Project Report on

CS512 – Artificial Intelligence
Intelligent Agent to Play Chrome Dino Game

Submitted by

Suthram Vinay Kumar - 2020CSM1019

Patel Mit Kumar Rajeshbhai – 2020CSM1016

in Department of Computer Science Engineering, Indian Institute of Technology (IIT Ropar)

Under the Guidance of

Dr. Shashi Shekhar Jha
Assistant Professor, Indian Institute of Technology Ropar (IIT Ropar),
Department of Computer Science Engineering

Abstract

In a game play, there's always a chance of stepping onto a wrong move that can eventually lead to lose the game when it's being played by a gamer and it's always a challenge to the gamer that he has to take only the right moves to successfully finish the game or to score high. Imagine in place of a gamer playing the game, what if the character in game itself plays, what if the game character learns its own mistakes made in the game and tries not to make the same mistakes or wrong steps in next time such that it can score high in the game.

In this project, we developed an intelligent rational agent that performs actions automatically in an Atari game play by making quick decisions and respond accordingly. With the model of convolution neural network, we train our agent to learn from its own feedback and data sets by using deep reinforcement learning which helps the agent to make decisions with trial-and-error approach. There are many variants we can use in deep reinforcement learning, but we mainly use Deep Q-learning algorithm technique in order to perform quality actions. We apply our implementation on an Atari kind play of Chrome Dinosaur game and then we compare our outcomes with some other similar implementations on Atari games and shows that our implementation gives better result in terms of consuming memory and time.

Introduction

- We have implemented the game for agent (the player in game) to play automatically by implementing the deep reinforcement learning models that are, Deep Q-learning and a convolution neural network. So, The player in Dino Game learns how to play using the convolutional neural network model trained with the variant of Deep Q-learning.
- The implementation is developed with the idea of determining the pixels of screen of the game. We initially stores the states of pixels and correlate these states with original pixels of screen which we fetch during the game play. The original pixels will also be converted as states format to compare with the stored states. Based on this comparison, we generate some values and rewards for taking future actions by agent.
- So, the input would be raw pixels(states) and output would be a value function that estimates future rewards, and based on these parameters, the right actions will be performed by the agent.

Introduction

- The figure shows that the Agent is the game player which runs towards right. Obstacle is the thing that tries to block or stop the dinosaur to run.
- For each step or the position of dinosaur in game, there can be a state and some actions need to perform by dinosaur to escape the obstacles. Each step counts an increment of score by 1.
- The dinosaur has to run by escaping these obstacles and score as much as possible.
- There are only 3 actions which can be performed by the agent. We defined them as stay, jump, & down.
- Jump is perform jump over obstacles, down is to bend over the obstacle, and stay is to perform no action.

