Dynamic_Programming_Solution

July 7, 2018

1 Mini Project: Dynamic Programming

In this notebook, you will write your own implementations of many classical dynamic programming algorithms.

While we have provided some starter code, you are welcome to erase these hints and write your code from scratch.

1.0.1 Part 0: Explore FrozenLakeEnv

Use the code cell below to create an instance of the FrozenLake environment.

```
In [1]: from frozenlake import FrozenLakeEnv

        env = FrozenLakeEnv()
```

The agent moves through a 4×4 gridworld, with states numbered as follows:

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
```

and the agent has 4 potential actions:

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

Thus, $\mathcal{S}^+ = \{0, 1, \ldots, 15\}$, and $\mathcal{A} = \{0, 1, 2, 3\}$. Verify this by running the code cell below.

```
In [2]: # print the state space and action space
        print(env.observation_space)
        print(env.action_space)

        # print the total number of states and actions
        print(env.nS)
        print(env.nA)
```

```
Discrete(16)
Discrete(4)
16
4
```

Dynamic programming assumes that the agent has full knowledge of the MDP. We have already amended the `frozenlake.py` file to make the one-step dynamics accessible to the agent.

Execute the code cell below to return the one-step dynamics corresponding to a particular state and action. In particular, `env.P[1][0]` returns the the probability of each possible reward and next state, if the agent is in state 1 of the gridworld and decides to go left.

In [3]:
```
env.P[1][0]
```

Out[3]:
```
[(0.3333333333333333, 1, 0.0, False),
 (0.3333333333333333, 0, 0.0, False),
 (0.3333333333333333, 5, 0.0, True)]
```

Each entry takes the form

```
prob, next_state, reward, done
```

where: - `prob` details the conditional probability of the corresponding (`next_state`, `reward`) pair, and - `done` is `True` if the `next_state` is a terminal state, and otherwise `False`.

Thus, we can interpret `env.P[1][0]` as follows:

$$\mathbb{P}(S_{t+1} = s', R_{t+1} = r | S_t = 1, A_t = 0) = \begin{cases} \frac{1}{3} \text{ if } s' = 1, r = 0 \\ \frac{1}{3} \text{ if } s' = 0, r = 0 \\ \frac{1}{3} \text{ if } s' = 5, r = 0 \\ 0 \text{ else} \end{cases}$$

To understand the value of `env.P[1][0]`, note that when you create a FrozenLake environment, it takes as an (optional) argument `is_slippery`, which defaults to `True`.

To see this, change the first line in the notebook from `env = FrozenLakeEnv()` to `env = FrozenLakeEnv(is_slippery=False)`. Then, when you check `env.P[1][0]`, it should look like what you expect (i.e., `env.P[1][0] = [(1.0, 0, 0.0, False)]`).

The default value for the `is_slippery` argument is `True`, and so `env = FrozenLakeEnv()` is equivalent to `env = FrozenLakeEnv(is_slippery=True)`. In the event that `is_slippery=True`, you see that this can result in the agent moving in a direction that it did not intend (where the idea is that the ground is *slippery*, and so the agent can slide to a location other than the one it wanted).

Feel free to change the code cell above to explore how the environment behaves in response to other (state, action) pairs.

Before proceeding to the next part, make sure that you set `is_slippery=True`, so that your implementations below will work with the slippery environment!

1.0.2 Part 1: Iterative Policy Evaluation

In this section, you will write your own implementation of iterative policy evaluation.

Your algorithm should accept four arguments as **input**: - `env`: This is an instance of an OpenAI Gym environment, where `env.P` returns the one-step dynamics. - `policy`: This is a 2D numpy array with `policy.shape[0]` equal to the number of states (`env.nS`), and `policy.shape[1]` equal to the number of actions (`env.nA`). `policy[s][a]` returns the probability that the agent takes action `a` while in state `s` under the policy. - `gamma`: This is the discount rate. It must be a value between 0 and 1, inclusive (default value: `1`). - `theta`: This is a very small positive number that is used to decide if the estimate has sufficiently converged to the true value function (default value: `1e-8`).

The algorithm returns as **output**: - `V`: This is a 1D numpy array with `V.shape[0]` equal to the number of states (`env.nS`). `V[s]` contains the estimated value of state `s` under the input policy.

Please complete the function in the code cell below.

