DynamicProgramming

September 24, 2018

Dynamic Programming

The first set of methods for solving the RL problem that we will cover are dynamic programming methods. The two most common methods are called policy iteration and value iteration. The key property that all dynamic programming methods share is that they require a perfect model of the environment's dynamics i.e. they require $P^a_{ss'}$ and $R^a_{ss'}$. Unfortunately this limits their applicability because this is often not the case and we either need to use trial and error experience to gain insight into the environment's dynamics or learn an approximation of the environment's dynamics. With this being said, dynamic programming methods are often a nice solution method to learn first because they provide the necessary intuitions to understand how the other methods tackle the RL problem when we don't have a perfect model of the environment's dynamics.

Both policy iteration and value iteration can be viewed as a form of generalized policy iteration (GPI), as mentioned in the previous notebook. They can therefore be broken down into a policy evaluation step and a policy improvement step. Lets begin with policy iteration and see how it tackles the problems of policy evaluation and policy improvement.

1 Policy Iteration

1.1 Policy Evaluation

Remember that for polcy evaluation we want to estimate the value function for a given policy, that is we want to calculate V^π. Since we assume that we know $P^a_{ss'}$ and $R^a_{ss'}$, policy iteration can solve this problem simply by turning the bellman equation into a sequential update rule, known as iterative policy evaluation:

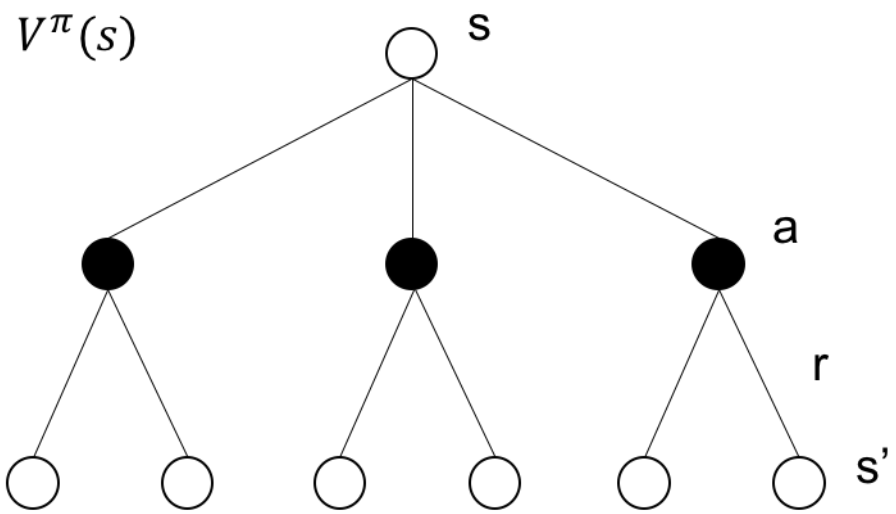
$$\begin{aligned} V_{k+1}(s) &= \mathbb{E}_\pi[r_{t+1} + \gamma V_k(s_{t+1} \mid s_t = s] \\ &= \sum_a \pi(s,a) \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V_k(s')] \qquad \text{for all } s \in S \end{aligned}$$

By definition of the bellman equation, once $V_k = V^\pi$ this update rule will not change any of the values and the iterative procedure will have converged. In general, iterative policy evaluation can be shown to converge to V^π as $k \to \infty$. An important property of this update rule is that it performs a 'full backup' because it takes into account all possible successor states from the state being

evaluated. In contrast, a 'sample backup' would only take into account a single successor state from the state being evaluated. The backup of iterative policy evaluation can also be described as a 'shallow' backup because it only looks one step ahead to make it's update. This is the opposite of a 'deep backup' which may look several states ahead in order to perform its update. Below is the backup diagram for iterative policy evaluation:

```
In [30]: from IPython.display import Image
         Image(filename='VBackup.png')
```

Out[30]:



Typically we think of iterative policy evaluation as sweeping through all the states s in the state space S applying the above update rule to each one. This process is then repeated until the magnitude of the largest update is below some small value ϵ. The algorithm can be also be run 'in-place' so that the calculated value for $V_{k+1}(s)$ is used straight away to overwrite $V_k(s)$, rather than waiting until the end of a sweep to overwrite all the value. This 'in-place' version can lead to faster convergence, however it also means that the order in which you sweep the state space will have an impact upon convergence speed.

