Dynamic Programming

A natural start on the journey to mastering reinforcement learning is Dynamic Programming (DP). DP is one of the foundational subdomains in reinforcement learning and introduces the basic concepts used in many more advanced reinforcement learning topics. DP was originally introduced by the applied mathematician Richard Bellman in the 1950s as a theory to treat the mathematical problems arising from the study of various multi-stage decision processes (Bellman, 1954) . DP often considers problems which are recursive in nature and has many applications in domains such as engineering, mathematics and industrial control.

Reinforcement learning, loosely stated, is a means by which an agent learns an optimal policy for obtaining some goal, or reward, in an environment. The quintessential example of reinforcement learning is one in which an agent attempts to navigate a specific grid world environment to achieve the goal of arriving at a given cell while avoiding obstacles as shown in Figure 1. In each cell of the grid world the agent has the ability to take one of four actions (up, down, left, or right). The agent moves through the grid world by taking actions from the cell in which the agent currently resides, to land in a new cell. The set of cell transitions, determined by the actions taken by the agent, is the policy used by the agent. The dotted line followed by the agent in Figure 1 represents an optimal policy of transitions the agent should take to achieve the goal (arriving at the star). The gray shaded cells represent obstacles in the grid world.

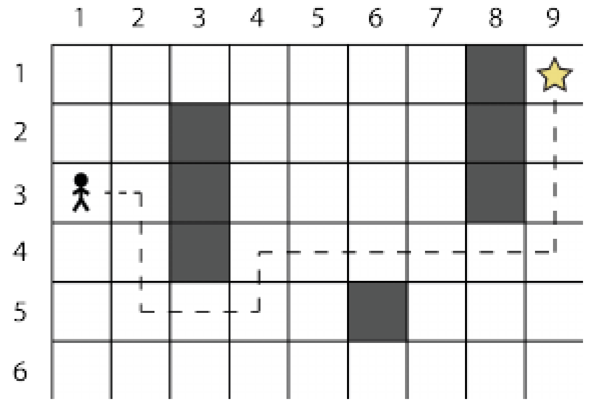

Figure 1: Example grid world environment.

In the parlance of reinforcement learning, each cell of the grid world is called a state. The agent receives a reward based upon this state, the action the agent executes from that state, and the resulting new state that the agent enters by taking the action. Suppose that the agent receives a reward of -0.1 for every state, action, new state tuple when the new state is not the goal state, and a reward of 10 otherwise. Given that scenario, an agent that wanders around aimlessly for an extended period will receive a much lower reward than an agent which follows the optimal policy and arrives at the goal in rapid fashion. In the case of the optimal policy given by the dashed lines in Figure 1, the agent would receive a total reward of 10 – (0.1 x 13) = 8.7. The optimal policy delivers the maximum reward to the agent.

The grid world reinforcement learning problem described above is a perfect example of a problem that dynamic programming is capable of solving. Dynamic programming can utilize the recursive nature of the state, action, new state tuple to generate an optimal policy for transitioning through the grid world. Dynamic programming utilizes a mathematical formula known as the Bellman equation, for determining the optimal policy. The remainder of this article will illustrate how the Bellman equation can be used to derive the optimal policy for a grid world environment.

Before diving deeply into Bellman equation, and the actual grid world reinforcement learning exercise, it is important to formally define some of the concepts that have been loosely given up to this point. The first of these terms is **reward**. As stated previously the agent receives some form of reward for each step, or action, it takes in the environment. Reward, *r*, is often given numerically, especially in reinforcement learning, and can be positive, negative, or zero. Reward, or return as it is often called, is used to guide the agent towards the solution to the reinforcement learning exercise, and can be one of the hardest parts of the reinforcement learning program to design. In an environment in which there is a specific goal given, the cumulative reward an agent receives should be considered a finite value as there is a definite end to the exercise (once the agent reaches the goal). Cumulative reward, *R*, or total return, for finite exercises is given by Eq. 1 below. We will not give full consideration to exercises with infinite time horizons in this article.

$$R_t = r_{t+1} + r_{t+2} + r_{t+3} + \ldots + r_{t+n} = \sum_{k=1}^{n} r_{t+k} \qquad \text{Eq. 1}$$

It is very common to consider a discounted reward rather than simply cumulative reward. Discounted reward is a reward function in which successive future rewards are reduced in some , often exponential, fashion. The reason for discounting future rewards is two fold. First, discounting future rewards allows for better reward handling when the time horizon is infinite (not considered in this article) and second, it is often more desirable to receive rewards sooner rather than later (Blackwell, 1965; Tsitsiklis & Van Roy, 2002). Discounting future rewards is performed via a discount factor, γ, that is exponentially increased the further in the future the factor is applied. Cumulative discounted reward is then calculated as given in Eq. 2, below.

