TemporalDifferenceLearning

November 28, 2018

1 Temporal Difference Learning

Temporal difference (TD) learning represents the final basic solution method for solving the RL problem. After completeing this notebook you should hopefully be able to see the relationships between Dynamic Programming (DP), Monte Carlo (MC) and TD methods. In fact, more advanced methods for solving the RL problem combine these approaches in an attempt to get the benefits of all three. At its core, TD learning is similar to DP methods in that it bootstraps information (updates estimated values using other estimates) and it is also similar to MC methods in that it can learn from sampled experience without a model of the environment.

In short, classic TD learning samples one step of the underlying MDP by selecting an action in a given state and observing the outcome. The resulting state and reward is then used in combination with existing value estimates to update the value estimate for the state or state-action pair. This is possible due to the recursive definition of the Bellman equations, much as it was with dynamic programming.

As with all the solution methods we have covered so far TD learning still fits under the umbrella of Generalized Policy Iteration (GPI) and so we shall now use the GPI framework to outline TD learning in more detail. As a side note DP, MC and TD predominantly differ in the way they tackle the policy evaluation step of GPI.

1.1 Policy Evaluation

Recall from the notebook on MC methods that we evaluated the policy using a samples of the full return R_t:

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

In the practical example we just recorded all the returns we experienced for a given state-action pair and calcualted their average. However this can be computationally expensive as the number of episodes increases. Another option is to incrementally calcualte the average return, without explicitly storing all of the return values themselves. Let R_i denote the return from state s on

episode i and k denote the number of episodes previously experienced. Using this notation we can incrementally calculate the new average return after episode $k+1$ as follows:

$$\begin{aligned} V(s)_{k+1} &= \frac{1}{k+1}\sum_{i=1}^{k+1} R_i \\ &= \frac{1}{k+1}(R_{k+1} + \sum_{i=1}^{k} R_i) \\ &= \frac{1}{k+1}(R_{k+1} + kV(s)_k + V(s)_k - V(s)_k) \\ &= \frac{1}{k+1}(R_{k+1} + (k+1)V(s)_k - V(s)_k) \\ &= V(s)_k + \frac{1}{k+1}[R_{k+1} - V(s)_k] \end{aligned}$$

By re-arranging the formula in this way we now only need to keep a record of our current estimate of $V(s)$ and the number of times we have sampled a return from s (i.e. the value of k). We can actually write the update rule in a much more general fashion, which you will find is pervasive throughout the field of reinforcement learning:

$$NewEstimate \leftarrow OldEstimate + StepSize[Target - OldEstimate]$$

Where $[Target - OldEstimate]$ is often called the $error$ of our estimate. In effect we are just moving our current estimate in the direction of our target value, which in the case of MC methods is the newest sampled return R_{k+1}.

The above approach works well for calculating the average of a stationary distribution but with RL we often deal with non-stationary distributions because our estimates are constantly changing necause of changes in the policy. One way to overcome this is to calculate a moving average of our target value e.g. the full return R_t in the case of MC methods. A common approach for calculating a moving average of the return for MC methods is as follows:

$$V(s_t) \leftarrow V(s_t) + \alpha[R_t - V(s_t)]$$

Where α is a constant between 0 and 1. This equation increments our estimate of $V(s_t)$ by a small amount towards the target R_t after each episode. More concretely this operation calculates an exponentially weighted average of R_t, with higher weightings for more recent values, which allows it to deal with non-stationary values better. We can prove this as follows:

$$
\begin{aligned}
V_k &= V_{k-1} + \alpha[R_k - V_{k-1}] \\
&= \alpha R_k + (1-\alpha)V_{k-1} \\
&= \alpha R_k + (1-\alpha)\alpha R_{k-1} + (1-\alpha)^2 V_{k-2} \\
&= \alpha R_k + (1-\alpha)\alpha R_{k-1} + (1-\alpha)^2 R_{k-2} + ... + (1-\alpha)^{k-1} R_1 + (1-\alpha)^k V_0 \\
&= (1-\alpha)^k V_0 + \sum_{i=1}^{k} \alpha(1-\alpha)^{k-i} R_i
\end{aligned}
$$

As you can see as the number of returns k increases, the weight for older returns decreases exponentially according to the exponent of $(1-\alpha)$. It is important to go through this method of calculating an incremental, exponentially weighted average because TD methods rely heavily on them to update their estimates of the value function. In TD learning, not only are the returns changing as the policy changes but because it relies on bootstrapping the other estimates are also changing. It is therefore especially important that we use an update rule for TD learning that can handle non-stationary distributions.

