Traffic Simulator

Aditya Singh

February 14, 2022

1 Environment Generation

This Traffic Simulator[1] aims to simulate real time traffic scenarios which can aid in the training of various autonomous vehicles. The environment is 2-dimensional and relies on the readily avalaible pygame library in Python3, making it easy to use and not demanding on the user's system unlike some heavier simulation softwares like CARLA.

The user has to create his environments by following the given steps :

1. Simulate the road network
2. Simulate infinitely generated auto vehicles
3. Simulate single generated auto and ego vehicles

1.1 Simulate the road network

To first create our simulation, our user first has to create an object from the Simulation class. This object will hold all the necessary data of the roads, vehicles, traffic signs, etc.

```
sim = Simulation()
```

[1]GitHub Repo : https://github.com/aditya3434/TrafficSimulator

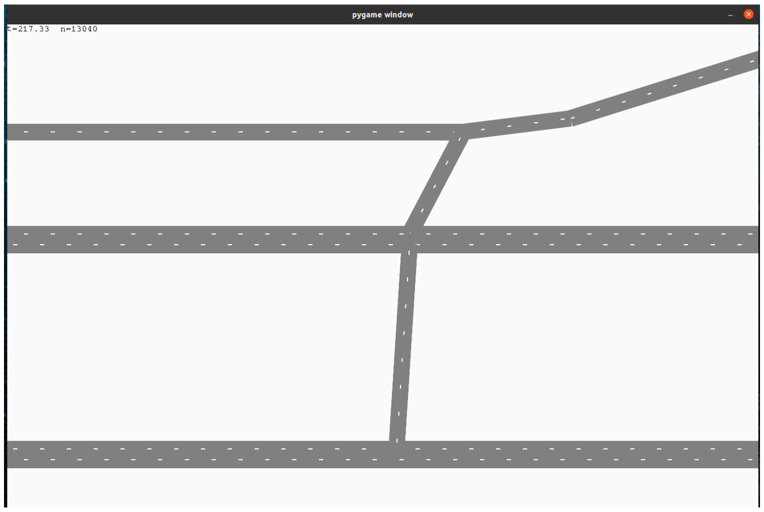
After which, the user will use the `create_roads()` function to enter an array of roads. Each road is defined by 2 (x, y) coordinates that signify the start and the end of the road respectively.

```
sim.create_roads([
    ((300, 98), (0, 98)),
    ((0, 102), (300, 102)),
    ((180, 60), (0, 60)),
    ((220, 55), (180, 60)),
    ((300, 30), (220, 55)),
    ((180, 60), (160, 98)),
    ((0, 178), (300, 178)),
    ((300, 182), (0, 182)),
    ((160, 102), (155, 180)),
    ((160, 98), (0, 98))
])
```

To run this simulation, we create a window by instantiating an object of the Window class by giving the simulation object as an argument. After which the user can use the various methods of said class to create offsets, resize, run window, etc. We will discuss about the `run()` function in detail later.

```
sim = create_sim()
win = Window(sim)
win.offset = (-150, -110)
win.run(steps_per_update=5)
```

Running this code will give us the following output.



```
t=217.33  n=13040
```

1.2 Simulate infinitely generated auto vehicles

As we all know, road traffic is always random and hundreds of vehicles are always entering and exiting different roads. Hence, to simulate such traffic, the user can use the `create_gen()` function to create a vehicle generator.

```
sim.create_gen({
    'vehicle_rate': 10,
    'vehicles': [
        {"path": [4, 3, 5, 9]},
        {"path": [4, 3, 2]},
        {"path": [1]},
        {"path": [6]},
        {"path": [7]}
    ]
})
```

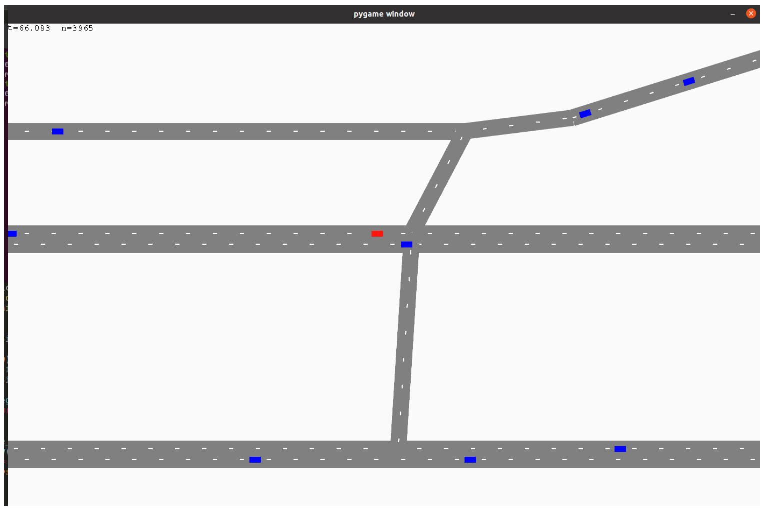
The generator will keep spawning vehicles listed in the array at a speed dependant on the `vehicle_rate`. Each vehicles is depicted in the format `[1, ... "path": [A]]` where A is the array of roads that form the path the vehicle is gonna take. For example, if the 3rd vehicle spawns, it'll drive along road 1. Whereas if the 2nd vehicle spawns, it'll go along road 4, then 3 and then till the end of road 2. These vehicles are "auto" meaning that they follow predetermined paths and can't take acceleration and steer values as their input.

