# 7066788 Basim Jaaskelainen, 7839078 Riva Pereira, 7847233 Mariah Khalifa

# **Executive Summary**

This report presents the application of reinforcement learning (RL) techniques to the Chrome Dino game, an offline game popular among Google Chrome users. The project's premise is the development and implementation of a Deep Q-Network (DQN) which would be trained and fine-tuned to automate gameplay.

The project's sole success criteria was the successful integration of the DQN model with the Chrome Dino game, in turn creating an environment where the model could learn, adapt, and evolve its gameplay strategies. As the game progresses, the model continuously refines its approach based on the feedback from its interactions with the game.

# **Background Information About the Project**

The Chrome Dino game is a simple and popular game that starts when you're offline on Google Chrome. In this game, you control a dinosaur that needs to jump over cacti and go under obstacles. It's easy to play but gets harder as you go, making it faster and more challenging.

This game is perfect for testing reinforcement learning (RL), a type of AI that learns by trying different things and seeing what works best. As the game gets harder, the AI has to learn and react faster, making it a great way to see how well reinforcement learning can work in games. This project uses the Chrome Dino game to show how AI can learn and get better at games, giving us insights into how reinforcement learning can be used in other gaming applications.

## **Naming Conventions Used**

This Program follows the standard pep8 style guide for python where Classes use Camel Case along with Instances and Methods using Snake-Case. This aids in readability.

## **Dependencies Installed**

### Pytorch

!pip install orch torchvision torchaudio

### Reinforcement Learning

!pip install stable-baselines3[extra] protobuf==3.20.\*

!pip install gym

### Image Processing

!pip install Pillow

### Screen Capturing

!pip install pytesseract

!pip install mss pydirectinput pytesseract

## Imports

from mss import mss

import pydirectinput

import cv2

import numpy as np

import pytesseract

from matplotlib import pyplot as plt

import time

from gym import Env

from gym.spaces import Box, Discrete

from stable\_baselines3 import DQN

from stable\_baselines3.common.monitor import Monitor

from stable\_baselines3.common.vec\_env import DummyVecEnv, VecFrameStack

# **Objectives and Goals**

The primary objective of this project is to develop an artificial intelligence (AI) model which would be able to play the Chrome Dino game, utilizing the capabilities of reinforcement learning (RL). Our focus is on enabling the AI to not only comprehend the game's mechanics but also to enhance its performance progressively, like human learning patterns.

Our approach entails understanding the foundations of RL before and then applying these principles practically in the Chrome Dino game. And the end goal is to achieve a level of automation in the game where the AI operates independently and efficiently.

# **Overview of Reinforcement Learning**

Reinforcement Learning falls under the umbrella of machine learning and is where an agent would learn to make decisions within an environment through trial and error. It is a crucial technique in many domains, like financial modeling, robotics and gaming.

The mechanism for learning is based on the interaction between an agent and the environment defined/used. The agent would perform certain functions/actions and according to predefined conditions would receive a “reward” for this action. The more correct actions performed, the higher the reward.

The fundamental parts of our model (and any Reinforcement Learning model) are:

· **Agent:** The decision-maker/ or learner.

·  **Environment:** The system with which the agent interacts (in this case the chrome browser)

·  **Action:** possible moves or decisions the agent can make (in this case, jumping, crouching or no action)

·  **State:** the current condition of the environment.

·  **Reward:** feedback given to the agent, defined by the user, which is positive for desired outcomes and null for undesired ones.

# **Description of the Reinforcement Learning Model**

Below is a description of the various components and how they are implemented in our model:

·  **Deep Q-Network Implementation:**

DQN was our chosen model architecture due to its effectiveness for handling high-dimensional input spaces. This means it is a ideal model for processing pixels in a scene. It utilizes a convolutional neural network to process the visual input, mapping raw pixel data to action values.

The model learns by updating Q-Value, which provides an estimate of the rewards for each action. Learning involves using a combination of random actions and known actions to maximize the total reward.