1.2 Policy Improvement

Policy iteration improves the policy by acting greedily with respect to the current value function V^{π}. One way to think about this is that for any given state s we are looking for an action a that gives us a larger expected return given that we follow π after having taken that action. If there exists an action that does produce a larger expected return then this defines a new policy π', which is better than the previous one π:

$$Q^{\pi}(s, \pi'(s)) \geq V^{\pi}(s) \qquad \text{for all } s \in S$$
$$V^{\pi'}(s) \geq V^{\pi}(s) \text{for all } s \in S$$

This result is known as the policy improvement theorem and can be proved as follows:

$$
\begin{aligned}
V^{\pi}(s) &\leq Q^{\pi}(s, \pi'(s)) \\
&= \mathbb{E}_{\pi'}[r_{t+1} + \gamma V^{\pi}(s_{t+1}) \mid s_t = s] \qquad * \\
&\leq \mathbb{E}_{\pi'}[r_{t+1} + \gamma Q^{\pi}(s_{t+1}, \pi'(s_{t+1})) \mid s_t = s] \\
&= \mathbb{E}_{\pi'}[r_{t+1} + \gamma \mathbb{E}_{\pi'}[r_{t+2} + \gamma V^{\pi}(s_{t+2})] \mid s_t = s] \\
&= \mathbb{E}_{\pi'}[r_{t+1} + \gamma r_{t+2} + \gamma^2 V^{\pi}(s_{t+2}) \mid s_t = s] \qquad * \\
&\leq \mathbb{E}_{\pi'}[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 V^{\pi}(s_{t+3}) \mid s_t = s] \qquad * \\
&. \\
&. \\
&. \\
&\leq \mathbb{E}_{\pi'}[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + ... \mid s_t = s] \qquad * \\
&= V^{\pi'}(s)
\end{aligned}
$$

I find that following the starred lines tends to make it more obvious why the policy improvement theorem holds. Hopefully this makes it obvious why acting greedily with respect to the current value function improves the policy and why it will only stop giving us a better policy when the original policy is already optimal. The actual update for policy improvement is as follows:

$$
\begin{aligned}
\pi'(s) &= \arg\max_a Q^{\pi}(s, a) \\
&= \arg\max_a \mathbb{E}[r_{t+1} + \gamma V^{\pi}(s_{t+1}) \mid s_t = s, a_t = a] \\
&= \arg\max_a \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V^{\pi}(s')]
\end{aligned}
$$

1.3 Policy Iteration

We can now combine these two components to get the full policy iteration algorithm. As is always the case with GPI, we just need to alternate between policy evaluation and policy improvement. We start by initiliasing a random starting policy π and computing V^{π}, then we improve π using $V^p i$ to obtain a new policy π'. This process then repeats so that we get increasingly better policies and value functions. Usually when we are computing the new value function i.e. $V^{\pi'}$, we just use the old value function i.e. V^{π}, as a starting point as this helps with convergence. Below is an outline of the policy iteration algorithm in full:

```
In [31]: Image(filename='PolicyIterationPseudocode.png')
```

Out[31]:

Algorithm 1 Policy Iteration

Initialize $V(s) \in \mathbb{R}$ and $\pi(s) \in A(s)$ arbitrarily for all $s \in S$
$\theta \leftarrow$ a small positive number

1. Perform Policy Evaluation
repeat
  $\Lambda \leftarrow 0$
  for each $s \in S$ **do**
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_{s'} P_{ss'}^{\pi(s)}[R_{ss'}^{\pi(s)} + \gamma V(s')]$
    $\Lambda \leftarrow \max(\Lambda, \mid v - V(s) \mid)$
  end for
until $\Lambda < \theta$

2. Perform Policy Improvement
$policy\_stable \leftarrow true$
for each $s \in S$ **do**
  $b \leftarrow \pi(s)$
  $\pi(s) \leftarrow \arg\max_a \sum_{s'} P_{ss'}^{a}[R_{ss'}^{a} + \gamma V(s')]$
  if $b \neq \pi(s)$ **then**
    $policy\_stable = false$
  end if
end for
if *policy_stable* **then**
  stop
else
  go to 1
end if

2 Value Iteration

The fundamental difference between policy iteration and value iteration is how they alternate between the policy evaluation and the policy improvement steps. As we have just seen, policy iteration performs policy evaluation until convergence and then performs policy improvement until convergence. However it is not a strict requirement that we run each of these steps until convergence before switching to the other one. We can in fact run as many policy evaluation steps as we want before switching to policy improvement, which leaves us with a truncated form of policy iteration.

