

Reinforcement Learning Programming Assignment #1

John Vitz

1. Implementation of basic function (10%)

In this section, you need to read through `code_base.py` and `get_args.py`. Finish your code in `code_base.py` between the pound sign lines denoted with "Your Code" in each function. Test your code on FrozenLake-v1 8*8 deterministic environment. Suggestions:

- Please fully understand the FrozenLake game, analyze the structure of the programs, especially the input and output of each function according to the comments, and read the notes under "Your Code" carefully before implementation. The tutorial for FrozenLake is [here](#).
- Please do not change the default settings in `get_args.py` for this section.
- Please use `np.max` and `np.argmax` if you need to return or find the maximum in a numpy array.
- All functions needed for policy iteration and value iteration are defined and technically you do not need to create new ones.

(a) Implement `policy_evaluation` and `policy_improvement`, and then use these two functions to finish `policy_iteration`. (Hint 1: Please be careful about the definition of iteration. The iteration does not include the steps of policy evaluation.) (Hint 2: Please update the value function using the synchronous update method (refer to specific hints in the corresponding code sections)).

Terminal Input:

`python code_base.py`

```
---- Policy Iteration----

There are 15 iterations in policy iteration.
policy: [['D' 'D' 'D' 'D' 'D' 'D' 'D' 'D']
 ['D' 'D' 'D' 'R' 'D' 'D' 'D' 'D']
 ['D' 'D' 'D' 'L' 'D' 'R' 'D' 'D']
 ['R' 'R' 'R' 'R' 'D' 'L' 'D' 'D']
 ['R' 'R' 'U' 'L' 'D' 'D' 'R' 'D']
 ['D' 'L' 'L' 'R' 'R' 'D' 'L' 'D']
 ['D' 'L' 'R' 'U' 'L' 'D' 'L' 'D']
 ['R' 'R' 'U' 'L' 'R' 'R' 'R' 'L']]
Total running time is: 1.0706861019134521 average number of iteration is: 15.0
Episode reward: 1.0
```

Continue on the next page.

(b) Implement value_iteration.

Terminal Input:

python code_base.py -method value_iteration

```
---- Value Iteration----

There are 15 iterations in value iteration.
policy: [['D' 'D' 'D' 'D' 'D' 'D' 'D' 'D']
 ['D' 'D' 'D' 'R' 'D' 'D' 'D' 'D']
 ['D' 'D' 'D' 'L' 'D' 'R' 'D' 'D']
 ['R' 'R' 'R' 'R' 'D' 'L' 'D' 'D']
 ['R' 'R' 'U' 'L' 'D' 'D' 'R' 'D']
 ['D' 'L' 'L' 'R' 'R' 'D' 'L' 'D']
 ['D' 'L' 'R' 'U' 'L' 'D' 'L' 'D']
 ['R' 'R' 'U' 'L' 'R' 'R' 'R' 'L']]
Total running time is: 1.061493158340454 average number of iteration is: 15.0
Episode reward: 1.0
```

2. Evaluate the performance (20%)

In this section, you will evaluate the performance of the methods you just finished in Section I. Use parameter: method in get_args.py to select to run policy_iteration or value_iteration. (Hint: Iteration refers to the process of repeatedly updating the policy or value function. The steps in policy evaluation or policy improvement are not included.)

(a) Set seeds = 1 in get_args.py and take the screenshots of all the output results after running policy_iteration and value_iteration. (The screenshots should include the method you are using, the policy generated, the total running time and the episode reward.)

Terminal Input:

python code_base.py -method policy_iteration -seeds 1

```
---- Policy Iteration----

There are 15 iterations in policy iteration.
policy: [['D' 'D' 'D' 'D' 'D' 'D' 'D' 'D']
 ['D' 'D' 'D' 'R' 'D' 'D' 'D' 'D']
 ['D' 'D' 'D' 'L' 'D' 'R' 'D' 'D']
 ['R' 'R' 'R' 'R' 'D' 'L' 'D' 'D']
 ['R' 'R' 'U' 'L' 'D' 'D' 'R' 'D']
 ['D' 'L' 'L' 'R' 'R' 'D' 'L' 'D']
 ['D' 'L' 'R' 'U' 'L' 'D' 'L' 'D']
 ['R' 'R' 'U' 'L' 'R' 'R' 'R' 'L']]
Total running time is: 1.0346684455871582 average number of iteration is: 15.0
Episode reward: 1.0
```

Terminal Input:

```
python code_base.py -method value_iteration -seeds 1
```

```
---- Value Iteration----

There are 15 iterations in value iteration.
policy: [['D' 'D' 'D' 'D' 'D' 'D' 'D' 'D']
['D' 'D' 'D' 'R' 'D' 'D' 'D' 'D']
['D' 'D' 'D' 'L' 'D' 'R' 'D' 'D']
['R' 'R' 'R' 'R' 'D' 'L' 'D' 'D']
['R' 'R' 'U' 'L' 'D' 'D' 'R' 'D']
['D' 'L' 'L' 'R' 'R' 'D' 'L' 'D']
['D' 'L' 'R' 'U' 'L' 'D' 'L' 'D']
['R' 'R' 'U' 'L' 'R' 'R' 'R' 'L']]
Total running time is: 0.850776195526123 average number of iteration is: 15.0
Episode reward: 1.0
```

(b) Please modify the parameter seeds in get_args and run each of policy_iteration and value_iteration 50 times and take the screenshots of the total running time starting with the line: "Total running time is".

Terminal Input:

```
python code_base.py -method policy_iteration -seeds 50
```

```
Total running time is: 13.23658537864685 average number of iteration is: 14.42
Episode reward: 1.0
```

Terminal Input:

```
python code_base.py -method value_iteration -seeds 50
```

```
Total running time is: 13.210450172424316 average number of iteration is: 15.0
Episode reward: 1.0
```

(c) Question: What is the time complexity of policy iteration in one iteration? Please provide your answer in terms of $|S|$ (the length of the state vector) and $|A|$ (the length of the action vector). (Hint: "in one iteration" means we only consider the running time for just one run of policy evaluation and policy improvement function. You may also refer to the annotation in the codebase to calculate the running time.)

For policy evaluation, the following loop structure: while $\rightarrow nS \rightarrow T$. So the time complexity is $O(k_p * nS * T)$ where k_p is the number of iterations that the while loop takes to converge, nS is the number of states (length of state vector), and T is the number of transitions at the given state-action pair. The number of actions here is 1, so it is not included in time complexity.

For policy improvement, the following loop structure: $nS \rightarrow nA \rightarrow T$. So the time complexity is $O(nS * nA * T)$ where nS is the number of states (length of state vector), nA is the number of actions possible at the given state, and T is the number of transitions at the given state-action pair. There is no convergence criterion here, so there is no k_p in the time complexity.

Policy iteration has to combine these steps together sequentially, so that means that its time complexity is dependent on both of the previously described functions.

Single Iteration Policy evaluation: $O(k_p * nS * T)$, Policy Improvement: $O(nS * nA * T)$

Single Iteration Policy Iteration = Policy evaluation + Policy Improvement

Single Iteration Policy Iteration = $O(k_p * nS * T) + O(nS * nA * T) =$

Single Iteration Policy Iteration = $O((k_p * nS * T) + (nS * nA * T))$

Single Iteration Policy Iteration (vector notation) = $O((k_p * |S| * T) + (|S| * |A| * T))$

Above is my answer for a Single Iteration of Policy Iteration. If multiple iterations were used, there would have to be an additional multiplicative term to account for the number of iterations it takes until the new policy is the same as the previous policy, denoted by p .

