Chapter – 2 (implementation – Part 1 – Installation & Environment)

**Artificial Intelligence**

**AI-app (Deep-Q): Self Driving Car** (part 1)

Installation

Environment & Libraries

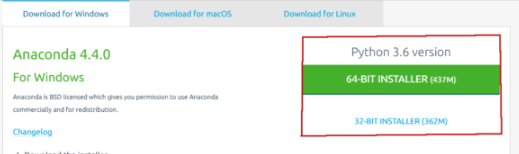
**2.1 Installations**

* Plan of Attack (Practical Tutorials): In this Module we are going to build a Self Driving Car... from scratch !
* First we will install everything required to make that car.
* After that we will make the environment containing the map, the car and all the features that go with it.
* Then we will build the AI and integrate it to the car.

We will implement the ***Self-Driving Car*** with PyTorch. We would recommend to have a first look at the PyTorch documentation to start getting familiar with PyTorch.

* Windows Option 1 End-to-End installation steps: Pytorch +Kivy are quite easy to install on Mac and Linux, but there are 2 options for installing the tools on Windows. Instructional manual for you on how to install PyTorch + Kivy on Windows 10 (<https://www.superdatascience.com/blogs/pytorch-windows-installation-walkthrough>)
* To simulate installing the packages from scratch, I removed Anaconda, Python, all related environmental variables from my system and started from scratch
* Step 1: Installing Anaconda in your system

1. Go to https://www.continuum.io/downloads
2. Click on the windows icon as shown:
3. Download the right installer of python 3.6 (64-bit if you are on a 64-bit machine and 32-bit if you are on a 32-bit machine)
4. Install Anaconda from the downloaded exe



* Step 2: Download the Kivy Installation Wheel "Kivy-1.10.1.dev0-cp36-cp36m-win\_amd64.whl"

1. Download the wheel file to a known path (Path in example C:\Users\abc\_user\Downloads )

* Step 3: Open Anaconda Prompt as Admin

1. Open Anaconda Prompt from your windows search by right-clicking on it and selecting
2. ***Run as administrator*** *(This is very important for package permissions)*

* Step 4: Operations in the Anaconda Prompt

1. Once the Anaconda Prompt is open, type in these commands in the order specified. Enter Y to proceed when prompted. (first we try to install kivy with torch )
2. conda install -c anaconda python=3.6.1
3. conda install -c peterjc123 pytorch=0.1.12
4. Change the directory in the Anaconda Prompt to the known path where the kivy wheel was downloaded. ( For me this path is C:\Users\abc\_user\Downloads, so change the below command accordingly for your system)
5. Once the path has been changed successfully, you should now enter these commands in the Anaconda prompt to install Kivy.
6. pip install docutils pygments pypiwin32 kivy.deps.sdl2 kivy.deps.glew
7. pip install kivy.deps.gstreamer
8. pip install Kivy-1.10.1.dev0-cp36-cp36m-win\_amd64.whl

*Kivy* is now *setup* on your anaconda *environment successfully*

1. Let us check the installation : In your anaconda prompt, enter the following command

conda list

You should now be able to view the packages we just installed in the list generated

* Step 5: Setup your Spyder

1. Open Spyder from the start menu
2. Go to Tools -> Preferences as shown
3. In Run, now set the option to Execute in an external System Terminal and click Apply and Ok as shown.

* Step 6: Test your Program:

1. In Spyder, Navigate to the Self\_Driving\_Car folder,
2. Open map.py
3. Hit Run
4. If a prompt comes up asking where you want to run the program, select External Terminal
5. Enjoy the program

https://www.superdatascience.com/pytorch/

* Option 2: You can install a Virtual Box on your Windows and then install Ubuntu on top of it. Once done you will be able to install PyTorch and Kivy into Ubuntu.See the following 2tutorials for instructions on this.
* Our Previous Environment (computer vision): We used our previous environment, python 3.5 called "py354", where Pytorch 0.4.1 installed, we've used following commands to install **KIVY 1.10.1**
* Installations:

activate py354

pip install docutils pygments pypiwin32 kivy.deps.sdl2 kivy.deps.glew

pip install kivy.deps.gstreamer

pip install kivy.deps.angle (optional)

pip install kivy==1.10.1

* Use **ipython console** instead of classic console in spyder

view> panes > iPython console

**2.2 Why Pytorch & Kivy**

Why we chose PyTorch rather than TensorFlow for this course?

* PyTorch can handle dynamic graphs: That means that you have a graph of computations when you have to compute the *gradients* in *Deep-learning equations* lying behind artificial intelligence models.
* And this graph of computations allows us to compute very quickly the *gradients* and the *composition functions*.
* Indeed when you *feed forward the signals* in the Neural Network and when you *back propagate* the *loss error* back into the ***Neural Network*** you'll have these *gradients* of the *loss error* with *respect* to the *weights*.
* Since we have several layers in the neural networks you have to compute gradients of composition functions because *one layer* is expressed as a function of the previous layer which is also expressed as a function of the *previous-previous* layer.
* So you get these composition functions on which you have to compute the gradient. And that leads to nasty computations.
* The dynamic graph of the Pytorch will handle those computations very efficiently.
* The training with ***Pytorch*** & with a ***strong GPU acceleration*** will be much faster and therefore you will be able to train your model much more efficiently. So that's the main reason why we chose Pytorch.
* Besides it is developed by the best teams in the world by Facebook and NVIDIA.
* Kivy: Kivy is ***not related to building an AI***, but to make the app of the self-driving car. We're going to ***make*** a ***map*** with a ***car*** in it then we draw some roads for the car to stay in. For this we need Kivy.
* Recommended: Install the "Homebrew with pip".

**2.3 Getting started**

* We're going to have three code files.
* Two files to make the map: with the car in it and the movements of the car and some code to define the goals that the code will have to accomplish.
* Some code that will make the connection between the map and the AI.
* Then we'll have the *AI code* and that's the one we will *implement step by step* line by line so that you can clearly understand what's going on.

|  |  |
| --- | --- |
| * Before we start, set the right folder as working directory. We are going to have ***three files*** to ***implement*** the cars and all the cars are connected to each other. So these files should stay in the same working directory. |  |

* What are these three files:

1. **ai.py** contains the brain of the car. In this file we will implement the *artificial intelligence* that will be *integrated* to the *car*. It is based on a neural network.
2. **car.kv** is a **kivy** file, can open it from a text editor (vs-code/notepad++/sublime) and actually that's what I'm going to do right now.

* Basically we create several objects (car, angle, rotation, shape) that will be on the map.
* Sensors-ball1, ball2, ball3 will detect if there is some obstacles around the car. These are just to highlight the sensors on the car.
* The <game> object connects all the objects such as <car> & <ball1>, <ball2>, <ball3>.

1. **map.py** is where we make the whole map and also the whole game because you will see that we'll be playing some games with the car.

* We will give it some challenges like avoiding some obstacles or doing some round trips between two destinations on a more and more difficult road.
* So we will make a game and that game happens in this big file.
* We also connect the **ai.py** in this **map.py**. Next we'll understand the connection between the map and the AI.

