INTERACTIVE CONSTRUCTION OF REWARD FUNCTION

A Project report submitted in partial fulfillment of the requirements for the award of the degree of

INTEGRATED MASTERS OF TECHNOLOGY

in

COMPUTER SCIENCE AND ENGINEERING

by

Advait Lonkar

Register No: IMT2017002

Semester: VII

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY, BANGALORE

BANGALORE - 560100

NOVEMBER 2020

TABLE OF CONTENTS

List of Figures

Chapter 1

Introduction

Reinforcement Learning is a machine learning concept that involves how AI agents perform some actions in an environment. It is one of the three main paradigms of machine learning along with: supervised learning and unsupervised learning. This project involves the concept of reinforcement learning through an AI agent which plays the game *Super Mario Bros*. This AI agent will be trained and tested on the *Super Mario Bros* game environment and its performance scrutinized and enforced through some rules. These rules are the *enforcers* for the reinforcement learning to take place.

1.1 Reward Functions

Reward functions are commonly used for reinforcement learning models to help them arrive at results or conclusions. In a gaming environment, an AI agent is capable of training on the game-play system to reap the maximum specified reward. For example, the AI agent in *Super Mario Bros* will have the ability to move the character, then the reward function would be the negative of the distance between the starting and finishing point of the character in that specific game level. In addition to that, the AI agent will be rewarded more if it collects coins during its movement to the finishing point. It can be observed in this case that, the reward function can be defined as a mathematical construct dependent on the game level design and the reward system of the game. [1]

1.2 Shortcomings of Reward Functions

The reward function setup in a reinforcement learning model can break in many ways. This could be attributed to the ingenuity of the AI agent to manipulate the game mechanics in a way which is undesirable but maximizes the reward function. This could also be due to wrongly specified reward functions which would give the AI agent incentive to maximize the reward function while disregarding the otherwise measure of success that might be expected . This is a common theme with reinforcement learning models where it is often very difficult to specify the reward functions in a way that would make the AI agents do exactly what we want. But more often than not, this may lead to undesirable behaviours from the AI agents. [2]

1.3 Report Outline

The report is organized as follows:

1. Literature Review of some of the resources used for this project.
2. The Mario Environment description that was chosen for this project.
3. Development experience of the DQN Agent for Mario.
4. Final results showing the interactive construction of reward functions.

Chapter 2

Literature Review

This chapter will contain the reviews for the various academic papers, blogs, articles and other resources explored and studied to aid with the development of this project.

2.1 Understanding the role of reward functions in reinforcement learning. [1]

This topic forms the basis of the project. That is to understand how important reward functions are for a particular reinforcement learning model. This article explains the importance of the reward function very succinctly and paints a picture of why a reward function is necessary as a judgement for a reinforcement learning model. It also provides the relevant example of Deep Mind's AlphaGo as the recent development (at the time of publication of the article) in this particular niche area.
The author then provokes the readers thoughts on a meta-level by stating that whether or not machine learning can replace human decision-making.

2.2 Faulty reward functions. [2]

This topic forms the problem statement for this project. This blog article very eloquently describes how a reward function design can manipulate the AI behaviours which maybe deemed undesirable for us.
This article goes through a game situation in a game called *CoastRunners* which is a boat racing game. It describes how the AI agent got manipulative in its behaviour and instead of finishing the race, it urged to collect more and more targets laid out on the map.
This behaviour is a result of poor or negligent design of the reward function metric and this article describes it very well.

2.3 Q-Learning [3]

This topic was the backbone of this project. The major reference used for this topic was tensorflow's official website's learning resources.

2.4 Mario Maker [4]

This project was found on github as an open source implementation of the game Mario.

2.5 ReimproveJS [5]

This open source library has the implementation of a dqn agent. This library's functionalities were used for the development of this project.

Chapter 3

Mario Environment

In this chapter, I will describe the the Mario game, its mechanics and its implementation that was available as the code-base to build this project on.

3.1 Mario-maker

3.1.1 History of the game

Mario is a game series created and developed by Nintendo featuring the character *Mario*. Alternatively called *Super Mario* or *Super Mario Bros* (which feature his brother Luigi) was released by Nintendo Entertainment System (NES) in 1985. It is regarded as the first side-scrolling 2D game. The main objective of the original game is to find and rescue the princess Peach from the dragon's castle. But to do that, Mario (and/or Luigi) have to go through a barrage of complex levels, crossing obstacles, eliminating enemies and reaching the castles. The multiple game series involving Mario have had 8 levels or more in their rendition. The first game had 8 worlds with 4 levels each, making a total of 32 levels. This game is widely regarded as one of the best-selling video games of all time. The Super Mario game series has had great critical as well as audience reception and considered one of the best game series ever. In fact, the game is so popular that every single new gaming console released has had one or the other game from the Super Mario series made for it.