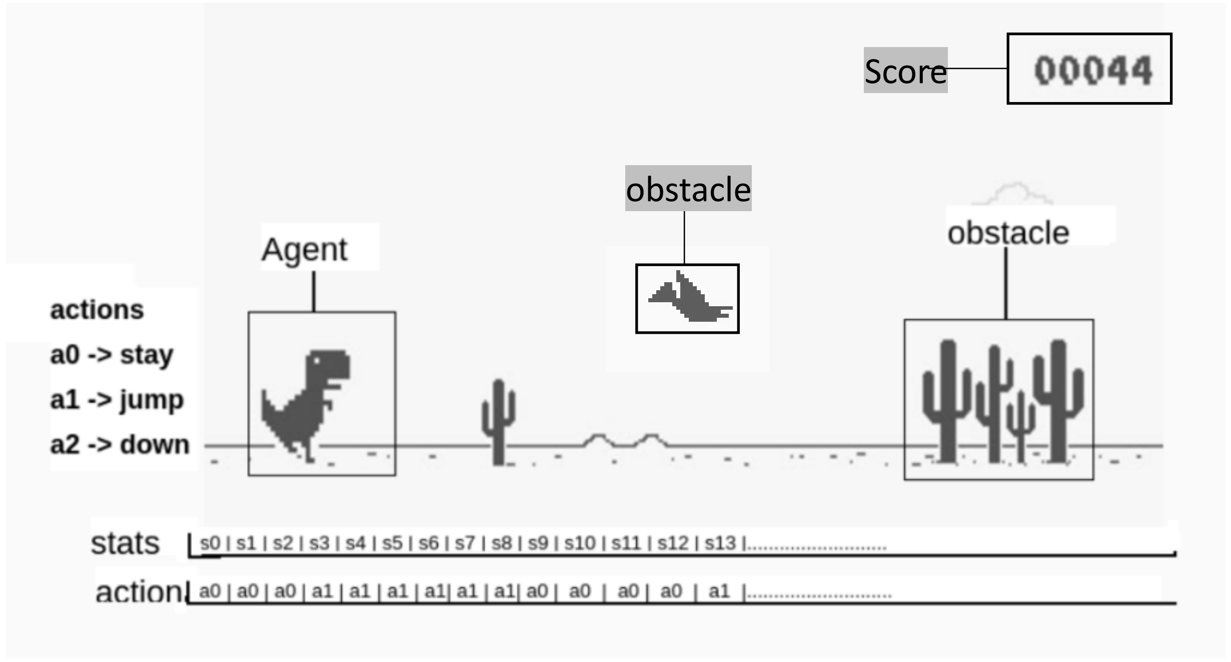
Score
00044
obstacle
obstacle
Agent
actions
a0 -> stay
a1 -> jump
a2 -> down
stats
s0 | s1 | s2 | s3 | s4 | s5 | s6 | s7 | s8 | s9 | s10 | s11 | s12 | s13 |.........................
action
a0 | a0 | a0 | a1 | a1 | a1 | a1| a1 | a1| a0 | a0 | a0 | a0 | a1 |.........................

Fig: The screen of Chrome Dino Game

Introduction

- This single agent has to act as rationally with the actions specified whenever it reaches near to the obstacle.
- How near is determines by the learning phase of our implementation for the agent game play.
- There are two types of obstacles in this game, one is cactus and another is bird. The agent requires only the jump action for cactus obstacle to escape by jumping over it. For the bird obstacle, agent requires either the jump action or down action based on the position of bird. Basically, the down action will be needed if the bird is at upper position and the jump action if the bird is at lower position. The bird in the figure we shown here is at the upper position.
- Remaining all the time, the agent can perform stay action, or sometimes it can perform any of the 3 actions. but the stay action can be most suitable.