**2.4 Step 1 : map.py (part 1)**

Let's describe the ***environment*** on which we'll implement our AI &the car that will train to drive itself and to avoid obstacles. Also we'll *draw some roads* and *some blocks* for our ***cars*** to ***navigate around them***.

* There are two separate **.py** files, in ***ai.py*** that will do all the training to train the car how to self drive. We're going to implement it line by line.
* ***map.py:*** The another file is map.py. It's actually 200 lines of code & little more. This code is not typically related to AI. It is just a code to make the environment make the map.
* We're *not going to implement this code line by line from scratch* because we want to focus on ***artificial intelligence***.
* But let's go through the sections of this ***map.py*** one by one to understand what's happening.

1. First we import the ***essential libraries*** and all the ***kivy*** ***packages***.

# *Importing the libraries*

**import** numpy **as** np

**from** random **import** random, randint

**import** matplotlib.pyplot **as** plt

**import** time

# *Importing the Kivy packages*

**from** kivy.app **import** App

**from** kivy.uix.widget **import** Widget

**from** kivy.uix.button **import** Button

**from** kivy.graphics **import** Color, Ellipse, Line

**from** kivy.config **import** Config

**from** kivy.properties **import** NumericProperty, ReferenceListProperty, ObjectProperty

**from** kivy.vector **import** Vector

**from** kivy.clock **import** Clock

1. Import car's brain. The brain of the car will be an object of this ***Dqn*** class. The ***Dqn*** class is our artificial intelligence itself. ***Dqn*** stands for deep Q networks.

* We'll implement this ***Dqn*** line by line in next sections, once it's ready it will be imported here with this line from the **ai.py** file.

# *Importing the Dqn object from our AI in ai.py*

**from** ai **import** Dqn

* **How we will train this car:** The idea of how we can train the car to drive itself and to avoid obstacles.
* In real life if you want to train a real car to avoid some walls or some obstacles, you would definitely not take real walls or real big obstacles and smash your car onto them. That would cost you a lot of money.
* Instead a more intelligent idea would be to punish your car when it goes onto (smash) some sand-barrier.
* It's like you have a ***field*** with some ***roads***, on which the car has to stay and the roads are delimited by some sand.
* And each time the car goes into the sand it's like it's going smash on to an obstacle.
* Once the car goes into some sand it will be slowed down and we will make sure that the car is penalized/punished for that.

It is one essential point of artificial intelligence. The bad reword comes whenever the car goes into some sands and is slowed down.

1. And therefore here we're introducing **last\_x** and **last\_y**. Which are the ***coordinates*** of the ***last point*** in ***memory*** when we ***draw*** some ***sand-barrier*** on the map.

# *Adding this line if we don't want the right click to put a red point*

**Config.set**('input', 'mouse', 'mouse,multitouch\_on\_demand')

# *Introducing last\_x and last\_y, used to keep the last point in memory when we draw the sand on the map*

last\_x = 0

last\_y = 0

n\_points = 0

length = 0

1. We get our artificial intelligence which we call brain and that contains our neural network and we will call it brain because this is actually the brain of the car and that contains our NN.

# *Getting our AI, which we call "brain", and that contains our neural network that represents our Q-function*

brain = **Dqn**(5,3,0.9)

action2rotation = [0,20,-20]

last\_reward = 0

scores = []

* **Dqn**(5,3,0.9)
* **5** corresponds to the states that are encoded vectors of five dimensions (5D, three signals and +orientation and -orientation), describing what's happening in the environment on the map.
* **3** is the number of actions there will be *three possible actions* go ***left*** go ***straight*** or go ***right***.
* **0.9** is a Gama parameter in the **Deep-Q-learning** algorithm.
* **action2rotation:** It is a vector of three elements (0, 20, -20).
* If the action that is selected at time **t** is **0**, **0** corresponds to the index of this **action2rotation** vector and the value is also 0. Therefore agent will go straight.
* If the action selected is **1**, then the value at index **1** of this **action2rotation** vector is **20**. 20 corresponds to a rotation of 20 degrees and that means the agent (car) will go 20 degrees to the right
* Similarly if action selected is **2**, then car will do a rotation of **- 20 degrees** and therefore it will go to the 20 degree left.
* **last\_reward:** At each stage car will be getting the last reward.
* If the car doesn't go into some sand-barrier then the reward will be positive.
* It the car hits the sand0barrier, it will get bad reward.
* **scores:** It is a vector that will contain the reward. Not all of them but onto a sliding window so that we can make a curve of the mean-square the reward with respect to time.

1. Initialize the map:

# *Initializing the map*

first\_update = **True**

**def** **init**():

**global** sand

**global** goal\_x

**global** goal\_y

**global** first\_update

    sand = **np.zeros**((longueur,largeur))

    goal\_x = 20

    goal\_y = largeur - 20

    first\_update = **False**

# *Initializing the last distance*

last\_distance = 0

* **sand** variable is actually going to be an array in which the cells will be the pixels of the map and in each cell we will have a **1** if there is sand and **0** if there is no sand. At the beginning we initialize the map with all 0's.
* **(goal\_x, goal\_y)** is a point in the map which will train the car to reach point. So it's like a destination.
* It's going to be the *upper left corner of the map* we'll train the car to go to the upper left corner of the map.
* And once it reaches the upper left corner of the map, then we will train it to go to the *bottom right corner of the map* so we can imagine the following scenario.
* The *upper left corner* of the map is the airport of a city and the *bottom right corner* of the map is the downtown of the city.
* And we will train a taxi to do some round trips between the airport and the downtown.
* We'll make the *task difficult* to this taxi by drawing some *difficult roads* and adding *more obstacles* on the street to see if the taxi can still manage to go from the airport to downtown.
* The map will be like a ***square*** and the ***origin*** is at ***bottom left corner***. So the coordinates **(20, largeur - 20)** will be the *upper left corner of the map*.
* We choose **20** and not **0** because *we want* to *train* the car *not to rush* into the *walls*. Therefore it's not be zero because we don't want the car to touch the wall, we want to touch the goal.
* **last\_distance:** It gives the current distance from the car to the goal and it initialized to zero.

**2.5 Step 2 : map.py (part 2)**

* car and the game: We're going to make *two classes* one class for the **car** and one class for the **game**. And inside these classes we'll make some connections with our AI.

