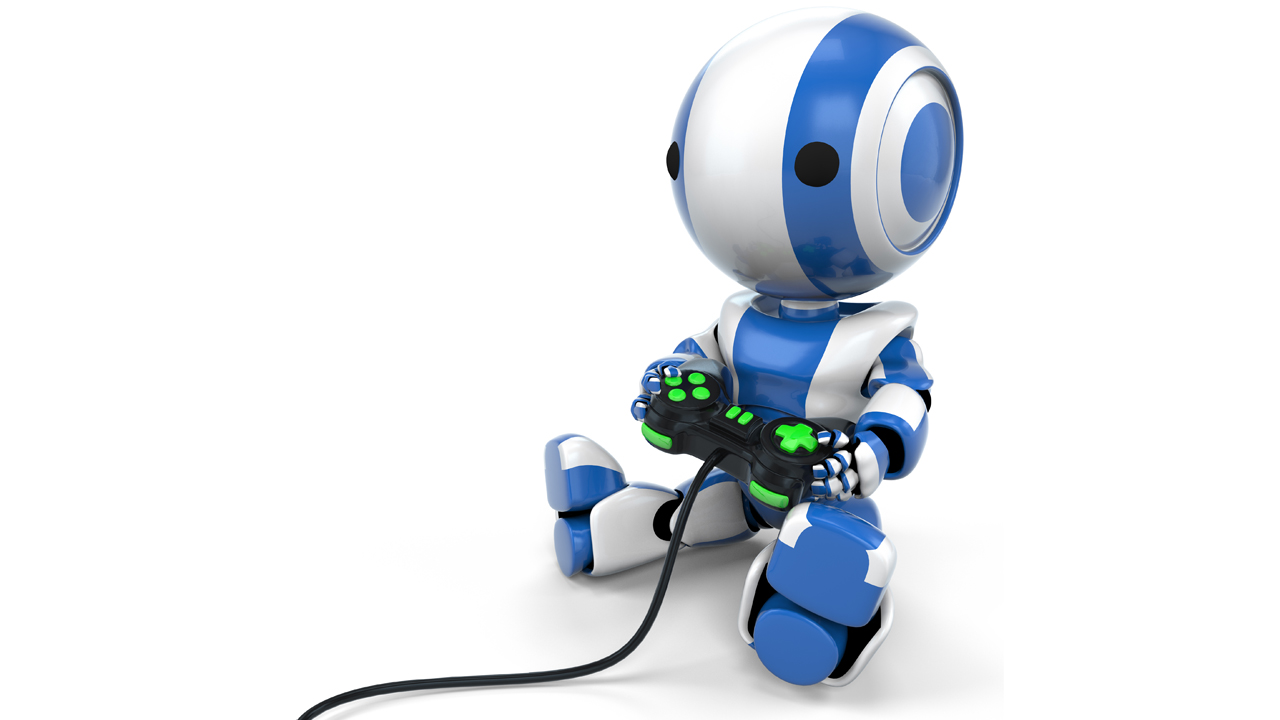
Game Play with Reinforcement Learning

Video game, a game played on computer or mobile or any gaming console. And we all love to play video games. Technically there are many types of video games. Some are multiplayer games, where two of more people play the game together on a same or different machine. Some are single player games, only a player plays it on a single machine. Most of the single player game is actually played against the computer. Like cheese, soccer, boxing etc. etc. But computer can’t think or can’t take any decision like a human, then how it played against us? Actually, it’s mostly hardcoded or randomly generated. And some times it become really hard to beat the computer in a game.



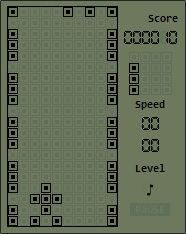
But in the current situation, where the technology is growing so fast, and AI become so powerful and taking decisions in many critical situations like a human, can a computer beat another computer in a simple game, without hardcoding, like a human? If we technically analyze the problem then first of all the computer have to learn how to play the games, what are the rules of the game, how to win the game and what will be happen if it did any action. And all of the solutions of these problems in hidden in the *Reinforcement Learning*. Most of us actually know that there are two types of learning in Machine learning. One is Supervised Learning and another one is Unsupervised Learning. But there are also another types of learning which is Reinforcement Learning.

In Supervised Learning there are labeled dataset to train the model. In Unsupervised Learning there are unlabeled dataset to train the model. But in Reinforcement Learning it doesn’t require any dataset. It learns from the feedback. First of all, it required an environment, a set of action, and feedbacks. At the beginning it takes some random actions and based on the feedback it creates a policy and as it learns more, it updates that policy which will help it to choose the correct action at the current state. And in video game all of these are available. So, we can build Reinforcement Model which can learn and play video games.

**Procedure**

* Create a game, or a game environment which can tells us about the current state of the game and can return any feedback value based on the current action.
* Using Q Learning, a Reinforcement Learning algorithm, build the model that will explore the game environment, get the feedbacks and will take actions.
* Make the game and qlearning model together that, all can work together and record some important data for performance measurement.
* Visualize the saved data and see what could be improve and fine tune in the model’s parameters.
* Optionally if we want then we can create another program that will automatically check the recorded data and fine tune the parameters according to the need, like sklearn’s grid search function. But it will really be complicated and that’s not our topic. So, we are not going to do that.

**Game Environment**



We all have played this game in our childhood and it’s a really interesting game and also very tough. Here the car automatically moves forward, it doesn’t have any break and we have to go as long as we can avoiding the other cars by steering our car left or right.

