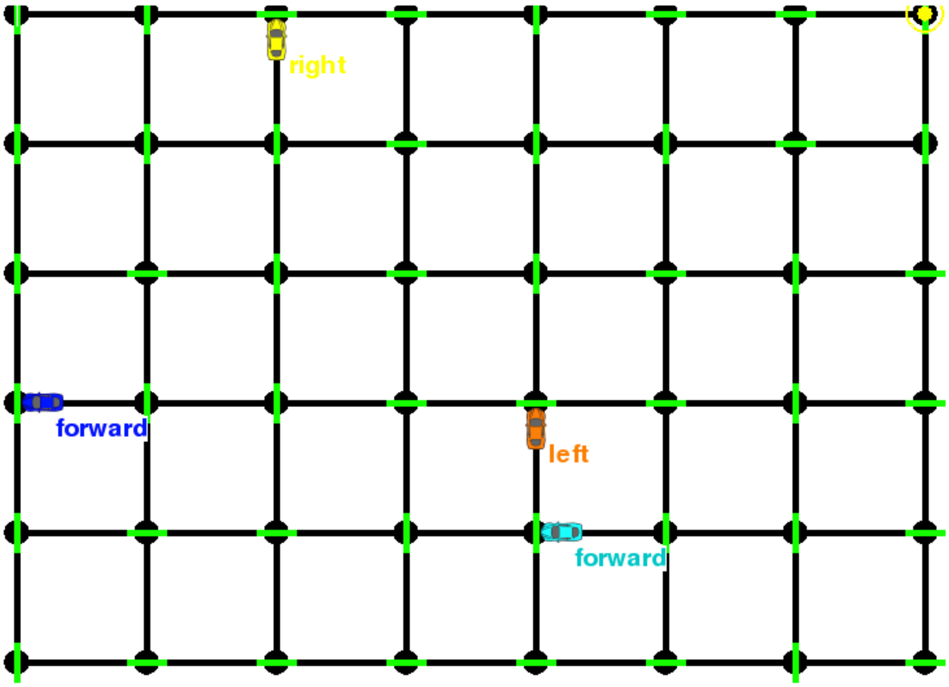
TRAIN A SMARTCAB HOW TO DRIVE

Project 3: Reinforcement Learning

- A smartcab is a self-driving car from the not-so-distant future that ferries people from one arbitrary location to another. This project utilizes reinforcement learning to train a smartcab how to drive. Design an AI driving agent for the smartcab. It should receive an input at each time step t, and generate an output move. Based on the rewards and penalties it gets, the agent should learn an optimal policy for driving on city roads, obeying traffic rules correctly, and trying to reach the destination within a goal time.

Environment

- The smartcab operates in an idealized grid-like city, with roads going North-South and East-West. Other vehicles may be present on the roads, but no pedestrians. There is a traffic light at each intersection that can be in one of two states: North-South open or East-West open. US right-of-way rules apply: On a green light, you can turn left only if there is no oncoming traffic at the intersection coming straight. On a red light, you can turn right if there is no oncoming traffic turning left or traffic from the left going straight. At any instant, the smartcab can either stay put at the current intersection, move one block forward, one block left, or one block right (no backward movement). The smartcab gets a reward for each successfully completed trip. A trip is considered "successfully completed" if the passenger is dropped off at the desired destination (some intersection) within a pre-specified time bound (computed with a route plan). It also gets a smaller reward for each correct move executed at an intersection. It gets a small penalty for an incorrect move, and a larger penalty for violating traffic rules and/or causing an accident. Below is a screenshot of the grid world (5X7) used in this project.



Inputs

- Assuming that a higher-level planner assigns a route to the smartcab, splitting it into waypoints at each intersection. And time in this world is quantized. At any instant, the smartcab is at some intersection. Therefore, the next waypoint is always either one block straight ahead, one block left, one block right, one block back or exactly there (reached the destination). The smartcab only has an egocentric view of the intersection it is currently at (no accurate GPS or global location). It is able to sense whether the traffic light is green for its direction of movement (heading), and whether there is a car at the intersection on each of the incoming roadways (and which direction they are trying to go). In addition to this, each trip has an associated timer that counts down every time step. If the timer is at 0 and the destination has not been reached, the trip is over, and a new one may start.

