Self Driving Car Using Deep Q-Learning

Achyuth Nandikotkur

December 10, 2021

Abstract

Reinforcement learning is a strategy through which agents, often computer-based, learn to find an optimal solution to a problem by imitating how humans/animals learn in the real world. Often in reinforcement learning, similar to the real world, the environment rewards or punishes the agent based on how conducive its actions in a particular state are for it to reach the goal. In this project, I applied my learnings from the ECE-559B course to build a self-driving car that learns to drive towards the current goal location on the map while maximizing the reward it obtains from the environment, using the Deep Q Learning algorithm.

1 Introduction

Reinforcement learning problems involve an agent learning to take actions at different states, such that the reward it obtains from the environment is maximized. An agent in a reinforcement problem can be any decision-making entity, put inside a Software or real-world environment. It is also important to mention that often an agent is equipped with sensors that give it awareness about its current situation, and this awareness forms the state of an agent. In this project, the agent is a car placed inside an environment built with a python framework called Kivy, a Cross-platform Python Framework for NUI Development. Self-driving cars today make use of supervised learning algorithms to train a machine learning model, such that a curve is fit through the training data points and consequently it can help in the prediction of actions that need to be taken on the real-world data. However the accuracy of the model is dependent on the volume and quality of the data used for training, and furthermore, the model cannot adapt if the environment changes. Hence I decided to use the reinforcement learning approach, where a car can learn about the environment by interacting with it and getting rewards based on the value of the actions that it takes. This very value of actions at different states can be learned by the car using the Q-learning algorithm. Furthermore, Q-learning would be a good starting point because in this case as the statistical description/model of system dynamics is not readily available, and it can only be learned by interacting with the environment. The Q-learning algorithm is based on Temporal difference learning, the simplest form of it is when an agent calculates the value of taking an action at a particular state using the following equation.

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[R(s_t, a_t) + \gamma max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)]$$

In the above equation, we are updating the value of taking an action at s by taking a temporal difference between rewards obtained after taking an action at a particular state and the known state action value so far. This difference is bound to become smaller as the agent learns. In simpler terms, I'd like to think of this as how surprised/frustrated the agent is with seeing a reward for an action taken at state s. However, Q-learning requires the states to be finite, and an exhaustive list of different transitions is required to determine optimal policy by iteration. In a situation, where the state space is very high, it becomes computationally expensive to calculate the value of every state and sometimes this task may be impossible. Instead, we can approximate the state-action values by training a neural network[1] with the transitions experienced by the agent and back-propagating the T.D loss.

2 Problem formulation

In this section, I will formally define the problem as an MDP (Markov decision process) and discuss individual components in detail.

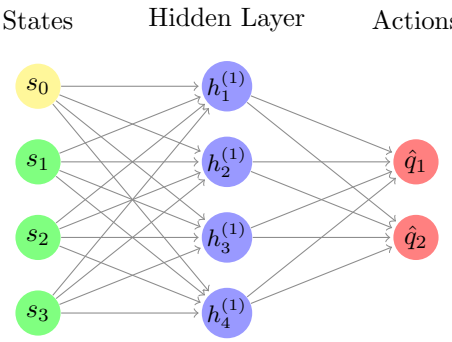


Figure 1: A simple neural network showing states as inputs, and action values as outputs.

$$back - propagated - loss = 1/2 * [R_t + \gamma max_a(Q(s_{t+1}, a)) - Q(s_t, a_t)]^2$$

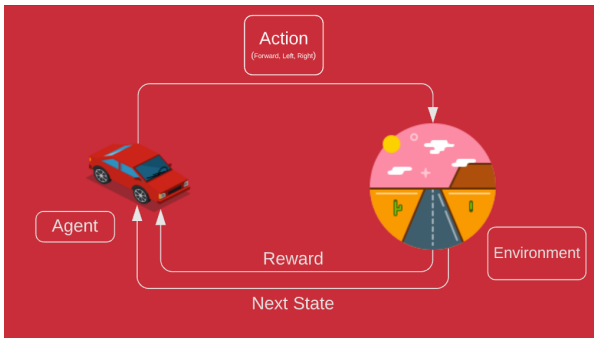


Figure 2: Agent-environment interaction cycle.

2.1 Environment

In this problem, the environment is a piece of land (made up of a track, 3 goal locations, some vegetation, and sand, etc.) that provides, information about itself to the agent along with the next states and rewards. To develop such an environment, an open-source Python UI development framework called Kivy is used as it made user interaction programming very easy.

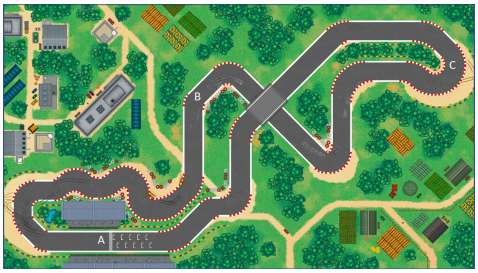


Figure 3: Environment containing a track, 3 goal locations, sand and some vegetation.

2.2 Agent

The agent is a car, equipped with an orientation and three sand density measurement sensors. The orientation sensor lets that car know of its orientation with respect to the goal location, and the three sand density measurement sensors let the car know of the density of the sand beneath the car on the left, middle and right sides of the car.

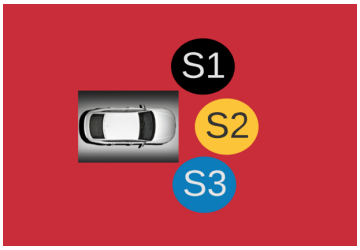


Figure 4: A car equipped with sand density measurement sensors on left, middle and right sides of it.

