Using Reinforcement Learning and Neural Networks to Play Snake

Anonymous Authors[1]

Abstract

This document explores two areas of Machine Learning to aid in the development of an agent that plays the game Snake: Reinforcement Learning and Neural Networks. Using these methods, this document will report operations and performance regarding the development of this agent. To do so, the agent will be developed using Python along with various Python packages to facilitate research.

1. Introduction

1.1. Domain of Research

Snake is a classic video game spanning many casual players worldwide. A web version of this game can be found at:

`http://www.me.umn.edu/~dockt036/snake.html`

It is a very simple game. The player controls a snake that is fixed to a rectangular grid with the objective being to collect randomly spawn food while not dying. The two conditions in which a snake dies are:

- The snake hits a wall.
- The snake hits itself.

1.2. Motivation

The primary reason for exploring the development of an agent to play Snake is because the game is simple. It does not take very long for a human to learn to play this game, rather it is very intuitive. By developing an agent that plays Snake, one is able to compare performance of the agent versus a human. One is also able to review differing methods of Machine Learning to analyze computational efficiency and performance of an agent's execution in a traditional Snake game setting.

1.3. Proposed Methods of Development

First, the game of Snake will be implemented so that all aspects of the game can be controlled. This process will be fulfilled using Python 3.7 and PyOpenGL, a cross platform package that binds OpenGL rendering to Python. In doing so, the user will be able to view the performance of the agent as well as a human player in the traditional Snake game setting. For the sake of computational efficiency, the game will take place in a 10 by 10 grid.

Next, Reinforcement Learning will be explored to facilitate the agent in reaching food using the minimum number of steps possible and can be developed using solely Python. Reinforcement Learning refers to an agent being facilitated by the mechanics of a state and reward system. When the agent encounters a state by performing an action, it will observe a reward. In doing so, the agent updates their decision making to incentivize or disincentivize the action made at that state. In this setting, each location of the grid in the game will serve as a state in this scenario, holding some sort of reward to direct the agent towards the randomly spawned food. The actions used in this process will represent the four directions that the agent is able to take: up, down, left, and right. A Reinforcement Learning algorithm named Q-learning will be implemented to fulfill this task. The algorithm is as follows:

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha[R_t + \gamma \max_a Q(S_t+1, a) - Q(A_t, A_t)]$$

In this equation, alpha is the learning rate, which affects how quickly the rewards are updated. Gamma is the discount factor, which indicates the level of importance regarding future rewards. The max term refers to the maximum future reward the agent can achieve.

Different reward schemes and parameters will be then explored and documented to analyze performance and computational efficiency using Q-learning in the development of an agent to play Snake.

Following this, data regarding the actions taken by the Q-

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

learning agent in this setting will be sampled to facilitate the initial training of a Neural Network. An artificial Neural Network is a framework which outputs a set of mathematically aggregated values given a set of input values. More specifically, every neural network features a certain formation of neurons: an input layer, which feeds into a number of hidden layers, which then perform mathematical aggregations to ultimately feed into the output layer. By giving the network sample data from the Q-learning stage regarding differing features of the game, the network will hopefully learn the mathematical aggregations sufficient for predicting the direction for an agent to take.

The network proposed will consist of four layers: an input layer consisting of twelve input neurons, two hidden layers each containing eighteen neurons, and a final output layer of two neurons. This suggested input set will consist of the snake's distance to food, distance to itself, as well as distance to a wall in the four different directions it is able to travel: up, down, left, and right. The final output layer will decide what direction the snake will ultimately take by summing respective X and Y coordinates ((0,1) is up, (0,-1) is down, (-1,0) is left, and (1,0) is right).

As an out of scope requirement, to further train this Neural Network, a genetic algorithm will be implemented, if possible. In training artificial Neural Networks, a genetic algorithm assesses a generation of agents based on their fitness. The algorithm then selects the top performing agents of this population and uses them to create the next generation, while also introducing random changes to the network, repeating until the fitness of higher performing agents no longer increases.

In the implementation of this genetic algorithm, the population of the first generation will be trained using the data sampled from Q-learning. This generation will be assessed by their overall score, or by the quantity of food they are able to achieve in one game. Given the top performers of this generation, the weights or mathematical aggregations of the following generation will be a recombination of these top performers. Given the new population, ten percent of the networks will be changed randomly, and then assessed again for their fitness. This process will repeat until the top performing agents no longer gain overall fitness.

1.4. Intuition

The proposed methods of development will hopefully create an agent that competitively plays the game of Snake in comparison to a human player. It is assumed that the implementation of the Q-learning algorithm will allow to the agent to fulfill the intended goal of performing similar to a human, but will most likely yield a lengthy or everlasting processing time. Therefore, a Neural Network will hopefully alleviate this issue and facilitate more responsive decision making as well as more consistent performance. Even better performance can be expected in the long run when trained using a genetic algorithm. Since it is impossible to sample data of higher scoring games, and it is imperative to approach this problem using a genetic algorithm.