The above MC calculations assume that the full sample return R_t is our target value and we want our value estimate to be an average of this target. In comparison, TD methods use $r_{t+1} + \gamma V_t(s_{t+1})$ as the target i.e. they just sample one step of the MDP to obtain r_{t+1} and s_{t+1}, and then use another estimate $V_t(s_{t+1})$ to perform the update. The TD update rules for policy evaluation therefore become:

$$
\begin{aligned}
V(s_t) &\leftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)] \\
Q(s_t, a_t) &\leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]
\end{aligned}
$$

We can see that this is still calculating the expected return by re-arranging the following equation:

$$
\begin{aligned}
V^\pi(s) &= \mathbb{E}_\pi[R_t \mid s_t = s] \\
&= \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s] \\
&= \mathbb{E}_\pi[r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \mid s_t = s] \\
&= \mathbb{E}_\pi[r_{t+1} + \gamma V^\pi(s_{t+1}) \mid s_t = s]
\end{aligned}
$$

So we are sampling like in the MC case, but we are only sampling one step as opposed to all the steps up to some terminal state. We can perform the update rule after just sampling one step of the MDP because we use bootstrapping techniques like in DP. Recall from the first notebook that the bellman equations are defined as:

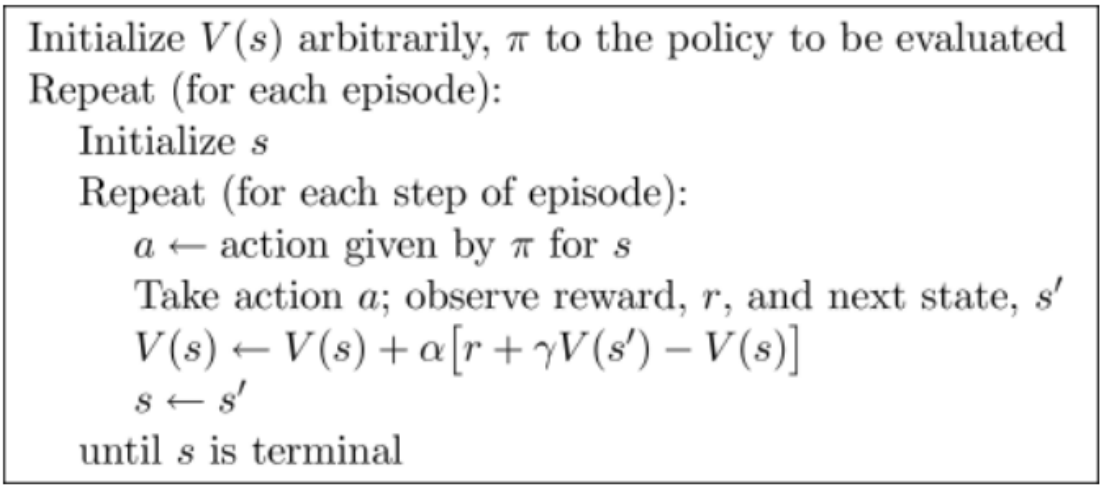
$$V^{\pi}(s) = \sum_{a} \pi(s,a) \sum_{s'} P^{a}_{ss'}[R^{a}_{ss'} + \gamma V^{\pi}(s')]$$
$$Q^{\pi}(s,a) = \sum_{s'} P^{a}_{ss'}[R^{a}_{ss'} + \gamma \sum_{a'} \pi(s',a') Q^{\pi}(s',a')]$$

The bellman equations therefore form the basis of our update rule in TD learning, just as they did in dynamic programming. The main difference however, is that we just sample one outcome and calculate an estimated average / expectation because we don't have access to the environments dyanmics $P^{a}_{ss'}$ and $R^{a}_{ss'}$. This is akin to performing a sample backup rather than a full backup as in DP.

Hopefully you can see how TD learning combines the sampling of MC methods with the bootstrapping of DP methods when it comes to policy evaluation. Below is some example pseudo code from Reinforcement Learning: An Introduction by Sutton and Barto (1998) outlining the basic steps for performing TD policy evaluation.

