Automatic Parking Using Reinforcement Learning

Tao Chen

Shanghai LingXian Robotics

Nov 29, 2016

Outline

Working Principle

Source Code

Training

Outline

Working Principle

Source Code

Training

Reinforcement Learning Framework

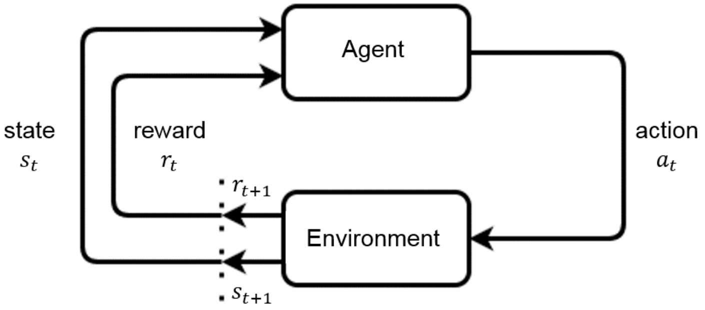

Figure: Reinforcement Learning Framework

Q-Learning

- What is Q-Learning?
  - A model-free reinfocement learning technique
  - Work by learning an action-value function that ultimately gives the expected utility of taking a given action in a given state and following the optimal policy thereafter
  - The utility function tells how good an action is given a certain state
  - An optimal policy selects an action with the highest utility value in each state

Q-learning has been proven to converge to the optimum action-values so long as all actions are repeatedly sampled in all states and the action-values are represented discretely.

Q-Learning Algorithm

- Initialize Q-values ($Q(s, a)$) arbitrarily for all state-action pairs
- While not stopped:
  - Choose an action $\boldsymbol{a}$ in the current state $\boldsymbol{s}$ based on current Q-value estimates ($\boldsymbol{Q(s, *)}$)
  - Execute the action $\boldsymbol{a}$, receive immdediate reward $\boldsymbol{r}$, observe the outcome state $\boldsymbol{s'}$
  - Update the table entry for $\boldsymbol{Q(s, a)}$:

$$Q(s, a) = Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$$

  - $\boldsymbol{s = s'}$
- Remark: α: learning rate, γ: discount factor

Exploration and Exploitation

- **Exploitation:** make best decision given current information
  - short-term, immediate, certain benefits
- **Exploration:** gather more information
  - long-term, risky, uncertain
- **All Exploitation:** locked-in to suboptimal equilibria (local maxima), cannot adapt to changing circumstances
- **All Exploration:** costs of experimentation without any of its benefits
- The best **long-term strategy** may involve **short-term sacrifies**
- Gather **enough information** to make the best overall decisions

ϵ-greedy algorithm

- Greedy Algorithm: an algorithm that always takes whatever action seems best at the present moment
- Epsilon Greedy Algorithm: an algorithm that generally exploits the best available option, but has probability ϵ to randomly explore action space

$$a = \begin{cases} \underset{a}{argmax}(Q(s,*)) & \text{with probability } 1-\epsilon \\ random\ action & \text{with probability } \epsilon \end{cases}$$

Q-Learning Parameters

- **Learning rate α:** determine to what extent the newly acquired information will override the old information
  - 0: make the agent not learn anything
  - 1: make the agent consider only the most recent information
- **Discount factor γ:** determine importance of future rewards
  - 0: make the agent short-sighted by only considering current rewards
  - 1: make the agent strive for a long-term high reward

Outline

Working Principle

Source Code

Training

Auomatic Parking Agent Source Code

```
def update(self):
    agent_pose = self.env.sense()
    self.state = states(x = agent_pose[0], y = agent_pose[1], theta = agent_pose[2])

    step = self.env.get_steps()
    if self.env.enforce_deadline:
        deadline = self.env.get_deadline()

    # Select action according to the policy
    action, max_q_value = self.get_action(self.state)

    # Execute action and get reward
    next_agent_pose,reward = self.env.act(self, action)

    # Learn policy based on state, action, reward
    if not self.test:
        if self.prev_action != None:
                self.update_q_values(self.prev_state, self.prev_action,
                                     self.prev_reward, max_q_value)

        if self.env.enforce_deadline:
            print "LearningAgent.update(): step = {}, deadline = {},
                   state = {}, action = {}, reward = {}".format(step, deadline,self.state,
                   action, reward)
        else:
            print "LearningAgent.update(): step = {}, state = {},
                   action = {}, reward = {}".format(step, self.state, action, reward)

    self.save_state(self.state, action, reward)
```