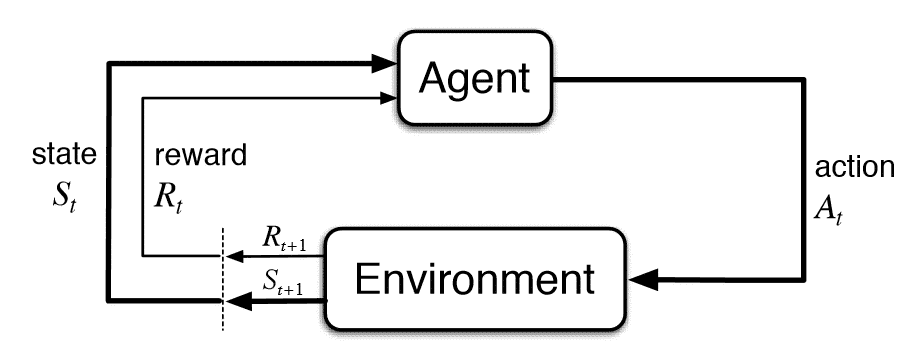
Here the environment is the road with the other cars and also our own car. And the state of the environment is our car’s location and also the other car’s location. And the actions we have is steer left or steer right. And the feedback is as we will move on without crashed with other cars, we will get scores which is a positive reward that encourage us to play the game this way. And if we crashed with other car then the game will over which is a negative reward which tells us to improve the way you are playing. And this way playing more and more we learn when and how fast we have to steer left or steer right.

For this experiment this game like environment is created with some modification. First of all is reward system. For going forward the q learning agent will receive 10 reward points, for avoiding a car it will receive 30 reward points and if it crashes with another car the it will receive a high negative reward, -999. This way it will know that going forward is a good thing, then avoiding another car is also a good thing and it’s more important than going forward and it have to avoid the other cars because it’s a too bad thing. And the action set it will have 0 and 1. If it chose action 0 then the car will steer left and for 1 the car will steer right.

**Reinforcement Learning**

Reinforcement Learning has 5 key parts,

(i) agent, (ii) environment, (iii) reward, (iv) policy, (v) action



**Agent:** Which explore the environment, take actions, get rewards, and learn about the environment and its policies. It may be a virtual agent or a physical agent. Here the game player is the agent. And a real world agent may be a maze solving Arduino robot, where the Arduino robot is the agent.

**Environment:** It’s the world or the place where the qlearning will be use. It may be the real world or the virtual world. Here in this case the game is the environment. And a real-world example is maze solving Arduino robot, where the maze is the real-world environment.

**Action:** Things, the agent can do in that environment. Like move up, move down, rotate, click a button etc. etc.

**Policies:** Every environment has its own rule and condition, like in this game if our car touch any other car then it will count as a crash and a negative reward will be given for that. The policy is a function that learn from the environment about those rules and conditions and based on the current knowledge it chose what action have to take at that state. Usually this function is denoted as , and it return the probability of taking action **a** in state **s**.

**Reward:** As we know that Reinforcement Learning is based on the feedback, this is the feedback from which the agent learn which action is best for that state and what have to avoid. And the agent’s main aim is to maximize the reward.

That is the total reward the agent will get at the time *t*. But this has some problem. Thus, the total reward may go to infinity, and because the agent is always looking for maximum reward it may confuse the agent about the maximum reward point. And we also want to predict what reward the agent may get in future.

One way to full fill our need is to use a decreasing factor for future rewards.

Now, if we assign = 1, it will take us back to the first equation where every reward is equally important. And if we set = 0, then it will just focus on the immediate reward only. By setting the value between 0 to 1 we can adjust how much have to focus on the future rewards and how much have to focus on the current reward.

**QLearning**

Qlearning is a model free and off policy reinforcement learning algorithm. It’s very easy and efficient technique for our task. It uses the Bellman equation and take two input, state and action and the Q value.

**Q-Table**

It’s a table where the agent calculates the maximum expected future rewards for action at each state. In other word this table will guide the agent to select the best action at each state, like a policy function.

**QLearning algorithm**

* **Initialization**

First of all, a Q-table is built of n columns and m rows, where the n is the number of actions and m is the number of states.

* **Perform an action**

In qlearning any action is chosen based on the q-table. But currently the Q-table is initialized with all zeros. So, here the concept of exploration and exploitation comes.

**Epsilon greedy strategy:** The epsilon is a value ranging between 0 to 1, which tells the agent to chose the action based on the previously learned information or explore the environment more by taking any random action.

At the initial state the epsilon value is very high ( close to 1, like 0.9, 0.92, 0.97) and the agent focus more to explore the environment and take new and new actions and thus experiment with the environment and learn more about it and store the learning data into the q table. And as the agent learn more about the environment and the epsilon value decrease with each action and comes close to zero, like 0.2 or 0.23.

* **Evaluation**

When the agent will have more confidence (epsilon value close to zero) then it will chose action based on its previous observation (Q-table).

**Code of the game environment**

Here we are creating the game environment from scratch. So, lets first understand what’s going and how it’s working. Because without knowing that creating the model is so difficult.

First of all, here in the following section we import some important python libraries.

The ‘os’ library is imported to know currently in which operation system the program is running. It will help the code the select the command to clear the console. Because in windows ‘cls’ command is used to clear the screen and in linux or mac os ‘clear’ command is used to clear the screen.

Here the time library is not so important it’s just used to testing and debugging purpose.

And the random function is used to randomly generated the other cars on the road.

And there is just a little role of numpy.

import os

import time

import random

import numpy as np

This is the main class where we will build the environment.

class Game:

…

These all of the methods are inside the Game class. First of all, in the *\_\_init\_\_* method we initialize all of the variables and the default environment state.

    def \_\_init\_\_(self):

        self.screen\_buf = []  # virtual screen buffer

        self.road\_wl = ['||', '   ', '|', '   ', '||']   # road with line

        self.road\_wol = ['  ', '   ', ' ', '   ', '  ']   # road without line

        # my cars part 1, here two backslash (\\) represent only one backslash (\)

        self.m\_car\_1 = '/^\\'

        self.m\_car\_2 = ']#['

        self.m\_car\_3 = '=-='

        self.o\_car\_3 = '=-='  # other's car part 3

        self.o\_car\_2 = '!#!'  # other's car part 2

        # other's car part 1, here two backslash (\\) represent only one backslash (\)

        self.o\_car\_1 = '\\\_/'

        self.lcar = -1

        self.rcar = -1

        self.mcar = 1

        self.distance = 0

        self.state\_LOW = np.array([-1, -1, 0])

        self.state\_HIGH = np.array([15, 15, 2])

        self.action\_set = np.array([0, 1])

        # load the screen cleaner for different OS

        if os.name == 'nt':

            self.clr\_scr\_cmd = 'cls'

        else:

            self.clr\_scr\_cmd = 'clear'

        os.system(self.clr\_scr\_cmd)

        self.car\_passed = 0

        self.crashed = 0

Let’s discuss about these variables.

