MonteCarloMethods

November 8, 2018

1 Monte Carlo Methods

In the previous notebook we saw how we can use dynamic programming to solve the RL problem when we have a perfect model of the world i.e. when we have $P^a_{ss'}$ and $R^a_{ss'}$. Unfortunately, for many RL problems, we do not have access to a perfect model of the world. Monte Carlo (MC) methods circumvent this problem by relying on experience of the world. The general premise behind MC methods is that we can sample the environments dynamics by acting in the world and then use this information to estimate a value function. The beauty of this is that we can still obtain an optimal policy without any explicit understanding of the environment's dynamics. Conceptually you can think of this as using trial and error to solve the RL problem, whereby information about the environment is simply backed up into our value function estimates as we sample the environment's dynamics.

More specifically, MC methods are methods that sample the full return R_t for a given state or state-action pair either through real or simulated experience. They then average these samples to obtain a value estimate for the state or state-action pair. Recall that the return is defined as some function of the reward sequence that the agent experiences from time t onwards e.g.

$$\begin{aligned} R_t &= r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... \\ &= \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \end{aligned}$$

and that the value of a state or state-action pair is just the expected value of this return:

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_\pi[R_t \mid s_t = s] \\ Q^\pi(s,a) &= \mathbb{E}_\pi[R_t \mid s_t = s, a_t = a] \end{aligned}$$

Therefore all a MC method has to do is record the reward values during its sampling of the environment and use them to calculate the average return. An important limitation with this approach is that it can only be used for episodic tasks, where there is a terminal state and interactions with the environment can be divided into clear episodes. This restriction is required so that the returns are well-defined i.e. the return R_t is a sum up to some finite number rather than ∞. We shall see in the next notebook that other methods, such as temporal difference (TD) learning, also rely on

sampled experience to solve the RL problem but do not suffer from the episodic restriction because they bootstrap information. With that being said, it is still useful to explore MC methods because they involve some interesting concepts that will be useful in understanding more complex solution methods later on.

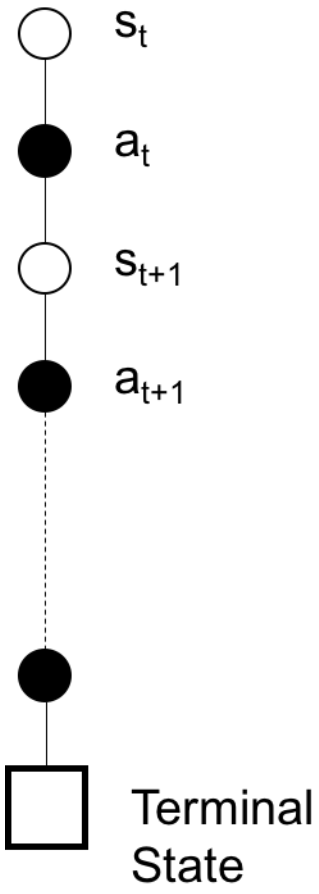
As always, we can view MC methods as a form of Generalized Policy Iteration (GPI) and so we shall now cover them in more detail using GPI as our framework.

1.1 Policy Evaluation

As already mentioned, Monte Carlo methods use sample returns from a state or state-action pair to calculate their value with respect to a certain policy. As more sample returns are collected for a state or state-action pair, the average of those returns should converge to the expected return, which by definition is the true value. The backup diagram for MC policy evaluation can be seen below:

```
In [3]: from IPython.display import Image
        Image(filename='MCBackup.png')
```

```
Out[3]:
```



This is known as a sample backup because it uses just one outcome or trajectory to form the backup. In comparison, dynamic programming methods use full backups because they average over all possibilities from a state or state-action pair, based on the probability of the outcome occuring. In addition, we also say that this backup is a deep backup because it samples all the way to the terminal state of an episode. Dynamic programming used shallow backups because it only looked one state or state-action pair ahead to formulate a backup. This highlights one of the key differences between dynamic programming and MC methods; MC methods don't bootstrap information. In MC methods, the estimated value for each state or state-action pair is independent of the estimated value for other states or state-action pairs because they sample all the way to the end of the episode. This is a particularly useful property when you only want to evaluate a subset of states or state-action pairs because you don't need to waste computation on evaluating the other ones.

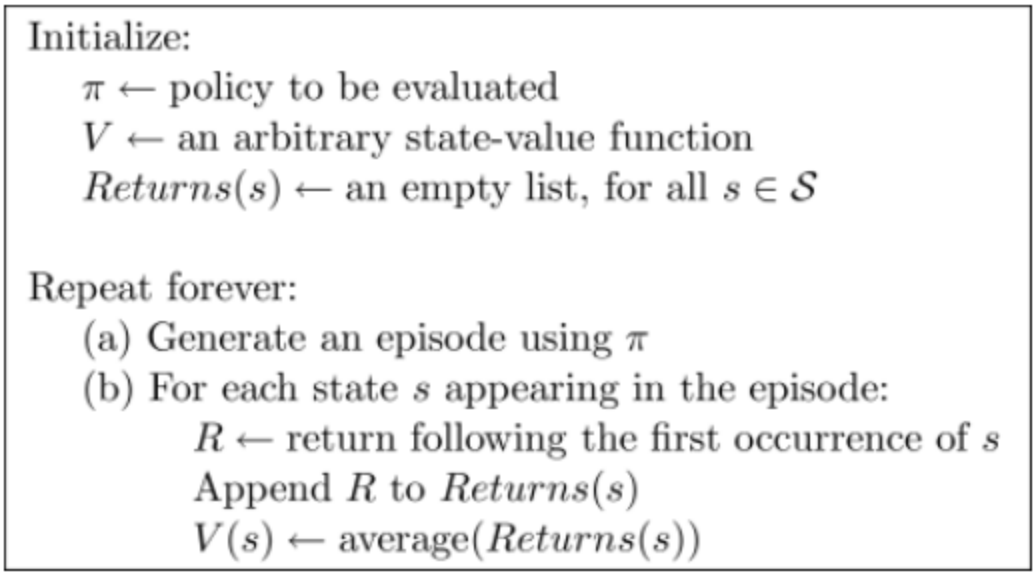
One decision that does have to be made is how to treat repeated visits to a state or state-action pair within an episode. There are essentially two options in such cases:

