SmartSnake

CSCI 761 Final Project

Melissa Lynch, Shadman Quazi, Zachary Wilson

May 19, 2020

1 Introduction

Games are a very important field in AI, as they can be used to create and test new algorithms. Games can be formed as a search problem, where the agent (or agents) must develop a strategy to maximize their score, given a set of rules and a start state. A single agent system can be responsible for solely maximizing their own scores and a multi-agent system can be designed to work competitively or cooperatively.

Snake is a game with a simple set of rules, a player object (a snake) in an NxN grid can move one space at a time. In a randomly determined part of the map there is a fruit, the snake must move to and eat the fruit to increase its score, after eat the fruit the snake gets longer by one square. The snake's objective is to collect as many fruit as possible while avoid collisions with its own body and with the walls, otherwise the game is over. Small revisions to the rules can be made such as time limits per round or different fruits spawning with different point values but the basics rules stay the same.

For this project we trained an agent to play snake, with the end goal that it can play the game as well, if not better than a human could play the game. There were many means by which to tackle this problem, for example we could have implemented a heuristic-based search algorithm such as A* search. However,

this would end up being a very inefficient task, given the size of a typical snake board this would lead to a very large state space. As a result of this large state space it would make traversing the search tree a very expensive task. Finding the next best move would be very difficult even with a well tuned heuristic function, as such we decided to implement a machine learning based technique called Deep Q-learning. We also explored some alternate scenarios of snake, explained later in this paper.

2 Background

Before we begin discuss DQN's, let's first discuss the general concept of reinforcement learning. Reinforcement learning is a learning technique that focuses on developing a sequence of decisions that aim to maximize a reward, for example, a robot trying to stay upright while walking. Reinforcement learning bases itself in the Markov decision process (MDP), a decision making process that bases its future action and state, s_{t+1} and a_{t+1} on the present state and action, s_t and a_t. It will base it's action on a set of transition probabilities and potential rewards. The MDP will therefore try to maximize its cumulative reward over a sequence of steps. From here the MDP needs to employ a value function, to assess how much total reward a policy will generate. A common value function to use is the Bellman equation.

In Q-learning, the function we seek to maximize is called the q-function. To assist in evaluating the q-value, we utilize what's called a Q-table. A Q-table is a set of actions and states, and it's associated q-value.

Initialized

Q-Table		Actions					
		South (0)	North (1)	East (2)	West (3)	Pickup (4)	Dropoff (5)
States	0	0	0	0	0	0	0
	. . .	. . .	. . .	. . .	. . .	. . .	. . .
	327	0	0	0	0	0	0
	. . .	. . .	. . .	. . .	. . .	. . .	. . .
	499	0	0	0	0	0	0

Training

Q-Table		Actions					
		South (0)	North (1)	East (2)	West (3)	Pickup (4)	Dropoff (5)
States	0	0	0	0	0	0	0
	. . .	. . .	. . .	. . .	. . .	. . .	. . .
	328	-2.30108105	-1.97092096	-2.30357004	-2.20591839	-10.3607344	-8.5583017
	. . .	. . .	. . .	. . .	. . .	. . .	. . .
	499	9.96984239	4.02706992	12.96022777	29	3.32877873	3.38230603

Figure: Example q-table before and after learning, https://www.learndatasci.com/tutorials/reinforcement-q-learning-scratch-python-openai-gym/

The q-values of each state-action pair are initialized to 0, and are updated continually as the agent(the snake) explores the game environment, converging on an optimal policy. The new Q-values are updated according to the Bellman equation, Q (s, a) = $\max_{pi}$ E [R_t—s_t = s, a_t = a, pi],, where pi is a policy, R is the reward, s and a represent a state and action pair, s_t and a_t represent a state and action pair at a time-step Deep Q-learning is similar in that it finds an Q-value for each state/action pair but instead of updating a static matrix it updates a neural network.

3 Q-Network Architecture

For the Q-network, we used a very simple multi-layered perceptron. It consists of an input layer of size 11, a hidden layer of size 256, and an output layer of size 3.