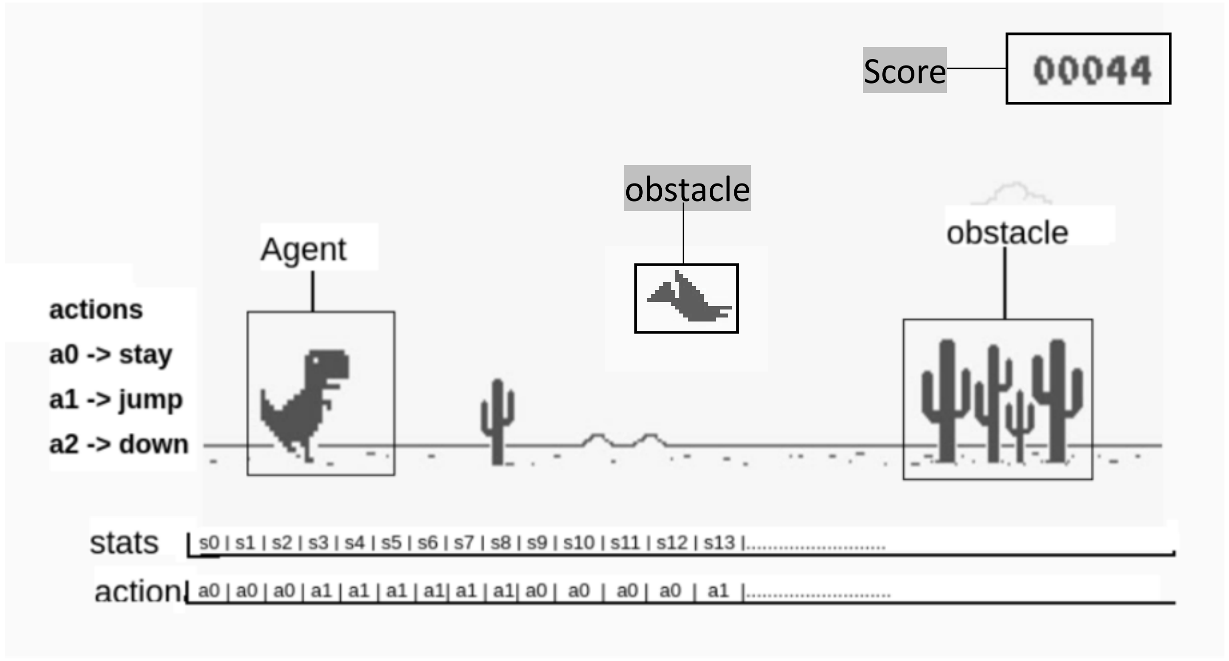
Score
00044
obstacle
obstacle
Agent
actions
a0 -> stay
a1 -> jump
a2 -> down
stats s0 | s1 | s2 | s3 | s4 | s5 | s6 | s7 | s8 | s9 | s10 | s11 | s12 | s13 |..........................
action a0 | a0 | a0 | a1 | a1 | a1 | a1| a1 | a1| a0 | a0 | a0 | a0 | a1 |..........................

Fig: The screen of Chrome Dino Game

Motivations

- There are few similar implementations implemented for the same problem with the use of simple reinforcement models such as Q-learning, markov decision process, etc..
- The problem with these models is memory consumption and time taken for the decisions of actions.
- The memory consumption and time taken by those algorithms are much higher if the states are needed for the agent to learn from a batch of experience stored in the models.

Motivations

- Here in the figure shown at right side, we can see the Q-learning using table structure to store the data of states and actions. The table will give an appropriate Q-value as an output. The disadvantage in Q-learning table is, it will store all the states and all the actions of each state. The memory will become huge if we use this model in our problem as the number of states will always get incremented for every game improvement with its learning.
- In deep Q-learning, we use a neural network to approximate the Q-value function. The state is given as the input and the Q-value of all possible actions is generated as the output. The maximum output of this neural network decides the next action for agent.