Implement a basic driving agent

- Implement the basic driving agent, which processes the following inputs at each time step:
  - Next waypoint location, relative to its current location and heading
  - Intersection state (traffic light and presence of cars)
  - Current deadline value (time steps remaining)

- And produces some random move/action (None, 'forward', 'left', 'right'). The aim is not to implement the correct strategy; this is what the agent is supposed to learn.

Question 1: Observing the agent's behavior. Does it eventually make it to the target location?

- Given the valid actions the smartcab can take (None, forward, left, right) and the valid_inputs (light, oncoming, left, right), the goal for the agent is to reach the final destination by picking the optimal actions. In this instance we see the agent's actions are completely at random, regardless of the conditions of the environment. For example, there are numerous examples throughout the iterations where the agent does not learn that running a red light results in a negative reward, thus this action should not be repeated. Another case would be when the light turns green and there is oncoming traffic, the action the agent takes is 'None', translating into a negative reward. Majority of the time the smartcab does not reach the destination within the deadline, but in few cases it does. Calculating the success rate (number of times the agent reaches the destination per number of trials), nearly 23% of the time the smartcab reaches the destination in the allotted time (n_trials=100). Furthermore, the number of negative regards (or punishments) were tallied and this refers to the number of penalties the agent makes over the course of each trial and the given time step (i.e. running red light, left turn with oncoming traffic, etc.). The results show that 723 penalties were made out of 2716 time steps; this translates to 26.12% of the time, a penalty were induced. As we see, the performance of the smartcab is rather poor as expected given the actions were at random, thus this will act as our benchmark for comparing how much the agent improved by implementing Q-learning.

Identify and update state

- Identify a set of states that are appropriate for modeling the driving agent. The main sources of state variables are current inputs, but not all of them may be worth representing. Also, you can choose to explicitly define states, or use some combination (vector) of inputs as an implicit state. At each time step, process the inputs and update the current state. Run it again (and as often as you need) to observe how the reported state changes through the run.

Question 2: Justify why you picked these set of states, and how they model the agent and its environment.

- The following states are considered in the model:
  - {Light: green, red}
  - {Oncoming: None, forward, left, right}
  - {Left: None, forward, left, right}
  - {Right: None, forward, left, right}
  - {Way_point: None, forward, left, right}

- The four states listed above (light, oncoming, right, left, next_waypoint) gives the agent a good start for learning the surrounding environment and traffic laws by observing numerous variation of the environment at each intersection. For instance, the smartcab needs to know if the light is red/green and whether it should proceed through the intersection or stop. If the light is red, the cab can turn right if there is no oncoming traffic. Alternatively, if the light is green, the cab can turn left (assuming no turn signal) if there is no oncoming traffic. Hence the importance of these four states being included in the model so that the cab can learn and obey the traffic laws. The agent will learn by visiting every state multiple times in order to learn the value of each action. If too many states are present, the time it will take for the agent to learn will be significantly longer. However, if there are only a few states then it may be unable to determine between which actions to take. It should also be stated that the rewards are function of both the traffic light and next_waypoint; to simplify the problem, traffic conditions are not considered.

Implement Q-Learning

- The Q-Learning algorithm is will now be implemented by initializing and updating a table/mapping of Q-values at each time step. Instead of randomly selecting an action as before, the agent will now pick the best action available from the current state based on Q-values, and return that. Each action generates a corresponding numeric reward or penalty (which may be zero). The reinforcement learning technique is applied and parameters such as learning rate, discount factor, action selection method (epsilon-greedy) parameters are tweaked in order to improve the performance of the agent. The goal is to get it to a point so that within 100 trials, the agent is able to learn a feasible policy - i.e. reach the destination within the allotted time, with net reward remaining positive.

Question 3: What changes do you notice in the agent's behavior?