The input to the network is the state, which is the current situation in which the snake finds itself. The state is represented by an array containing 11 boolean variables. These variables take into account: any immediate danger for the snake, a wall or its own body (straight ahead, to the left, or to the right); the direction the the snake is currently moving (up, down, left or right); the location of the spawned fruit in relation to the snake (above, below, left or right). There is no standard for choosing the width of the hidden layer. We chose 256 because it seemed to be the most popular when looking at other examples for similar problems, and it worked well in practice. There is an output node for each possible action, that is, continue straight, turn left or turn right. These outputs are the predicted Q-values for each possible action, given the input state; thus, we are able to compute the Q-values for all possible actions using only a single forward pass through the network.

When the network is created, a linear transformation function, **linear1**, is defined using the input layer of as the size of the input and the hidden layer as the size of the output. It also defines a second linear transformation function, **linear2**; this one is using the hidden layer for the input size and the output layer for the output size of the function. When the model is called, the forward function is called. Given a state as input, called x, **forward()** will first apply linear1 to this input. The output from this is the size of the hidden layer. To introduce non-linearity into our network, this result is then passed into a Rectified Linear Unit (ReLU) function. Very simply, if the input is negative, the output is 0; otherwise, if the input is positive, the output is just the input

unchanged. ReLU accelerates the convergence of stochastic gradient descent when compared to sigmoid or tanh functions. Finally, this result is passed into the linear2 function, which takes as input the hidden layer of size 256, and has an output of size 3, corresponding to all possible actions. Thus, given a state as input, all possible actions and their Q-values are returned. The loss function used is Mean Square Error; this is the sum of squared distances between the target and the prediction.

4 Deep Reinforcement Learning

The algorithm we used to train our agent follows the approach discussed in this paper: https://www.cs.toronto.edu/ vmnih/docs/dqn.pdf (Playing Atari with Deep Reinforcement Learning).

As previously mentioned, the optimal action-value function is defined as the maximum expected reward achievable. The goal of reinforcement learning algorithms is to estimate the action-value function, by using the Bellman equation to update the policy after each iteration. These algorithms will converge to the optimal action-value function as the number of iterations increases. Typically, a linear function approximator is used to to estimate the action-value function. However, a non-linear function approximator, such as a neural network, can be used instead. As defined in the *Atari* paper, we refer to a neural network function approximator with weights θ as a Q-network. A Q-network can be trained by minimizing a sequence of loss functions $L_i(\theta_i)$, that changes at each iteration i. The target for the iteration is determined using the Bellman Equation: the reward plus gamma times the maximum Q-value. In other words, the maximum future reward given a state and an action is the immediate reward plus the maximum future reward for the next state. Thus, computing the targets is dependent on the network weights, as that is what is calculating

the Q-values. This is different from supervised learning, in which targets are fixed before training begins. The prediction is determined simply by calling the model on the current state and choosing the max Q-value returned (the best move to make given that state). The target and prediction for the iteration are then fed to the loss function, and a gradient is computed. The parameters of the network are then updated based on the computed gradient. The optimizer uses popular a method for stochastic optimization called Adam.

This algorithm solves the reinforcement learning task using only partial information from the emulator E. The task is partially observed, as it is completed without explicitly constructing an estimate of E. That is, the agent will learn just by the inputs given in the state; it doesn't have knowledge of the whole environment.

The algorithm is off-policy to ensure that the state space is adequately explored; it will make random movements with a certain probability. As the iterations increase, movements are chosen less randomly and more based on the policy. In our approach, we set epsilon to be equal to 80 minus the number of games played. A random number between 0 and 200 is chosen. If that value is less than epsilon, then the action is chosen randomly. Otherwise, the agent uses the policy to choose an action. This means that at game 1, epsilon has a value of 0.4. At game 60 that value is 0.1. After 80 games, epsilon becomes 0 and the agent no longer chooses actions randomly. This is considered a decaying ϵ-greedy approach.

