QLearning

March 14, 2024

0.1 Quick introduction to jupyter notebooks

- Each cell in this notebook contains either code or text.
- You can run a cell by pressing Ctrl-Enter, or run and advance to the next cell with Shift-Enter.
- Code cells will print their output, including images, below the cell. Running it again deletes the previous output, so be careful if you want to save some results.
- You don't have to rerun all cells to test changes, just rerun the cell you have made changes to. Some exceptions might apply, for example if you overwrite variables from previous cells, but in general this will work.
- If all else fails, use the "Kernel" menu and select "Restart Kernel and Clear All Output". You can also use this menu to run all cells.
- A useful debug tool is the console. You can right-click anywhere in the notebook and select "New console for notebook". This opens a python console which shares the environment with the notebook, which let's you easily print variables or test commands.

0.1.1 Setup

[1]:
```
# Automatically reload modules when changed
%reload_ext autoreload
%autoreload 2
# Plot figures "inline" with other output
%matplotlib inline

# Most important package
import numpy as np

# The reinforcement learning environment
from gridworld import GridWorld

# Configure nice figures
from matplotlib import pyplot as plt
plt.rcParams['figure.facecolor']='white'
plt.rcParams['figure.figsize']=(14,7)
```

0.1.2 *! IMPORTANT NOTE !*

Your implementation should only use the `numpy` (`np`) module. The `numpy` module provides all the functionality you need for this assignment and makes it easier debuging your code. No other modules, e.g. `scikit-learn` or `scipy` among others, are allowed and solutions using modules other than `numpy` will be sent for re-submission. You can find everything you need about `numpy` in the official documentation.

0.2 1. Reinforcement Learning, introduction

In the previous assignments we have explored supervised learning, in other words, methods that train a model based on known inputs and targets. This time, we will instead look at a branch of machine learning that is much closer to the intuitive notion of “learning”. Reinforcement learning, or RL for short, does not work with inputs and targets, but instead learns by performing **actions** in an **environment** and observing the generated **rewards**.

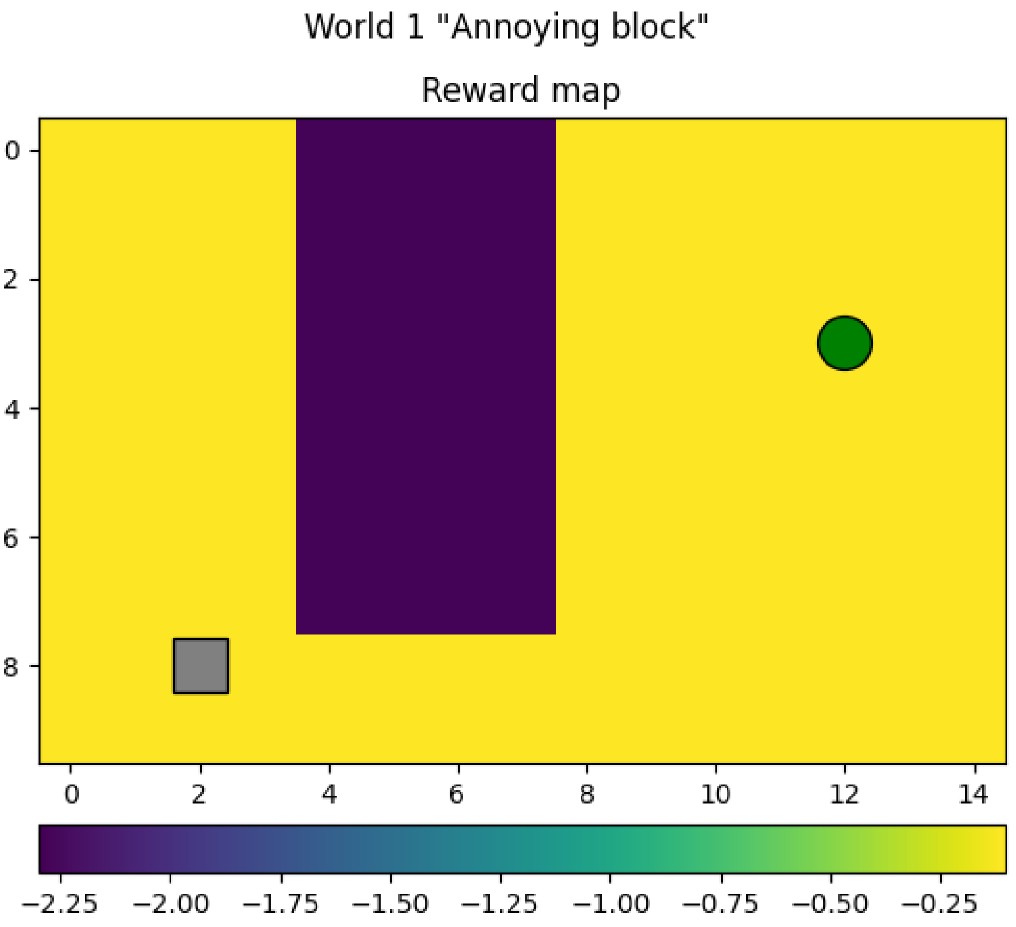
RL is a very broad concept and many different algorithms have been deviced based on these general concepts of actions and rewards. Perhaps the main advantage of RL over other machine learning techniques is that we do not explicitly tell the model what the right answer is (like we have done in the previous assignments), but instead only tell the model when the desired outcome has been acheived. This might seem like the same thing at first, but the key difference is that RL allows the model to device solutions that outperform the human teacher. This is usually not possible in traditional supervised learning since the model can only get as good as the training data (the teacher). With the freedom to explore new strategies, which is inherent to RL, this is no longer true and some truly astounding results have been acheved. The most famous example is probably AlphaGo, the first computer program to beat a human expert in the board game Go. Here is an excellent documentary, if you have some time to spare. For those of you that want a quicker and more fun example, here is a video about RL agents playing hide and seek, which very clearly demonstrates the power of RL to invent new and hidden strategies.

Of course, these examples are from the very forefront of current research in RL, and are unfortunately too complex for this assignment. We will instead work on a much simpler problem, but the core concepts that you will implement and investigate here are the same that made the above possible.

0.2.1 1.1 Getting to know the environment interface

To do this assignment you must first get familiar with the code interface to the environment, or “World”, as we will call it. You will work with a special type of environment called a **GridWorld**. The GridWorld is, as the name suggests, a world where each state is represented by a square on a grid. To create an instance of a GridWorld, run the following code. You can change the input number to select a different world. You will work with worlds 1-4, but there are other optional worlds as well, which we encourage you to explore at the end of the notebook.

```
[3]: W = GridWorld(1)
     W.init()
     W.draw()
```



Question 1: The colored background represents the reward for entering each state. Notice that all rewards are negative. Can you think of why this is important?