In [4]:
```
from IPython.display import Image
Image(filename='TDPolicyEvaluation.png')
```

Out[4]:



In general there are several advantages to using TD policy evaluation over MC policy evaluation. Firstly, the value update can be applied after every step of the MDP and so it can be applied in an online fashion. You do not need to wait until the end of an episode to apply the update rule, as is the case in MC methods. TD methods start learning straight away e.g. after the very first action taken in the MDP. This is important if we want to start improving the policy quickly rather than waiting for some far off terminal state before we apply what we have learnt. Importantly this also means that TD methods can be used for continuing tasks where there are no terminal states at all or when the episodes are extremely long and MC methods would have to wait a long

time to obtain a sample of the full return. Another key advantage of TD methods is that due to their bootstrapping properties they reduce the high variance associated with MC methods. The flip side of this is that they can suffer from high bias because they rely so heavily on other estimates, which may themselves be misleading. Another disadvantage of bootstrapping using the aforementioned TD method is that only the value of the previous state is backed up whereas in MC all the states visited in a single episode are backed up at once. Simple one-step online TD methods can therefore be fairly inefficient in propagating information back through the visited states. In a future notebook we shall see how we can use methods that actually combine TD and MC approaches to try and get the best of both worlds i.e. to produce low variance, low bias and propogate information back over several states at once. Such methods rely on the fact that there is actually a spectrum of backups between one-step backups (TD) and full-backups (MC) and so we can perform a single backup that takes a weighted sum of all these different lengths of backup.

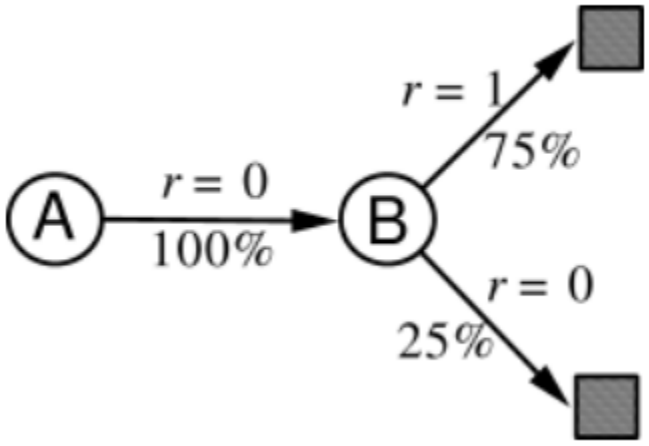
As a final note it is worth mentioning that constant α TD and MC policy evaluation don't actually converge to the same value function for all MDPs. To see this we can take another classic example from Reinforcement Learning: An Introduction by Sutton and Barto (1998). Imagine you simulate experience in an unknown MDP and you get the following results (each result is a separate episode), where A and B are different states:

States	Rewards
A, B	0, 0
B	1
B	1
B	1
B	1
B	1
B	1
B	0

Now in policy evaluation we want to work out what the value of states A and B are i.e. $V(A)$ and $V(B)$. The value of $V(B)$ appears relatively straight forward because we obtained reward from state B six times out of eight and so $V(B) = \frac{3}{4}$. Both MC and TD policy evaluation would agree on this value for $V(B)$, however they would disagree on the value for $V(A)$. For example, MC policy evaluation would state that the average return from state A is 0 ($\frac{0}{1} = 0$) and so $V(A) = 0$. In comparison TD policy evaluation would back up the value of state B, which in the undiscounted case would converge to $\frac{3}{4}$ and so $V(A) = \frac{3}{4}$. Which of these estimates do you think is more accurate? In practice the MC estimate gives us a perfect fit of the sampled data. However, the TD estimate would still be considered better because it makes use of the fact that the process is an MDP with states that satisfy the Markov property. As long as the problem is truly an MDP the TD estimate should give us less error on future data despite not being as good on the current data. This is because TD policy evaluation utilises the Markov property and the Bellman equations in its update rule and so it is better at capturing the underlying MDP. In fact TD policy evaluation converges to the maximum likelihood solution for an MDP i.e. it finds the values that maximise the probability of the observed data given the process is an MDP. In this case the maximum likelihood solution would be as follows:

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

Out[4]:



1.2 Policy Improvement

As we saw with MC methods the policy improvement step will generally depend on whether we are performing 'on-policy' or 'off-policy' control (see next section). In the 'on-policy' case it is common to improve the policy in an $\epsilon - greedy$ fashion whereas in the 'off-policy' case we can revert to the classic full $greedy$ policy improvement.

1.3 Temporal Difference Control

As was the case with MC control methods, we will generally focus on control methods that use state-action values ($Q^{\pi}(s, a)$) because they don't require a model of the environment's dynamics in order to select the best action given the current value estimates.