**class** Car(Widget):

    angle = **NumericProperty**(0)

    rotation = **NumericProperty**(0)

    velocity\_x = **NumericProperty**(0)

    velocity\_y = **NumericProperty**(0)

    velocity = **ReferenceListProperty**(velocity\_x, velocity\_y)

    sensor1\_x = **NumericProperty**(0)

    sensor1\_y = **NumericProperty**(0)

    sensor1 = **ReferenceListProperty**(sensor1\_x, sensor1\_y)

    sensor2\_x = **NumericProperty**(0)

    sensor2\_y = **NumericProperty**(0)

    sensor2 = **ReferenceListProperty**(sensor2\_x, sensor2\_y)

    sensor3\_x = **NumericProperty**(0)

    sensor3\_y = **NumericProperty**(0)

    sensor3 = **ReferenceListProperty**(sensor3\_x, sensor3\_y)

    signal1 = **NumericProperty**(0)

    signal2 = **NumericProperty**(0)

    signal3 = **NumericProperty**(0)

**def** **move**(self, rotation):

        self.pos = **Vector**(\*self.velocity) + self.pos

        self.rotation = rotation

        self.angle = self.angle + self.rotation

        self.sensor1 = **Vector**(30, 0).**rotate**(self.angle) + self.pos

        self.sensor2 = **Vector**(30, 0).**rotate**((self.angle+30)%360) + self.pos

        self.sensor3 = **Vector**(30, 0).**rotate**((self.angle-30)%360) + self.pos

        self.signal1 = **int**(**np.sum**(sand[**int**(self.sensor1\_x)-10:**int**(self.sensor1\_x)+10, **int**(self.sensor1\_y)-10:**int**(self.sensor1\_y)+10]))/400.

        self.signal2 = **int**(**np.sum**(sand[**int**(self.sensor2\_x)-10:**int**(self.sensor2\_x)+10, **int**(self.sensor2\_y)-10:**int**(self.sensor2\_y)+10]))/400.

        self.signal3 = **int**(**np.sum**(sand[**int**(self.sensor3\_x)-10:**int**(self.sensor3\_x)+10, **int**(self.sensor3\_y)-10:**int**(self.sensor3\_y)+10]))/400.

**if** self.sensor1\_x**>**longueur-10 **or** self.sensor1\_x**<**10 **or** self.sensor1\_y**>**largeur-10 **or** self.sensor1\_y**<**10:

            self.signal1 = 1.

**if** self.sensor2\_x**>**longueur-10 **or** self.sensor2\_x**<**10 **or** self.sensor2\_y**>**largeur-10 **or** self.sensor2\_y**<**10:

            self.signal2 = 1.

**if** self.sensor3\_x**>**longueur-10 **or** self.sensor3\_x**<**10 **or** self.sensor3\_y**>**largeur-10 **or** self.sensor3\_y**<**10:

            self.signal3 = 1.

* Car class: We create the car using the *car class*. The class have a *lot of properties* some *variables*, some *functions* that will make the car move to the left or right or going straight.
* angle: is the angle between the x axis and the axis of the direction of the car.
* rotation: last rotation, it is either **0** degree or **20** degrees or **- 20** degrees.
* Velocity: the x coordinate and the y coordinates of the velocity vector.
* Sensors: Then we have the sensors and the signals. Those are very important, the car that we're making will have three sensors

sensor\_1 : detect if there is any sand-barrier in front of the car.,

sensor\_2 : detect if there is any sand-barrier at the left of the car

sensor\_3 : detect if there is any sand-barrier at the right of the car

* Signals: From these three sensors we get the signals received by each of the sensors. These signals are signal\_1, signal\_2, signal\_3 from the sensors 1, 2, 3.

signal\_1, signal\_2, signal\_3 measures the density of sand around sensor\_1, sensor\_2, sensor\_3.

* How do we compute this density of sand:
* We take some big squares around each of the sensors.
* These are actually squares of 200 by 200 and for each of the squares we ***divide*** the ***number of 1's*** in the square by the ***total number of cells*** in the square that is .
* And that gives us the density of sand because the **1's** correspond to the **sand**.
* We do this for each sensor and that gives us the density of sand around each sensor (i.e. signals).
* Move Function: It will allow the **car** to go to the **left/straight/right**.
* First we update of the position of the car with its last position **self.pos** and the velocity vector. The position will be updated in the direction of the velocity vector.
* Then we apply the rotation which is defined in the **update()** function in the **Game** class. It's used to rotate the car to the left or to the right.
* Then we update the angle (the angle between the x-axis and the axis of the direction of the car).
* Once the car has moved we have to update the sensors and the signal because the sensors have rotated as well.
* Why do we have this vector of **Vector(30, 0)**.
  + We used that because the distance between the car and the sensor.
  + It's is the distance between the car and what the car detects.
* Next we update the signals to compute the signals, we get the x-coordinates of our sensor.
* We take all the cells from **-10** to **+10** then we do the same for the y coordinate taking all the cells from **-10** to **+10** to get the square of pixels surrounding the sensor.
* Inside the square we some All the 1's.
* Basically we'll sum all the cells because the cells contain 0 or 1.
* Since in a square there is 400 cells, we'll divide it by 400 to get the density of 1's inside the square.

That's how we get the signal of the density of sands around the sensor.

We do the same for the second sensor and the third sensor to get the second signal and the third signal.

* Getting rewards (3 if statements): These 3 if-conditions are very important.
* It's another bad-reward for the car when it's reaching one of the edges of the map.
* Because, we ***don't want*** the car to ***rush into some walls*** and therefore we want to penalize it when it's getting too close to Wall.

self.sensor1\_x>longueur-10 : i.e. closer to the right edge of the map

self.sensor1\_x<10 : i.e. closer to the left edge of the map

self.sensor1\_y>largeur-10 : i.e. closer to upper edge of the map

self.sensor1\_y<10: i.e. closer to lower edge of the map

* If the **sensor1** is reaching any of these four edges we set the signal for this sensor i.e. **signal1** the value **1**.
* It means full sand i.e. the full density of sand. There's so much sands that it's going to stop your car. It's a *terribly* *bad reward*.

And then we do the same for signal2 and signals3.

* Game class: Basically it's the class to create the game. We have only created the car and now of course we have to create the map.
* We have to create the game itself.
* We'll not be playing the game; it's our AI that will be playing the game. The game is actually *to avoid the obstacles and to go from the airport to downtown* and vice versa.

**class** Game(Widget):

    car = **ObjectProperty**(**None**)

    ball1 = **ObjectProperty**(**None**)

    ball2 = **ObjectProperty**(**None**)

    ball3 = **ObjectProperty**(**None**)

**def** **serve\_car**(self):

        self.car.center = self.center

        self.car.velocity = **Vector**(6, 0)

**def** **update**(self, dt):

**global** brain

**global** last\_reward

**global** scores

**global** last\_distance

**global** goal\_x

**global** goal\_y

**global** longueur

**global** largeur

        longueur = self.width

        largeur = self.height

**if** first\_update:

**init**()

        xx = goal\_x - self.car.x

        yy = goal\_y - self.car.y

        orientation = **Vector**(\*self.car.velocity).**angle**((xx,yy))/180.

        last\_signal = [self.car.signal1, self.car.signal2, self.car.signal3, orientation, -orientation]

        action = **brain.update**(last\_reward, last\_signal)