```
In [4]: import numpy as np

        def policy_evaluation(env, policy, gamma=1, theta=1e-8):
            V = np.zeros(env.nS)
            while True:
                delta = 0
                for s in range(env.nS):
                    Vs = 0
                    for a, action_prob in enumerate(policy[s]):
                        for prob, next_state, reward, done in env.P[s][a]:
                            Vs += action_prob * prob * (reward + gamma * V[next_state])
                    delta = max(delta, np.abs(V[s]-Vs))
                    V[s] = Vs
                if delta < theta:
                    break
            return V
```

We will evaluate the equiprobable random policy π, where $\pi(a|s) = \frac{1}{|\mathcal{A}(s)|}$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$.

Use the code cell below to specify this policy in the variable `random_policy`.

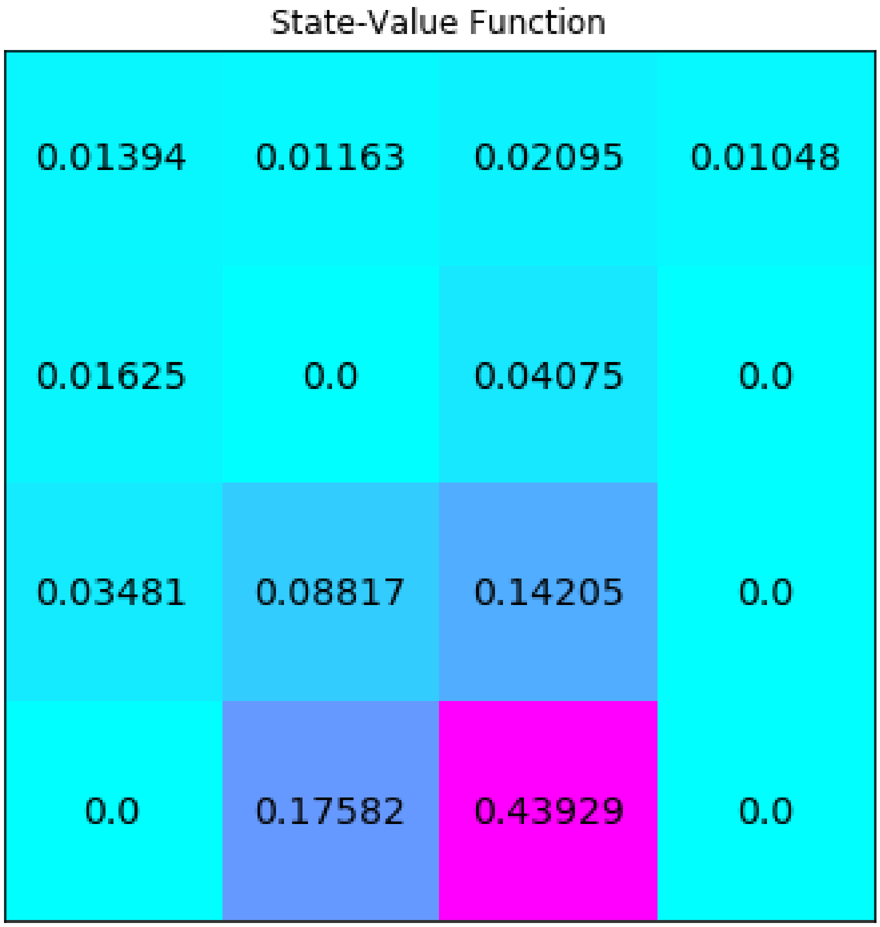
```
In [5]: random_policy = np.ones([env.nS, env.nA]) / env.nA
```

Run the next code cell to evaluate the equiprobable random policy and visualize the output. The state-value function has been reshaped to match the shape of the gridworld.

