Using Deep Q-Learning to Train Google Dinosaur Game Agent (A.K.A *"Lecio Rex"*)

Lecio Ribeiro

CS 470, Winter 2025

Department of Computer Science

Brigham Young University

**Github repo - https://github.com/leciojor/QlearningDino*
Agents Video -* ***https://youtu.be/571XWfvsm1c*
Time Spent - Around* ***22 hours*
**Models Trained Drive Folder - https://drive.google.com/drive/folders/1g76ifkWFXrleadCCc6gLeKu02RvYiaiR?usp=sharing*

Abstract

This project's main goal was to build a Deep Q-Learning-based agent that would be capable of playing the famous Dinosaur game from Google. This idea has been replicated in different ways throughout the internet, like in the paper *Chrome Dino Run using Reinforcement Learning* (Ariyalur & Arya, 2020), so there was a lot to base on. The first step was to set up the environment in which the agent would act. This led to the definition of the other aspects of Markovian decision systems (considering Q-learning was the main algorithm used) like the states given to the Q values network (A.K.A, the agent sensors), the rewards values for each state/action pair, etc. Once that was all defined, two main NN architectures were tested with several different hyperparameter variations along with variations of the state sensor logic and the reward logic. The final results were pretty disappointing, but at least the final models were able to perform some individual tasks required by the game (but not play it fully and stay alive for very long). Finally, this paper presents some things that could have been done for better results (e.g., parallel training, using CNN and game screenshots as states, etc).

1. Agent Purpose

The agent was designed to play the famous Dino Run game from Google for indefinite time. This game consists of jumping/crawling obstacles in a 2D environment, while managing the speed increase over time. I.e., the agent needs to learn the right timing to jump/crawl obstacles, along with when to jump, crawl, or not do anything. To reflect this, the reward logic was designed to encourage better timing and when to do certain actions (better explained in the next section).

.

2. Agent Design

This section describes how the agent was designed based on the assumptions and elements of a Markovian Decision Process.

a. Environment Setup

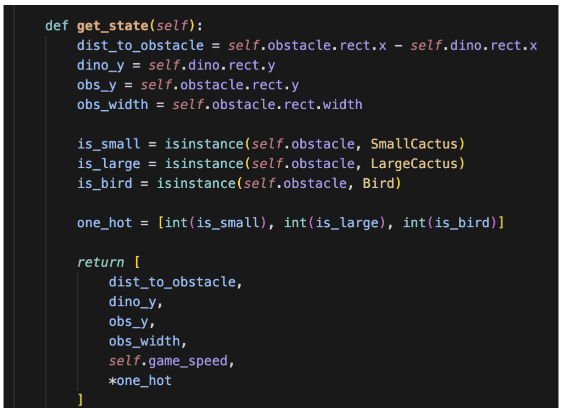
To fully represent the game's environment, collect the state's information and better visualize the agent performance (instead of just seeing the raw reward numbers), this project needed a fully playable version of the game. So, to save time, this implementation "https://github.com/MaxRohowsky/chrome-dinosaur" from Max Rohowsky of the game with the pygame framework was used as the base logic of the environment. Moreover, for a full implementation of the logic of passing the sensor inputs to the agent, using the agents, etc, additional logic was implemented and can be found at this project's repo on the top of this paper.

b. Emulator Setup for Training

Several modifications were made to the base pygame logic, including the reward function R(state, action) and the state information passed as input to the agent, to represent the environment for training the agent network.

