Playing the Chrome Dino Game with Reinforcement Learning: A Deep Q-Learning Approach with Custom Environment Design

Malvi Bid

COMP3071 Designing Intelligent Agents
School of Computer Science,
University of Nottingham Malaysia

Abstract. This report presents the development and evaluation of a Deep Q-Network (DQN) agent for playing the Chrome Dino Game by interacting with a custom-made Chrome Dino Game environment. The project involved multiple training rounds, with the first round resulting in the best-performing model with the highest observed score of 1618. Despite some limitations in handling specific obstacles, the agent achieved considerable success in the game. The findings provide a foundation for future research in optimizing DQN agents for more complex tasks and environments, including the exploration of advanced learning strategies such as Prioritized Experience Replay.

Keywords: Chrome Dino Game, Reinforcement Learning, DQN, Game Automation.

1 Introduction

In this report, I address the challenge of designing an intelligent agent capable of efficiently playing the Chrome Dino game and achieving high scores. The Chrome Dino game is a seemingly simple yet engaging game in which the player controls a dinosaur traversing a desert landscape, avoiding obstacles, which include Small Cactus, Large Cactus and Pterodactyl, by either jumping over or ducking under the obstacles. Despite its apparent simplicity, the game necessitates rapid reflexes and precise timing, rendering it a fitting testbed for the development of intelligent agents that can acquire complex skills through trial and error.

The primary challenge in constructing such an agent lies in devising an environment that accurately reflects the game's dynamics, encompassing the dinosaur's movements, the obstacle placements, and the reward structure. Moreover, the agent must learn a policy that maximizes its long-term reward, despite the stochastic and partially observable nature of the game. Additionally, the agent should be capable of adapting to variations in the game's difficulty level, including fluctuations in the game speed and obstacles frequency, ensuring robust performance.

To overcome these challenges, I created a custom environment for the game, which allows for accurate representation and interaction with the game's dynamics, and I employed Reinforcement Learning with Deep Q-Learning, to develop an intelligent agent adept at playing the Chrome Dino game.

2 Literature Review

This review examines different approaches used for playing the Chrome Dino game, including Rule-based Systems, Genetic Algorithms, Neural Networks, and Hybrid Approaches.

Rule-based systems rely on predefined rules for decision-making and action-taking within the game. These rules can be hand-coded or generated automatically using techniques like decision trees. This results in simple reflex agents which select actions based on the current percept (state), ignoring the rest of the percept history. For example, an agent might jump when an obstacle is detected or adjust the jump height and distance based on an observed obstacle's size and position. Although simple to implement, these systems have limitations in handling complex situations and require extensive manual tuning to perform well. Moreover, reflex agents base their actions on a direct mapping from states to actions, thus, such agents cannot operate well in environments, like the Chrome Dino game, for which this mapping would be too large to store and would take too long to learn. (Russell & Norvig, 2014)

In contrast, Goal-based agents such as Genetic algorithms are more flexible; along with current state description, the agent has a goal to complete. Genetic algorithms evolve a population of Chrome Dino players by applying natural selection to different sets of inputs and output actions (Ridley, 2009) (Mitchell, 1998). Over time, the best-performing (fittest) players can be identified and used to create a new generation of players with slightly modified input/output mappings. This approach is capable of exploring a wide range of possible solutions and adapting to changing conditions. However, genetic algorithms may require many generations to find a good solution, resulting in high computational costs and potentially suboptimal solutions (Mitchell, 1998).

Then we have Learning agents such as Neural Networks, where the agent is allowed to operate in initially unknown environments and then become more competent than its initial knowledge alone might allow. Neural networks can be trained to learn a policy for playing the Chrome Dino game using datasets of game screens and corresponding actions (Mnih et al., 2013). This approach often results in more accurate and robust game-playing strategies than other methods but requires a large amount of training data and significant computational resources. Recent research has demonstrated the potential of neural networks in achieving high scores in the Chrome Dino game and similar tasks (Ke, Zhao & Wei, 2016) (Mnih et al., 2015).