```
In [6]: from plot_utils import plot_values

        # evaluate the policy
        V = policy_evaluation(env, random_policy)

        plot_values(V)
```



Run the code cell below to test your function. If the code cell returns **PASSED**, then you have implemented the function correctly!

Note: In order to ensure accurate results, make sure that your `policy_evaluation` function satisfies the requirements outlined above (with four inputs, a single output, and with the default values of the input arguments unchanged).

In [7]:
```
import check_test

check_test.run_check('policy_evaluation_check', policy_evaluation)
```

PASSED

1.0.3 Part 2: Obtain q_π from v_π

In this section, you will write a function that takes the state-value function estimate as input, along with some state $s \in \mathcal{S}$. It returns the **row in the action-value function** corresponding to the input state $s \in \mathcal{S}$. That is, your function should accept as input both v_π and s, and return $q_\pi(s, a)$ for all $a \in \mathcal{A}(s)$.

Your algorithm should accept four arguments as **input**: - `env`: This is an instance of an OpenAI Gym environment, where `env.P` returns the one-step dynamics. - `V`: This is a 1D numpy array with `V.shape[0]` equal to the number of states (`env.nS`). `V[s]` contains the estimated value of state `s`. - `s`: This is an integer corresponding to a state in the environment. It should be a value between `0` and `(env.nS)-1`, inclusive. - `gamma`: This is the discount rate. It must be a value between 0 and 1, inclusive (default value: `1`).

The algorithm returns as **output**: - `q`: This is a 1D numpy array with `q.shape[0]` equal to the number of actions (`env.nA`). `q[a]` contains the (estimated) value of state `s` and action `a`.

Please complete the function in the code cell below.

```
In [8]: def q_from_v(env, V, s, gamma=1):
            q = np.zeros(env.nA)
            for a in range(env.nA):
                for prob, next_state, reward, done in env.P[s][a]:
                    q[a] += prob * (reward + gamma * V[next_state])
            return q
```

Run the code cell below to print the action-value function corresponding to the above state-value function.

```
In [9]: Q = np.zeros([env.nS, env.nA])
        for s in range(env.nS):
            Q[s] = q_from_v(env, V, s)
        print("Action-Value Function:")
        print(Q)
```

```
Action-Value Function:
[[ 0.0147094   0.01393978  0.01393978  0.01317015]
 [ 0.00852356  0.01163091  0.0108613   0.01550788]
 [ 0.02444514  0.02095298  0.02406033  0.01435346]
 [ 0.01047649  0.01047649  0.00698432  0.01396865]
 [ 0.02166487  0.01701828  0.01624865  0.01006281]
 [ 0.          0.          0.          0.        ]
 [ 0.05433538  0.04735105  0.05433538  0.00698432]
 [ 0.          0.          0.          0.        ]
 [ 0.01701828  0.04099204  0.03480619  0.04640826]
 [ 0.07020885  0.11755991  0.10595784  0.05895312]
 [ 0.18940421  0.17582037  0.16001424  0.04297382]
 [ 0.          0.          0.          0.        ]
 [ 0.          0.          0.          0.        ]
 [ 0.08799677  0.20503718  0.23442716  0.17582037]
 [ 0.25238823  0.53837051  0.52711478  0.43929118]
 [ 0.          0.          0.          0.        ]]
```

Run the code cell below to test your function. If the code cell returns **PASSED**, then you have implemented the function correctly!

Note: In order to ensure accurate results, make sure that the `q_from_v` function satisfies the requirements outlined above (with four inputs, a single output, and with the default values of the input arguments unchanged).

In [10]: check_test.run_check('q_from_v_check', q_from_v)

PASSED

1.0.4 Part 3: Policy Improvement

In this section, you will write your own implementation of policy improvement.

Your algorithm should accept three arguments as **input**: - `env`: This is an instance of an OpenAI Gym environment, where `env.P` returns the one-step dynamics. - `V`: This is a 1D numpy array with `V.shape[0]` equal to the number of states (`env.nS`). `V[s]` contains the estimated value of state `s`. - `gamma`: This is the discount rate. It must be a value between 0 and 1, inclusive (default value: `1`).

The algorithm returns as **output**: - `policy`: This is a 2D numpy array with `policy.shape[0]` equal to the number of states (`env.nS`), and `policy.shape[1]` equal to the number of actions (`env.nA`). `policy[s][a]` returns the probability that the agent takes action `a` while in state `s` under the policy.

Please complete the function in the code cell below. You are encouraged to use the `q_from_v` function you implemented above.