Auomatic Parking Agent Source Code

```
def update_q_values(self, prev_state, prev_action, prev_reward, max_q_value):
    old_q_value = self.Q_values.get((prev_state, prev_action), self.default_q)
    new_q_value = old_q_value + self.learning_rate * (prev_reward + self.gamma * max_q_value
                  - old_q_value)
    self.Q_values[(prev_state, prev_action)] = new_q_value
```

```
def get_maximum_q_value(self, state):
    q_value_selected = -10000000
    for action in car_sim_env.valid_actions:
        q_value = self.get_q_value(state, action)
        if q_value > q_value_selected:
            q_value_selected = q_value
            action_selected = action
        elif q_value == q_value_selected:
        # if there are two actions that lead to same q value,
        # we need to randomly choose one between them
            action_selected = random.choice([action_selected, action])
    return action_selected, q_value_selected
```

Auomatic Parking Agent Source Code

```
def get_action(self, state):
    if random.random() < self.epsilon:
        action_selected = random.choice(car_sim_env.valid_actions)
        q_value_selected = self.get_q_value(state, action_selected)
    else:
        action_selected, q_value_selected = self.get_maximum_q_value(state)
    return action_selected, q_value_selected
```

```
def get_q_value(self, state, action):
    return self.Q_values.get((state,action), self.default_q)
```

Auomatic Parking Environment Source Code

```
def sense(self):
    agent_pose = np.zeros(3) #[x, y, theta]
    agent_pose[0] = self.agent_pos[0]
    agent_pose[1] = self.agent_pos[1]
    agent_pose[2] = np.arctan2(self.agent_ori[2], self.agent_ori[3]) * 2
    agent_pose[2] = (agent_pose[2] + 2 * np.pi) % (2 * np.pi)
    agent_pose[0] = np.floor((np.floor(agent_pose[0] / (self.grid_width / 2)) + 1) / 2) *
                    self.grid_width
    agent_pose[1] = np.floor((np.floor(agent_pose[1] / (self.grid_width / 2)) + 1) / 2) *
                    self.grid_width
    idx = np.floor(agent_pose[2] / (self.angle_blockwidth / 2))
    if idx % 2 == 0:
        idx = idx / 2
    else:
        idx = (idx + 1) / 2
    agent_pose[2] = idx % 16
    # agent_pose[2] represents the region the car's angle belongs to
    # [-11.25, 11.25) is region 0
    # [11.25, 33.75) is region 1
    # ...
    # [-33.75, -11.25) is region 15
    return agent_pose
```

Auomatic Parking Environment Source Code

```
def reset(self):
    self.done = False
    self.t = 0

    x, y, z, theta = self.generate_agent_pose()
    self.reset_world(x,y,theta)
    if self.enforce_deadline:
        self.deadline = self.cal_deadline(x, y)
    print '   agent starting pose:', x, y, theta
```

Auomatic Parking Environment Source Code

```
def step(self):
    # Update agent
    self.agent.update()

    if self.done:
        return

    if self.agent is not None:
        if self.t >= self.hard_time_limit:
            print "Environment.step(): Primary agent hit hard time limit! Trial aborted."
            self.done = True
            self.num_hit_time_limit += 1

        elif self.enforce_deadline and self.t >= self.deadline:
            print "Environment.step(): Primary agent ran out of time! Trial aborted."
            self.done = True
            self.num_out_of_time += 1

        self.t += 1
```

Auomatic Parking Environment Source Code

```
def act(self, agent, action):
    self.set_agent_velocity(self.valid_actions_dict[action])
    tiem.sleep(self.step_length / self.speed)
    self.set_agent_velocity(np.array([0,0]))
    reward = 0.0

    agent_pose = self.sense()

    if self.hit_wall_check(agent_pose):
        self.hit_wall_times += 1
        self.done = True
        reward = -20.0

    elif self.reach_terminal(agent_pose):
        if self.t < self.hard_time_limit and self.t < self.deadline:
            reward = 40.0
            self.done = True
            self.succ_times += 1
            print '----------------------------------------------------------------------'
            print '----------------------------------------------------------------------'
            print '----------------------------------------------------------------------'
            print '----------------------------------------------------------------------'
            print "Environment.act(): Agent has reached destination!"


    elif self.fixed_car_movement_check():
        self.hit_car_times += 1
        self.done = True
        reward = -5.0

    return agent_pose, reward
```

