My Dinosaur Project

Vikas Munukutla, Juan Leis-Pretto, Elijah Freeman, Emanuel Pinilla

Department of Computer Science, Stanford University
450 Serra Mall, Stanford, CA 94305

vikasm2@stanford.edu
jleispre@stanford.edu
elijahf@stanford.edu
epinilla@stanford.edu

***Abstract*—Background: Chrome users find themselves without Internet connection once in a while, and when this happens, they enjoy playing the popular T-Rex, Run! game. In this game, the player controls a dinosaur and accumulates points by jumping over obstacles. The controls are simple – press the space bar or the up arrow to jump up or press the down arrow to duck. In this project, we represent the architecture of the game as a Markov Decision Process in order to train the game and let it play on its own. We collect data by repeatedly playing the game and feed this data into our algorithms. Then, we experiment between algorithms to determine which one is the highest scoring algorithm. Finally, we analyze our resulting scores to determine the effectiveness of our algorithm and areas of improvement for future iterations.**

Keywords—MDP, Q-Learning, T-Rex, Run!

I. Introduction

In the T-Rex, Run! game, players are constantly tasked with having to make a decision between moving, jumping, or ducking. In particular, they have to decide when during the game to jump based on the distance to the nearest obstacle and the velocity of the game. In addition, there is an aspect of uncertainty to these decisions – for example, pressing the space bar too soon or too late might prevent the player from jumping again if another obstacle is quickly approaching. Therefore, T-Rex, Run! is a good candidate for a project whose focus is decision making under uncertainty as some obstacles are not yet visible.

This challenge has been tackled before by other groups. Ke, Zhao, and Wei explored this challenge while taking CS229 in 2017. They used both feature-extracted based algorithms and an end-to-end deep reinforcement learning method to learn the game directly from high-dimensional game screen input [2]. They compared the scores between keep-jump, human-expert, human-optimized, MLP, and deep Q-Learning and realized that deep Q-Learning was the most performant algorithm [2].

Other attempts to tackle this challenge are by GitHub users across the world interested in building high-scoring AIs. For example, we found a repository from Code-Bullet that tried tackling this challenge by using a neural network [3]. This is a different approach than both Ke, Zhao, and Wei's as well as the one we eventually ended up trying out. We were unable to gauge the effectiveness of a neural network to solve this problem but can only imagine there are so many ways to view the problem and apply various AI techniques to it.

The group quickly agreed that an MDP would the best representation of the game out of the tools we had learned about in CS238. However, we needed a way to collect data for the game. So, we looked for open source code for the game in order to begin working on our project. We found a solid, bug-free implementation in this GitHub repo: https://github.com/shivamshekhar/Chrome-T-Rex-Rush - an implementation of the game in Python [1]. Although the implementation was not a perfect replica of the game, it was similar enough to the original to run our MDP on. First, we spent time analyzing the code and understanding how it worked – we spent about 4 days thoroughly understanding the mechanics of the game. Once we understood the code, we planned out the structure of the MDP that would best capture the game. Next, each team member played the game multiple times in order to collect data that the MDP could be trained on. Finally, this data was fed as input to various structure learning algorithms in order to determine the best algorithm to play the game. The following sections will go into detail about each step of the process and how we determined which states and actions to use for the MDP, which algorithm was the most performant, and what our results are.

II. Game Implementation

We used a recreated version of the T-Rex, Run! game in Python in order to collect our data. The implementation was built using the Pygame library in Python, developed by Shivam Shekhar. In a nutshell, Shivam's

implementation creates objects for the Dino, the Cactus, the Pterodactyl, the Ground, and the Scoreboard [1]. It uses a moving slider to slide the game screen to the left as the dino runs in place, moving the obstacles across the screen as it does so. The obstacles include cacti and pterodactyls – cacti can be avoided by jumping over them while pterodactyls are avoided by either jumping or ducking depending on the height of the pterodactyl in the sky.

III. State Representation

In order to start collecting data for our project, we needed to determine what the states in the MDP we were designing should be. In addition, we needed to determine the interval at which we should be collecting states.

Initially, we wanted to collect data after every action in the game. In addition, we determined the most effective state is a tuple consisting of distance to the nearest obstacle, height of the nearest obstacle, and the current velocity of the game. However, we faced some issues with this preliminary plan. First, collecting the data at every action meant that the dinosaur does not know when to start jumping. For example, if the dinosaur does not jump and runs into a cactus, then it will register that it should jump the moment it starts the game because that was the last state – state changes only if action changes. In addition, even if the obstacle passed out of the screen on the left side, its object was not removed from the list of obstacle objects in Shivam's implementation, so instead of calculating distance to the nearest object in the state, we had to calculate distance to the next object.