2.3 Objective

The car must try to reach the goal as fast as possible from any point on the map while maximizing the reward it obtains from the environment. There are three different goal locations in this problem (i.e A, B, C). As soon as the car reaches one goal (say A), the goal is updated to be at B, and as soon as the car reaches location B, the goal is updated to be at C and this process repeats as the task is continuous. For the car to reach a goal as fast as possible, it must be rewarded for moving towards the goal and penalized for moving away or going off-road. It is important to note that the states and rewards must be defined in such a way that they can contain enough information for the agent to make decisions that make it move towards the goal.

2.4 States

State is a vector that best describes the current situation of an agent in the environment with respect to the goal. Each state vector must contain information, with which the car can take actions that lead to the goal. In this problem, we want the car to move towards the current goal location as fast as possible, this can only happen if the car moves towards the goal while being on the track. Therefore the car is equipped with certain sensors which inform it of its orientation with respect to the goal, and whether if it is being on the top of the road. The values of these sensors form the state vector.

- Orientation sensor: The angle between the axis of the car and the center of the current goal location tells us if the car is moving towards or away from it, this is helpful in determining the action that must be taken to correct this.

Figure 5: Orientation w.r.t to the current goal location

- Sand density measurement sensors: These sensors are placed on the left, center and right sides of the car which help in detecting the density of the sand underneath each sensor. For example, If any sand is detected on the left side of the car, the car can turn rightwards to avoid it and thus stay on the road. Each sensor measures the density of sand particles beneath itself. This can be computed by considering a square of 20 by 20 cells around each sensor. The environment makes available this information to the sensor in the form of a matrix whose elements are either 1 or 0 depending on the presence of a sand particle.

$$Density = nnz(X)/numel(X); SensorValueRange = 0to1(stepsof0.0025)$$

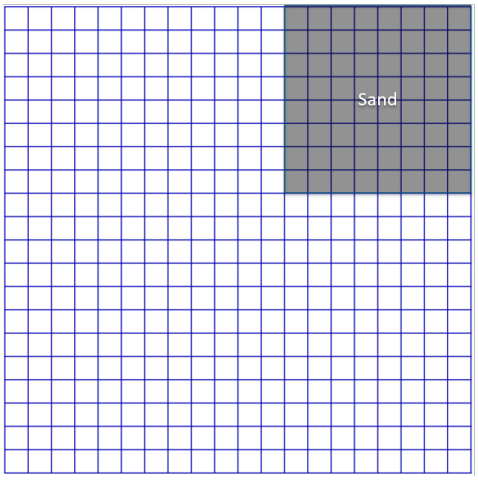


Figure 6: Sample sand density sensor measurement

$$nnz = numberofnonzeros; X = 20x20matrix;$$

Since there are about 400 cells, the density can vary from 0 to 1 in steps of 0.0025 (1/400). Hence there can be a total of 400 states for each sensor.

2.5 Total State Space

It is important to estimate the total state space as it helps us to pick the most appropriate learning algorithm to solve this problem. After discretizing the orientation sensor to have a resolution of 0.1^0, there can be a total of 3600 different states because of it. Furthermore, sand density measurement sensors have a value range between 0 to 1 with a resolution of 0.0025, hence, they contribute a total of 400 * 400 * 400 states to the total state space. Therefore the total state space is estimated to be around $3600 * 400 * 400 * 400$.

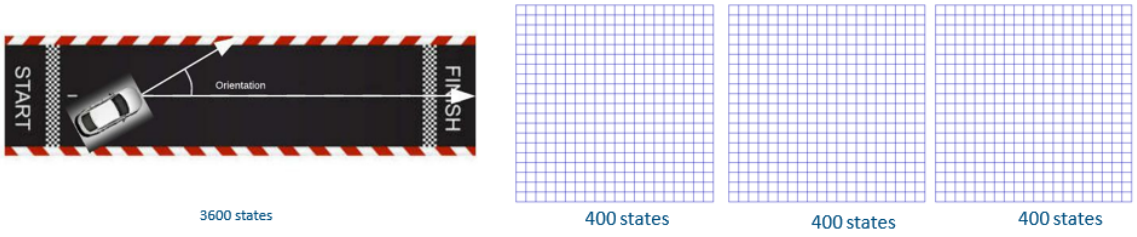


Figure 7: Estimation of total state space.

$$TotalStateSpace = 3600 * 400 * 4000 * 400 \approx 2^{32}$$

2.6 Actions

The actions must be defined in such a way that they allow the agent to interact with the environment, obtain rewards and move towards the goal. Hence, the actions that the car can take are listed below.

- Move forward.
- Turn left.

- Turn right.

2.7 Rewards

The rewards must be defined intelligently so that the agent can infer from its interaction with the environment as to what's a valuable action in a particular state and what isn't. In this problem, the rewards are deterministic and are defined as follows.

- +0.5 for if the agent moves close to the goal.
- -1 if the car moves away from the goal.
- -5 if the car drives into the sand.
- -1 for every step the cars takes while on top of the road.
- -6 if the car hits the boundary of the map.

2.8 Transition Model

Moving from one state to another is called a Transition. In this problem, all transitions are deterministic and the task itself is continuous. If the car is not at the edge of the map, each action will lead to a valid transition into the next state. If the car is at the edge of the map, actions that cause the car to go off-board, will result in no transition and, the car will remain in the same state. Each step is expected to take less than 100 milliseconds. The time step includes the time taken for the neural network to predict an action + time taken for the environment to render the transition.