**scores.append**(**brain.score**())

        rotation = action2rotation[action]

        self**.car.move**(rotation)

        distance = **np.sqrt**((self.car.x - goal\_x)\*\*2 + (self.car.y - goal\_y)\*\*2)

        self.ball1.pos = self.car.sensor1

        self.ball2.pos = self.car.sensor2

        self.ball3.pos = self.car.sensor3

**if** sand[**int**(self.car.x),**int**(self.car.y)] **>** 0:

            self.car.velocity = **Vector**(1, 0).**rotate**(self.car.angle)

            last\_reward = -1

**else**: # *otherwise*

            self.car.velocity = **Vector**(6, 0).**rotate**(self.car.angle)

            last\_reward = -0.2

**if** distance **<** last\_distance:

                last\_reward = 0.1

**if** self.car.x **<** 10:

            self.car.x = 10

            last\_reward = -1

**if** self.car.x **>** self.width - 10:

            self.car.x = self.width - 10

            last\_reward = -1

**if** self.car.y **<** 10:

            self.car.y = 10

            last\_reward = -1

**if** self.car.y **>** self.height - 10:

            self.car.y = self.height - 10

            last\_reward = -1

**if** distance **<** 100:

            goal\_x = self.width-goal\_x

            goal\_y = self.height-goal\_y

        last\_distance = distance

* So in this **Game** class we need to create some objects such as: *car*, *ball1*, *ball2* etc.
* Then we need to define the update function. It is the most important function because it will select the action of the car each time to accomplish its goal.
* This action is exactly the output of our neural network of artificial intelligence.

action = **brain.update**(last\_reward, last\_signal)

* **brain** is an object of **Dqn** class from our **ai.py** file.
* And this object has a method that is called update and it takes as input the last\_reward (last reward obtained by the car) and the last\_signal (signal\_1, signal\_2, signal\_3 from sensor\_1, sensor\_2, sensor\_3 and orientation, -orientation).
* In the last\_signal vector we have *five inputs*:
* First three are signal\_1, signal\_2, signal\_3 from sensor\_1, sensor\_2, sensor\_3.
* **orientation*:*** orientation of the car with respect to the goal.
* If the car is heading towards the goal then the orientation will be equal to zero.
* If it goes *slightly to the* right then the orientation will be close to **45** degrees.
* If it goes *slightly to the* left the orientation will be close to **-45** degrees.
* **-orientation*:*** Usually the ***inputs*** of ***neural network*** are independent. There is no ***multi-collinearity*** but it doesn't really matter if we add this is because the neural network will just fix that with the weights.
* But still I notice that by adding this ***-orientation*** will allow the training of the car to stabilize the exploration.
* AI doesn't always explore in the same direction, by adding this ***-orientation*** we make sure that it *explores in both directions* ***right*** or ***left***.
* So this vector :

last\_signal = [self.car.signal1, self.car.signal2, self.car.signal3, orientation, -orientation]

containing *three signals* and the *orientation*, *-orientation* are the five inputs of our encoded vector which will go into the network with *last\_reward* (because the action to play also depends on the last reword).

* Then the network will return the outputs- the action to play at each time. The output returned by this update function contains the network itself and the output of the network.
* Then we update the mean score of rewords,
* We update the rotation, we then use the move function to rotate the car according to the action that was selected.
* We update the distance of the car to the goal and we update the positions of the sensors.
* **ball1**, **ball2**, **ball3** will represent the sensors on the map.
* Next, in all the if-statements we penalize the car if it goes into some sands.
* **First If-else:**The *velocity* is *reduced* (*velocity* is usually **6**), if it goes on to some *sands* it will be *slowed down* to **1**.
* Also it'll get a *-1* reward (bad/worst reward).
* If it stays *further from the sand* it's velocity won't change (remains 6). If the car ***moves toward the goal*** a *slightly* **+ve** *reward* **0.1** is given ***if it getting further away from the goal*** a *slightly* **–ve** *reward* **-0.2** is given.
* In **next 4 if-condition** penalization is applied if the car hits the edges of the map.
* In the last if-condition is to update the goal, when the goal is reached.
* So you know when the car reaches the Airport which is the first goal (the upper left corner of the map), the goal changes to the bottom right corner of the map which is Downtown.
* So we update the x, y coordinate of the goal to set a new goal.

# *Creating the car class*

**class** Ball1(Widget):

**pass**

**class** Ball2(Widget):

**pass**

**class** Ball3(Widget):

**pass**

# *Creating the game class*

# *Adding the painting tools*

**class** MyPaintWidget(Widget):

**def** **on\_touch\_down**(self, touch):

**global** length, n\_points, last\_x, last\_y

**with** self.canvas:

**Color**(0.8,0.7,0)

            d = 10.

            touch.ud['line'] = **Line**(points = (touch.x, touch.y), width = 10)

            last\_x = **int**(touch.x)

            last\_y = **int**(touch.y)

            n\_points = 0

            length = 0

            sand[**int**(touch.x),**int**(touch.y)] = 1

**def** **on\_touch\_move**(self, touch):

**global** length, n\_points, last\_x, last\_y

**if** touch.button **==** 'left':

            touch.ud['line'].points += [touch.x, touch.y]

            x = **int**(touch.x)

            y = **int**(touch.y)

            length += **np.sqrt**(**max**((x - last\_x)\*\*2 + (y - last\_y)\*\*2, 2))

            n\_points += 1.

            density = n\_points/(length)

            touch.ud['line'].width = **int**(20 \* density + 1)

            sand[**int**(touch.x) - 10 : **int**(touch.x) + 10, **int**(touch.y) - 10 : **int**(touch.y) + 10] = 1

            last\_x = x

            last\_y = y

# *Adding the API Buttons (clear, save and load)*

**class** CarApp(App):

**def** **build**(self):

        parent = **Game**()

**parent.serve\_car**()

**Clock.schedule\_interval**(parent.update, 1.0/60.0)

        self.painter = **MyPaintWidget**()

        clearbtn = **Button**(text = 'clear')

        savebtn = **Button**(text = 'save', pos = (parent.width, 0))

        loadbtn = **Button**(text = 'load', pos = (2 \* parent.width, 0))

**clearbtn.bind**(on\_release = self.clear\_canvas)

**savebtn.bind**(on\_release = self.save)

**loadbtn.bind**(on\_release = self.load)

**parent.add\_widget**(self.painter)

**parent.add\_widget**(clearbtn)

**parent.add\_widget**(savebtn)

**parent.add\_widget**(loadbtn)

**return** parent

**def** **clear\_canvas**(self, obj):

**global** sand

        self**.painter.canvas.clear**()

        sand = **np.zeros**((longueur,largeur))

**def** **save**(self, obj):

**print**("saving brain...")

**brain.save**()

**plt.plot**(scores)

**plt.show**()

**def** **load**(self, obj):

**print**("loading last saved brain...")

**brain.load**()

# *Running the whole thing*

**if** \_\_name\_\_ **==** '\_\_main\_\_':

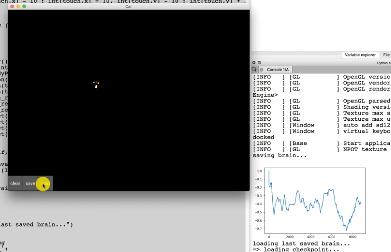
**CarApp**().**run**()

* MyPaintWidget class: This class is less important.
* That's just a class that will add the painting tools, to paint some roads or some obstacles on the map. It's more related to Kivy.
* CarApp class: Also related to kivy.
* It's adds the API buttons **clear\_map**, **save** and **load**.
* **Save** is for saving the AI-brain.
* **Load** is used to load a saved AI-brain, to load the *memory* of the car how to *navigate* in the *map*.
* And then finally we have the last code section which runs the whole thing i.e. runs the map and the AI itself.