i. States Sensors

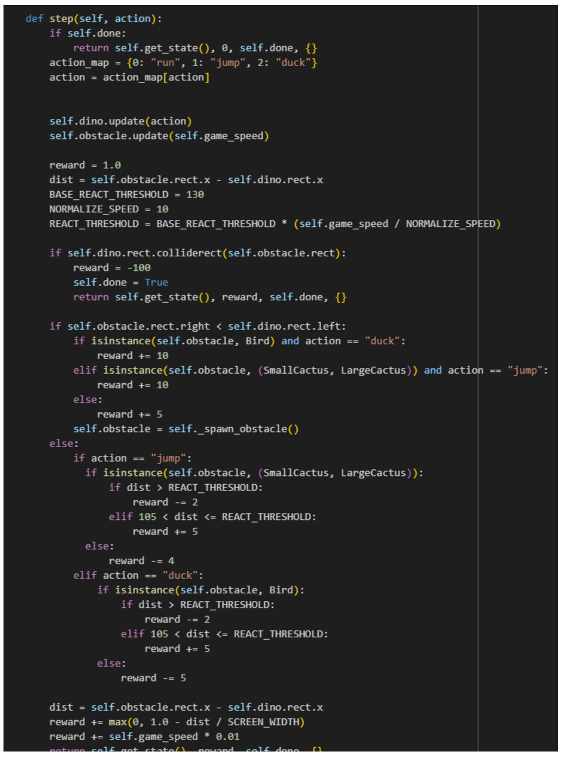
The sensors definition was mainly based on the article from Shekatkar (Shekatkar, n.d.) and consists of the following state components: The agent distance to the next obstacle, the agent y coordinate, the obstacle y coordinate, the obstacle width/size, the current environment speed (how fast the obstacles are coming in the direction of the agent), and the type of obstacle.



```python
def get_state(self):
    dist_to_obstacle = self.obstacle.rect.x - self.dino.rect.x
    dino_y = self.dino.rect.y
    obs_y = self.obstacle.rect.y
    obs_width = self.obstacle.rect.width

    is_small = isinstance(self.obstacle, SmallCactus)
    is_large = isinstance(self.obstacle, LargeCactus)
    is_bird = isinstance(self.obstacle, Bird)

    one_hot = [int(is_small), int(is_large), int(is_bird)]

    return [
        dist_to_obstacle,
        dino_y,
        obs_y,
        obs_width,
        self.game_speed,
        *one_hot
    ]
```

ii. Reward Logic

The reward logic was designed from scratch and had a lot of different versions tested across the models trained. Since this research did not keep track of all of them, this section is an overview of some of the ideas tested for it, along with the final version of it. The initial reward logic was based on simply rewarding the agent if it stayed alive, and giving a major bad reward if it died. That did not work for obvious reasons, it was too simple. So, from that, improvements were added in a way that the agent was rewarded in the scenarios that: it was acting within a good threshold distance between itself and the obstacle (that way it shouldn't jump or crawl unnecessarily), it was crawling for birds and jumping for cactus obstacles, and it was reaching bigger distances (i.e., the environment was getting faster). On the other hand, the agent would be penalized in scenarios where the opposite would happen. After many changes and tests, this was the final reward logic setup (can be found within the Jupiter notebook within the model folder):