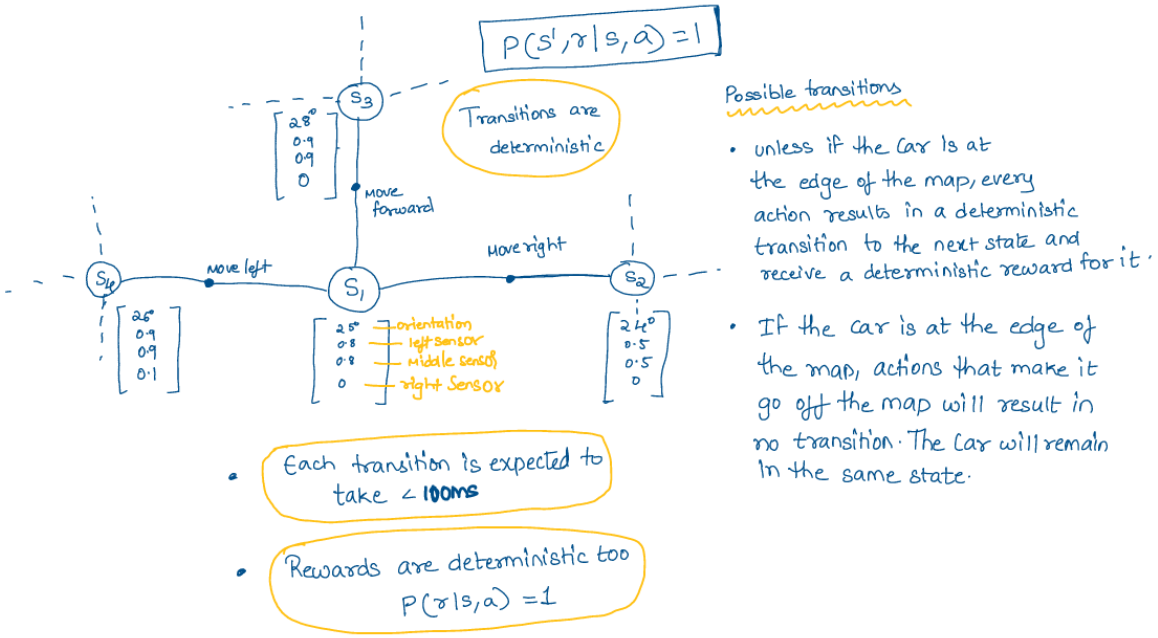
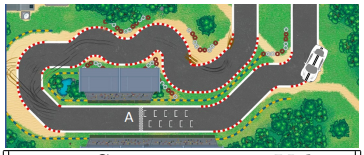


Figure 8: Transition model showing example transitions

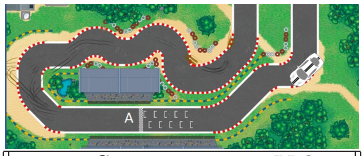
2.9 Sample Transition

Here is a sample transition where the car takes an action A (output by the neural network) at state S and gets a reward R for it. I will discuss more about how the neural network is used to predict the actions in the latter part of this report.



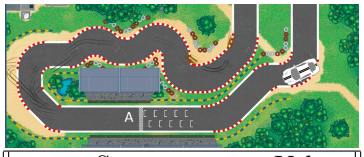
Sensor	Value
Orientation Sensor	5.31^0
SDS-Left	1
SDS-Middle	0.5
SDS-Right	0
Reward	-5

Figure 9: Sample state-1.
Output of neural-network = 'Turn right'



Sensor	Value
Orientation Sensor	2.31^0
SDS-Left	0.8
SDS-Middle	0.2
SDS-Right	0
Reward	-5

Figure 10: Sample state-2.
Output of neural-network = 'Turn right'



Sensor	Value
Orientation Sensor	1.31^0
SDS-Left	0.3
SDS-Middle	0
SDS-Right	0

Figure 11: Sample state-3 (Reward = 0.2 + 1)

3 Solution Method

Q-learning is a model-free reinforcement learning technique to learn the value of an action in a particular state. It does not require a model of the environment, and it can handle problems with stochastic transitions and rewards without requiring adaptations. However, in vanilla Q-learning, all possible states must be explored to calculate the action values of all states and thus converge at an optimal policy. When the state space is very large, it becomes extremely expensive, both in terms of time and space complexity to arrive at an optimal solution. In this problem, the number of states is estimated to be around 2^{32}, hence we need a better mechanism that helps in approximating action values at certain states, this is where Deep Q Learning helps. Under this method, we will be able to train a neural network with transitions seen so far so that it can predict the action values of states that were never visited with a reasonable degree of accuracy. Such an approximation helps reduce the computational load and reach a near-optimal solution relatively faster.

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[R(s_t, a_t) + \gamma max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)]$$

In the above equation, the neural network approximates the value of $Q(s_t, a_t)$ and $Q(s_{t+1}, a')$ terms, and the agent obtains the value of $R(s_t, a_t)$ term by interacting with the environment. α is the learning rate, which is generally fixed or equated to $1/N$ where N is the number of times a particular action A is picked in a state S. γ is called the discount factor, which helps in allowing the agent to take actions based on a long term reward, and it also helps in keeping the q-values to remain finite, as this is a continuous task. It is also important to note that, when in a certain state S if the agent always picks an action that has the highest q-value calculated so far, the agent will never be able to explore, and thus never update the q-values of certain states. This will result in the agent getting stuck at local optimal solutions. Some degree of exploration is extremely important for the agent to explore all the states, and thus constantly be on the lookout for better solutions.

3.1 Exploration vs Exploitation

When in a state S, the neural network outputs action values for different actions that the agent can take at that state. When in such a situation, the agent has two options, either take an action that has the maximum value or take an action randomly. When taking an action that has the maximum value, it is important to understand that the other actions may be having a lesser value due to the fact that they were never taken and consequently the following equation for those actions was never updated, thus resulting in a lower value.