3.1.2 Why do AI researchers like Super Mario?

This game is one of the best games ever made. Millions of people have happily spent hours of their life navigating obstacles, smashing *goombas* and rescuing the princess for decades.

However in recent years, for some people, this iconic game has become a testing facility for artificial intelligence (AI) researchers. This particular game has had countless numbers of AIs made for it to train on the game environment and to perfect the game-play needed to "beat" the game. Games are, in general, great for developing AIs since games usually require logical skills, situational awareness and decision-making skills to play and create levels.
Developing AIs for games like Mario is a means of exploring how the best levels could be created and played, and how one day machines will help humans design games. Mario in particular is so intriguing to researchers because of its simplicity and the complex nuances that come with it. The algorithms needed to perfect the game-play are quite interesting but not impossible to solve.

3.1.3 Selecting the Mario environment

The first task for this project was to find the mario environment required to develop an AI for it. An extensive search was done to find the game environment with an available code which was open-source. Finally, the environment that I selected was called **Mario-maker** and was found on github developed by the user *pratishshr*. The splash-screen for the game environment looks like shown in figure 3.1.

Figure 3.1: Splash screen for the Mario-maker environment.

This game has two modes denoted by the two buttons on the splash-screen: *Start Game* to play the game as Mario and the *Level Editor* to create levels for Mario to play. This project is only concerned with the playing game part of **Mario-maker**.

3.2 Mario-maker Mechanics

Mario-maker has 5 built-in levels for Mario to navigate and complete the game. The levels start with just the character Mario on the ground and certain elements visible on the screen. As the Mario progresses, the screens scrolls forward and he discovers other elements which include coinblocks, *goombas* (enemies), pipes, powerups and the level-ending flag. The game starts in the state as shown in the figure 3.2.

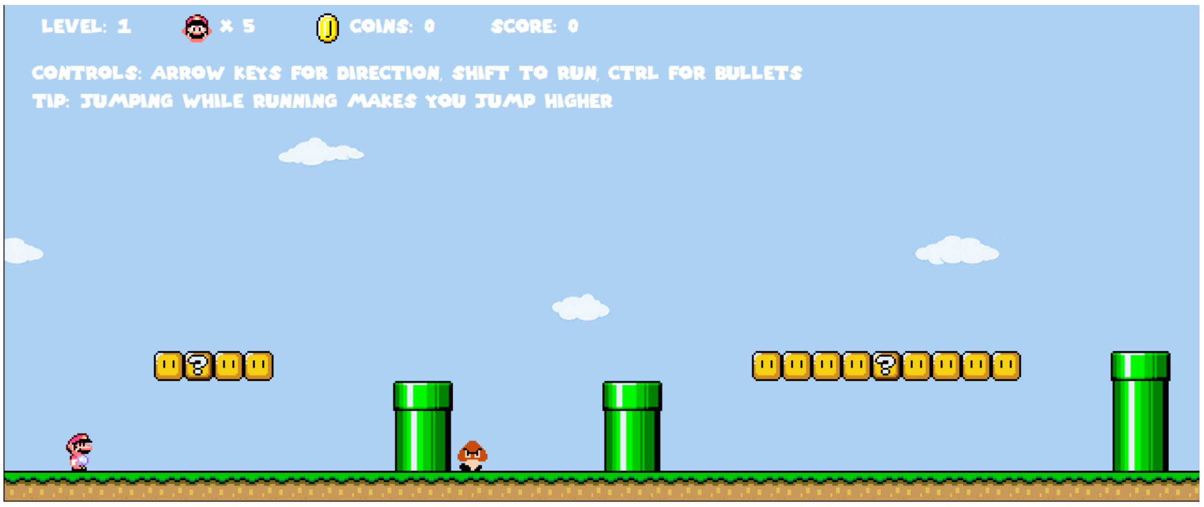

LEVEL: 1
× 5
COINS: 0
SCORE: 0
CONTROLS: ARROW KEYS FOR DIRECTION, SHIFT TO RUN, CTRL FOR BULLETS
TIP: JUMPING WHILE RUNNING MAKES YOU JUMP HIGHER

Figure 3.2: Starting point of the game.

At the top of the start screen, there is an information panel. It shows the level at which Mario is currently at, the number of lives Mario currently has before he dies, the number of coins collected in the course of the gameplay and the last item being the current score of Mario.