Answer: The reasons to set all rewards to negative are mainly because:

1. We can use negative reward to punish the undesired actions. Like the walls or boundaries, we can set the reward to negative to discourage the agent from hitting the walls or boundaries again after several trials.
2. We can model the real-world scenarios. In real world scenarios, the cost existed everywhere. For example when we drive a car on the road, when we hit the wall or out of the road, we will definitely get a negative reward. So negative reward can fit the real-world scenarios better.
3. Negative reward can guide the learning system to find the optimal solution. In this World game, only the target will have a positive reward, all the other points have negative rewards. This will help agent not to get stuck somewhere and keep exploring the environment to find the target. Since for every step, the agent will get a negative reward, this will encourage the agent to find the target as soon as possible.

The **Agent** is represented by the gray square, and will traverse the environment in order to reach the **goal** state, represented by the green circle. You can access all information you need regarding the state of the GridWorld by the methods of the World class. Here is the full list with explanations for each method:

- `getWorldSize()` - Returns a tuple with the size of each dimension in the state space. For the GridWorlds, this is the y-size and x-size of the grid.
- `getDimensionNames()` - Returns a list with the names for each dimension. This is only used to understand the world better, and should not be used to design the algorithm.
- `getActions()` - Returns a list of available actions in the form of strings. These are the only accepted values to pass to `doAction`.
- `init()` - Initializes the World. For example this resets the position of the agent in the GridWorlds. Do this at the beginning of each epoch.
- `getState()` - Returns the current state of the World, which for a GridWorld is the position of the agent.
- `doAction(act)` - Performs an action and returns a 2-tuple indicating if the actions was valid, and the corresponding reward.
- `draw(epoch, Q)` - Update any plots associated with the World. The two arguments are optional but will include more information in the plots if you provide them.

Here are some examples:

[4]:
```
W = GridWorld(1)

print("World size:", W.getWorldSize())
print("Dimension names:", W.getDimensionNames())
print("Actions:", W.getActions())
```

```
World size: (10, 15)
Dimension names: ['Y', 'X']
Actions: ['Down', 'Up', 'Right', 'Left']
```

Here is an example of some actions in the first GridWorld. Read the code and output and make sure you understand how this works before proceeding. You can quickly run the cell multiple times by holding `Ctrl` and pressing `Enter` to generate a new output.

[5]:
```
W = GridWorld(1)
W.init()

# Check state
state, isTerm = W.getState()
print(f"State initialized to {state}.")

# Make action
a = "Down"
isValid, reward = W.doAction(a)
print(f"Action '{a}' was {'' if isValid else 'not '}valid and gave a reward of␣
 ↪{reward}.")
```

```
# Check state
state, isTerm = W.getState()
print(f"State is {state} and is {('' if isTerm else 'not ')}terminal.")

# Make action
a = "Right"
isValid, reward = W.doAction(a)
print(f"Action '{a}' was {'' if isValid else 'not '}valid and gave a reward of␣
↪{reward}.")

# Check state
state, isTerm = W.getState()
print(f"State is {state} and is {('' if isTerm else 'not ')}terminal.")
```

```
State initialized to (8, 9).
Action 'Down' was valid and gave a reward of -0.1.
State is (9, 9) and is not terminal.
Action 'Right' was valid and gave a reward of -0.1.
State is (9, 10) and is not terminal.
```

0.3 2. Implementing the Q-learning algorithm

You will now implement the main algorithm of this assignment, **Q-learning**. This algorithm is powerful since it allows the simultaneous exploration of different **policies**. This is done by a state-action table **Q**, keeping track of the expected reward associated with each action in each state. By iteratively updating these estimates as we get new rewards, the policies explored by the agent eventually converges to the optimal policy. This can all be summarized in the following equation:

$$Q\left(s_t, a\right) \leftarrow \underbrace{Q\left(s_t, a\right)}_{\text{Oldvalue}} \cdot (1-\alpha) + \alpha \cdot \underbrace{\left(r + \gamma V\left(s_{t+1}\right)\right)}_{\text{Newestimate}}$$

This defines that the value of Q in a state s_t for action a, i.e $Q\left(s_t, a\right)$, should be updated as a weighted average of the old value and a new estimate, where the weighting is based on the learning rate $\alpha \in (0, 1)$. The new estimate is a combination of the reward r for the action we are updating, and the estimated value V of the next state s_{t+1}, discounted by the factor $\gamma \in (0, 1]$. By increasing γ, the future value is weighted higher, which is why we say that this optimizes for long-term rewards.

0.3.1 2.1 The training function

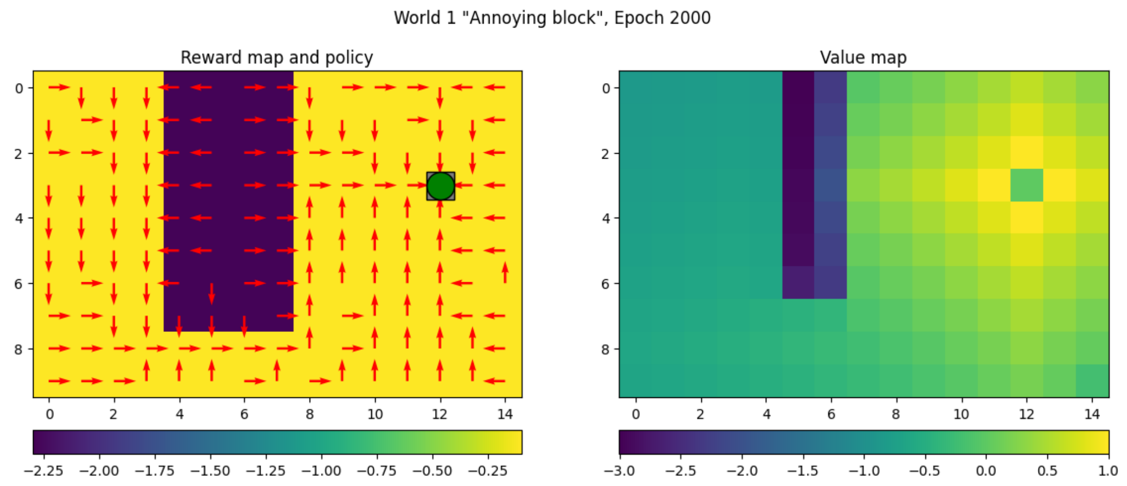
First, you will implement the Q-learning algorithm training loop in the following function. The inputs to this function is a World object, and a dictionary for any parameters needed for the training. This dictionary will contain the following parameters, which you will need `params = {"Epochs": 100, "MaxSteps": 100: "Alpha": 0.5: "Gamma": 0.9, "ExpRate": 0.5, "DrawInterval": 100}`. Note that these values are only examples, you will have to change them when optimizing each world. You access the content of the dictionary by it's name, for example `params["Gamma"]`. Using this style makes it very easy later in the notebook to try new worlds and parameter combinations.