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[R(s_t, a_t) + \gamma max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)]$$

Therefore, when in a state S picking an action that has the highest value, could result in the agent sticking to a sub-optimal solution. Similar to human life, exploration is important, to find efficient solutions for the problems of humanity. Exploration will result in the agent taking different actions in certain states than the best action known so far, this may lead to getting a bigger reward than seen before, or a poorer reward, which will, in turn, result in action values of different actions at this particular state being updated thus the learning happens. The more the agent knows about the environment, the better the actions it can take. On the other hand, always exploring could never result in converging to an optimal solution, as at any state S the agent prefers to take exploratory actions instead of the known best action. Hence, there needs to be a trade-off between exploration and exploitation. In this problem, I aim to create exploration by picking action from the neural network using the softmax function shown below.

$$a_t = Softmax_a(Q(s_t, a))$$

In the Softmax method we're going to give each action a probability proportional to its Q-value, such that the higher the Q-value, the higher the probability. This creates, exactly, a distribution of the performable actions. Then finally, the action performed will be selected as a random draw from that distribution. Let me explain with an example.Let us say that, at a specific time t and state S, the neural network predicts the following Q-values:

Action	Value
Move forward	54
Turn left	31
Turn right	5

The way we can create the distribution of probabilities we need is by dividing each Q-value by the sum of the three Q-values, which results each time in the probability of a particular action. Let's perform those sums:

$$Probability\ of\ Moving\ Forward = \frac{54}{54 + 31 + 5} = 60\%$$

$$Probability\ of\ Turning\ Left = \frac{31}{54 + 31 + 5} = 34.4\%$$

$$Probability\ of\ Turning\ Right = \frac{5}{54 + 31 + 5} = 5.5\%$$

the probabilities sum to 1 and they are proportional to the Q-values. That gives us a distribution of the actions. To perform an action, the Softmax method takes a random draw from this distribution, such that:

- The action of Moving Forward has a 60% chance of being selected.
- The action of Turning Left has a 34.4% chance of being selected.
- The action of Turning Right has a 5.5% chance of being selected.

It is important to note, that every action has a chance of being selected unlike in the greedy approach, where we always select the action having the highest q-value.

3.2 Deep Q Learning

This is a technique where Q-learning is combined with an Artificial Neural Network. Inputs are encoded vectors, each one defining a state of the environment. These inputs go into an ANN, and the output contains the predicted Q-values for each action. More precisely, if there are n possible actions the AI could take, the output of the artificial neural network is a 1D vector comprised of n elements, each one corresponding to the Q-values of each action that could be performed in the current state. Then, the action performed is chosen via the Softmax method described in detail above.

Hence, in each state S:

- The prediction is the Q-value , where a_t is performed by the Softmax method.
- The target is $R(s_t, a_t) + \gamma\, max_{a'}(Q(s_{t+1}, a'))$
- The loss error between the prediction and the target is the square of the temporal difference

$$back\ propagated\ loss = 1/2 * [R_t + \gamma max_a(Q(s_{t+1}, a)) - Q(s_t, a_t)]^2$$

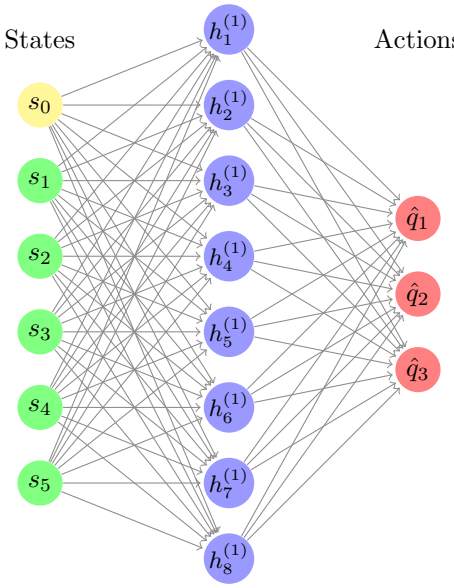
  This loss error is back-propagated into the neural network, and the weights are updated according to how much they contributed to the error, through any optimization techniques such as stochastic gradient descent or mini-batch optimization.

Since we now know how the neural network works, let's build one. Firstly, we must determine the number of hidden layers. The Universal Approximation Theorem states that any function could be approximated using one input layer, one hidden layer, and one output layer. However, we must also now decide the number of nodes in the hidden layer. Deciding the number of neurons in the hidden layers is a very important part of deciding the overall neural network architecture. Though these layers do not directly interact with the external environment, they have a tremendous influence on the final output. Both the number of hidden layers and the number of neurons in each of these hidden layers must be carefully considered. Using too few neurons in the hidden layers will result in something called under-fitting. Under-fitting occurs when there are too few neurons in the hidden layers to adequately detect the signals in a complicated data set. Using too many neurons in the hidden layers can result in several problems. First, too many neurons in the hidden layers may result in over-fitting. Overfitting occurs when the neural network has so much information processing capacity that the limited amount of information contained in the training set is not enough to train all of the neurons in the hidden layers. A second problem can occur even when the training data is sufficient. An inordinately large number of neurons in the hidden layers can increase the time it takes to train the network. The amount of training time can increase to the point that it is impossible to adequately train the neural network. Obviously, some compromise must be reached between too many and too few neurons in the hidden layers. There are many rule-of-thumb methods for determining the correct number of neurons to use in the hidden layers[6], such as the following:

- The number of hidden neurons should be between the size of the input layer and the size of the output layer.

- The number of hidden neurons should be 2/3 the size of the input layer, plus the size of the output layer.
- The number of hidden neurons should be less than twice the size of the input layer.

In the light of above information, I decided to use 8 neurons in the hidden layer.



$$s0 = orientation$$

$$s1 = sand\ density\ sensor\ 1$$

$$s2 = sand\ density\ sensor\ 2$$

$$s3 = sand\ density\ sensor\ 3$$

$$s4 = sand\ density\ sensor\ 4$$

$$q1 = Move\ forward$$

$$q2 = Move\ left$$

$$q3 = Move\ right$$

Figure 12: Neural network showing states as inputs, and action values as outputs.