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots + \gamma^{n-1} r_{t+n} = \sum_{k=1}^{n} \gamma^{k-1} r_{t+k} \qquad \text{Eq. 2}$$

Considering the discount factor, γ, in Eq. 2 above, one can see that if $\gamma = 1$, then Eq. 2 becomes Eq. 1. When $\gamma = 0$ then future rewards are not considered at all and the algorithm only considers the current reward. The case most frequently used is one in which $0 < \gamma < 1$. The smaller γ is, the smaller effect future rewards have on the discounted cumulative reward.

Reward is used in reinforcement learning to help determine the optimal **policy** the agent should follow. As stated above, a policy defines the way an agent acts within the environment. A policy can be viewed as a function that takes in an action and a state and returns the probability of taking that action in that state. This is often given as π(*s*,*a*), where π is the policy function, *s* is the state, and *a* is the action. Due to the fact that π(*s*,*a*) returns a probability for the action in the state, the sum of all π(*s*,*a*) for all actions taken in *s* should equal 1 as given in Eq. 3.

$$\sum_a \pi(s,a)=1 \qquad \text{Eq. 3}$$

The goal of reinforcement learning is to learn a optimal policy, π^*. The optimal policy instructs the agent on how to act to receive the optimal reward in the environment. Most often, there is a single optimal action to take in each state, written as $\pi^*(s) = a$. To learn the optimal policy, the agent utilizes yet another set of functions, known as the **state value function** and the **action value function**. The state value function, written as *V*(*s*), gives the value of the **expected reward**, *E*, when starting in a given state *s* and following the policy π as given in Eq. 4 below.

$$V^\pi(s)=E_\pi[R_t|s_t=s] \qquad \text{Eq. 4}$$

The state value function depends completely on the current policy and therefore can change as the policy changes. This is due to the fact that as the agent changes the way in which it acts at each state, the reward the agent receives for taking the action also changes. Similar to the state value function, the action value function returns the expected reward for taking the action in the given state when following the policy π as given in Eq. 5.

$$Q^\pi(s)=E_\pi[R_t|s_t=s,a_t=a] \qquad \text{Eq. 5}$$

The final concept to consider prior to delving into the actual grid world exercise is known as the Bellman Equations or the Bellman Updates. Bellman Equations are well known for optimizing Markov Decision Processes (Sutton & Barto, 2014). Markov Decision Processes (MDP) are a mathematical framework for understanding decision making in discrete time. A MDP considers a system that at each time step is in a given state in which an agent may choose an action to perform from a set of actions that will result in a new state. If this sound familiar, it should. MDPs are a concise mathematical means by which one can describe the reinforcement learning scenario of the previously discussed grid world (please see (Howard, 1958), for a complete explanation of MDPs). Due to the fact that the Bellman Equation can optimally solve MDPs and the fact that MDPs are simply a mathematical representation of the reinforcement learning problem attempted herein, the Bellman Equation can be used to optimally solve the grid world problem.

As will be shown, The Bellman Equation simply states that if an agent behaves optimally in the current state and follows an optimal policy thereafter, the agent will obtain the maximum reward for the problem, thus solving the problem. The Bellman Equation considers the probability, *p*, of transitioning from one state to another and the expected reward *R* received for starting in state *s*, taking action *a*, and moving to state *s' (Greaves, 2017).* Eq. 6 mathematically describes the probability *p* of starting in state *s* and taking action *a* to move to state *s'*.

$$p_a^{ss'} = Pr(s_{t+1} = s' | s_t = s, a_t = a) \qquad \text{Eq. 6}$$

Eq. 7 describes the expected reward *R* received for starting in state *s*, taking action *a*, and moving to state *s'*.

$$R_{ss'}^a(s) = E[r_{t+1} | s_t = s, s_{t+1} = s', a_t = a] \qquad \text{Eq. 7}$$

With the definitions of the transition probabilities and the expected reward given above it is now possible to define the Bellman Equation mathematically. The Bellman Equation can be derived for both the state value function and the action value function given above. The first step in deriving the Bellman Equation for the state value function is to rewrite Eq. 4 by substituting Eq. 2. for the return as given in Eq. 8 below.