3.2.1 Elements in Mario-maker

The main objective of this game is to make progress through the various levels, collecting coins, killing goombas and building up the score without dying. So let's see a brief description of all the elements involved in any level that Mario can encounter -

1. **Coins:**
   Coins are collectable items which Mario gets when it jumps and hits a coin-block. The coin score is displayed at the top of the game screen as long as Mario is playing. The collected coins also contribute towards the overall score on the game screen, also displayed at the top.

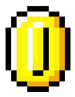


Figure 3.3: The coin item

2. **Goombas:**
   Goombas or the enemies are the ones which Mario has to kill by jumping on top of them. If a Mario touches a *goomba* from any other side, then it dies if he is in his smallest state, or becomes small if he is in his bigger state. Killing a *goomba* also counts towards the overall score.

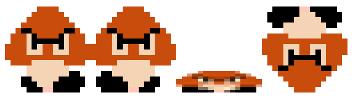


Figure 3.4: The *Goombas*

3. **Elements:**
   These are the building blocks of the level. As shown in the figure below, from the left these are - ground-block, coin-block, powerup-block, normal block, flag-post, flag, pipes and mystery block.

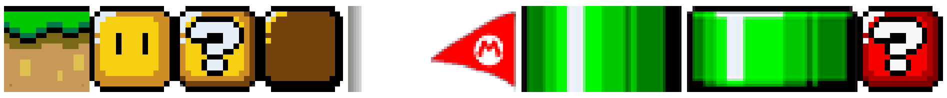


Figure 3.5: The Elements

4. **Flag:**
   The flag always appears at the end of any level on top of a flag post. Mario has to jump as high as possible on the flag-post to receive the maximum score.

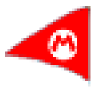


Figure 3.6: The Flag

5. **Mario:**
   Mario is the main playable character in this game. He starts off as a small Mario and turns bigger if he touches any powerup. If Mario touches any *goomba* in its small state then he dies, otherwise he becomes small. The figure below shows the various sprites of Mario used to animate his movements in his small and big form.

Figure 3.7: Mario sprites

6. **Powerups:**
   There are two types of powerups in this game. One is the mushroom which makes the smaller Mario big but has no effect on the bigger Mario, while the other one is a flower which gives Mario the ability to shoot bullets at *goombas*.

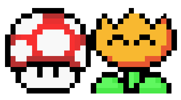


Figure 3.8: The Powerups - Mushroom and Flower

3.3 Mario-maker Implementation

The game logic was written in Javascript and hosted online as github pages. The animations were done on the HTML DOM canvas object. Shown below is the directory structure of the project. Only some of the directories' content is shown in an expanded form because of their importance towards my project.

```
/
├── css
│   ├── style.css
│   └── reset.css
├── fonts
├── images
├── js
│   ├── levelEditor
│   ├── mainGame
│   │   ├── Bullet.js
│   │   ├── Element.js
│   │   ├── Enemy.js
│   │   ├── GameSound.js
│   │   ├── Mario.js
│   │   ├── Powerup.js
│   │   ├── Score.js
│   │   └── MarioGame.js
│   ├── GameUI.js
│   ├── MarioMaker.js
│   ├── Preloader.js
│   └── View.js
├── sounds
├── README.md
└── index.html
```

So, let us look at the implementation details of some of the files involved.

3.3.1 The *js* directory:

This directory contains the main source code for all the game-logic and the animations for the Mario game written in JavaScript. So let us look at its contents -

The *levelEditor* directory:

This directory is concerned with the level editor part of **Mario-maker**, so I did not interfere with this directory at all.

GameUI.js:

- This file contains the implementation of the GameUI. GameUI has the HTML DOM canvas object on which the whole game is drawn, updated and animated.
- It contains methods to update the dimensions of the canvas. It contains a context of the canvas which is an API for the canvas object to get its context to draw anything on it.
- It contains methods to scroll the canvas and a generic draw method which would be used by different elements to draw themselves.
- And the GameUI is running as soon as the browser is loaded, so an instance of the GameUI is always active which could be used elsewhere in the code for other purposes.

View.js:

- This file is mainly concerned about arranging the various components of the game.
- It contains methods to create new HTML DOM elements to add onto the screen to display various things.
- It also contains methods to create a hierarchy in the HTML DOM elements so that parent and child elements could be created effectively through JavaScript without making too many visits to the HTML code.
- And like GameUI, a View instance is always available to be used elsewhere in the code to display the animations and the related information.