·  **Game Environment:**

The environment is set as a simulation of the chrome dino game, using the python mss library and the pydirectinput library for screen capture and keyboard input simulation respectively, and the OpenCV library for image processing. The observation space is a fixed dimension (100 by 83 pixels) and is captured as grayscale images, and the action space consists of 3 actions (0 for space, 1 for down and 2 for no action).

·  **Data Processing and Interaction with the Model:**

The raw data (pixel input) from the screen is resized and converted to grayscale which is the input passed to the DQN model, and this step is critical for allowing the DQN to perform well and reduce the input data complexity. After this, the DQN would predict the best action for each state, and over time the model would learn from the feedback.

The model would be trained by having random episodes of the game, which would be terminated when the dino has crashed into the cactus or the bird, and whenever a action is done which doesn’t result in the dino crashing, the reward is incremented by one.

**Algorithms and Techniques Used**

The DQN algorithm was selected for its efficiency in handling high-dimensional input spaces, such as pixels from a game screen. The model uses a convolutional neural network (CNN) to process the visual input from the game.

**Game Environment Setup**

The game environment, created using Python libraries like mss and pydirectinput, simulates the Chrome Dino game. The observation space is defined as the game screen's pixel data, and the action space includes jumping, ducking, or doing nothing.

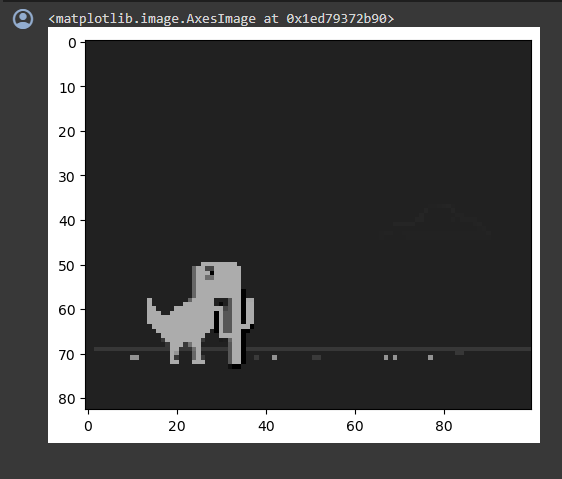
**Data Collection**

Data collection involved capturing pixel data from the game screen. This data was used as input for the DQN model, allowing it to learn and make decisions based on real-time visual information.

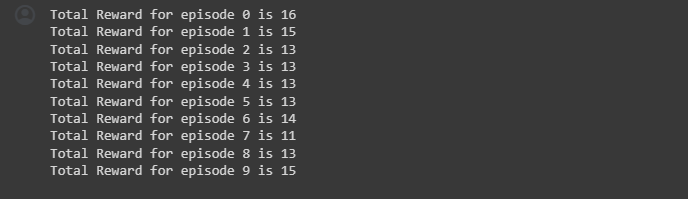
**Model Training Process**

The training process involved running multiple episodes of the game, with the model learning from each run. The model's parameters, such as the buffer size and learning rate, were fine-tuned during this process for optimal performance.

**Screenshots and Explanation of Output**

****

Above is what the CNN would take as input and process. It is a scan of what is on the users screen in a predefined array (game\_location). After the initial scan is taken, it would be converted to grayscale first, then resized and reshaped and stored in the channel variable which would then be returned as part of the defined get\_observation() function.

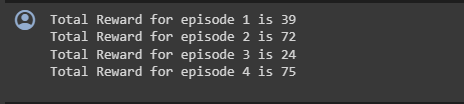


Initially, we decided to test the code for interacting with the environment and the random move, and we displayed the reward for each episode. As you can see, the reward is fairly standard, averaging at around 11-16, as this is before any obstacle has been crossed.



