"Modelled Version of Self driving Car"

Minor Project Report

(IT 457)

Mentor Details:-

Dr. Sartaj Singh Sodhi

Associate Professor

USICT, GGSIPU

Submitted By:-

Esshaan Mahajan

01916403219

B.Tech CSE (7thSem)

Index

Certificate

This is to certify that project report entitled Modelled Version of Self driving Car submitted by Mr. Esshaan Mahajan of B. Tech (CSE, 7th Semester) of USICT, GGSIPU in partial fulfilment for the award of the degree of B. Tech in Computer Science and Engineering is a bona fide record of project work carried out by him under my supervision. The contents of this report, in full or in parts, have not been submitted to any other Institution or University for the award of any degree or diploma.

________________.

Mentor's Signature

Mentor's Name: Dr. Sartaj Singh Sodhi

Declaration

I declare that this project report titled 'Modelled Version of Self Driving Car' submitted in partial fulfillment of the degree of B. Tech in (Computer Science and Engineering) is a record of original work carried out by me under the supervision of Dr. Sartaj Singh Sodhi and has not formed the basis for the award of any other degree or diploma, in this orany other Institution or University. In keeping with the ethical practice in reporting scientific information, due acknowledgements have been made wherever the findings of others have been cited.

Esshaan Mahajan

01916403219

B. Tech CSE

Acknowledgement

I would like to express my special thanks of gratitude to my supervisor "Dr. Sartaj Singh Sodhi" for his able guidance and support in completing my project. Your useful advice and suggestions were really helpful to me during the project's completion. In this aspect, I am eternally grateful to you.

Introduction

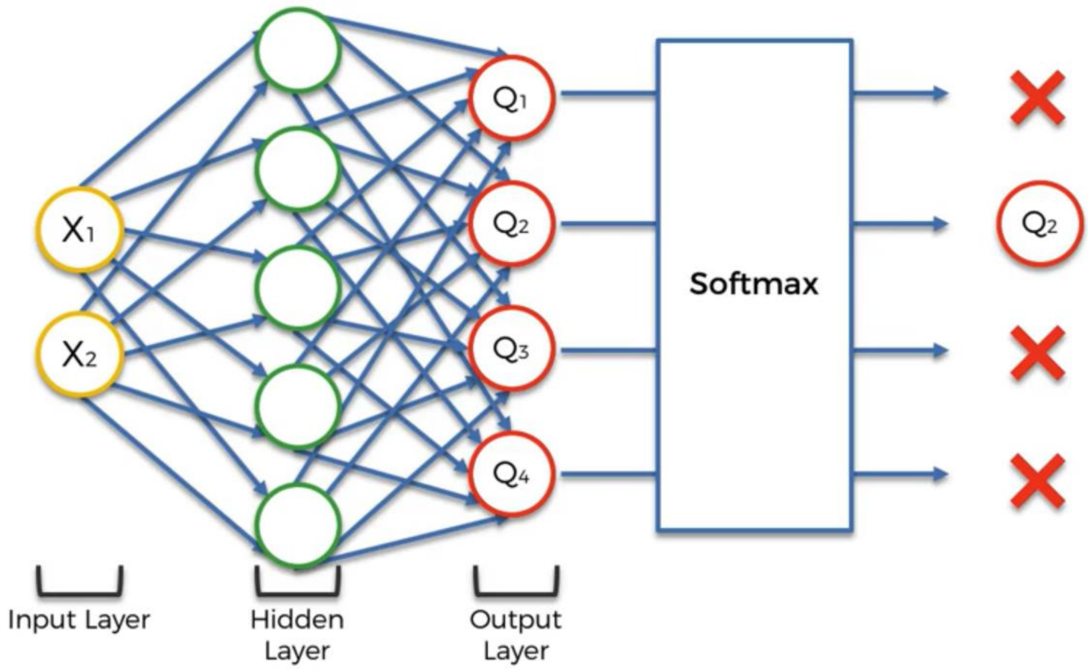
A self-driving car in itself provides a revolutionary solution to the economic and ecological problems related to transportation and traffic. It can prevent car crashes, enhance traffic efficiency and promote environment friendly. Furthermore, a modelled version of a self-driving car can be a very useful tool in creating a suitable environment for training a system which is capable of learning from its surroundings. This way a system can learn about complex situations that may arise on real-life roads without actually causing nuisance on roads.