# *Running the whole thing*

**if** \_\_name\_\_ **==** '\_\_main\_\_':

**CarApp**().**run**()



**2.6 Step 3 : ai.py (import libraries)**

* Connection between **map.py** and **ai.py**:
* The purpose of **ai.py** is to select the right action at each time.
* In **ai.py** we define the **Dqn** class.
* In **map.py** we import the **Dqn** class.

# *Importing the Dqn object from our AI in ai.py*

**from** ai **import** Dqn

* We create the brain by creating an ***object*** of this ***Dqn*** class

brain = **Dqn**(5,3,0.9)

* **5** corresponds to the states that are encoded vectors of five dimensions (5D, three signals and +orientation and -orientation), describing what's happening in the environment on the map.
* **3** is the number of actions there will be *three possible actions* go ***left*** go ***straight*** or go ***right***.
* **0.9** is a **Gamma** parameter in the **Deep-Q-learning** algorithm. That's the only parameters of the **Dqn** class that we'll be making.
* Once we create that object we select in the **Game** class then action to play at each time, which depends on the last\_reward and last\_signal. In ***map.py*** we select the *right action to play* at each time by executing this line:

action = **brain.update**(last\_reward, last\_signal)

* last\_reward and last\_signal will be the input of the NN, last\_signal composed of three signals and +orientation and -orientation.

That's the only purpose of making this ***ai.py*** in order to have a *real artificial intelligence* playing the *right actions* then each time *instead* of having *random actions*.

# *-------------    Deep Q-learning (Dqn)    -------------*

# *AI for Self Driving Car*

# *Importing the libraries*

**import** numpy **as** np

**import** random

**import** os

**import** torch

**import** torch.nn **as** nn

**import** torch.nn.functional **as** F

**import** torch.optim **as** optim

**import** torch.autograd **as** autograd

**from** torch.autograd **import** Variable

1. **numpy** used to work with arrays.
2. **random** used because we will be taking some ***random samples*** from ***different batches*** when implementing **experience replay**.
3. **os** is useful to load/save the brain to re-use the trained brain anytime.
4. **torch** library is essential to handle ***dynamic graphs***, to build the NN of the AI.
5. **torch.nn** contains all the tools to implement some neural networks.

* The NN takes 3-**signals** of the three sensors, '**+orientation**' and '**-orientation**' as input and return as output the action to play (it return the q-values of the different actions and using a Soft-Max, we'll return the ***only one most relevant action to play)***.

1. **torch.nn.functional** has different functions that we use when implementing a neural network.

* For example: for loss function we'll use HUBER\_LOSS (because that improves convergence).

1. **torch.optim** for the optimizer to perform stochastic gradient descent.
2. **torch.autograd** and its Variable class needed to convert tensors into the ***tensors that contains also a gradient***.

* So we put the tensor into a variable that will also contain a gradient and to do this we need to use the variable class.

All code at once: map.py

# *Self Driving Car*

# *Importing the libraries*

**import** numpy **as** np

**from** random **import** random, randint

**import** matplotlib.pyplot **as** plt

**import** time

# *Importing the Kivy packages*

**from** kivy.app **import** App

**from** kivy.uix.widget **import** Widget

**from** kivy.uix.button **import** Button

**from** kivy.graphics **import** Color, Ellipse, Line

**from** kivy.config **import** Config

**from** kivy.properties **import** NumericProperty, ReferenceListProperty, ObjectProperty

**from** kivy.vector **import** Vector

**from** kivy.clock **import** Clock

# *Importing the Dqn object from our AI in ai.py*

**from** ai **import** Dqn

# *Adding this line if we don't want the right click to put a red point*

**Config.set**('input', 'mouse', 'mouse,multitouch\_on\_demand')

# *Introducing last\_x and last\_y, used to keep the last point in memory when we draw the sand on the map*

last\_x = 0

last\_y = 0

n\_points = 0

length = 0

# *Getting our AI, which we call "brain", and that contains our neural network that represents our Q-function*

brain = **Dqn**(5,3,0.9)

action2rotation = [0,20,-20]

last\_reward = 0

scores = []

# *Initializing the map*

first\_update = **True**

**def** **init**():

**global** sand

**global** goal\_x

**global** goal\_y

**global** first\_update

    sand = **np.zeros**((longueur,largeur))

    goal\_x = 20

    goal\_y = largeur - 20

    first\_update = **False**

# *Initializing the last distance*

last\_distance = 0

# *Creating the car class*

**class** Car(Widget):

    angle = **NumericProperty**(0)

    rotation = **NumericProperty**(0)

    velocity\_x = **NumericProperty**(0)

    velocity\_y = **NumericProperty**(0)

    velocity = **ReferenceListProperty**(velocity\_x, velocity\_y)

    sensor1\_x = **NumericProperty**(0)

    sensor1\_y = **NumericProperty**(0)

    sensor1 = **ReferenceListProperty**(sensor1\_x, sensor1\_y)

    sensor2\_x = **NumericProperty**(0)

    sensor2\_y = **NumericProperty**(0)

    sensor2 = **ReferenceListProperty**(sensor2\_x, sensor2\_y)

    sensor3\_x = **NumericProperty**(0)

    sensor3\_y = **NumericProperty**(0)

    sensor3 = **ReferenceListProperty**(sensor3\_x, sensor3\_y)

    signal1 = **NumericProperty**(0)

    signal2 = **NumericProperty**(0)

    signal3 = **NumericProperty**(0)

**def** **move**(self, rotation):

        self.pos = **Vector**(\*self.velocity) + self.pos

        self.rotation = rotation

        self.angle = self.angle + self.rotation

        self.sensor1 = **Vector**(30, 0).**rotate**(self.angle) + self.pos

        self.sensor2 = **Vector**(30, 0).**rotate**((self.angle+30)%360) + self.pos

        self.sensor3 = **Vector**(30, 0).**rotate**((self.angle-30)%360) + self.pos

        self.signal1 = **int**(**np.sum**(sand[**int**(self.sensor1\_x)-10:**int**(self.sensor1\_x)+10, **int**(self.sensor1\_y)-10:**int**(self.sensor1\_y)+10]))/400.

        self.signal2 = **int**(**np.sum**(sand[**int**(self.sensor2\_x)-10:**int**(self.sensor2\_x)+10, **int**(self.sensor2\_y)-10:**int**(self.sensor2\_y)+10]))/400.

        self.signal3 = **int**(**np.sum**(sand[**int**(self.sensor3\_x)-10:**int**(self.sensor3\_x)+10, **int**(self.sensor3\_y)-10:**int**(self.sensor3\_y)+10]))/400.

