HW7 Report

Author: Ewan Lister

# Abstract:

This report explores the application of reinforcement learning (RL) in hacking the classic game "Snake." With a focus on understanding the fundamental principles of RL, we aim to provide insights applicable to RL at all scales. The theoretical background on RL through Markov Decision Processes is presented, followed by a detailed description of model implementation and tuning. By training an agent to navigate the n by n grid, avoiding self-collision and walls while collecting apples, we demonstrate the practical application of RL. The report concludes with a discussion of the model's results, highlighting its effectiveness and implications for future RL endeavors.

# Introduction:

The great Malcom Gladwell once said, “It takes 10,000 hours to truly master anything.” To this, the machine learning community responds, “What if we could speed up time?” The field of Reinforcement Learning (RL), where an agent or “bot” is taught to accomplish complex, human-relevant tasks through many trials of experience, has become more and more relevant to our daily lives. It is the technology behind the most sophisticated of robots, as well as autonomous vehicles. Like many concepts within ML, its foundational theories are actually quite simple, and easy to grasp. Implementing these theories is where the hard part lies.

In this report we will use a reinforcement learning approach to hack the classic game “Snake.” In this game, a snake tries to eat all available apples on an n by n grid, without running into itself or the walls, until it completely occupies the grid space. By covering an elementary example, we hope to highlight the basic principles on will encounter at all scales of RL. First, the report will give some theoretical background on RL through Markov Decision Processes, followed by a description of implementing and tuning the model. To finish, the report will discuss the results of the model, and provide a conclusion.

# Theoretical Background:

Reinforcement Learning (RL) is a branch of machine learning that focuses on training agents to make sequential decisions in an environment to maximize a reward signal. The mathematical foundation of RL lies in Markov Decision Processes (MDPs) and the theory of dynamic programming.

An MDP is defined by a tuple , where:

- represents the set of possible states the agent can be in.

- denotes the set of available actions the agent can take.

- is the state transition probability function, which defines the probability of transitioning from one state to another given an action.

- is the reward function, which assigns a scalar value to each state-action pair representing the immediate reward obtained.

- represents the discount factor, determining the importance of future rewards compared to immediate rewards.

The goal in RL is to learn an optimal policy which is a mapping from states to actions, that maximizes the expected cumulative reward over time. The optimal value function or represents the maximum expected cumulative reward achievable starting from a particular state or state-action pair.

The value function can be computed using the Bellman equations, which express the relationship between the value of a state or state-action pair and the values of its neighboring states. The Bellman equations for state-value function and action-value function are given by:

Reinforcement learning algorithms, such as Q-learning or policy gradient methods, utilize these equations to iteratively update the value function estimates based on observed experiences. By exploring the environment and exploiting learned knowledge, the agent gradually converges to an optimal policy that maximizes the expected cumulative reward.

Additional factors that one must consider in training a reinforcement learning algorithm are and . determines how much consideration the value function gives to future states over the present state, inspecting both principal equations above, one can see that multiplies the sum of the probability of a state transition times the value function or . The value function term appears on both sides of its respective equations, meaning that there is a recursive relationship between the gamma term and the non-gamma term. To effectively train our model, we must look as far into the future as possible, across all possible states, and then balance the perceived value of the sum of these states with the immediate value of a present action.

is sometimes referred to as the exploitation/exploration constant. It deals with balancing the agents capacity to explore with its capacity to exploit its newfound knowledge. Consider life—if one does not try new things, one may never discover their true potential. However, if one does not stick to their guns, they may never reach their true potential. In the same way, an epsilon must be used which gives the agent the opportunity to explore and act randomly, while also gaining value from core demonstrated-value actions. As a result, epsilon is usually decreased over time during an agents training.

The mathematical framework of reinforcement learning provides a solid basis for developing intelligent systems capable of learning from interactions and making effective decisions in complex environments. With the theory of the RL system now covered, the report will now discuss implementing these techniques for Snake.

# Implementation:

The present iteration of the game is adopted from a codebase developed by Vincent Van Wynendaele. In this system there is an established game, Snake.py, which is possible to play on as the user or to train. There is a Snake environment, SnakeEnv.py, which stores functions for calling methods which implement the above mentioned Bellman equations. Then there is the training file, launch.py, which calls the methods in SnakeEnv.py to train an agent over many iterations. As this is the file at the top level of abstraction, we will list it. Here are the important things to consider:

Import statements

from game.Snake import Snake

from reinf.SnakeEnv import SnakeEnv

from reinf.utils import perform\_mc, show\_games

hyperparameters for tuning the training process, note epsilon and gamma, as well as the rewards array. This reward array shapes the value landscape visible to the agent, so values should closely correspond to the outcomes we desire out of the snake.