$$V^\pi(s) = E_\pi[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... | s_t = s] = E_\pi[\sum_{k=1}^{n} \gamma^{k-1} r_{t+k+1} | s_t = s] \qquad \text{Eq. 8}$$

Knowing that the agent will follow the policy π in the future, it is possible to pull the first reward from the sum as given in Eq. 9.

$$V^\pi(s) = E_\pi[r_{t+1} + \gamma \sum_{k=1}^{n} \gamma^{k-1} r_{t+k+1} | s_t = s] \qquad \text{Eq. 9}$$

Doing so allows for describing the expected return over all possible actions and all possible states as given in Eq. 10 below and substituting the latter part of Eq. 9 into Eq. 10 produces Eq. 11

$$E_\pi[r_{t+1} | s_t = s] = \sum_a \pi(s,a) \sum_{s'} p_{ss'}^a R_{ss'}^a \qquad \text{Eq. 10}$$

$$E_\pi[\gamma \sum_{k=1}^{\infty} \gamma^{k-1} r_{t+k+1} | s_t = s] = \sum_a \pi(s,a) \sum_{s'} p_{ss'}^a \gamma \, E_\pi[\sum_{k=1}^{n} \gamma^k r_{t+k+1} | s_{t+1} = s'] \qquad \text{Eq. 11}$$

By distributing the expectation between the two parts of Eq. 11 it is possible to form Eq. 12.

$$V^{\pi}(s)=\sum_{a}\pi(s,a)\sum_{s'}p_{ss'}^{a}[R_{ss'}^{a}+\gamma E_{\pi}[\sum_{k=1}^{n}\gamma^{k-1}r_{t+k+1}|s_{t+1}=s']] \qquad \text{Eq. 12}$$

Finally, noting that Eq. 4 is of the same form as the latter part of Eq. 12 a simple substitution can be made to give Eq. 13 which is the Bellman Equation for the state value function.

$$V^{\pi}(s)=\sum_{a}\pi(s,a)\sum_{s'}p_{ss'}^{a}[R_{ss'}^{a}+\gamma V^{\pi}(s')] \qquad \text{Eq. 13}$$

The Bellman Equation for the action value function, given in Eq. 14, can be derived in a similar manner.

$$Q^{\pi}(s,a)=\sum_{s'}p_{ss'}^{a}[R_{ss'}^{a}+\gamma\sum_{a'}\pi(s',a')Q^{\pi}(s',a')] \qquad \text{Eq. 14}$$

The Bellman Equations allow for the determination of optimality of the policy. This is known as the principle of optimality or **Bellman Optimality**. The principle of optimality can be stated as a theorem stated as,

A policy π(a|s) achieves the optimal value from state s, I.E. $V_{\pi}(s)=V^{Opt}(s)$ *, if and only if, for any state s' reachable from s, π achieves the optimal value from state s',* $V_{\pi}(s')=V^{Opt}(s')$

With the Bellman Equations and Optimality, given above, it is now possible to generate an optimal policy for acting in the grid world problem described above. The next section of this article shall focus on applying the Bellman Equations to that exercise.

The Grid World Exercise

The task of reinforcement learning, in the case of the grid world, is to discover the optimal policy by taking actions and receiving reward as described above. For each state the agent finds itself in, four actions exist and a reward can be determined for each of these state-action pairs through exploring the environment. The reward for the state-action pairs can be recorded in a tabular format for easy look up and overall mathematical manipulation. Table 1 illustrates such a table for the last two cells ([3,9] and [2,9]) of the grid world in Figure 1.

At the beginning of the learning exercise, the state-action table is populated with all zeros as the agent has yet to determine the value of executing any actions in any given state. Thus, the state-action table defines the value of executing the action in the given state. As the agent moves about in the grid world the transitions also represent an iteration of values received. Value Iteration, as this methodology is often called, can be used to instruct the agent along the path towards the goal. The agent, in a current state, can refer to the state-action table, and choose the action which returns the maximum value (this is known as acting in a greedy manner). If all of the actions are equal, such as at the beginning of the learning exercise, then a random action is chosen and the resulting value is recorded in the state-action table. The process of choosing the appropriate action to maximize the agent's reward can be seen as the execution of a function.

Table 1: Subset of a state-action pair reward table.