After trial and error, 80 seemed to be the best value to stop off-policy actions. By then, the agent has explored enough, and gathered sufficient new information to lead it to better decisions in the future. The agent then switches to exploitation, where it makes the best decision given current information. The agent regularly reaches a top score of 50 within 200 games. The agent performed

considerably worse with a fixed epsilon of 0.2, only achieving a top score of 9 in 200 games.

The concept of regret is how much worse the agent performed than the best it could have possibly done. We want to maximize total reward and minimize total regret. The lower bound on regret is logarithmic in the number of time steps. This is also true of a decaying ϵ-greedy algorithm.

The algorithm updates the parameters of the Q-Network (which estimates the value function) directly from on-policy samples of experience (state, action, reward, next state, next action) drawn from the agent's interactions with the environment.

We use a technique called replay memory, also called experience replay, where we store the agent's experiences at each time-step in a data-set. We use replay memory to apply Q-learning updates, or minibatch updates, to samples of experience which are drawn at random from the pool of stored samples.

The psuedocode for the algorithm is as follows:

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

The training loop begins by initializing the agent, which is the snake, and the environment, which is the game board. Inside the main loop, there is an option to limit the number of games played in a single training session. The

state is returned by calling the getState function on the current game; this will return the state given the games current configuration. As mentioned previously, this creates an 11-variable array, checking for immediate danger for the snake, which direction the snake is moving, and where the fruit is in relation to the snake. Given that state, we use the getAction function to select an action. This determines the action the agent should take given that state. Based on epsilon, this will either be a randomly chosen action (off-policy) or an action chosen by the policy (on-policy). Now that we have an action for that state, we can emulate that in the game environment using frameStep. This will perform the action in the game environment, resulting in a reward, the game's score, and whether or not that move caused the game to end. The next state is computed after the game is updated using frameStep. Now we have everything needed to update the policy: state, action from state, reward, and the next state. We also want to store this transition into memory, so we can use it when we employ experience replay.

We update the policy given the inputs as follows. The target is the reward if the game is done. To compute the target if the game is not done, we use the Bellman Equation with the model's prediction of the next state's action. We then need to get the model's prediction of the old state's action. This is then optimized by first calling the loss function on the target and the prediction. We then compute the gradient using loss.backward(). Finally we call the step method on the optimizer, which will actually apply the gradient descent and update the parameters of the network.

If done is true, that is, the action from the state ended the game, we reset the game environment and record the total score for that game. Now that the game is over, we want to train using replay memory. Training using replay memory is similar to updating the policy on a new state. But instead of being trained

on new information from the environment, the memory which has the history of transitions will be randomly sampled, and the policy updated using those old transitions as input. Target and prediction are calculated the same as in the other training method, and it is optimized the same way. The only difference is the input data is coming from memory, previously performed transitions, rather than new transitions.

Learning directly from consecutive samples is inefficient, because of the strong correlations between the samples; randomizing the samples breaks these correlations. This greatly stabilizes and reduces the variance of the model.

When using the Q-network to make decisions, the current weights of the network are what determines the next data sample to be trained on; it is easy to see how unwanted feedback loops can arise. The parameters of the network can become stuck in a local minimum. For example, if the best action is to move left, the training samples will be dominated by samples from the left-hand side. When using replay memory, the behavior distribution is averaged over many of its previous states. This will smooth out learning and avoiding oscillations (high variance) in the model.

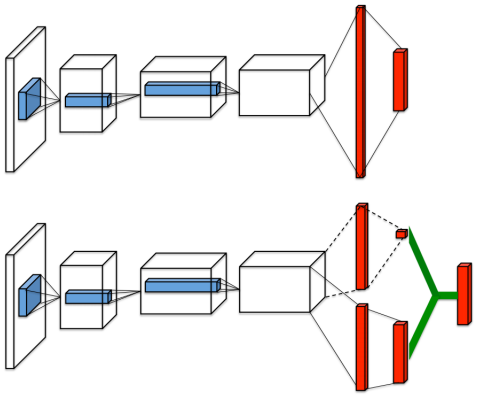
There are some limitations to this approach. In practice, our algorithm only stores the last 100,000 experiences in the memory. The memory does not differentiate important transitions from non-important ones. When the capacity is reached, it will always overwrite older transitions with recent transitions. Sampling uniformly at random from the memory gives equal importance to all transitions, when again, this may not be the case. It is possible to emphasize the transitions from which the agent learns the most.