```
In [11]: def policy_improvement(env, V, gamma=1):
             policy = np.zeros([env.nS, env.nA]) / env.nA
             for s in range(env.nS):
                 q = q_from_v(env, V, s, gamma)

                 # OPTION 1: construct a deterministic policy
                 # policy[s][np.argmax(q)] = 1

                 # OPTION 2: construct a stochastic policy that puts equal probability on maximi
                 best_a = np.argwhere(q==np.max(q)).flatten()
                 policy[s] = np.sum([np.eye(env.nA)[i] for i in best_a], axis=0)/len(best_a)

             return policy
```

Run the code cell below to test your function. If the code cell returns **PASSED**, then you have implemented the function correctly!

Note: In order to ensure accurate results, make sure that the `policy_improvement` function satisfies the requirements outlined above (with three inputs, a single output, and with the default values of the input arguments unchanged).

Before moving on to the next part of the notebook, you are strongly encouraged to check out the solution in **Dynamic_Programming_Solution.ipynb**. There are many correct ways to approach this function!

```
In [12]: check_test.run_check('policy_improvement_check', policy_improvement)
```

PASSED

1.0.5 Part 4: Policy Iteration

In this section, you will write your own implementation of policy iteration. The algorithm returns the optimal policy, along with its corresponding state-value function.

Your algorithm should accept three arguments as **input**: - `env`: This is an instance of an OpenAI Gym environment, where `env.P` returns the one-step dynamics. - `gamma`: This is the discount rate. It must be a value between 0 and 1, inclusive (default value: `1`). - `theta`: This is a very small positive number that is used to decide if the policy evaluation step has sufficiently converged to the true value function (default value: `1e-8`).

The algorithm returns as **output**: - `policy`: This is a 2D numpy array with `policy.shape[0]` equal to the number of states (`env.nS`), and `policy.shape[1]` equal to the number of actions (`env.nA`). `policy[s][a]` returns the probability that the agent takes action `a` while in state `s` under the policy. - `V`: This is a 1D numpy array with `V.shape[0]` equal to the number of states (`env.nS`). `V[s]` contains the estimated value of state `s`.

Please complete the function in the code cell below. You are strongly encouraged to use the `policy_evaluation` and `policy_improvement` functions you implemented above.

```
In [13]: import copy

         def policy_iteration(env, gamma=1, theta=1e-8):
             policy = np.ones([env.nS, env.nA]) / env.nA
             while True:
                 V = policy_evaluation(env, policy, gamma, theta)
                 new_policy = policy_improvement(env, V)

                 # OPTION 1: stop if the policy is unchanged after an improvement step
                 if (new_policy == policy).all():
                     break;

                 # OPTION 2: stop if the value function estimates for successive policies has co
                 # if np.max(abs(policy_evaluation(env, policy) - policy_evaluation(env, new_pol
                 #     break;