1. *Every-Visit MC* - Record the return for every visit to state s (or state-action pair s, a) and then average them along with returns from other episodes
2. *First-Visit MC* - Record the return for the first visit to state s (or state-action pair s, a) in an episode and average over episodes

   Both of these approaches will converge to the true value for the state s (or state-action pair s, a) as the number of visits to state s (or state-action pair s, a) approaches infinity. Deciding on which approach to use will often depend on the nature of the RL problem you are trying to solve. Below is an outline of the first-visit MC method for policy evaluation taken from Reinforcement Learning: An Introduction by Sutton and Barto (1998):

```
In [4]: Image(filename='FirstVisitMC.png')
```

Out[4]:



Typically in Monte Carlo policy evaluation we want to estimate state-action pair or action values (i.e. $Q^{\pi}(s, a)$) rather than state values (i.e. $V^{\pi}(s)$). This is because action values contain all the information we need to select the action with the highest expected return from the current state. In comparison, state values only contain information about the value of the state averaged over all possible actions from that state. To select actions using state values we therefore need to use a model of the environment to look ahead one step and choose the action that leads to the highest combination of immediate reward and disocunted next state value. Of course if we are using MC methods then we generally assume we don't have such a model and so action values are the only viable way to select good actions. This commonly comes at the cost of representing many more entries in our value function (if using a tabular representation). The main problem with is that we still need to ensure that all state-action pairs are visited so that we can reliable choose between actions. One way to ensure this is to use a stochastic policy π that ensures that every state-action pair has a non-zero probability of being chosen.

1.2 Policy Improvement

For MC methods we often (see below for exceptions) use the same policy improvement method seen in dynamic programming. That is we change our policy so that it acts greedily with respect to our current estimated value function:

$$\pi_{t+1} \leftarrow greedy(Q^{\pi_t})$$

Under the policy improvement thereom discussed in the previous notebooks, this gaurantees us that π_{t+1} will be better than π_t, unless it is the same at which point it will be the optimal policy π^*.

1.3 Monte Carlo Control

Now that we have defined our policy evaluation and policy improvement steps we can combine them into a full MC control method that will find us the optimal policy π^*. Remember that under the GPI framework we improve our value function so that it more closely resembles the true value function for our policy and then improve our policy with respect to our estimated value function. By repeating this process we are able to converge to the optimal value function and policy.

The most naive approach would be to sample many epsiodes of experience for each policy evaluation step so that our value function converges to the true value function for our policy. Only then would we switch to the policy improvement step by making our policy greedy with respect to our value function. Unfortunately, waiting for convergence in this manner can take a long time and is often impractical. However, as we saw in dynamic programming, it is not neccessary to have our policy evaluation step converge to the true value function before we switch to the policy improvement step. Indeed, in the case of 'in-place' value iteration we switch between the policy evaluation step and policy improvement step for just a single state. We can therefore use a similar approach when it comes to implementing MC control methods. A particularly common approach is to apply one step of policy evaluation and one step of policy improvement after each episode of experience. Once an episode has been completed we have the necessary return values for each

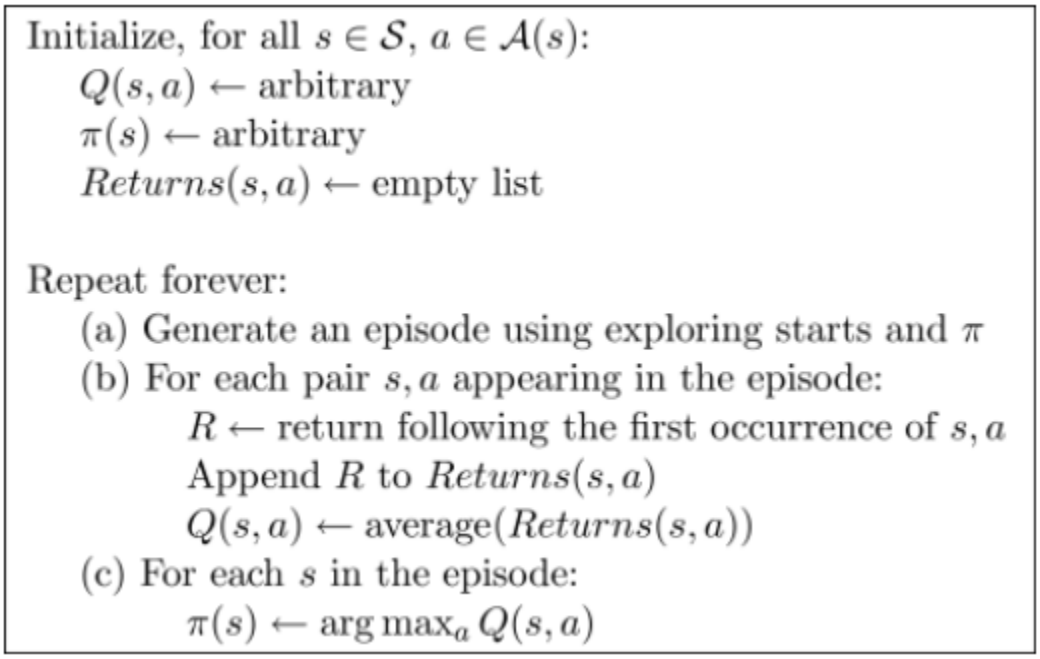
of the visited states and so we can update our value function and then improve our policy before starting the next episode.