1.3 Simulate single generated auto and ego vehicles

While spawning common vehicles infinitely seems alright, there might be a scenario where only one car is present on the road. Or in other cases, we might want to put emphasis on an action/ego vehicle whose maneuver we want to customize and analyze. For these vehicles, we use the `create_gen()` function to create a single vehicle generator, which only spawns 1 vehicle.

```
ego_vehicle = sim.create_single_gen({
    'auto' : False,
    'vehicle': {"acc": 0.5, "x": 0, "y": 98, "model": ...
        "Kinematic", "L": 2.9, "c_a": 2.0, "c_r": 0.01}
})
```

The single vehicle spawned can be either "auto" or "ego". These vehicles can take many attributes as input like acceleration, steer, driving type, etc. to give the car freedom of movement and can maneuver in the way the user wants. The figure below shows the vehicles `(Auto : Blue, Ego : Red)` running along the road network.



```
t=66.083  n=3965
```

2 Play testing and Training Models

Now that we know how to simulate the environment, we observe how we can use this environment to play test certain scenarios as well as train different machine learning models. This would include defining rules in our environments, writing the step function, passing different arguments in the `run()` function, and many other steps that we will study.

2.1 Play testing scenarios

When we run the traffic simulator, we can see a visual representation of how the ego an the auto vehicles are driving on the road. However, we can observe that the vehicles have no interaction with each other and the environment. The user might want to declare certain rules that when broken, ends the simulation (e.g. collisions, off-roading, etc). To introduce these types of rules, the user can introduce certain play testing functions which they can pass as an argument to the `run()` function of the Window class.

For instance, if we want the simulation to end when ego vehicles collide, we can create the following function.

```
def function(sim):
    ego1 = sim.action_vehicles[0]
    ego2 = sim.action_vehicles[1]

    # Check if the ego vehicles collide with each other or with ...
        any auto vehicles
    if sim.ego_collide(ego1, ego2) or sim.auto_collide(ego1) or ...
        sim.auto_collide(ego2):
        return True

# Create simulation
sim = create_sim()

# Create window to display sim
win = Window(sim)
win.offset = (-150, -110)

# Run the simulation in the window with the test function
win.run(function, steps_per_update=5)
```

If the function returns True at any point, the simulation ends. This function can be modified to check for any other simulation ending conditions like off-roading and scenario timeouts.

2.2 Training ML models

The test function designed above can also be used as a step function to train ML models. However since different models have different number of parameters, it is better if the user creates a child class of the `Window` class and overrides its run function. For instance, given below is the implementation of a DQN model to train the going straight maneuver.

We first create a child class `ModelWindow` of the `Window` class and implement our own run function.

```
class ModelWindow(Window):

    def __init__(self, sim, config={}):
        super().__init__(sim, config)

    def run(self, step=None, epi = 0, loss = None, model = None, ...
        steps_per_update=1):
```

```
        """Runs the simulation by updating in every loop."""
        def loop(sim):
            sim.run(steps_per_update)

            if step:
                done = False
                reward = 0
                new_model = None
                if model:
                    new_model, reward, done = step(sim, epi, ...
                        loss, model, steps_per_update)
                else:
                    done = step(sim)
                if done:
                    self.running = False
                return reward, new_model
            else:
                return 0, None

        return self.loop(loop)
```

The `loop()` function defined in `run()` should always return 2 values. The first one is the reward calculated by the model whereas the second return value can be an accumulation of relevant data depending on the model being used. In this case, we are returning the updated DQN model.

```
# Discrete choices for our vehicle
def choose_action(choice):
    if choice == 0:
        return [0, 0]
    elif choice == 1:
        return [0.3, 0]
    else:
        return [0.8, 0]

# Agent step function
def step(sim, epi, loss, model, steps_per_update):

    # Initializing reward and done state
    reward = 0
    done = False

    # Ego vehicle which we are training
    ego = sim.action_vehicles[0]

    # X-coordinate of ego vehicle
    x = ego.get_state()[1][0]

    # Velocity of ego vehicle
    vel = ego.get_state()[0]

    # Current state
    state = [x, vel]
    state = np.reshape(state, (1,2))

    # Model gives choice
    choice = model.act(state)
    action = choose_action(choice)
    ego.set_state(action)

    # Agent takes choice and updates environment
    sim.run(steps_per_update)

    # Analyzing new state
    x = ego.get_state()[1][0]
    vel = ego.get_state()[0]
    next_state = [x, vel]
    next_state = np.reshape(next_state, (1,2))

    # Calculating step reward
    reward -= (3-0.01*x)

    # Model learning
    model.remember(state, choice, reward, next_state, done)
    model.replay(done, epi, loss)

    # If ego vehicle reaches the end of the road, sim ends
    if x > 299:
        reward += 1000
        done = True

    return model, reward, done
```

Since we are using DQNs, we describe the discrete actions and write the step/reward function accordingly.

```
# Keeping track of scores
loss = []

# DQN model for our autonomous vehicle
straight_model = md.DQN(3,2,'straight_model')
x = []

# No. of training episodes
episodes = 500

# Training Loop
for i in range(episodes):
    # Create simulation
    sim = create_sim()

    # Create window to display simulation
    win = ModelWindow(sim)
    win.offset = (-150, -110)

    # Get total score an updated model after simulation ends
    score, straight_model = win.run(step, i, loss, ...
        straight_model, steps_per_update=4)

    # Append total score to list
    loss.append(score)
    print("Episode : ",i+1, " --> Score : ", score)
    x.append(i+1)

plt.plot(x, loss)
plt.savefig('train.png')
```

We then run our iterations to train our model and finally save its performance. Hence, our simulator is flexible enough to create various scenarios as well as train different machine learning models, while being extremely light-weight and not demanding unlike CARLA. This makes it the perfect tool to implement RL policies on autonomous vehicles very efficiently.