3.3 Experience replay

Whenever an agent takes an action $a1$ at state $s1$ and moves into a next state $s2$ and gets a reward $r1$ for it, a single transition $s1, a1, r1, s2$ is said to be complete. If the neural network is trained with such sequential transitions of experiences, it could become biased to features of predominantly correlated sequential states of the environment. For example, if the track is straight, the network needs to learn to output the 'move forward' action continuously for many time steps, making it more sensitive to straight roads than corners. With the help of experience replay (shown in Figure 13), we can train the network in batches of random experiences from this store instead of using sequential samples, thus dealing with correlation effectively. Instead of uniformly sampling from a replay memory, we can even replay important transitions more frequently[5] and therefore learn more efficiently. This could be improved if, instead of only considering the last transition each time, we considered the last m transitions, where m is a large number. This set of the last m transitions is what is called the experience replay memory, or simply memory. From this memory, we sample some random transitions into small batches. Then we train the neural network with these batches to then update the weights through

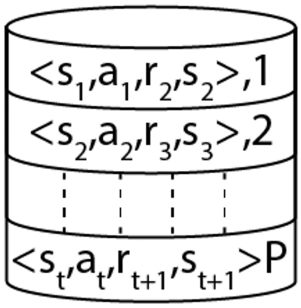


Figure 13: Replay buffer containing different transitions.

mini-batch gradient descent.

4 Implementation

In this section we will discuss in step by step, the method followed to translate the design defined in the earlier sections into a working project.

4.1 Building the environment

The whole environment for this project is built using Kivy, an open-source Python framework, used for the development of games. In kivy, *.kv* file is used to define all the entities that are used in the environment. The icon for the car and its sensors, and the image of the map containing the track, can simply be declared in the .kv file so that they can be imported later for further processing.

4.1.1 Building the agent

The agent and its sensors are defined as shown below. An icon for the agent is picked from a sample image found on the internet. The sensors however are bounded inside a rectangle whose width and height are 10 pixels each and the shape for the sensor is defined as an ellipse.

```
car.kv
#:kivy 1.0.9
# ref: https://kivy.org/docs/tutorials/pong.html

<Car>:
    size: 20, 10
    origin: 10, 5
    canvas:
        PushMatrix
        Rotate:
            angle: self.angle
            origin: self.center
        Rectangle:
            pos: self.pos
            size: self.size
            source: "./images/Mycar.jpg"
        PopMatrix
```

```
<Ball1>:
    size: 10,10
    canvas:
        Color:
            rgba: 0.48,0.97,0.74,0.3
        Ellipse:
            pos: self.pos
            size: self.size
<Ball2>:
    size: 10,10
    canvas:
        Color:
            rgba: 0.48,0.97,0.74,0.3
        Ellipse:
            pos: self.pos
            size: self.size

<Ball3>:
    size: 10,10
    canvas:
        Color:
            rgba: 0.48,0.97,0.74,0.3
        Ellipse:
            pos: self.pos
            size: self.size
```

4.1.2 Building the map

The base image for creating the map of the environment is picked from an existing PC game called 'Race Arcade'[7]. The whole environment along with the base image of the map and previously declared entities is defined in the *car.kv* file as shown below

```
    <Game>:
    car: game_car
    ball1: game_ball1
    ball2: game_ball2
    ball3: game_ball3
    canvas:
        Rectangle:
            pos: self.pos
            size: 1203, 678
            source: "./images/Map.png"

    Car:
        id: game_car
        center: self.parent.center
    Ball1:
        id: game_ball1
        center: self.parent.center
    Ball2:
        id: game_ball2
        center: self.parent.center
    Ball3:
        id: game_ball3
        center: self.parent.center
```

4.1.3 Initializing the map

To proceed further, a pure black and white version of the base image is necessary. The black part on the map represents the track and the white part represents the sand. I was able to create such a version by following the instructions mentioned in Chapter-10 of the book titled 'AI Crash Course'[8] and save the resultant image as $Mymask2.png$. This image can be divided by 255 to convert it into $1s$ and $0s$ which we will be later used by the agent to know whether if it's on the track or on the sand. Coming to the location of the goal, in this problem, I placed three different finish lines at three different locations (A, B, C) on the map, so that the entire part of the track is covered by the agent. Otherwise, the agent may sometimes try to go over the sand to reach the finish line.

```
# Initializing the map
def init():
    # an array to store the location of the sand particles on the map
    global sand

    # x-coordinate of the goal
    global goal_x

    # y-coordinate of the goal
    global goal_y

    # initializing the array with zeros (width * height of the base image)
    sand = np.zeros((self.width,self.height))

    # Mymask2 is the black and white conversion of the image declared in car.kv file.
    img = PILImage.open("./images/Mymask2.png").convert('L')

    # Image is converted into 0s and 1s where 0s represent the sand
    # and 1s represent the track.
    sand = np.asarray(img)/255

    # location of goal - A on the map
    goal_x = 252
    goal_y = 75

    # The variable swap is used to swap
    # the location of the goal with a new one
    # as the agent reaches it.
    global swap
    swap = 0
```

4.1.4 Creating the Car Class

The car class is created with the following member elements that represent the agent's current situation.

- angle: The angle between the x-axis of the map and the axis of the car
- rotation: The last rotation made by the car (we will see later that, taking action will result in a rotation)
- position = (self.car.x, self.car.y): The position of the car (self.car.x is the x-coordinate of the car, self.car.y is the y-coordinate of the car)
- velocity = (velocity_x, velocity_y): The velocity vector of the car
- sensor1 = (sensor1_x, sensor1_y): The position of the sand density measurement sensor located on the left side of the car.

