Using The Q Learning Algorithm To Beat a Game

Mitchell Kolb
11670080
School of Electrical Engineering and Computer Science
Washington State University
Pullman, USA
mitchell.kolb@wsu.edu

Flavio Alvarez Penate
11689700
School of Electrical Engineering and Computer Science
Washington State University
Pullman, USA
f.alvarezpenate@wsu.edu

***Abstract*—Artificial Intelligence can be trained to find the optimal way to complete a task in a simulation of sorts. These simulations can include finding the fastest way to travel from one location to another, having a self-driving car, etc. When an AI is first introduced, it has no previous knowledge of the situation that it is in or the requirements expected of it. Hence, there's methods to train such AI through Q learning. Q learning is based on a reward and punishment system for good and bad behavior, respectively. This reward and punishment system is carefully selected among certain criteria that we want the AI to fit. After each iteration of the Q learning algorithm, the AI starts to be rewarded for good behavior and it looks for ways to reproduce the behavior that achieved this positive stimulant. After numerous iterations and time spent training the model, the Ai will have optimized behavior to achieve the tasks assigned to it. In order to prove the ability of Q learning in training AI, we will produce a video game that will train an AI to perform a certain task, such as avoiding certain objects, traveling towards a certain point, or going from one location to another in the fastest way possible through an obstacle course. This project's goal is not only to showcase the capabilities of Q learning, but to emphasize on its broad uses and the ability to be implemented into more than one situation. responses.**

Keywords*—*Q Learning, Artificial Intelligence, reward system, behavior, training.

I. Introduction

Training an Artificial Intelligence program to perform a task in the most efficient way possible can have many advantages. It could show the fastest route to get from point A to point B, teach a car how to drive itself, or show a non-player character in a video game that is controlled by the computer how to move so that it doesn't collide with walls, follows a certain path, or has the most effective combination of attacks, etc. The way that Q Learning starts out is with an artificial intelligence that knows nothing, rewarding good or correct behavior and punishing bad or failed behavior. For instance, if a non-player character is being trained using Q Learning to walk across a bridge, every movement that the artificial intelligence commits to that makes progress towards the other size is rewarded; meanwhile, any movement that results in getting off track, turning back, or falling off the bridge, is punished by the algorithm.

One of the difficulties that come with Q learning is deciding how to implement the reward system. While the punishment system can be easily defined as some action that we're trying to avoid, the reward system has to be carefully selected in order for the artificial intelligence to be able to make some progress through random actions when it begins. Hence, if one implements the only reward system for the bridge example as making it across the bridge, it is very likely that the artificial intelligence will never cross the bridge, since it is a much more distant goal than taking a step forward and rewarding each correct step individually. Thus, reward systems can be implemented for various aspects or types of behavior, each having a different reward value attached to them.

The objective of Q Learning is to find the policy that is optimal in the sense that the expected return over all successive time steps is the maximum achievable. Find optimal policy by learning the maximal values for each q pair. The Bellman optimality equation is used to calculate q*. The algorithm iteratively updates the q values for each state action pair using the bellman equation until the q function converges to q*. Once we have a optimal q function, q*, we can determine the optimal policy by applying a reinforcement learning algorithm to find the action that maximizes q* for each state.

II. Literature Review

This project was inspired by a tutorial on an artificial intelligence lizard jumping tiles in a game that consists of tiles containing bugs, birds, or nothing in order to eat (or land on) the highest number of bugs possible. In this game, the artificial intelligence is using a reinforcement learning technique where it is rewarded one point for ending on a tile that has a single bug and ten points for ending on the maximum prize. Likewise, if the artificial intelligence ends on a tile that's empty, it loses one point, and if it ends on a tile that contains a bird, it loses ten points. Additionally, if the artificial intelligence stumbles upon the maximum reward or the maximum punishment, the game ends and that trial is over. The final goal of this game is to have the artificial intelligence lizard be able to reach the tile with the five bugs in the shortest number of moves possible, thus optimizing the q*. It uses a q table to store the q action pairs for every move that a particular lizard iteration takes. Over time it will use the data from this table to determine its moves and use it to choose smarter moves to maximize the most amount of points.

We were inspired to stick with this Q Learning model because we see lots of applications for this model outside of the lizard game. This model can be used for any game where you can let it use a set control scheme and reward system and let the model learn over many generations. We have looked into the math behind Q Learning and we found an article by Chathurangi Shyalika and she explains reinforcement learning and the six steps that make that work and the mathematical functions that allow Q Learning to take in data and store it to then determine in the future if the reward given by that decision is worth it.

III. Technical Plan

The techniques that we plan to use in the implementation of the Q Learning model is the Q-Table. This is a data structure that is used to calculate the maximum expected future rewards for every action at each state. The table mathematically shows a layout of the best decision to reach the goal. The Q-Table uses the Q Learning algorithm which uses part of the Bellman equation.