One method that we researched, but did not have time to implement, is the concept of a dueling architecture for the network. This approach is presented in the following paper: https://arxiv.org/pdf/1511.06581.pdf

The goal is to provide a neural network architecture that is better suited for model-free reinforcement learning. Our algorithm is model-free, as the agent only partially observes the environment. The proposed network can be easily combined with existing reinforcement learning algorithms.

The dueling architecture explicitly separates the representation of state values and (state-dependent) action advantages. Intuitively, the dueling architecture can learn which states are (or are not) valuable, without having to learn the effect of each action for each state. This is particularly useful in states where actions do not affect the environment in any relevant way (no change in reward).

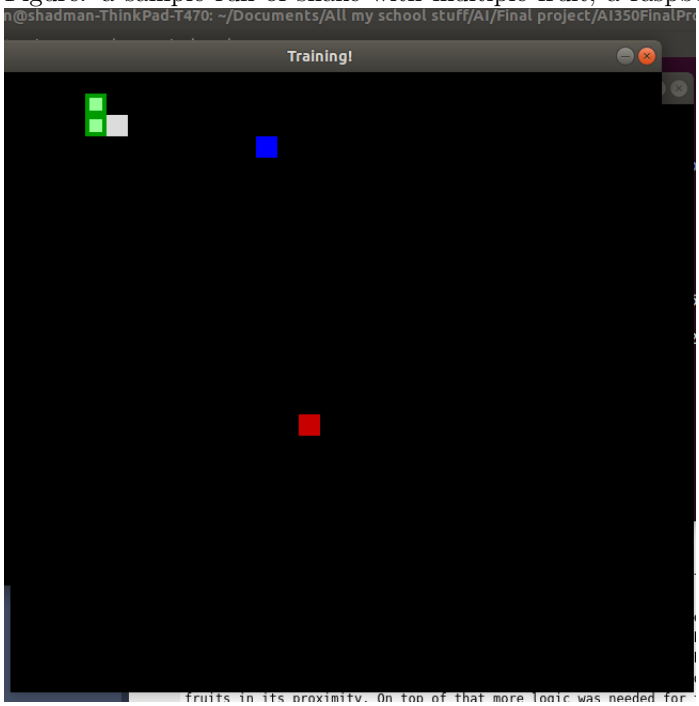
In the context of snake, the value stream learns to play the game in general; the advantage stream learns to pay attention only when there is danger immediately in front, so as to avoid collisions. The action choice of the advantage stream is nearly irrelevant when there is no danger; it makes a relevant action choice only when there is an immediate danger. Below is a traditional network with just one stream, and the proposed dueling network with two streams.



5 Improvements on Baseline Snake

5.1 Multiple fruit

Figure: a sample run of snake with multiple fruit, a raspberry and blueberry.