Hence, in this project, the goal would be to create a modelled version of a self-driving car which would learn itself to drive from one point to another without any prior instructions to operate. It will learn from its surroundings and follow a road while dodging obstacles. The motive of the car would be to reach the goal destination from initial point of start. The car would explore the environment and use this knowledge to learn the appropriate path.

Methodology Adopted

Overall Approach:

The learning process will be fulfilled using Deep Q-learning. Deep Q-Learning is the result of combining Q-Learning with an Artificial Neural Network. The states of the environment will be encoded by a vector which will be passed as input into the Neural Network. Then the Neural Network will try to predict which action should be played, by returning as outputs a Q-value for each of the possible actions. Eventually, the best action to play will be chosen by either taking the one that has the highest Q-value, or by overlaying a Softmax function.



Tools and Technology Used:

- Python3.9
- Numpy
- Random
- PyTorch
- Matplotlib
- Kivy

Procedure

Initialization:

For all couples of actions a and states s, the Q-values are initialized to 1:

$$\forall a \in A, s \in S, Q_0(a, s) = 1$$

The Experience Replay is initialized to an empty list M.

We start in the initial state s_0. We play a random action and we reach the first state s_1.

1. We play the action a_t, where a_t is a random draw from the W_s distribution:

$$a_t \sim W_{s_t}(.) = \frac{\exp(Q(s_t,.))^\tau}{\sum_{a'} \exp(Q(s_t,a'))^\tau}, \text{ with } \tau \geq 0$$

2. We get the reward $r_t = R(a_t, s_t)$
3. We get into the next state s_{t+1}, where s_{t+1} is a random draw from the $T(a_t, s_t, .)$ distribution:

$$s_{t+1} \sim T(a_t, s_t, .)$$

4. We append the transition (s_t, a_t, r_t, s_{t+1}) in M.
5. We take a random batch $B \subset M$ of transitions. For each transition $(s_{t_B}, a_{t_B}, r_{t_B}, s_{t_B+1})$ of the random batch B:
   - We get the prediction:

$$Q(s_{t_B}, a_{t_B})$$

   - We get the target:

$$r_{t_B} + \gamma \max_a (Q(a, s_{t_B+1}))$$

   - We get the loss:

$$\text{Loss} = \frac{1}{2}\left(r_t + \gamma \max_a (Q(a, s_{t+1})) - Q(a_t, s_t)\right)^2 = \frac{1}{2} TD_t(a_t, s_t)^2$$

   - We backpropagate this loss error and update the weights according to how much they contributed to the error.

Project Specifications

Custom roads/obstacles: Provided tools to make custom roads or obstacles in the environment. These are done by clicking and dragging the mouse through the screen. The affected pixels turn their value to 1.

Car: Created a class car which contains all the properties and methods for the car. It is used to define velocity, rotation and angle of the car at any instant of time.

Sensors: A car contains three sensors. They sense data from front, left and right. These are indicated through three dots.

Signals: These are the quantity used to detect the density of the pixels with value 1 in front of the sensors. This is done by taking a box of pixels in front of the sensors and adding all the pixels with value 1 and dividing them with total number of pixels.

Other features:

- Velocity gets slow at corners of the screen or at obstacles. This is done through reward system. All rewards lie between -1 to +1.
- A clear, load and save button to clear all the obstacles so as to create a new environment, load a saved model and to save a model.
- Taking the top left and right bottom as the goal state. The bottom left is considered as the origin.