Hybrid approaches combine different techniques, such as rule-based algorithms and neural networks, to create more effective Chrome Dino players. These approaches leverage the strengths of each method while mitigating their weaknesses. By incorporating Reinforcement Learning with Neural Networks, a hybrid approach can learn from experience and adapt to the game's stochastic nature while benefiting from the generalization capabilities of neural networks.

Although each method has its strengths and weaknesses, the use of a hybrid approach that combines Reinforcement Learning with Neural Networks offers a promising avenue for achieving high performance and adaptability in playing the Chrome Dino game. Future research in this area should continue to explore and refine these hybrid techniques, potentially leading to even better performance in the game and other AI tasks.

3 Methodology [*,**]

The Reinforcement learning (RL) problem involves an agent interacting with an environment by executing a sequence of actions and learning from the observations and rewards received [12].

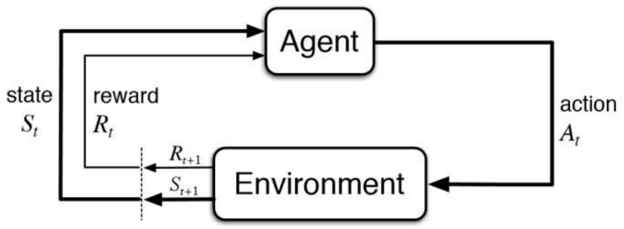

Fig. 1. Data flow in Reinforcement Learning training pipeline

I created a custom environment, `DinoEnvironment` which is a subclass of the `Env` class from the Gymnasium library. I used the Selenium Test Automation library to access the Chrome Dino game webpage, get information about the game state as well as to perform actions predicted by the agent. Key methods of the `DinoEnvironment` class include:

- `__init__`: Initializes the environment, including the screen dimensions, observation space, and action space. The actions include Jump, Start Ducking, Stop Ducking or Run/Do nothing. I separated the Ducking action into two separate actions as I wanted the agent to also learn the ideal Duck duration.
- `_create_driver`: Sets up the `ChromeDriver` instance for controlling the browser.
- Various helper methods for extracting game state information, such as obstacles, game speed, and T-Rex position. To extract information about the game state I used properties of the games `Runner.instance_` JavaScript object which is accessible via the Chrome browsers console. This state information are the input features of the agent's neural network.
- `reset`: Resets the environment and starts a new game.
- `get_observation`: Returns the current state of the game.
- `is_game_over`: Determines if the game is over.

[*] Project repository: https://github.com/malvibid/COMP3071-Designing-Intelligent-Agents/tree/coursework
[**] Video featuring a selection of test rounds demonstrating the best trained agent playing the game: https://youtu.be/QFf0_4FCh0w.

- `get_reward`: Returns the reward for the current state of the game.
- `step`: Takes an action and returns the resulting observation, reward, done, and additional information.
- `render`: Visualizes the game in the specified mode.
- `close`: Closes the game environment and the driver.

The agent is defined in the `DinoDQNAgent` class, and it includes an implementation of a Deep Q-Network (DQN). DQN is an extension of Q-learning that uses neural networks as function approximators to estimate the Q-values for state-action pairs. Q-values are used to determine how good an action a taken at a particular state s is. A shallow neural network is used with three fully connected (linear) layers: input layer (10 input neurons, 64 output neurons), 1 hidden layer (64 input neurons, 128 output neurons), and an output layer (128 input neurons, 4 output neurons), with ReLU activation functions between them.

The game is stochastic, and its internal state is not observed by the agent. Instead, the agent only observes a snapshot of the current game state which is received from the environment. Thus, the agent doesn't start with perfect knowledge of the environment but learns through experience as it interacts with the environment.