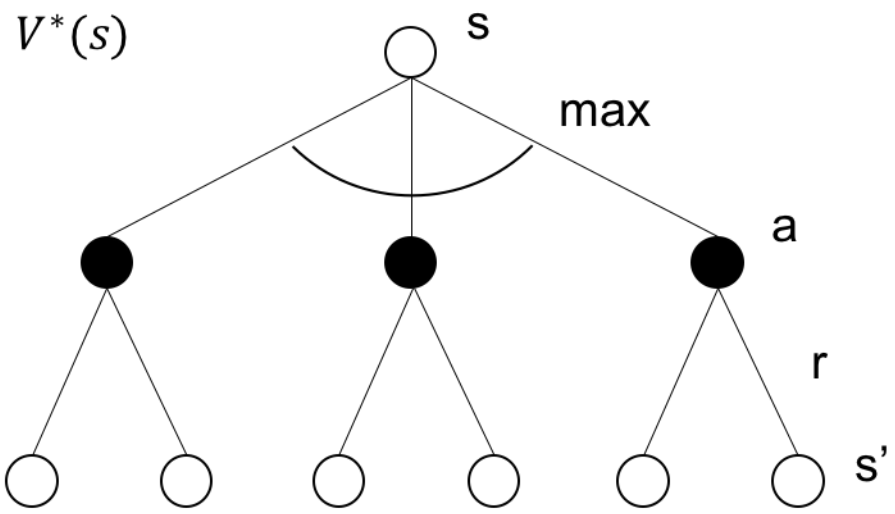
Value iteration is an extreme case of truncated policy iteration whereby for each sweep over $s \in S$, one step of policy evaluation is combined with one step of policy improvement. In other words policy evaluation is stopped after just one sweep of the states. The update rule for value iteration is as follows:

$$V_{k+1}(s) = \max_a \mathbb{E}[r_{t+1} + \gamma V_k(s_{t+1}) \mid s_t = s, a_t = a]$$
$$= \max_a \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V_k(s')]$$

Note how this is just the Bellman optimality equation in the form of an update rule! The following backup diagram for value iteration is therefore exactly the same as the one for V^*:

In [32]: `Image(filename='VOptimalBackup.png')`

Out[32]:



Since the policy evaluation and policy improvement steps are combined into a single update rule, the full algorithm for value iteration is relatively simple:

In [33]: `Image(filename='ValueIterationPseudocode.png')`

Out[33]:

Algorithm 1 Value Iteration

Initialize $V(s) \in \mathbb{R}$ arbitrarily for all $s \in S$
$\theta \leftarrow$ a small positive number

repeat
  $\Lambda \leftarrow 0$
  for each $s \in S$ **do**
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V(s')]$
    $\Lambda \leftarrow \max(\Lambda, \mid v - V(s) \mid)$
  end for
until $\Lambda < \theta$

Output a deterministic policy π such that
$\pi(s) = \arg\max_a \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V(s')]$

As an additional note, we have seen that both policy and value iteration use entire sweeps through all $s \in S$. If S is large then this can be extremely expensive in terms of computation time. Interestingly we don't actually have to sweep through all $s \in S$ and instead we can choose any arbitrary order in which to perform the backups. We could for example back up some states many times before backing up others. Methods that do this are called asynchronous dynamic programming methods. The only requirement for convergence is that the values of all states must be backed up i.e. no states can be fully ignored. One major advantage of asynchronous methods is that the agent can improve its policy before having to wait for a full sweep of S.