Outline

Training Stages

- Training is separated into three stages:
  - stage one: far region to close region
  - stage two: close region to region adjacent to fixed car
  - stage three: region adjacent to fixed car to terminal

Stage One

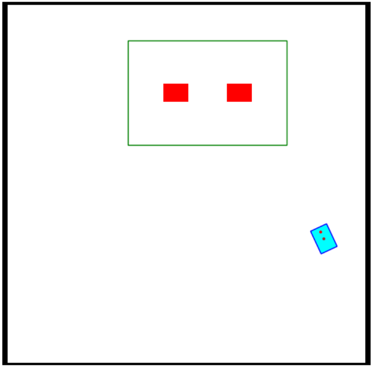

Figure: Stage One

Stage Two

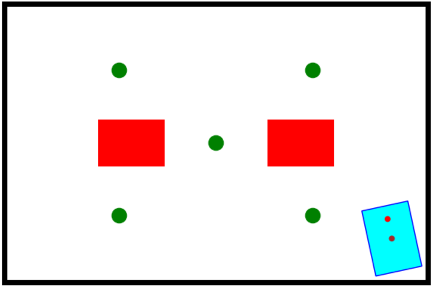

Figure: Stage Two

Stage Three

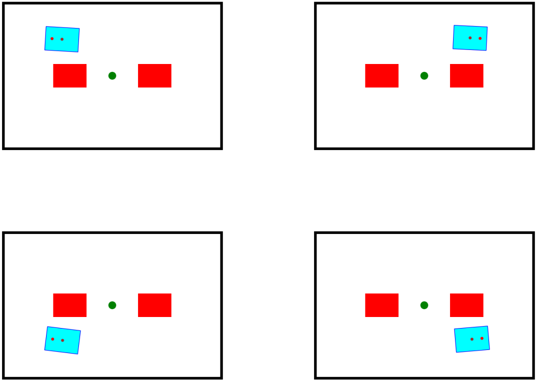

Figure: TOP LEFT, TOP RIGHT, BOTTOM LEFT, BOTTOM RIGHT

Stage One Training

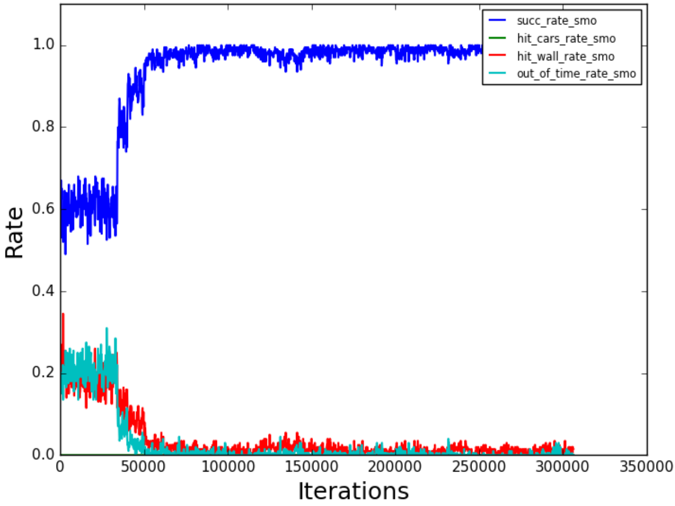

Figure: Stage One Training

Stage Two Training

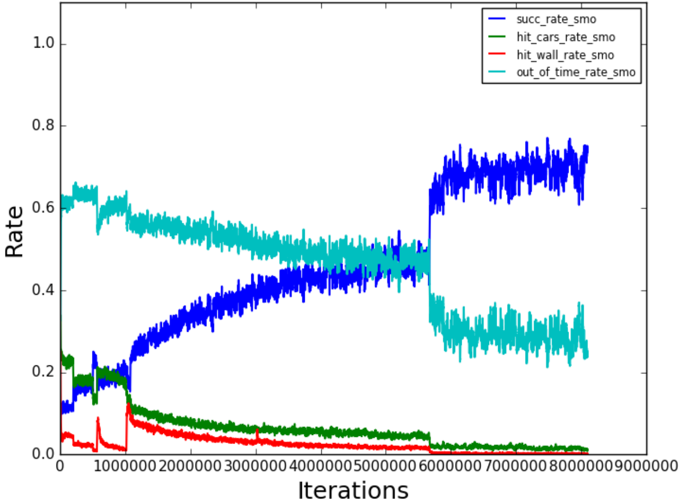

Figure: Stage Two Training

Stage Three Training

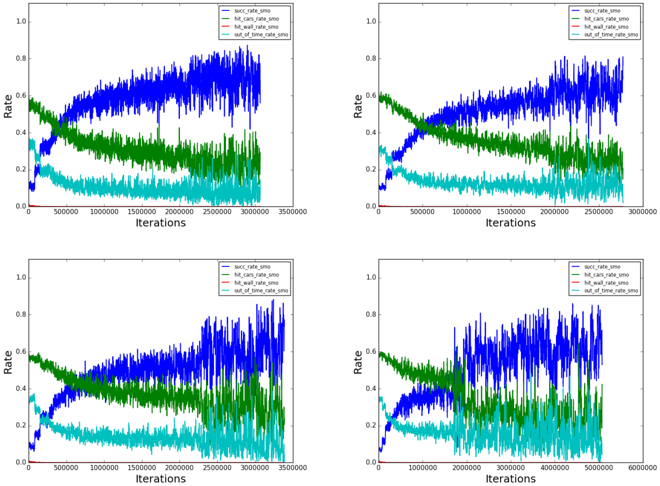

