. Deep Q-learning takes this a step further but uses a neural network to predict Q values while using a similar technique Q-learning uses. The majority of examples that tackled this snake AI problem used deep Q-learning as their method of reinforcement learning, which is a large reason why I chose to also use deep Q-learning for my AI. An important view for Deep Q-learning is the use of memory and past actions/outcomes in their way of updating the Q values. All these Q-learning methods use an equation to estimate and update Q values for each iteration of the environment, the Bellman's equation is a popular equation used for doing such.

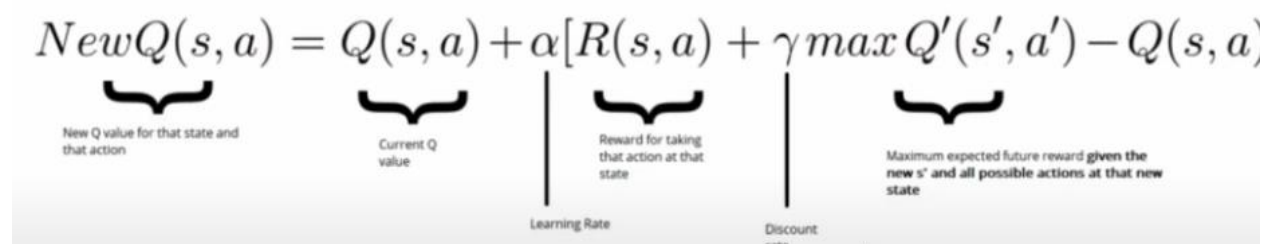
$$NewQ(s, a) = Q(s, a) + \alpha [R(s, a) + \gamma \max_{a'} Q'(s', a') - Q(s, a)]$$


Diagram illustrating the Bellman's equation for Q-learning, showing the components of the equation and their corresponding labels:

- $NewQ(s, a)$ : New Q value for that state and that action
- $Q(s, a)$ : Current Q value
- $\alpha$ : Learning Rate
- $R(s, a)$ : Reward for taking that action at that state
- $\gamma$ : Discount rate
- $\max_{a'} Q'(s', a')$ : Maximum expected future reward given the new  $s'$  and all possible actions at that new state

Figure 1.1

In order to explain Bellman's equation, learning what states are first is important to understand this diagram. States which are shown as  $s$ , are instances of the environment where the agent has multiple moves to choose from depending on where they are in that current instance. For

example, Snake has an easy to create state since the entire game is played on a grid of squares where the snake can be  $x$  number of squares long and  $x$  number of squares away from a piece of food. A state can show the agent if there's danger around him, where they are on the map, where is the food relative to their position, and more. Another variable in this equation is an action, shown by  $a$ , which is the move taken from the agent. The bellman equation uses pairs of states and actions to update new  $Q$  values, which are shown from  $Q(s,a)$ . It's important to note that at the beginning of training an agent, these  $Q$ -values normally either start as random or zero values which is what creates exploration in our agent. Over time however due to updating and using the equation,  $Q$  values are more and more accurate in predicting the value of state and action pairs. The  $\alpha$  is used for the learning rate which is a number between 0 and 1. This would be the rate that which your  $Q$  values will update, a number closer to 0 will update in much smaller jumps with the opposite being true with a number closer to 1. The function of  $R(s,a)$  is the immediate predicated reward from the state-action pair. The  $\gamma$  or gamma is a number between 0 and 1 that is used to indicate in the current state how far you want the agent to look down a set of moves. A number closer to 0 would look for the immediate reward while a number closer to 1 would look for the reward or value of a path farther in the future. The final function in the equation uses  $\max Q'(s',a')$  as the function to predict the maximum reward for the next upcoming state and action pair. The  $'$  is used to indicate that a state or action is a future one which uses how the equation uses memory to improve its  $Q$  values. Putting the entire equation together you get the updated  $Q$  value that's used for the next iteration of the agent.