Multiple Iteration = $O(p * ((k_p * |S| * T) + (|S| * |A| * T)))$

(d) Question: What is the time complexity of value iteration in one iteration? Please provide your answer in terms of $|S|$ (the length of the state vector) and $|A|$ (the length of the action vector). (Hint: "in one iteration" means we only consider the running time for just one time's value updating. You may also refer to the annotation in the codebase to calculate the running time.)

For value iteration, the following loop structure: $\text{while} \rightarrow nS \rightarrow nA \rightarrow T$. So the time complexity is $O(k_v * nS * nA * T)$ where k_v is the number of iterations that the while loop takes to converge, nS is the number of states (length of state vector), nA is the number of actions per state (length of action vector), and T is the number of transitions in a given state-action pair.

Value Iteration = $O(k_v * nS * nA * T)$

Value Iteration (vector notation) = $O(k_v * |S| * |A| * T)$

But what is asked for is the time complexity in **one iteration**, so this means the outermost loop will not apply. The time complexity then drops the k_v , because there is only a single iteration of the outermost loop (the while loop).

Single Iteration of Value Iteration = $O(|S| * |A| * T)$

Continue on the next page.

e) Question: Based on your answer for part(c) and part(d), theoretically which algorithm is faster in one iteration? Aside from the running time in a single iteration, what other factors could influence the total running time?

$$\text{Single Iteration Policy Iteration} = O((k_p * |S| * T) + (|S| * |A| * T))$$

versus

$$\text{Single Iteration of Value Iteration} = O(|S| * |A| * T)$$

I think that theoretically a single iteration of Value Iteration is faster than a single iteration of Policy Iteration. This is because Policy Iteration includes the same components of time complexity that Value iteration does in $O(|S| * |A| * T)$ with its use of Policy Improvement, but Policy Iteration also has to include Policy Evaluation which adds the term $O(k_p * |S| * T)$ to the time complexity. This means that Policy Iteration has a single iteration time complexity that is higher than Value Iteration.

I think that the biggest factor that could affect run-time for multiple iterations / until termination of each iteration method is reached is the convergence rate for each method. I will consider now not just Single Iteration.

$$\text{Multiple Iteration Policy Iteration} = O(p * ((k_p * |S| * T) + (|S| * |A| * T)))$$

versus

$$\text{Multiple Iteration Value Iteration} = O(k_v * |S| * |A| * T)$$

In this case, now there are two convergence components in Policy Iteration, and there is one in Value Iteration. This means that the time complexity of Policy Iteration's convergence can be determined by p and k_p , while the time complexity of Value Iteration's convergence can be determined by k_v . If $p * k_p$ is much less than k_v , this could mean that Value Iteration takes longer than Policy Iteration in practice despite the seemingly large advantage that Value Iteration has over Policy Iteration purely based on big O notation. So, the convergence rate for each method is definitely a large factor that can influence the total running time.

The size of the state and action space significantly affect both Single and Multiple Iteration methods, as they both directly multiplicatively increase the number of computations per iteration which may also lead to significant increase in the time to converge for both methods.

The convergence criterion / epsilon, also has a massive effect on the number of iterations to converge, as changing how strict it is directly affects the number of iterations that occur within the inner loops of in both the Policy Evaluation component of Policy Iteration and Value Iteration before early stoppage.

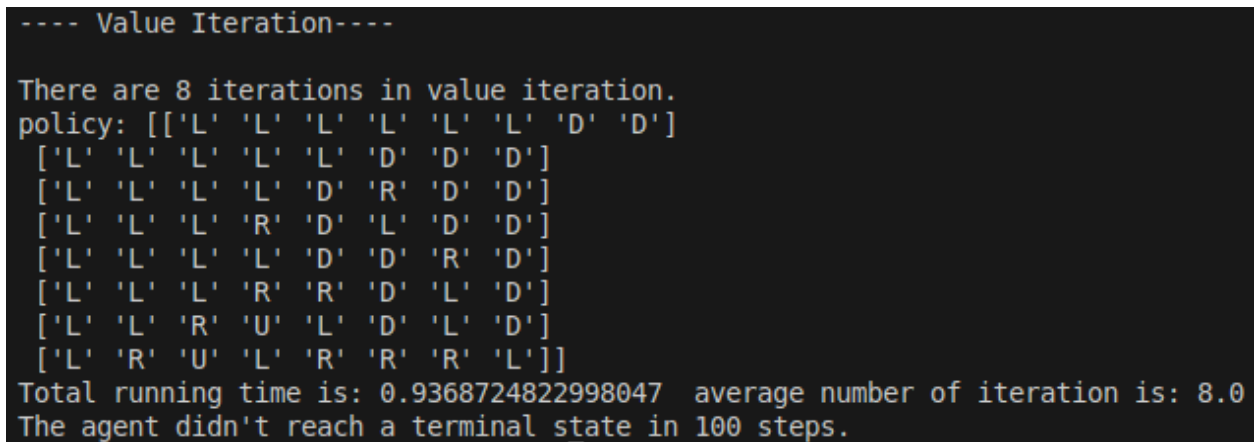
3. Effect of parameter epsilon ϵ (20%)

In this section, you will use value iteration to further understand the effect of epsilon ϵ . The definition of ϵ is introduced in the class. The setting of ϵ is in the 14th line of `get_args.py` named as ϵ , and you can change default to the desired value.

(a) Reset the parameter seeds as 1 in `get_args.py`. Use the value iteration method by setting the method as `value_iteration`. Set the value of ϵ as 0.5. Rerun the program and take the screenshot of all the output results. (The screenshots should include the method you are using, the policy generated, the total running time and the episode reward.)

Terminal Input:

```
python code_base.py -method value_iteration -seeds 1 -epsilon 0.5
```



```
---- Value Iteration----

There are 8 iterations in value iteration.
policy: [['L' 'L' 'L' 'L' 'L' 'L' 'D' 'D']
['L' 'L' 'L' 'L' 'L' 'D' 'D' 'D']
['L' 'L' 'L' 'L' 'D' 'R' 'D' 'D']
['L' 'L' 'L' 'R' 'D' 'L' 'D' 'D']
['L' 'L' 'L' 'L' 'D' 'D' 'R' 'D']
['L' 'L' 'L' 'R' 'R' 'D' 'L' 'D']
['L' 'L' 'R' 'U' 'L' 'D' 'L' 'D']
['L' 'R' 'U' 'L' 'R' 'R' 'R' 'L']]
Total running time is: 0.9368724822998047 average number of iteration is: 8.0
The agent didn't reach a terminal state in 100 steps.
```

(b) Question: How many iterations does the value iteration method need to generate the policy? Is the policy the optimal policy?

The value iteration method needs 8 iterations to generate the policy shown above, but this policy is not the optimal policy because there is no termination of the episode / no reward. An optimal policy would mean that the agent reaches the termination state and maximizes reward, which does not happen with $\epsilon = 0.5$.

(c) Question: How many iterations does the value iteration method need to generate the policy from part II.(a)? Is it less or more than the iterations you get when setting $\epsilon = 0.5$?

The value iteration method with $\epsilon = 0.5$ doesn't generate the policy from part II.(a). The value iteration method with $\epsilon = 0.5$ generates a different policy and the episode doesn't reach termination state, while the policy from part II.(a) does. The method doesn't work with $\epsilon = 0.5$ in a way that is comparable with part II.(a); no valid policy can be generated from Value Iteration with $\epsilon = 0.5$.

(d) Question: Based on the comparison between part (b) and part (c), what will happen if ϵ is decreased? Conversely, what will happen if ϵ is increased? Can you explain the reason? (Hint: Based on the definition from lecture that the condition of convergence is the maximum difference $\|V_{k+1} - V_k\|_{\infty} \leq \epsilon$, how does ϵ affect the number of iteration and the optimal policy?)

The value of ϵ plays a pivotal role in determining the convergence rate and the optimality of the policy generated by value iteration.