The other main issue to consider when formulating a MC control method is how to ensure that every state-action pair is visited infinitely often. If we cannot satisfy this requirement then we cannot gaurantee that a MC control method will find the optimal policy because it may not sufficiently sample certain areas of the state-action space. One solution to this problem is to use a method known as 'exploring starts', whereby a state-action pair is randomly chosen to start an epsiode. Each state-action pair has a non-zero probability of being chosen as the starting point for an episode and so this gaurantees that all state-action pairs will be visited infinitely often given an infinite number of episodes.

With the concept of one iteration of policy evaluation and policy improvement after each episode and the concept of exploring starts we can now define our first practical MC control method known as Monte Carlo with Exploring starts (Monte Carlo ES). The pseudocode for Monte Carlo ES can be found below and is taken from Reinforcement Learning: An Introduction by Sutton and Barto (1998):

```
In [5]: Image(filename='MC_ES.png')
```

Out[5]:



It is worth noting that in Monte Carlo ES we actually average over all the returns experienced regardless of the policy that was used to generate them. Fortunately this does not mean that we will converge to a suboptimal solution because if the value function does converge then it will converge to the value function for the current policy, at which point the policy will change and

improve. Both the value function and policy can only stop changing when an optimal solution is found.

The exploring starts method is not always feasible and so there are other methods available to us to ensure that all state-action pairs are visited. As mentioned in the policy evaluation section, one approach is to ensure that we use a stochastic policy that has a non-zero probability of chosing any state-action pair ($\pi(s, a) > 0$ for all $s \in S$ and $a \in A(s)$). One such policy is an ϵ-greedy policy, whereby we select a random action from the current state with probability ϵ. Otherwise we just choose the action in our current state with the highest value (the greedy action). We therefore select the greedy action with probability $1 - \epsilon + \frac{\epsilon}{|A(s)|}$ and all other actions with probability $\frac{\epsilon}{|A(s)|}$. This approach ensures that all state-action pairs will be visited infinitely often given an infinite number of episodes.

Approaches that use stochastic policies such as ϵ-greedy can be split into 'on-policy' and 'off-policy' methods, which are characterised as follows: 1. On-policy $\rightarrow$ The policy that is being evaluated and improved is the same as the one used to generate the sampled experience 2. Off-policy $\rightarrow$ The policy that is being evaluated and improved is different from the one used to generate the sampled experience

Importantly in both cases the policy that is used to generate the sampled experience is stochastic (e.g. ϵ-greedy), with a non-zero probability of selecting any state-action pair. We therefore ensure that our sampled experience will sample all the state-action pairs infinitely often given an infitinite number of episodes.

To begin with lets consider a simple on-policy MC control method that uses an ϵ-greedy policy. As we are describing an on-policy method, both the policy used to sample the experience and the policy being evaluated and improved will use an ϵ-greedy policy. In other words, we sample experience using an ϵ-greedy policy and evaluate the value function for this policy, we then improve the policy by making it an ϵ-greedy policy with respect to our estimated value function. Pseudocode for the ϵ-greedy on-policy MC control method can be found below and is taken from Reinforcement Learning: An Introduction by Sutton and Barto (1998):

```
In [6]: Image(filename='OnPolicyMC.png')
```

```
Out[6]:
```

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
$Q(s,a) \leftarrow$ arbitrary
$Returns(s,a) \leftarrow$ empty list
$\pi \leftarrow$ an arbitrary ε-soft policy

Repeat forever:
(a) Generate an episode using π
(b) For each pair s,a appearing in the episode:
$R \leftarrow$ return following the first occurrence of s,a
Append R to $Returns(s,a)$
$Q(s,a) \leftarrow$ average$(Returns(s,a))$
(c) For each s in the episode:
$a^* \leftarrow \arg\max_a Q(s,a)$
For all $a \in \mathcal{A}(s)$:
$$\pi(s,a) \leftarrow \begin{cases} 1-\varepsilon+\varepsilon/|\mathcal{A}(s)| & \text{if } a = a^* \\ \varepsilon/|\mathcal{A}(s)| & \text{if } a \neq a^* \end{cases}$$

If we just improved our policy so that it was greedy with respect to our estimated value function then we would no longer gaurantee that all state-action pairs would be visited because the policy used to generate the sampled experience uses the same policy. Fortunately, even though we only make the new policy ϵ-greedy with respect to our estimated value function, the policy improvement theorem states that the new ϵ-greedy policy will still be better than the previous ϵ-greedy policy (unless the optimal ϵ-greedy policy has been found). That is any policy $\pi^{'}$ that is ϵ-greedy with respect to a value function Q^{π} will always be at least as good as the ϵ-greedy policy π. Intuitively, remember that in the GPI framework the policy improvement step just tries to make our policy better and so a completely greedy policy is not strictly neccessary to see an improvement in this case. Compared to Monte Carlo ES, we can only converge to the best ϵ-greedy policy but we no longer need to rely on exploring starts.