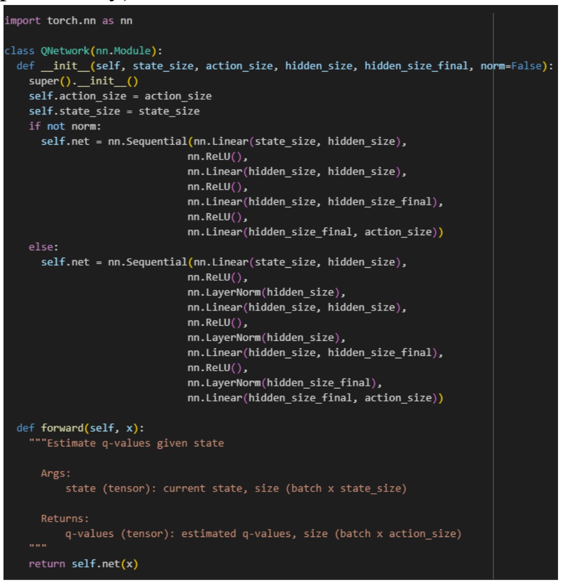
```python
def step(self, action):
    if self.done:
        return self.get_state(), 0, self.done, {}
    action_map = {0: "run", 1: "jump", 2: "duck"}
    action = action_map[action]


    self.dino.update(action)
    self.obstacle.update(self.game_speed)

    reward = 1.0
    dist = self.obstacle.rect.x - self.dino.rect.x
    BASE_REACT_THRESHOLD = 130
    NORMALIZE_SPEED = 10
    REACT_THRESHOLD = BASE_REACT_THRESHOLD * (self.game_speed / NORMALIZE_SPEED)

    if self.dino.rect.colliderect(self.obstacle.rect):
        reward = -100
        self.done = True
        return self.get_state(), reward, self.done, {}

    if self.obstacle.rect.right < self.dino.rect.left:
        if isinstance(self.obstacle, Bird) and action == "duck":
            reward += 10
        elif isinstance(self.obstacle, (SmallCactus, LargeCactus)) and action == "jump":
            reward += 10
        else:
            reward += 5
        self.obstacle = self._spawn_obstacle()
    else:
        if action == "jump":
          if isinstance(self.obstacle, (SmallCactus, LargeCactus)):
              if dist > REACT_THRESHOLD:
                  reward -= 2
              elif 105 < dist <= REACT_THRESHOLD:
                  reward += 5
          else:
              reward -= 4
        elif action == "duck":
            if isinstance(self.obstacle, Bird):
                if dist > REACT_THRESHOLD:
                    reward -= 2
                elif 105 < dist <= REACT_THRESHOLD:
                    reward += 5
            else:
                reward -= 5

    dist = self.obstacle.rect.x - self.dino.rect.x
    reward += max(0, 1.0 - dist / SCREEN_WIDTH)
    reward += self.game_speed * 0.01
    return self.get_state(), reward, self.done, {}
```

c. Q-Learning Network Architectures

This section describes the network architectures used to predict the Q-values based on the current state. Both networks were fully connected structures.

i. Regular MLP

The first ANN trained was a fully connected network with 4 hidden layers. The first 3 layers were defined with the same number of nodes, while the last hidden layer was defined with half the number of nodes. This was thought to be the ideal depth since this task was not supposed to require too much computational power. Furthermore, the decrease in the number of nodes of the last layer was supposed to help the model abstract even more of its data transformations. Once that was defined, three different hidden node amounts were tested for both the first three layers and the last layer: (128, 64), (64,32), (32, 16). Their performance is superficially analyzed in the results section. Finally, as an attempt to get better results, batch normalization transformations were further added to this architecture (can also be found within the Jupyter notebook mentioned previously).



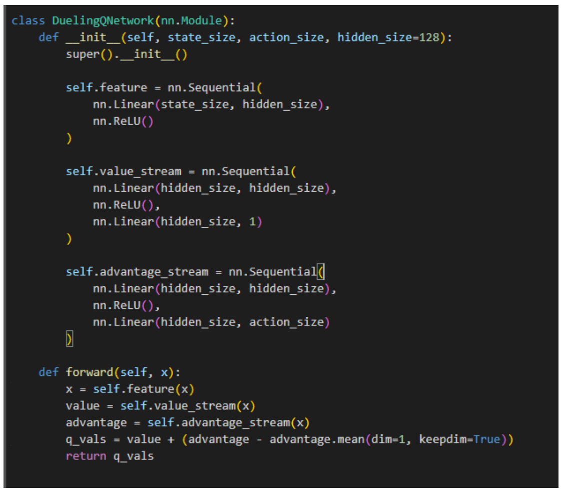
```python
import torch.nn as nn

class QNetwork(nn.Module):
  def __init__(self, state_size, action_size, hidden_size, hidden_size_final, norm=False):
    super().__init__()
    self.action_size = action_size
    self.state_size = state_size
    if not norm:
      self.net = nn.Sequential(nn.Linear(state_size, hidden_size),
                               nn.ReLU(),
                               nn.Linear(hidden_size, hidden_size),
                               nn.ReLU(),
                               nn.Linear(hidden_size, hidden_size_final),
                               nn.ReLU(),
                               nn.Linear(hidden_size_final, action_size))
    else:
      self.net = nn.Sequential(nn.Linear(state_size, hidden_size),
                               nn.ReLU(),
                               nn.LayerNorm(hidden_size),
                               nn.Linear(hidden_size, hidden_size),
                               nn.ReLU(),
                               nn.LayerNorm(hidden_size),
                               nn.Linear(hidden_size, hidden_size_final),
                               nn.ReLU(),
                               nn.LayerNorm(hidden_size_final),
                               nn.Linear(hidden_size_final, action_size))

  def forward(self, x):
    """Estimate q-values given state

      Args:
          state (tensor): current state, size (batch x state_size)

      Returns:
          q-values (tensor): estimated q-values, size (batch x action_size)
    """
    return self.net(x)
```