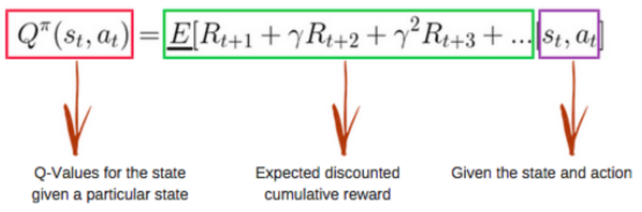
$Q^\pi(s_t, a_t) = \underline{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots | s_t, a_t]$
Q-Values for the state given a particular state
Expected discounted cumulative reward
Given the state and action

Figure 1: Q-Table Method [2].

This equation when used in conjunction with the Q-table can determine whether a certain action is "rewarding" or not by the resulting value. Using this equation over many different generations will allow the network to highlight the best decisions to maximize Q to get the highest rewards. Another part of our plan is to apply this to a game we will have to build from scratch.

When choosing an environment to train the artificial intelligence through Q Learning, we needed to choose a game in which a clear reward system can be implemented, and a failed trial can be detected without any chance of ambiguity. Therefore, an adequate fit for implementing the Q-Table method would be a 2-dimensional racing game. In this game, the artificial intelligence has to drive a car for an entire lap across the track past the finish line in order to obtain a reward. Hit detection is implemented on the border of the track, so that if the artificial intelligence driver crashes, the algorithm recognizes that run as a failure.

IV. Intermediate Results

In order to proceed with our plan of using a Q-Table, we had to implement an environment in which to train our artificial intelligence model using the Q-Table method from Q Learning. We have used Python and its pygame library in order to create a 2-dimensional racing game, in which artificial intelligence will learn how to drive and avoid the border of the track [7].

Aside from the visuals, there's only three entities that need to be kept track of in the game: the car, the border of the track, and the finish line. The artificial intelligence will be responsible for controlling the movement of the car, which can only go forward and rotate both left and right with a 360 degrees range of motion in order to turn. The hitbox of the track border will serve as a method of identifying failure, and the finish line will be the reward for the artificial intelligence driver finishing a lap of the track.

One of the main obstacles we encountered during this step of the project was how to determine when the car hits the side of the track. This element is an essential part of the implementation since it is how the artificial intelligence knows that it failed during a trial. In order to implement this feature, we photopped the track image into two separate images, one that contains the entire track, and another that contains just the red and white border for collision tracking. Separating the track into these two images ensures that no collision is detected when the car is just traveling in the gray area that colors the path of the track as seen in Figure 2.

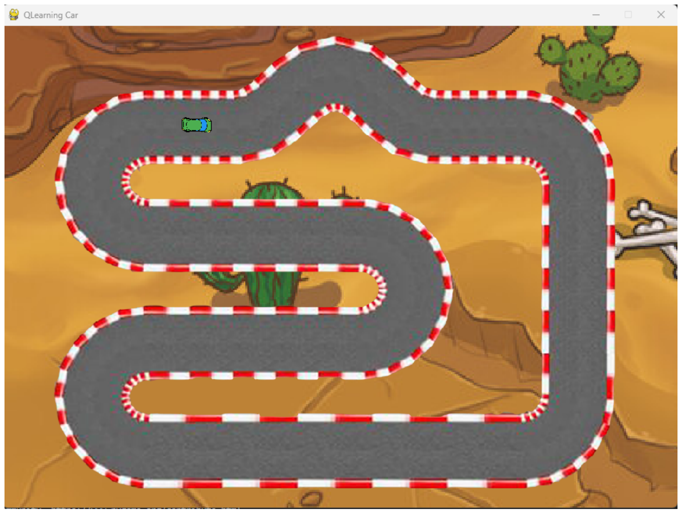

Figure 2: Racing game used as environment.

In regard to the Q-Learning implementation into python to be able to use within our game, it is going to take many steps. Some of our first steps in this project is to understand the overarching ideas when it comes to the code of Q-Learning and Deep Q-Learning. Deep Q-Learning is the fusing of Q-learning and deep neural networks to estimate what is called the Q function. The Q function takes in any given state of the environment (in this case it is our racing game) and returns a Q-value for each possible action the agent can take in the given environment. The Q-value is the expected reward from all future states given that we take the action. Our implementation of this network is going to rely on the main fundamentals of Q-learning with some Deep Q-Learning placed in where we can make it work. This is because using deep neural networks can make the amount of tests needed to achieve a certain goal like one lap around the track in our game take exponentially less attempts then a network that doesn't use it.

This is the main cycle of ideas that we are trying to implement in our code to bring Deep Q-Learning to fruition [8]. The neural network predicts the Q-values for each particular action given the state. In this case there are four different actions that we can take in this environment. Drive forward, backward, and turn left and right. We use an epsilon grading policy to select an action with the highest Q-value or sometimes a random action which is used to incentivise exploration and learning about the environment. We take the action and send it to the simulator which is the game in this case. The simulator updates its state based on that action and returns a reward. The reward can be many different things like the score, lives lost, or completion of a level. At the same time the simulator returns a new state that the neural network takes back in and predicts the new q values and then the loop begins again. How the neural network updates the parameters based on its reward is the most important factor of the Q-Learning algorithm and will be the most complicated for us to implement.

While performing our initial testing and observations we have included a section of terms that we believe are important before we continue moving forward with explaining the details of the neural network we are trying to implement.