The final approach we will cover in this notebook is off-policy Monte Carlo control. As mentioned above an off-policy method is one that evaluates and improves a different policy to the one being used to generate sample experience. More concretely we are faced with the problem of estimating V^{π} or Q^{π} when we only have experience generated from the policy $\pi^{'}$. Intuitively, this is only possible when all the actions taken according to π are also at some point taken by $\pi^{'}$ (i.e. if $\pi(s,a)$ > 0 then $\pi^{'}$(s,a) must also be > 0 so that we have information about all the neccessary state-action pairs). Of course even if both policies have non-zero probability for the same state-action pairs, their actions after a given state-action pair are likely to be different and so we need a way of accounting for that. Fortunately we can do this using the probability of a sampled trajectory from a state-action pair given each of the policies. Let $R_i(s)$ be the return from state s on episode i and

let $p_i(s)$ and $p_i^{'}(s)$ be the probabilities of the episode trajectory from state s onwards given policy π and $\pi^{'}$ respectively e.g.:

$$p_i(s_t) = \prod_{k=t}^{T_i(s)-1} \pi(s_k, a_k) P^{a_k}_{s_k s_{k+1}}$$
$$p_i^{'}(s_t) = \prod_{k=t}^{T_i(s)-1} \pi^{'}(s_k, a_k) P^{a_k}_{s_k s_{k+1}}$$

Where $T_i(s)$ is the time of termination for episode i. With these quantities we can easily estimate $V^{\pi}(s)$ by weighting each of the returns from state s by the relative probability of them occuring according to $p_i(s)$ and $p_i^{'}(s)$ i.e. $\frac{p_i(s)}{p_i^{'}(s)}$:

$$V^{\pi}(s) = \frac{\sum_{i=1}^{n_s} \frac{p_i(s)}{p_i^{'}(s)} R_i(s)}{\sum_{i=1}^{n_s} \frac{p_i(s)}{p_i^{'}(s)}}$$

As seen above the calculation of $p_i(s)$ or $p_i^{'}(s)$ requires knowledge of the environment's dynamics, which is often not the case when we are using MC methods. We can get around this by noticing that we only need the ratio of $p_i(s)$ and $p_i^{'}(s)$ and this can be calculated without any knowledge of the dynamics. The only information we need are the probabilities described by the two policies:

$$\begin{aligned} \frac{p_i(s)}{p_i^{'}(s)} &= \frac{\prod_{k=t}^{T_i(s)-1} \pi(s_k, a_k) P^{a_k}_{s_k s_{k+1}}}{\prod_{k=t}^{T_i(s)-1} \pi^{'}(s_k, a_k) P^{a_k}_{s_k s_{k+1}}} \\ &= \prod_{k=t}^{T_i(s)-1} \frac{\pi(s_k, a_k)}{\pi^{'}(s_k, a_k)} \end{aligned}$$

Putting this all together we can construct a full off-policy MC method. Commonly we refer to the policy that is used to generate sample experience as the behaviour policy and the policy that is used for evaluation and improvement as the estimation policy. One of the main advantages of having these two different policies is that the estimation policy can be deterministic or greedy while the behaviour policy can remain stochastic in order to sample all possible actions. Remember though that for an off-policy MC method to work, we require that the behaviour policy must have some non-zero probability of selecting any action in the estimation policy. Generally any ϵ -greedy behaviour policy will satisfy this requirement. Below is pseudocode for an off-policy MC control method taken from Reinforcement Learning: An Introduction by Sutton and Barto (1998):

```
In [7]: Image(filename='OffPolicyMC.png')
```

```
Out[7]:
```

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:

$Q(s,a) \leftarrow$ arbitrary

$N(s,a) \leftarrow 0$; Numerator and

$D(s,a) \leftarrow 0$; Denominator of $Q(s,a)$

$\pi \leftarrow$ an arbitrary deterministic policy

Repeat forever:

(a) Select a policy π' and use it to generate an episode:

$s_0, a_0, r_1, s_1, a_1, r_2, \ldots, s_{T-1}, a_{T-1}, r_T, s_T$

(b) $\tau \leftarrow$ latest time at which $a_\tau \neq \pi(s_\tau)$

(c) For each pair s, a appearing in the episode after τ:

$t \leftarrow$ the time of first occurrence (after τ) of s, a

$w \leftarrow \prod_{k=t+1}^{T-1} \frac{1}{\pi'(s_k, a_k)}$

$N(s,a) \leftarrow N(s,a) + wR_t$

$D(s,a) \leftarrow D(s,a) + w$

$Q(s,a) \leftarrow \frac{N(s,a)}{D(s,a)}$

(d) For each $s \in \mathcal{S}$:

$\pi(s) \leftarrow \arg\max_a Q(s,a)$

It is worth noting that one draw back of this method is that learning can be extremely slow. If the behaviour policy chooses actions that are unlikely under the estimation policy often then learning will be slowed due to the small value of the ratio $\frac{p_i(s)}{p_i'(s)}$. This problem will be much worse for states that occur early on in episodes because it is more likely that the actions selected under the two policies will be dissimiliar.

2 Example - Racetrack (Sutton and Barto, 1998)

Like in the previous notebook we shall use an example from the book 'Reinforcement Learning: An Introduction' by Sutton and Barto (1998), to cement some of the concepts described thus far. In this example we shall solve the 'racetrack' problem which is described as follows:

- We want to drive a race car around a rightward turn as fast as possible i.e. in as few a timesteps as possible, without going off the track
- We shall divde the track up into discrete positions on a 2D grid
- We shall also describe the cars velocity discretely as the number of positions moved horizontally and vertically per time step

- Our actions correspond to independently changing the horizontal and vertical velocity of the car by -1, 0 or 1 per time step (9 actions in total)
- The horizontal and vertical velocity are restircted to be between 0 and 5 and can't both be 0 (we are therefore only able to make rightward turns)
- To start an episode, the car starts on a random position on the start line and the episode finishes when the car reaches the finish line
- On every time step we receive a reward of -1 until we reach the finish line
- The car cannot leave the track and if it attempts to do so its position is just incremented by at least one step horizontally or vertically and a reward of -5 is recieved. This gaurantees that all epsiodes will eventually terminate.