3 Example - Jack's Car Rental (Sutton and Barto, 1998)

Often the best way to understand an RL solution method is to work through an example, we shall therefore use dynamic programming to solve the classic 'Jack's Car Rental' problem from the book Reinforcement Learning: An Introduction (Sutton and Barto, 1998). Lets start by outlining the problem and characterising the components of the MDP that we need to solve. The general problem is as follows:

- Jack runs a car rental business at two different locations
- Each day customers arrive and rent cars for \$10 per car
- Cars are available for rental the day after they are returned by a customer
- To ensure that there are always cars available to rent at each of the locations, Jack can move up to 5 cars overnight from one location to the other at a cost of \$2
- There is a maximum limit of 20 cars per location
- We assume that the rentals and returns follow poisson distributions i.e. $P(n) = \frac{\lambda^n}{n!}e^{-\lambda}$. For rentals at location 1 and 2, $\lambda = 3$ and 4 respectively, whereas for returns $\lambda = 3$ and 2 respectively

Using this description we can define the problem as an MDP with the following components:

1. S - The state signal is the number of cars at each location at the end of the day
2. A - The actions are the number of cars moved between each location overnight
3. $P^a_{ss'}$ - The transition function is defined by the poisson distributions
4. $R^a_{ss'}$ - The reward function is defined by our rental and movement costs
5. γ - We shall set the discount factor to be 0.9

It is also worth noting that our time steps are days i.e. a_{10} is the number of cars moved by Jack at the end of day 10. Just from looking at this problem it is not immediately obvious how Jack should move the cars in order to maximise profit. Fortunately for such a small problem where we know the environment's dynamics we can use dynamic programming to do the hard work for us.

The code below creates a class that describes the MDP defined by Jack's Car Rental problem and defines the functions needed for policy and value iteration. Some key components of the code are as follows:

1. init() $\rightarrow$ Initalises a few key variables and also calculates the probabilities needed to describe the transition function $P^a_{ss'}$. $P^a_{ss'}$ is stored as a tabular representation in *rental_probs* and *return_probs* because our state space is relatively small. Remember that we define $P^a_{ss'}$ as the probability of ending up in state s' having taken action a in state s. We only calculate these probabilities up to *poisson_upper* because for any n > *poisson_upper* the probability is neglible for our values of lambda.
2. ExpectedReturn() $\rightarrow$ Calculates $\sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V_k(s')]$ i.e. the expected return for taking action a from state s. After Jack has moved a certain number of cars between locations (i.e. selected action a in state s) this function iterates through all possible rental and return scenarios for the next day (i.e. all s') and uses their probability of occuring to compute the expected return.
3. PlotPolicy() $\rightarrow$ Plots a 2D graph that represents the current policy. The y-axis shows the number of cars at location one and the x-axis shows the number of cars at location two. The colour represents the number of cars Jack would move from location one to location two given that number of cars at each location. A negative value corresponds to Jack moving cars from location 2 to location 1.
4. PlotValueFunction $\rightarrow$ Plots a 2D graph that represents the current value function estimate. The axis are the same as PlotPolicy() but this time the colour represents the expected return from that state.

```
In [4]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt
        import matplotlib as mpl
        mpl.rcParams['figure.dpi']= 100
```

```
class JacksCarRental(object):

    def __init__(self):

        # Gamma (discount factor)
        self.discount = .9

        # S (state space)
        self.num_locations = 2
        self.max_cars = 20
        self.state_values = np.zeros((self.max_cars + 1, self.max_cars + 1))

        # Pi (policy)
        self.policy = np.zeros((self.max_cars + 1, self.max_cars + 1))

        # R (reward function)
        self.rental_reward = 10
        self.movement_cost = 2

        # P (transition function)
        self.max_to_move = 5
        self.poisson_upper = 10
        self.lambda_rentals = np.array([3, 4])
        self.lambda_returns = np.array([3, 2])

        self.rental_probs = np.zeros((2, self.poisson_upper + 1))

        for loc in range(self.num_locations):
            for i in range(self.poisson_upper+1):
                self.rental_probs[loc, i] = (np.power(self.lambda_rentals[loc], i) /
                                             np.math.factorial(i)) * np.exp(
                    -self.lambda_rentals[loc])

        self.return_probs = np.zeros((2, self.poisson_upper + 1))

        for loc in range(self.num_locations):
            for i in range(self.poisson_upper+1):
                self.return_probs[loc, i] = (np.power(self.lambda_returns[loc], i) /
                                             np.math.factorial(i)) * np.exp(
                    -self.lambda_returns[loc])

        return

    def ExpectedReturn(self, num_loc1, num_loc2, action):

        expected_return = -self.movement_cost * np.abs(action) # cost of action

        # number at the start of the day at loc 1
```

```
            morning_loc1 = min(num_loc1 - action, self.max_cars)
            # number at the start of the day at loc 2
            morning_loc2 = min(num_loc2 + action, self.max_cars)

            # Iterate over all possible rental and return scenarios i.e. all possible s'
            for rent_num_loc1 in range(self.poisson_upper+1):

                for rent_num_loc2 in range(self.poisson_upper+1):

                    rent_prob = self.rental_probs[
                        0, rent_num_loc1] * self.rental_probs[1, rent_num_loc2]

                    total_rent_loc1 = min(morning_loc1, rent_num_loc1)
                    total_rent_loc2 = min(morning_loc2, rent_num_loc2)

                    rewards = (total_rent_loc1 + total_rent_loc2) * self.rental_reward

                    for return_num_loc1 in range(self.poisson_upper+1):

                        for return_num_loc2 in range(self.poisson_upper+1):

                            return_prob = self.return_probs[
                                0, return_num_loc1] * self.return_probs[
                                1, return_num_loc2]

                            evening_loc1 = min(morning_loc1 - total_rent_loc1 +
                                               return_num_loc1, self.max_cars)
                            evening_loc2 = min(morning_loc2 - total_rent_loc2 +
                                               return_num_loc2, self.max_cars)

                            # Weight outcome by probability and add to expected
                            # return value
                            expected_return += rent_prob * return_prob * (
                                rewards + (self.discount * self.state_values[
                                    int(evening_loc1), int(evening_loc2)]))

        return expected_return


    def PolicyIteration(self):

        bPolicy_stable = False
        iter = 0

        while(not bPolicy_stable):

            print("Policy Iteration Step: " + str(iter))
            self.PolicyEvaluation()
```

```
            bPolicy_stable = self.PolicyImprovement()

            iter += 1

        self.PlotPolicy()
        self.PlotValueFunction()

        return


    def PolicyEvaluation(self):

        bEvaluation_finished = False

        while(not bEvaluation_finished):

            theta = .1
            diff = 0

            for num_loc1 in range(self.max_cars+1):

                for num_loc2 in range(self.max_cars+1):

                    old_value = self.state_values[num_loc1, num_loc2]
                    action = self.policy[num_loc1, num_loc2]

                    new_value = self.ExpectedReturn(num_loc1, num_loc2, action)
                    self.state_values[num_loc1, num_loc2] = new_value

                    diff = max(diff, np.abs(old_value - new_value))

            if(diff < theta):
                bEvaluation_finished = True

        return

    def PolicyImprovement(self):

        bPolicy_stable = True

        for num_loc1 in range(self.max_cars+1):

            for num_loc2 in range(self.max_cars+1):

                current_action = self.policy[num_loc1, num_loc2]
                action_values = np.zeros(((self.max_to_move * 2) + 1))

                for action in range(-self.max_to_move, self.max_to_move + 1):
```

```
                if((action >= 0 and num_loc1 >= action) or
                   (action < 0 and num_loc2 >= np.abs(action))):
                    action_values[
                        action + self.max_to_move] = self.ExpectedReturn(
                        num_loc1, num_loc2, action)
                else:
                    action_values[action + self.max_to_move] = -9999999

            best_action = np.argmax(action_values) - self.max_to_move
            self.policy[num_loc1, num_loc2] = best_action

            if(current_action != best_action):
                   bPolicy_stable = False


    return bPolicy_stable

def ValueIteration(self):

    bPolicy_stable = False
    iter = 0

    while(not bPolicy_stable):

        print('Value Iteration Step: ' + str(iter))
        bPolicy_stable = self.ValueIterationUpdate()

        iter += 1

    self.PlotPolicy()
    self.PlotValueFunction()


    return

def ValueIterationUpdate(self):

    bPolicy_stable = False
    theta = .1
    diff = 0

    for num_loc1 in range(self.max_cars+1):

        for num_loc2 in range(self.max_cars+1):

            old_value = self.state_values[num_loc1, num_loc2]
            action_values = np.zeros(((self.max_to_move * 2) + 1))
```