Finally before you begin, here are some concrete tips to keep in mind while working: * Try your code often! Jump ahead to section 3.1 to easily run the training in the first GridWorld. * As part of this implementation, you must also implement the functions `getpolicy` and `getvalue` in `utils.py`. When you have implemented these the `draw` function will automatically show the results of the training!

```
[7]: def QLearning(World, params={}):

         # Init world and get size of dimensions
         WSize = World.getWorldSize()
         A = World.getActions()
         NA = len(A)

         # ---------------------------------------------
         # === Your code here ==========================
         # ---------------------------------------------

         gamma = params["Gamma"]
         alpha = params["LR"]
         exp_rate = params["Eps"]
         epochs = params["Epochs"]
         max_steps = params["MaxSteps"]
         draw_interval = params["DrawInterval"]

         if "Term_Reward" in params:
             term_reward =params["Term_Reward"]
         else:
             term_reward = 1


         # Initialize the Q-matrix (use the size variables above)
         Q = np.zeros((*WSize, NA))

         for i in range(epochs):
             # reset the agent to random position
             World.init()

             # Limiting the number of steps in an epoch prevents getting stuck in␣
       ↪infinite loops
             for j in range(max_steps):

                 # state id
                 state,_ = World.getState()

                 # Choose action
                 if np.random.rand() < exp_rate:
                     # Random action
```

```
                action = np.random.choice(A)
            else:
                # Greedy action
                action = A[np.argmax(Q[state])]

            # Perform action
            isValid, reward = World.doAction(action)

            next_state,isTerm = World.getState()

            #if not isValid:
            #    reward = -1

            if isTerm:
                reward = term_reward

            action_id = A.index(action)

            #old_value = Q[state[0],state[1]][action_id]
            #next_max = np.max(Q[next_state[0],next_state[1]])

            old_value = Q[(*state,action_id)]
            next_max = np.max(Q[next_state])

            if World.Name == "Annoying random block":
                # contain random reward
                # area [:8, 4:8] has 0.2 chance to get -11  reward
                # area [:8, 4:8] has 0.8 chance to get -0.1 reward
                if next_state[0] < 8 and next_state[1] >= 4 and next_state[1] <␣
↪8:
                    reward = (0.2 * -11) + 0.8 * (-0.1)

            new_value = (1 - alpha) * old_value + alpha * (reward + gamma *␣
↪next_max)

            #Q[state[0],state[1]][action_id] = new_value
            Q[(*state,action_id)] = new_value

            if isTerm:
                break

        # Update plots with regular intervals
        if ((i+1) % draw_interval == 0) or (i == epochs - 1):

            World.draw(epoch=(i+1), Q=Q)

    # ============================================
    return Q
```

[16]:
```
W1 = GridWorld(1)

# ---------------------------------------------
# === Your code here ========================
# ---------------------------------------------
Q1 = QLearning(W1, params={"LR": 0.5, "Gamma": 0.9, "Eps": 0.5, "Epochs": 2000,␣
 ↪"MaxSteps": 200, "DrawInterval": 100, "Term_Reward": 1})
```



0.3.2 2.2 The test function

It's important to test the performance of the trained model. This *could* be done with some heuristic function that measures properties such as path lenghts and total rewards, but here we choose to instead use a more direct evaluation method. In the following function you should implement a test loop where you follow the optimal policy and draw the world after *each* action. Since this is code to test the trained model, you should not update Q, only use it to determine the optimal actions.

[9]:
```
def QLearningTest(W, Q, params={}):

    # The number of epochs is now the number of tests runs to do
    for i in range(params["Epochs"]):

        # Init the world and get state
        W.init()
        A = W.getActions()
        s,_ = W.getState()

        # Again we limit the number of steps to prevent infinite loops
        for j in range(params["MaxSteps"]):
            # --------------------------------------------
            # === Your code here =========================
            # --------------------------------------------

            # Choose and perform optimal action from policy
            #action = np.argmax(Q[s[0],s[1]])
            action = np.argmax(Q[s])
            W.doAction(A[action])
            # ============================================

            # Get updated state and draw
            s,isTerm = W.getState()
            W.draw(epoch=(i+1), Q=Q)

            # Check if goal
            if isTerm:
                break
```

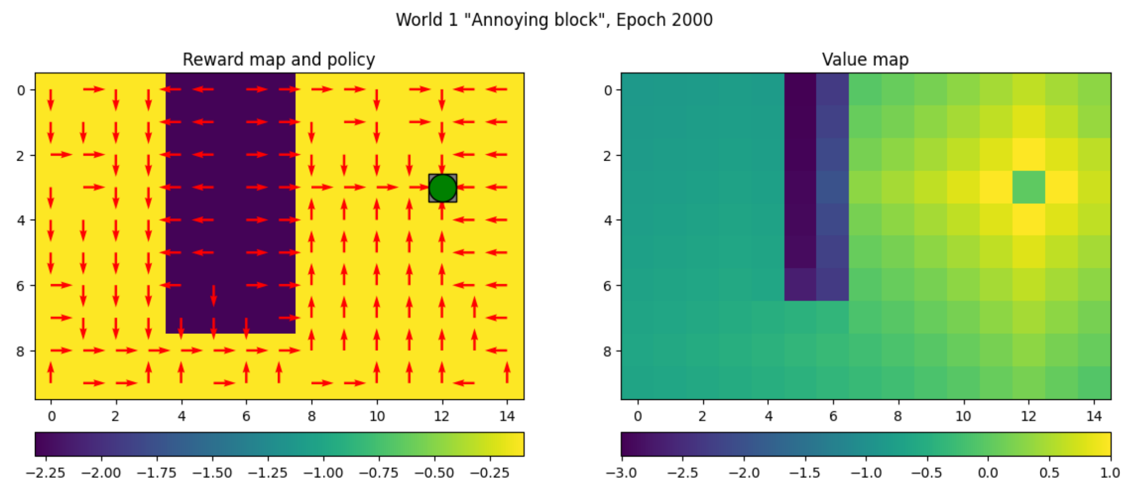
0.4 3. Optimizing the different worlds

In this section you will optimize the hyperparameters to train the 4 first GridWorlds.