1.3.1 SARSA (On-Policy)

The first TD control method we will explore is an 'on-policy' method called SARSA. This method gets its name from the fact that it relies upon the tuple $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$. SARSA samples one step of the environment and the resulting tuple is used to perform the TD update rule mentioned previously:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

This corresponds to the policy evaluation step. The tuples are generated using a policy that is $\epsilon - greedy$ with respect to our state-action value estimates ($Q^{\pi}(s, a)$). This ensures that all state-action pairs will be picked infinitely often as the number of samples tends towards infinity. Since SARSA is an 'on-policy' method the policy improvement step also improves the policy using an

$\epsilon - greedy$ approach. Hopefully you can see how because we are using an $\epsilon - greedy$ policy for both picking actions and improving our policy, we end up with an on-policy method where both policies are implicitly the same.

Pseudo code for the SARSA algorithm from Reinforcement Learning: An Introduction by Sutton and Barto (1998) can be found below as well as the backup diagram.

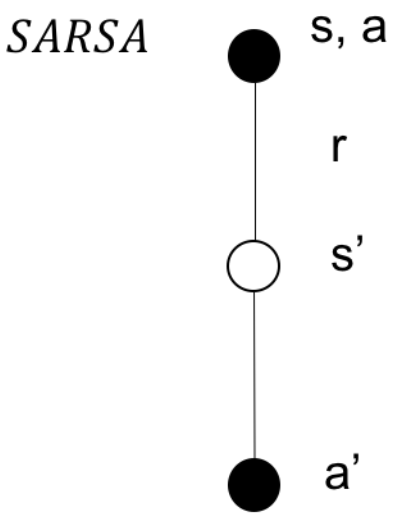
```
In [2]: Image(filename='SARSA.png')
```

Out[2]:

Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode):
  Initialize s
  Choose a from s using policy derived from Q (e.g., ε-greedy)
  Repeat (for each step of episode):
    Take action a, observe r, s'
    Choose a' from s' using policy derived from Q (e.g., ε-greedy)
    $Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma Q(s',a') - Q(s,a)\right]$
    $s \leftarrow s'$; $a \leftarrow a'$;
  until s is terminal

```
In [3]: Image(filename='SARSABackup.png')
```

Out[3]:



1.3.2 Q-Learning (Off-Policy)

The other TD control method we shall cover in this notebook is an 'off-policy' method known as Q-Learning. Q-Learning can be defined in just a single update rule that combines both one step of policy evaluation and one step of policy improvement:

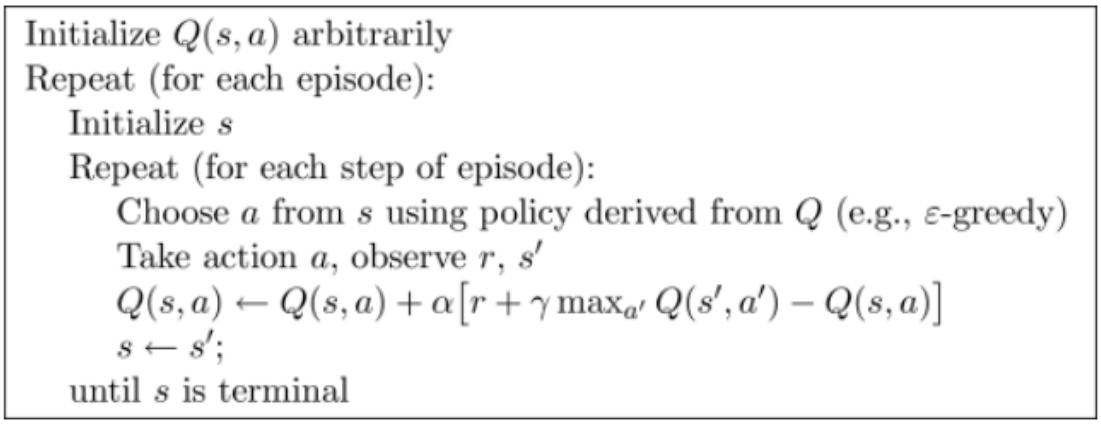
$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$$

This combination of policy evaluation and policy improvement means that Q-learning directly approximates the optimal value function Q^* because it takes the max over actions in the successor state s_{t+1}. Hopefully you can see the similarities to value iteration in DP where we used the Bellman optimality equations to define our update rule. The Q-Learning update rule is applied to experience generated from an $\epsilon - greedy$ policy to again ensure that all state-action pairs are visited. As a result Q-Learning is defined as an 'off-policy' method because the policy we are evaluating and improving is a $greedy$ policy but the one we are following is an $\epsilon - greedy$ policy.