- sensor2 = (sensor2_x, sensor2_y): The position of the sand density measurement sensor located in the middle center of the car.
- sensor3 = (sensor3_x, sensor3_y): The position of the sand density measurement sensor located on the right side of the car.
- signal1: The signal received by the sensor on the left side.
- signal2: The signal received by the middle sensor.
- signal3: The signal received by the sensor on the right side..]

```
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.position = Vector(*self.velocity) + self.position
    self.rotation = rotation
    self.angle = self.angle + self.rotation

    # Left sensor's position initialized.
    self.sensor1 = Vector(30, 0).rotate(self.angle) + self.position

    # Middle sensor's position initialized.
    self.sensor2 = Vector(30, 0).rotate((self.angle+30)%360) + self.position

    # Right sensor's position initialized.
    self.sensor3 = Vector(30, 0).rotate((self.angle-30)%360) + self.position

    # getting the signal received by sensor 1 (density of sand around sensor 1)
    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 2 (density of sand around sensor 2)
    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 3 (density of sand around sensor 3)
    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 the sensor reaches the boundaries of the map, make a note of this scenario
# by setting all corresponding signal values to be 10
if self.sensor1_x>self.width-10 or self.sensor1_x<10 or
self.sensor1_y>self.height-10 or self.sensor1_y<10:
    self.signal1 = 10.
if self.sensor2_x>self.width-10 or self.sensor2_x<10
or self.sensor2_y>self.height-10 or self.sensor2_y<10:
    self.signal2 = 10.
if self.sensor3_x>self.width-10 or self.sensor3_x<10 or
self.sensor3_y>self.height-10 or self.sensor3_y<10:
    self.signal3 = 10.
```

A few things to note:

- Since the sand arrays only contain 1s and 0s, we can calculate the density of the sand underneath the car by simply summing the cells of the sand array in this 20 by 20 square and dividing by 400.

```
self.car.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.
```

- The $move(self, rotation)$ function will move the car one step. This will be called in equal intervals to animate the transition. The new position is calculated by creating a 2D vector from current velocity and adding current position to it. I was able to create this function after referring an example provide in the Kivy documentation[9].

```
# The move function will be called after the neural network updates itself
with the states and outputs an action. The neural network is set to update
once every 60th of a second.

# Neural network update function called 60 times a second
lock.schedule_interval(parent.nnupdate, 1.0/60.0)
```

- Therefore, as mentioned in the MDP, each time step takes a total time of (100 milliseconds + time for the neural network to output an action).
- If the car reaches the boundary of the map, the sand density measurement sensors are initialized with a value of 10 just so that the neural network can learn to distinguish between states when the agent is on the sand and at the boundary of the map.

4.1.5 Creating the Game Class

This class is responsible for extracting the last state vector from the car class, calling the $brain.update(state, reward)$ in *ai.py* with the extracted state vector, and the rewards it received when in that state, and consuming the action output by that function. Furthermore, it is also responsible for executing the current action and assigning the rewards to the agent based on the rules mentioned above in the rewards section. Internally, the $brain.update(state, reward)$ calls the neural network with the current state, uses the reward passed into it, to backpropagate the TD error, and passes the action output by the neural network into a softmax function, and returns the selected action. The *Game* class is also responsible for updating the goal to a new location if the agent reached the earlier one.

```
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
    global swap


    longueur = self.width
    largeur = self.height
    if first_update:
        init()

    xx = goal_x - self.car.x
    yy = goal_y - self.car.y

    # Calculating the orientation of the car.
    orientation = Vector(*self.car.velocity).angle((xx,yy))/180.

    # Preparing a state vector from the orientation calculated,
    the values of all the 3 sand density sensors.
    last_signal = [self.car.signal1, self.car.signal2,
    self.car.signal3, orientation, -orientation]

    # Here the brain.update function is called with the last state
    and the reward received in it.
    # brain.update function will be explained in more details in
    the upcoming sections below
    action = brain.update(last_reward, last_signal)

    # brain.score() function holds the running average of the score
    # received in each step.
    scores.append(brain.score())

    # action to rotation is an array which simply converts
    # the actions 0(move forward), 1(move left), 2(move right)
    # to 0, 5 and -5 degrees respectively
    action2rotation = [0,5,-5]
    rotation = action2rotation[action]

    # Here the car class's move function is called with the rotation
    self.car.move(rotation)

    # Here we calculate the distance between the car's current location and the current goal.
```

```
distance = np.sqrt((self.car.x - goal_x)**2 + (self.car.y - goal_y)**2)

# If the car is inside the sand, assign it a reward of -5
if sand[int(self.car.x),int(self.car.y)] > 0:
    self.car.velocity = Vector(0.5, 0).rotate(self.car.angle)
    print(1, goal_x, goal_y, distance, int(self.car.x),int(self.car.y),
    im.read_pixel(int(self.car.x),int(self.car.y)))

    last_reward = -5
else:
    self.car.velocity = Vector(2, 0).rotate(self.car.angle)
    print(0, goal_x, goal_y, distance, int(self.car.x),int(self.car.y),
    im.read_pixel(int(self.car.x),int(self.car.y)))
    # if the car is moving towards the goal, assign it
    # a positive reward.
    if distance < last_distance:
        last_reward = 0.1
    else:
        # if the car is moving away, assign
        # it a negative reward of -0.2
        last_reward = last_reward +(-0.2)

# Here we continuously update the location
# of the goal, as soon as each goal is
# reached by the car.
if distance < 25:
    if swap == 1:
        goal_x = 252
        goal_y = 75
        swap = 2
    elif swap == 2:
        goal_x = 495
        goal_y = 443
        swap = 3
    else:
        goal_x = 1148
        goal_y = 523
        swap = 1
last_distance = distance
```