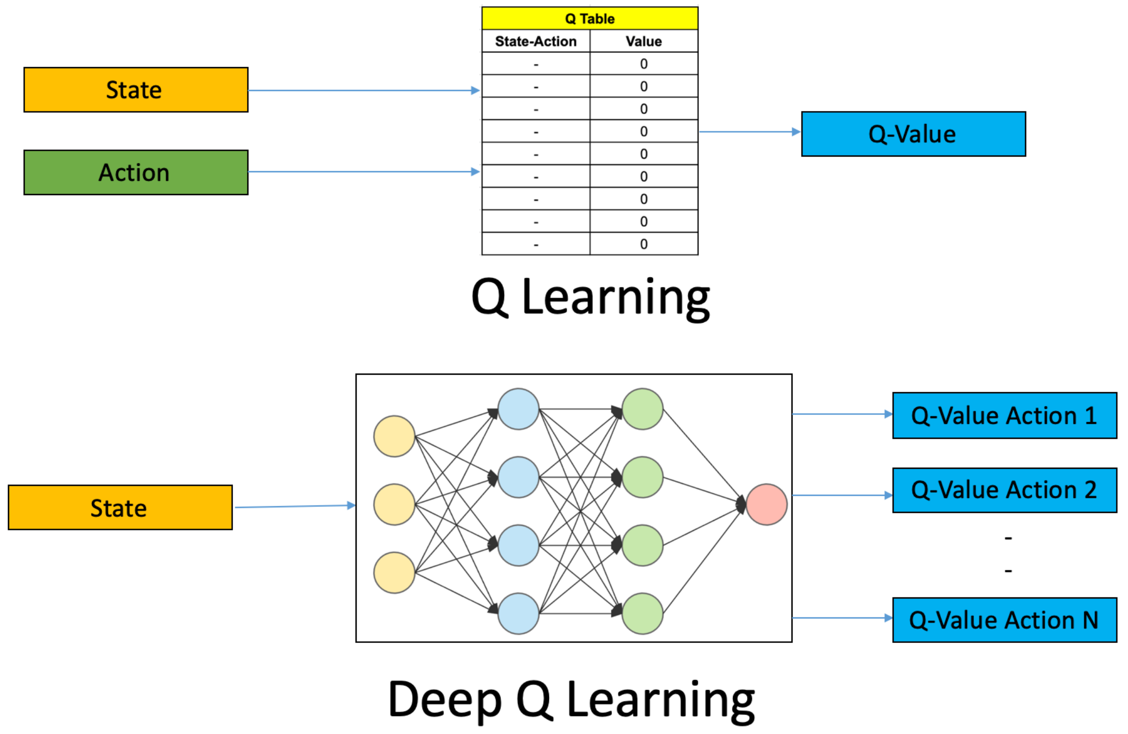

Fig: Deep Q-Leaning vs Q-Learning

Algorithm

- The algo of Q-Learning and Deep Q-Learning is same except the Deep Q-Learning stores the predicted value to perform by the next state.
- We have used the same Q-Learning algo in our implementation but, with the neural network implementation instead of table model.

```
1)  $Initialize\ \boldsymbol{Q}\ \boldsymbol{s}, \boldsymbol{a}\ arbitrarily$
2)  $Repeat\ for\ each\ episode$ :
3)  $Initialize\ \boldsymbol{S}$
4)  $Repeat\ for\ each\ step\ of\ episode$ :
5)  $Choose\ \boldsymbol{a}\ from\ \boldsymbol{s}\ using\ policy\ derived\ from\ \boldsymbol{Q}$
6)  $e.g. \in -greedy$
7)  $Take\ action\ \boldsymbol{a}, observe\ \boldsymbol{r}, \boldsymbol{s'}$
8)  $\boldsymbol{Q}\ (\boldsymbol{s}, \boldsymbol{a})$ <$--$ $\boldsymbol{Q}\ (\boldsymbol{s}, \boldsymbol{a}\ ) + \boldsymbol{\alpha}\ [\ \boldsymbol{r} + \boldsymbol{\gamma}.\ \boldsymbol{max}(\ \boldsymbol{Q}\ (\boldsymbol{s'}\ , \boldsymbol{a'}) - \boldsymbol{Q}\ (\boldsymbol{s},\boldsymbol{a})\ ]$
9)  $\boldsymbol{s}$ <$--$$\boldsymbol{s'}$
10) $until$ s $is$ **terminate**
```

Algo: Q-Learning

Algorithm

- We take screenshots during the game play for every state. We process those images and add appropriate states data by using neural network model once the game gets over.
- In the same way, we fetch the saved data from neural network and predicts the next actions which can be taken by the states.
- So, most of the work in the algorithm revolves around taking screenshots during game play, converting them to appropriate states info and store using the neural network info, then finally predicting the actions for states.