Therefore, we decided to collect data every frame of the game instead of every action of the game. This means that we would collect data even as the dinosaur is running in the map and not actively jumping or ducking. Every time the dinosaur jumps over an obstacle, its distance to the nearest obstacle is reset to 550 pixels until it gets closer to the next obstacle. Distance to the next obstacle ranges from 0 to 550, except for the beginning of the game – the initial value for the game was 1000. Velocity is determined directly from Shivam's implementation, which increments velocity at a constant interval – it ranges from 4 – 35. Height is 0 for cacti obstacles and 90, 112, and 122 for low-flying, medium-flying, and high-flying birds respectively. We also determined numeric values for different actions: -1 for ducking, 0 for walking, and 1 for jumping. In addition, our initial reward model was the following: the reward for walking or jumping is equal to the distance traveled by taking the action, whereas the reward for hitting an obstacle was -500.

IV. Collecting Data

Once we determined the state tuple, the action, and the reward, as well as the interval at which to collect this data, we wrote some code in Shivam's implementation that would collect this data in every iteration of gameplay, or every frame of the game. This data would be stored in a csv file every time one of us played the game to gather the data.

In total, we collected around 170 games worth of data. This data included a column for every item in the state, which included distance to next obstacle, velocity, height of next obstacle, the action that the dino took from that state, the reward for taking the action from the state, and the next state of the dino. Fig. 1 shows some example rows from the data collection process.

(20	0	4)	1	4	(350	0	5)
(350	0	5)	0	9	(40	0	5)
(40	0	5)	1	5	(446	0	5)
(446	0	5)	0	10	(81	0	5)
(81	0	5)	1	6	(291	0	5)
(291	0	5)	0	7	(31	0	5)
(31	0	5)	1	5	(366	0	5)

Fig. 1. Example data

The first three columns in Fig. 1 represent the starting state of the dino. The fourth column is the action taken, the fifth column is the reward achieved from taking that action, and the last three columns are the resulting state of taking that action from the starting state. The dino occasionally jumps near obstacles and the velocity changes from 4 to 5, as can be seen in the data above. We combined all 170 games worth of data that we had collected into one csv, naming it final_data.csv.

V. Q-Learning V1

Our first attempted implementation was to run a Q-Learning algorithm on the data we had collected. We decided to denote this algorithm as Q-Learning V1. Because we had played so many games, we had essentially seen every possible state and every possible action within the game – we made sure to have trials that ended early because we hit an obstacle on purpose as well as games where we tried to last as long as we could.

Because we had current state as well as next state in final_data.csv, we could run Q-Learning in order to determine the optimal policy for the MDP we had designed for the game. In order to do so, we initialized all Q values for (state, action) pairs to 0. Then, we used the equation in (1) to perform updates on the values of Q.

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma \max Q(s',a') - Q(s,a)) \quad (1)$$

We set α = 0.7 and γ = 0.95 and ran the Q-Learning algorithm for 1,000 games, updating a policy map with the action that should be taken for the given state tuple. We experimented with various values of α and γ and also varied the number of iterations to run the algorithm and realized that α = 0.7 and γ = 0.95 were optimal values for our algorithm. Increasing the number of iterations improved the quality of updates but took longer to run.

After experimenting with the values of α, γ, and the number of iterations and finding the right combination, we saved the policy map in a file called policy.csv. This file contains the state and the action to take from that state. Fig. 2 below shows some example rows from the policy file.

(16	0	4)	1
(16	0	5)	1
(16	0	6)	1
(16	0	7)	0
(16	0	8)	0
(16	0	9)	0

Fig. 2. Examples of policies for each state

Fig. 2 shows that when the dinosaur is 16 pixels away from the next obstacle and there is a cactus coming ahead (denoted by a height of 0). When the velocities are low, such as 4, 5, or 6, the policy map says that the dinosaur should jump over the cactus at this state. However, when the velocity is 7, 8, or 9, the policy map says the optimal action is 0 at the state. This is probably because at higher velocities, the dinosaur should have jumped earlier, and so it no longer needs to jump at a distance of 16 away from the next obstacle.

VI. Results From Q-Learning V1

As we ran Q-Learning V1, we kept track of the scores that the game achieved. We noticed that on average, scores tended to waver around the high 300s/low 400s, with a maximum score of 553. The initial implementation played better than we had expected and was able to successfully determine when to jump over obstacles based on the distance and velocity in the state. Fig. 3 shows the results of 5 separate runs in terms of scores.

