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.

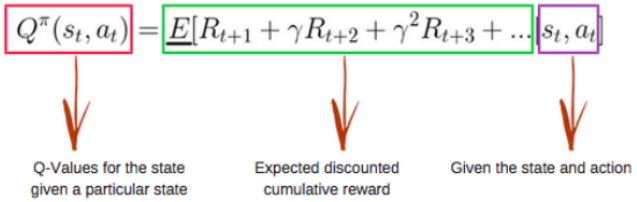


Figure 1: Q-Table Method [2].

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 = 0.9; is our discount rate for computing our temporal difference target

BATCH_SIZE = 1000; 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.

LEARNING_RATE = 0.001; Controls the pace at which an algorithm updates the values.

MAX_MEMORY = 100,000; used to store and manage the experience tuples that are collected during the Q-learning process.

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.

Here is a plan with our current progress taken into account with what we have to do to get a semi working prototype. 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. If we complete these steps we will be able to test our AI and see if it can actually learn from its environment and improve upon itself over time.

V. Complete Results

We have modified the plan since the last documentation of it in the last report. Here is a recap of what we have completed in that time. Last time we had a working pygame code file that is a top down racing game. This game has one simple objective, it is to use the arrow keys to drive around and reach the finish line. The only obstacles are the walls of the track. This game as we left it last time was only controlled by a human player. Here is the work we have done on this project after spending the last couple weeks working on it, we have expanded our code to include 4 files. They are game.py, agent.py, aiModel.py, graph.py.

Game.py contains the racing files and linked textures to make the racing game run. This file now no longer has any features to allow a human player to play the game, this file only allows the agent to control the player's car.

Agent.py implements a Deep Q-Learning algorithm to train an agent to play the Racing game. The code consists of several classes and functions:

The RacingGameAI class: This is a class that represents the Racing game environment. It contains methods to initialize the game, get the current state of

the game, perform a game step (i.e., move the car in the current direction), and check whether the game is over.

The Direction and Point classes: These are helper classes that represent the directions and points on the game board.

The Linear_QNet class: This is a class that defines the neural network used to approximate the Q-value function. It is a simple linear model with one hidden layer.

The QTrainer class: This is a class that defines the optimizer used to train the neural network. It uses the Mean Squared Error (MSE) loss and the Adam optimizer.

The Agent class: This is a class that represents the agent or AI that plays the game. It contains methods to get the current state of the game, remember a transition, train the neural network, and select an action based on the current state.

The train function: This is the main function that trains the agent to play the game. It initializes the game environment and the agent, and then runs a loop where it performs game steps, trains the agent, and updates the plot of the game scores.

The get_state method of the Agent class computes the state of the game based on the current position of the car and the finish line, and the directions of the car and the finish relative to each other. The state is a binary vector of length 11, where each element represents a feature of the game state, such as the proximity of the car to a wall or to its own body, the position of the finish line relative to the car, and the direction of the car.

The get_action method of the Agent class selects an action based on the current state of the game. The agent uses an epsilon-greedy strategy, where it selects a random action with probability epsilon and the action with the highest Q-value with probability 1-epsilon. The epsilon value decreases over time as the agent becomes more experienced.

The train_short_memory method of the Agent class trains the neural network on a single transition, using the Q-learning update rule. The train_long_memory method trains the neural network on a batch of transitions, using the same update rule.

The remember method of the Agent class stores a transition in a memory buffer, which is used to train the neural network in batches. The memory buffer has a maximum size, and when it is full, the oldest transition is removed from the buffer.

The file aiModel.py contains a PyTorch implementation of a Q-learning algorithm for training an AI to play the racing game. The file contains two classes: Linear_QNet and QTrainer. The Linear_QNet class defines the neural network architecture used to approximate the Q-value function. It consists of two linear layers with ReLU activation function. The input to the network is the current state of the game, and the output is a Q-value for each possible action. The class also includes a method for saving the model to a file. The QTrainer class is responsible for training the neural network. It takes in the Linear_QNet model, a learning rate, and a discount factor gamma. It uses the Adam optimizer and mean squared error (MSE) loss function. The train_step method performs one iteration of the Q-learning algorithm. It takes in a tuple of (state, action, reward, next_state, done) as input and updates the weights of the neural network according to the Q-learning update rule:

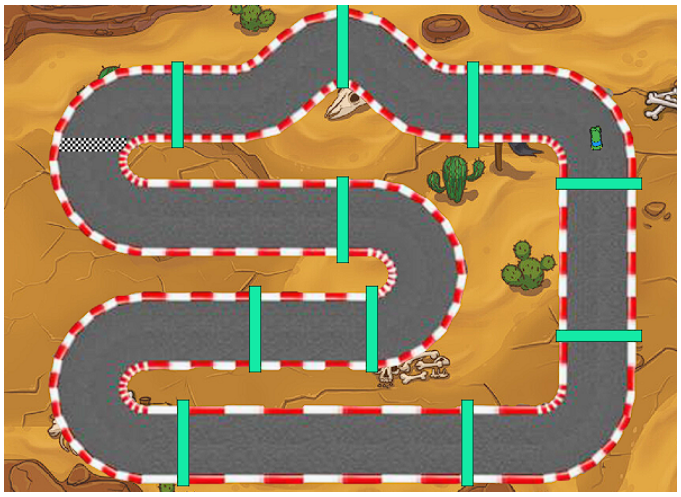
Q(s, a) <- Q(s, a) + alpha * (r + gamma * max(Q(s', a')) - Q(s, a))

where s is the current state, a is the action taken, r is the reward received, s' is the next state, and a' is the next action. If the game is not over (done=False), then the update includes the discounted maximum Q-value for the next state. The loss is then calculated as the MSE between the predicted Q-value and the target Q-value, and the gradients are back propagated to update the weights of the neural network.