### **Creating my AI**

By using a deep Q-learning method I can start looking at how to create my AI. Starting off with the most fundamental elements of my AI I need an environment, a reward system, a way of

showing actions, a way of showing the current state, and a model. The environment is easy since I already chose to make a snake ai that would be the game “Snake”. The reward system should be based on the goal of the environment, which is to fill the entire map with the snake body from eating food that makes the snake grow. Based on this goal my agent should be given a positive reward for eating food and a negative reward for dying. The specifics that I chose to start with were +10 for eating food and -10 for dying. Changes to this element and others as well changed a lot during my testing phase on my specifics. For showing my action I saw in my research that it would be easier for the snake to have three actions instead of four where the actions were to go forward, turn left, or turn right, which would look like:

**Forward = 1 or 0**

**Turn left = 1 or 0**

**Turn right = 1 or 0**

Figure 1.2

Where 1 is equal to firing the neuron and 0 is to not fire. I wanted to also look at having four actions in my testing to see how it would affect the learning. Next is a way of showing the current state, which the easiest way I found in my research was to show a Boolean value for each individual information given in the state. My research led me to start with the direction of where the snake is facing, the direction of danger, and the direction of where the food was all relative to the snake.



My state would look like this:

Danger forward = 1 or 0

Danger left = 1 or 0

Danger right = 1 or 0

Facing up = 1 or 0

Facing right = 1 or 0

Facing down = 1 or 0

Facing left = 1 or 0

Food is up = 1 or 0

Food is right = 1 or 0

Food is down = 1 or 0

Food is left = 1 or 0

Figure 1.3

Finally, the model, which takes the state of inputs and has an output of an action that is directly correlated to the number of Boolean values my state and actions have. The select number of neurons is debatable in each layer but the most debatable layer is the hidden layer which I saw in my research to use 250 neurons, but I also changed this later in my testing.