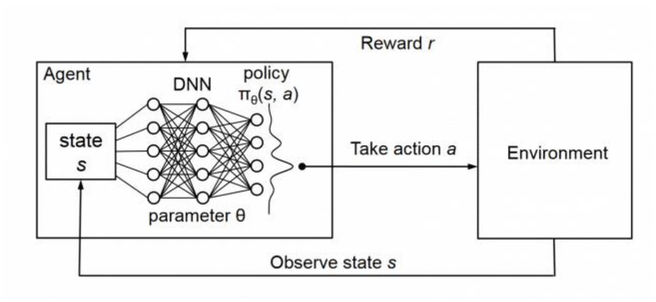

Fig. 2. Deep Q-Network

The agent iteratively updates its Q-function $Q(s, a)$, i.e., the neural network, based on its experiences in the environment. At each time-step the agent selects an action a_t from a set of actions $A_t = \{0,1,2,3\}$ using the epsilon-greedy exploration-exploitation strategy, balancing the need to explore new actions and exploit the current knowledge of the Q-function. The action is passed to `DinoEnvironment` and modifies the games internal state and the game score. The agent collects the experience, in the form of a tuple of `(state, action, reward, next_state, done)`, which is generated from its interaction with the environment, and it stores this experience in a replay buffer (which is the agent's memory). The replay buffer is designed to break the correlation between consecutive state transitions by randomly sampling a batch of experiences for the agent to learn from during each update step. This process is known as experience replay and helps improve the stability and convergence of the learning process.

The state is represented as a tuple of ten 32-bit floating point values which form the DQN model's input : `(trex_y, trex_is_jumping, trex_is_ducking, game_speed, distance_to_next_obstacle, obstacle_type, obstacle_x, obstacle_y, obstacle_width, obstacle_height)`. The goal is to find an optimal policy that maximizes the cumulative reward. To achieve this, the agent aims to learn the optimal Q-function, denoted as $Q^*(s, a)$, which represents the maximum expected cumulative reward when the agent starts from state s, takes action a, and follows the optimal policy afterward.

As the agent interacts with the environment and updates its Q-function using the Bellman equation (Mnih et al., 2013) [6], the Q-values converge to the optimal Q-function. The optimal policy can then be derived from the optimal Q-function by choosing the action with the highest Q-value in each state.

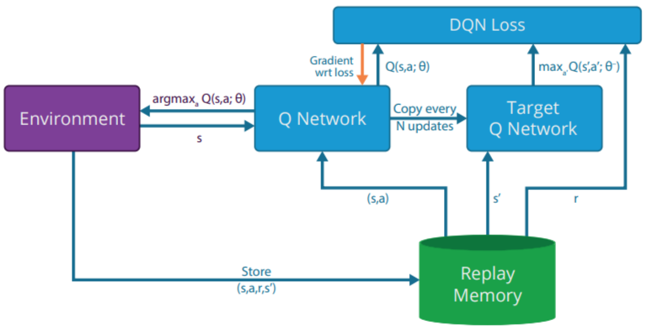

Fig. 3. Deep Q-Learning

Key methods of the `DinoDQNAgent` class include:

- `__init__` : Initializes the agent's parameters, such as the environment, state and action sizes, discount factor (gamma), exploration rate (epsilon) and its decay, learning rate, batch size, and memory size. It also creates the DQN model, optimizer, and loss function. All parameters are initialized with default values that have been known to work well for deep learning tasks.
- `_build_model`: Builds the DQN model architecture using PyTorch, which is a simple feed-forward neural network with one hidden layer and ReLU activation functions.
- `remember`: Appends a new experience (state, action, reward, next_state, done) to the agent's memory, which is used for training.
- `act`: Determines the action to take given a state, following the epsilon-greedy strategy. It either selects a random action (exploration) or the action with the highest predicted Q-value (exploitation of gathered knowledge) based on the current epsilon value.
- `replay`: This method is called periodically during training to update/train the DQN model using a batch of experiences randomly sampled from the memory. It updates the model's weights to minimize the mean squared error (MSE) between the target Q-values (ground truth) and the predicted Q-values for a set of state-action pairs. The target Q-values are calculated by adding the immediate reward and the discounted maximum Q-value of the next state. The epsilon value is also decayed in this method.
- `save_model` and `load_model`: Functions to save and load the trained DQN model's weights to/from a `.pth` file.

4 Experiment

During the training process, I discovered that the method of rewarding the agent had a notable impact on its performance. Consequently, I conducted several training rounds with different reward strategies to optimize the agent's learning.

In the initial training round, train_1, the reward structure consisted of the following:

- A reward of 0.1 for staying alive at each step.
- A reward of 0.5 for passing an obstacle.
- A penalty of -10 for crashing into an obstacle.
- A bonus reward of 1 for surpassing the high score at each step.