Pseudo code from Reinforcement Learning: An Introduction by Sutton and Barto (1998) and the backup diagram for Q-Learning can be found below:

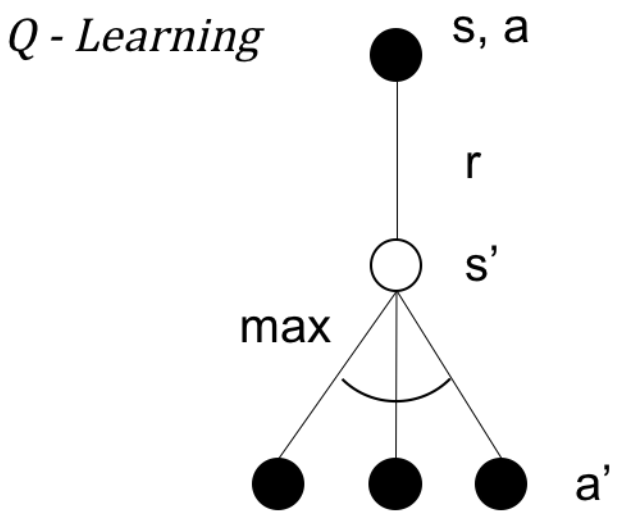
In [3]:
```
Image(filename='QLearning.png')
```

Out[3]:



In [2]:
```
Image(filename='QLearningBackup.png')
```

Out[2]:



2 Example - Racetrack (Sutton and Barto, 1998)

To demonstrate a TD control method in action let's use Q-Learning to solve the racetrack MDP that we saw in the previous MC notebook. Please refer back to the MC notebook to remind yourself of how we formulated this problem as an MDP. Note how in a given amount of time we can run more iterations of the Q-Learning method than the ϵ-greedy on-policy MC control method from the previous notebook because Q-Learning doesn't need to store all the returns in a large dictionary. We could of course turn the previous MC method into an incremental one by using a fixed step size in order to reduce the memory demand. Feel free to play with this code and see which method produces the best results, you could also implement SARSA and see how that compares.

In [13]:
```
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.alpha = .1
        self.gamma = .99

        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.step_reward = -1
        self.off_track_reward = -5
        self.max_velocity = 5
        self.epsilon = .1
        self.num_episodes = 1000000
        self.results_interval = 62500

        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
```

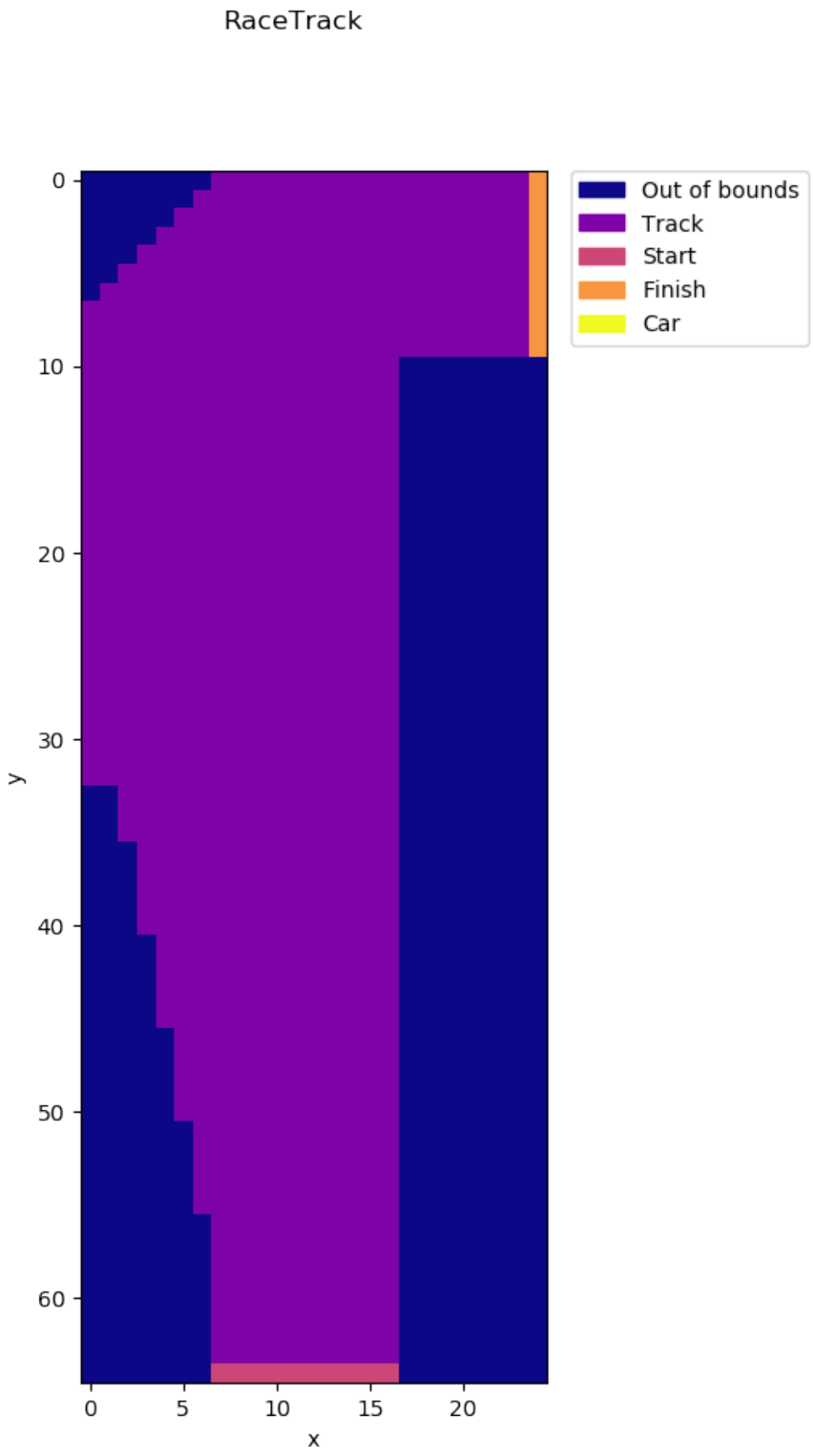
```
        for row in range(self.height):
            for col in range(self.width):
                if ((row + col < 7) or
                        (row > 9 and col > self.start_start +
                         self.start_length - 1) or
                        (row > self.height / 2 and col <
                         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 QLearning(self):

        example_episodes = {}

        for episode_num in range(self.num_episodes):

            episode = self.RunEpisode()

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

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