Graph.py contains code that defines a function called plot that is used to plot the number of finish lines crossed during the races obtained during the training of the Racing AI. It uses the Matplotlib library to create the plot. The plt.ion() command turns on interactive mode in Matplotlib, which allows the plot to be updated in real-time as the training progresses. The plot function takes in two lists of scores: scores, which contains the number of finish lines crossed obtained for each game played, and mean_scores, which contains the rolling mean of the scores over a specified window size. The function then clears the current figure, creates a new plot, and adds the scores to the plot. The title, x-axis label, and y-axis label are also set. The function also displays the last score obtained and the last mean score obtained on the plot. Finally, the function shows the plot and pauses for 0.1 seconds to allow time for the plot to update. The block=False argument ensures that the function returns immediately so that the program can continue running while the plot is being displayed. This

is incredibly helpful for seeing the AI work in real time because over time the graph should go up because the AI is learning and if it isn't going up it is because we need to adjust our parameters or because we need to give it more time to learn.

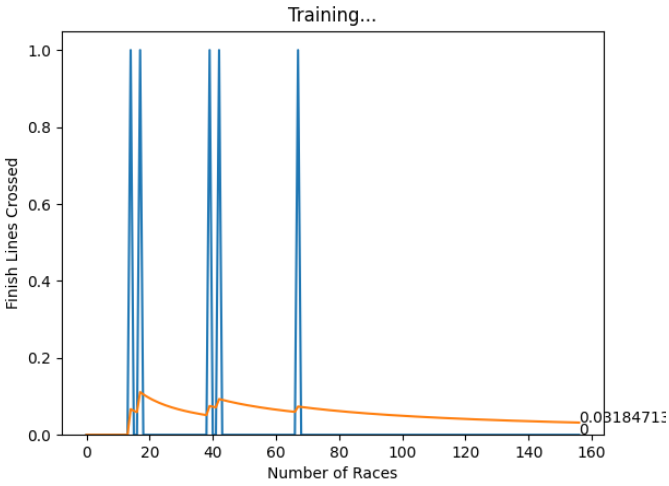
Here are some of our results and observations when working through this project. Originally we had one finish line in our track but we learned that if crossing the finish line is the only way to have the algorithm get rewarded we would have to add more barriers to cross so it will have incentive to go around the track and not get stuck at the start. Also we had to create a negative response if the car went backwards because the algorithm tried early on to just go back and forth over a line and gain positive rewards. To counter this, we made it so that once a barrier is crossed it no longer gives a positive reward and if the barrier directly behind it is crossed a large negative reward is given. This is supposed to incentivise forward movement around the track. Here is a screenshot of the map after we added more barriers for the car to cross.



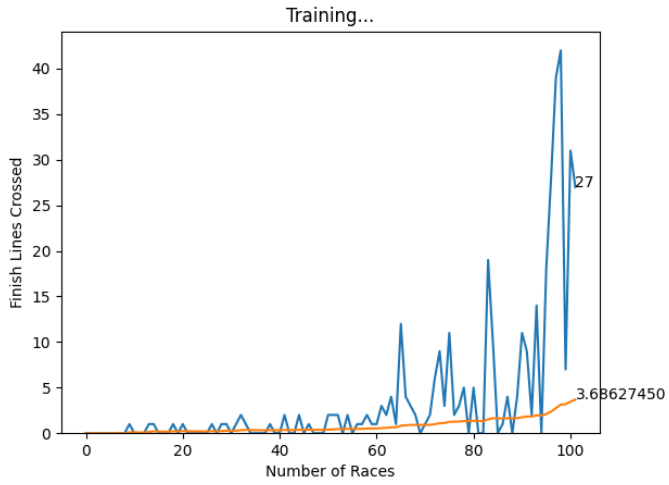
Here is the code to check if the car has crossed the finish line

```
# determine whether car texture collides with finish line texture
finish_poi_collide = player_car.collide(FINISH_MASK, *FINISH_POSITION)
if finish_poi_collide != None:
    # AI has driven in to the finish line from the wrong side
    if finish_poi_collide[1] == 0:
        iteration += 1
        reward = -10
        player_car.bounce()
    # AI has driven through finish line and got reward
    else:
        iteration += 1
        player_car.score += 1
        reward = 10
        player_car.reset()
```

After we were able to get the Q-learning algorithm to work here is the first graph that was produced. This shows an error we made and how the AI is bottlenecked and hits a learning ceiling. The bounce() method set the AI back a distance equal to its velocity, rather than restarting the track. In this test the AI got lucky a couple of the first attempts and got to a barrier and then since it was rewarded early on it just kept on waiting to spawn and go into a finish line. We found out this error was caused by one of our parameters that is supposed to have the AI try a new action if the same result is produced multiple times. In this case the AI would literally try the same thing forever hoping that the results would be the same as its first couple of attempts.