When ϵ is increased:

Increasing ϵ means that the condition for convergence becomes less strict; this means that the maximum allowable difference between successive value functions becomes larger. This will cause the value iteration algorithm to terminate earlier, which means that it will perform fewer iterations/fewer updates to the policy. This speeds up the process of value iteration, but it can risk failure to converge to an optimal policy as the full amount of significant updates required to generate the optimal policy can be undershot.

When ϵ is decreased:

Decreasing ϵ means that the condition for convergence becomes more strict; this means that the maximum allowable difference between successive value functions becomes smaller. This will cause the value iteration algorithm to terminate later, which means that it will perform more iterations/more updates to the policy. This slows down the process of value iteration, but it can risk overshooting the amount of iterations necessary to generate the optimal policy as the full amount of significant updates required to generate the optimal policy can be overshoot.

Summary:

Increasing ϵ speeds up the convergence of the value iteration process, but can lead to a failure to converge to an optimal policy due to a lack of ability to capture significant updates (early termination). Decreased ϵ slows down the convergence of the value iteration process, which increases the chance of converging to an optimal policy but also increases the chance of over-shooting the point where updates become insignificant, which means that there can be an excess in iterations and therefore a cost in unnecessary time and computation (late termination).

So, ϵ greatly affects the number of iterations to algorithm termination and convergence to the optimal policy. If ϵ is too high, we may fail to converge to an optimal policy, and if ϵ is too low, we may make updates far beyond what is necessary to converge to an optimal policy. Ideally, if ϵ is balanced correctly, we will reach an optimal policy upon convergence without greatly overshooting the point where updates become insignificant.

Continue on the next page.

4. Effect of parameter gamma γ (30%)

In this section, you will still use the value iteration to further understand the effect of gamma γ . The definition of γ is introduced in the class. The setting of γ is in the 16th line of `get_args.py`, and you can change default to modify the value of the parameters. Remember to recover ϵ to $1e-3$. You can print out the value function of the states to find some clues about the answers.

(a) Set $\gamma = 0$, and take the screenshot of all the output results from your code.
(The screenshots should include the method you are using, the policy generated, the total running time and the episode reward.)

Terminal Input:

```
python code_base.py -method value_iteration -seeds 1 -epsilon 0.001 -gamma 0
```

```
---- Value Iteration----

There are 2 iterations in value iteration.
policy: [['L' 'L' 'L' 'L' 'L' 'L' 'L' 'L']
['L' 'L' 'L' 'L' 'L' 'L' 'L' 'L']
['L' 'L' 'L' 'L' 'L' 'L' 'L' 'L']
['L' 'L' 'L' 'L' 'L' 'L' 'L' 'L']
['L' 'L' 'L' 'L' 'L' 'L' 'L' 'L']
['L' 'L' 'L' 'L' 'L' 'L' 'L' 'D']
['L' 'L' 'L' 'L' 'L' 'L' 'R' 'L']]
Total running time is: 0.8482604026794434 average number of iteration is: 2.0
The agent didn't reach a terminal state in 100 steps.
```

(b) Question: A computer science student Micheal conducted an experiment setting the value of γ to be 0, 1 and 2, and respectively obtained the results shown in the screenshots of Figure 1. Based on his results, can you analysis:

When $\gamma = 0$, does the value iteration generate the optimal policy? If not, can you explain the Reason? When $\gamma = 1$, does the value iteration generate the optimal policy? If not, can you explain the Reason? When $\gamma = 2$, can you explain why we got the value function and the policy as shown? (Hint: when $\gamma = 0, 1, 2$, how does the value function update? You may print out the value function to help you analyze this question.)

This experiment is meant to showcase what happens when you choose extreme or outside of normal range discount values. Choosing $\gamma = 0$ will cause the final policy to only care about immediate reward and not care at all about future reward. Choosing $\gamma = 1$ will cause the final policy to basically mostly future reward over immediate reward. Choosing $\gamma = 2$ will cause some sort of error or unpredictable behavior because future reward will be valued much, much more than current reward in a way that will cause it to increase exponentially ($2^1, 2^2, 2^3, 2^n$ where n = number of future steps). So the final 2 or 3 future rewards will end up contributing to

most of the reward values, which will cause an inability to close out the episode, since the policy will (I think) incentivise being further away from the terminal state position on the grid.

When $\gamma = 0$, the value iteration does not generate the optimal policy. This is because the policy generated does not allow for the agent to reach the terminal state, hence reward is not maximized. This is because a $\gamma = 0$ translates to a full exclusion of future reward from consideration during policy evaluation, so the policy provides no ability to look ahead beyond immediate actions to plan a path to the terminal state. This becomes obvious when looking at the output.

Terminal Input:

python code_base.py -method value_iteration -seeds 1 -epsilon 0.001 -gamma 0

```
---- Value Iteration----
```

```
There are 2 iterations in value iteration.
```

```
policy: [['L' 'L' 'L' 'L' 'L' 'L' 'L' 'L']
```

```
['L' 'L' 'L' 'L' 'L' 'L' 'L' 'L']
```

```
['L' 'L' 'L' 'L' 'L' 'L' 'L' 'L']
```

```
['L' 'L' 'L' 'L' 'L' 'L' 'L' 'L']
```

```
['L' 'L' 'L' 'L' 'L' 'L' 'L' 'L']
```

```
['L' 'L' 'L' 'L' 'L' 'L' 'L' 'L']
```

```
['L' 'L' 'L' 'L' 'L' 'L' 'L' 'D']
```

```
['L' 'L' 'L' 'L' 'L' 'L' 'R' 'L']]
```

```
Total running time is: 0.8482604026794434 average number of iteration is: 2.0
```

```
The agent didn't reach a terminal state in 100 steps.
```

As you can see, the only policy values that seem to have been updated are in the bottom right corner, directly adjacent to the terminal state (very bottom right) location. That reinforces my answer; with $\gamma = 0$, there is no consideration for policy changes beyond benefits to the immediate reward, which means that any location that is not directly adjacent to the terminal state fails to update as these locations receive 0 reward for any action that would direct the agent towards any tile, let alone the terminal state location.

Continue on the next page.

Terminal Input:

```
python code_base.py -method value_iteration -seeds 1 -epsilon 0.001 -gamma 1
```

```
---- Value Iteration----

There are 15 iterations in value iteration.
policy: [['L' 'L' 'L' 'L' 'L' 'L' 'L' 'L']
['L' 'L' 'L' 'L' 'L' 'L' 'L' 'L']
['L' 'L' 'L' 'L' 'D' 'L' 'L' 'L']
['L' 'L' 'L' 'L' 'L' 'L' 'D' 'L']
['L' 'L' 'L' 'L' 'D' 'L' 'L' 'L']
['L' 'L' 'L' 'D' 'L' 'L' 'L' 'D']
['L' 'L' 'D' 'L' 'L' 'D' 'L' 'D']
['L' 'L' 'L' 'L' 'D' 'L' 'L' 'L']]
Total running time is: 0.846782922744751 average number of iteration is: 15.0
The agent didn't reach a terminal state in 100 steps.
```

As you can see above, a $\gamma = 1$ for value iteration also fails to result in a convergence to optimal policy. This is because rewards that are further in steps away are overpowering more immediate rewards, which is causing a lack of path creation to the terminal state grid position.