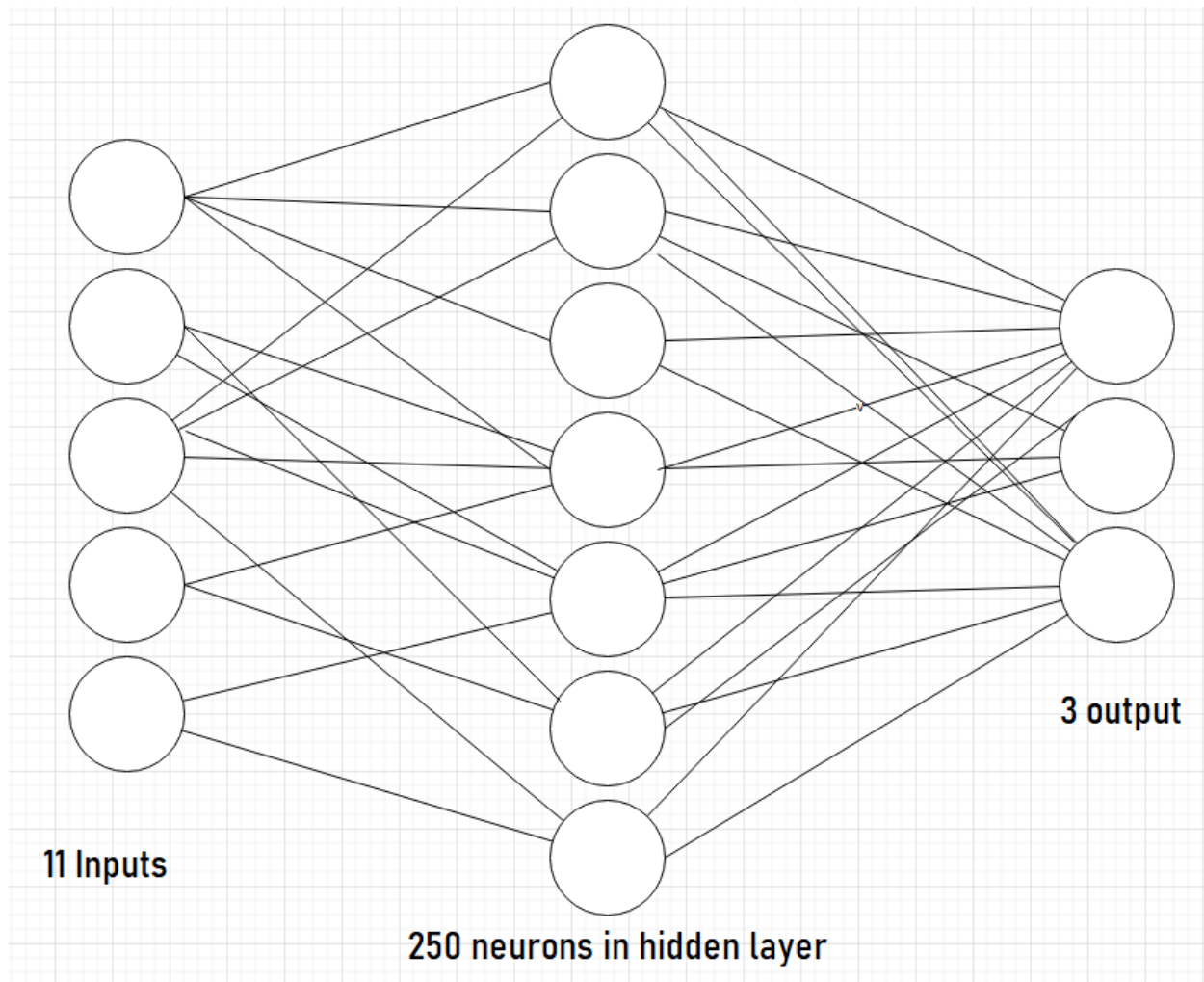


Figure 1.4

As I said 11 inputs for 11 state Boolean values, hidden layer is changeable but I started with 250, and 3 outputs for 3 different action Boolean values.

The general algorithm that my agent will follow is:

```
Set random Q values  
  
loop  
    use model to predict action  
    do that action  
    record results (reward, game over?)  
    update Q value
```

Figure 1.5

### Testing

In testing my agent, I found/changed many different variables in my code. One is that the policy of the agent during the beginning to mid-stages favored looping around in a small circle since due to the snake not getting a negative reward it would stale out in a loop the entire time to avoid dying. To avoid this policy, I thought of either giving a negative reward for looping for a set amount of time-based on how long the snake was and restarting the game, or giving the snake a small negative reward for each time it got farther away from the food and a small positive reward for each time it got closer to the food. I tested both options and saw that even though they both gave a similar late-game policy, the second option favored going straight towards the food which caused a policy that made decisions with no idea where the snake's tail was, which resulted in the snake dying by wrapping itself up a lot. Like I said however both options gave a similar policy which I believe is a fault in my state where it doesn't show the snake where its tail is. I wanted to implement this in my state in order to fix this policy however I couldn't get a working version in time. Also, in testing the number of hidden layer nodes I saw that going over 400 and under 100 gave poor results and so I kept between 200 and 300 nodes. I ended up just

staying with 250 nodes since I didn't see a massive difference between the range of those two numbers. A final main function I tested was the size of the map where the smaller the map the faster the snake trained and closer it got to obtaining its goal but the larger the map the more impossible it got to doing the same. This is due to the exponential growth of possibilities that increasing the size of the map entails.

### **Gym Open AI**

Gym open ai is a tool that's used to use stock environments for people who want to create AIs to beat the stock games. I found out about this tool later in my research and wanted to show a simpler AI, however, I couldn't finish a reinforcement learning one in time. I did however show an algorithm-based AI that follows a non-learning algorithm in order to show off gym.