**screen\_buf:** This is a virtual screen buffer where we will store the visual elements before rendering and printing to the screen.

**road\_wl & road\_wol:** Road with lines & road without lines. These are just a graphic element.

**mcar\_1, mcar\_2 & mcar\_3:** Graphics element of our car. Divided into 3 parts for simplification of rendering.

**ocar\_1, ocar\_2 & ocar\_3:** As like before this is the other car’s graphics element divided into 3 parts.

**distance:** The distance we traveled yet. It we covert 1000 unit distance then we will win the game.

**state\_LOW, state\_HIGH, action\_set:** What are the minimum values of the state, what are the maximum values of the state and the set of actions the q learning agent can take.

Then it chose the right command to clear the output screen based on the different operating system.

**car\_passed:** How may car passed between we go from 0 to 1000-unit distance.

**car\_crashed:** How many times we crashed with the other car.

This reset function is also a part of the Game class, which reset the state and score and also the screen buffer for the next game and then return the state.

    def reset(self):

        self.screen\_buf = []

        self.lcar = -1

        self.rcar = -1

        self.mcar = 1

        self.distance = 0

        self.crashed = 0

        self.car\_passed = 0

        return [self.lcar, self.rcar, self.mcar]

The following 3 methods are the method which are responsible for the changing of the background, means the road lines.

    def screen\_1(self):

        for i in range(4):

            self.screen\_buf.append(self.road\_wl.copy())

            self.screen\_buf.append(self.road\_wol.copy())

            self.screen\_buf.append(self.road\_wol.copy())

    def screen\_2(self):

        for i in range(4):

            self.screen\_buf.append(self.road\_wol.copy())

            self.screen\_buf.append(self.road\_wl.copy())

            self.screen\_buf.append(self.road\_wol.copy())

    def screen\_3(self):

        for i in range(4):

            self.screen\_buf.append(self.road\_wol.copy())

            self.screen\_buf.append(self.road\_wol.copy())

            self.screen\_buf.append(self.road\_wl.copy())

It has three different screen, and changing these three different screen at a high frame per second make it looks like moving from the upward to the downwarn of the screen.

**render\_scren\_buf,** the main method of Game class for rendering the graphics element store in the *screen\_buf* , virtual screen buffer and print to the screen. And then clean the virtual screen buffer to store the next frame’s element.

    def render\_screen\_buf(self):

        os.system(self.clr\_scr\_cmd)

        for i in range(12):

            tmp\_row = ''

            for j in self.screen\_buf[i]:

                tmp\_row += j

            self.screen\_buf[i] = tmp\_row

        print('-'\*11)

        print('\n'.join(self.screen\_buf))

        print('-'\*11)

        self.screen\_buf = []

And that’s the main method of Game class that run the game. All the game logic and rules are in it and it check for crashes update score and get action from the q learning agent and return the new state, reward and if it wins the game or not. And that’s the part with which our q learning agent will talk. It’s like an API.

    def game\_play(self, action):

        self.mcar = action

        # becoming car from the left lane's probability is 1/10

        # check if the left lane is free or not and the random state

        # and also check if the differece between two car is greater than 6 unit or not

        if self.lcar == -1 and (self.rcar == -1 or abs(self.lcar-self.rcar) > 6) and random.randint(0, 10) == 10:

            self.lcar = 0

        # becoming car from the right lane's probability is 1/12

        # check if the right lane is free or not and the random state

        # and also check if the differece between two car is greater than 6 unit or not

        if self.rcar == -1 and (self.lcar == -1 or abs(self.lcar-self.rcar) > 6) and random.randint(0, 12) == 12:

            self.rcar = 0

        if self.distance % 3 == 0:

            self.screen\_1()

        elif self.distance % 3 == 1:

            self.screen\_2()

        else:

            self.screen\_3()

        self.distance += 1

        # if any other car is on the screen then set it properly and update the state of that car

        if self.lcar > -1:

            if self.lcar < 12:

                self.screen\_buf[self.lcar][1] = self.o\_car\_1

            if self.lcar > 0 and self.lcar < 12:

                self.screen\_buf[self.lcar-1][1] = self.o\_car\_2

            if self.lcar > 1 and self.lcar < 12:

                self.screen\_buf[self.lcar-2][1] = self.o\_car\_3

            self.lcar += 1

        if self.rcar > -1:

            if self.rcar < 12:

                self.screen\_buf[self.rcar][3] = self.o\_car\_1

            if self.rcar > 0 and self.rcar < 12:

                self.screen\_buf[self.rcar-1][3] = self.o\_car\_2

            if self.rcar > 1 and self.rcar < 12:

                self.screen\_buf[self.rcar-2][3] = self.o\_car\_3

            self.rcar += 1

        if self.mcar == 0:

            self.screen\_buf[9][1] = self.m\_car\_1

            self.screen\_buf[10][1] = self.m\_car\_2

            self.screen\_buf[11][1] = self.m\_car\_3

        elif self.mcar == 1:

            self.screen\_buf[9][3] = self.m\_car\_1

            self.screen\_buf[10][3] = self.m\_car\_2

            self.screen\_buf[11][3] = self.m\_car\_3

        self.render\_screen\_buf()

        reward = 10

        if self.distance >= 1000:

            done = True

        else:

            done = False

        if self.lcar > 8 and self.mcar == 0:

            print('Car crashed!')