The *mainGame* directory:

This directory contains the playable Mario game logic and animation implementation. An object-oriented approach was used to implement the Mario game without using real classes in JavaScript. The implementation was quite interesting to explore, since it was treating functions like how classes are treated in other object-oriented programming languages. So let us the look at the content of the core game directory. Following are the JavaScript files that were implemented for the Mario game -

1. *Bullet.js*
   - This file has the implementation for the bullet that Mario can fire once it gets the flower powerup.
   - The bullet contains methods to draw and update its position and velocity considering gravity.
   - This uses a GameUI instance to draw the bullet element.
2. *Element.js*
   - This file has the implementation for all the elements to make the map for Mario.
   - It contains methods to draw and update positions for platform, coinbox, powerup-box, flag and pipes.
   - This too uses GameUI instance to draw the respective elements.
3. *Enemy.js*
   - All the implementation is similar to the elements above, the only thing different is the logic of its death by Mario.
4. *Mario.js*
   - For Mario, there are checks for its type - big or small.
   - It also contains the update and draw methods using GameUI instance similar to the other elements.
5. *Score.js*
   - This uses a View instance to create the top panel on the game-screen to display all the different scores.
   - This contains methods for updating the total score, coin score, life-count and level number.

6. ***MarioGame.js***
   - This file contains all the game-logic coming together from all the different elements discussed so far.
   - All the initialization for all the elements in an *init* method in this file. Mario, the level map and the score are initialized before the game even begins.
   - It contains the key-bindings that allow the player to control Mario's movements. The arrow keys are used for direction, the shift key in combination with the arrow keys is used for speed and the control key is used to fire bullets.
   - Then comes the main game-loop called ***startGame***. This uses the GameUI instance to render the level map. Draw and update methods for all the elements are called in this loop. The key-presses are checked and then different collision-checks are done on Mario with respect to different elements to update Mario's state. Finally, Mario is updates and the control is transferred to the beginning of the loop again.

***MarioMaker.js*:**

- This file brings together the whole packaged game. It puts together the Mario game with the level editor using the GameUI and View instances.
- MarioMaker is the main file of the whole project which has its own instance running as soon as the browser is loaded.
- This also contains the level design of 5 levels in the form of JSON object.
- The MarioGame instance created here uses one of the 5 levels depending on the progress of the Mario and starts the main game-loop.

Chapter 4

Developing the DQN Agent

A lot of machine learning models in this world have a connection between inputs and outputs that does not change with time. The models are trained on certain datasets and tested on similar datasets to make predictions. However, in real world, the models make a decision which may have an impact on the surroundings and hence the model might need to update itself to make better prediction in the next iteration. The time-variance in the inputs of the model come into play more often than not and this is where reinforcement learning comes into play.

In reinforcement learning, the model, also known as the *agent*, interacts with the *environment* and performs an *action* from a possible set in each *state* of the environment to gain positive or negative *rewards*. The rewards are an indication of whether the action taken was good or bad. Combining all these factors, the agent updates its decision-making *policy* to maximize the reward that it can get.

In this chapter, I will explain the basics of Q-learning that I learnt over the course of this project.

4.1 Q-Learning

- In reinforcement learning, a common way to help the *agent* to take the optimal *action* is by Q-learning. It is based on Q-values which are nothing but the discounted future rewards that the *agent* receives by taking various actions moving through the different *states* of the *environment*.
- These Q-values themselves are approximated through training, using deep neural network in our case. The selection that the *agent* makes

is more often than not the action with the highest Q-value.

- To approximate the Q-values, a Q-function computes the difference between the predicted Q-values and the *true* Q-values which correspond to the ideal *action* that should have been taken at that particular *state* of the *environment* to gain the maximum award.
- The Q-function obeys the following *Bellmann* optimality equation -

  $$Q^*(s,a) = E[r + \gamma max_{a'} Q^*(s^{'}a^{'})]$$

  This means that the maximum return from state s and action a is the sum of the immediate reward and the return (discounted by γ) obtained by following the optimal policy thereafter until the end of the episode.
- Based on the loss calculated through this difference, the neural network parameters are update using any standard gradient descent algorithm.
- After enough iterations, the expectation from this neural network is to converge such that it can approximate the Q-values of the *next state* given the *current state* of the *environment*.
- The basic idea behind Q-Learning is to use the Bellman optimality equation as an iterative update -

  $$Q_{i+1}(s,a) \leftarrow E[r + \gamma max_{a'} Q_i(s^{'}a^{'})]$$

  , which then converges to the optimal Q-function.
- This is how the *agent* will decide which action is the best for the maximum possible reward collection.