0.4.1 3.1 GridWorld 1

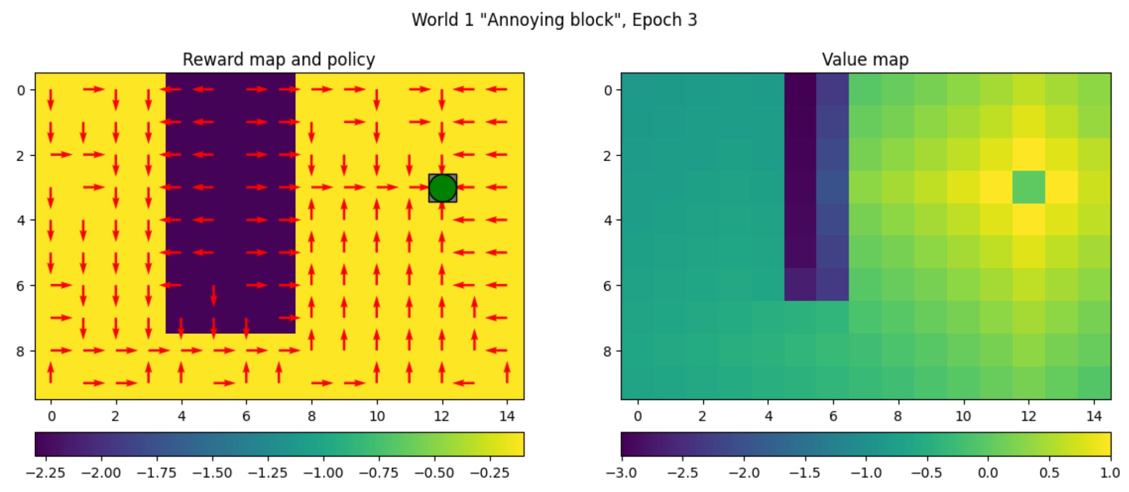
We start with the simplest of the worlds, "Annoying block". The policy should converge without much difficulty, so use this as a test to see if your implementaion is correct. If you use a good set of hyperparameters, you can expect a rather neat policy in about 1000 epochs.

[17]:
```
W1 = GridWorld(1)

# --------------------------------------------
# === Your code here =========================
# --------------------------------------------

Q1 = QLearning(W1, params={"LR": 0.5, "Gamma": 0.9, "Eps": 0.5, "Epochs": 2000,␣
 ↪"MaxSteps": 200, "DrawInterval": 100,"Term_Reward": 1})
```



Don't forget to run a few tests with the optimized policy to see if the solution looks reasonable.

[18]:
```
QLearningTest(W=W1, Q=Q1, params={"Epochs": 3, "MaxSteps": 50})
```



Question 2:

1. Describe World 1.
2. What is the goal for the agent in this world?
3. What is a good choice of learning rate in this world? Motvate your answer.

Answer:

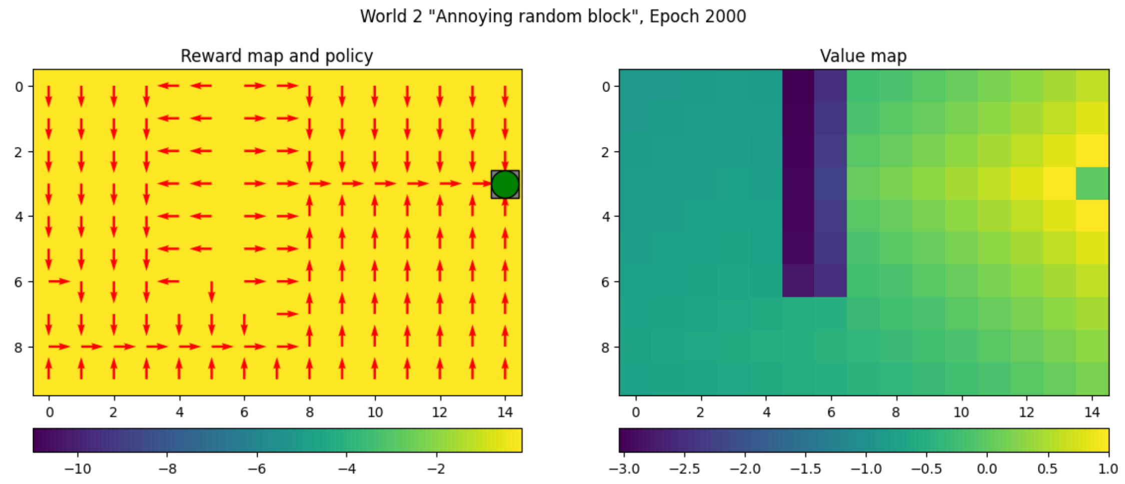
1. World 1 is a 10 * 15 world with a dark area in it. All the yellow area points have -0.1 reward. Points in the dark area[:8, 4:8] have -2.28 reward. The final target located at (3, 12) and reward is set to 1.

2. The goal for the agent in this world is point (3, 12) , and also need to make sure move from the init point to the goal with max sum of rewards and min steps.
3. We choose the default learning rate = 0.5 here.Since after 2000 epochs, the test result seems not bad.And the arrows in the plot are also reasonable.

Now continue optimizing worlds 2-4. Note that the optimal hyperparmeters potentially are very different for each world.

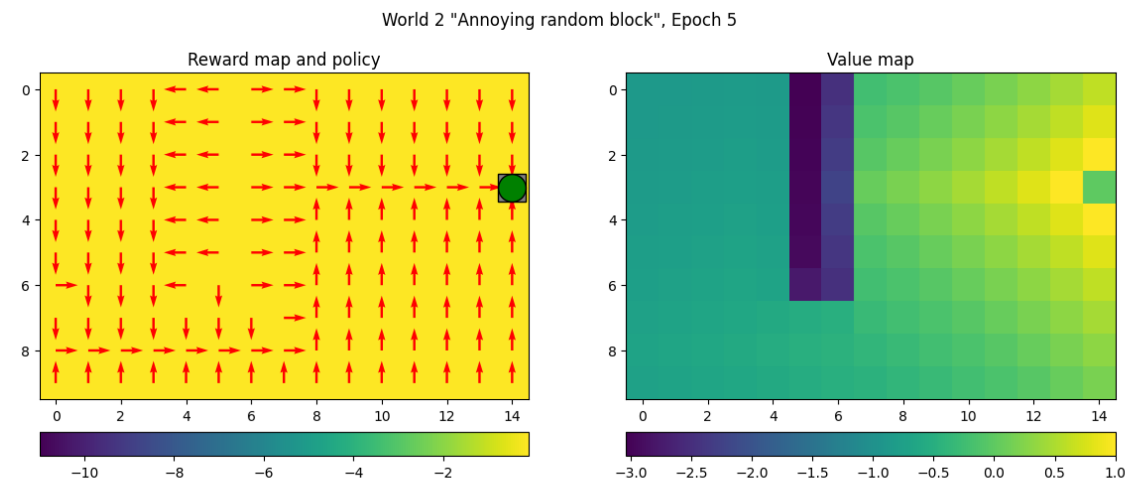
0.4.2 3.2 GridWorld 2

[19]:
```
W2 = GridWorld(2)
W2.init()
# ---------------------------------------------
# === Your code here ========================
# ---------------------------------------------
Q2 = QLearning(W2, params={"LR": 0.5, "Gamma": 0.9, "Eps": 0.8, "Epochs": 2000,␣
 ↪"MaxSteps": 200, "DrawInterval": 100,"Term_Reward": 1})
# ============================================
```



Don't forget to run a few tests with the optimized policy to see if the solution looks reasonable.