```
        return

    def RunEpisode(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

        state = (row, col)

        while (not bEpisode_over):

            episode.states.append(state)

            if (np.random.rand() < self.epsilon):
                action_num = np.random.randint(self.num_actions)
                action = self.actions[action_num]
            else:
                action_num = np.argmax(
                    self.state_action_values[state[0], state[1], :])
                action = self.actions[action_num]

            episode.actions.append(int(action_num))

            vel, new_state, reward, bEpisode_over = self.SimulateOneStep(
                vel, state, action, bEpisode_over)
            self.ApplyQLearningUpdate(
                bEpisode_over, state, action_num, new_state, reward)

            episode.rewards.append(reward)

            state = new_state

        return episode

    def SimulateOneStep(self, vel, state, action, bEpisode_over):

        row = state[0]
        col = state[1]

        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):
```

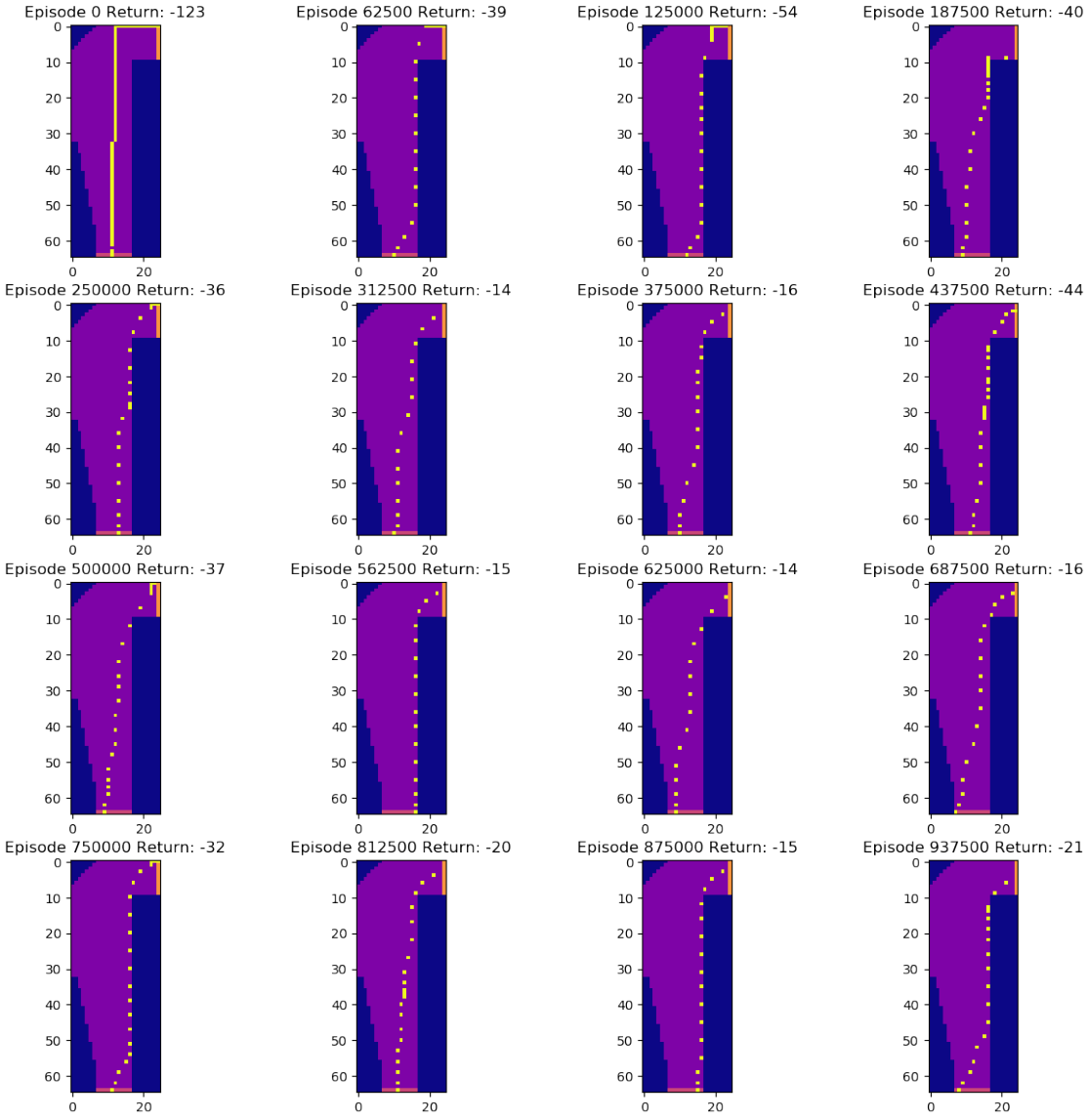
```
            new_vel_y = 1
            new_vel_x = 0

        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

        return vel, (new_row, new_col), reward, bEpisode_over


    def ApplyQLearningUpdate(self, bEpisode_over, state, action,
                             new_state, reward):

        value = self.state_action_values[state[0], state[1], action]
```

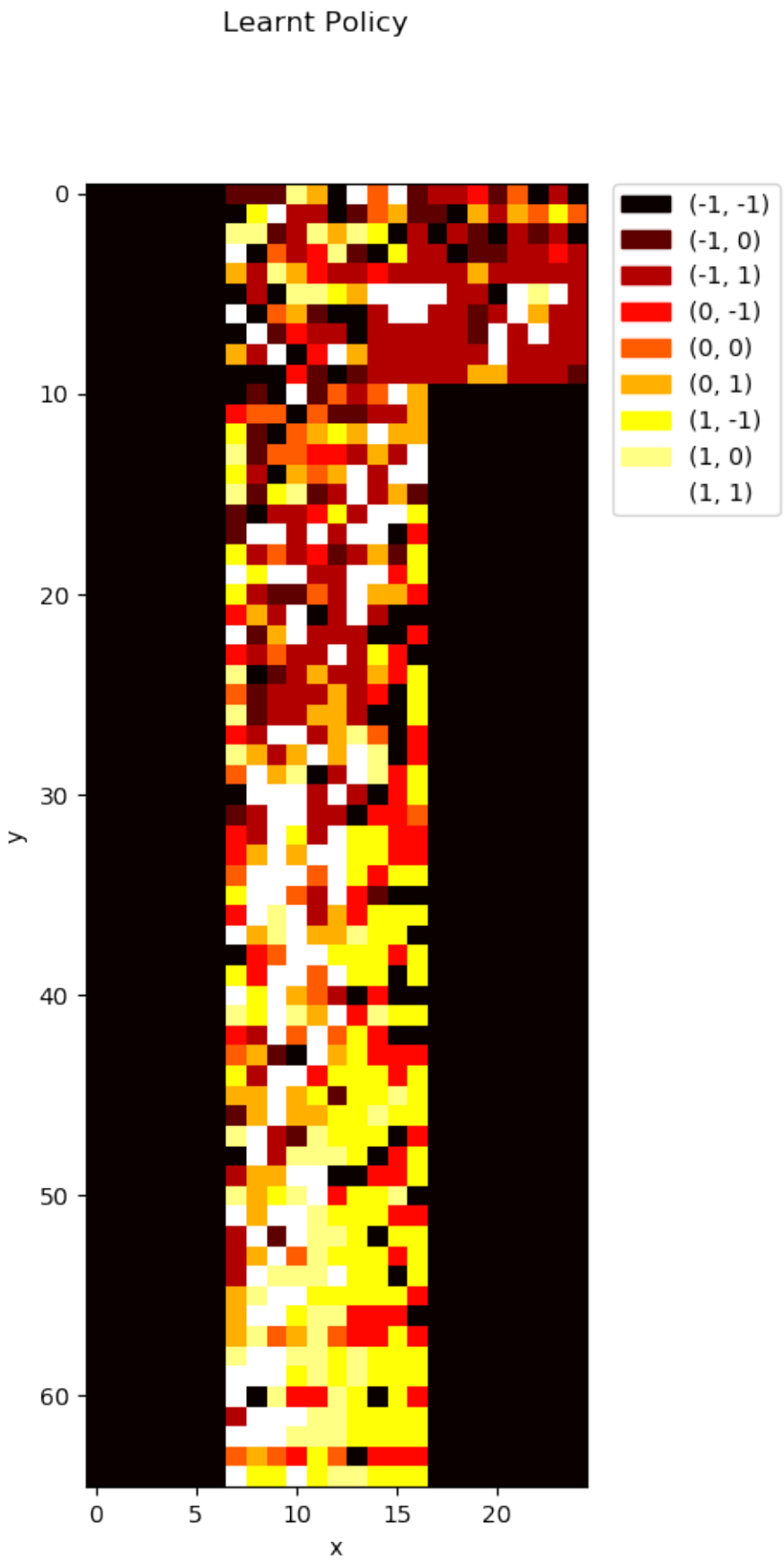
```
            if(not bEpisode_over):
                new_action = np.argmax(self.state_action_values[
                    new_state[0], new_state[1], :])
                self.state_action_values[
                    state[0], state[1], action] = value + self.alpha * (
                    reward + self.gamma * self.state_action_values[
                        new_state[0], new_state[1],
                        new_action] - value)