As before let's take this problem description and formulate it as an MDP that we want to solve:

1. S - The state signal is the position of the car in x and y co-ordinates
2. A - The actions are the changes to the horiztonal and vertical velocities i.e. $(-1,-1),(-1,0),(-1,1),(0,-1),(0,0),(0,1),(1,-1),(1,0),(1,1)$
3. $P^a_{ss'}$ - The transition function is deterministic and is defined by the velocity of the car and the rules for attempting to drive off the track. In this case we are using a MC method and so we assume that we don't know this transition function explicitly.
4. $R^a_{ss'}$ - The reward function is -1 for transitions within the track and -5 for transitions off the track. Again we assume that we do not know this function because we are using a MC method.
5. γ - We shall assume there is no discounting in this task i.e. $\gamma = 1$

The following code uses an ϵ-greedy on-policy MC control method to solve the ractrack problem. As before we use a class to define the neccessary functions and then call the ϵ-greedy on-policy MC control method to solve the MDP.

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        import matplotlib.patches as mpatches
        import matplotlib as mpl

        mpl.rcParams['figure.dpi'] = 100


        class Episode(object):

            def __init__(self):
                self.states = []
                self.actions = []
                self.rewards = []
```

```
class Racetrack(object):

    def __init__(self):

        self.width = 25
        self.height = 65

        self.start_start = 7
        self.start_length = 10

        self.actions = {0: (-1, -1),
                        1: (-1, 0),
                        2: (-1, 1),
                        3: (0, -1),
                        4: (0, 0),
                        5: (0, 1),
                        6: (1, -1),
                        7: (1, 0),
                        8: (1, 1)}

        self.num_actions = self.actions.__len__()

        self.state_action_values = np.zeros((self.height, self.width,
                                             self.num_actions))
        self.policy = np.ones((self.height, self.width)) * 7
        self.returns = {}

        self.step_reward = -1
        self.off_track_reward = -5
        self.max_velocity = 5
        self.epsilon = .1
        self.num_episodes = 100000
        self.results_interval = 6250

        self.ConstructRacetrack()
        self.PlotRacetrack(self.racetrack)

    def ConstructRacetrack(self):

        self.racetrack = np.ones((self.height, self.width))

        # Input 2s for the start and 3s for the finish line
        self.racetrack[self.height - 1, self.start_start:(
            self.start_start + self.start_length)] = 2
        self.racetrack[:self.start_length, self.width - 1] = 3

        # Input 0s for the track boundaries
        for row in range(self.height):
```

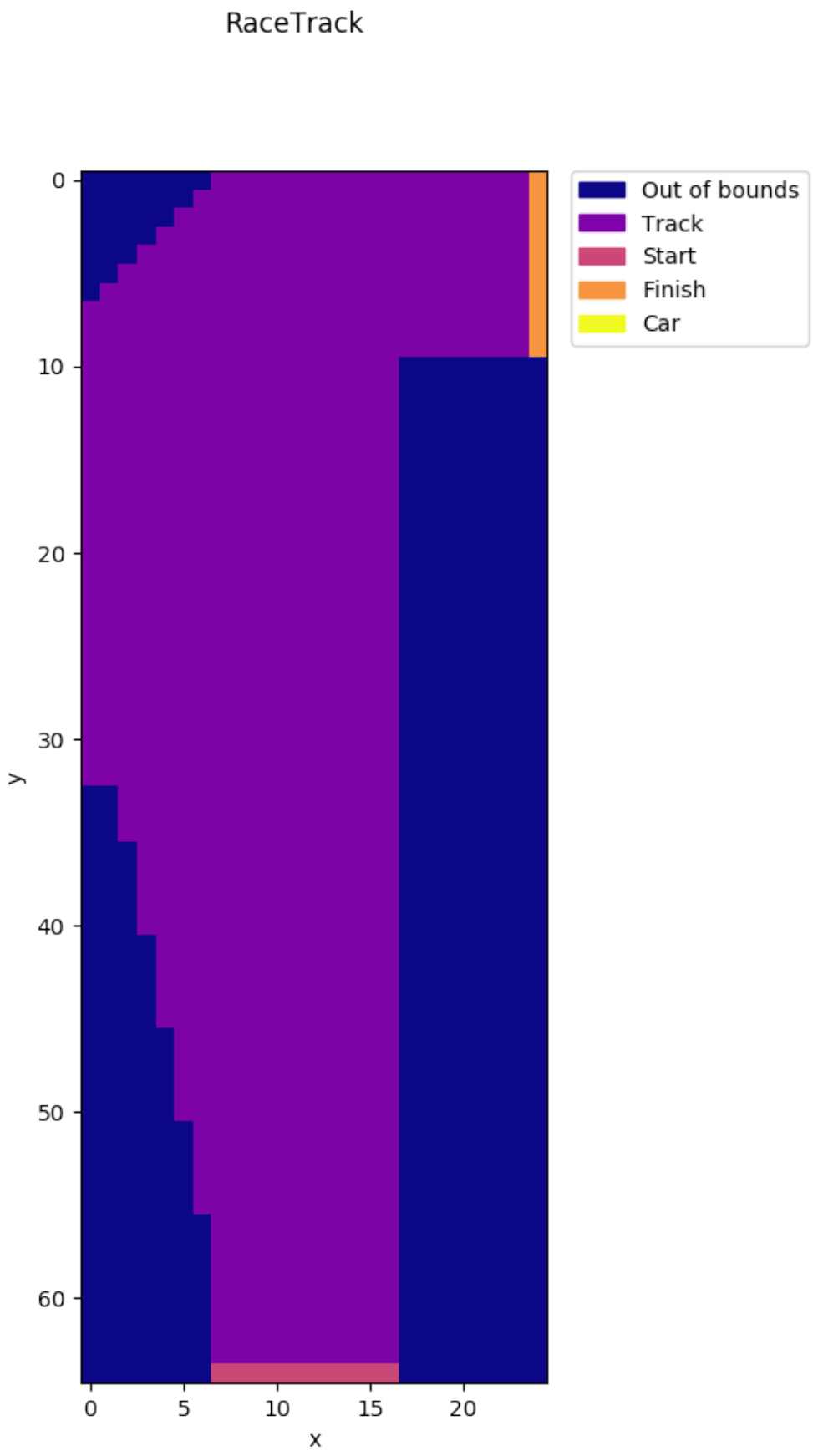
```
            for col in range(self.width):
                if ((row + col < 7) or
                        (row > 9 and col > self.start_start +
                         self.start_length - 1) or
                        (row > self.height / 2 and col <
                         self.start_start and col + 0.2 * (
                             self.height - row) < 8)):
                    self.racetrack[row][col] = 0

