CS747: Programming Assignment 3

Aayush Rajesh
200070001

1 Task 1

1.1 First Approach

The first approach I took was to attempt an implementation of control using function approximation. My features were given by

$$f(s,a) = (x_{new}, y_{new}) \tag{1}$$

where the two quantities are given by,

$$\begin{aligned} x_{new} &= x + v_{new} \cdot \cos(\theta_{new}) \\ x_{new} &= y + v_{new} \cdot \sin(\theta_{new}) \\ v_{new} &= v + \text{acceleration} \\ \theta_{new} &= \theta + \text{steer} \end{aligned}$$

Here, v_{new} and θ_{new} are the expected car velocity and angle after taking a steer and acceleration action. It is important to note that the values of expected velocity, x_{new} and y_{new} do not represent the true state of the car after taking the action, but provide a *sense* of how the action will affect the current state in terms of position and velocity.

Using these features (after scaling down), I tried to implement the **Sarsa(λ)** algorithm. However, there was no evident convergence in the feature weights, which were taking too long to exhibit learning. This can be attributed to the abstractness of my selected features, as well as the constant nature of the rewards during episode play.

1.2 Second Approach

An intuitive notion for driving the car is to minimize the angle between its current motion and the direction it targets. My second try at an algorithm stemmed from this logic. I retained the calculation of the expected position and velocity from the previous attempt and looked to minimize the angle between the movement towards these expected coordinates and the direction of our target location (350, 0). More precisely, my single defining feature of a state and action was:

$$f(s,a) = \vec{\mathbf{h}} \cdot \vec{\mathbf{t}} \tag{2}$$

where $\vec{\mathbf{h}}$ is the "heading" vector and $\vec{\mathbf{t}}$ is the "target" vector given by

$$\vec{\mathbf{h}} = (x_{new} - x, y_{new} - y)$$
$$\vec{\mathbf{t}} = (350 - x, -y)$$

Given a state, different actions will result in different heading vectors, but the same target vector. Optimal motion will look to maximize the dot product between the two vectors, thus minimizing the angle between them. Since there is only one feature, there is no requirement for weights, since the approximated action-value function is proportional to the feature. Therefore, my algorithm proceeds by acting greedily with respect to the approximated action-value function.

This algorithm returned favorable results upon testing. However, there were a few cases where the car drove too close to the ice while attempting to maximize the action value function. A flaw that I noticed in this implementation was that the action capable of maximizing the dot product may also involve increasing the velocity of the car. This may lead to failing the task when the car is too close to the borders of the parking lot. The unpredictability in motion due to the icy surface may cause the car to overshoot its intended action and cross the border.

1.3 Final Approach

In my final approach to the problem, which resulted in a successful implementation, I used some of the takeaways from my previous attempts at the problem. I introduced another dimension to my feature vector, which was the cross-product between the heading and target vectors. That is, my feature vector was:

$$f(s,a) = \left(\vec{\mathbf{h}} \cdot \vec{\mathbf{t}}, \|\vec{\mathbf{h}} \times \vec{\mathbf{t}}\|\right) \tag{3}$$

The `newFeatures` function code listed below estimates these features for a given state and action.

```python
def newFeatures(self, state, actions):
    v = max(state[2] + self.acceleration[actions[1]]/5, 0)
    angle = state[3] + self.steer[actions[0]]
    x = state[0] + v*math.cos(angle*math.pi/180)
    y = state[1] + v*math.sin(angle*math.pi/180)
    #new = self.discrete([x, y])
    new = [x, y]

    dot_prod = np.dot([new[0]-state[0],new[1]-state[1]],[350-state[0],-state[1]])
    cross_prod = np.linalg.norm(np.cross([new[0]-state[0],new[1]-state[1]],
                 [350-state[0],-state[1]]))
    return np.array([dot_prod, cross_prod])
```

For optimal motion, the `dot_prod` should be maximized, while `cross_prod` should be minimized. Therefore, it is evident that the weights should be of the form (a, -b). A choice of weights (1, -1) (which looks to maximize `dot_prod` and minimize `cross_prod` with equal priority) gave successful results within a reasonable run time. However, it was possible to further optimize the algorithm by providing less priority on minimizing `cross_prod`, since the main goal of the algorithm is the move the car in the proper direction. Armed with this intuition, I managed to **tune** the weight parameters optimally to (1, -0.5) which produced smooth movement of the car irrespective of starting configuration.

The next_action function code which returns the optimal action to be taken for a given state is given below. Since we act greedily with respect to the approximated action-value function, it returns the action which maximizes the dot product between the weight and feature vectors. In case of multiple such actions, it randomly selects one action. Note that self.eps is set to 0 since random exploration is not needed given the high performance of my tuned weights.

```
def next_action(self, state, old_action):
    """
    Input: The current state
    Output: Action to be taken
    TO BE FILLED
    """

    # Replace with your implementation to determine actions to be taken

    action_steer = old_action[0]
    action_acc = old_action[1]
    if(np.random.random() < self.eps):
        action_steer = np.random.randint(self.nsteer)
        action_acc = np.random.randint(self.nacc)
    else:
        maxqval = -1e18
        for i in range(self.nsteer):
            for j in range(self.nacc):
                qval = self.actionValue(self.newFeatures(state, [i,j]), self.weights)

                if(abs(qval - maxqval) < 1e-4):
                    action_steer = np.random.choice([i, action_steer])
                    action_acc = np.random.choice([j, action_acc])
                elif(qval - maxqval > 0):
                    maxqval = qval
                    action_steer = i
                    action_acc = j

    action = np.array([action_steer, action_acc])

    return action
```

2 Task 2

Given the success of my algorithm for Task 1, it was natural to adopt a similar approach to Task 2, albeit with a few differences. My feature vector is modeled the same way as before, as given in Equation (3). The heading vector resulting from taking an action at a given state is also the same as that in my algorithm in Task 1. The target vector, however, has some key differences due to the nature of obstacles in the parking lot.