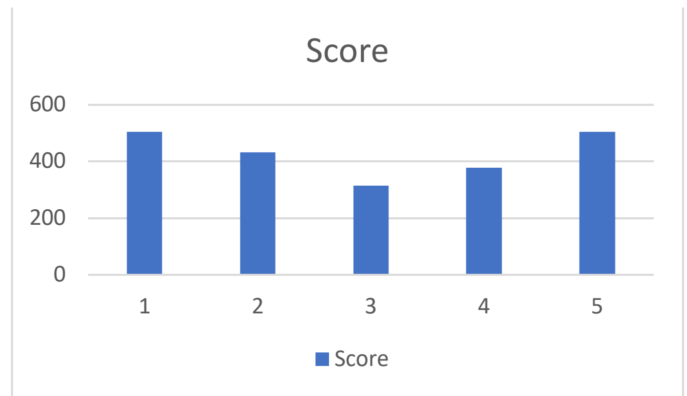

Fig. 3. Scores on 5 runs

However, a pitfall our implementation ran into was with the height aspect of the state. There was nothing inherently wrong with the height, but while the dino was able to clear cacti easily, having only height was insufficient when avoiding pterodactyls. When the pterodactyl was flying so high that the dino had to duck underneath it, the dino would correctly recognize that it had to do so. However, because state is kept track in every frame, it would duck for one frame and immediately stand back up. The length of the pterodactyls in the game were multiple frames long, and so the dino would hit the bird on its way up every single time. We realized that we should have had a mechanism to hold the down key as long as the dino is underneath the pterodactyl, whether that be through a timer or additional state information.

In addition, Q-Learning V1 was limited by how good our group could play the game. This is because the dino was running Q-Learning on the data we provided it, and in particular, it was relying on the velocity field within the state tuple. Unfortunately, this meant that our team only reached a certain velocity threshold while playing the game, and this threshold was the maximum velocity the dino game could predict actions on. There was an inherent cap on the high score the dino could receive because the state space was sparse.

Overall, we believe the dino game ended whenever the dino hit a bird that it had to go under or the velocity became too high.

VII. Q-Learning V2

The aim of Q-Learning V2 was to address as many of the shortcomings as possible from Q-Learning V1. First, we decided to make a small modification to address excess jumping – a problem that occurred to us in the beginning stages as we were formulating the template for state space. While the dino in Q-Learning V1 did not jump into obstacles, it occasionally jumped without a reason to do so. Deeming this risky behavior, we decided to change the reward system to reflect that we didn't want the dino to jump when it didn't need to. The rewards in Q-Learning

V2 for jumping over a cactus, jumping over no cactus, hitting an obstacle, and doing nothing are +10, -10, -500, and 0, respectively.

In addition, after facing difficulties with the pterodactyl in the game, we decided to remove pterodactyls entirely and only deal with cacti in our initial project timeline. We wanted to focus on improving our algorithm given the state space we had already prepared for – incorporating pterodactyls into the game could result in recollecting 170 games worth of data. Given the schedule of the project, we wanted to work on the meat of the project rather than collect more data. After removing the pterodactyl from the game, we could safely ignore any entries where the action was -1 because we no longer needed to duck. Because we only had two actions now (0 for continue walking and 1 for jumping), we decided to use false to represent walking and true to represent jumping. In addition, we could ignore the height parameter in the state tuple because all obstacles were of height 0 now.

The biggest improvement of the algorithm over Q-Learning V1 was the way the dino learned the MDP structure. In V1, we provided the data as input for the dino, providing it with starting states, actions, rewards, and the ending states. In V2, we started the dino with a blank slate by completely discarding our data. The dino would learn on its own by continually losing the game and learning from its mistakes, updating Q values for each (state, action) pair based on the reward it recognized while playing the game. This was an especially interesting implementation to watch because the first game the dino played, we could see the dino crash into the first cactus it saw, then in later games try jumping all the time to see if it would avoid the cactus, learn that it needs to recognize cacti in front of it before jumping, and slowly reduce the amount of times it jumps until it only jumps before cacti. Because the dino was learning through its own experience and playing the game itself, it was no longer limited by human experience because our data did not come with a velocity threshold. The dino could figure out the optimal action at any velocity by continually trying until it recognized which distance to jump at for the optimal reward.

We experimented with other additions, not all of which survived the cut to be a part of Q-Learning V2. For example, we experimented with an ε-greedy approach with $\varepsilon = 0.1$, but we found that we got far lower scores by randomly selecting an action with this probability since there would be many chances to jump onto or run into a cactus. Thus, we ultimately opted not to use an ε-greedy algorithm since the exploration came in the form of randomly jumping at unknown states. Lastly, for faster convergence, we used interpolation. After each death, for every unseen state su, we check the eight states nearest (from closest to farthest). If we find state ss that has been seen, we set the Q-value of the pair consisting of su and best action from ss to be 1.0 (the other value will be 0.0). In other words $Q[su][\max_a Q[ss][a]] = 1.0$.

VIII. Results From Q-Learning V2

As we ran Q-Learning V2, we kept track of the scores that the game achieved. We noticed that on average, scores tended to waver around the low 700s, with a maximum score of 909. Q-Learning V2 played better than Q-Learning V1, mostly probably because it no longer had a velocity cap and could not hit pterodactyls anymore. Fig. 4 shows the results of 5 separate runs in terms of scores.