After just 100 rounds of training, I obtained the best model. This model had learned to jump and run effectively; however, it did not duck and occasionally performed unnecessary jumps and ducks. Consequently, in subsequent training rounds (train_2 through train_5), I experimented with various reward values and rules to encourage the agent to learn the correct actions. The rules included the following:

- Penalizing crashes into obstacles.
- Rewarding surpassing the high score.
- Rewarding staying alive.
- Penalizing unnecessary jumps and ducks when there are no obstacles.
- Penalizing jumping when there's enough space to duck under flying obstacles.
- Penalizing ducking when the obstacle is on the ground.
- Rewarding passing an obstacle.
- Providing a small reward for every step the agent surpasses the high score.

During the 3rd and 4th training iteration (`train_3` and `train_4`), I also tried to add more information about the T-rex to the state: Trex's standing and ducking width and height. As this increased the shape of the observation space from 10 to 14, I created a new class to test this strategy called `ModifiedDinoEnvironment`.

However, in all my experiments, incorporating numerous reward rules turned out to be excessively strict, and the model often struggled to learn any aspect of the game effectively.

In training iteration 6 (`train_6`), I introduced an experimental technique called "exploration breaks." This approach periodically sets the epsilon value to zero, allowing the agent to fully exploit its learned knowledge without any exploration. The primary motivation behind implementing exploration breaks is twofold. First, it enables a more accurate assessment of the agent's training progress by observing its performance in a purely exploitative mode. Second, it helps the agent reach more challenging stages of the game and gather experiences from those stages if it has already mastered the earlier stages. After the break, the epsilon value is reset to its previous value, and the agent continues with the scheduled epsilon decay until the next exploration break is initiated. This method aims to enhance the learning process by balancing exploration and exploitation while providing valuable insights into the agent's performance during training.

All the trained models along with the best trained model are accessible through the project repository[1].

5 Result

Among all the trained models, the best performing agent was obtained from the first training round, named as "train_1: episode_100.pth". The agent demonstrated proficiency in jumping over obstacles and running when there were no obstacles. Although it occasionally ducked at crucial moments, the rationale behind this choice remained unclear since in other similar situations, the agent opted to jump. The model also adapted well to the game's gradual increase in speed. However, most of the instances in which the agent crashed occurred because it failed to duck when it was necessary to do so. This can be attributed to the epsilon-greedy strategy that was used to train the model. The random actions taken while exploring the environment often resulted in crashes, preventing the agent from reaching more difficult stages of the game. Consequently, the agent doesn't get the opportunity to train and learn from more complex scenarios, Thus, the agent struggled to learn how to duck properly, which is primarily required later in the game. Overall, the agent's performance was satisfactory, but it could have significantly improved if it had mastered ducking.

The highest score observed when testing this model was 1,618. To further evaluate the performance of this model, it was run for 50 episodes. The key performance metrics are shown in **Fig. 4**. The highest score achieved by the agent during these test runs was 1,467 and the lowest score was 55 (It is worth noting that the low score of 55 may have been influenced by a screen recording running simultaneously, which caused the browser to glitch and affect the agent's performance.). The average score is trending upwards for each episode during testing, indicating that the agent is continuing to learn and improve its performance in the Chrome Dino Game. This suggests that the model demonstrates a promising capacity for adaptability and ongoing learning.

Epsilon was set to zero during testing to exclusively evaluate the model's acquired knowledge, without any exploration. The noticeable fluctuations in the episode reward and episode score graphs emphasize the stochastic nature of the Chrome Dino Game. These variations can be attributed to the inherent randomness within the game, which presents the agent with diverse and unpredictable scenarios to manage. It not only showcases the challenges faced by the agent in learning an optimal policy, but also highlight the dynamic and complex nature of the game environment, making the agent's task of mastering the game even more demanding.

This video features a selection of test rounds demonstrating the best trained Dino agent skillfully playing the game: https://youtu.be/QFf0_4FCh0w.