- First, the Q-Learning is an algorithm that is used during reinforced learning where the agent attempts to learn what the optimal policy is from its history of interacting with the environment. The history of the agent is just a sequence of the state-action-reward-next state <s, a, r, s`>. Q-learning is a method in which the agent learns by trial and error through interacting with the environment. At each step, an action is chosen such that it maximizes the function Q(state, action); where Q is defined as the estimated utility function, meaning it tells us how good the action is given the corresponding state. A more general explanation of the equation shown below is that Q(s, a) at a given time step is the sum of the immediate reward r(s, a) plus the best utility (Q) for the subsequent state. The algorithm begins by assigning each state an estimated vale (i.e. zero or non-zero), which is referred to as the Q-value. When a state is visited, a reward is assigned and this is used to update the estimate of the value of that state. The algorithm stores a Q-value for each state-action pair Q(s, a) in either a matrix or a table. In every state the Q-Learning algorithm visits it will determine what is the optimal action to take based on the selection policy. The steps Q-Learning steps are listed below, and this process is repeated until eventually the approximation converges to the true Q function, however every state-action pair needs to be visited many times over. Just to note, this can require a large amount of memory given a situation where there are numerous states and actions. Also, only the state-action pairs that were visited are updated. In some cases, the agent may end up always taking the best path it knows of, however this may not mean there are other paths that are perhaps even better. Therefore allowing the agent to explore a new path and not the same path over and over is addressed through the ε-greedy policy. As previously discussed, the Q-learning algorithm will converge towards the solution that is the best action in the respected state. The caveat to Q-Learning is that you have to explore enough of the grid world to eventually reduce the learning rate over time while in the meantime using what the agent already knows to get a high reward, which is referred to as exploitation. I implemented the ε-greedy (ε = epsilon) method for selecting the optimal action. The ε-greedy approach chooses an action with the highest Q(s, a) with a probability of (1-ε) and explores with a probability ε, given the state s. This is a binary probability approach where a coin is flipped at every time step and then decides to make either a random action (exploration) or follow the optimal policy given the inequality -> if epsilon is greater than some random integer (0<x<1), the agent will then act randomly; meaning a random action is made a fraction of the time, else follow the current best Q value. The equation is given below for how the Q-values are updated at a given time. [Source: "*Q-Learning for a Simple Board Game*", Arvidsson and Wallgren, 2010]

Q-Learning Algorithm Steps:

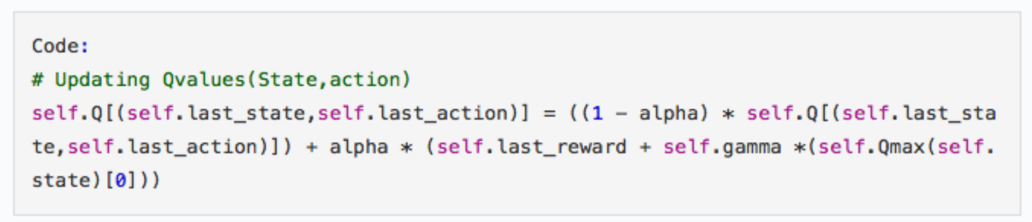
1. Initialize the Q-values; Q(s, a)
2. Observe the current state (s)
3. For the current state, choose an action based on the selection policy (ε-greedy)
4. Take the action, and observe the reward and as the new state (s')
5. Update the Q-value for the state using the observed reward and the maximum reward possible for the next state.
6. Set the state to the new state, and repeat steps until smartcab arrives at destination

Equation:

$$Q(s_t, a_t) \leftarrow Q(s_t, \alpha_t) + \alpha * [r_t + \gamma * max_\alpha Q(s_{t+1}, a) - Q(s_t, a_t)]$$

Where:

- $\boldsymbol{\gamma}$ = Discount Factor – relative value of delayed vs immediate reward. Future rewards are worth less than the current reward; 0< γ<1, as γ -> 0, the agent ignores all future rewards
- $\boldsymbol{\alpha}$ = Learning Rate; closer to 1, means the agent will only consider future rewards with greater weight. As alpha approaches 0, the agent is not capable of learning new information and only previous experience is used to make a new action.
- $\boldsymbol{r_t}$**=** The action gives feedback from the environment (positive or negative reward); negative reward which is considered a penalty. If the agent reaches the destination on time, a reward of 10 is assigned; if the agent makes a penalty, such as running the red light, a reward of -1 is given.
- $\boldsymbol{r_t + \gamma * max_\alpha Q(s_{t+1}, a)}$ = Expected discounted feedback
- $\boldsymbol{Q(s_t, a_t)}$ = Old Q-Value
- $\boldsymbol{max_\alpha Q(s_{t+1}, a)}$ = Max future Q-value
- $\boldsymbol{max_\alpha}$: Maximum reward attainable in s` following the current one; basically the reward for taking the optimal subsequent action