            else:
                self.state_action_values[
                    state[0], state[1], action] = value + self.alpha * (
                    reward - value)

            return

        def PlotEpisode(self, episode):

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

            self.PlotRacetrack(trajectory)

            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):

        policy = np.zeros((self.height, self.width))

        for row in range(self.height):
            for col in range(self.width):
                policy[row, col] = np.argmax(
                    self.state_action_values[row, col, :])

        plt.figure(figsize=(10, 10))
        im = plt.imshow(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.QLearning()
```


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

```
Episode: 0/1000000
Episode: 62500/1000000
Episode: 125000/1000000
Episode: 187500/1000000
Episode: 250000/1000000
Episode: 312500/1000000
Episode: 375000/1000000
Episode: 437500/1000000
Episode: 500000/1000000
Episode: 562500/1000000
Episode: 625000/1000000
Episode: 687500/1000000
Episode: 750000/1000000
Episode: 812500/1000000
Episode: 875000/1000000
Episode: 937500/1000000
```


Example Episodes Over Training
Episode 0 Return: -123
Episode 62500 Return: -39
Episode 125000 Return: -54
Episode 187500 Return: -40
Episode 250000 Return: -36
Episode 312500 Return: -14
Episode 375000 Return: -16
Episode 437500 Return: -44
Episode 500000 Return: -37
Episode 562500 Return: -15
Episode 625000 Return: -14
Episode 687500 Return: -16
Episode 750000 Return: -32
Episode 812500 Return: -20
Episode 875000 Return: -15
Episode 937500 Return: -21


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)

That concludes our coverage of the fundamental solution methods for the RL problem. With a good understanding of DP, MC and TD methods you are in a great position to tackle some of the more advanced topics in RL. I hope you can see the similarities and differences between each of these methods and have started to develop an intuition as to when to use one over another. In general, TD methods tend to be the most popular of the three because they can be used in an online fashion, with little computational cost, and only require sampled experience. It is worth mentioning that the TD methods outlined above are actually 'one-step, tabular, modelfree' TD methods. TD methods can be much more general than this, for example they can:

- Use information from more than one step to perform an update
- Use function approximators to calculate the value function
- Use a model of the environment to perform updates

I hope to cover some these topics in future notebooks, as well as other topics such as policy gradient methods that search for the optimal policy directly. I hope the notebooks so far have made you excited about the world of RL and that you will continue to find out more about it! If you have any questions about what we have covered so far or about RL in general please do not hesitate to contact me!