Neural Network Architecture: Made the neural network architecture which would take the environment state vectors as input features and produce Q-values as the output.

Experience Replay: Used experience replay to store the past states or experiences which is used to make future decisions.

Source Code:

- For the environment map, the car and features.

```
# 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]
```

```python
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)
```

```python
    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_
        self.signal2 = int(np.sum(sand[int(self.sensor2_x)-10:int(self.sensor2_x)+10, int(self.sensor2_y)-10:int(self.sensor2_
        self.signal3 = int(np.sum(sand[int(self.sensor3_x)-10:int(self.sensor3_x)+10, int(self.sensor3_y)-10:int(self.sensor3_
        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
```

```python
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
```

```python
        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:
```

```python
# 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

```

```python
# 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...")
```

```python
        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()

```

- For Deep Q-learning, experience replay and neural network.

```python
# 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

# Creating the architecture of the Neural Network

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, nb_action)

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

```

```python

# Implementing Experience Replay

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)

```

```python

# Implementing Deep Q Learning

class Dqn():

    def __init__(self, input_size, nb_action, gamma):
        self.gamma = gamma
        self.reward_window = []
        self.model = Network(input_size, nb_action)
        self.memory = ReplayMemory(100000)
        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):
        probs = F.softmax(self.model(Variable(state, volatile = True))*100) # T=100
        action = probs.multinomial(num_samples = 1)
        return action.data[0,0]
```

```python
    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)]), torch.Tensor([self.last_rewar
        action = self.select_action(new_state)
        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) > 1000:
            del self.reward_window[0]
        return action

```

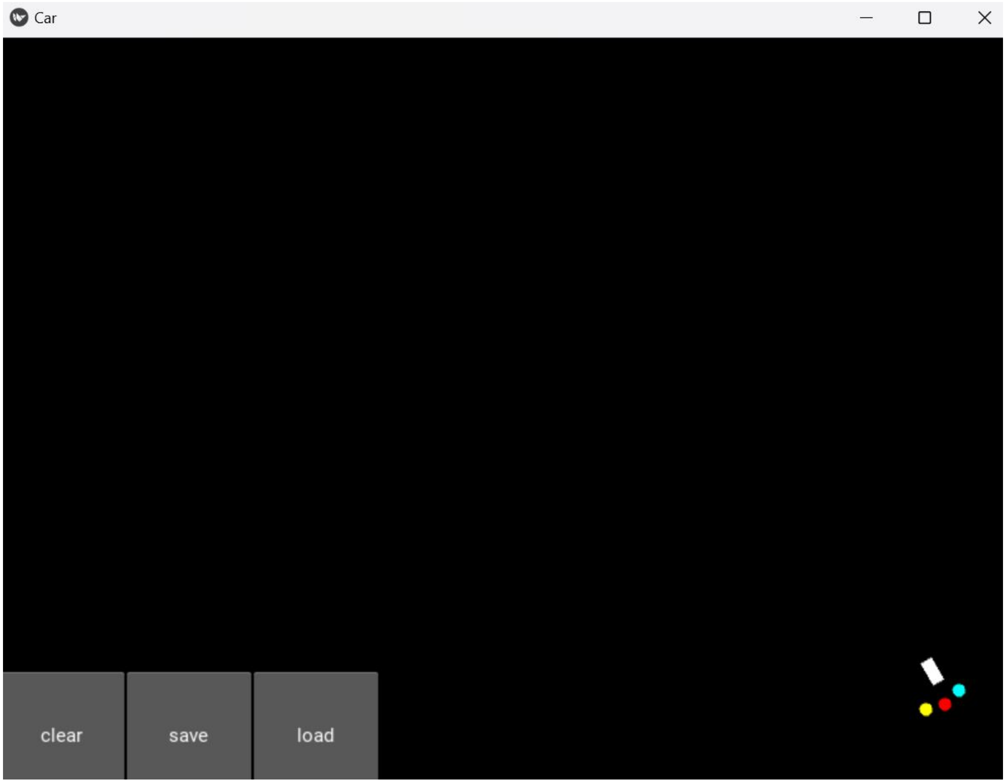
```python

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

    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...")
```