Terminal Input:

```
python code_base.py -method value_iteration -seeds 1 -epsilon 0.001 -gamma 2
```

```
---- Value Iteration----

/home/marethu/DS669_RL/Assignment-1/code_base.py:274: RuntimeWarning: overflow encountered in scalar multiply
q_values[action] += prob * (reward + gamma * value_function[next_state] * (not done))
/home/marethu/DS669_RL/Assignment-1/code_base.py:285: RuntimeWarning: invalid value encountered in subtract
delta = np.linalg.norm(value_function - value_function_prev, np.inf)

Killed
```

I used both `max()` and `np.linalg.norm()` to try to get a visual on what happens here. The overflow error prevents visualization for `np.linalg.norm()` but allows it for `max()`. I used the requested method (`np`) in all of the examples and will include it in the final code submission. As you can see above, a $\gamma = 2$ for value iteration fails to result in convergence and runs for enough iterations for an overflow error to occur. This is because in this situation, we have input a γ that is outside of the valid range $[0, 1]$, so the final future reward term so greatly dominates the more immediate rewards, and the updates lead to exponential increases in policy values that eventually overflow. There is no path generation towards the terminal state in the policy as it gets stuck in an infinite value update loop.

Continue on the next page.

(c) Set $\gamma = 0.5$, and take the screenshot of the output. Question: Does the value iteration generate the optimal policy? **How is γ related to ϵ ?**

Terminal Input:

```
python code_base.py -method value_iteration -seeds 1 -epsilon 0.001 -gamma 0.5
```

```
---- Value Iteration----

There are 11 iterations in value iteration.
policy: [['L' 'L' 'L' 'D' 'D' 'D' 'D' 'D']
['L' 'L' 'D' 'R' 'D' 'D' 'D' 'D']
['L' 'D' 'D' 'L' 'D' 'R' 'D' 'D']
['R' 'R' 'R' 'R' 'D' 'L' 'D' 'D']
['R' 'R' 'U' 'L' 'D' 'D' 'R' 'D']
['D' 'L' 'L' 'R' 'R' 'D' 'L' 'D']
['D' 'L' 'R' 'U' 'L' 'D' 'L' 'D']
['R' 'R' 'U' 'L' 'R' 'R' 'R' 'L']]
Total running time is: 0.9117031097412109 average number of iteration is: 11.0
The agent didn't reach a terminal state in 100 steps.
```

Terminal Input:

```
python code_base.py -method value_iteration -seeds 50 -epsilon 0.001 -gamma 0.5
```

```
Total running time is: 12.865140438079834 average number of iteration is: 11.0
The agent didn't reach a terminal state in 100 steps.
```

No, the value iteration method does not generate the optimal policy at $\gamma = 0.5$ as it does not reach the terminal state, however it does get much closer to doing so as many more of the policy's directions are updated from 'L' to various other values. The main problem with the policy at $\gamma = 0.5$ is that the start state is the top left corner of the grid, but the policy still causes the agent to move to the left at the start state as well as both grid positions directly adjacent to the start state, which means that there is no way for the agent to traverse any grid positions with the policy generated via $\gamma = 0.5$. That being said, this policy seems to be much more diverse than the policies at $\gamma = 0$, $\gamma = 1$, and $\gamma = 2$, so there is probably room for adjustment here.

How is γ related to ϵ ?

My following answers will be in relation to the above -gamma 0.5 case.

Continue on the next page.

Terminal Input:

```
python Assignment-1/code_base.py -method value_iteration -seeds 1 -epsilon 0.001 -gamma 0.9
```

```
---- Value Iteration----

There are 15 iterations in value iteration.
policy: [['D' 'D' 'D' 'D' 'D' 'D' 'D' 'D']
['D' 'D' 'D' 'R' 'D' 'D' 'D' 'D']
['D' 'D' 'D' 'L' 'D' 'R' 'D' 'D']
['R' 'R' 'R' 'R' 'D' 'L' 'D' 'D']
['R' 'R' 'U' 'L' 'D' 'D' 'R' 'D']
['D' 'L' 'L' 'R' 'R' 'D' 'L' 'D']
['D' 'L' 'R' 'U' 'L' 'D' 'L' 'D']
['R' 'R' 'U' 'L' 'R' 'R' 'R' 'L']]
Total running time is: 0.6721968650817871 average number of iteration is: 15.0
Episode reward: 1.0
```

Terminal Input:

```
python Assignment-1/code_base.py -method value_iteration -seeds 50 -epsilon 0.001 -gamma 0.9
```

```
Total running time is: 12.893057346343994 average number of iteration is: 15.0
Episode reward: 1.0
```

As observed above, if gamma increases and epsilon is same: convergence criterion is harder to fulfill for value_iteration method because of an increase in linalg.norm . This means that there will be more iterations of the value_iteration method.

Terminal Input:

```
python Assignment-1/code_base.py -method value_iteration -seeds 1 -epsilon 0.001 -gamma 0.1
```

```
---- Value Iteration----

There are 5 iterations in value iteration.
policy: [['L' 'L' 'L' 'L' 'L' 'L' 'L' 'L']
['L' 'L' 'L' 'L' 'L' 'L' 'L' 'L']
['L' 'L' 'L' 'L' 'L' 'L' 'L' 'D']
['L' 'L' 'L' 'L' 'L' 'L' 'D' 'D']
['L' 'L' 'L' 'L' 'L' 'D' 'R' 'D']
['L' 'L' 'L' 'L' 'R' 'D' 'L' 'D']
['L' 'L' 'L' 'L' 'L' 'D' 'L' 'D']
['L' 'L' 'L' 'L' 'R' 'R' 'R' 'L']]
Total running time is: 0.6798660755157471 average number of iteration is: 5.0
The agent didn't reach a terminal state in 100 steps.
```

Continue on the next page.

Terminal Input:

```
python Assignment-1/code_base.py -method value_iteration -seeds 50 -epsilon 0.001 -gamma 0.1
```

```
Total running time is: 12.837612390518188 average number of iteration is: 5.0
The agent didn't reach a terminal state in 100 steps.
```

As observed above, if gamma decreases and epsilon is same: convergence criterion is easier to fulfill for value_iteration method because decrease in linalg.norm . This means that there will be less iterations of the value iteration method.

Terminal Input:

```
python code_base.py -method value_iteration -seeds 1 -epsilon 0.01 -gamma 0.5
```

```
---- Value Iteration----

There are 8 iterations in value iteration.
policy: [['L' 'L' 'L' 'L' 'L' 'L' 'D' 'D']
['L' 'L' 'L' 'L' 'L' 'D' 'D' 'D']
['L' 'L' 'L' 'L' 'D' 'R' 'D' 'D']
['L' 'L' 'L' 'R' 'D' 'L' 'D' 'D']
['L' 'L' 'L' 'L' 'D' 'D' 'R' 'D']
['L' 'L' 'L' 'R' 'R' 'D' 'L' 'D']
['L' 'L' 'R' 'U' 'L' 'D' 'L' 'D']
['L' 'R' 'U' 'L' 'R' 'R' 'R' 'L']]
Total running time is: 0.8821027278900146 average number of iteration is: 8.0
The agent didn't reach a terminal state in 100 steps.
```

Terminal Input:

```
python code_base.py -method value_iteration -seeds 50 -epsilon 0.01 -gamma 0.5
```

```
Total running time is: 13.025489330291748 average number of iteration is: 8.0
The agent didn't reach a terminal state in 100 steps.
```

As observed above, if gamma is the same and epsilon increases: convergence criterion is easier to fulfill for value_iteration method because of a less strict filter of linalg.norm . This means that there will be less iterations of the value iteration method.

Terminal Input:

```
python code_base.py -method value_iteration -seeds 1 -epsilon 0.0001 -gamma 0.5
```

```
Total running time is: 13.039476871490479 average number of iteration is: 15.0
Episode reward: 1.0
```

Continue on the next page.