*# hyperparameters*

grid\_length = 4

n\_episodes = 50000

epsilon = 0.04

gamma = 0.6

rewards = [-1000, -400, 40, 100]

this portion is avaible for the user to play in case they want to demo the game to understand its characteristics

*# Playing part*

game = Snake((800, 800), grid\_length)

game.start\_interactive\_game()

pre-training header for easily reading and documenting results on the command line

print("Epsilon: " + str(epsilon) + " Gamma: " + str(gamma))

print("Rewards: " + str(rewards))

print(**f**"Number of episodes: {n\_episodes}")

calling SnakeEnv to establish a training environment, then running perform\_mc to simulate the sometimes random sometimes estimated actions of the agent. These results are stored in q\_table as state and action groupings

*# Training part*

*# use pickle library to save q\_table*

env = SnakeEnv(grid\_length=grid\_length, with\_rendering=False)

q\_table = perform\_mc(env, n\_episodes, epsilon, gamma, rewards)

visualizing results by displaying the outcome of the trained agent playing the first 100 games

*# Viz part*

env = SnakeEnv(grid\_length=grid\_length, with\_rendering=True)

show\_games(env, 100, q\_table)

# Results:

Initially, one might simply try random values for epsilon, gamma, n\_episodes, and the reward values. One important thing to consider is the practicality of n\_episodes. When the value is in the 0 – 10,000 range. Subsequent demonstrations showed the snake constantly hitting walls, and reaching peak scores of two apples. Due to the weak law of large numbers, only for very large numbers of trials can we expect the intended the agent to learn an ideal policy for navigating its environment. This goes hand in hand with the 10,000 hours proverb. After using n\_episodes within the range of 10,000 to 1 million, we begin to see better results. Yet again there is a trade off, as a 5 million trial run takes approximately 20 minutes to run, but the agent is very matured, and a 10,000 trial run takes only 30 seconds, but the agent does poorly.

By utilizing low n\_episodes to roughly dial parameters, and then confirming via high n\_episode runs, one can effectively narrow down on good parameters. Using only 10,000 trials, a value of 0.04 for epsilon and 0.5 for gamma were established. Shown below is an example of the sort of interpretation one uses to inform hyperparameter tuning, where the value below each loading bar is the average value per trial for the agent.

A screenshot of a computer

Description automatically generated with medium confidence

A reward structure around [, .4 , 40, 100] appears to bring the agent into reasonable performance where it avoids walls, but does not vibrate between one square and the other. Taking the output from these images:

A screenshot of a video game

Description automatically generated with medium confidenceA screenshot of a video game

Description automatically generated with medium confidence

To more ideal behavior where it is almost able to occupy every square:

A picture containing graphics, clipart, cartoon, screenshot

Description automatically generated

Training just a few more n\_episodes put the snake at the winning mark, and only cost an additional 10 minutes. It appears that a very negative value for loss and inefficiency, combined with slight non zero values for efficient moves and winning moves creates ideal behavior for the agent.

# Conclusion:

In this report, we explored the principles and implementation of reinforcement learning (RL) through the example of hacking the classic game "Snake." The foundation of RL lies in Markov Decision Processes (MDPs) and the theory of dynamic programming, which enable agents to make sequential decisions in an environment to maximize rewards. The key factors in training RL algorithms are the discount factor gamma and the exploration-exploitation constant epsilon. We discussed how gamma influences the agent's consideration of future rewards and the recursive relationship it creates in the value function. Additionally, we highlighted the importance of \epsilon in balancing exploration and exploitation during training.

Through the implementation of RL techniques, we trained an agent to play the game "Snake" on an n by n grid. The hyperparameters, such as epsilon and gamma, were tuned to optimize the agent's performance. We conducted multiple iterations of training with varying numbers of episodes n\_episodes and observed the impact on the agent's learning and performance. It was found that larger values of n\_episodes led to better results, demonstrating the significance of extensive training for achieving optimal policies.

The results indicated that with a higher number of trials, the agent improved its ability to navigate the game environment and maximize its score. We compared the performance of the agent trained with different values of n\_episodes, emphasizing the trade-off between training time and agent competence. By using a combination of lower n\_episodes for parameter exploration and higher n\_episodes for final evaluation, we were able to determine suitable hyperparameters for our model.

In conclusion, this report showcases the practical application of reinforcement learning in the context of the "Snake" game. By understanding the underlying theoretical concepts, implementing the training process, and iteratively refining the hyperparameters, we successfully trained an agent that achieved notable performance. The study highlights the importance of thorough exploration and exploitation trade-offs in RL and demonstrates the potential for RL techniques to solve complex tasks in various domains.