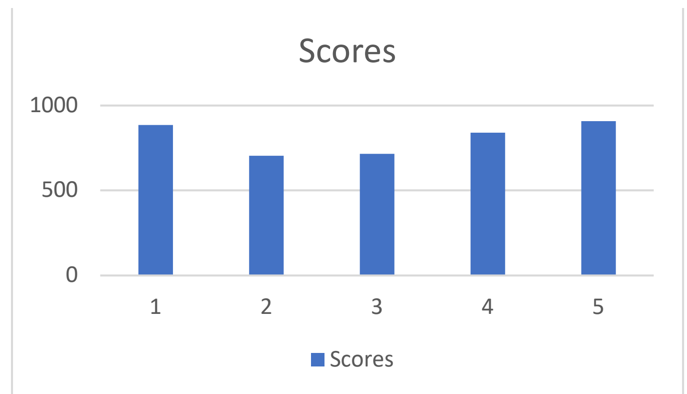

Fig. 4. Scores on 5 runs

These scores show major improvement over the scores achieved in Q-Learning V1. As we ran Q-Learning V2, we saw the dino get progressively better on every run. Eventually, we had to stop running the program because it seemed like the dino had converged at a high score of 909. This is probably because the updates as the dino plays at those high scores are so minute that it barely budges the dino's policy, and it needs a more nuanced approach that is more advanced than the one we are attempting to do, perhaps like the concepts in Ke, Zhao, and Wei's project [2].

IX. Conclusion

Overall, we were very excited to see our dino get progressively better at playing the game. However, this was our first iteration of the project timeline and we hope to incorporate more advanced features in the future.

For example, we would definitely like to reincorporate pterodactyls into the game. To do so, we could use the height field in the state tuple again, and if the height is low enough the dino jumps over the pterodactyl. This functionality exists in our current implementation. However, when the height is high and the dino has to duck, we have multiple options. The dino can be programmed to "hold" the down button for a fixed time that is determined after experimentation. This could be adjusted based on the

length of the pterodactyl and the velocity of the game at the time. However, to make the game completely autonomous, we could instead have the dinosaur detect the vertical distance to the first thing that blocks it directly above it. As long as the distance is smaller than the distance to the ceiling, the dino could continue holding the down arrow – the state of holding "down" would have to be added to the state tuple.

In addition, as we saw through our experimentation, Q-Learning has its limitations. It is not perfect and can only run until convergence or until it becomes too costly to continue training. Given the tools we have learned in CS238, we hope to explore other interpolation methods and see which ones are the most effective. In general, we hope to explore some of the techniques outlined in Ke, Zhao, and Wei's implementation [2] or come up with our own as we continue taking classes in AI and are able to combine algorithms from various AI classes to compare results between the algorithms.

References

[1] Shekhar, S. (2019). *shivamshekhar/Chrome-T-Rex-Rush*. [online] GitHub. Available at: https://github.com/shivamshekhar/Chrome-T-Rex-Rush [Accessed 6 Dec. 2019].

[2] Ke, Zhao, Wei, "AI For Chrome Offline Dinosaur Game", Department of Computer Science, Stanford University, Stanford, Sep. 22, 2017

[3] GitHub. (2019). *Code-Bullet/Google-Chrome-Dino-Game-AI*. [online] Available at: https://github.com/Code-Bullet/Google-Chrome-Dino-Game-AI/tree/master/DinoGame [Accessed 6 Dec. 2019].

Group Contributions

All members contributed to all parts, but certain members took charge with different aspects. Taking ownership helped increase the development process while still allowing each member to contribute to the team effort. These are the areas each team member honed.

Juan Leis-Pretto worked on the initial logging of data. He edited the game file to log data at every action taken during a game run. He did this by identifying the points in which the state and reward could be calculated. Furthermore, he helped decide how to log state and how to feed the dinosaur actions based on a generated policy.

Emanuel Pinilla helped Juan Leis-Pretto with the initial data logging and early definition of states. Emanuel had a lot of focus on collecting game data for the initial version of our MDP. Emanuel also contributed with an early Q-Learning program which helped identify early issues with the data logging and state definition but was not used in the final product.

Vikas Munukutla contributed by reading and understanding Shivam's codebase in order to begin the data logging and collection phase of the project. Vikas also helped collect data and spent the majority of his time and energy on documenting the progress of the project while crafting the final paper.

Elijah Amir Freeman helped write the Q-Learning algorithm. He focused on figuring out the best hyperparameters in order to optimize the dino's run score. He further contributed by deciding on the specifics of the algorithm, such as the exploration algorithm. Lastly, Elijah Amir Freeman helped Vikas on crafting the paper by helping identify key turning points during the process.