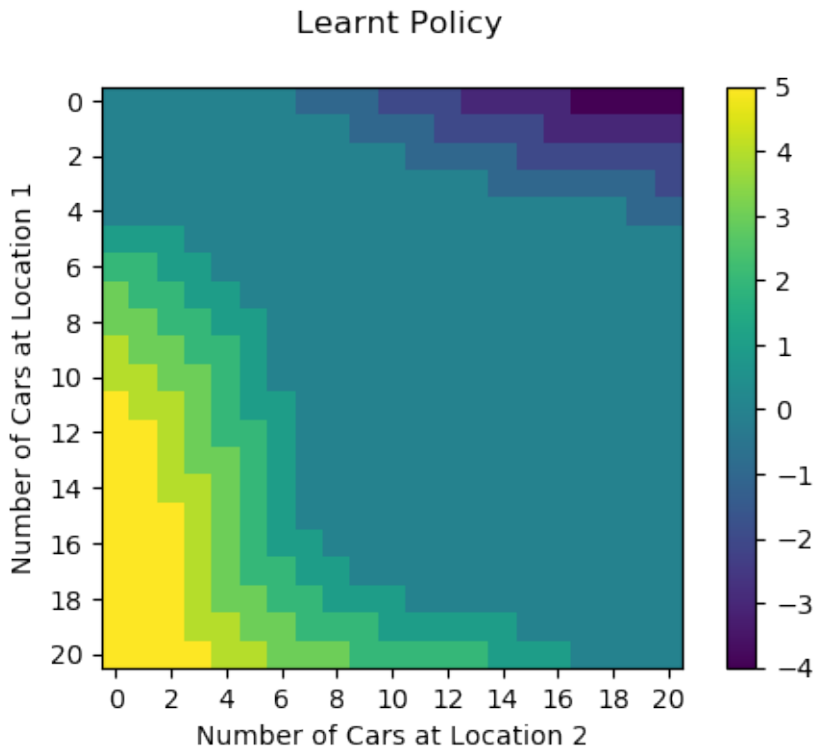
```
                for action in range(-self.max_to_move, self.max_to_move+1):

                    if((action >= 0 and num_loc1 >= action) or
                       (action < 0 and num_loc2 >= np.abs(action))):

                       action_values[
                           action + self.max_to_move] = self.ExpectedReturn(
                           num_loc1, num_loc2, action)

                    else:

                       action_values[action + self.max_to_move] = -9999999

                self.state_values[num_loc1, num_loc2] = np.amax(action_values)
                self.policy[num_loc1, num_loc2] = np.argmax(action_values)
                - self.max_to_move

                diff = np.amax([diff, np.abs(old_value -
                                             self.state_values[
                                                 num_loc1, num_loc2])])

        if(diff < theta):
            bPolicy_stable = True

        return bPolicy_stable

    def PlotPolicy(self):

        plt.figure()
        plt.imshow(self.policy)
        plt.colorbar()
        plt.suptitle('Learnt Policy')
        plt.ylabel('Number of Cars at Location 1')
        plt.xlabel('Number of Cars at Location 2')
        plt.xticks(np.arange(0, self.max_cars+1, 2))
        plt.yticks(np.arange(0, self.max_cars+1, 2))
        plt.show()

        return

    def PlotValueFunction(self):

        plt.figure()
        plt.imshow(self.state_values)
        plt.colorbar()
        plt.suptitle('Learnt Value Function')
        plt.ylabel('Number of Cars at Location 1')
```

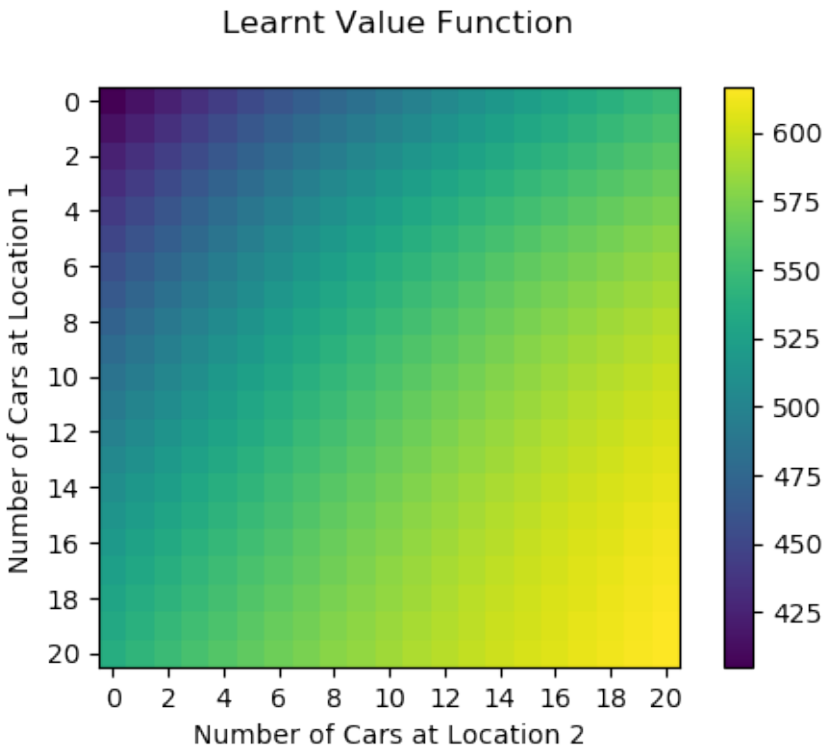
```
            plt.xlabel('Number of Cars at Location 2')
            plt.xticks(np.arange(0, self.max_cars+1, 2))
            plt.yticks(np.arange(0, self.max_cars+1, 2))
            plt.show()

            return

        def Reset(self):

            self.state_values = np.zeros((self.max_cars + 1, self.max_cars + 1))
            self.policy = np.zeros((self.max_cars + 1, self.max_cars + 1))

            return
```