Implementation

- We have implemented the code in total 6 files, main.py, DianosarAgent.py, GameEnv.py, Train.py, Test.py, and Restart.py
- We have also used one .h5 extension file to store all the trained info. The .h5 file is an open source file which can be used for storing large amount of data. It basically stores data in a hierarchical structure. The filename of .h5 extension file is DinoTrain.h5
- main.py is used for giving instructions for the user to give inputs and call the functions accordingly based on the input given by user
- DianosarAgent.py is the file where all the agent game play will happen. Predictions of actions by comparing the current states with neural network model states will be done. Based on the q values it fetches from states saved in, the actions will be taken by the agent.
- GameEnv.py is to set up the environment of screen of chrome dino game.
- Train.py is used to train the agent and saves the trained info into DinoTrain.h5
- Test.py is used to test the game without saving the gameplay in DinoTrain.h5
- Restart.py is used for restarting the game automatically by processing the image

Implementation

- To detect the game is over and need to be restarted automatically, we uses a value to track the symbol of restart which appears on the screen when the game gets over.
- As the different screens have different sizes and different pixels, the value of restart is not same with every screen.
- We first initialized the value of restart with some random number manually, then we are finding the correct value of restart after one game is over. We make changes of that value in GameEnv.py file at line no.125
- This can be one limitation in our program. Every time we have to set this restart value whenever we execute it in different screen.

Results and Discussions

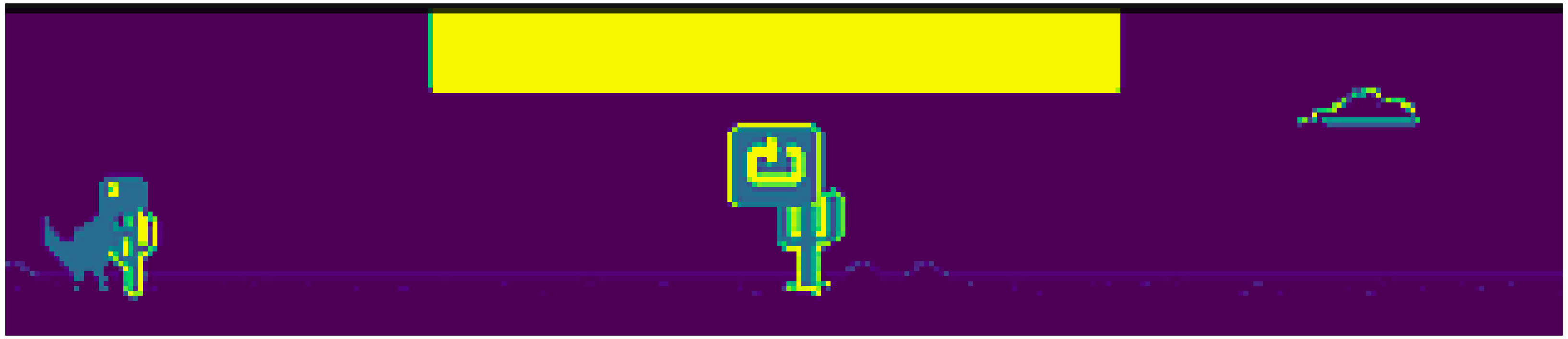

Fig: Screenshot taken by program

- This is how our program takes screenshot and this image is the plotted image we are outputting using plot image functions
- The figure we shown here is the screenshot taken by our program and it’s from the moment of game over. We can see the restart symbol in the middle of picture.
- The values to track screen gets changed according to the screen moments. The value gets changed when the screen will be changed with the restart symbol on screen. We fetch that value and fixes it in the program for the screen which is being executed with our program.
- After fixing the value of restart, we don’t require to restart the game manually anymore. The game will always get restarted automatically.