[20]:
```
QLearningTest(W=W2, Q=Q2, params={"Epochs": 5, "MaxSteps": 100})
```



Question 3:

1. Describe World 2.
2. This world has a hidden trick. Describe this trick and why this can be solved with reinforcement learning.
3. What is the goal for the agent in this world?
4. What is a good choice of learning rate in this world? Motvate your answer.
5. Compared to the optimal policy in World 1, how do we expect the optimal policy to look in this world? Motivate your answer.

Answer:

1. World 2 is a 10 * 15 world with a dark area:8, 4:8 in it.All the yellow area points have -0.1 reward. Points in the dark area (with probity of 0.2) have -11 reward.The final target located at (3, 14).

2. Points in the dark area (with probity of 0.2) have -11 reward. Since the dark area's reward will be - 11 * 0.2 - 0.1 * 0.8 = -2.8, so reward of points in the dark area are same as World 1.

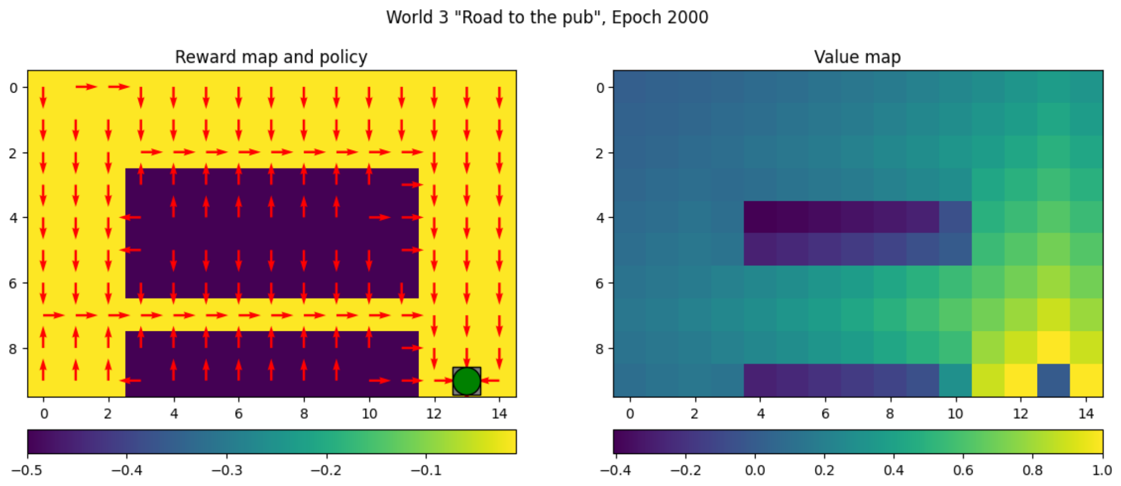
3. The goal for the agent in this world is point (3, 14) , and also need to make sure move from the init point to the goal with max sum of rewards.

4. We choose the default learning rate = 0.5 here.Since after 2000 epochs, the test result seems not bad.And the arrows in the plot are also reasonable.

5. As mentioned before, since the dark area points have the same reward as World 1, so the optimal policy in World 2 should be similar to World 1.

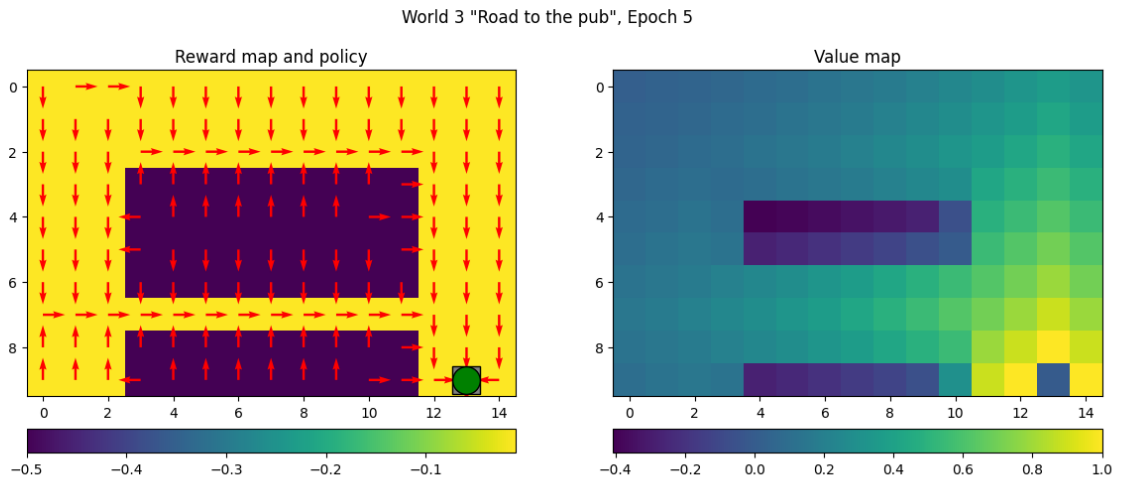
0.4.3 3.3 GridWorld 3

[22]:
```
W3 = GridWorld(3)

# ---------------------------------------------
# === Your code here ========================
# ---------------------------------------------
Q3 = QLearning(W3, params={"LR": 0.5, "Gamma": 0.9, "Eps": 0.8, "Epochs": 2000,␣
↪"MaxSteps": 200, "DrawInterval": 100,"Term_Reward": 1})
# =============================================
```



Don't forget to run a few tests with the optimized policy to see if the solution looks reasonable.