ii. Dueling MLP

As a final attempt to get better results, this fully connected architecture is commonly used in RL, was implemented and trained. It consists of 3 different shallow networks that form the following forward pass:

```python
x = self.feature(x)
value = self.value_stream(x)
advantage =
self.advantage_stream(x)
q_vals = value + (advantage -
advantage.mean(dim=1,
keepdim=True))
```

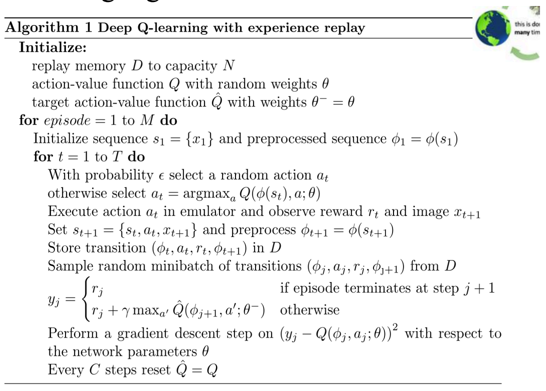
This time, the only hidden size tested was 128.



```python
class DuelingQNetwork(nn.Module):
    def __init__(self, state_size, action_size, hidden_size=128):
        super().__init__()

        self.feature = nn.Sequential(
            nn.Linear(state_size, hidden_size),
            nn.ReLU()
        )

        self.value_stream = nn.Sequential(
            nn.Linear(hidden_size, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, 1)
        )

        self.advantage_stream = nn.Sequential(
            nn.Linear(hidden_size, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, action_size)
        )

    def forward(self, x):
        x = self.feature(x)
        value = self.value_stream(x)
        advantage = self.advantage_stream(x)
        q_vals = value + (advantage - advantage.mean(dim=1, keepdim=True))
        return q_vals
```

d. Training Setup

The training was fully based on the original algorithm of Deep Q-Learning, which consists of backtracking over a sum of squares loss of the model outputted q-values and the reward + the future expected q-value (assuming the maximum q-value actions are chosen). This Bellman function based approach tends to be efficient in game scenarios for RL. Here is a full pseudocode of the training logic used:



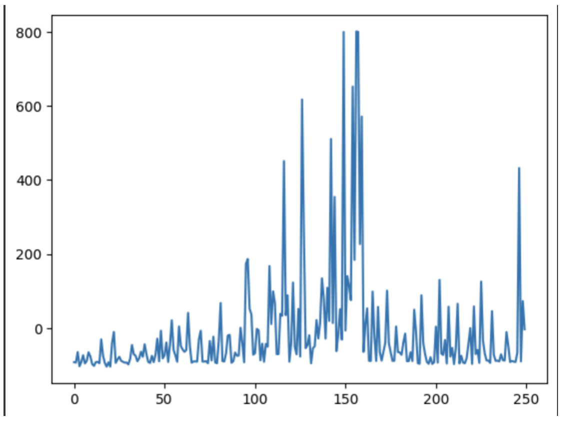
```
Algorithm 1 Deep Q-learning with experience replay
Initialize:
  replay memory $D$ to capacity $N$
  action-value function $Q$ with random weights $\theta$
  target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
for episode $= 1$ to $M$ do
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  for $t = 1$ to $T$ do
    With probability $\epsilon$ select a random action $a_t$
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
    Set $s_{t+1} = \{s_t, a_t, x_{t+1}\}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
    $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to
    the network parameters $\theta$
    Every $C$ steps reset $\hat{Q} = Q$
```