Terminal Input:

```
python code_base.py -method value_iteration -seeds 1 -epsilon 0.0001 -gamma 0.5
```

```
---- Value Iteration----
```

```
There are 15 iterations in value iteration.
```

```
policy: [['D' 'D' 'D' 'D' 'D' 'D' 'D' 'D']
```

```
['D' 'D' 'D' 'R' 'D' 'D' 'D' 'D']
```

```
['D' 'D' 'D' 'L' 'D' 'R' 'D' 'D']
```

```
['R' 'R' 'R' 'R' 'D' 'L' 'D' 'D']
```

```
['R' 'R' 'U' 'L' 'D' 'D' 'R' 'D']
```

```
['D' 'L' 'L' 'R' 'R' 'D' 'L' 'D']
```

```
['D' 'L' 'R' 'U' 'L' 'D' 'L' 'D']
```

```
['R' 'R' 'U' 'L' 'R' 'R' 'R' 'L']]
```

```
Total running time is: 0.8706731796264648 average number of iteration is: 15.0
```

```
Episode reward: 1.0
```

As observed above, if gamma is same and epsilon decreases: convergence criterion is harder to fulfill for value_iteration method because the epsilon filter of linalg.norm is more strict. This means that there will be more iterations of the value iteration method.

Terminal Input:

```
python code_base.py -method value_iteration -seeds 1 -epsilon 0.0001 -gamma 0.9
```

```
---- Value Iteration----
```

```
There are 15 iterations in value iteration.
```

```
policy: [['D' 'D' 'D' 'D' 'D' 'D' 'D' 'D']
```

```
['D' 'D' 'D' 'R' 'D' 'D' 'D' 'D']
```

```
['D' 'D' 'D' 'L' 'D' 'R' 'D' 'D']
```

```
['R' 'R' 'R' 'R' 'D' 'L' 'D' 'D']
```

```
['R' 'R' 'U' 'L' 'D' 'D' 'R' 'D']
```

```
['D' 'L' 'L' 'R' 'R' 'D' 'L' 'D']
```

```
['D' 'L' 'R' 'U' 'L' 'D' 'L' 'D']
```

```
['R' 'R' 'U' 'L' 'R' 'R' 'R' 'L']]
```

```
Total running time is: 0.8610212802886963 average number of iteration is: 15.0
```

```
Episode reward: 1.0
```

Terminal Input:

```
python code_base.py -method value_iteration -seeds 50 -epsilon 0.0001 -gamma 0.9
```

```
Total running time is: 13.075594186782837 average number of iteration is: 15.0
```

```
Episode reward: 1.0
```

As observed above, if gamma increases and epsilon decreases: convergence criterion is harder to fulfill due to both parameters for the value_iteration method. The filter epsilon decreases (more strict) and the update size for linalg.norm values increases, so the number of iterations increases because it's less likely to trigger the convergence criterion condition if everything else is constant.

Terminal Input:

```
python code_base.py -method value_iteration -seeds 1 -epsilon 0.01 -gamma 0.1
```

```
---- Value Iteration----

There are 4 iterations in value iteration.
policy: [['L' 'L' 'L' 'L' 'L' 'L' 'L' 'L']
['L' 'L' 'L' 'L' 'L' 'L' 'L' 'L']
['L' 'L' 'L' 'L' 'L' 'L' 'L' 'L']
['L' 'L' 'L' 'L' 'L' 'L' 'L' 'D']
['L' 'L' 'L' 'L' 'L' 'L' 'R' 'D']
['L' 'L' 'L' 'L' 'L' 'D' 'L' 'D']
['L' 'L' 'L' 'L' 'L' 'D' 'L' 'D']
['L' 'L' 'L' 'L' 'R' 'R' 'R' 'L']]
Total running time is: 0.8470184803009033 average number of iteration is: 4.0
The agent didn't reach a terminal state in 100 steps.
```

Terminal Input:

```
python code_base.py -method value_iteration -seeds 50 -epsilon 0.01 -gamma 0.1
```

```
Total running time is: 13.096455097198486 average number of iteration is: 4.0
The agent didn't reach a terminal state in 100 steps.
```

As observed above, if gamma decreases and epsilon increases: convergence criterion is easier to fulfill due to both parameters for the value_iteration method. The filter epsilon increases (less strict) and the update size for `linalg.norm` values decreases, so the number of iterations decreases because it's more likely to trigger the convergence criterion condition if everything else is constant.

Terminal Input:

```
python code_base.py -method value_iteration -seeds 50 -epsilon 0.002 -gamma 0.9
```

```
Total running time is: 13.061380624771118 average number of iteration is: 15.0
Episode reward: 1.0
```

Epsilon inc small, gamma inc large, iterations increase

Terminal Input:

```
python code_base.py -method value_iteration -seeds 50 -epsilon 0.1 -gamma 0.51
```

```
Total running time is: 13.091384410858154 average number of iteration is: 5.0
The agent didn't reach a terminal state in 100 steps.
```

Epsilon inc small, gamma inc large, iterations decrease

If gamma increases and epsilon increases: depends on what increases more, `linalg.norm` vs epsilon. In the above cases, both outcomes are shown.

Continue on the next page.

Terminal Input:

```
python code_base.py -method value_iteration -seeds 50 -epsilon 0.0009 -gamma 0.1
```

```
Total running time is: 13.08980417251587 average number of iteration is: 5.0
The agent didn't reach a terminal state in 100 steps.
```

Epsilon decrease small, gamma dec large, iterations decrease

Terminal Input:

```
python code_base.py -method value_iteration -seeds 50 -epsilon 0.0001 -gamma 0.49
```

```
Total running time is: 13.06526494026184 average number of iteration is: 14.0
Episode reward: 1.0
```

Epsilon decrease large, gamma dec small, iterations increase

If gamma decreases and epsilon decreases: depends on what decreases more, linalg.norm vs epsilon. In the above cases, both outcomes are shown.

(d) Question: If the policy found in the previous question is not optimal, can you help the value iteration find the optimal policy by modifying the value of ϵ ? Please show your modified value of ϵ and provide a screenshot.

I think that the problem is that the top left values are not updating. This could suggest that the future reward is not being valued enough by these positions to result in a change in policy behavior at the grid position. That means that increasing the weighting of the future reward via increasing gamma should result in a change in behavior and conversely actual convergence to the optimal policy. I will try an increase of gamma to $\gamma = 0.69$ and a decrease of epsilon to $\epsilon = 0.001$ to test my intuition/hypothesis.

Terminal Input:

```
python code_base.py -method value_iteration -seeds 1 -epsilon 0.001 -gamma 0.69
```

```
---- Value Iteration----

There are 15 iterations in value iteration.
policy: [['D' 'D' 'D' 'D' 'D' 'D' 'D' 'D']
['D' 'D' 'D' 'R' 'D' 'D' 'D' 'D']
['D' 'D' 'D' 'L' 'D' 'R' 'D' 'D']
['R' 'R' 'R' 'R' 'D' 'L' 'D' 'D']
['R' 'R' 'U' 'L' 'D' 'D' 'R' 'D']
['D' 'L' 'L' 'R' 'R' 'D' 'L' 'D']
['D' 'L' 'R' 'U' 'L' 'D' 'L' 'D']
['R' 'R' 'U' 'L' 'R' 'R' 'R' 'L']]
Total running time is: 1.0364775657653809 average number of iteration is: 15.0
Episode reward: 1.0
```

Cont on next page

As you can see above, $\gamma = 0.69$ achieved the desired optimal policy; the increased weighting of future reward versus the $\gamma = 0.5$ case finally causes the top left grid position values to update and direct the agent outwards towards a path that leads to the terminal state.