2. Experiments

In this section, the ultimate goal of these methods are to produce an agent capable of playing Snake with performance comparable to a human.

2.1. Reinforcement Learning Performance using Q-learning

This subsection will report the number of steps to convergence from the agent to the randomly spawned food using different reward schemes and parameters in the implementation of Q-learning. The time elapsed as well as the number of total iterations using the proposed Q-learning algorithm will also be reported.

In this scenario, three combinations of parameters are tested: A high learning rate (.75) and a low discount factor (.1), a low learning rate (.1) and a high discount factor (.75), and finally, the learning rate and discount factor being the same value (.5).

The first reward scheme implemented gives a reward of 100 to the state corresponding to the location of food, while also giving a reward of 5 for moving to a newly found position of the map. If this state is visited again, the reward then is -1. Subsequent visits to these states result in the reward being subtracted by 1. The intuition of this proposed method is that the agent is incentivized to visit every new spot on the grid, eventually reaching the food. The metrics regarding these settings can be found in Figures 1, 2, and 3.

Given Figure 2, it is apparent that the number of iterations increases with distance, but takes a shorter overall time to converge.

The parameters of Alpha = .5 and Gamma = .5 seem to be the most optimal settings, as the route learned by the agent converges the fastest out of the three parameter combinations tested for the initial reward scheme, as seen in Figure 3.

The next reward scheme proposed is to make every point on the grid have a reward of -1 excluding the location of food, which is still a reward of 100. If subsequent points are visited again, the reward is subtracted by 1. This method removes the positive reward given from approaching a new state. The resulting metrics using this scheme can be seen in Figures 4, 5, and 6.

Given the two stated reward schemes and their respective pa-

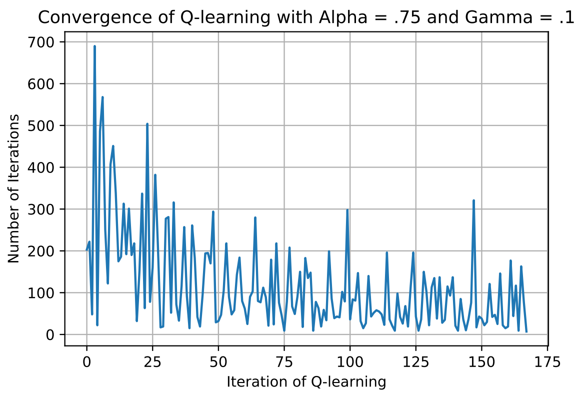


Figure 1. This is the first plot showing the number of steps per iteration or training period using Q-learning to determine the optimal path for the agent 7 blocks away from food, given the initial reward scheme stated. The total time elapsed was 5.034 seconds.

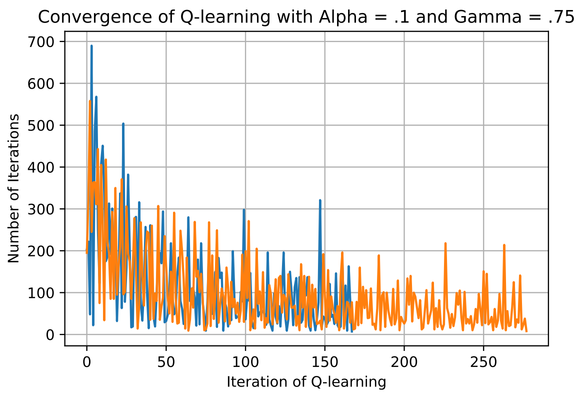


Figure 2. This is the second plot showing the number of steps per iteration or training period using Q-learning to determine the optimal path for the agent 8 blocks away from food in comparison to Figure 1, using the first stated reward scheme. The total time elapsed was 4.294 seconds.

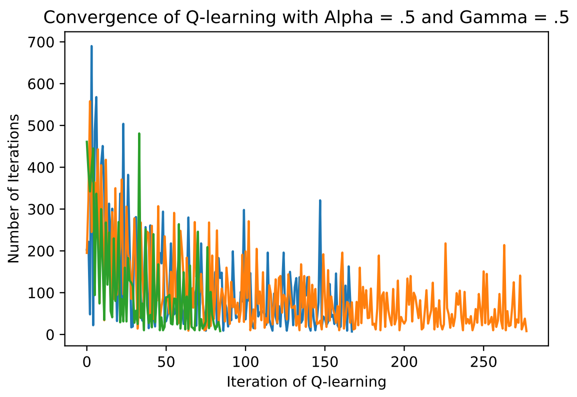


Figure 3. This is the third plot showing the number of steps per iteration or training period using Q-learning to determine the optimal path for the agent 8 blocks away from food in comparison to Figure 1 and Figure 2, again using the same reward scheme. The total time elapsed was 1.505 seconds.