[23]:
```
QLearningTest(W=W3, Q=Q3, params={"Epochs": 5, "MaxSteps": 100})
```



Question 4:

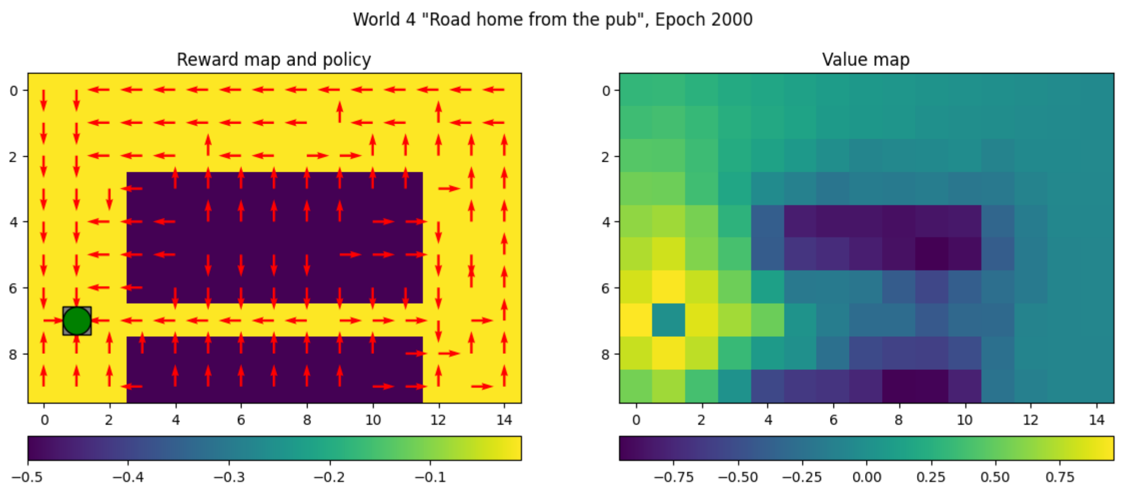
1. Describe World 3.
2. From the perspective of the learning algorithm, how does this world compare to World 1?
3. What is the goal for the agent in this world?
4. Is it possible to get a good policy in every state in this world? If so, which hyperparameter is particulary important to acheive this?

Answer:

1. World 3 is a 10 * 15 world.The following areas [:3, :],[7, :],[:, :3],[:, 12:15] have -0.01 reward, other areas have -0.5 reward.
2. compare to World 1, World 3 has more areas with different rewards. World 1 only one different reward area , but World 3 has 4 different reward areas. It also means the agent has more choices to make in World 3.(different optimized path)
3. World 3 has a fixed goal point at (9,13).
4. To adopt the best policy, we can change the eps rate to a small value. However, for some complex worlds, if we set eps to a small value, the agent may get stuck in some local optimal. So we need to set a proper eps value to make sure the agent can explore the environment and find the optimal policy.

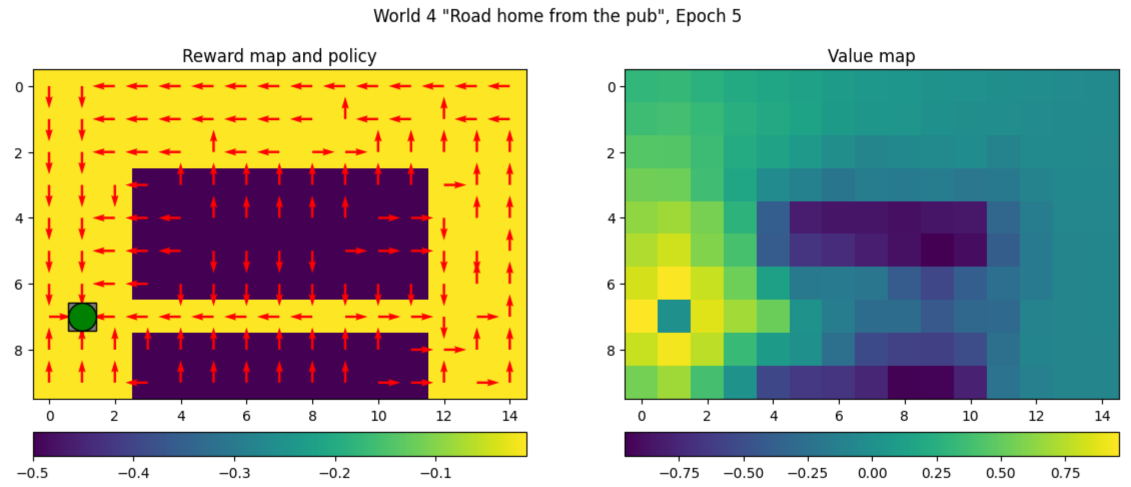
0.4.4 3.4 GridWorld 4

[33]:
```python
W4 = GridWorld(4)

# ---------------------------------------------
# === Your code here =========================
# ---------------------------------------------

Q4 = QLearning(W4, params={"LR": 0.1, "Gamma": 0.9, "Eps": 0.5, "Epochs": 2000,␣
 ↪"MaxSteps": 200, "DrawInterval": 100,"Term_Reward": 1})

# =============================================
```



Don't forget to run a few tests with the optimized policy to see if the solution looks reasonable. **Important**: You might think the policy above looks bad, but we encourage you to run this test even if you think it's not optimal. It might give you some insight into the world behaviour.

[34]:
```
QLearningTest(W=W4, Q=Q4, params={"Epochs": 5, "MaxSteps": 100})
```



Question 5:

1. Describe World 4 using your own words.
2. This world has a hidden trick. What is it, and how does this world differ from World 3?
3. What is the goal for the agent in this world?
4. What is a good choice of learning rate in this world? Motvate your answer.
5. How should we expect the optimal policy too look like? In other words, what is the optimal path from start to goal in this world? Motivate your answer.

Answer:

1. World 4 is a 10 * 15 world.The following areas [:3, :],[7, :],[:, :3],[:, 12:15] have -0.01 reward, other areas have -0.5 reward.

2. compare to World 3, World 4 will do some random action when we call doAction function.

   ```
   if np.random.rand() < 0.3:
       act = np.random.choice(self.getActions())
   ```

3. World 4 has a fixed goal point at (7,1).Start point is fixed at (9,13).

4. Since there has some random action in World 4, so we need to set a small learning rate to make sure the agent can explore the environment and find the optimal policy.Since small learning rate will help to minimize the impact of the random action.