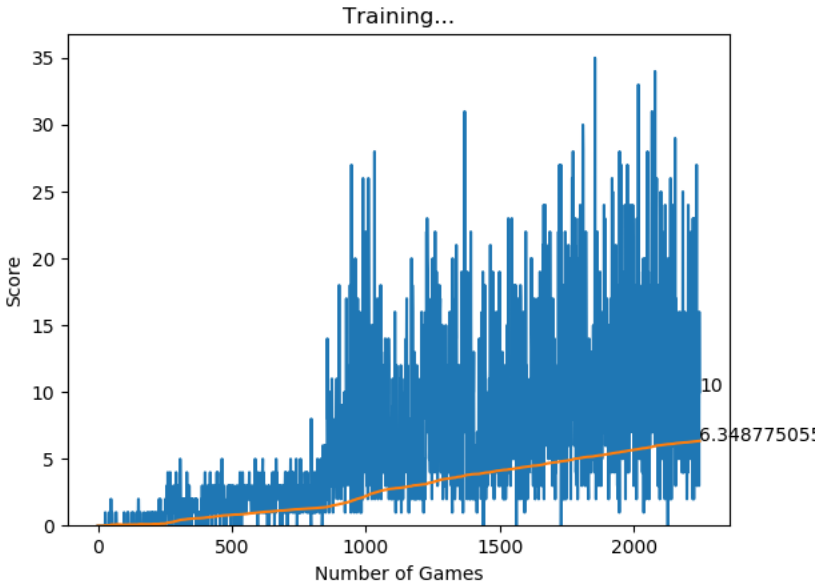


Using our single snake, single fruit environment as a baseline, we had opted to look into other environments to see how our network would perform and see what, if any trends would arise when we trained with these new scenarios. The first alternate scenario we tried was multiple fruit and a single snake.

For this, some changes need to be made to our Snake AI, because there were multiple fruit present in the environment the agent need a way to discern which fruit would be the best to eat. For this we calculated the Euclidean of each fruit and had the snake move to whichever would be closest to the snake. This function would be updated at the start of each game loop to make the snake aware of other fruits in its proximity. On top of that more logic was needed for

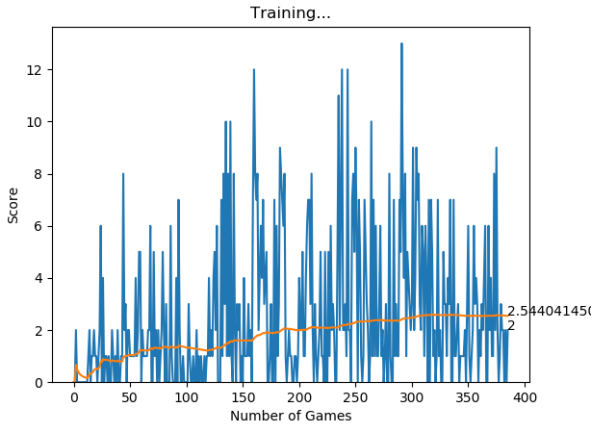
tasks like generating multiple fruit locations and handling collision logic between them.

Figure: Two raspberry set-up after 2500 iterations.



This idea went through different iterations, first was with two raspberries, with equal reward values. This scenario ended up faring worse than our standard environment. After 2500 iterations, our network was able to achieve a max score of 35 and and average score of 6. The second multi-fruit scenario we tried was a raspberry and a blueberry, each with different reward values.

Figure X:Graph showing the progress of the multi-fruit snake, over 400 runs

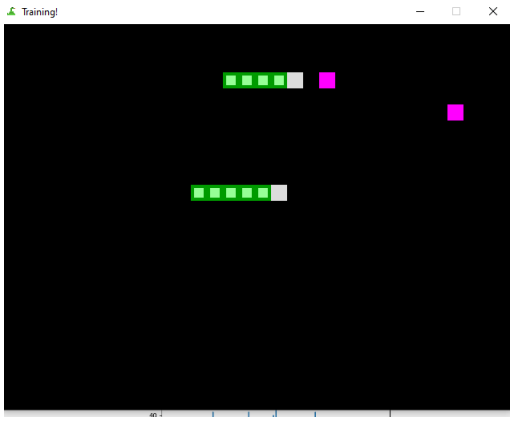


This again required an update to the logic to handle multiple reward values in the network. With this version, our network was able to obtain a record score of 13 after 400 iterations, with an average score of 2.5 during one of our runs and stayed at the score for the another 100 iterations. The agent did not seem to show any preference for one fruit type over the other, despite the blueberry having a higher reward value.

The biggest issue with multi-fruit behavior had to do with the distance function we used. A common behavior we noticed during training was that the snake would get stuck in a loop when two fruit were equidistant as it was traversing the game board. It would be unable to break out of the loop and be forced to timeout and restart. A potential fix to this distance function would have between to more like a weighted sum, taking into account the different reward values of each fruit. This in theory would have made the choice in which fruit to take more consistent and it would have made developing a policy easier going forward. Another factor to consider was score value, while we gave blueberries a higher reward than raspberries, they had the same score, 1 point. If there was a way to track the score's of each fruit in conjunction with the two previous factors, distance and reward value a better policy could have

been developed. This could have again been a weighted sum or some kind of product value. We could have also tried changes to the network architecture to account for multiple fruit in it's input layer, this likely would have required deeper changes to the activation functions and loss calculations as well.