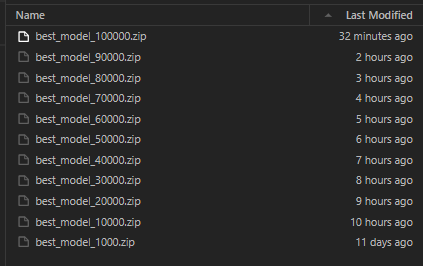
The above is what would be displayed when the Game is over and since the location is constant we can specify the done\_location pixel range at the beginning of the code. The program would constantly be using OCR to recognize if the character G is present in that pixel range, meaning the game is over and the dinosaur has crashed. The library we have used to achieve this is PyTesseract.

We defined a array of possible outputs which we identified after doing multiple test, and we noticed that the letter G was present in the beginning of all of them. So we defined a conditional statement to check if the first letter detected by the PyTesseract.image\_to\_string() function is the letter G, and if so we would return a variable called done for which the value would be set as True. If done is true, the reinforcement learning model would either restart the game or would end the training, depending on the number of steps it has completed up until that point.



After initially training the model and making it execute 30000 times, the above was the rewards for 4 episodes. The maximum score is 75 which is a significant improve from the previous 11-16 reward range. For the final model we aim to train 100 000 times to get a more accurate model.

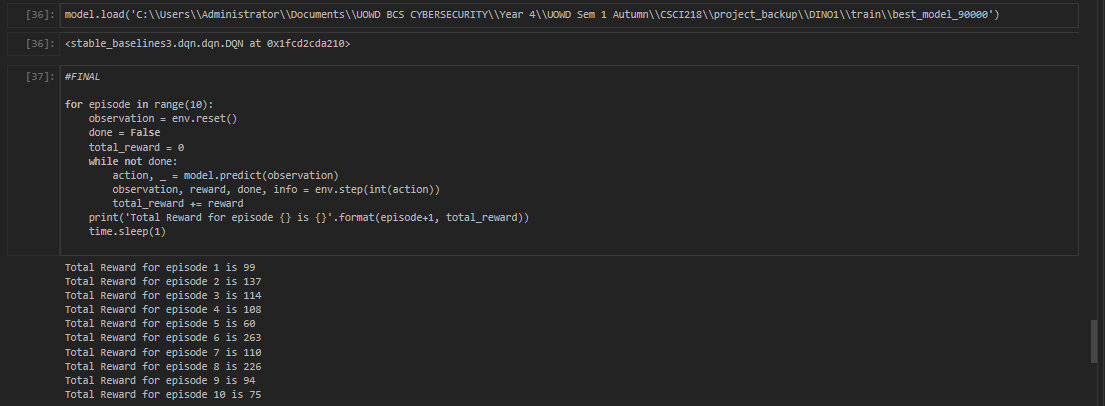
We trained the model 100 000 times and saved a instance of the model every 10000 executions.