            self.crashed += 1

            reward = -999

        elif self.rcar > 8 and self.mcar == 1:

            print('Car crashed!')

            self.crashed += 1

            reward = -999

        if self.lcar > 8 and self.mcar == 1:

            reward = 30

        elif self.rcar > 8 and self.mcar == 0:

            reward = 30

        # if the other car goes out of the screen then the it reset the state of that car

        if self.lcar == 14:

            self.lcar = -1

            self.car\_passed += 1

            reward = 50

        if self.rcar == 14:

            self.rcar = -1

            reward = 50

            self.car\_passed += 1

        return ([self.lcar, self.rcar, self.mcar], reward, done)

**Implementation of our Qlearning agent**

As before first of all import all the important libraries.

import time

import joblib

import numpy as np

from game import Game

Here the time library will be use just for set a proper frame changing time. Joblib library is used to store the recorded data, like how many car passed and number of crashes, for graph plotting and performance measurement. Here numpy has an important role to implement the qlearning. And here the Game class is imported to connect the agent and the game environment together.

game\_env = Game()

First of all an object of game environment is created, named *game\_env.*

Then two empty list is created to store the data, car\_passed to store the number of car passed, crased to store the number of crashes happened.

car\_passed = []

crashed = []

Now in this section, some very important things are initialized. Number one is LEARNING\_RATE, which is how fast our agent will try to go from initial state to the optimal state. It we set a too high learning rate then it will always jump over the optimal state and never can reach there. And if we set it too low then it will take too much time to reach at the optimal state. There is no equation of formula to find the right learning rate. We just have to change it based on the trial and error. The DISCOUNT, which is the epsilon value, which tells the agent if it has to go for exploration or exploitation. And as we discuss earlier, the epsilon value is initially assigned with a high value (close to 1). And here we assign 0.95. And then episode is how many times the agent will play the game. It’s like the epochs in machine learning or deep learning. For more episodes it can learn more.

LEARNING\_RATE = 0.09

DISCOUNT = 0.95

EPISODES = 25000

The following section is to create the Q-table. But here instead of 0, we initialize the q-table with random values ranging between -2 to 0.

DISCRETE\_OSV\_SIZE = [10] \* len(game\_env.state\_HIGH)

discrete\_osv\_win\_size = (game\_env.state\_HIGH -

                         game\_env.state\_LOW)/DISCRETE\_OSV\_SIZE

q\_table = np.random.uniform(

    low=-2, high=0, size=(DISCRETE\_OSV\_SIZE+[len(game\_env.action\_set)]))

Return the discrete state of the passing state.

def get\_discrete\_state(state):

    discrete\_state = (state - game\_env.state\_LOW)/discrete\_osv\_win\_size

    return tuple(discrete\_state.astype(np.int))

This for loop will run the game for 25000 (EPISODES = 25000) episodes. At the first of every loop the game environment will be reset, and as we discuss earlier, after reset the environment the reset method returns the state. And then the discrete state of that state is calculated in stored inside the *discrete\_state* variable. And before every game the *done* is initialize with *False*. Then the while loop run the game until the game is over.

for eps in range(EPISODES):

    discrete\_state = get\_discrete\_state(game\_env.reset())

    done = False

    while not done:

        action = np.argmax(q\_table[discrete\_state])

        new\_state, reward, done = game\_env.game\_play(action)

        new\_discrete\_state = get\_discrete\_state(new\_state)

        if not done:

            max\_future\_q = np.max(q\_table[new\_discrete\_state])

            current\_q = q\_table[discrete\_state + (action,)]

            new\_q = (1-LEARNING\_RATE)\*current\_q+LEARNING\_RATE \* \

                (reward+DISCOUNT\*max\_future\_q)

            q\_table[discrete\_state+(action,)] = new\_q

        discrete\_state = new\_discrete\_state

        print('Crashed : ', game\_env.crashed)

        print('Episode : ', eps)

        time.sleep(0.001)

    crashed.append(game\_env.crashed)

    car\_passed.append(game\_env.car\_passed)

    # because it will take too much time to complete 25000 episodes,

    # just for testing purpose we are doing only 100 episodes.

    if eps == 100:

        break

Inside the while loop, first of all based on the discrete state, the action probability set is selected from the *q\_table.* And the action with higher probability is performed. Then in the next line the *game\_play* method is called with the selected action and then the method checks all the conditions bring the other car randomly on the road and render the frames. And finally returns the new\_state, reward and tells if the game is over or not. If the game is not over yet then it calculates the new q value and then update the Q-table with the new q value. And create the discrete state of the current state. And this process is continuing until the game is over. Then it prints the scores and other values and store it in the proper list.