I have modeled all entities in the parking lot as **field sources**, similar to electrostatic fields. The point (350, 0) emanates an **attractive field** that follows an inverse law. All obstacles (mud pits and border of parking lot) emanate a **repulsive field** that follows an inverse-square law (the border emanates the field perpendicular to the border itself). The specific nature of these fields is due to the fact that the car prioritizes reaching the road over avoiding obstacles. Therefore, in most regions, the attractive field dominates the repulsive field. Furthermore, I have also ensured that the repulsive field of the mud pits, which are approximated to circles, is maximum at the circumference. This dissuades the car from entering a circular region surrounding each mud pit. The nature of these fields is summarized in Table 1.

Entity	Field
Road	$\frac{\hat{\mathbf{d}}}{d}$
Mud Pit	$-\frac{\hat{\mathbf{d}}}{d^2}$
Border	$-\frac{\hat{\mathbf{d}}}{d^2}$

Table 1: Field characteristics. $\hat{\mathbf{d}}$ points towards the entity.

Using this model of the environment, we can define the target vector at a point as the vector sum of the fields emanated by all sources at that point. This target vector now holistically captures both the negative effect of the environment (obstacles) as well as the positive effect (desired location). Taking this model into account, the code for the `newFeatures` function, which calculates the feature vector as per Equation (3), is given below. Furthermore, the vector field depicting the direction (and magnitude) of the target vector at different points in the environment is shown in Figure 1.

```python
def newFeatures(self, state, actions):
    v = max(state[2] + self.acceleration[actions[1]]/5, 0)
    angle = state[3] + self.steer[actions[0]]
    x = state[0] + v*math.cos(angle*math.pi/180)
    y = state[1] + v*math.sin(angle*math.pi/180)
    #new = self.discrete([x, y])
    new = np.array([x, y])
    heading = np.array([new[0]-state[0],new[1]-state[1]])
    target = np.array([350-state[0],-state[1]])
    target = target/(pow(np.linalg.norm(target), 2))

    for i in range(len(self.pits)):
        obst = np.array([state[0] - self.pits[i][0], state[1] - self.pits[i][1]])
        obst = obst/(pow(np.linalg.norm(obst)-73, 3))
        target += obst

    target += np.array([0, state[1] - 350])/
              (pow(np.linalg.norm([0, state[1] - 350]), 3))
    target += np.array([0, state[1] + 350])/
              (pow(np.linalg.norm([0, state[1] + 350]), 3))
    target += np.array([state[0] - 350, 0])/
              (pow(np.linalg.norm([state[0] - 350, 0]), 3))
    target += np.array([state[0] + 350, 0])/
              (pow(np.linalg.norm([state[0] + 350, 0]), 3))

    dot_prod = np.dot(heading, target)
    cross_prod = np.linalg.norm(np.cross(heading, target))

    return np.array([dot_prod, cross_prod])
```

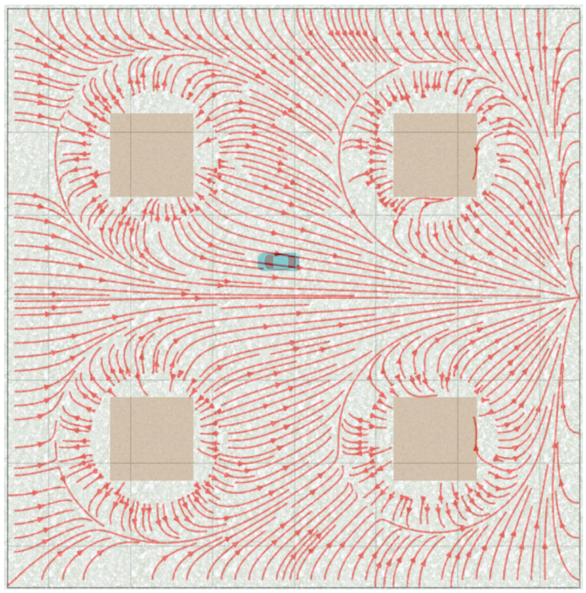

Figure 1: Target vector ($\vec{\mathbf{t}}$) field

The weight vector remains the same as that in Task 1 since the implemented trade-off between maximizing the dot product and minimizing the cross product yields optimal results. Similarly, there is no change in the code for `next_action` function.

3 Final Note

One striking approach I have taken in both of my successful attempts to both Task 1 and 2, is that I **do not** look to exactly approximate the action-value function through my choices of features and weights. Instead, I look to appropriately model the action value, so that the action taken by being greedy with respect to the action-value function is close to the optimal action for that state. In short, my estimate $\hat{Q}$ aims to be such that

$$\arg\max_a\{\hat{Q}(s,a)\} = \arg\max_a\{Q^*(s,a)\}$$

whereas function approximation looks for

$$\hat{Q}(s,a) \approx Q^*(s,a)$$

By aiming for the former, rather than the latter, I was able to rely on a sense of intuition while identifying characteristic features, and tuning feature weights.