Figure: TOP LEFT, TOP RIGHT, BOTTOM LEFT, BOTTOM RIGHT

Combining Three Stages

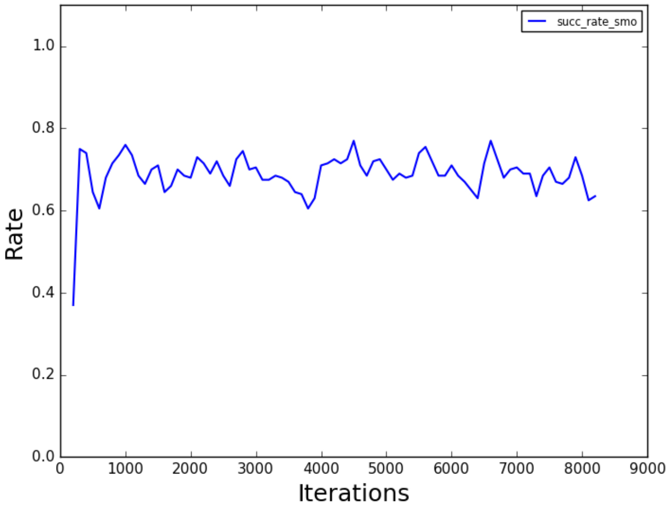

Figure: Combining Three Stages

Effect of ϵ

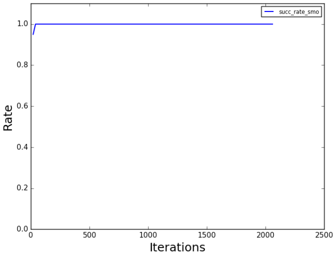

Figure: $\epsilon = 0$

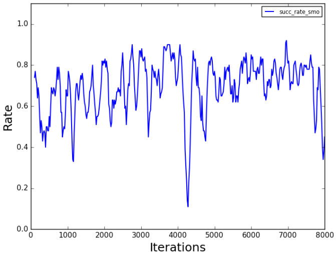

Figure: $\epsilon = 0.05$

Effect of ϵ

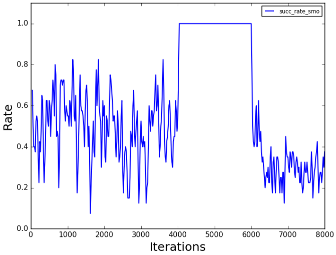

Figure: $\epsilon = 0.1$

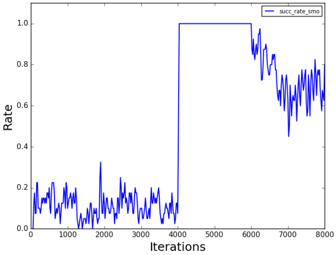

Figure: $\epsilon = 0.4$

Procedure:

- use the ϵ specified above to train the agent starting from top left region to the terminal for 4000 episodes
- test the agent with $\epsilon = 0$ for 2000 episodes
- add noise to the agent's initial angle ($\pm 11.25^\circ$ for 2000 episodes)