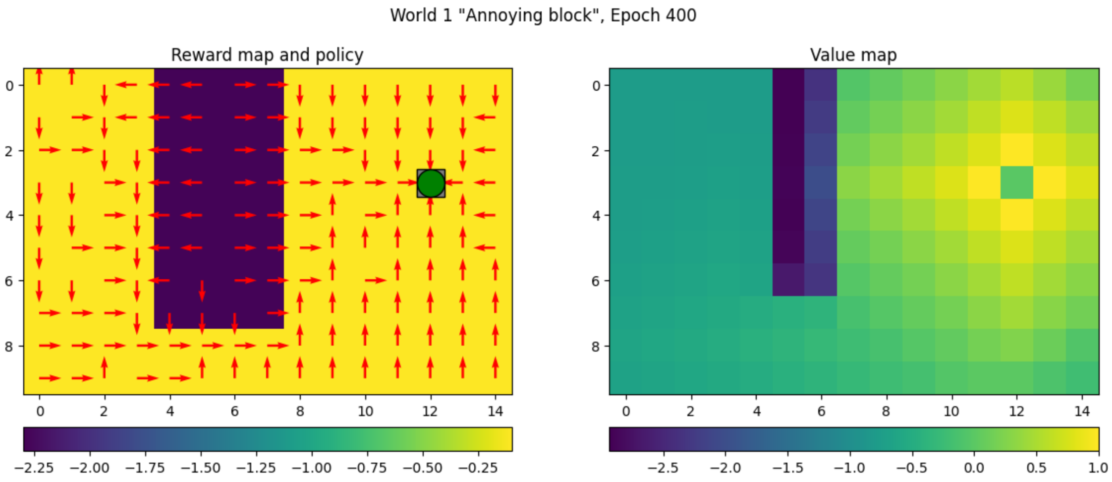
5. Since [:3, :],[7, :],[:, :3],[:, 12:15] area have a smaller negative reward -0.01 compare to -0.5 in other areas, so the optimal policy should utilize these areas and find the shortest path to the goal point.

0.5 4. Investigating the effects of hyperparameters

You will now design a series of experiments to show the impact of the three main hyperparameters - learning rate, discount factor, and exploration rate - in different environments. You are free to extend the experiments as you see fit in order to make your point in the discussions, but a recommended strategy is to try two extreme cases (low vs high values). For each parameter, there is one world in particular of the four you have already used where it is easy to show the effects we are looking for. Figuring out which worlds is part of the excercise.

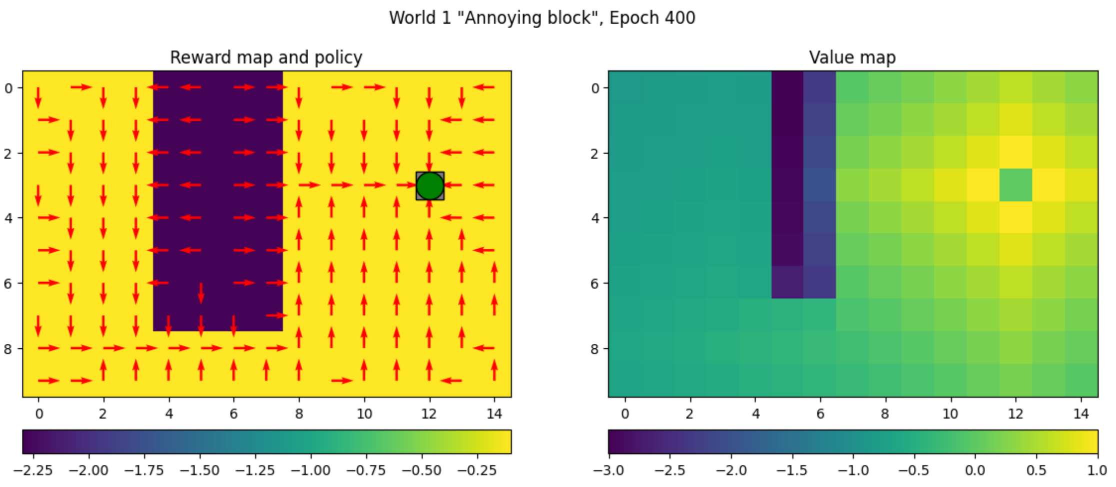
0.5.1 4.1 Learning rate

[35]:
```
# ---------------------------------------------
# === Your code here =========================
# ---------------------------------------------

W_LR = GridWorld(1)
Q_41_L = QLearning(W_LR, params={"LR": 0.2, "Gamma": 0.9, "Eps": 0.8, "Epochs":␣
↪400, "MaxSteps": 200, "DrawInterval": 100})

# =============================================
```



[36]:
```
# ---------------------------------------------
# === Your code here =========================
# ---------------------------------------------
```

```
Q_LR_H = QLearning(W_LR, params={"LR": 0.8, "Gamma": 0.9, "Eps": 0.8, "Epochs":␣
↪400, "MaxSteps": 200, "DrawInterval": 100})

# ============================================
```

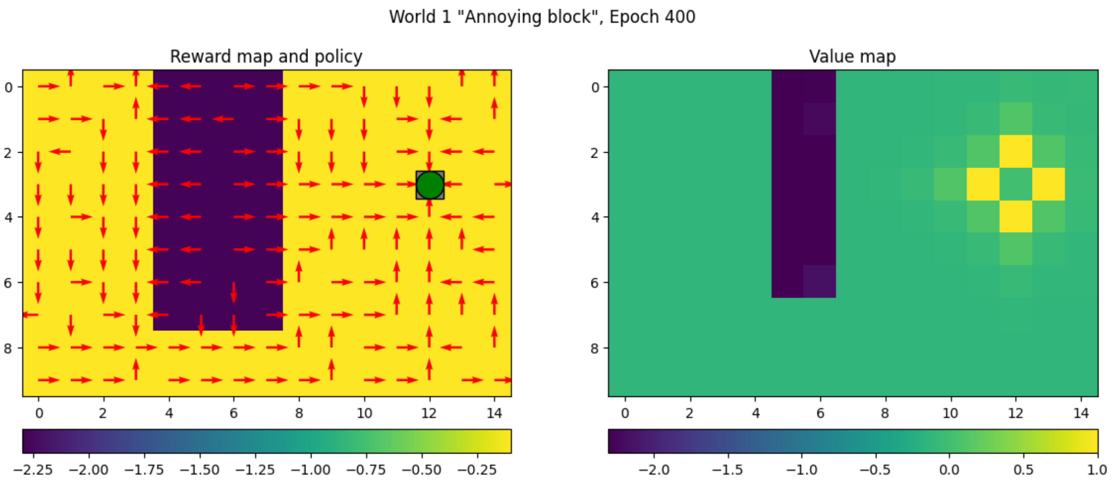


Question 6: Explain your experiment and results, and why you choose this world (your answers should be based on the output of the cells above).