[1] **Trained models**
https://github.com/malvibid/COMP3071-Designing-Intelligent-Agents/tree/coursework/COMP3071-DIA-CW/src/trained_models

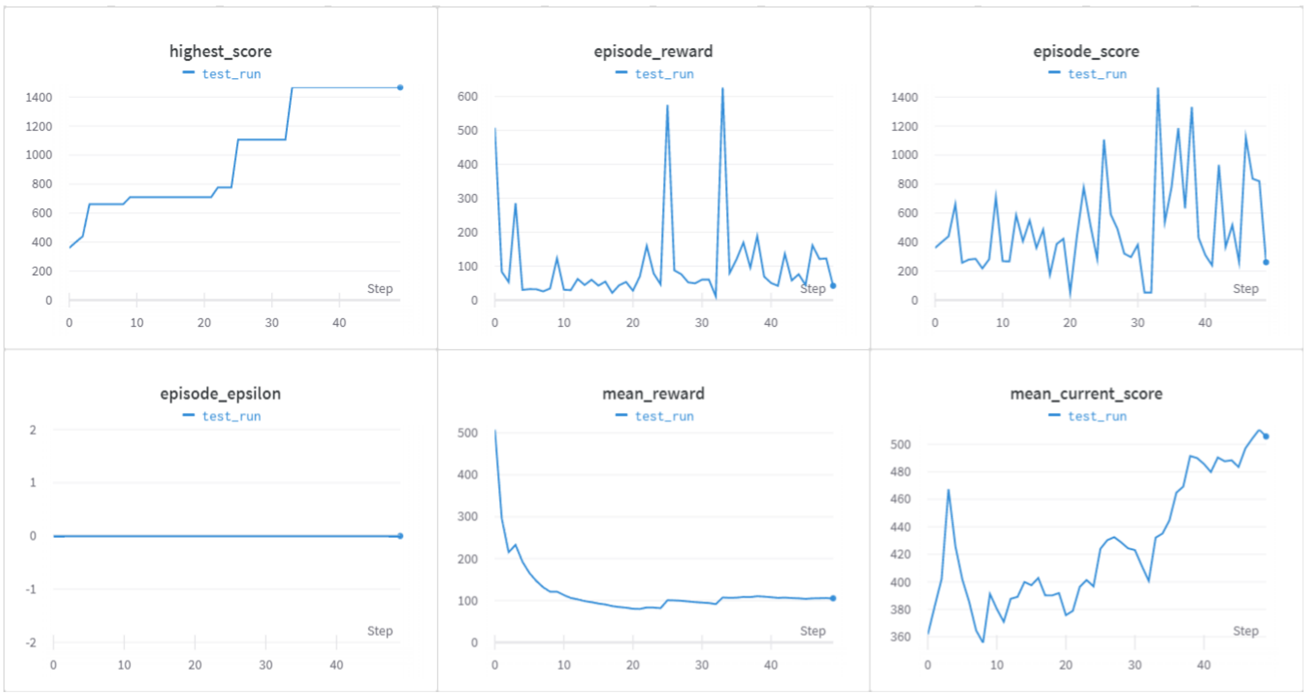

Fig. 4. Test performance metrics of best model - 50 episodes

6 Discussion

Initially, I had planned to collected the observations from the environment using a series of screenshots of the game screen as done in (Renotte, 2022) [8] and (Ivanova, 2021) [2]. This can give a richer representation of the game and allow the agent to learn complex features without manually specifying them. However, it also adds a layer of complexity to the model as it requires defining and training a Convolutional Neural Network to extract features from the screenshots to feed to the DQN model. For playing the Chrome Dino Game, this approach seems inefficient and computationally expensive. This inefficiency is evident in the fact that after running the training overnight, for 88000 steps, (Renotte, 2022) received a high score of only 444 while (Ivanova, 2021) achieved a modest score of 258 after 1 980 000 steps and 24 hours of training. Instead, I opted to access properties of the game's `Runner.instance_` JavaScript object to extract relevant information, such as obstacles on the screen, the T-Rex, game speed, and so on. This approach provided a compact and direct numerical representation of the game. Admittedly, one could argue that I introduced bias by hand-selecting the features used to represent the game state based on what I deemed important for the agent to know. Nevertheless, my approach ultimately proved to be more efficient with less computation, as it offered a direct representation of the game.