4.2 Deep Q-Learning

- The problem with simple Q-learning is it's greedy approach.
- The *agent* will always pick the *action* with the highest Q-value which might only be optimal locally, whereas some other action with a lower value could lead to other actions in further *states* to be available with higher Q-values than what would be available otherwise.
- The aim is to provide some leverage to the "worth-a-try" actions that have lesser Q-values than the most optimum local actions. This problem is dealt with by using deep Q-learning.

- In deep Q-learning, the *agent* chooses the *action* to perform not simply based on the maximum current Q-value, but the future Q-values are computed after any *action* is taken and are discounted by a factor.
- These then determine the target Q-value set from which the maximum valued *action* is chosen.

4.3 ReimproveJS

- ReimproveJS is a library to create Reinforcement Learning environments with Javascript.
- This library was used to develop the DQN Agent on top of the available Mario Maker code-base.
- ReimproveJS works with an *Academy* which contains *Teachers* and *Agents.*
- *Teachers* are added to to the *Academy* and *Students* are assigned to them.
- The *Teachers's* inputs are taken at each iteration by the *Academy* and then it takes care of the learning.
- ReimproveJS implements the DQN Agent as a wrapper around the TensorflowJS neural networks.
- *Model* is created as a wrapper around TensorflowJS neural network.
- *Agent* is then specified with various metrics. A step-learn is setup after which the *Agent* can be given the reward.

4.4 Mario Agent Implementation

The directory structure for the implementation of the mario-agent with the given code-base looks as follows:

```
/
├── css
│   ├── style.css
│   └── reset.css
├── fonts
├── images
├── js
│   ├── levelEditor
│   ├── marioAgent
│   │   ├── MarioAgent.js ...... MAIN CODE FOR THE AGENT
│   │   ├── GameMetricsPanel.js ........... UI CODE
│   │   ├── CoinPlot.js .. PLOT FOR THE COLLECTED COINS
│   │   ├── DistancePlot.js .......... PLOT FOR THE DISTANCE COVERED
│   │   └── RewardPlot.js .. PLOT FOR THE OVERALL REWARD
│   ├── mainGame
│   │   ├── Bullet.js
│   │   ├── Element.js
│   │   ├── Enemy.js
│   │   ├── GameSound.js
│   │   ├── Mario.js
│   │   ├── Powerup.js
│   │   ├── Score.js
│   │   └── MarioGame.js
│   ├── GameUI.js
│   ├── MarioMaker.js
│   ├── Preloader.js
│   └── View.js
├── sounds
├── README.md
└── index.html
```

MarioAgent.js

- This file contains all the code written for the development of the DQN Agent to play Mario using the library ReimproveJS.
- The initialisation of the model is done as shown below -

```
const modelFitConfig = {     // Exactly the same idea here by using
    tfjs's model's
    epochs: 1,               // fit config.
    stepsPerEpoch: 16
};

const numActions = 10;       // The number of actions your agent
   can choose to do
const inputSize = 10;        // Inputs size (10x10 image for
   instance)
const temporalWindow = 1;    // The window of data which will be
   sent to your agent

// For instance the x previous inputs, and what actions the agent
   took

const totalInputSize = inputSize * temporalWindow + numActions *
   temporalWindow + inputSize;


```

- The teacher configuration is set as shown below -

```
const teacherConfig = {
    lessonsQuantity: 10,         // Number of training lessons
   before only testing agent
    lessonsLength: 100,          // The length of each lesson (in
   quantity of updates)
    lessonsWithRandom: 2,        // How many random lessons before
   updating epsilon's value
    epsilon: 1,                  // Q-Learning values and so on ...
    epsilonDecay: 0.995,         // (Random factor epsilon,
   decaying over time)
    epsilonMin: 0.05,
    gamma: 0.8                   // (Gamma = 1 : agent cares really
    much about future rewards)
};

```

- The agent configuration is set as shown below -

```
const agentConfig = {
    model: model,                          // Our model corresponding to the agent
    agentConfig: {
        memorySize: 50000,                      // The size of the agent's memory (Q-Learning)
        batchSize: 128,                         // How many tensors will be given to the network when fit
        temporalWindow: temporalWindow          // The temporal window giving previous inputs & actions
    }
};

```

- The learning process is stepped asynchronously as shown below -

```
    // Step the learning process
    let result = await this.academy.step([               // Let the magic operate ...
        {teacherName: this.teacher, agentsInput: this.inputs}
    ]);

```

- Finally the algorithm to train the AI agent in the main game-loop is implemented as follows -

Algorithm 1 Training the AI agent to play Mario

```
Result: Trained AI agent
mario;
 marioAgent;
 while true do
    renderMap();

    marioAgent.processInputs();
     marioAgent.stepLearn();
     marioAgent.setKeys();

    mario.update();

    marioAgent.giveRewards();
end
```