Terminal Input:

```
python code_base.py -method value_iteration -seeds 1 -epsilon 0.001 -gamma 0.31
```

```
---- Value Iteration----

There are 7 iterations in value iteration.
policy: [['L' 'L' 'L' 'L' 'L' 'L' 'L' 'D']
['L' 'L' 'L' 'L' 'L' 'L' 'D' 'D']
['L' 'L' 'L' 'L' 'L' 'R' 'D' 'D']
['L' 'L' 'L' 'L' 'D' 'L' 'D' 'D']
['L' 'L' 'L' 'L' 'D' 'D' 'R' 'D']
['L' 'L' 'L' 'R' 'R' 'D' 'L' 'D']
['L' 'L' 'R' 'U' 'L' 'D' 'L' 'D']
['L' 'L' 'U' 'L' 'R' 'R' 'R' 'L']]
Total running time is: 0.9982242584228516 average number of iteration is: 7.0
The agent didn't reach a terminal state in 100 steps.
```

Out of curiosity, I also tried to lower gamma to $\gamma = 0.31$; the result was a failure to achieve the optimal policy. To me, this means that the optimal values for gamma here are above $\gamma = 0.5$ and below $\gamma = 1$ in order to achieve the optimal policy assuming all other factors are held constant.

5. Effect of initialization setting (20%)

In this section, you will use policy iteration to further understand the effects of initial settings. Please remember to recover $\gamma = 0.9$ and $\epsilon = 1e - 3$. For the previous sections, we use random action for each state as the initial policy:

```
init_policy = np.random.randint(0, nA, nS) if init_action == -1 else np.ones(nS, dtype=int) *
init_action
```

(a) Please change the initial action to "go left" for all states and rerun the policy iteration. Take the screenshot of all the output results from your code. Please use the init_action in get_args.py to control the initialization of the policy. (The screenshots should include the method you are using, the policy generated, the total running time and the episode reward.)

The mappings of numbers to actions are as follows: {0: L, 1: D, 2: R, 3: U}

-1 is a random value from the list mapping 0-3 to directions

So I have to use init_action=0 for the initial action to be "go left" for all states.

Terminal Input:

```
python code_base.py -method policy_iteration -seeds 1 -epsilon 0.001 -gamma 0.9 -init_action 0
```

```
---- Policy Iteration----

There are 15 iterations in policy iteration.
policy: [['D' 'D' 'D' 'D' 'D' 'D' 'D' 'D']
['D' 'D' 'D' 'R' 'D' 'D' 'D' 'D']
['D' 'D' 'D' 'L' 'D' 'R' 'D' 'D']
['R' 'R' 'R' 'R' 'D' 'L' 'D' 'D']
['R' 'R' 'U' 'L' 'D' 'D' 'R' 'D']
['D' 'L' 'L' 'R' 'R' 'D' 'L' 'D']
['D' 'L' 'R' 'U' 'L' 'D' 'L' 'D']
['R' 'R' 'U' 'L' 'R' 'R' 'R' 'L']]
Total running time is: 0.9211387634277344 average number of iteration is: 15.0
Episode reward: 1.0
```

(b) Question: Is the policy the optimal policy? How many iterations does the policy iteration need?

This policy with “go left” initialization achieves a reward of 1 because it reaches the termination state. Strictly speaking, the policy itself is optimal depending on the definition of optimal policy. If minimum iterations to converge is required of the optimal policy, this policy is not optimal. If only a maximization of reward matters, this policy is the optimal policy. It achieves max reward 1 and does so in the minimum number of transitions (14) consisting of 7 rights and 7 downs.

(c) Please try "go right", "go up", and "go down" for all states and rerun the policy iteration method. Take a screenshot for each case.

Terminal Input:

```
python code_base.py -method policy_iteration -seeds 1 -epsilon 0.001 -gamma 0.9 -init_action 2
```

```
---- Policy Iteration----

There are 13 iterations in policy iteration.
policy: [['D' 'D' 'D' 'D' 'D' 'D' 'D' 'D']
['D' 'D' 'D' 'R' 'D' 'D' 'D' 'D']
['D' 'D' 'D' 'L' 'D' 'R' 'D' 'D']
['R' 'R' 'R' 'R' 'D' 'L' 'D' 'D']
['R' 'R' 'U' 'L' 'D' 'D' 'R' 'D']
['D' 'L' 'L' 'R' 'R' 'D' 'L' 'D']
['D' 'L' 'R' 'U' 'L' 'D' 'L' 'D']
['R' 'R' 'U' 'L' 'R' 'R' 'R' 'L']]
Total running time is: 0.8511569499969482 average number of iteration is: 13.0
Episode reward: 1.0
```

This policy with “go down” initialization achieves a reward of 1.0, it needs 13 iterations to converge.

Terminal Input:

```
python code_base.py -method policy_iteration -seeds 1 -epsilon 0.001 -gamma 0.9 -init_action 3
```

```
---- Policy Iteration----

There are 13 iterations in policy iteration.
policy: [['D' 'D' 'D' 'D' 'D' 'D' 'D' 'D']
['D' 'D' 'D' 'R' 'D' 'D' 'D' 'D']
['D' 'D' 'D' 'L' 'D' 'R' 'D' 'D']
['R' 'R' 'R' 'R' 'D' 'L' 'D' 'D']
['R' 'R' 'U' 'L' 'D' 'D' 'R' 'D']
['D' 'L' 'L' 'R' 'R' 'D' 'L' 'D']
['D' 'L' 'R' 'U' 'L' 'D' 'L' 'D']
['R' 'R' 'U' 'L' 'R' 'R' 'R' 'L']]
Total running time is: 0.9912800788879395 average number of iteration is: 13.0
Episode reward: 1.0
```

This policy with “go right” initialization achieves a reward of 1.0, it needs 13 iterations to converge.

Terminal Input:

```
python code_base.py -method policy_iteration -seeds 1 -epsilon 0.001 -gamma 0.9 -init_action 1
```

```
---- Policy Iteration----

There are 15 iterations in policy iteration.
policy: [['D' 'D' 'D' 'D' 'D' 'D' 'D' 'D']
['D' 'D' 'D' 'R' 'D' 'D' 'D' 'D']
['D' 'D' 'D' 'L' 'D' 'R' 'D' 'D']
['R' 'R' 'R' 'R' 'D' 'L' 'D' 'D']
['R' 'R' 'U' 'L' 'D' 'D' 'R' 'D']
['D' 'L' 'L' 'R' 'R' 'D' 'L' 'D']
['D' 'L' 'R' 'U' 'L' 'D' 'L' 'D']
['R' 'R' 'U' 'L' 'R' 'R' 'R' 'L']]
Total running time is: 0.9296715259552002 average number of iteration is: 15.0
Episode reward: 1.0
```

This policy with “go up” initialization achieves a reward of 1.0, it needs 15 iterations to converge.

Continue on the next page.

(d) Question: Which action(s) initialization can reduce the number of iterations? Can you explain the reason?

The initializations going right and going down can reduce the number of iterations. This is because “right” and “down” are both advantageous towards an initial policy due to the starting top left grid position (1, 1) and the terminal state bottom right grid position (8, 8). In order to path from the top left corner to the bottom right corner, the agent has to go down and to the right in all solutions; so, initializing the full grid to “go right” and/or “go down” allows for some of this necessary behavior to be derived through the initialization rather than updates due to iterations. That is why “go right” and “go down” initialization can reduce the required number of iterations to converge to the optimal policy.

(e) In the policy_evaluation function of the policy iteration method, we initialize the value function with all 0 by: value_function = np.zeros(nS) Question: How many evaluation steps do policy evaluation need to converge? Please print the evaluation_steps for each iteration and take a screenshot.

To print the evaluation_steps for each iteration, I am printing contained in policy_iteration:

```
while True:
    iteration += 1
    value_function, evaluation_steps = policy_evaluation(P, nS, policy_prev, gamma, epsilon)
    new_policy = policy_improvement(P, nS, nA, value_function, gamma)

    print(f'Evaluation Steps in Iteration {iteration}: {evaluation_steps}')
```

This allows me to keep track of how many evaluation_steps each iteration that the policy_evaluation of the current policy_iteration requires, which I can't do in the standalone policy_evaluation function.

Cont on next page.