        return

    def PlotRacetrack(self, racetrack):

        plt.figure(figsize=(10, 10))
        im = plt.imshow(racetrack, cmap='plasma', vmin=0, vmax=4)
        plt.suptitle('RaceTrack')
        plt.xlabel('x')
        plt.ylabel('y')
        labels = ['Out of bounds', 'Track', 'Start', 'Finish', 'Car']
        values = [0, 1, 2, 3, 4]
        colors = [im.cmap(im.norm(value)) for value in values]
        patches = [mpatches.Patch(color=colors[i], label=labels[i])
                   for i in range(len(values))]
        plt.legend(handles=patches, bbox_to_anchor=(1.05, 1), loc=2,
                   borderaxespad=0.)
        plt.show()

        return

    def OnPolicyMonteCarlo(self):

        example_episodes = {}

        for episode_num in range(self.num_episodes):

            episode = self.GenerateEpisode()

            if (episode_num % self.results_interval == 0):
                print('Episode: ' + str(episode_num) + '/' + str(
                    self.num_episodes))
                example_episodes[episode_num] = episode

            self.UpdateStateActionValues(episode)
            self.UpdatePolicy(episode)

        self.PlotExampleEpisodes(example_episodes)
        self.PlotPolicy()
```

```
        return

    def GenerateEpisode(self):

        episode = Episode()

        row = self.height - 1
        col = (np.random.randint(self.start_start, self.start_start +
                                 self.start_length))
        vel = np.array([1, 0])
        bEpisode_over = False

        while (not bEpisode_over):

            episode.states.append((row, col))

            if (np.random.rand() < self.epsilon):
                action_num = np.random.randint(self.num_actions)
                action = self.actions[action_num]
            else:
                action_num = self.policy[row][col]
                action = self.actions[action_num]

            episode.actions.append(int(action_num))

            new_vel_y = np.clip(vel[0] + action[0], 0, self.max_velocity)
            new_vel_x = np.clip(vel[1] + action[1], 0, self.max_velocity)

            if (new_vel_y == 0 and new_vel_x == 0):
                new_vel_y = 1
                new_vel_x = 0

            vel[0] = new_vel_y
            vel[1] = new_vel_x

            new_row = row - vel[0]
            new_col = col + vel[1]

            reward = self.step_reward

            # Check for finish line
            if (new_row >= 0 and new_row < self.start_length and
                new_col >= self.width):
                reward = 0
                bEpisode_over = True

            else:
                # Check for out of bounds
```

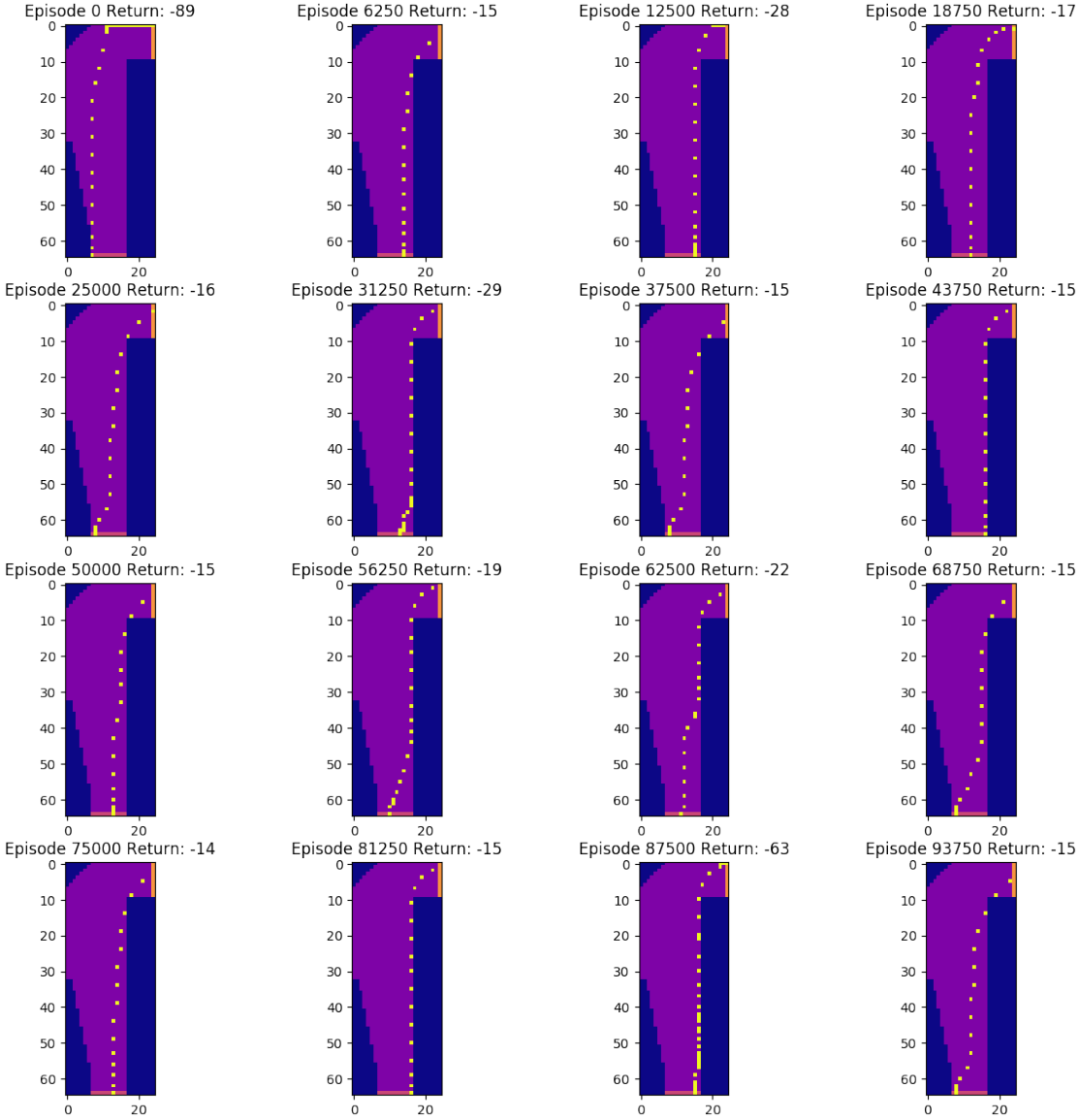
```
                if(new_row < 0 or new_col < 0 or
                   self.racetrack[new_row][new_col] == 0):