Chapter 5

Interactive Reward Functions

The initial plan for this task was as shown in the figure below -

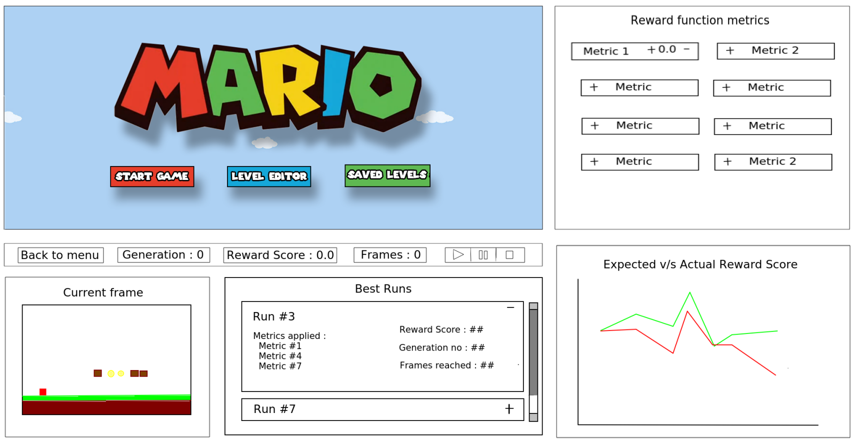


Figure 5.1: Initial plan

- The generation number, reward score and the frames passed will be displayed on a panel at the bottom of the game screen.
- Next, the window to add or remove the metrics to and from the reward function will be developed. This window will also allow to set the weights of the said reward function metrics.
- Subsequently, the window displaying a graph of the expected vs actual reward score will be developed.
- Another window will be developed to display the current frame that the AI agent sees to decide its next move.
- And the last window to be developed would be to view the best few runs yet.

5.1 Plots

Three different plots were made to analyze the AI agent's progress shown as follows -

5.1.1 Coins collected

This plot was to show the coins collected by Mario v/s time.
A sample plot is as shown below.

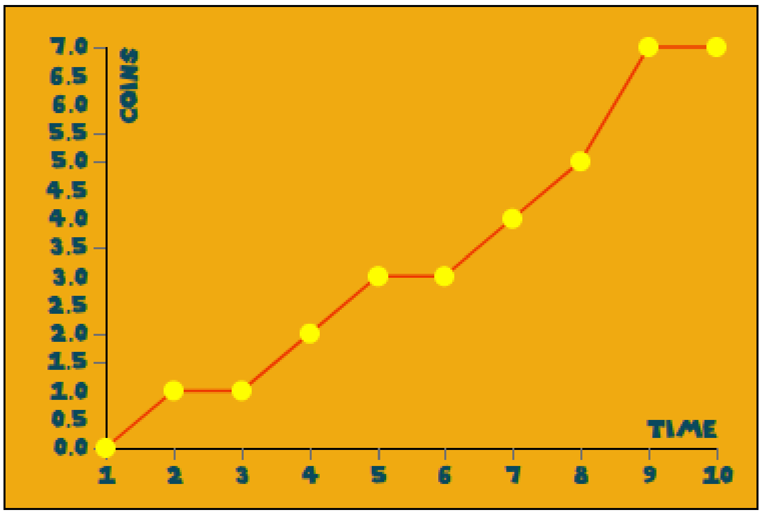


Figure 5.2: Coin-plot

5.1.2 Distance covered

This plot was to show the distance covered by Mario v/s time.
A sample plot is as shown below.

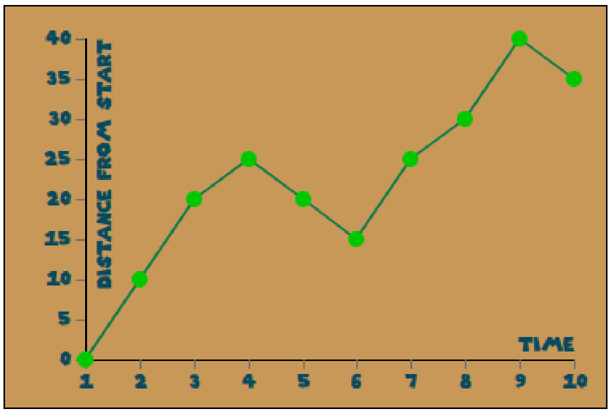


Figure 5.3: Distance plot

5.1.3 Total reward

This plot was to show the reward score obtained as a weighted sum by Mario v/s time.
A sample plot is as shown below.