### **Code Review**

I wanted to show and explain a little bit of code that relates back to some of my research in this paper.

```
# creating the neural network
class QNet(torch.nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super().__init__()
        self.linear1 = torch.nn.Linear(input_size, hidden_size)
        self.linear2 = torch.nn.Linear(hidden_size, output_size)

    #forward function
    def forward(self, x):
        # activation function
        x = torch.nn.functional.relu(self.linear1(x))
        x = self.linear2(x)
        return x
```

Figure 1.7

This shows creating the neural network using pytorch. A class is made that takes the input size hidden size and output size parameters. I talked about how my input size would be 11 for the state, 250 for the hidden layer, and 3 for output/action. You can see the object being created in the agent.py file. A forward function is also being made which is needed for every neural network made with pytorch. It uses the relu activation function that activates the neural network.

```
# training function/general algorithm i talk about on page 11 in research paper
def train():
    plot_scores = []
    plot_mean_scores = []
    total_score = 0
    record = 0
    agent = Agent()
    game = snakegameAI()
    while True:

        # use model to get action and state pair
        state_old = agent.get_state(game)
        final_move = agent.get_action(state_old)

        # use the action
        reward, done, score = game.play_step(final_move)
        state_new = agent.get_state(game)

        # updating q value/ training memory
        agent.train_short_memory(state_old, final_move, reward, state_new, done)

        # remember/record results
        agent.remember(state_old, final_move, reward, state_new, done)
```

Figure 1.8

This code uses the general algorithm I talked about before. The comments explain how they relate to my paper.