                 policy = copy.copy(new_policy)
             return policy, V
```

Run the next code cell to solve the MDP and visualize the output. The optimal state-value function has been reshaped to match the shape of the gridworld.

Compare the optimal state-value function to the state-value function from Part 1 of this notebook. *Is the optimal state-value function consistently greater than or equal to the state-value function for the equiprobable random policy?*

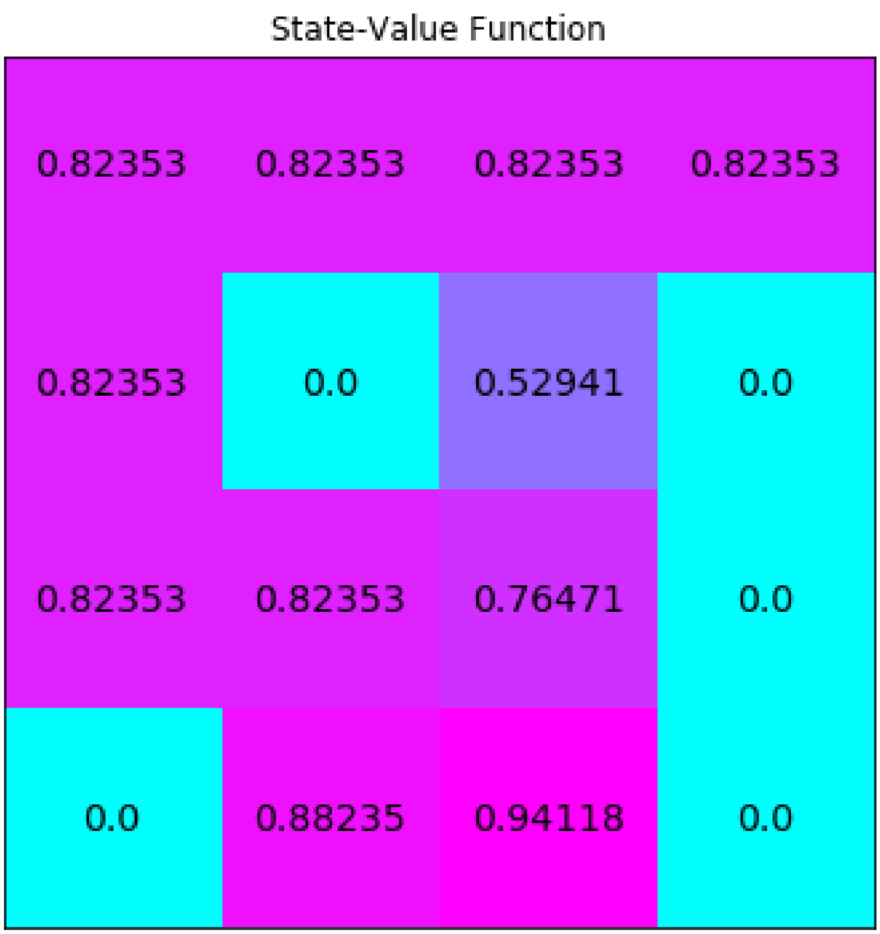
```
In [14]: # obtain the optimal policy and optimal state-value function
         policy_pi, V_pi = policy_iteration(env)

         # print the optimal policy
         print("\nOptimal Policy (LEFT = 0, DOWN = 1, RIGHT = 2, UP = 3):")
         print(policy_pi,"\n")

         plot_values(V_pi)
```

```
Optimal Policy (LEFT = 0, DOWN = 1, RIGHT = 2, UP = 3):
```

```
[[ 1.    0.    0.    0.  ]
 [ 0.    0.    0.    1.  ]
 [ 0.    0.    0.    1.  ]
 [ 0.    0.    0.    1.  ]
 [ 1.    0.    0.    0.  ]
 [ 0.25  0.25  0.25  0.25]
 [ 0.5   0.    0.5   0.  ]
 [ 0.25  0.25  0.25  0.25]
 [ 0.    0.    0.    1.  ]
 [ 0.    1.    0.    0.  ]
 [ 1.    0.    0.    0.  ]
 [ 0.25  0.25  0.25  0.25]
 [ 0.25  0.25  0.25  0.25]
 [ 0.    0.    1.    0.  ]
 [ 0.    1.    0.    0.  ]
 [ 0.25  0.25  0.25  0.25]]
```



Run the code cell below to test your function. If the code cell returns **PASSED**, then you have implemented the function correctly!

Note: In order to ensure accurate results, make sure that the `policy_iteration` function satisfies the requirements outlined above (with three inputs, two outputs, and with the default values of the input arguments unchanged).

```
In [15]: check_test.run_check('policy_iteration_check', policy_iteration)
```

PASSED

1.0.6 Part 5: Truncated Policy Iteration

In this section, you will write your own implementation of truncated policy iteration.

You will begin by implementing truncated policy evaluation. Your algorithm should accept five arguments as **input**: - `env`: This is an instance of an OpenAI Gym environment, where `env.P` returns the one-step dynamics. - `policy`: This is a 2D numpy array with `policy.shape[0]` equal to the number of states (`env.nS`), and `policy.shape[1]` equal to the number of actions (`env.nA`). `policy[s][a]` returns the probability that the agent takes action `a` while in state `s` under the policy. - `V`: This is a 1D numpy array with `V.shape[0]` equal to the number of states (`env.nS`). `V[s]` contains the estimated value of state `s`. - `max_it`: This is a positive integer that corresponds to the number of sweeps through the state space (default value: `1`). - `gamma`: This is the discount rate. It must be a value between 0 and 1, inclusive (default value: `1`).

The algorithm returns as **output**: - `V`: This is a 1D numpy array with `V.shape[0]` equal to the number of states (`env.nS`). `V[s]` contains the estimated value of state `s`.

Please complete the function in the code cell below.