State	Action	Reward
[3,9]	Up	-0.1
[3,9]	Down	-0.1
[3,9]	Left	-0.1
[3,9]	Right	-0.1
[2,9]	Up	10
[2,9]	Down	-0.1
[2,9]	Left	-0.1
[2,9]	Right	-0.1

After executing the chosen action and receiving a reward, the agent updates the table with the new information it just learned by executing that action through the application of the Bellman Equations. The Bellman Equations will help to ensure that the agent develops better policies as learning continues. Updating the current policy means determining if the value of the current policy is the best value for the current state, given optimal action in successive states, or if the current policy should be updated. This determination is performed by selecting action *a* in *s* and following the existing policy from then on and calculating the cumulative reward for the action. This value is given by the Bellman Equation for the action value function shown in Eq. 14.

The key concern in updating the policy is if the action *a* chosen in state *s* is better than the maximum value for that state, as given in the state value function of Eq. 13. If the value is larger than the maximum value, then it is better to select action *a* in state *s* and then follow the current policy thereafter. In this case, it would be desirable to update the current policy to indicate that taking action *a* in state *s* provides the maximum value. Thus, the look up table would be updated with the new action value function $q(s,a)$ as calculated using Eq. 14.

As the agent continues to explore the environment of the grid world, the state action table is constantly being updated following the previous methodology. Once the table ceases to change further, it is known that the optimal policy can be extracted from the table, given the principles of the Bellman Equations have been followed. Policy extraction from the table is performed by simply selecting the maximum action value for each state in the table as given in Eq. 15. Thus, the Bellman optimality, described by the equations of the previous section is satisfied.

$$v_{\pi} = \max_{a \in A} q_{\pi}(s, a) \qquad \text{Eq. 15}$$

Programming the Grid World Exercise

This section details how to program the grid world exercise in python. The environment used herein will be supplied by an extension of the OpenAI gym library (OpenAI, n.d.). The OpenAI gym library is a python library that contains a set of RL environments and can be installed via pip ("PyPi," n.d.). The extension of the gym environment is given below.

```
import io
import numpy as np
import sys
from gym.envs.toy_text import discrete

UP = 0
RIGHT = 1
DOWN = 2
LEFT = 3

class GridworldEnv(discrete.DiscreteEnv):
    """
    Grid World environment from Sutton's Reinforcement Learning book chapter 4.
    You are an agent on an MxN grid and your goal is to reach the terminal
    state at the top left or the bottom right corner.
    For example, a 4x4 grid looks as follows:
    T  o  o  o
    o  x  o  o
    o  o  o  o
    o  o  o  T
    x is your position and T are the two terminal states.
    You can take actions in each direction (UP=0, RIGHT=1, DOWN=2, LEFT=3).
    Actions going off the edge leave you in your current state.
    You receive a reward of -1 at each step until you reach a terminal state.
    """

    metadata = {'render.modes': ['human', 'ansi']}
```

```
def __init__(self, shape=[4,4]):
    if not isinstance(shape, (list, tuple)) or not len(shape) == 2:
        raise ValueError('shape argument must be a list/tuple of length 2')
    self.shape = shape
    nS = np.prod(shape)
    nA = 4
    MAX_Y = shape[0]
    MAX_X = shape[1]
    P = {}
    grid = np.arange(nS).reshape(shape)
    it = np.nditer(grid, flags=['multi_index'])
    while not it.finished:
        s = it.iterindex
        y, x = it.multi_index
        # P[s][a] = (prob, next_state, reward, is_done)
        P[s] = {a : [] for a in range(nA)}
        is_done = lambda s: s == 0 or s == (nS - 1)
        reward = 0.0 if is_done(s) else -1.0
        # We're stuck in a terminal state
        if is_done(s):
            P[s][UP] = [(1.0, s, reward, True)]
            P[s][RIGHT] = [(1.0, s, reward, True)]
            P[s][DOWN] = [(1.0, s, reward, True)]
            P[s][LEFT] = [(1.0, s, reward, True)]
        # Not a terminal state
        else:
            ns_up = s if y == 0 else s - MAX_X
            ns_right = s if x == (MAX_X - 1) else s + 1
            ns_down = s if y == (MAX_Y - 1) else s + MAX_X
            ns_left = s if x == 0 else s - 1
            P[s][UP] = [(1.0, ns_up, reward, is_done(ns_up))]
            P[s][RIGHT] = [(1.0, ns_right, reward, is_done(ns_right))]
            P[s][DOWN] = [(1.0, ns_down, reward, is_done(ns_down))]
            P[s][LEFT] = [(1.0, ns_left, reward, is_done(ns_left))]
        it.iternext()
    # Initial state distribution is uniform
    isd = np.ones(nS) / nS
    # We expose the model of the environment for educational purposes
    # This should not be used in any model-free learning algorithm
    self.P = P
    super(GridworldEnv, self).__init__(nS, nA, P, isd)
def _render(self, mode='human', close=False):
    """ Renders the current gridworld layout
     For example, a 4x4 grid with the mode="human" looks like:
        T  o  o  o
        o  x  o  o
        o  o  o  o
        o  o  o  T
    where x is your position and T are the two terminal states.
    """
    if close:
        return
    outfile = io.StringIO() if mode == 'ansi' else sys.stdout
    grid = np.arange(self.nS).reshape(self.shape)
    it = np.nditer(grid, flags=['multi_index'])
    while not it.finished:
        s = it.iterindex
        y, x = it.multi_index
        if self.s == s:
            output = " x "
        elif s == 0 or s == self.nS - 1:
            output = " T "
        else:
            output = " o "
        if x == 0:
            output = output.lstrip()
        if x == self.shape[1] - 1:
            output = output.rstrip()
        outfile.write(output)
        if x == self.shape[1] - 1:
            outfile.write("\n")
        it.iternext()# -*- coding: utf-8 -*-
```