Results and Discussions

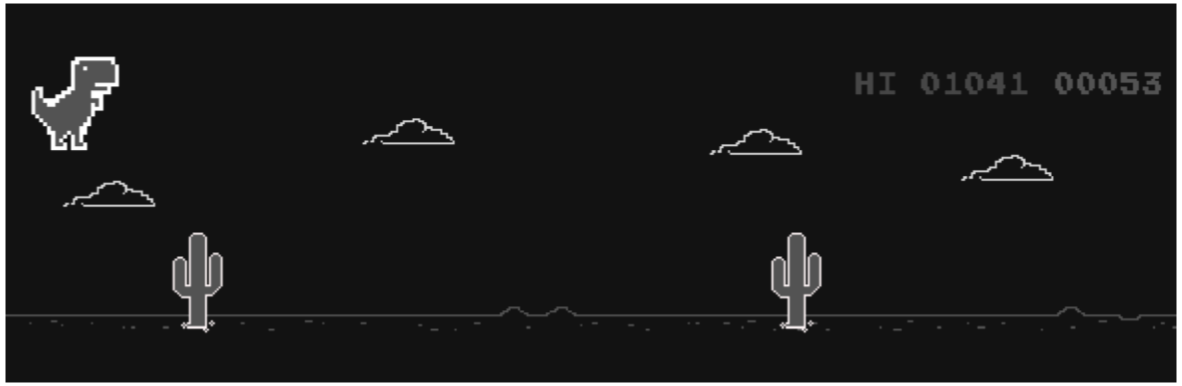

Fig: Jump action

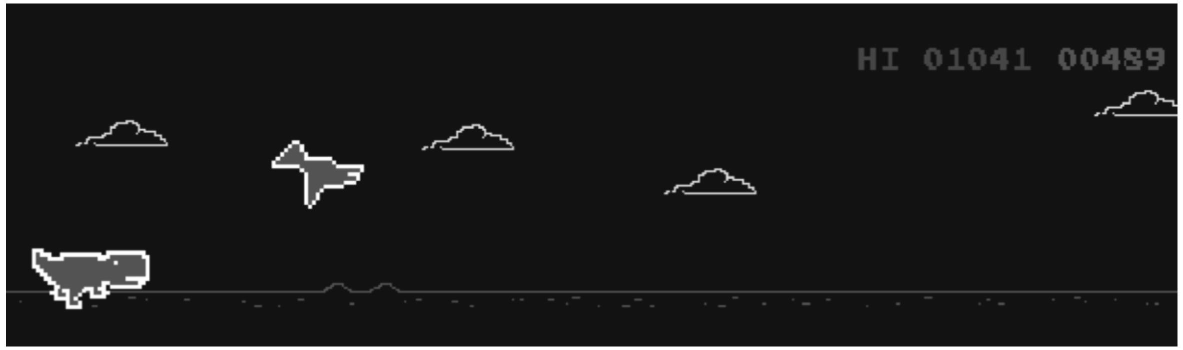

Fig: Down action

- The above two figures are the sample screenshots taken by us during the game play playing by the agent which is dinosaur in the game.
- You can see in the left side figure that the jump action has taken by our agent to jump over or escape the cactus obstacle. Similarly in the right side figure, the agent has taken down action to bend over or escape the bird obstacle.

Results and Discussions

- We have trained our model up to 1 hour and before that also we have trained the model quite a many times for 10 to 20 minutes during the implementation of codes. Even after running these many times, the DinoTrain.h5 file is still just a 2MB file.
- Suppose if we have used the traditional simple reinforcement approaches, the file would have became much larger size after running the game these many times.
- The main advantage here in our implementation is, we have used convolution neural network unlike a table entry storage in Q-learning.
- Searching in the small file is lesser time than the searching in large file, so I can say Deep Q-learning is much better than the simple reinforcement models either in terms of memory consumption or in terms of time consumption.

Results and Discussions

- We have analyzed the performance of our implementation by comparing the different models trained.
- The figure at right side shows a comparison between less time trained models and more time trained models.
- On x-axis we have number of games and on y-axis we have scores we got for the particular game.
- Three colored lines shows 3 different models, one with the model is not trained, one with model trained for 30 minutes, and one with the 1 hour model trained.