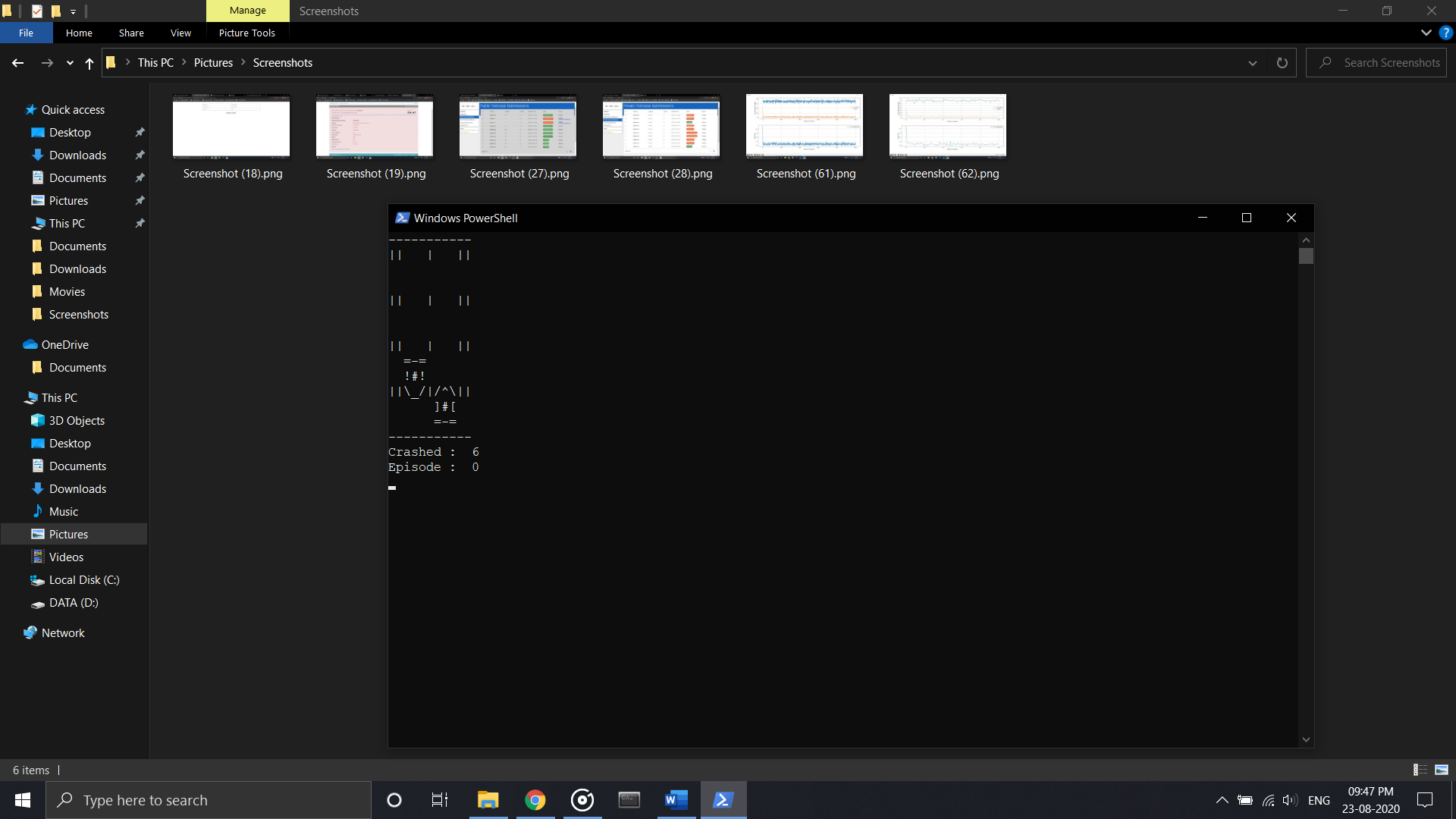
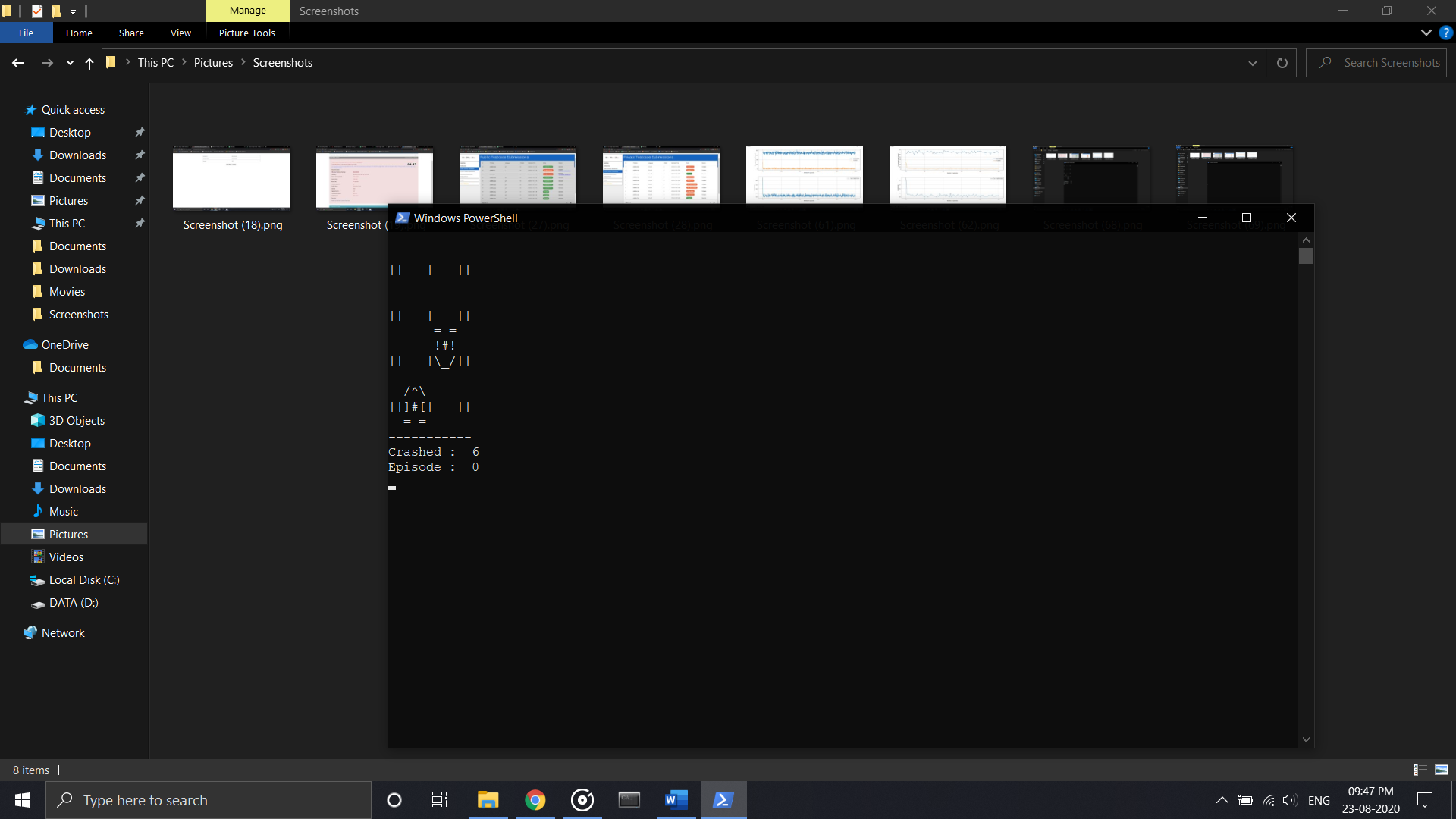