                    reward = self.off_track_reward
                    bInc = False

                    if(row - 1 >= 0):
                        if(self.racetrack[row - 1][col] != 0):
                            new_row = row - 1
                            new_col = col
                            bInc = True

                    if(not bInc):
                            new_row = row
                            new_col = col + 1

                    # Check for finish line
                    if (new_row >= 0 and new_row <
                        self.start_length and new_col >= self.width):
                        reward = 0
                        bEpisode_over = True

            row = new_row
            col = new_col

            episode.rewards.append(reward)

        return episode

    def PlotEpisode(self, episode):

        trajectory = np.copy(self.racetrack)
        for inds in episode.states:
            trajectory[inds] = 4

        self.PlotRacetrack(trajectory)

        return

    def UpdateStateActionValues(self, episode):

        for t in range(episode.states.__len__()):

            row = episode.states[t][0]
            col = episode.states[t][1]
            action = episode.actions[t]

            ret = self.GetReturn(t, episode)
```

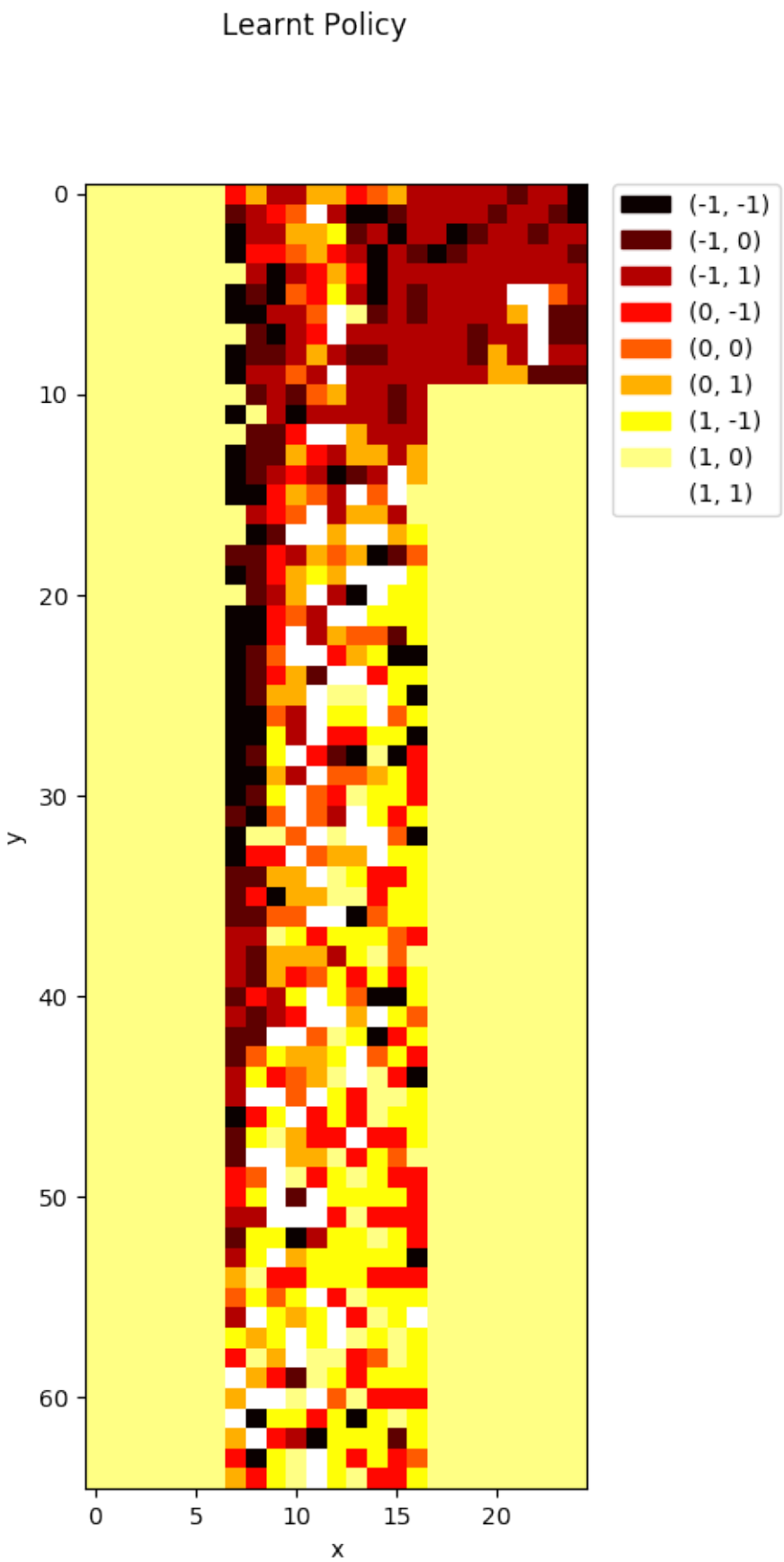
```
            self.CheckEntry(row, col, action)
            self.returns[row][col][action].append(ret)
            self.state_action_values[row, col, action] = np.mean(
                self.returns[row][col][action])