```
In [16]: def truncated_policy_evaluation(env, policy, V, max_it=1, gamma=1):
             num_it=0
             while num_it < max_it:
                 for s in range(env.nS):
                     v = 0
                     q = q_from_v(env, V, s, gamma)
                     for a, action_prob in enumerate(policy[s]):
                         v += action_prob * q[a]
                     V[s] = v
                 num_it += 1
             return V
```

Next, you will implement truncated policy iteration. Your algorithm should accept five arguments as **input**: - `env`: This is an instance of an OpenAI Gym environment, where `env.P` returns the one-step dynamics. - `max_it`: This is a positive integer that corresponds to the number of sweeps through the state space (default value: `1`). - `gamma`: This is the discount rate. It must be a value between 0 and 1, inclusive (default value: `1`). - `theta`: This is a very small positive number that is used for the stopping criterion (default value: `1e-8`).

The algorithm returns as **output**: - `policy`: This is a 2D numpy array with `policy.shape[0]` equal to the number of states (`env.nS`), and `policy.shape[1]` equal to the number of actions (`env.nA`). `policy[s][a]` returns the probability that the agent takes action `a` while in state `s` under the policy. - `V`: This is a 1D numpy array with `V.shape[0]` equal to the number of states (`env.nS`). `V[s]` contains the estimated value of state `s`.

Please complete the function in the code cell below.

```
In [17]: def truncated_policy_iteration(env, max_it=1, gamma=1, theta=1e-8):
             V = np.zeros(env.nS)
             policy = np.zeros([env.nS, env.nA]) / env.nA
             while True:
                 policy = policy_improvement(env, V)
                 old_V = copy.copy(V)
                 V = truncated_policy_evaluation(env, policy, V, max_it, gamma)
                 if max(abs(V-old_V)) < theta:
                     break;
             return policy, V
```

Run the next code cell to solve the MDP and visualize the output. The state-value function has been reshaped to match the shape of the gridworld.

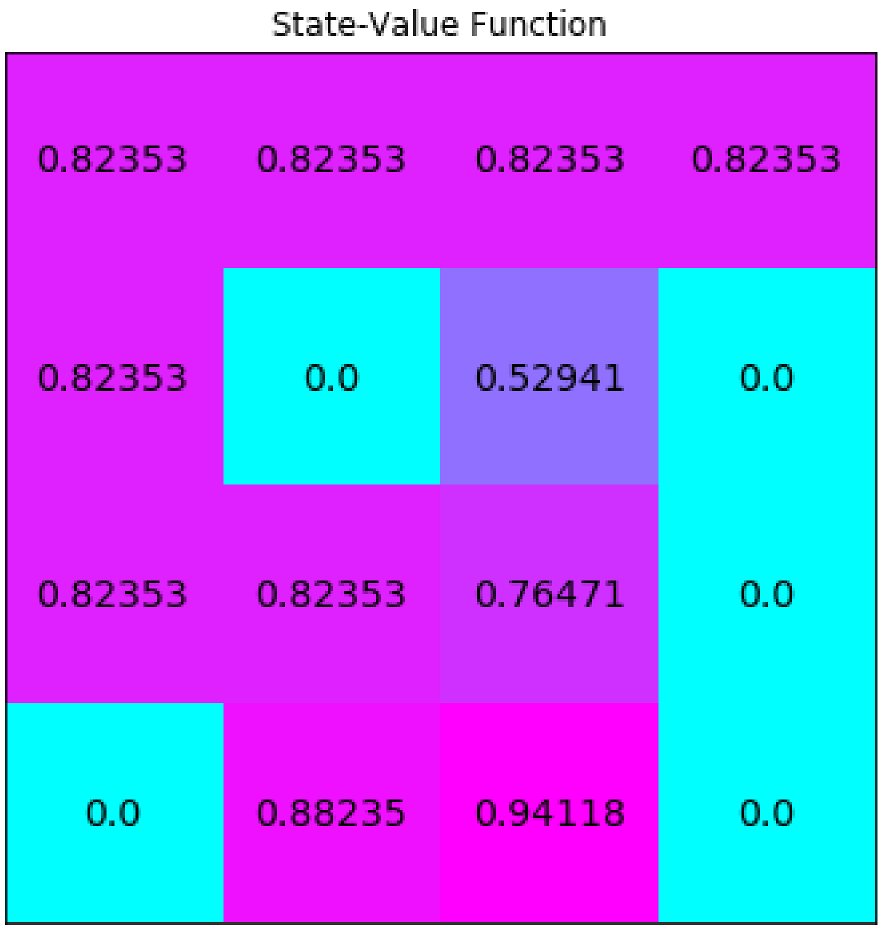
Play with the value of the `max_it` argument. Do you always end with the optimal state-value function?