## Results and Future Steps

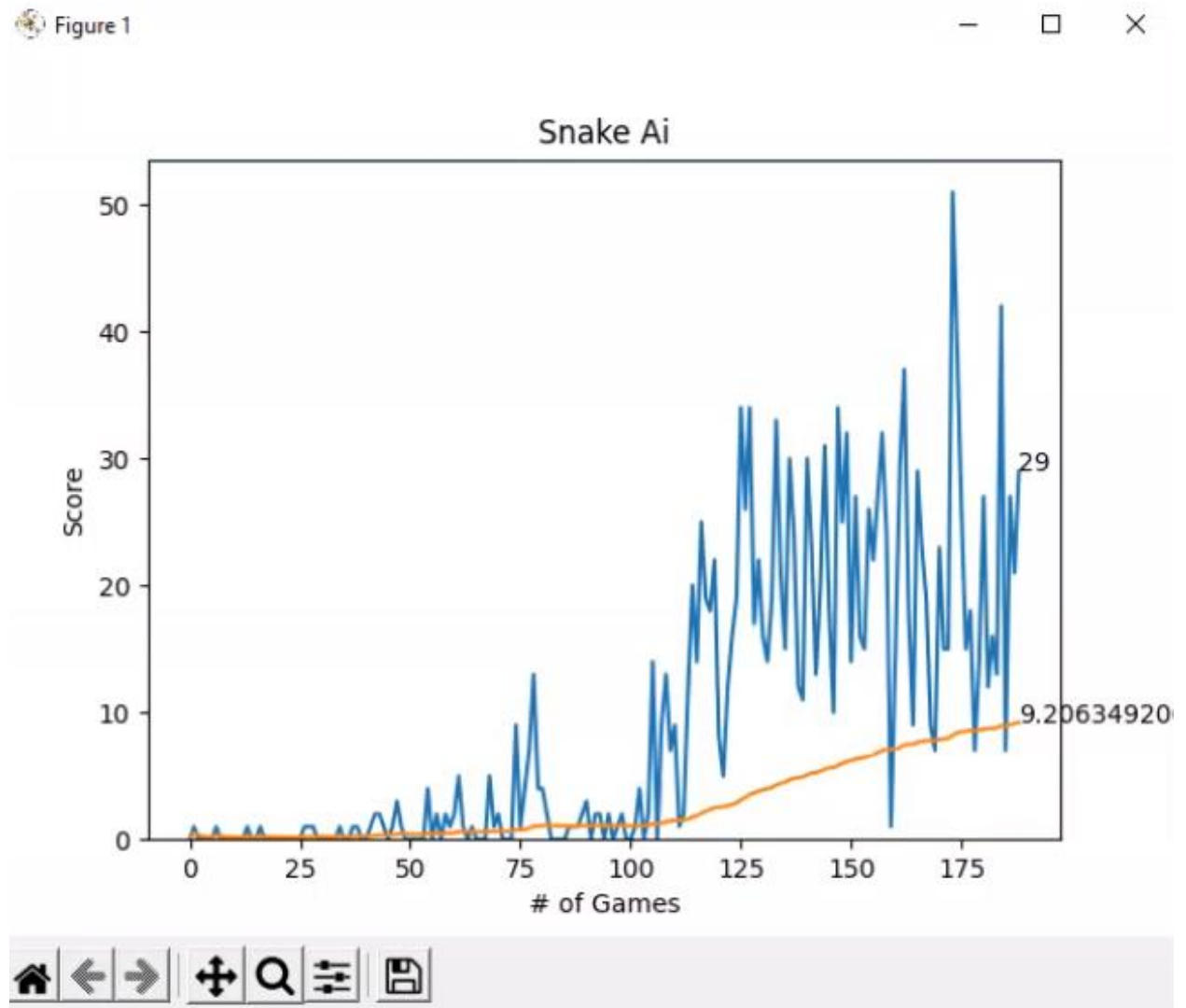


Figure 1.6

In testing I talked about how my snake couldn't see its tail which created a poor policy of wrapping itself consistently. This showed a plateau area that my agent couldn't fix on its own. I realize allowing the agent to train for days could have got me closer to beating the game however I didn't want to leave my desktop running my agent that long. I wanted to fix this plateau by adding more state values to where the snake got information on where its tail was so it would stop wrapping itself. I realized this would be difficult to show in a couple Boolean values due to

the increasing size of its tail. After reviewing how others tackled this problem, I saw people use an area of squares around the head of the snake as the state so that the agent will gain far more information on the state of the game. This worked for small to medium-sized maps however for larger-sized maps it didn't work out too well. In my research, I saw that using deep Q-learning wasn't going to work for large-scale maps with others looking at more advanced algorithms like Proximal Policy Optimization or Monte Carlo Tree Search for fixes in large maps. As for steps after this course, I want to change my state to look at an area of squares rather than Boolean values to get close to beating smaller size maps. I also want to work with gym more since I want to get a working reinforcement learning AI for the cart pole environment I was working on.



## References

freeCodeCamp.org. (2018, March 29). *An introduction to Deep Q-Learning: let's play Doom*.

<https://www.freecodecamp.org/news/an-introduction-to-deep-q-learning-lets-play-doom-54d02d8017d8>

GeeksforGeeks. (2022, February 1). *AI Driven Snake Game using Deep Q Learning*.

<https://www.geeksforgeeks.org/ai-driven-snake-game-using-deep-q-learning/>

GeeksforGeeks. (2022a, January 4). *Snake Game in Python - Using Pygame module*.

<https://www.geeksforgeeks.org/snake-game-in-python-using-pygame-module/>

GeeksforGeeks. (2021, November 9). *Q-Learning in Python*. <https://www.geeksforgeeks.org/q-learning-in-python/>

Loeber P, snake-ai-pytorch, (2021), GitHub repository,

<https://github.com/python-engineer/snake-ai-pytorch>

Patil, Y. (n.d.). Snake Game Using Reinforcement Learning, 1–7.

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction*. The MIT Press.

Zhang, J. (2021, May 5). *Tutorial: An Introduction to Reinforcement Learning Using*. . .

Gocoder.One. <https://www.gocoder.one/blog/rl-tutorial-with-openai-gym>