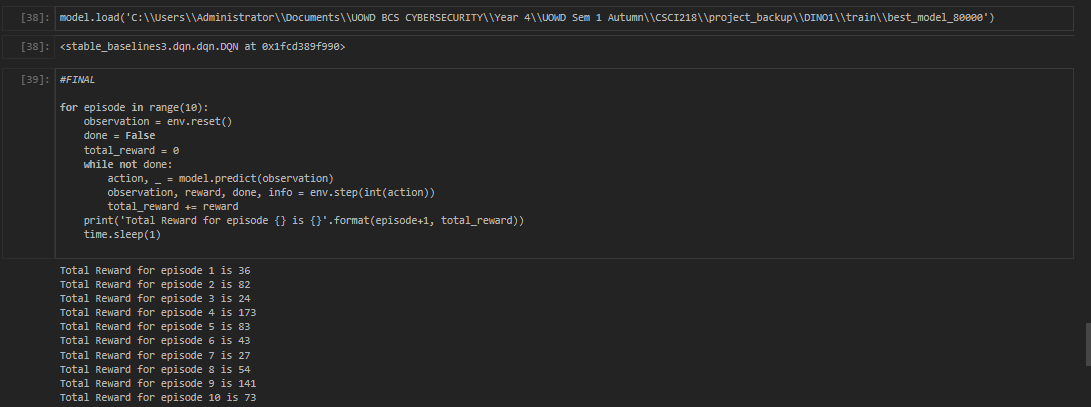


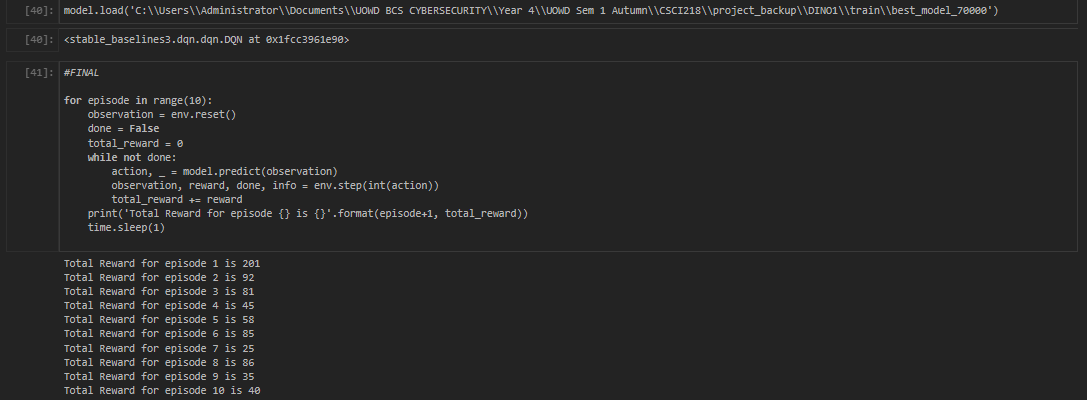
Initially we thought that best\_model\_100000 would be the most effective due to the extensive traintime and the size of data that would be available however the performance was not as expected.

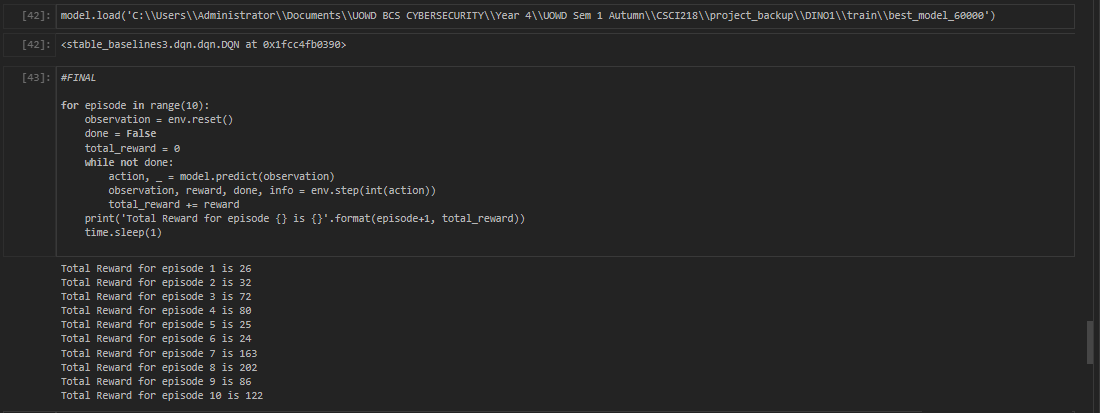
Below are screenshots of 10 trial runs for each of the models and the average score for each.