```
In [18]: policy_tpi, V_tpi = truncated_policy_iteration(env, max_it=2)

         # print the optimal policy
         print("\nOptimal Policy (LEFT = 0, DOWN = 1, RIGHT = 2, UP = 3):")
         print(policy_tpi,"\n")

         # plot the optimal state-value function
         plot_values(V_tpi)
```

```
Optimal Policy (LEFT = 0, DOWN = 1, RIGHT = 2, UP = 3):
[[ 1.    0.    0.    0.  ]
 [ 0.    0.    0.    1.  ]
 [ 0.    0.    0.    1.  ]
 [ 0.    0.    0.    1.  ]
 [ 1.    0.    0.    0.  ]
 [ 0.25  0.25  0.25  0.25]
 [ 0.5   0.    0.5   0.  ]
 [ 0.25  0.25  0.25  0.25]
 [ 0.    0.    0.    1.  ]
 [ 0.    1.    0.    0.  ]
 [ 1.    0.    0.    0.  ]
 [ 0.25  0.25  0.25  0.25]
 [ 0.25  0.25  0.25  0.25]
 [ 0.    0.    1.    0.  ]
 [ 0.    1.    0.    0.  ]
 [ 0.25  0.25  0.25  0.25]]
```



Run the code cell below to test your function. If the code cell returns **PASSED**, then you have implemented the function correctly!

Note: In order to ensure accurate results, make sure that the `truncated_policy_iteration` function satisfies the requirements outlined above (with four inputs, two outputs, and with the default values of the input arguments unchanged).

```
In [19]: check_test.run_check('truncated_policy_iteration_check', truncated_policy_iteration)
```

PASSED

1.0.7 Part 6: Value Iteration

In this section, you will write your own implementation of value iteration.

Your algorithm should accept three arguments as input: - `env`: This is an instance of an OpenAI Gym environment, where `env.P` returns the one-step dynamics. - `gamma`: This is the discount rate. It must be a value between 0 and 1, inclusive (default value: `1`). - `theta`: This is a very small positive number that is used for the stopping criterion (default value: `1e-8`).

The algorithm returns as **output**: - `policy`: This is a 2D numpy array with `policy.shape[0]` equal to the number of states (`env.nS`), and `policy.shape[1]` equal to the number of actions (`env.nA`). `policy[s][a]` returns the probability that the agent takes action `a` while in state `s` under the policy. - `V`: This is a 1D numpy array with `V.shape[0]` equal to the number of states (`env.nS`). `V[s]` contains the estimated value of state `s`.