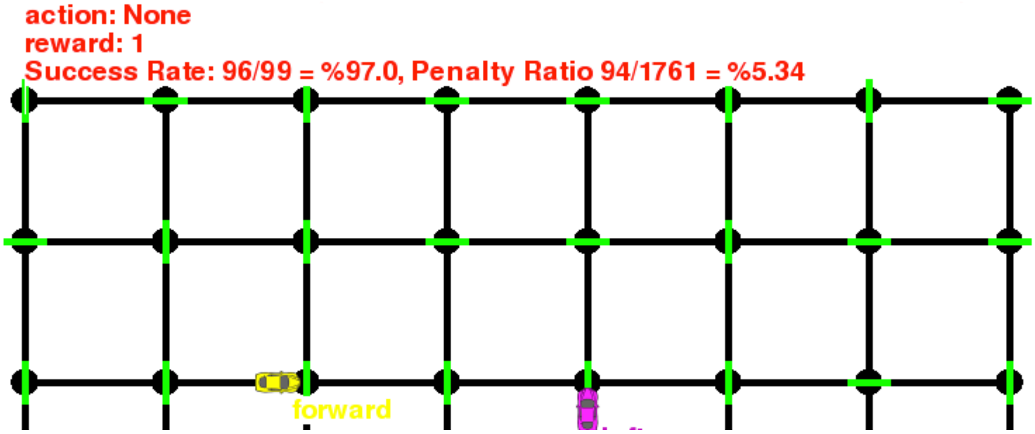


```
Code:
# Updating Qvalues(State,action)
self.Q[(self.last_state,self.last_action)] = ((1 - alpha) * self.Q[(self.last_sta
te,self.last_action)]) + alpha * (self.last_reward + self.gamma *(self.Qmax(self.
state)[0]))
```

Results: After implementing the Q-Learning algorithm, the results were significantly improved in contrast to the initial configuration where the agent was selecting all possible actions at random. The results show that after running n_trials = 100, the success rate improved from 23% to 95-97% with a penalty (negative reward) occurring ~5% of the total time.

Question 4: Report what changes you made to your basic implementation of Q-Learning to achieve the final version of the agent. How well does it perform?

- The main change after implementing Q-Learning to assist the agent in making the optimal action was applying a decay rate for the learning curve (α) and epsilon (ε) over time. If epsilon is decaying over time, then the initial actions selected will be more random since ε is starting off at unity and decaying from there. At first, I adjusted both α and ε values but when altering the respected parameters, the maximum success rate (agent reaching the destination on time) observed did not exceed 70%. At first, the parameters (α, ε) were set up such that they each decay with respect to the trial number. However, this approach yielded a poor success rate and thus was not considered in the analysis. Instead of decaying the parameters with respect to the trail number, a decline rate of f(x) = 1/time_step was applied for each iterative time step (α and ε were resets for each trial number). The success rate improved, ranging between 80-90%. In addition to making the adjustment to the decay rate, the initialization of the Q-values was also changed from 0 to 1.0, thus affecting the degree of exploration. This change resulted in the success rate increasing further to now range between 90-97% with a corresponding penalty ratio of 5.30% as shown in the below screenshot from the terminal window. Not considered in the analysis, but could applying a decay rate to gamma (discount factor) could also be examined since this parameter is associated with future rewards, and given the environment conditions, it is unclear what reward will be assigned to the agent (i.e. traffic lights behave random).



```
action: None
reward: 1
Success Rate: 96/99 = %97.0, Penalty Ratio 94/1761 = %5.34
```

Question 5: Does your agent get close to finding an optimal policy, i.e. reach the destination in the minimum possible time, and not incur any penalties?

- From the results, we see that out of nearly 1600 time steps (n_trials = 100) that ~100 penalties were committed. This means that roughly 97% of the time no traffic laws were violated during the simulation. Some of the penalties that occurred consisted of the agent taking a left turn or proceeding forward in the intersection when the traffic light was red; the agent learns over time not to take an action such that it generates a negative reward (-1). The error rate is still significantly small (~5%), however performing more trials/episodes could help the agent learn not to commit the penalties and could ultimately reduce the penalty rate even further. And finally, with the high success rate (~95-97%) of reaching the destination in the allotted time (with a discount factor of 0.10) in addition to reaching the destination in fewer steps, it would be reasonable to say that the agent has converged to the optimal policy.