Using the gridworld class described above, the code for the grid world exercise is given below.

```
import gridworld

import numpy as np

import random

import copy


class DynamicProgramming:

    def __init__(self, environmentName):

        """

        Class for performing value interation in the given environment

        Parameters

        ----------

        environmentName : string

            Name of gym environment to utilize.

        Returns

        -------

        None.

        """

        self.env = gridworld.GridworldEnv()

        self.theta = 0.0001

        self.discount_factor = 0.9


    def One_Step_LookAhead(self, state, V):

        """

        Function for calculating the value of a given state

        Parameters

        ----------

        state : int

            Current state.

        V : Value array

            list.

        Returns

        -------

        A : List

            Action Value Array.

        """

        A = np.zeros(self.env.nA)

        for a in range(self.env.nA):

            for prob, next_state, reward, done in self.env.P[state][a]:

                A[a] += prob * (reward + self.discount_factor*V[next_state])

        return A
```

```python
    def run(self):
        V = np.zeros(self.env.nS)
        while True:
            delta = 0
            for s in range(self.env.nS):
                A = self.One_Step_LookAhead(s, V)
                best_action_value = np.max(A)
                delta = max(delta, np.abs(best_action_value - V[s]))
                V[s] = best_action_value
            if delta < self.theta:
                break
        # Create a deterministic policy using the optimal value function
        policy = np.zeros([self.env.nS, self.env.nA])
        for s in range(self.env.nS):
            # One step lookahead to find the best action for this state
            A = self.One_Step_LookAhead(s, V)
            best_action = np.argmax(A)
            # Always take the best action
            policy[s, best_action] = 1.0
        return policy, V

if __name__ == '__main__':
    dp = DynamicProgramming('FrozenLake-v0')
    pi, v = dp.run()

    print("Reshaped Grid Policy (0=up, 1=right, 2=down, 3=left):")
    print(np.reshape(np.argmax(pi, axis=1), dp.env.shape))
    print("")

    print("Reshaped Grid Value Function:")
    print(v.reshape(dp.env.shape))
    print("")
```

Works Cited

Bellman, R. (1954). Some Problems in the Theory of Dynamic Programming. *Econometrica*, *22*(1), 37. https://doi.org/10.2307/1909830

Blackwell, D. (1965). Discounted Dynamic Programming. *The Annals of Mathematical Statistics*, *36*(1), 226–235. https://doi.org/10.1214/aoms/1177700285

Greaves, J. (2017). Understanding RL: The Bellman Equations. Retrieved September 20, 2010, from https://joshgreaves.com/reinforcement-learning/understanding-rl-the-bellman-equations/

Howard, R. A. (1958). *Studies in Discrete Dynamic Programming*. 115.

OpenAI. (n.d.). OpenAI Gym. Retrieved from https://gym.openai.com/

PyPi. (n.d.). Retrieved from https://pypi.org/project/gym/0.7.4/

Sutton, R. S., & Barto, A. G. (2014). *Reinforcement Learning: An Introduction*. Cambridge, MA: The MIT Press.

Tsitsiklis, J. N., & Van Roy, B. (2002). On average versus discounted reward temporal-difference learning. *Machine Learning*, *49*(2–3), 179–191.