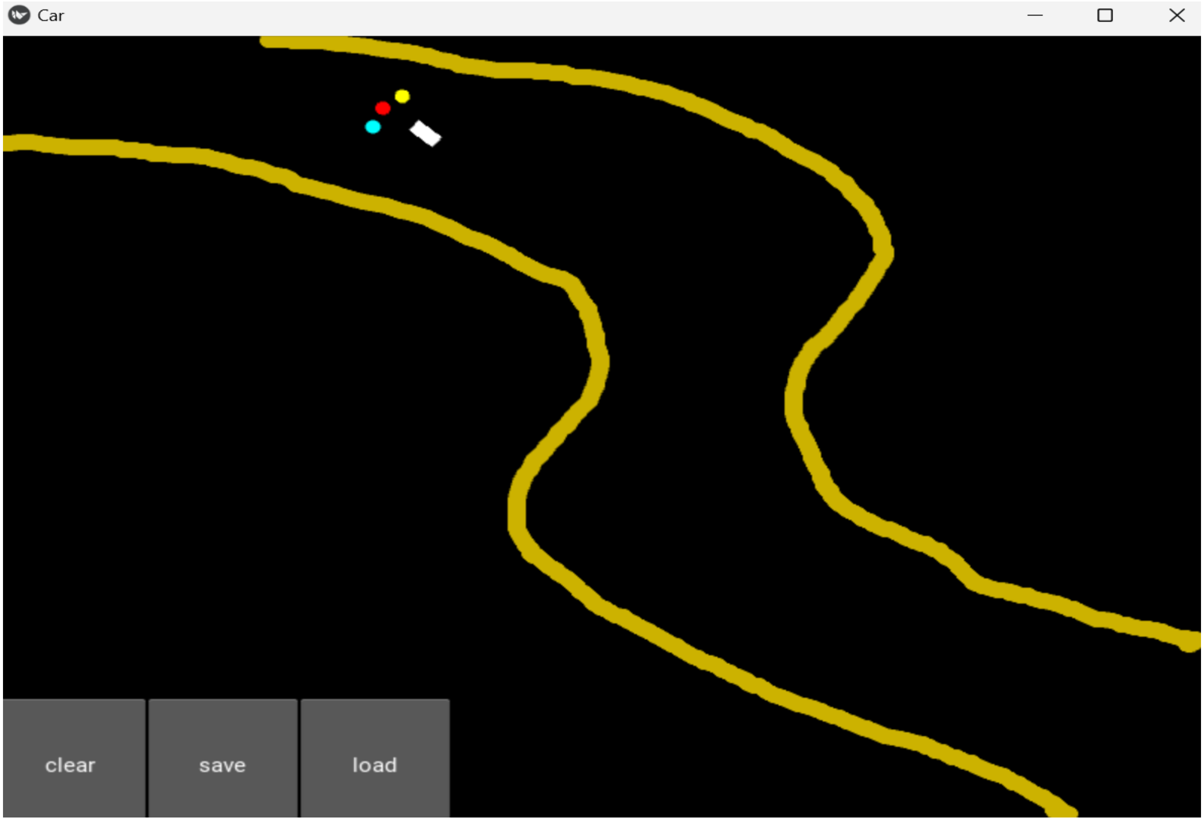
Results Screenshots:



The interface of car and environment

Creating Obstacle/Road using mouse

Experimenting with different roads and obstacles:



Conclusion and Future work

Deep Q- learning proved out to be an efficient method to learn and execute a self-driving car. The model generated superior results while handling random situations, similar to a real car on roads. The environment allowed the user to make custom obstacles giving the car difficult challenges. The car made sensible motion from its source to destination. The next steps should be to add more moving objects in the environment.