4.1.6 Creating the Deep Q Network Class

The Dqn is responsible for implementing the Deep Q Learning part of the problem. It internally uses the neural network and replay memory classes which will be created in the subsequent sections. $update(reward, new\_signal)$ method of this class is called from the Game class with state vector, and associated reward at time step t as parameters. The state vector along with the reward will be stored as a transition in the replay buffer, and when the number of transitions stored in the replay buffer accumulates to a value beyond a certain set threshold, the q-values of all involved state-action pairs are updated in a batch, where the q-values themselves at each state are approximated with the help of the neural network as explained in the earlier sections above. For building the neural network, I used PyTorch, which is an open-source machine learning library based on the Torch library, used for applications such as computer vision and natural language processing, primarily developed by Facebook's AI Research lab. In the above description, I also want to clarify that the *update* method in the Game class will in turn be called by the Kivy framework once every 60^{th} of a second.

It is important to see how the error is back propagated through the network and the loss function is minimized using the Adam optimizer, to consequently update the weights of different leafs appro-

priately.

$$back - propagated - loss = 1/2 * [R_t + \gamma max_a(Q(s_{t+1}, a)) - Q(s_t, a_t)]^2$$

```
def learn(self, batch_state, batch_next_state, batch_reward, batch_action):
  # Use neural network to predict Q(s_t, a) in the above equation.
  outputs = self.model(batch_state).gather(1, batch_action.unsqueeze(1)).squeeze(1)

  # Use neural network to predict Q(s_{t+1}, a) and take max
  next_outputs = self.model(batch_next_state).detach().max(1)[0]

  # back-propagated-loss
  target = self.gamma*next_outputs + batch_reward

  # Here we create a simple loss function representing the equation above.
  td_loss = F.smooth_l1_loss(outputs, target)

  # back propagate this loss, such that the networks
  # weights are updated
  self.optimizer.zero_grad()
  td_loss.backward(retain_graph = True)
  self.optimizer.step()
```

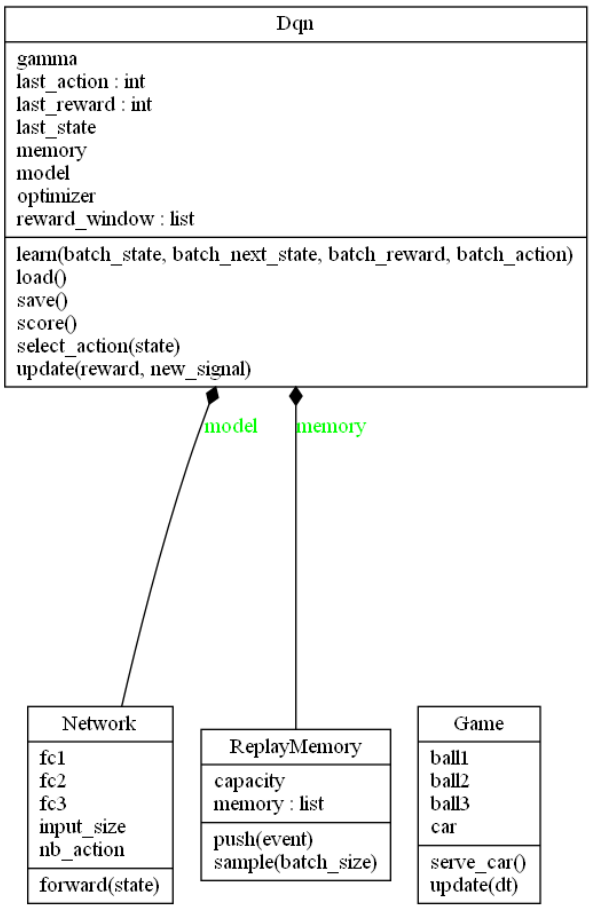


Figure 14: UML Diagram of *ai.py* file containing Dqn, Network and ReplayMemory classes

```
class Dqn():

def __init__(self, input_size, nb_action, gamma):
    # Setting gamma to 0.8
    self.gamma = 0.8
    self.reward_window = []

    # The model of dqn is initialized as an of the Network class
    # The network class is simply a wrapper over PyTorch neural network
    # containing a network created for this problem.
    self.model = Network(input_size, nb_action)

    # Replay memory is initialized to $10^5$
    self.memory = ReplayMemory(100000)

    # Adam optimizer is used instead of stochastic gradient descent
    self.optimizer = optim.Adam(self.model.parameters(), lr = 0.001)
    self.last_state = torch.Tensor(input_size).unsqueeze(0)
    self.last_action = 0
    self.last_reward = 0

def select_action(self, state):
    # Probabilities of choosing an action is calculated from the
    # neural network output as described in the sections above.
    probs = F.softmax(self.model(Variable(state, volatile = True))*1000) # T=100

    # An action is selected based on the probability distribution
    # of actions determined above.
    action = probs.multinomial(num_samples=1)
    return action.data[0,0]

def learn(self, batch_state, batch_next_state, batch_reward, batch_action):
    outputs = self.model(batch_state).gather(1, batch_action.unsqueeze(1)).squeeze(1)
    next_outputs = self.model(batch_next_state).detach().max(1)[0]
    target = self.gamma*next_outputs + batch_reward
    td_loss = F.smooth_l1_loss(outputs, target)
    self.optimizer.zero_grad()
    td_loss.backward(retain_graph = True)
    self.optimizer.step()

def update(self, reward, new_signal):
    new_state = torch.Tensor(new_signal).float().unsqueeze(0)
    self.memory.push((self.last_state, new_state, torch.LongTensor([int(self.last_action)]), torc
    action = self.select_action(new_state)

    # While the replay buffer contains more than 100 transitions
    # We will pick a hundred samples randomly, and use them to train
    # the network as well as calculate the q-values
    # of involved state-action pairs.
    if len(self.memory.memory) > 100:
        batch_state, batch_next_state, batch_action, batch_reward = self.memory.sample(100)
        self.learn(batch_state, batch_next_state, batch_reward, batch_action)
    self.last_action = action
    self.last_state = new_state
    self.last_reward = reward
    self.reward_window.append(reward)
```

```
        if len(self.reward_window) > 2000:
            del self.reward_window[0]
        return action

    def score(self):
        return sum(self.reward_window)/(len(self.reward_window)+1.)

    # A method to store the network values calculated so far, so that next
    # time when we run the script, the stored network values can be used
    # by the agent.
    def save(self):
        torch.save({'state_dict': self.model.state_dict(),
                    'optimizer' : self.optimizer.state_dict(),
                   }, 'last_brain.pth')

    def load(self):
        if os.path.isfile('last_brain.pth'):
            print("=> loading checkpoint... ")
            checkpoint = torch.load('last_brain.pth')
            self.model.load_state_dict(checkpoint['state_dict'])
            self.optimizer.load_state_dict(checkpoint['optimizer'])
            print("done !")
        else:
            print("no checkpoint found...")
```