        return

    def GetReturn(self, t, episode):

        ret = 0

        for tp in range(t, episode.states.__len__()):
            ret += episode.rewards[tp]

        return ret

    def CheckEntry(self, row, col, action):

        try:
            self.returns[row]
        except KeyError:
            self.returns[row] = {}

        try:
            self.returns[row][col]
        except KeyError:
            self.returns[row][col] = {}

        try:
            self.returns[row][col][action]
        except:
            self.returns[row][col][action] = []

        return


    def UpdatePolicy(self, episode):

        for t in range(episode.states.__len__()):
            action_values = self.state_action_values[
                episode.states[t][0], episode.states[t][1], :]
            self.policy[episode.states[t][0],
                        episode.states[t][1]] = np.argmax(action_values)

        return

    def PlotExampleEpisodes(self, example_episodes):
```

```
            fig = plt.figure(figsize=(15, 15))
            plt.suptitle('Example Episodes Over Training')

            for i in range(example_episodes.__len__()):
                episode = example_episodes[i*self.results_interval]
                ax = fig.add_subplot(4, 4, 1 + i)
                ax.set_title('Episode ' + str(i*self.results_interval) +
                             ' Return: ' + str(np.sum(episode.rewards)))

                trajectory = np.copy(self.racetrack)
                for inds in episode.states:
                    trajectory[inds] = 4

                im = ax.imshow(trajectory, cmap='plasma', vmin=0, vmax=4)

            plt.show()

            return

        def PlotPolicy(self):

            plt.figure(figsize=(10, 10))
            im = plt.imshow(self.policy, cmap='hot', vmin=0, vmax=8)
            plt.suptitle('Learnt Policy')
            plt.xlabel('x')
            plt.ylabel('y')
            labels = []
            for i in range(self.actions.__len__()):
                labels.append(str(self.actions[i]))
            values = [0, 1, 2, 3, 4, 5, 6, 7, 8]
            colors = [im.cmap(im.norm(value)) for value in values]
            patches = [mpatches.Patch(color=colors[i], label=labels[i])
                       for i in range(len(values))]
            plt.legend(handles=patches, bbox_to_anchor=(1.05, 1), loc=2,
                       borderaxespad=0.)
            plt.show()

            return


racetrack_game = Racetrack()
racetrack_game.OnPolicyMonteCarlo()
```


RaceTrack
Out of bounds
Track
Start
Finish
Car
0
10
20
30
40
50
60
y
0
5
10
15
20
x

```
Episode: 0/100000
Episode: 6250/100000
Episode: 12500/100000
Episode: 18750/100000
Episode: 25000/100000
Episode: 31250/100000
Episode: 37500/100000
Episode: 43750/100000
Episode: 50000/100000
Episode: 56250/100000
Episode: 62500/100000
Episode: 68750/100000
Episode: 75000/100000
Episode: 81250/100000
Episode: 87500/100000
Episode: 93750/100000
```


Example Episodes Over Training
Episode 0 Return: -89
Episode 6250 Return: -15
Episode 12500 Return: -28
Episode 18750 Return: -17
Episode 25000 Return: -16
Episode 31250 Return: -29
Episode 37500 Return: -15
Episode 43750 Return: -15
Episode 50000 Return: -15
Episode 56250 Return: -19
Episode 62500 Return: -22
Episode 68750 Return: -15
Episode 75000 Return: -14
Episode 81250 Return: -15
Episode 87500 Return: -63
Episode 93750 Return: -15


Learnt Policy
0
10
20
30
40
50
60
y
0
5
10
15
20
x
(-1, -1)
(-1, 0)
(-1, 1)
(0, -1)
(0, 0)
(0, 1)
(1, -1)
(1, 0)
(1, 1)

As you can see from the example episodes over training, our ϵ-greedy on-policy MC control method has learnt a reasonably effective policy for the racetrack problem. In general the car takes the corner as tightly as possible and reaches the finish line in much fewer steps than at the start of training. Remember that the main advantages of MC methods over dynamic programming methods are: - MC methods can learn through interaction with the environment and so do not require a model of the environment's dynamics - MC methods can be used for both real or simulated experience (we shall see more about simulated experience in later notebooks) - MC methods can be focused on particular states that are of interest to our problem without estimating the value of all the states in our state space

Note how the learnt learnt policy is fairly noisey, this is due to the high variance that is characteristic of MC methods. In MC methods we sample the full return all the way up to the end of an episode and do not rely on any bootstrapping of past estimates. Our sampled returns can therefore vary wildly in their value as full episodes can be very different, particularly during learning. This creates high variance in our estimate of the return from a given state-action pair and we may need many samples to generate a reliable estimate and ultimately a smooth, stable policy.

With this in mind it is reasonable to ask whether an approach exists that samples experience (and therefore doesn't require a model of the world) in a manner similiar to MC methods but that also alleviates the high variance of MC methods by bootstrapping estimates similiar to dynamic programming. Indeed, such an approach exists and is called Temporal Difference (TD) learning. TD learning will be the subject of the next notebook and represents the final basic solution method for solving the RL problem. Armed with an understanding of dynamic programming, MC methods and TD methods, you will be in a good position to tackle some of the more advanced solution methods that use these concepts as fundamental building blocks.