With the class defined lets run both policy and value iteration and see what policies they learn.

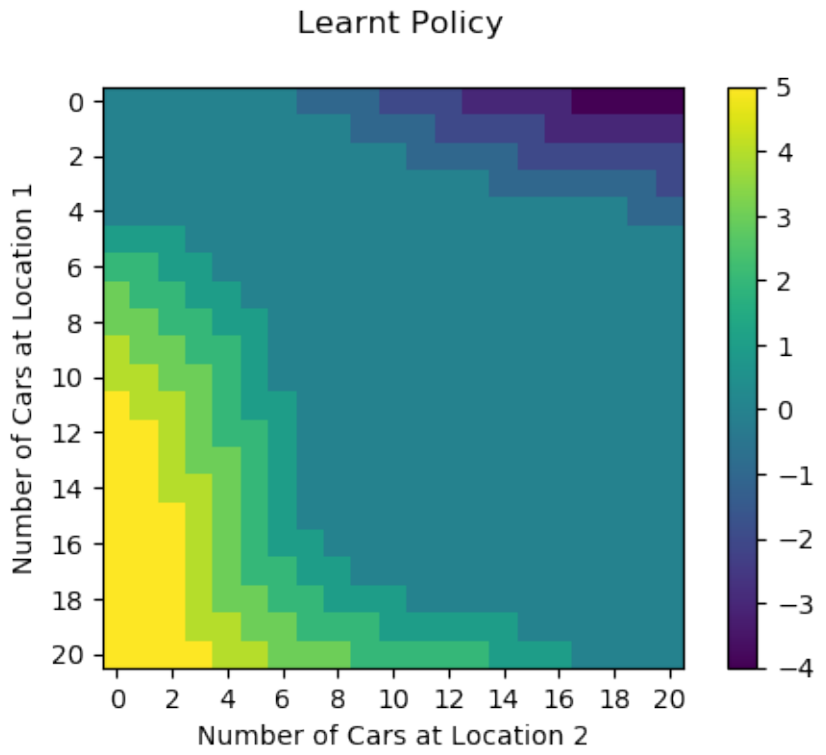
```
In [42]: jcr_game = JacksCarRental()
         jcr_game.PolicyIteration()
         jcr_game.Reset()
         jcr_game.ValueIteration()
```