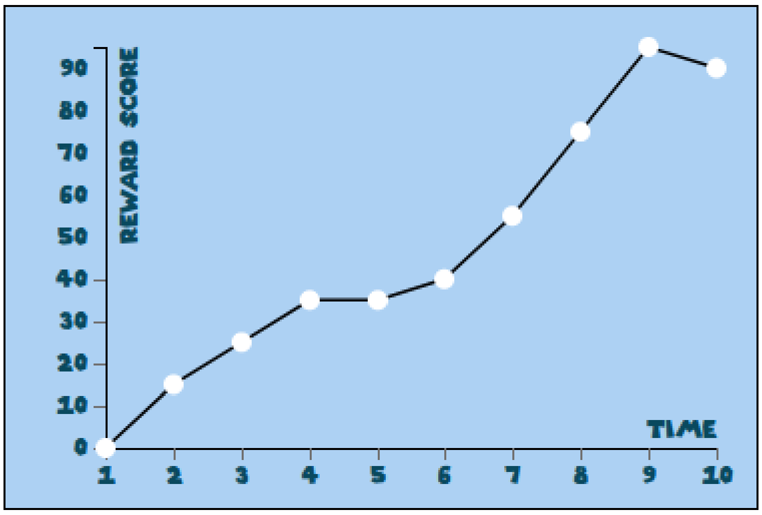


Figure 5.4: Reward plot

The reward function was calculated as a weighted sum of the coins collected and the distance covered by Mario.
So the reward function happens to be -

$$rewardScore = distanceWeight * distance\_covered + coinWeight * coins\_collected \quad (5.1)$$

The *distanceWeight* and the *coinWeight* were set through the Game Metrics Panel which is described in the next section.

5.2 Game Metrics Panel

5.2.1 Distance Weight Slider

The distance weight slider will allow you to set weight for the distance covered metric for the reward function of Mario.

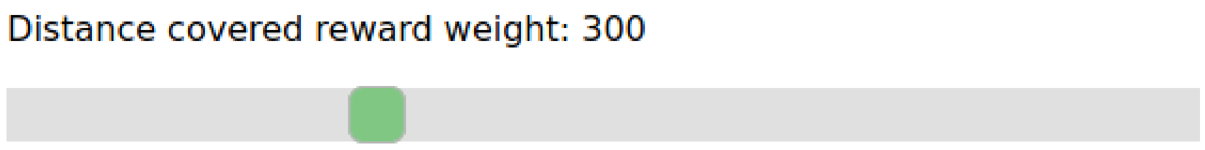

Distance covered reward weight: 300

Figure 5.5: Distance weight slider

5.2.2 Coin Weight Slider

The coin weight slider will allow you to set weight for the coins collected metric for the reward function of Mario.