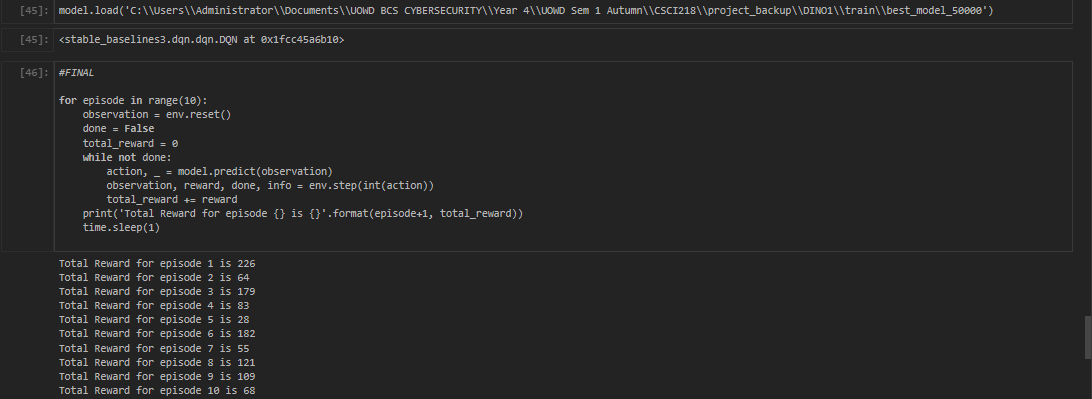
ch

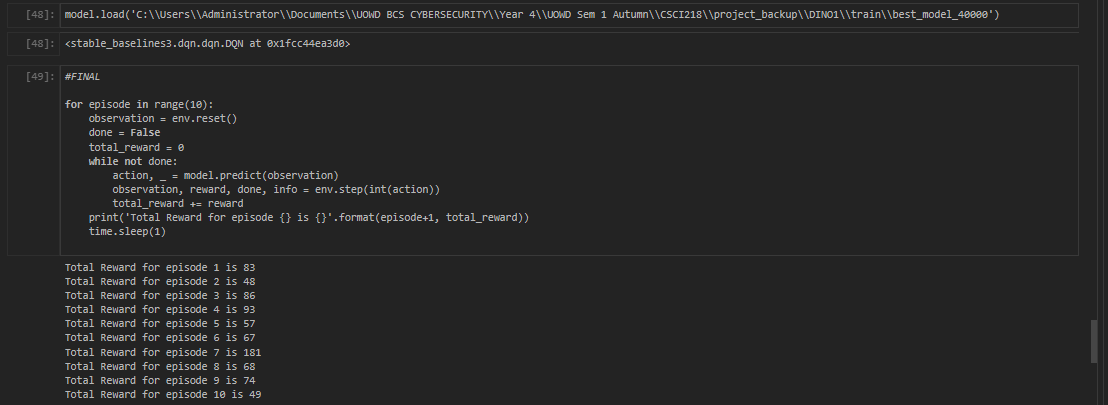


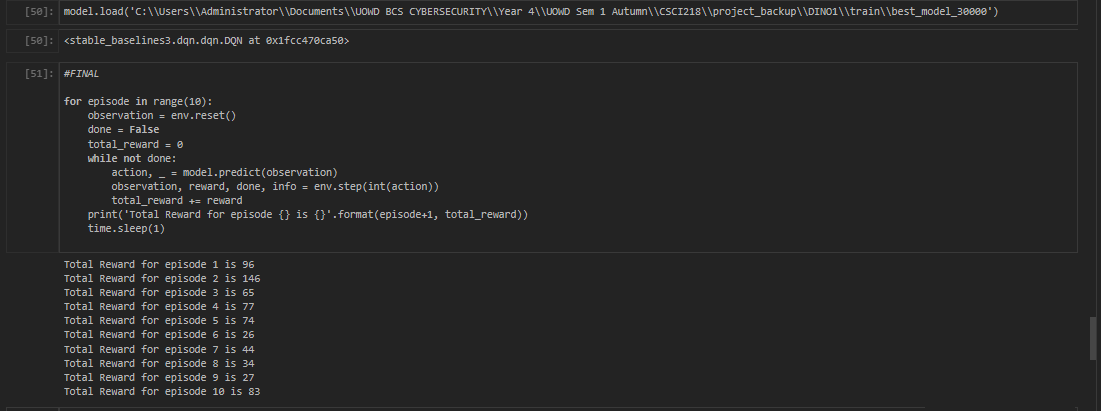


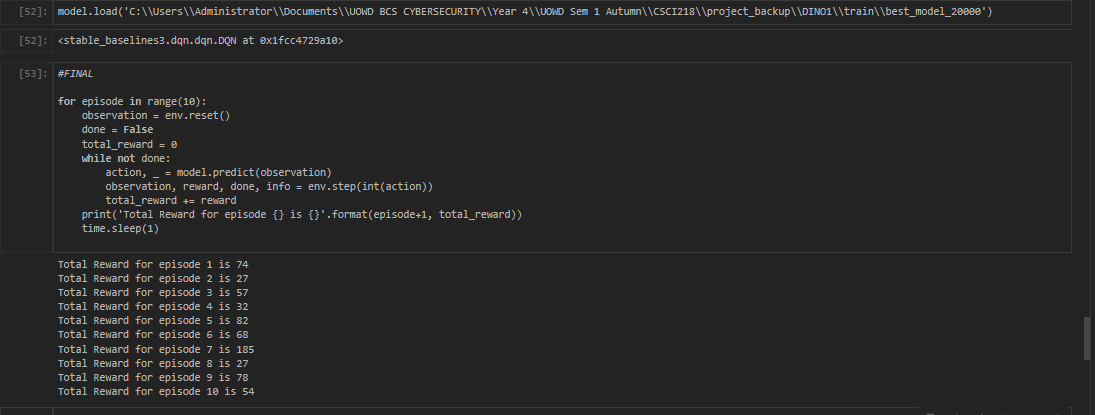


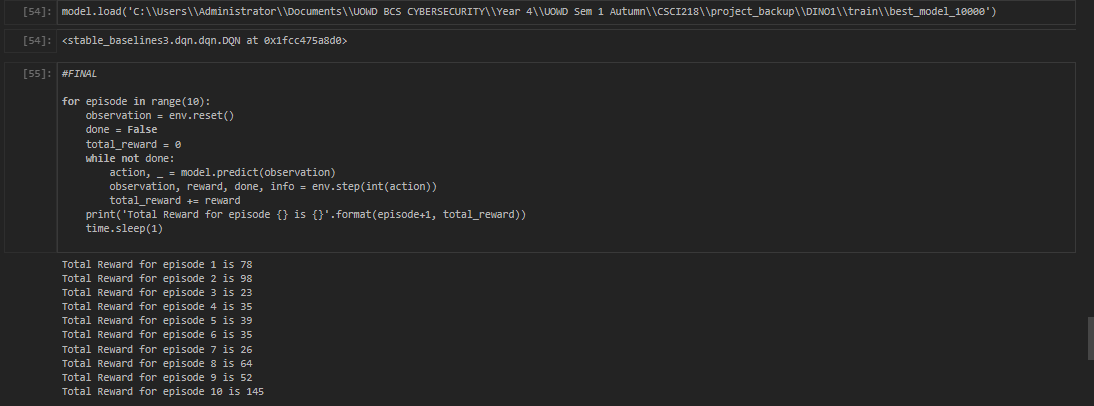












.

**Integration with Chrome Dino Game**

The model was integrated with the actual Chrome Dino game using Python scripting. This allowed real-time interaction between the model and the game, with the model sending commands based on its learned strategy.

**Testing Methodology**

The model's performance was tested by running it on the Chrome Dino game for a set number of episodes. Metrics such as total rewards per episode and survival time were tracked.

**Performance Metrics**

As shown in the testing screenshots, 10 models were generated with the only difference being the number of executions. Starting at 10 000, going up to 100 000 in increments of 10 000, 10 models were created. Below is a table which shows the average score per run and the highest score achieved.

The model with the highest average score and the maximum score was the model90000 with the worst model being the model100000.

| Model Name | Average Score (After 10 Executions) | Maximum Score (After 10 Executions) |
| --- | --- | --- |
| model10000 | 59.5 | 145 |
| model20000 | 68.4 | 185 |
| model30000 | 67.2 | 146 |
| model40000 | 80.6 | 181 |
| model50000 | 111.5 | 226 |
| model60000 | 93.2 | 202 |
| model70000 | 74.8 | 201 |
| model80000 | 73.6 | 176 |
| model90000 | 128.6 | 263 |
| model100000 | 57.9 | 134 |

**Encountered Issues**

Several challenges, such as optimizing the model's reaction time and dealing with varying game speeds, were encountered. Solutions included adjusting the model's learning rate and implementing a more responsive action selection process.