**if** self.sensor1\_x**>**longueur-10 **or** self.sensor1\_x**<**10 **or** self.sensor1\_y**>**largeur-10 **or** self.sensor1\_y**<**10:

            self.signal1 = 1.

**if** self.sensor2\_x**>**longueur-10 **or** self.sensor2\_x**<**10 **or** self.sensor2\_y**>**largeur-10 **or** self.sensor2\_y**<**10:

            self.signal2 = 1.

**if** self.sensor3\_x**>**longueur-10 **or** self.sensor3\_x**<**10 **or** self.sensor3\_y**>**largeur-10 **or** self.sensor3\_y**<**10:

            self.signal3 = 1.

**class** Ball1(Widget):

**pass**

**class** Ball2(Widget):

**pass**

**class** Ball3(Widget):

**pass**

# *Creating the game class*

**class** Game(Widget):

    car = **ObjectProperty**(**None**)

    ball1 = **ObjectProperty**(**None**)

    ball2 = **ObjectProperty**(**None**)

    ball3 = **ObjectProperty**(**None**)

**def** **serve\_car**(self):

        self.car.center = self.center

        self.car.velocity = **Vector**(6, 0)

**def** **update**(self, dt):

**global** brain

**global** last\_reward

**global** scores

**global** last\_distance

**global** goal\_x

**global** goal\_y

**global** longueur

**global** largeur

        longueur = self.width

        largeur = self.height

**if** first\_update:

**init**()

        xx = goal\_x - self.car.x

        yy = goal\_y - self.car.y

        orientation = **Vector**(\*self.car.velocity).**angle**((xx,yy))/180.

        last\_signal = [self.car.signal1, self.car.signal2, self.car.signal3, orientation, -orientation]

        action = **brain.update**(last\_reward, last\_signal)

**scores.append**(**brain.score**())

        rotation = action2rotation[action]

        self**.car.move**(rotation)

        distance = **np.sqrt**((self.car.x - goal\_x)\*\*2 + (self.car.y - goal\_y)\*\*2)

        self.ball1.pos = self.car.sensor1

        self.ball2.pos = self.car.sensor2

        self.ball3.pos = self.car.sensor3

**if** sand[**int**(self.car.x),**int**(self.car.y)] **>** 0:

            self.car.velocity = **Vector**(1, 0).**rotate**(self.car.angle)

            last\_reward = -1

**else**: # *otherwise*

            self.car.velocity = **Vector**(6, 0).**rotate**(self.car.angle)

            last\_reward = -0.2

**if** distance **<** last\_distance:

                last\_reward = 0.1

**if** self.car.x **<** 10:

            self.car.x = 10

            last\_reward = -1

**if** self.car.x **>** self.width - 10:

            self.car.x = self.width - 10

            last\_reward = -1

**if** self.car.y **<** 10:

            self.car.y = 10

            last\_reward = -1

**if** self.car.y **>** self.height - 10:

            self.car.y = self.height - 10

            last\_reward = -1

**if** distance **<** 100:

            goal\_x = self.width-goal\_x

            goal\_y = self.height-goal\_y

        last\_distance = distance

# *Adding the painting tools*

**class** MyPaintWidget(Widget):

**def** **on\_touch\_down**(self, touch):

**global** length, n\_points, last\_x, last\_y

**with** self.canvas:

**Color**(0.8,0.7,0)

            d = 10.

            touch.ud['line'] = **Line**(points = (touch.x, touch.y), width = 10)

            last\_x = **int**(touch.x)

            last\_y = **int**(touch.y)

            n\_points = 0

            length = 0

            sand[**int**(touch.x),**int**(touch.y)] = 1

**def** **on\_touch\_move**(self, touch):

**global** length, n\_points, last\_x, last\_y

**if** touch.button **==** 'left':

            touch.ud['line'].points += [touch.x, touch.y]

            x = **int**(touch.x)

            y = **int**(touch.y)

            length += **np.sqrt**(**max**((x - last\_x)\*\*2 + (y - last\_y)\*\*2, 2))

            n\_points += 1.

            density = n\_points/(length)

            touch.ud['line'].width = **int**(20 \* density + 1)

            sand[**int**(touch.x) - 10 : **int**(touch.x) + 10, **int**(touch.y) - 10 : **int**(touch.y) + 10] = 1

            last\_x = x

            last\_y = y

# *Adding the API Buttons (clear, save and load)*

**class** CarApp(App):

**def** **build**(self):

        parent = **Game**()

**parent.serve\_car**()

**Clock.schedule\_interval**(parent.update, 1.0/60.0)

        self.painter = **MyPaintWidget**()

        clearbtn = **Button**(text = 'clear')

        savebtn = **Button**(text = 'save', pos = (parent.width, 0))

        loadbtn = **Button**(text = 'load', pos = (2 \* parent.width, 0))

**clearbtn.bind**(on\_release = self.clear\_canvas)

**savebtn.bind**(on\_release = self.save)

**loadbtn.bind**(on\_release = self.load)

**parent.add\_widget**(self.painter)

**parent.add\_widget**(clearbtn)

**parent.add\_widget**(savebtn)

**parent.add\_widget**(loadbtn)

**return** parent

**def** **clear\_canvas**(self, obj):

**global** sand

        self**.painter.canvas.clear**()

        sand = **np.zeros**((longueur,largeur))

**def** **save**(self, obj):

**print**("saving brain...")

**brain.save**()

**plt.plot**(scores)

**plt.show**()

**def** **load**(self, obj):

**print**("loading last saved brain...")

**brain.load**()

# *Running the whole thing*

**if** \_\_name\_\_ **==** '\_\_main\_\_':

**CarApp**().**run**()

Commented

# *Self Driving Car*

# *Importing the libraries*

**import** numpy **as** np

**from** random **import** random, randint

**import** matplotlib.pyplot **as** plt

**import** time

# *Importing the Kivy packages*

**from** kivy.app **import** App

**from** kivy.uix.widget **import** Widget

**from** kivy.uix.button **import** Button

**from** kivy.graphics **import** Color, Ellipse, Line

**from** kivy.config **import** Config

**from** kivy.properties **import** NumericProperty, ReferenceListProperty, ObjectProperty

**from** kivy.vector **import** Vector

**from** kivy.clock **import** Clock

# *Importing the Dqn object from our AI in ia.py*

**from** ai **import** Dqn

# *Adding this line if we don't want the right click to put a red point*

**Config.set**('input', 'mouse', 'mouse,multitouch\_on\_demand')

# *Introducing last\_x and last\_y, used to keep the last point in memory when we draw the sand on the map*

last\_x = 0

last\_y = 0

n\_points = 0 # *the total number of points in the last drawing*

length = 0 # *the length of the last drawing*

# *Getting our AI, which we call "brain", and that contains our neural network that represents our Q-function*

brain = **Dqn**(5,3,0.9) # *5 sensors, 3 actions, gama = 0.9*

action2rotation = [0,20,-20] # *action = 0 => no rotation, action = 1 => rotate 20 degres, action = 2 => rotate -20 degres*

last\_reward = 0 # *initializing the last reward*

scores = [] # *initializing the mean score curve (sliding window of the rewards) with respect to time*