The main challenge faced during training stems from the stochastic nature of the game and the experience replay mechanism's random sampling approach. Due to the game's inherent randomness, the agent is exposed to various situations, making it more challenging to learn an optimal policy. Additionally, the experience replay technique uniformly samples experiences from the memory buffer, giving equal importance to both good and bad actions during training. Consequently, the agent may struggle to learn from more relevant experiences or distinguish between high-quality and low-quality actions, slowing down the learning process or leading to suboptimal performance as well as causing fluctuations in performance between episodes. This also means that training a new model with the same algorithm does not guarantee same performance, as the experiences with which the model learns are randomly selected.

The challenge is further exacerbated by the epsilon-greedy strategy employed for exploration during the agent's learning process. Due to the random actions taken by the agent while exploring, the game often ends prematurely, preventing the agent from reaching more difficult stages of the game. Consequently, the agent doesn't get the opportunity to train and learn from more complex scenarios, ultimately hindering its ability to develop an optimal policy for handling more advanced challenges within the game. This issue led to a limited learning experience and resulted in suboptimal performance when facing unfamiliar or difficult game situations.

Lastly, due to the inconsistent performance between different episodes, monitoring the model's performance during training was difficult and it became challenging to determine which model to save as a checkpoint. Given the unfeasibility of saving a model for every episode, there is a risk of losing potentially well-performing models that do not fall within the designated save intervals. This makes it difficult to retain and utilize the best-performing models throughout the training process.

It may be apparent that many of the challenges encountered during the training process stemmed from the random sampling of the replay buffer and the epsilon-greedy strategy's random actions during exploration. While these techniques are essential for effective reinforcement learning, they can be optimized to facilitate better learning for the agent. Approaches such as employing Prioritized Experience Replay or implementing a fixed exploration phase can help address these challenges and enhance the agent's learning process. Prioritized Experience Replay is a technique that assigns importance weights to each experience in the memory buffer, allowing the agent to sample more meaningful experiences more frequently during learning. On the other hand, a fixed exploration phase refers to a predetermined period during which the agent focuses on exploring the environment and gathering experiences before transitioning to the exploitation phase, where it starts learning and refining its policy.

7 Conclusion

In conclusion, the best-performing model demonstrated considerable success in playing the Chrome Dino Game despite some limitations in handling specific obstacles. This project has laid a foundation for future research in optimizing DQN agents for more complex tasks and environments. Further work could explore techniques that encourage agents to learn from meaningful experiences, such as Prioritized Experience Replay and other advanced learning strategies.

8 Reference

1. Brunton, S. (2022). *Q-Learning: Model Free Reinforcement Learning and Temporal Difference Learning*. https://youtu.be/0iqz4tcKN58
2. Ivanova, I. (2021). *How to play Google Chrome Dino game using reinforcement learning*. https://medium.com/deelvin-machine-learning/how-to-play-google-chrome-dino-game-using-reinforcement-learning-d5b99a5d7e04
3. Ke, J., Zhao, Y., & Wei, H. (2016). AI for Chrome Offline Dinosaur Game. http://cs229.stanford.edu/proj2016/report/KeZhaoWei-AIForChromeOfflineDinosaurGame-report.pdf
4. Krohn, J. (2018). *Deep Q Learning Networks*. https://www.youtube.com/watch?v=OYhFoMySoVs
5. Mitchell, M. (1998). *An Introduction to Genetic Algorithms*. The MIT Press. https://doi.org/10.7551/mitpress/3927.001.0001
6. Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing Atari with Deep Reinforcement Learning. https://doi.org/10.48550/ARXIV.1312.5602
7. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... & Hassabis, D. (2015). Human-level control through deep reinforcement learning. Nature, 518(7540), 529-533. https://doi.org/10.1038/nature14236
8. Renotte, N. (Director). (2022). Build a Chrome Dino Game AI Model with Python | AI Learns to Play Dino Game.
9. Ridley, M. (2009). Evolution (3rd ed). John Wiley & Sons.
10. Russell, S. J., & Norvig, P. (2014). *Artificial intelligence: A modern approach* (3. ed., Pearson new international ed). Pearson.
11. Sendtex (2019). *Reinforcement Learning*. https://youtube.com/playlist?list=PLQVvvaa0QuDezJFIOU5wDdfy4e9vdnx-7
12. Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction* (Second edition). The MIT Press.