**Limitations**

Such a model takes up computing resources and time in order to train it adequately and has to work within the constraints of processing speed and memory of a computer.

# 

# Conclusion

The project successfully demonstrates the application of reinforcement learning in a game environment. Future work can focus on refining the model for even better performance and exploring its application in more complex gaming scenarios.

# References:

<https://www.diva-portal.org/smash/get/diva2:1713716/FULLTEXT01.pdf>

VIDEO

Hello this is Riva and here is Basim and Mariah.Today, we're going to delve into our project, which revolves around applying reinforcement learning to the Google Dino game

The primary objective of this project is to develop an artificial intelligence model which would be able to play the Chrome Dino game, utilizing the capabilities of reinforcement learning. Our focus is on enabling the AI to not only comprehend the game's mechanics but also to enhance its performance progressively, like human learning patterns.

We’ve imported libraries such as mss, pydirectinput, cv2, pytesseract, gymnasium, and stableBaselines3.

Now that we've provided an overview of our Google Dino reinforcement learning project, let's delve into the intricacies of the game environment.

#Game enviroment

We define the game environment using the WebGame class, which sets up the observation and action spaces. The observation space captures the game screen as an image, while the action space represents three possible actions - jumping, crouching, and doing nothing.

Next, we initialize the screen capture and define the regions of interest within the game window. Functions like step, reset, get\_observation, and get\_done control the game interaction and information retrieval.

\*show screen captures\*

This screen capture determines whether the game should be reset or not by reading the screen.

The next screen capture is the observation space that is the input for the model to predict the next move

#test

This code sets up a virtual environment for a web-based game using the WebGame class. It then runs a test loop for any number of rounds, where the game is reset for each round.

\*play test loop\*

In each round, the game is played by the agent taking random actions until the game ends. The total reward earned by the agent in each round is calculated and printed. This process helps assess how well the agent performs in the game when making random moves. It's like watching the agent play the game without any strategy, just to see how much reward it gets in each try. This simple testing loop gives a quick look at how the agent interacts with the game environment.

#trainlogcallback

**The TrainAndLoggingCallback is a function in our reinforcement learning framework that enhances the robustness and efficiency of our training process. This function saves our reinforcement learning model at regular intervals, creating checkpoints that preserve progress and allow seamless resumption if needed. This strategic saving acts as a safety net, ensuring valuable insights gained during training are safeguarded, contributing to the overall resilience of our learning algorithm.**

**In addition to its role in model preservation, the TrainAndLoggingCallback also systematically logs essential training progress. By recording key metrics and performance indicators at predefined intervals, this callback generates a comprehensive record of our model's evolution. These detailed logs provide a nuanced understanding of training dynamics, enabling us to monitor performance trends, identify potential challenges, and make informed adjustments to our reinforcement learning strategy.**

#model

The reinforcement learning model, specifically a DQN (Deep Q-Network), is initialized using Stable Baselines3 and CNN Policy.

#learn

The model.learn() function is a method used in reinforcement learning libraries to train a reinforcement learning algorithm. It orchestrates the interaction between the learning algorithm and the environment, allowing the agent to learn from experiences, adjust its strategies, and improve its performance based on the provided training data or interactions with the environment.

#load

The model.load() function in reinforcement learning libraries like Stable Baselines3 is used to load a pre-trained model from a specified file path.

Here, we trained the bot overnight for 100,000 timesteps, and saved the training data for every 10,000 timesteps. The optimal training dataset seemed to be the 90,000 one so we’re gonna load that into our model.

#execution

Finally, we execute the trained model in the game environment across multiple episodes. We observe and display the total reward obtained in each episode.

\*stop and start recording\*

\*execute\*

\*stop and start recording\*

This is one of our previous executions

\*play video\*

The project successfully demonstrates the application of reinforcement learning in a game environment. Future work can focus on refining the model for even better performance and exploring its application in more complex gaming scenarios.