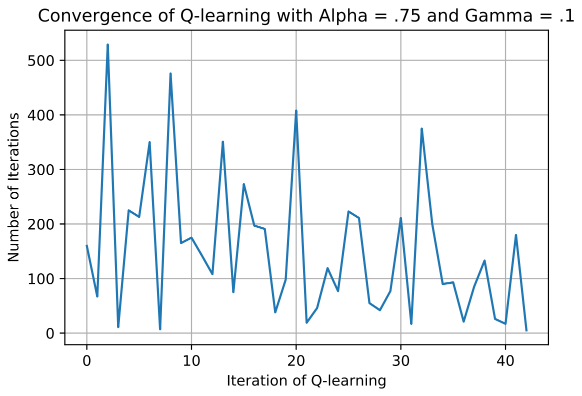


Figure 4. This is the first plot showing the number of steps per iteration or training period using Q-learning to determine the optimal path for the agent 5 blocks away from food, using the second stated reward scheme. The total time elapsed was 1.124 seconds.

rameter combinations, it would seem that the most efficient approach to playing Snake using Q-learning seems to be the second reward scheme paired with Alpha and Gamma values of .5. However, it should be noted that these metrics only measure successful convergence to an optimal path using Q-learning, and do not account for the amount of failures to convergence. With this in mind, it became important to measure the average score attainable by the agent in these differently tested scenarios. Here are the average scores over ten games given their corresponding circumstances stated previously:

Figure 1	1.5
Figure 2	1.3
Figure 3	2.4
Figure 4	1.5
Figure 5	1.3
Figure 6	1.7

It should also be noted that agents performing in the second reward structure were able to reach farther food distances than the first reward structure.

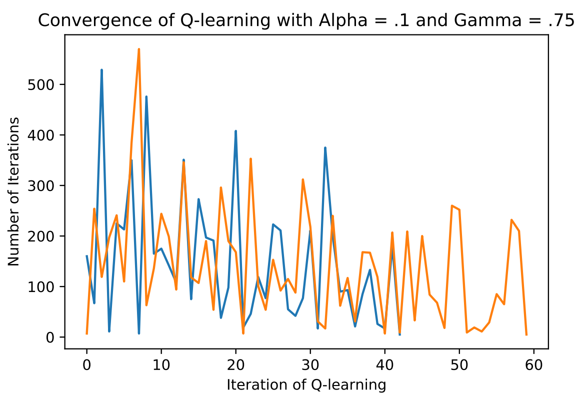


Figure 5. This is the second plot showing the number of steps per iteration or training period using Q-learning to determine the optimal path for the agent 5 blocks away from food, using the second stated reward scheme. The total time elapsed was 1.442 seconds.

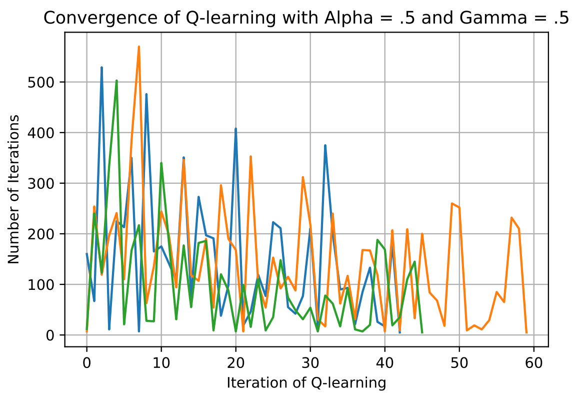
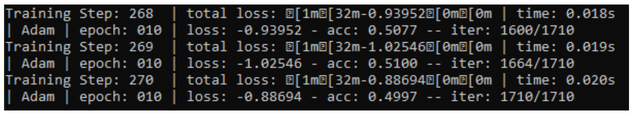


Figure 6. This is the third plot showing the number of steps per iteration or training period using Q-learning to determine the optimal path for the agent 5 blocks away from food, using the second stated reward scheme. The total time elapsed was 0.803 seconds.

2.2. Neural Network Performance using Data Sampled from Q-learning

To facilitate the development of an agent using Neural Networks to play Snake, the Python package TFLearn will be utilized. TFLearn is a higher level API for Tensorflow, a Python machine learning library.