Agent: The agent is the entity that interacts with the environment. It takes actions in the environment, receives rewards from the environment, and updates its policy based on the rewards received.

Environment: The environment is the external system with which the agent interacts. It provides feedback to the agent in the form of rewards based on the actions taken by the agent.

Action: An action is a decision made by the agent in response to the environment's state. The agent chooses an action based on its current policy.

Reward: A reward is a signal that the environment sends to the agent, indicating how well it is performing. The goal of the agent is to maximize the total reward it receives over time.

Epsilon: Epsilon is a hyperparameter used to balance the exploration and exploitation trade-off. Epsilon determines the probability that the agent will take a random action instead of choosing the action with the highest Q-value. This randomness helps the agent to explore new areas of the environment, which is important in the early stages of learning.

These are some of the main variables that we have in the codebase for this project with a description of their purpose.

GAMMA = 0.99; is our discount rate for computing our temporal difference target

BATCH_SIZE = 32; how many transitions we are going to sample from the replay buffer when we are computing our gradients.

BUFFER_SIZE = 50000; the maximum number of transitions we are going to store before we start overriding old transitions.

MIN_REPLAY_SIZE = 1000; How many transitions we want in the replay buffer before we start computing gradients and do training.

EPSILON_START = 1.0 and EPSILON_END = 0.05; The starting and ending values of our epsilon.

EPSLION_DECAY = 10000; The decay period in which the epsilon will linearly decline from. So eplsion will go from 1.0 to 0.02 in 10000 steps.

TARGET_UPDATE_FREQ = 1000; The number of steps where we set the target parameter equal to the online parameters.

In addition to all of this to get more experience writing code that works with games we have been taking advantage of OpenAI's gym feature on their website. This is a Github repo that is designed to provide you with the tools of a simulator and you write a deep neural network using reinforcement learning algorithms to solve. We have been treating this as the "Leetcode" for Neural Networks.

V. Future Work

As of now, the car in the pygame program is controlled manually by a human user. Our next step in the game development portion of this project is to start implementing an artificial intelligence driver that will start out with random movement commands for the vehicle and slowly progress, learning how to become a better driver avoiding track borders with each iteration of the Q Learning algorithm.

Additionally, we will need a reward system in order to maximize Q with the Q-Table method. In order to do this portion of the game, we will implement a finish line at the end of track and throughout the track to guide it to the end in the right direction. Each time a reward line is crossed we will reward the artificial intelligence driver for completing a lap.

Once the reward method is implemented, we can change how collisions with the border of the track are handled. As of now, colliding with a wall just stops the car's movement. In order to reinforce desirable behavior and punish failure through the Q-Table method, hitting a wall should be programmed to punish the artificial intelligence driver.

As far as the Deep Q-Learning network goes we have a couple more ideas to test out. We have the beginnings of a basic network that will most likely be used as a foundation for efficiency. We will try to optimize it by adding more hidden layers or convolution layers and adjusting the parameters and reward values. We have read research papers that state that convolution layers can increase efficiency over time. They will have to be put to the test in our game though to see if that holds up.

We also plan to try out different amounts of data to give the network during run time to see what it actually needs to perform well. For example the network can't see the map as we humans see it so we were thinking of having lines come out from the car which can be the network's eyes and ears in game. This would force it to explore around corners in the map to reach the finish line because the boundaries of the map would be clear to its view of the game. This is just one idea that we have but we will have to try a couple out to see if they are actually usable in this project.

VI. References

[1] "Exploration vs. Exploitation - Learning the Optimal Reinforcement Learning Policy." YouTube. YouTube, October 10, 2018. https://www.youtube.com/watch?v=mo96Nqlo1L8

[2] freeCodeCamp.org. "An Introduction to Q-Learning: Reinforcement Learning." freeCodeCamp.org. freeCodeCamp.org, August 9, 2018. https://www.freecodecamp.org/news/an-introduction

-to-q-learning-reinforcement-learning-14ac0b4493cc/

[3] "Playing Geometry Dash with Convolutional Neural Networks." Accessed February 20, 2023. http://cs231n.stanford.edu/reports/2017/pdfs/605.pdf

[4] "Q-Learning Explained - A Reinforcement Learning Technique." YouTube. YouTube, October 5, 2018. https://www.youtube.com/watch?v=qhRNvCVVJaA&list=TLPQMjAwMjIwMjP5wYy7pRhaxg&index=2

[5] "Q-Learning Explained - A Reinforcement Learning Technique." YouTube. YouTube, October 5, 2018. https://www.youtube.com/watch?v=qhRNvCVVJaA.

[6] Shyalika, Chathurangi. "A Beginners Guide to Q-Learning." Medium. Towards Data Science, July 13,2021. https://towardsdatascience.com/a-beginners-guide-to-q-learning-c3e2a30a653c

[7] "Getting Started - wiki" *pygame.org*. [Online]. Available: https://www.pygame.org/news

[8] Analytics Vidhya, "Introduction to Deep Q-Learning for Reinforcement Learning (in Python)," *Analytics Vidhya*, Apr. 18, 2019. https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python