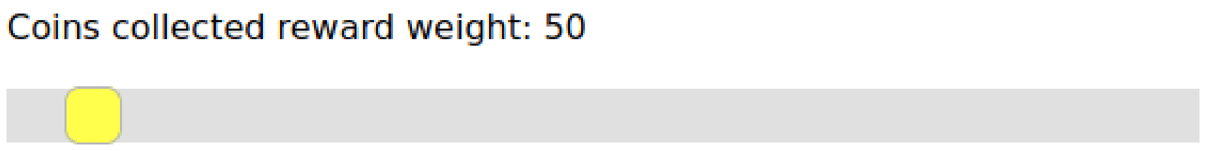

Coins collected reward weight: 50

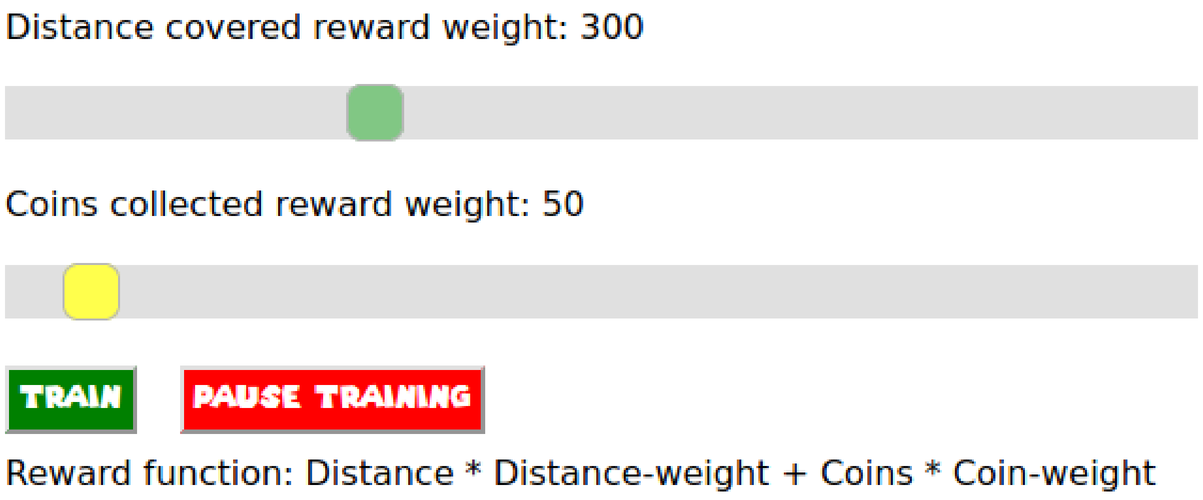
Figure 5.6: Coin weight slider

5.2.3 Control buttons

These buttons allow you to trains and pause Mario agent's training with a given reward function.
If you want to train Mario with a different set of metrics then you have to press train button again.

Figure 5.7: Train and pause buttons

And finally, the whole Game Metrics Panel looks like as shown in the figure below -



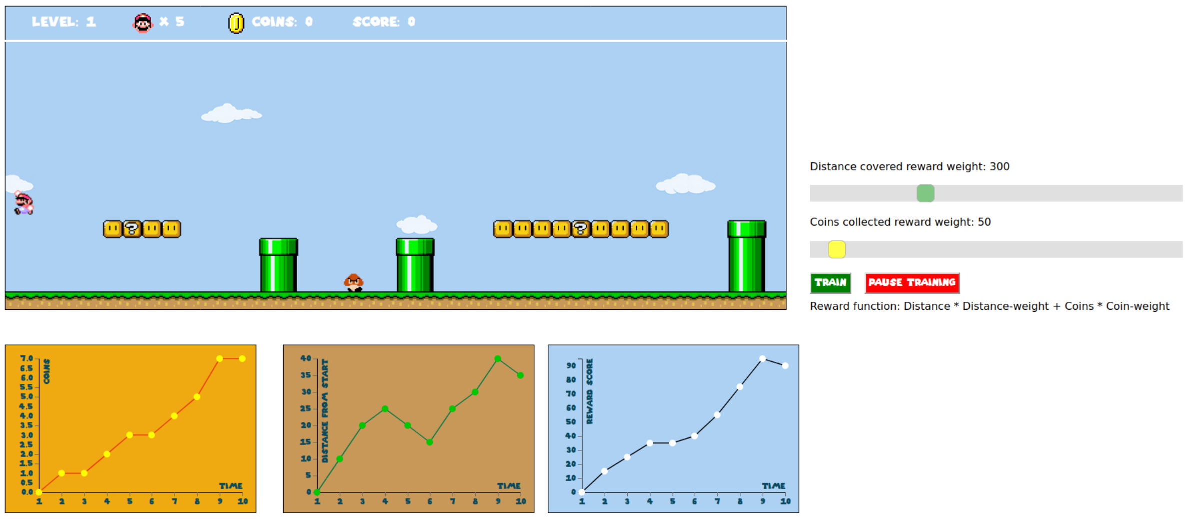
Distance covered reward weight: 300

Coins collected reward weight: 50

Reward function: Distance * Distance-weight + Coins * Coin-weight

Figure 5.8: Game Metrics Panel

The final look of the whole project looks like as shown in the figure below -



Distance covered reward weight: 300

Coins collected reward weight: 50

Reward function: Distance * Distance-weight + Coins * Coin-weight

Figure 5.9: Final screenshot

5.3 Conclusion

- The progress made in this project is quite limited albeit a lot of work was put towards it.
- Some of the future possibilities for this project could be -
  - More metrics to set weight for.
  - Saving trained Mario Agents.
  - Dynamic plots.

Bibliography

[1] Bharat Adibhatla. *Understanding The Role Of Reward Functions In Reinforcement Learning*. 2019. URL: https://analyticsindiamag.com/understanding-the-role-of-reward-functions-in-reinforcement-learning/ (visited on 08/20/2020).

[2] J. C. D. Amodei. *Faulty reward functions*. 2016. URL: https://openai.com/blog/faulty-reward-functions.

[3] *Q-Learning*. 2020. URL: https://www.tensorflow.org/agents/tutorials/1_dqn_tutorial.

[4] Github user: pratishr. *Mario Maker*. 2020. URL: https://github.com/pratishshr/mario-maker.

[5] Github user: BeTomorrow. *ReimproveJS*. 2020. URL: https://github.com/BeTomorrow/ReImproveJS.