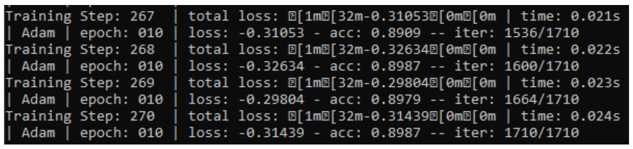
For training the network, around 2500 moves were sampled from the Q-learning algorithm. Initially, the model accuracy was around fifthy percent according to the metrics provided by TFLearn, which can be viewed in Figure 7.



```
Training Step: 268  | total loss: [1m[32m-0.93952[0m[0m | time: 0.018s
| Adam | epoch: 010 | loss: -0.93952 - acc: 0.5077 -- iter: 1600/1710
Training Step: 269  | total loss: [1m[32m-1.02546[0m[0m | time: 0.019s
| Adam | epoch: 010 | loss: -1.02546 - acc: 0.5100 -- iter: 1664/1710
Training Step: 270  | total loss: [1m[32m-0.88694[0m[0m | time: 0.020s
| Adam | epoch: 010 | loss: -0.88694 - acc: 0.4997 -- iter: 1710/1710
```

Figure 7. These are metrics provided by TFLearn during the model fitting process. The network being trained follows the same proposed structure of 12 input neurons, 2 layers of 18 hidden neurons, and 2 output neurons. All layers used a combination of rectified linear unit (ReLU) and sigmoid activation functions.

These result of this training seemed very poor, and because of this, the structure of the network was changed. Initially, the number of epochs to train the network was changed from 10 to 100, until it was noted that the accuracy fell even lower to around forty percent. After changing the two hidden layers from 18 neurons to 24 neurons, and changing the activation functions to sigmoid and softmax for the input and hidden layers respectively, an accuracy of almost ninety percent was reached, as seen in figure 8. The structure of the hidden layers was again readjusted from sets of 24 neurons to 48, but did not produce any changed effect.



```
Training Step: 267  | total loss: [1m[32m-0.31053[0m[0m | time: 0.021s
| Adam | epoch: 010 | loss: -0.31053 - acc: 0.8909 -- iter: 1536/1710
Training Step: 268  | total loss: [1m[32m-0.32634[0m[0m | time: 0.022s
| Adam | epoch: 010 | loss: -0.32634 - acc: 0.8987 -- iter: 1600/1710
Training Step: 269  | total loss: [1m[32m-0.29804[0m[0m | time: 0.023s
| Adam | epoch: 010 | loss: -0.29804 - acc: 0.8979 -- iter: 1664/1710
Training Step: 270  | total loss: [1m[32m-0.31439[0m[0m | time: 0.024s
| Adam | epoch: 010 | loss: -0.31439 - acc: 0.8987 -- iter: 1710/1710
```

Figure 8. These are metrics provided by TFLearn during the model fitting process. The network being trained follows the modified structure of 12 input neurons, 2 layers of 24 hidden neurons, and 2 output neurons. All layers used a combination of sigmoid and softmax activation functions.

The results picured in Figure 8 were puzzling considering the fact that the output of the neural network was only between 0 and 1, and therefore unable to predict moves such as down and left. To alleviate this, the activation of the output layer was changed from softmax to tanh. However, when utilizing this model to predict the movement of the snake in game, the agent was unable to move, consistently outputting (0,0), a signal for no movement. To combat this, labels or moves collected from the data were transformed into arrays of four elements to signal their respective direction ([1,0,0,0] is up, [0,1,0,0] is down, [0,0,1,0] is left, and [0,0,0,1] is right. Then the final output layer of the network was changed from 2 neurons to 4, having a sigmoid activation function. This modification unfortunately had no effect on the network's predictions.

3. Conclusions

The game of Snake is a classic, easy-to-learn casual game known to many who have played it worldwide. Because the game is fairly intuitive to play, it was determined that it should also be intuitive to develop an agent to learn how to play.

The implementation of Q-learning is fairly straightforward and simple. However, the use of the algorithm in Snake was unfortunately susceptible to endless calculations. As a result of this and the results posed in the Experiments section, Q-learning was not a viable method to approach the development of an agent to play snake with comparable performance to humans.

Similarly, in approaching Q-learning, the development of a Neural Network using TFLearn was again straightforward and simple. By sampling around 2500 correct and optimal moves that were able to be converged using Q-learning, a Neural Network was fitted. Unfortunately, numerous modifications to the model were made in attempt to direct the agent in a favorable direction, but did not render suitable results. If possible, the collection of much more data than what was sampled would likely have been beneficial to the training process of this network. Likewise, the implementation of a genetic algorithm could potentially train the network to become an excellent player.

Still, it is believed that the implementation of an agent to play Snake with performance comparable to humans can be simple. However, this document proves a certain level of finesse is needed in order to accomplish that.

A GitHub repository of this project can be found at:

`https://github.com/tylerlit/Snake-RL_NN`