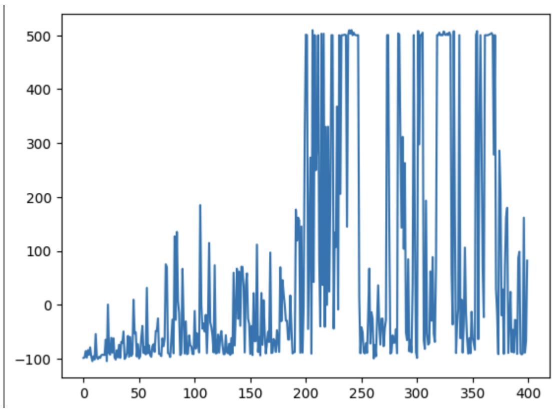
e. Different Hyperparameters Tested

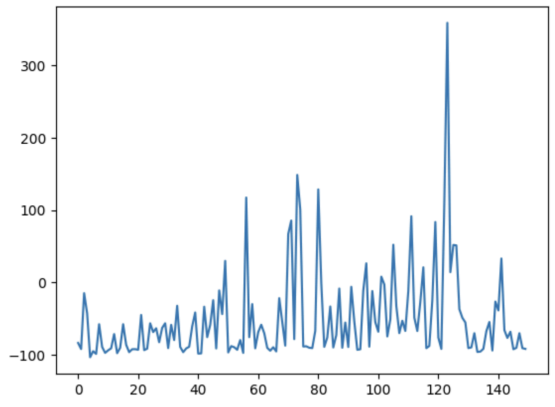
The following are all the variations of hyperparameters tested for this project, within 7 different main tests: hidden nodes amount, final hidden nodes amount, amount of episodes, reward limit for each episode tested. Some other variations were also tested (that are not hyperparameters), e.g. the model architecture, using and not using batch normalization, and different reward logics.

3. Results

Unfortunately, none of the agents trained managed to survive for very long in the game. However, the variations in the reward logic between them led them to individually master certain required action in individual circumstances. I.e., while some agents only learned that they needed to crawl when a bird was approaching, some others only learned that they should jump when a cactus obstacle was approaching. Furthermore, none of the agents managed to present the characteristic reward drastic increase after a certain threshold of epochs, which can be visualized in the following 3 reward over episodes graphs: *for 3 of the models trained







Even though the graph presents a reward increase over time, its really unstable and reinforcement learning algorithms usually present a more steep increase. Maybe training it for more episodes (which was not done due to limitations in terms of computational resources) could have led to better results. However, there is a strong indication that the problem really lies in the reward logic. That indication comes from the fact that the models would converge to some specific behavior really early, and not change throughout the rest of the training (which could explain why the reward was so unstable between episodes).

4. Conclusion

Even though this project did not fully reach its desired results, it still led to valuable insights about how deep Q-learning works. E.g., the results showed the importance of setting up a good reward logic that truly reflects the purpose of the agent and what it should do in different scenarios/states. Finally, this final section also presents some final insights on what else could have been done if there was more time/resources for this research.

a. What Could Have Been Done Differently?

Some of the things that could have been done are: using PPO instead of Q-learning since it is a lot more stable during training and usually leads to better results according to recent research comparing both. Other than that, the Deep Q-learning models could have been CNN's trained with the game screenshots. This could lead to better results since the screenshots contain more information than the plain information given in this research. Finally, parallel training could have been used to speed up training and reach better results due to more models being trained at the same time.

5. Bibliography

Ariyalur, S. R., & Arya, N. (2020). *Chrome Dino Run using Reinforcement Learning*. arXiv. https://arxiv.org/abs/2006.06822

Shekatkar, A. (n.d.). *Using Q-learning to play the Chrome Dinosaur game*. Medium. https://medium.com/@atharvashekatkar1.2/using-q-learning-to-play-the-chrome-dinosaur-game-e836da2bc74a

Rohowsky, M. (n.d.).
chrome-dinosaur [Source code].
GitHub.
https://github.com/MaxRohowsky/c
hrome-dinosaur