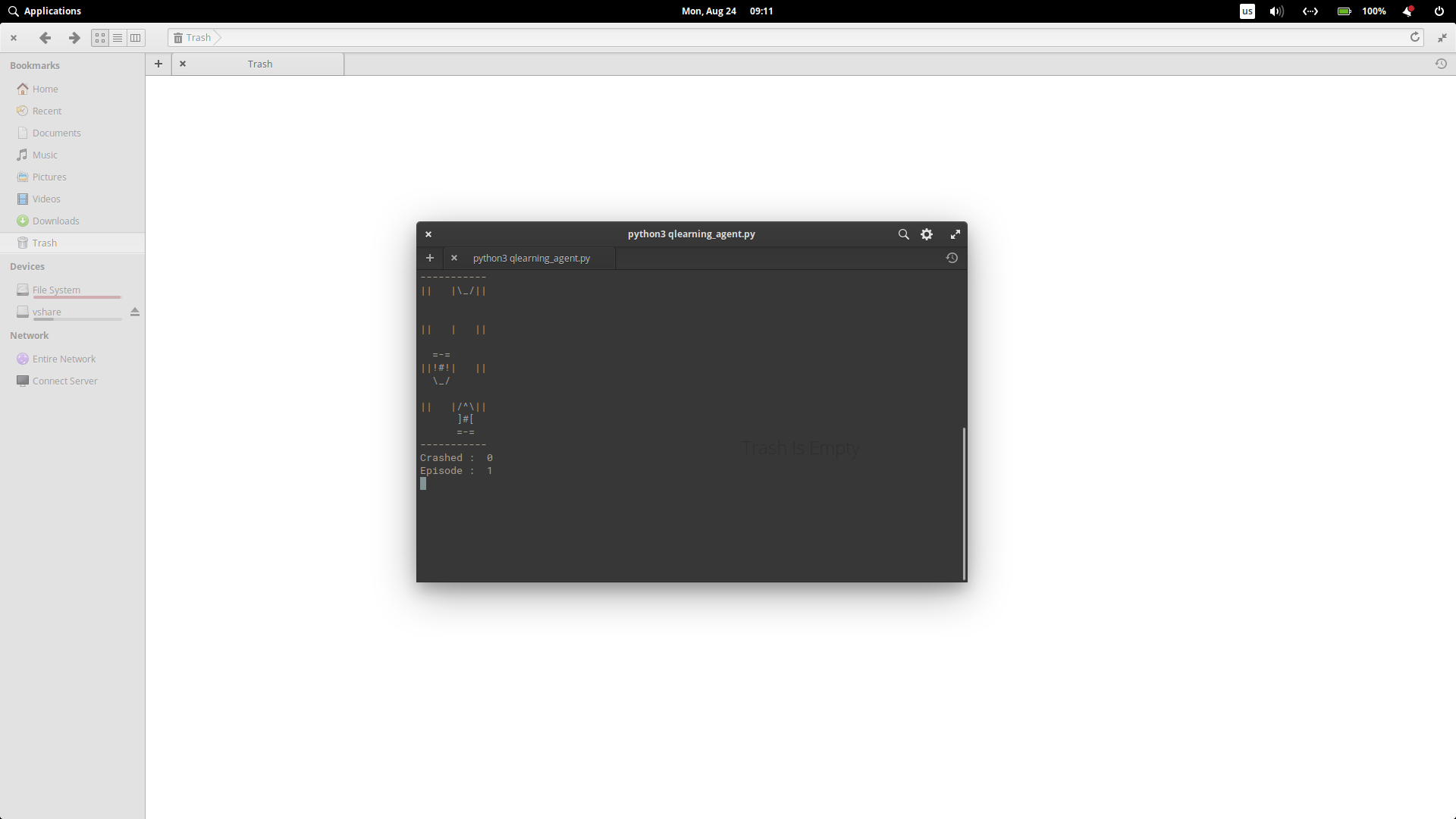
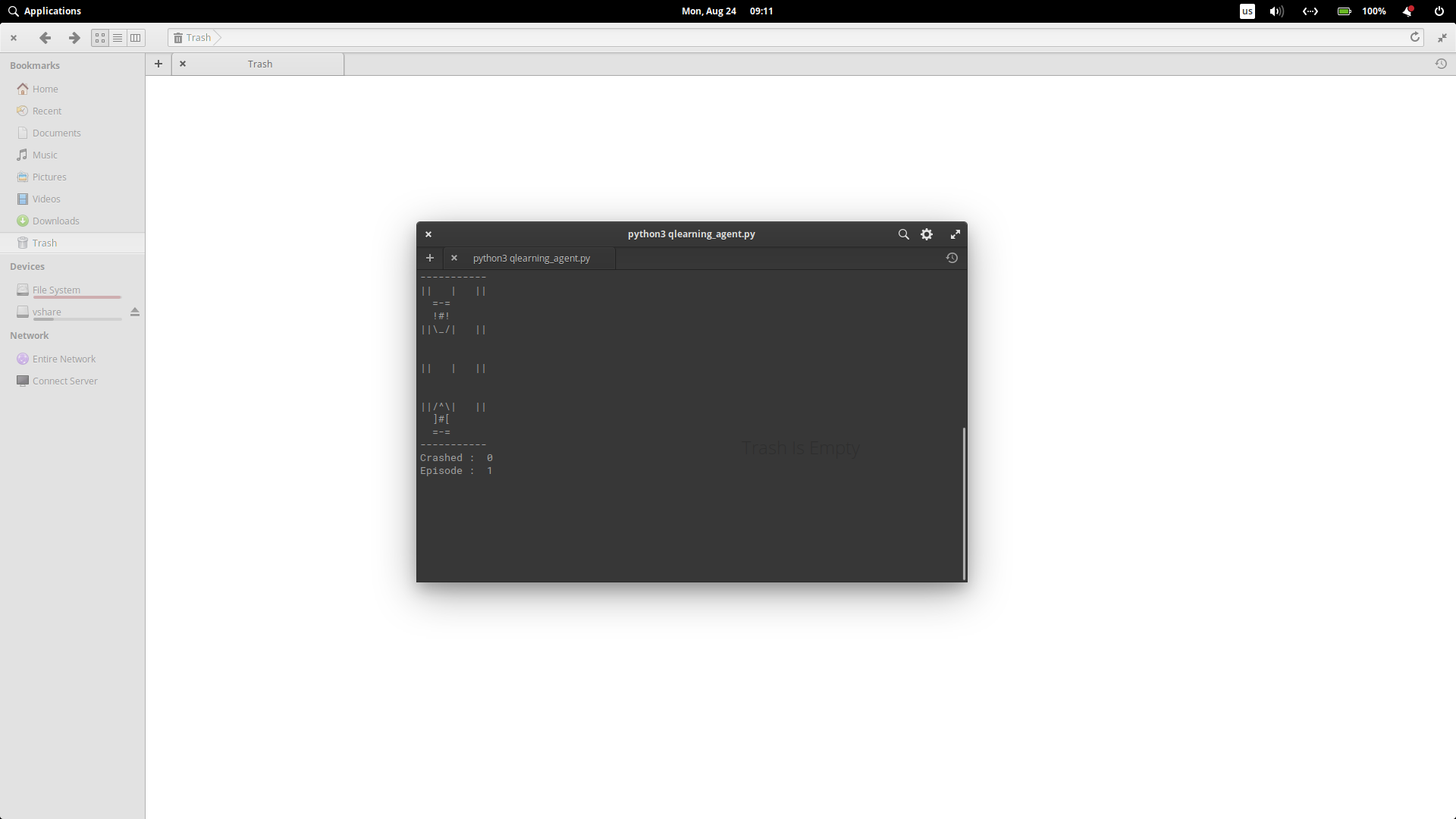
At the end the list will be stored as the pickle file in the local storage.