```
In [20]: def value_iteration(env, gamma=1, theta=1e-8):
             V = np.zeros(env.nS)
             while True:
                 delta = 0
                 for s in range(env.nS):
                     v = V[s]
                     V[s] = max(q_from_v(env, V, s, gamma))
                     delta = max(delta,abs(V[s]-v))
                 if delta < theta:
                     break
             policy = policy_improvement(env, V, gamma)
             return policy, V
```

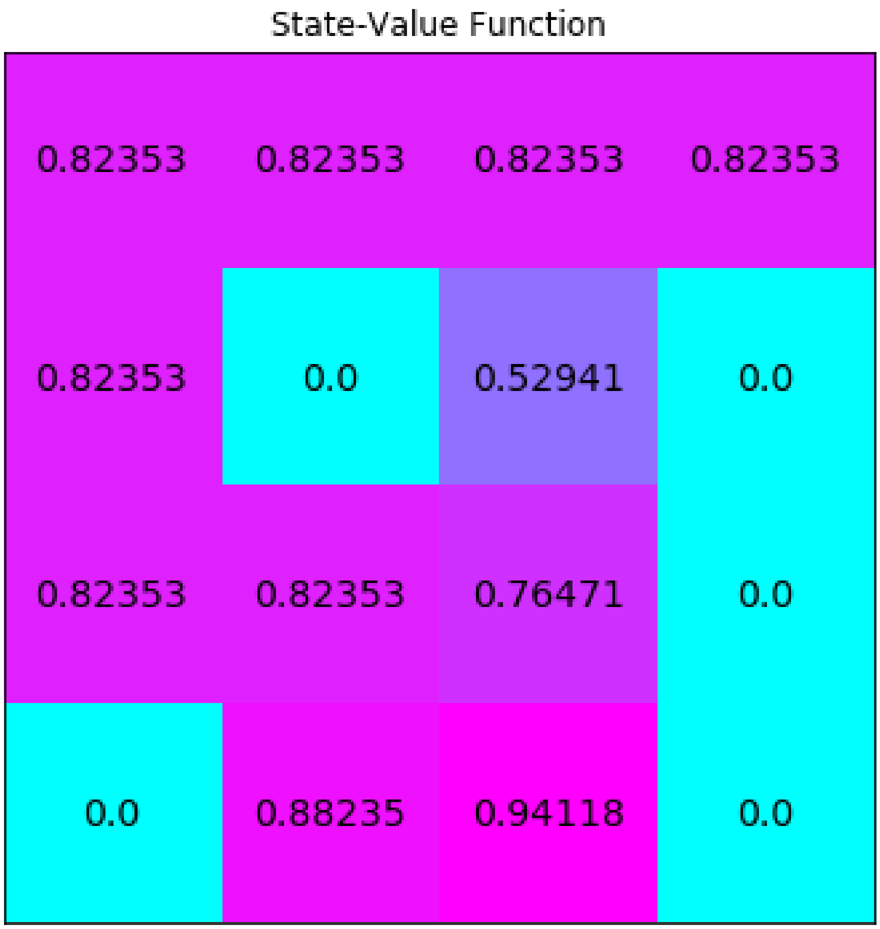
Use the next code cell to solve the MDP and visualize the output. The state-value function has been reshaped to match the shape of the gridworld.

```
In [21]: policy_vi, V_vi = value_iteration(env)

         # print the optimal policy
         print("\nOptimal Policy (LEFT = 0, DOWN = 1, RIGHT = 2, UP = 3):")
         print(policy_vi,"\n")

         # plot the optimal state-value function
         plot_values(V_vi)
```

```
Optimal Policy (LEFT = 0, DOWN = 1, RIGHT = 2, UP = 3):
[[ 1.    0.    0.    0.  ]
 [ 0.    0.    0.    1.  ]
 [ 0.    0.    0.    1.  ]
 [ 0.    0.    0.    1.  ]
 [ 1.    0.    0.    0.  ]
 [ 0.25  0.25  0.25  0.25]
 [ 0.5   0.    0.5   0.  ]
 [ 0.25  0.25  0.25  0.25]
 [ 0.    0.    0.    1.  ]
 [ 0.    1.    0.    0.  ]
 [ 1.    0.    0.    0.  ]
 [ 0.25  0.25  0.25  0.25]
 [ 0.25  0.25  0.25  0.25]
 [ 0.    0.    1.    0.  ]
 [ 0.    1.    0.    0.  ]
 [ 0.25  0.25  0.25  0.25]]
```



Run the code cell below to test your function. If the code cell returns **PASSED**, then you have implemented the function correctly!

Note: In order to ensure accurate results, make sure that the `value_iteration` function satisfies the requirements outlined above (with three inputs, two outputs, and with the default values of the input arguments unchanged).

```
In [22]: check_test.run_check('value_iteration_check', value_iteration)
```

PASSED