# *Initializing the map*

first\_update = **True** # *using this trick to initialize the map only once*

**def** **init**():

**global** sand # *sand is an array that has as many cells as our graphic interface has pixels. Each cell has a one if there is sand, 0 otherwise.*

**global** goal\_x # *x-coordinate of the goal (where the car has to go, that is the airport or the downtown)*

**global** goal\_y # *y-coordinate of the goal (where the car has to go, that is the airport or the downtown)*

    sand = **np.zeros**((longueur,largeur)) # *initializing the sand array with only zeros*

    goal\_x = 20 # *the goal to reach is at the upper left of the map (the x-coordinate is 20 and not 0 because the car gets bad reward if it touches the wall)*

    goal\_y = largeur - 20 # *the goal to reach is at the upper left of the map (y-coordinate)*

    first\_update = **False** # *trick to initialize the map only once*

# *Initializing the last distance*

last\_distance = 0

# *Creating the car class (to understand "NumericProperty" and "ReferenceListProperty", see kivy tutorials: https://kivy.org/docs/tutorials/pong.html)*

**class** Car(Widget):

    angle = **NumericProperty**(0) # *initializing the angle of the car (angle between the x-axis of the map and the axis of the car)*

    rotation = **NumericProperty**(0) # *initializing the last rotation of the car (after playing the action, the car does a rotation of 0, 20 or -20 degrees)*

    velocity\_x = **NumericProperty**(0) # *initializing the x-coordinate of the velocity vector*

    velocity\_y = **NumericProperty**(0) # *initializing the y-coordinate of the velocity vector*

    velocity = **ReferenceListProperty**(velocity\_x, velocity\_y) # *velocity vector*

    sensor1\_x = **NumericProperty**(0) # *initializing the x-coordinate of the first sensor (the one that looks forward)*

    sensor1\_y = **NumericProperty**(0) # *initializing the y-coordinate of the first sensor (the one that looks forward)*

    sensor1 = **ReferenceListProperty**(sensor1\_x, sensor1\_y) # *first sensor vector*

    sensor2\_x = **NumericProperty**(0) # *initializing the x-coordinate of the second sensor (the one that looks 30 degrees to the left)*

    sensor2\_y = **NumericProperty**(0) # *initializing the y-coordinate of the second sensor (the one that looks 30 degrees to the left)*

    sensor2 = **ReferenceListProperty**(sensor2\_x, sensor2\_y) # *second sensor vector*

    sensor3\_x = **NumericProperty**(0) # *initializing the x-coordinate of the third sensor (the one that looks 30 degrees to the right)*

    sensor3\_y = **NumericProperty**(0) # *initializing the y-coordinate of the third sensor (the one that looks 30 degrees to the right)*

    sensor3 = **ReferenceListProperty**(sensor3\_x, sensor3\_y) # *third sensor vector*

    signal1 = **NumericProperty**(0) # *initializing the signal received by sensor 1*

    signal2 = **NumericProperty**(0) # *initializing the signal received by sensor 2*

    signal3 = **NumericProperty**(0) # *initializing the signal received by sensor 3*

**def** **move**(self, rotation):

        self.pos = **Vector**(\*self.velocity) + self.pos # *updating the position of the car according to its last position and velocity*

        self.rotation = rotation # *getting the rotation of the car*

        self.angle = self.angle + self.rotation # *updating the angle*

        self.sensor1 = **Vector**(30, 0).**rotate**(self.angle) + self.pos # *updating the position of sensor 1*

        self.sensor2 = **Vector**(30, 0).**rotate**((self.angle+30)%360) + self.pos # *updating the position of sensor 2*

        self.sensor3 = **Vector**(30, 0).**rotate**((self.angle-30)%360) + self.pos # *updating the position of sensor 3*

        self.signal1 = **int**(**np.sum**(sand[**int**(self.sensor1\_x)-10:**int**(self.sensor1\_x)+10, **int**(self.sensor1\_y)-10:**int**(self.sensor1\_y)+10]))/400. # *getting the signal received by sensor 1 (density of sand around sensor 1)*

        self.signal2 = **int**(**np.sum**(sand[**int**(self.sensor2\_x)-10:**int**(self.sensor2\_x)+10, **int**(self.sensor2\_y)-10:**int**(self.sensor2\_y)+10]))/400. # *getting the signal received by sensor 2 (density of sand around sensor 2)*

        self.signal3 = **int**(**np.sum**(sand[**int**(self.sensor3\_x)-10:**int**(self.sensor3\_x)+10, **int**(self.sensor3\_y)-10:**int**(self.sensor3\_y)+10]))/400. # *getting the signal received by sensor 3 (density of sand around sensor 3)*

**if** self.sensor1\_x **>** longueur-10 **or** self.sensor1\_x**<**10 **or** self.sensor1\_y**>**largeur-10 **or** self.sensor1\_y**<**10: # *if sensor 1 is out of the map (the car is facing one edge of the map)*

            self.signal1 = 1. # *sensor 1 detects full sand*

**if** self.sensor2\_x **>** longueur-10 **or** self.sensor2\_x**<**10 **or** self.sensor2\_y**>**largeur-10 **or** self.sensor2\_y**<**10: # *if sensor 2 is out of the map (the car is facing one edge of the map)*

            self.signal2 = 1. # *sensor 2 detects full sand*

**if** self.sensor3\_x **>** longueur-10 **or** self.sensor3\_x**<**10 **or** self.sensor3\_y**>**largeur-10 **or** self.sensor3\_y**<**10: # *if sensor 3 is out of the map (the car is facing one edge of the map)*

            self.signal3 = 1. # *sensor 3 detects full sand*

**class** Ball1(Widget): # *sensor 1 (see kivy tutorials: kivy https://kivy.org/docs/tutorials/pong.html)*

**pass**

**class** Ball2(Widget): # *sensor 2 (see kivy tutorials: kivy https://kivy.org/docs/tutorials/pong.html)*

**pass**

**class** Ball3(Widget): # *sensor 3 (see kivy tutorials: kivy https://kivy.org/docs/tutorials/pong.html)*

**pass**

# *Creating the game class (to understand "ObjectProperty", see kivy tutorials: kivy https://kivy.org/docs/tutorials/pong.html)*

**class** Game(Widget):

    car = **ObjectProperty**(**None**) # *getting the car object from our kivy file*

    ball1 = **ObjectProperty**(**None**) # *getting the sensor 1 object from our kivy file*

    ball2 = **ObjectProperty**(**None**) # *getting the sensor 2 object from our kivy file*

    ball3 = **ObjectProperty**(**None**) # *getting the sensor 3 object from our kivy file*

**def** **serve\_car**(self): # *starting the car when we launch the application*

        self.car.center = self.center # *the car will start at the center of the map*

        self.car.velocity = **Vector**(6, 0) # *the car will start to go horizontally to the right with a speed of 6*