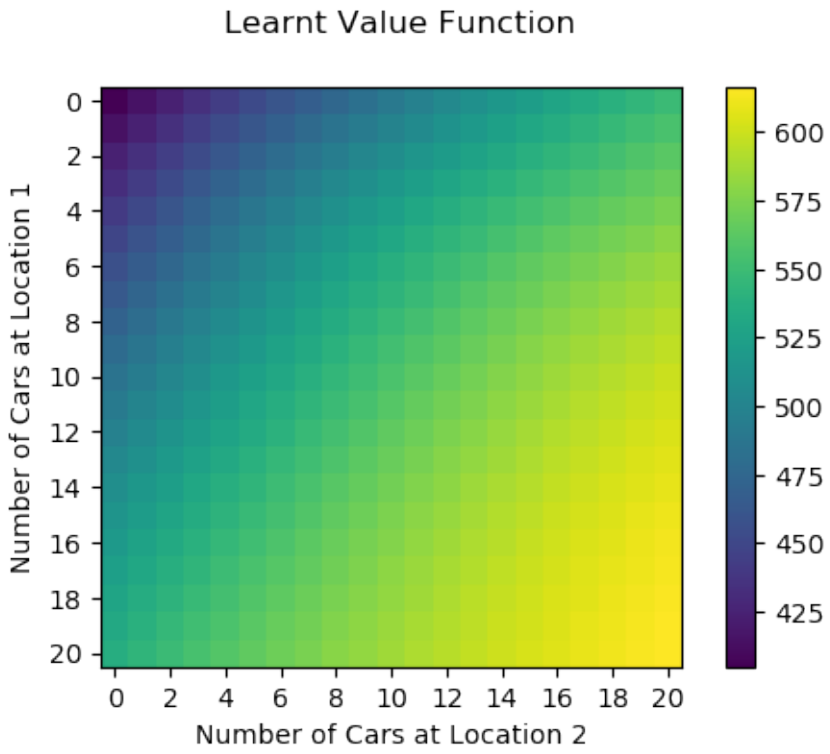
```
Policy Iteration Step: 0
Policy Iteration Step: 1
Policy Iteration Step: 2
Policy Iteration Step: 3
Policy Iteration Step: 4
```


Learnt Policy
Number of Cars at Location 1
Number of Cars at Location 2
0
2
4
6
8
10
12
14
16
18
20
5
4
3
2
1
0
−1
−2
−3
−4


Learnt Value Function
Number of Cars at Location 1
Number of Cars at Location 2
0
2
4
6
8
10
12
14
16
18
20
600
575
550
525
500
475
450
425

```
Value Iteration Step: 0
Value Iteration Step: 1
Value Iteration Step: 2
Value Iteration Step: 3
Value Iteration Step: 4
Value Iteration Step: 5
Value Iteration Step: 6
Value Iteration Step: 7
Value Iteration Step: 8
Value Iteration Step: 9
Value Iteration Step: 10
Value Iteration Step: 11
Value Iteration Step: 12
Value Iteration Step: 13
Value Iteration Step: 14
Value Iteration Step: 15
Value Iteration Step: 16
Value Iteration Step: 17
Value Iteration Step: 18
Value Iteration Step: 19
Value Iteration Step: 20
Value Iteration Step: 21
Value Iteration Step: 22
Value Iteration Step: 23
Value Iteration Step: 24
Value Iteration Step: 25
Value Iteration Step: 26
Value Iteration Step: 27
Value Iteration Step: 28
Value Iteration Step: 29
Value Iteration Step: 30
Value Iteration Step: 31
Value Iteration Step: 32
Value Iteration Step: 33
Value Iteration Step: 34
```


Learnt Policy
Number of Cars at Location 1
Number of Cars at Location 2
0
2
4
6
8
10
12
14
16
18
20
5
4
3
2
1
0
−1
−2
−3
−4


Learnt Value Function
Number of Cars at Location 1
Number of Cars at Location 2
0
2
4
6
8
10
12
14
16
18
20
600
575
550
525
500
475
450
425

Using just a few lines of code we have managed to produce a policy that will allow Jack to maximise his profit! Hopefully you can see from the policy plots that both algorithms have learnt sensible policies. For example, when there are many cars at location 1 but few cars at location 2 (bottom left corner of the plots) then it makes sense to move many cars overnight from location 1 to location 2. The plots are not symmetric because both algorithms take into account the different rates of rentals and returns at each location, as described by their respective poisson distributions.

I hope this simple example of using dynamic programming methods has helped to develop your understanding of them. The most important thing to remember is that dynamic programming is characterised by the following properties:

1. It requires a perfect model of the evnironment's dynamics
2. It uses Full backups
3. It is computationally expensive
4. It requires no 'real-world' experience
5. It 'bootstraps' information i.e. it estimates the state value using its own estimates of successor state values

While for simple problems like Jack's Car Rental these properties are agreeable, there are many examples of RL problems where this isn't the case. In particular points 1 and 3 are extremely limiting and so we must turn to other solution methods when they become prohibitive. In the next notebook we shall look at solution methods that do not require a perfect model of the environment's dynamics and that rely on trial and error through 'real-world' experience to solve the RL problem.