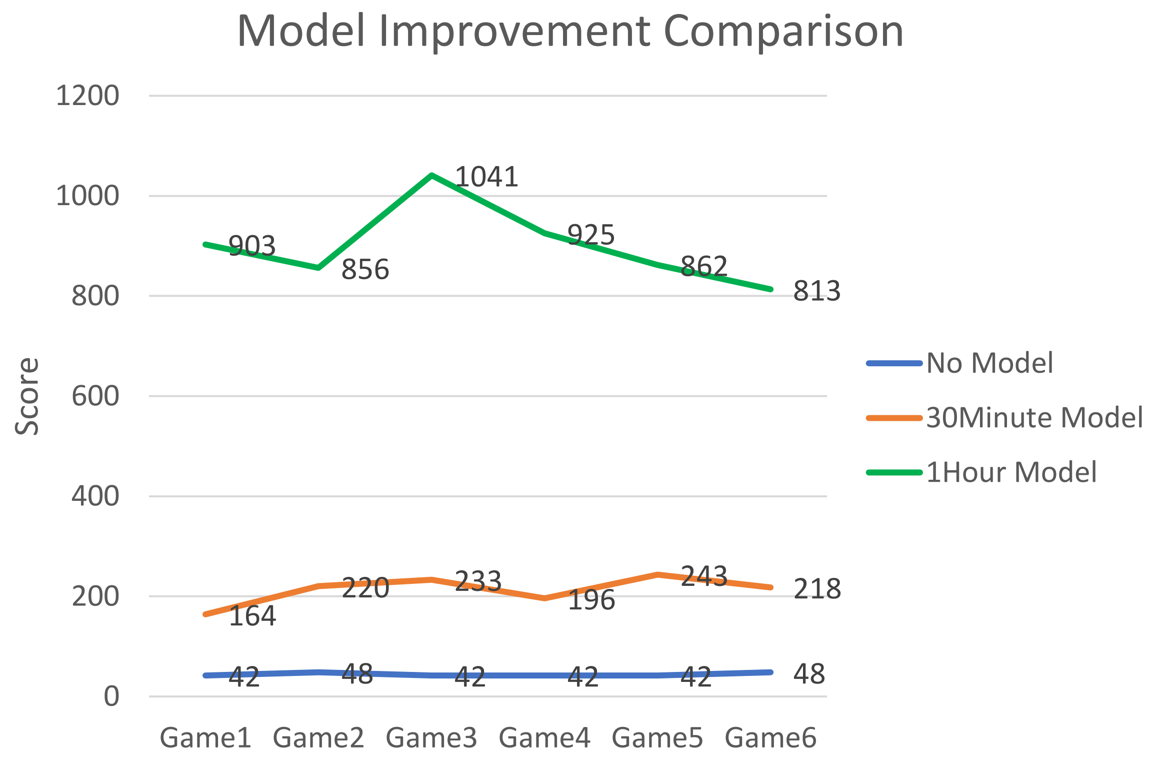

Fig: Graph of model improvement

Results and Discussions

- With the no model, as the agent hardly passes one or two obstacles, the average score we got is just only around 40
- With the 30 minutes model, the agent was able to get around 200. The 1 hour model was able to get up to the average of 900 score.
- The improvement percentage we got for these 30 minutes interval is around 450% to 500%. This can give a very big score if the model is trained more.