Terminal Input:

```
python code_base.py -method policy_iteration -seeds 1 -epsilon 0.001 -gamma 0.9 -init_action -1
```

```
---- Policy Iteration----  
  
Evaluation Steps in Iteration 1: 1  
Evaluation Steps in Iteration 2: 2  
Evaluation Steps in Iteration 3: 3  
Evaluation Steps in Iteration 4: 4  
Evaluation Steps in Iteration 5: 5  
Evaluation Steps in Iteration 6: 6  
Evaluation Steps in Iteration 7: 7  
Evaluation Steps in Iteration 8: 8  
Evaluation Steps in Iteration 9: 9  
Evaluation Steps in Iteration 10: 10  
Evaluation Steps in Iteration 11: 11  
Evaluation Steps in Iteration 12: 12  
Evaluation Steps in Iteration 13: 13  
Evaluation Steps in Iteration 14: 14  
Evaluation Steps in Iteration 15: 15  
There are 15 iterations in policy iteration.  
policy: [['D' 'D' 'D' 'D' 'D' 'D' 'D' 'D']  
['D' 'D' 'D' 'R' 'D' 'D' 'D' 'D']  
['D' 'D' 'D' 'L' 'D' 'R' 'D' 'D']  
['R' 'R' 'R' 'R' 'D' 'L' 'D' 'D']  
['R' 'R' 'U' 'L' 'D' 'D' 'R' 'D']  
['D' 'L' 'L' 'R' 'R' 'D' 'L' 'D']  
['D' 'L' 'R' 'U' 'L' 'D' 'L' 'D']  
['R' 'R' 'U' 'L' 'R' 'R' 'R' 'L']]  
Total running time is: 0.9042410850524902 average number of iteration is: 15.0  
Episode reward: 1.0
```

(f) Can you optimize the initialization to reduce the number of evaluation steps? You can add a new parameter to the `policy_iteration` function and use this parameter to control the initialization. Please print out `evaluation_steps` for each iteration and take the screenshot. Question: Please describe the main idea of your initialization method.

Note: I changed the 'Modify' line to question 5. (f) instead of 5. (a).

```
# Modify the following line for initialization optimization in question 5.(f)
```

Control:

My control value_function is the original value_function = np.zeros(nS)

Continue on the next page.

(Control) Terminal Input:

```
python code_base.py -method policy_iteration -seeds 1 -epsilon 0.001 -gamma 0.9 -init_action -1
```

```
---- Policy Iteration----  
  
Evaluation Steps in Iteration 1: 1  
Evaluation Steps in Iteration 2: 2  
Evaluation Steps in Iteration 3: 3  
Evaluation Steps in Iteration 4: 4  
Evaluation Steps in Iteration 5: 5  
Evaluation Steps in Iteration 6: 6  
Evaluation Steps in Iteration 7: 7  
Evaluation Steps in Iteration 8: 8  
Evaluation Steps in Iteration 9: 9  
Evaluation Steps in Iteration 10: 10  
Evaluation Steps in Iteration 11: 11  
Evaluation Steps in Iteration 12: 12  
Evaluation Steps in Iteration 13: 13  
Evaluation Steps in Iteration 14: 14  
Evaluation Steps in Iteration 15: 15  
There are 15 iterations in policy iteration.  
policy: [['D' 'D' 'D' 'D' 'D' 'D' 'D' 'D']  
         ['D' 'D' 'D' 'R' 'D' 'D' 'D' 'D']  
         ['D' 'D' 'D' 'L' 'D' 'R' 'D' 'D']  
         ['R' 'R' 'R' 'R' 'D' 'L' 'D' 'D']  
         ['R' 'R' 'U' 'L' 'D' 'D' 'R' 'D']  
         ['D' 'L' 'L' 'R' 'R' 'D' 'L' 'D']  
         ['D' 'L' 'R' 'U' 'L' 'D' 'L' 'D']  
         ['R' 'R' 'U' 'L' 'R' 'R' 'R' 'L']]  
Total running time is: 0.8547968864440918 average number of iteration is: 15.0  
Episode reward: 1.0
```

(Control) Terminal Input:

```
python code_base.py -method policy_iteration -seeds 50 -epsilon 0.001 -gamma 0.9 -init_action -1
```

```
Total running time is: 13.177120208740234 average number of iteration is: 14.42  
Episode reward: 1.0
```

The logic behind the first test is to just pass the `policy_iteration` function an optimal policy as the initial policy. Ideally this should result in instant convergence after the very first Iteration (one iteration would be required to adopt the policy necessary for termination of `policy_iteration`).

Cont on next page.

Test 1.

I'm just going to initialize the `init_policy` as a previous optimized policy.

```
init_policy = np.array(['D', 'D', 'D', 'D', 'D', 'D', 'D', 'D',  
                        'D', 'D', 'D', 'R', 'D', 'D', 'D', 'D',  
                        'D', 'D', 'D', 'L', 'D', 'R', 'D', 'D',  
                        'R', 'R', 'R', 'R', 'D', 'L', 'D', 'D',  
                        'R', 'R', 'U', 'L', 'D', 'D', 'R', 'D',  
                        'D', 'L', 'L', 'R', 'R', 'D', 'L', 'D',  
                        'D', 'L', 'R', 'U', 'L', 'D', 'L', 'D',  
                        'R', 'R', 'U', 'L', 'R', 'R', 'R', 'L'])
```

Test 1 (cont.).

(Test 1) Terminal Input:

```
python code_base.py -method policy_iteration -seeds 1 -epsilon 0.001 -gamma 0.9 -init_action  
-1
```

```
Evaluation Steps in Iteration 1: 15  
There are 1 iterations in policy iteration.  
policy: [['D' 'D' 'D' 'D' 'D' 'D' 'D' 'D']  
         ['D' 'D' 'D' 'R' 'D' 'D' 'D' 'D']  
         ['D' 'D' 'D' 'L' 'D' 'R' 'D' 'D']  
         ['R' 'R' 'R' 'R' 'D' 'L' 'D' 'D']  
         ['R' 'R' 'U' 'L' 'D' 'D' 'R' 'D']  
         ['D' 'L' 'L' 'R' 'R' 'D' 'L' 'D']  
         ['D' 'L' 'R' 'U' 'L' 'D' 'L' 'D']  
         ['R' 'R' 'U' 'L' 'R' 'R' 'R' 'L']]  
Total running time is: 0.5629584789276123 average number of iteration is: 1.0  
Episode reward: 1.0
```

(Test 1) Terminal Input:

```
python code_base.py -method policy_iteration -seeds 50 -epsilon 0.001 -gamma 0.9 -init_action  
-1
```

```
Total running time is: 13.05618166923523 average number of iteration is: 1.0  
Episode reward: 1.0
```

As you can see, this passing of an optimal policy results in only 15 total evaluation steps and only a single iteration. But this is pretty easy to expect, because there is no reason for the initial policy to update during the policy improvement stage.

Continue on the next page.