**def** **update**(self, dt): # *the big update function that updates everything that needs to be updated at each discrete time t when reaching a new state (getting new signals from the sensors)*

**global** brain # *specifying the global variables (the brain of the car, that is our AI)*

**global** last\_reward # *specifying the global variables (the last reward)*

**global** scores # *specifying the global variables (the means of the rewards)*

**global** last\_distance # *specifying the global variables (the last distance from the car to the goal)*

**global** goal\_x # *specifying the global variables (x-coordinate of the goal)*

**global** goal\_y # *specifying the global variables (y-coordinate of the goal)*

**global** longueur # *specifying the global variables (width of the map)*

**global** largeur # *specifying the global variables (height of the map)*

        longueur = self.width # *width of the map (horizontal edge)*

        largeur = self.height # *height of the map (vertical edge)*

**if** first\_update: # *trick to initialize the map only once*

**init**()

        xx = goal\_x - self.car.x # *difference of x-coordinates between the goal and the car*

        yy = goal\_y - self.car.y # *difference of y-coordinates between the goal and the car*

        orientation = **Vector**(\*self.car.velocity).**angle**((xx,yy))/180. # *direction of the car with respect to the goal (if the car is heading perfectly towards the goal, then orientation = 0)*

        last\_signal = [self.car.signal1, self.car.signal2, self.car.signal3, orientation, -orientation] # *our input state vector, composed of the three signals received by the three sensors, plus the orientation and -orientation*

        action = **brain.update**(last\_reward, last\_signal) # *playing the action from our ai (the object brain of the dqn class)*

**scores.append**(**brain.score**()) # *appending the score (mean of the last 100 rewards to the reward window)*

        rotation = action2rotation[action] # *converting the action played (0, 1 or 2) into the rotation angle (0°, 20° or -20°)*

        self**.car.move**(rotation) # *moving the car according to this last rotation angle*

        distance = **np.sqrt**((self.car.x - goal\_x)\*\*2 + (self.car.y - goal\_y)\*\*2) # *getting the new distance between the car and the goal right after the car moved*

        self.ball1.pos = self.car.sensor1 # *updating the position of the first sensor (ball1) right after the car moved*

        self.ball2.pos = self.car.sensor2 # *updating the position of the second sensor (ball2) right after the car moved*

        self.ball3.pos = self.car.sensor3 # *updating the position of the third sensor (ball3) right after the car moved*

**if** sand[**int**(self.car.x),**int**(self.car.y)] **>** 0: # *if the car is on the sand*

            self.car.velocity = **Vector**(1, 0).**rotate**(self.car.angle) # *it is slowed down (speed = 1)*

            last\_reward = -1 # *and reward = -1*

**else**: # *otherwise*

            self.car.velocity = **Vector**(6, 0).**rotate**(self.car.angle) # *it goes to a normal speed (speed = 6)*

            last\_reward = -0.2 # *and it gets bad reward (-0.2)*

**if** distance **<** last\_distance: # *however if it getting close to the goal*

                last\_reward = 0.1 # *it still gets slightly positive reward 0.1*

**if** self.car.x **<** 10: # *if the car is in the left edge of the frame*

            self.car.x = 10 # *it is not slowed down*

            last\_reward = -1 # *but it gets bad reward -1*

**if** self.car.x **>** self.width-10: # *if the car is in the right edge of the frame*

            self.car.x = self.width-10 # *it is not slowed down*

            last\_reward = -1 # *but it gets bad reward -1*

**if** self.car.y **<** 10: # *if the car is in the bottom edge of the frame*

            self.car.y = 10 # *it is not slowed down*

            last\_reward = -1 # *but it gets bad reward -1*

**if** self.car.y **>** self.height-10: # *if the car is in the upper edge of the frame*

            self.car.y = self.height-10 # *it is not slowed down*

            last\_reward = -1 # *but it gets bad reward -1*

**if** distance **<** 100: # *when the car reaches its goal*

            goal\_x = self.width - goal\_x # *the goal becomes the bottom right corner of the map (the downtown), and vice versa (updating of the x-coordinate of the goal)*

            goal\_y = self.height - goal\_y # *the goal becomes the bottom right corner of the map (the downtown), and vice versa (updating of the y-coordinate of the goal)*

        # *Updating the last distance from the car to the goal*

        last\_distance = distance

# *Painting for graphic interface (see kivy tutorials: https://kivy.org/docs/tutorials/firstwidget.html)*

**class** MyPaintWidget(Widget):

**def** **on\_touch\_down**(self, touch): # *putting some sand when we do a left click*

**global** length,n\_points,last\_x,last\_y

**with** self.canvas:

**Color**(0.8,0.7,0)

            d=10.

            touch.ud['line'] = **Line**(points = (touch.x, touch.y), width = 10)

            last\_x = **int**(touch.x)

            last\_y = **int**(touch.y)

            n\_points = 0

            length = 0

            sand[**int**(touch.x),**int**(touch.y)] = 1

**def** **on\_touch\_move**(self, touch): # *putting some sand when we move the mouse while pressing left*

**global** length,n\_points,last\_x,last\_y

**if** touch.button**==**'left':

            touch.ud['line'].points += [touch.x, touch.y]

            x = **int**(touch.x)

            y = **int**(touch.y)

            length += **np.sqrt**(**max**((x - last\_x)\*\*2 + (y - last\_y)\*\*2, 2))

            n\_points += 1.

            density = n\_points/(length)

            touch.ud['line'].width = **int**(20\*density + 1)

            sand[**int**(touch.x) - 10 : **int**(touch.x) + 10, **int**(touch.y) - 10 : **int**(touch.y) + 10] = 1

            last\_x = x

            last\_y = y

# *API and switches interface (see kivy tutorials: https://kivy.org/docs/tutorials/pong.html)*

**class** CarApp(App):

**def** **build**(self): # *building the app*

        parent = **Game**()

**parent.serve\_car**()

**Clock.schedule\_interval**(parent.update, 1.0 / 60.0)

        self.painter = **MyPaintWidget**()

        clearbtn = **Button**(text='clear')

        savebtn = **Button**(text='save',pos=(parent.width,0))

        loadbtn = **Button**(text='load',pos=(2\*parent.width,0))

**clearbtn.bind**(on\_release=self.clear\_canvas)

**savebtn.bind**(on\_release=self.save)

**loadbtn.bind**(on\_release=self.load)

**parent.add\_widget**(self.painter)

**parent.add\_widget**(clearbtn)

**parent.add\_widget**(savebtn)

**parent.add\_widget**(loadbtn)

**return** parent

**def** **clear\_canvas**(self, obj): # *clear button*

**global** sand

        self**.painter.canvas.clear**()

        sand = **np.zeros**((longueur,largeur))

**def** **save**(self, obj): # *save button*

**print**("saving brain...")

**brain.save**()

**plt.plot**(scores)

**plt.show**()

**def** **load**(self, obj): # *load button*

**print**("loading last saved brain...")

**brain.load**()

# *Running the app*

**if** \_\_name\_\_ **==** '\_\_main\_\_':

**CarApp**().**run**()