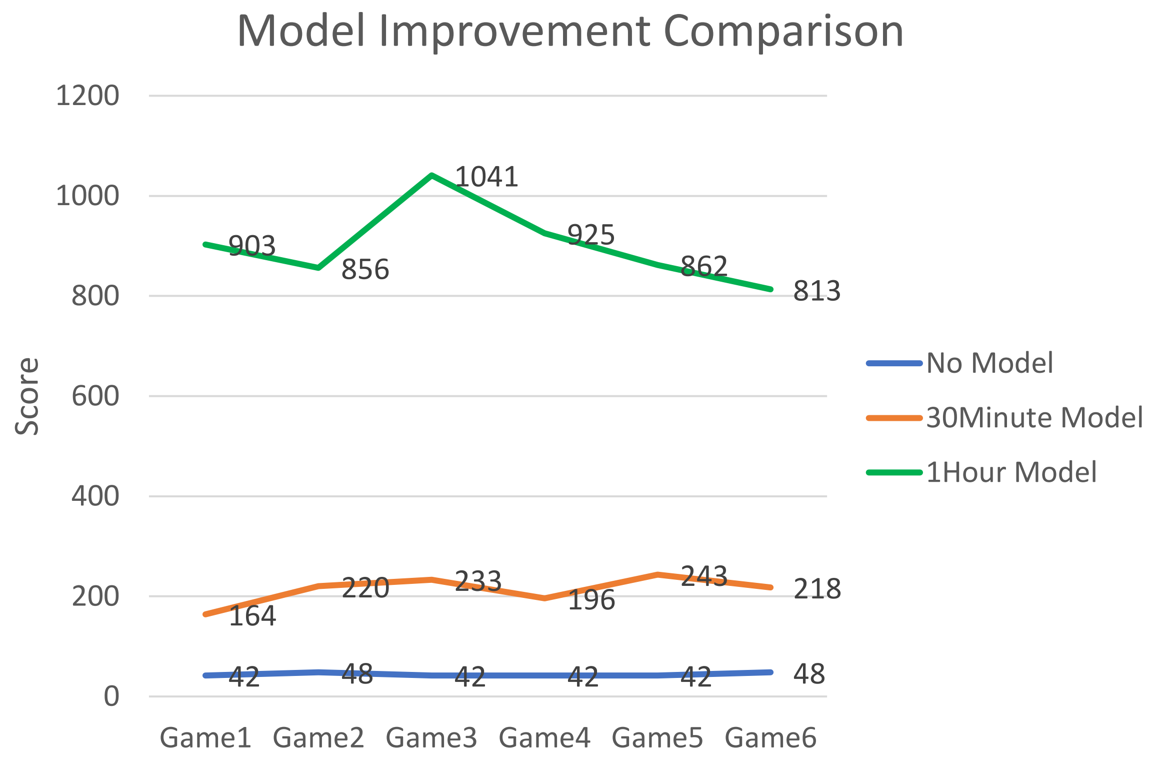

Fig: Graph of model improvement

Conclusion and Future Work

- We have trained the model in our personal systems and during the training/learning phase, the game play is utilizing CPU almost 100%. Utilizing CPU 100% for large amount of time can damage our systems.
- So, as we didn’t have enough resources to train our model for a good amount of time, We have trained our model only up to 1 hour, but we were still able to achieve 1041 as the highest score which is a very decent performance.
- If we were able to train our model more than 10 or 12 hours. We could definitely get even much more greater score and the game play also much consistent in getting higher scores.
- The comparison of our outcomes with the simple reinforcement approaches is made by assumptions and the understanding of concepts learned during our implementation.
- Future works or enhancements can be looked into improving methods to detect automatic restart and also getting higher scores with less time training.

References

- We have referred some YouTube lectures to understand the concepts and looked into some research papers and GitHub codes on how to start and approach our implementation. Below are the links that we have referred.
- Research Paper : A Hands-On Introduction to Deep Q-Learning using OpenAI Gym in Python
- Research Paper : Deep Reinforcement Learning for General Game Playing
- https://www.youtube.com/watch?v=gCJyVX98KJ4
- https://www.youtube.com/watch?v=hCeJeq8U0lo
- https://www.youtube.com/watch?v=MasxAN-xZIU
- https://github.com/IshmaelObeso/DinosaurGame-DeepQ
- https://github.com/vdutor/TF-rex
- https://github.com/SaralTayal123/ChromeDinoAI
- https://github.com/bensawyers/FinalYearProject