4.1.7 Creating a Neural Network using PyTorch

This class is a simple wrapper over the PyTorch framework containing a network whose inputs are the variables of the state vector defined for this problem, whose hidden layer is containing a set of 8 nodes as designed in the earlier sections, and whose outputs are the q-values of actions described for this problem. It is important to note that an instance of this class is used to initialize the model variable in the Dqn class.

```
    class Network(nn.Module):

    def __init__(self, input_size, nb_action):
        super(Network, self).__init__()
        self.input_size = input_size
        self.nb_action = nb_action
        self.fc1 = nn.Linear(input_size, 30)
        self.fc2 = nn.Linear(30, 30)
        self.fc3 = nn.Linear(30, nb_action)

    def forward(self, state):
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        q_values = self.fc3(x)
        return q_values
```

4.1.8 Creating ReplayMemory Class

As mentioned in the sections above, replay memory is necessary to prevent the network from being biased towards the predominantly correlated sequential states of the environment. For this purpose, I have created a class that can be instantiated to store and sample transitions.

```
class ReplayMemory(object):

def __init__(self, capacity):
```

```
    self.capacity = capacity
    self.memory = []

def push(self, event):
    self.memory.append(event)
    if len(self.memory) > self.capacity:
        del self.memory[0]

def sample(self, batch_size):
    samples = zip(*random.sample(self.memory, batch_size))
    return map(lambda x: Variable(torch.cat(x, 0)), samples)
```

4.2 Results

After taking about 40000 steps, the agent learned to stay on the top of the track, avoid moving on the sand, and move towards the goal. At this point, the agent on average was receiving a reward of -0.47 from the environment.

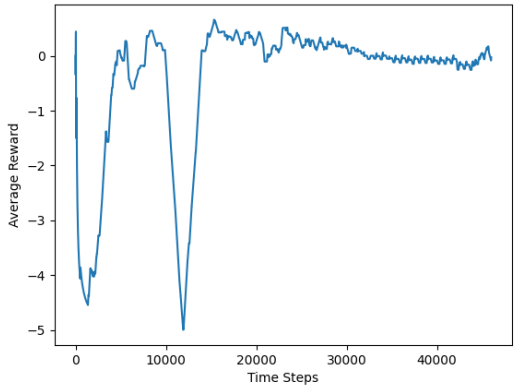


Figure 15: A plot showing the average reward vs time steps

4.2.1 Video

A video showing the working prototype of this implementation is posted here.

4.3 Conclusion and remarks

In overall, learning and building individual components of this project has made it an exhilarating journey for me. From not knowing how to implement a neural network to building one for this problem, empowered me in several ways. So much so that, after reading a few research papers on reinforcement learning, I was able to come up with new ideas in the area of experienced replay. In this regard, I was able to write an email to one of the leading researchers in RL, David Silver expressing my idea and even got a reply from him, where he mentioned that my idea was pragmatic and I should work on it. All this wouldn't have been possible without the lectures, assignments, listening to others' presentations, and most importantly working on this project. Coming to the area of improvement, I have noticed that my solution sometimes gets stuck in a local optimum, this I believe could be due to the combination of my choice for states and rewards. I tried fixing this by playing with new state variables such as the location of the goal, distance from the goal, etc, and I even tweaked the rewards a bit to penalize the agent more if it ever goes off the road and away from the goal. This improved the situation a bit, however, a more robust solution would be to use the pixel information from the image directly to prepare a state vector and train a convolutional neural network with it, this could further enhance the performance and lead to faster learning. However, for this approach, a GPU with

good memory is needed. I can also think of using prioritized experience replay[5] instead of sampling transitions randomly from the buffer to train the neural network in batches.

References

[1] Sutton, R.S. Barto, A.G., 2018. Reinforcement learning: An introduction, MIT press.

[2] Lecture slides by H.-C. Yang, University of Victoria.

[3] Lecture notes H.-C. Yang, University of Victoria.

[4] An Introduction to Reinforcement Learning with David Silver. link

[5] Schaul, Tom, Quan, John, Antonoglou, Ioannis, and Silver, David. Prioritized experience replay. arXiv preprint arXiv:1511.05952, 2015.

[6] Introduction to Neural Networks for Java (second edition) by Jeff Heaton.

[7] Race Arcade, a game by Iceflake Studios.

[8] AI Crash Course by Hadelin de Ponteves

[9] Pong Game, a tutorial presented in Kivy Documentation.