joblib.dump(crashed, 'crashed.pkl')

joblib.dump(car\_passed, 'car\_passed.pkl')

**Results**

And here we can see some screenshots of the game played by our q learning agent. And the program run very well on different OS. Currently the game has been tested on Windows 10 and Elementary OS (Linux), with two different python versions, (on Windows python 3.8.1 and on Linux python 3.6.9)

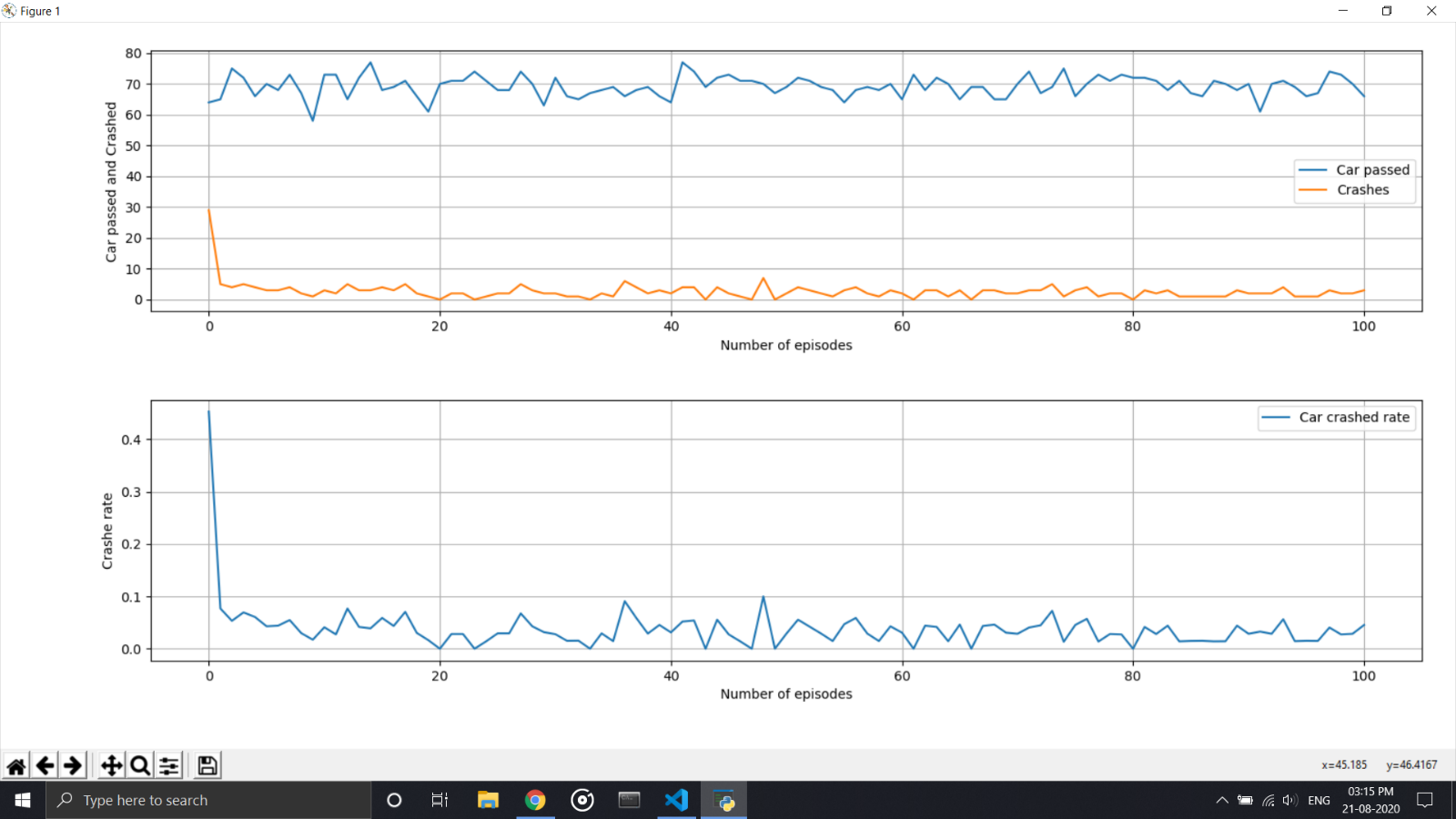
**   **

* **First two screenshots are on Windows & the last two screenshots are on Linux.**

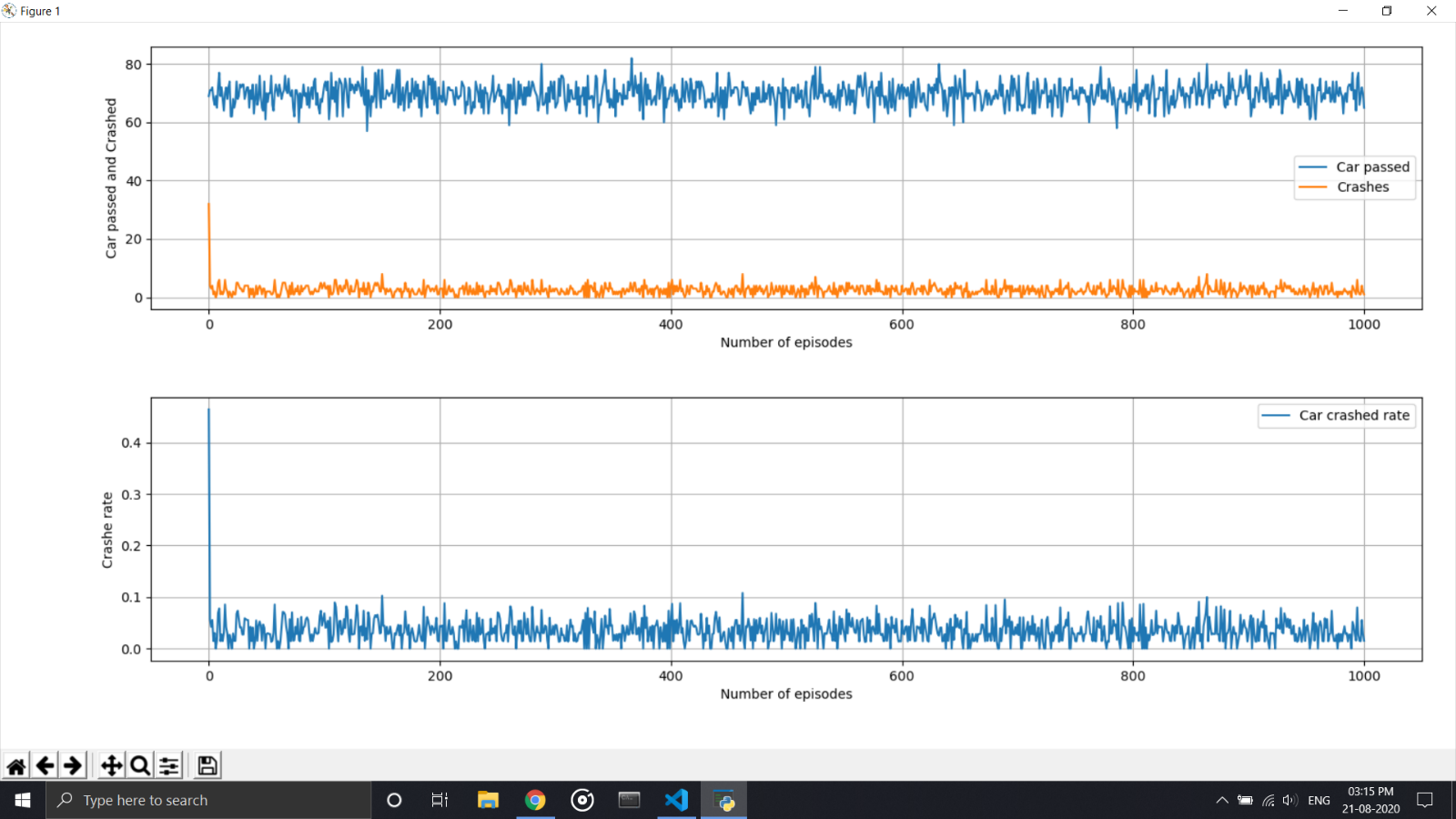
**Performance measurement**

After plotting the recorded data of the model, we can see if it did well or not. Here in the all following plot the x axis is for number of episodes. The game is over when our car went 1000-unit distance. And this one game is one episode.

**100 episodes:**

****

**1000 episodes:**

****

In any episodes plot the first plot shows that how may other car has passes (represented by the blue line) and among them with how may car we have crashed (represented by the orange line). In 100 or 1000 episodes, we can see that in every episode around 70 to 80 cars have passed. And around 5 to 10 times our car had been crashed with another car.

And in both of the episode’s plot set in the second plot the crash rate is plotted. Which is:

If we multiply 100 with it, we will find the percentage of crashes. And from the plot we can see that the crash rate is around 0.05 to 0.1. That’s mean around 5 to 10% time our car crashed with another car.