5.2 Multiple snakes

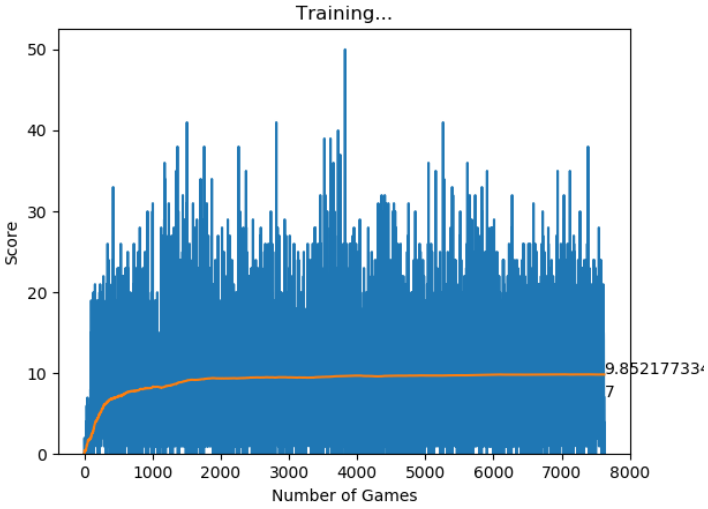


After we added multiple fruit the next obvious step was to add in multiple snakes. To add multiple snakes we needed to make several modifications to the code. The biggest was adding a Snake class instead of having the snake be a inherent part of the game world. The other big change was changing the state of the snake from viewing the walls adjacent to it and whether it is adjacent to itself, to viewing the walls, itself, and other snakes. It was somewhat challenging to alter the program such that it ended not when a snake hits a wall or some other obstacle but when all snakes have hit a wall or an obstacle. In a similar manner we had to properly handle the correct removal of snakes from the game board when one hit an obstacle.

One of the issues we noticed was that the snakes still commonly ended by running into each other. Usually most of the snakes will have "died" very shortly into the game leaving only one snake free to gather and reach a large

size. Part of this may be due to the fact that all the snakes are controlled by the same AI albeit one that is trying to maximize each snakes score individually. This results in that one remaining snake unduly influencing the training of the AI and resulting in a snake that acts little different from a snake trained in a solo environment. We theorize that increasing the penalty for death will help overcome this issue. This issue in turn means that the snakes do not react well to the presence of other snakes and as such the initial stages of the game can be quite variable. Leading to a large number of cases where the snakes do not leave a single survivor and instead wipe each other out leaving a very low high score, but having a very good high score when there is a single survivor. This greatly increases the variability of the outcome and brings down the average score.

Figure 3: The progress the the AI with multiple snakes and multiple fruits



We did a test run in which we trained the AI on two snakes with two fruits with the same value. We ran this for approximately 7500 games and in the end the average score of the highest scoring snake was a little under 10 for each run.

6 Results

We found that The snake AI showed significant improvement over it's training. With the largest improvement being after game 80 when the chance of a move being random fell to zero. We found that the improvement began to slow over time with our highest scores reaching approximately 50 after 200 or so games. After 500 games, the top score reached was 65. When we added multiple fruit we found the the rate of improvement fell considerably with the AI taking over 2000 games to reach an average score of 6 when the single snake AI reached an average of 13 within 200 games. When we added multiple snakes onto the game board we were surprised that the rate at which the AI improved increased considerably. With it reaching an average of 10 at about 500 games and mostly plateauing from there. We theorize that this is because with more snakes on the board the AI has a much broader set of moves to learn from.

7 Code

Our code can be found at https://github.com/ShadmanQ/AI350FinalProject