After we fixed that parameters and other errors this is a graph that was produced that shows that the AI is learning and adapting to the environment. This graph shows an increasing number of finish lines crossed over time. This test took about 15 minutes to run and by the end of the test the AI car was able to go around the track many times and it showed signs of trying to optimize its route to achieve rewards faster.

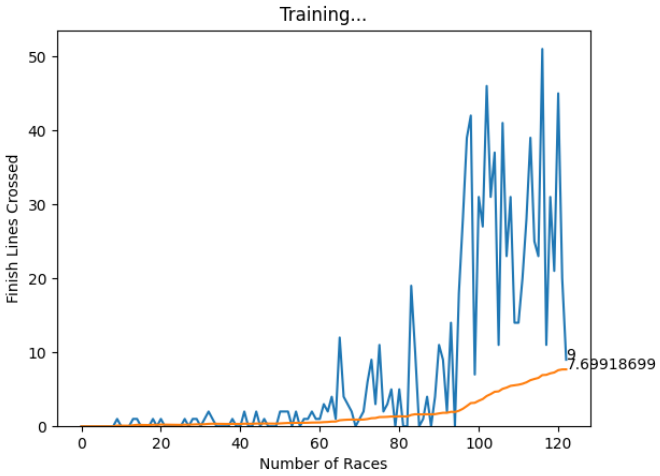


Some of the things that we learned indirectly hurt the AI is time in between positive rewards. Because there is little to negatively reward the AI, the AI started to treat

hitting the wall as a bad thing to do because it lengthened the amount of time it takes to reach the next finish line. Here is the code that checks if the AI car collides with a wall that we outlined around the map.

```
def collide(self, mask, x=0, y=0):
    car_mask = pygame.mask.from_surface(self.img)
    offset = (int(self.x - x), int(self.y - y))
    reward = -10
    self.iteration += 1
    poi = mask.overlap(car_mask, offset)
    return poi, reward, score
```

After spending time optimizing the Q-learning algorithm this is the best result that we were able to produce. It was better than previous tests by a couple points and if we were going to give it more time would probably continue to go upwards until the AI would be able to perfect the map that we had given it.



VI. Future Work After This Course

After working on this project throughout the semester we have learned a lot about Q-learning and reinforcement learning algorithms. There are many different types of algorithms that we could have used for this project, but we feel QLearning helped us gain grasp the concept of AI teaching itself on how to improve at a certain task because of how simple its reward and punishment system works, along with how well defined these two aspects are for a racing game.

As of the addition of checkpoint 'placebo' finish lines in our race track, our AI driver has started to make some more progress. However, some of this progress looks better graphically than it actually is due to the reward value of the checkpoint finish lines being the same as that of the "real" finish line at the end. In order to fix this problem in the future, we should make it so that the reward for crossing a checkpoint is less (about half) than that of crossing the real finish line, so that the AI driver can get some experience and learn the track, but still have an incentive to finish the entire track. Additionally, if we see that lowering the checkpoint reward negatively affects performance too much, we can increase the number of checkpoints along the track in order to make up for it.

Another issue with our implementation is with the collision mechanic. In our racing game, when a car crashes it just bounces back a distance equal to its speed. In a better future implementation of our game, crashing would reset the game and force the AI driver to repeat the entire track once again. This would ensure that results are more accurate and that the AI gets more iterations learning the beginning of the track, which will in turn help it perform better towards the end.

Finally, during training, we left the AI to run uninterrupted for a long duration of time (approximately one to two hours) and when we looked at the performance, the AI was stuck outside of the map. We believe this was due to the hitbox of the track not being created correctly, leading to a gap somewhere in the track. The AI found this gap by accident and got stuck on the outside of the track. In the future, we might want to take a second look at the textures for the track and the track border in order to make sure that they align correctly with one another and to check where that possible gap might be.

VII. 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
[9] J. Achiam and P. Abbell, “Welcome to spinning up in deep rl!¶,” *Welcome to Spinning Up in Deep RL! - Spinning Up documentation*. [Online]. Available: https://spinningup.openai.com/en/latest/index.html [Accessed: 13-Apr-2023].
[10] L. Willems, “LCSWILLEMS/RL-starter-files: RL starter files in order to immediately train, visualize and evaluate an agent without writing any line of code,” *GitHub*, 05-Oct-2022. [Online]. Available: https://github.com/lcswillems/rl-starter-files [Accessed: 17-Apr-2023].
[11]A. Paszke, “Reinforcement learning (DQN) tutorial¶,” *Reinforcement Learning (DQN) Tutorial - PyTorch Tutorials 2.0.0+cu117 documentation*, 23-Mar-2017. [Online]. Available: https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html [Accessed: 17-Apr-2023].
[12]