Test 2:

Resetting the initialization to the default value from the change in part 1 back to

```
init_policy = np.random.randint(0, nA, nS) if init_action == -1 else np.ones(nS, dtype=int) * init_action
```

I think that I can basically cheat the system here by trying to use the final value function values from control's final update of the value_function. This means I have to change the initialization in policy_evaluation of value_function = np.zeros(nS) to the aforementioned final value_function values as the start point for each call to policy_evaluation. The logic behind this approach is that if I already have the value_function from a prior optimal policy solution, then I can use that value_function to avoid the need to update value_function during the policy_evaluation stage. I think that this should result in a faster policy convergence because it will require a lower amount of iterations of value iterations to achieve the optimal policy. My value_function derived from the value_function of the value function of the policy convergence iteration of np.zeros(nS) at -seeds 1 -epsilon 0.001 -gamma 0.9 -init_action -1 is as follows:

```
def policy_evaluation(P, nS, policy, gamma=0.9, epsilon=1e-3):  
    # 5(f). Test 2.  
    value_function = np.array([0.254, 0.282, 0.314, 0.349, 0.387, 0.43, 0.478, 0.531,  
                               0.282, 0.314, 0.349, 0.387, 0.43, 0.478, 0.531, 0.59,  
                               0.314, 0.349, 0.387, 0., 0.478, 0.531, 0.59, 0.656,  
                               0.349, 0.387, 0.43, 0.478, 0.531, 0., 0.656, 0.729,  
                               0.314, 0.349, 0.387, 0., 0.59, 0.656, 0.729, 0.81,  
                               0.282, 0., 0., 0.59, 0.656, 0.729, 0., 0.9,  
                               0.314, 0., 0.478, 0.531, 0., 0.81, 0., 1.,  
                               0.349, 0.387, 0.43, 0., 0.81, 0.9, 1., 0. ])
```

(Test 2) Terminal Input:

```
python code_base.py -method policy_iteration -seeds 1 -epsilon 0.001 -gamma 0.9 -init_action  
-1
```

```
---- Policy Iteration----  
Evaluation Steps in Iteration 1: 48  
Evaluation Steps in Iteration 2: 49  
Evaluation Steps in Iteration 3: 48  
Evaluation Steps in Iteration 4: 47  
Evaluation Steps in Iteration 5: 44  
Evaluation Steps in Iteration 6: 43  
Evaluation Steps in Iteration 7: 35  
Evaluation Steps in Iteration 8: 35  
Evaluation Steps in Iteration 9: 35  
Evaluation Steps in Iteration 10: 34  
Evaluation Steps in Iteration 11: 1  
Evaluation Steps in Iteration 12: 1  
There are 12 iterations in policy iteration.  
policy: [['D' 'D' 'D' 'D' 'D' 'D' 'D' 'D']  
         ['D' 'D' 'D' 'R' 'D' 'D' 'D' 'D']  
         ['D' 'D' 'D' 'L' 'D' 'R' 'D' 'D']  
         ['R' 'R' 'R' 'R' 'D' 'L' 'D' 'D']  
         ['R' 'R' 'U' 'L' 'D' 'D' 'R' 'D']  
         ['D' 'L' 'L' 'R' 'R' 'D' 'L' 'D']  
         ['D' 'L' 'R' 'U' 'L' 'D' 'L' 'D']  
         ['R' 'R' 'U' 'L' 'R' 'R' 'R' 'L']]  
Total running time is: 0.6822342872619629 average number of iteration is: 12.0  
Episode reward: 1.0
```


(Test 2) Terminal Input:

```
python code_base.py -method policy_iteration -seeds 50 -epsilon 0.001 -gamma 0.9 -init_action -1
```

```
Total running time is: 12.948545932769775 average number of iteration is: 11.34
Episode reward: 1.0
```

As you can see, after the first test with all other values constant, I was able to decrease the number of iterations required for convergence, however I was not able to decrease the number of evaluation steps. Interestingly, if I increase epsilon to say, 1:

(Test 2) Terminal Input:

```
python code_base.py -method policy_iteration -seeds 50 -epsilon 1 -gamma 0.9 -init_action -1
```

```
---- Policy Iteration----

Evaluation Steps in Iteration 1: 1
Evaluation Steps in Iteration 2: 1
Evaluation Steps in Iteration 3: 1
Evaluation Steps in Iteration 4: 1
Evaluation Steps in Iteration 5: 1
Evaluation Steps in Iteration 6: 1
Evaluation Steps in Iteration 7: 1
There are 7 iterations in policy iteration.
policy: [['D' 'D' 'D' 'D' 'D' 'D' 'D' 'D']
['D' 'D' 'D' 'R' 'D' 'D' 'D' 'D']
['D' 'D' 'D' 'L' 'D' 'R' 'D' 'D']
['R' 'R' 'R' 'R' 'D' 'L' 'D' 'D']
['R' 'R' 'U' 'L' 'D' 'D' 'R' 'D']
['D' 'L' 'L' 'R' 'R' 'D' 'L' 'D']
['D' 'L' 'R' 'U' 'L' 'D' 'L' 'D']
['R' 'R' 'U' 'L' 'R' 'R' 'R' 'L']]
Total running time is: 0.8406710624694824 average number of iteration is: 7.0
Episode reward: 1.0
```

(Test 2) Terminal Input:

```
python code_base.py -method policy_iteration -seeds 50 -epsilon 1 -gamma 0.9 -init_action -1
```

```
Total running time is: 13.024660110473633 average number of iteration is: 10.0
Episode reward: 1.0
```

I can get the number of iterations required for convergence down to an average of 10, and the number of evaluation steps required for convergence to the optimal policy down to 1. I think this is possible because my value_function is already optimal, and increasing epsilon while holding gamma constant decreases the strictness of the filter in policy_evaluation, which means that minimal noise (from non-optimal policy associated updates to the value_function) affects the value_function before it is passed to policy_improvement function to generate a new_policy for the policy_iteration function.

I think that similar behavior can be observed by greatly lowering gamma:

(Test 2) Terminal Input:

```
python code_base.py -method policy_iteration -seeds 1 -epsilon 0.001 -gamma 0.001  
-init_action -1
```

```
---- Policy Iteration----  
Evaluation Steps in Iteration 1: 2  
Evaluation Steps in Iteration 2: 2  
Evaluation Steps in Iteration 3: 2  
Evaluation Steps in Iteration 4: 2  
Evaluation Steps in Iteration 5: 2  
Evaluation Steps in Iteration 6: 2  
Evaluation Steps in Iteration 7: 2  
Evaluation Steps in Iteration 8: 2  
Evaluation Steps in Iteration 9: 2  
Evaluation Steps in Iteration 10: 2  
Evaluation Steps in Iteration 11: 2  
Evaluation Steps in Iteration 12: 2  
There are 12 iterations in policy iteration.  
policy: [['D' 'D' 'D' 'D' 'D' 'D' 'D' 'D']  
['D' 'D' 'D' 'R' 'D' 'D' 'D' 'D']  
['D' 'D' 'D' 'L' 'D' 'R' 'D' 'D']  
['R' 'R' 'R' 'R' 'D' 'L' 'D' 'D']  
['R' 'R' 'U' 'L' 'D' 'D' 'R' 'D']  
['D' 'L' 'L' 'R' 'R' 'D' 'L' 'D']  
['D' 'L' 'R' 'U' 'L' 'D' 'L' 'D']  
['R' 'R' 'U' 'L' 'R' 'R' 'R' 'L']]  
Total running time is: 0.8904557228088379 average number of iteration is: 12.0  
Episode reward: 1.0
```

(Test 2) Terminal Input:

```
python code_base.py -method policy_iteration -seeds 50 -epsilon 0.001 -gamma 0.001  
-init_action -1
```

```
Total running time is: 13.087054252624512 average number of iteration is: 10.18  
Episode reward: 1.0
```

What I observed here is that the average number of iterations tends towards 10, and the average number of evaluation steps per iteration is about 2. This matches up with my conclusion from earlier that all things held constant, decreasing gamma will decrease the number of iterations required. It also decreases the total number of evaluation steps per iteration and in total required. This is because the lower gamma causes enough update to the optimal value_function to pass the convergence criterion while still retaining the faint signal from the optimal value_function.

In summary, you can optimize the initialization of some values in order to reduce the number of evaluation steps required to reach an optimal policy. You can also optimize the number of iterations required to reach an optimal policy as well. You can do this via changing the initial policy defined during policy_iteration to a previous optimal policy or via changing the initial value_function defined during policy_evaluation to the previous optimal policy's termination state (final iteration) value_function.

End of Assignment