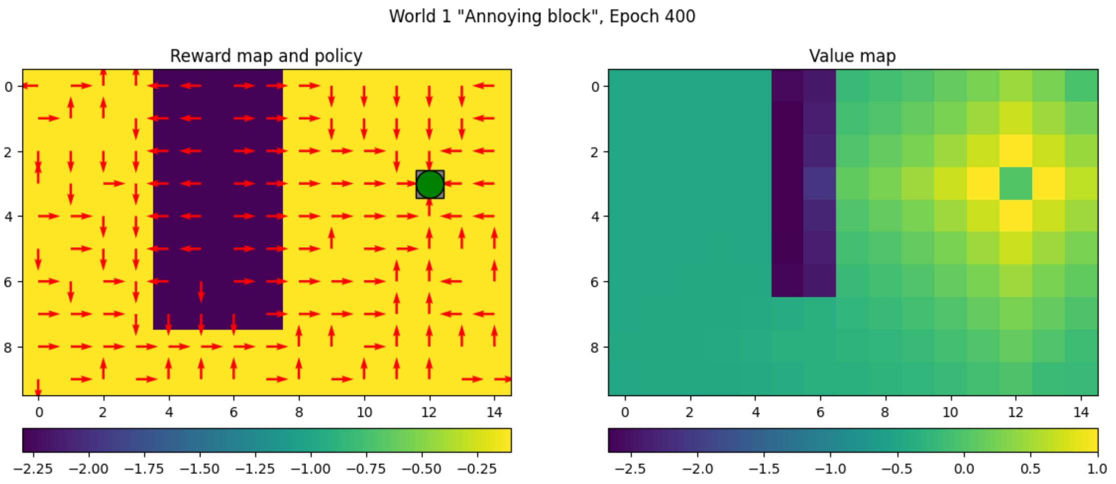
Answer: When learning rate is larger like 0.8, from the value map ,we can found that it get to the optimized value much faster than that of the small learning rate version. However, like what we saw in the World 4 example, if there are some random actions in the world, we should set the learning rate to a relatively small value to make sure the agent can explore the environment safely and find the optimal policy without influenced by those random actions.

0.5.2 4.2 Discount factor (gamma)

```
[46]: # --------------------------------------------
# === Your code here ========================
# --------------------------------------------

W_DF = GridWorld(1)
Q_DF_L = QLearning(W_DF, params={"LR": 0.5, "Gamma": 0.2, "Eps": 0.5, "Epochs":␣
↪400, "MaxSteps": 200, "DrawInterval": 100})

# ============================================
```



[49]:
```python
# -------------------------------------------
# === Your code here ========================
# -------------------------------------------

Q_DF_H = QLearning(W_DF, params={"LR": 0.5, "Gamma": 0.8, "Eps": 0.5, "Epochs":␣
↪400, "MaxSteps": 200, "DrawInterval": 100})

# ===========================================
```

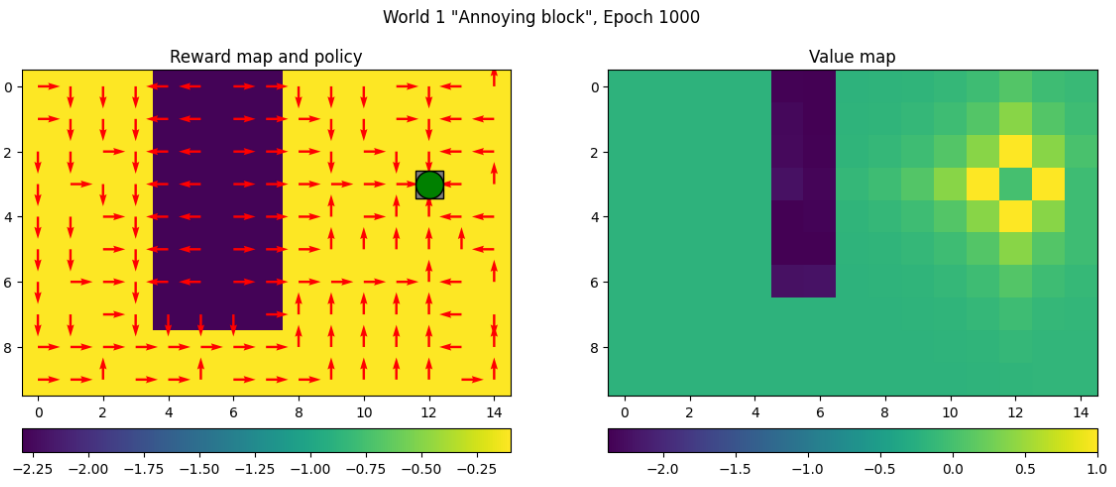


Question 7: Explain your experiment and results, and why you choose this world (your answers should be based on the output of the cells above).

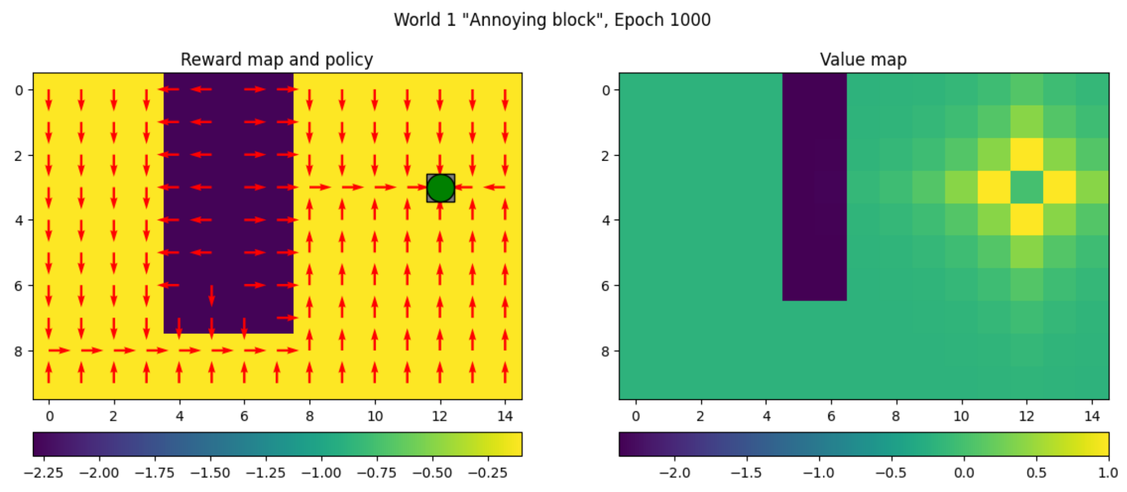
Answer: Discount factor can influence the agent's behavior. When we set Gamma to a higher value will influence more values in value map, which means it will encourage far-sighted behavior. How ever, when we set Gamma to a small value will influence less points value around the start point, which means it will encourage near-sighted behavior.

0.5.3 4.3 Exploration rate (epsilon)

[56]:
```
# ---------------------------------------------
# === Your code here ==========================
# ---------------------------------------------

W_ER = GridWorld(1)
Q_ER_L = QLearning(W_ER, params={"LR": 0.5, "Gamma": 0.5, "Eps": 0.1, "Epochs":␣
 ↪1000, "MaxSteps": 200, "DrawInterval": 100})

# =============================================
```



[55]:
```
# ---------------------------------------------
# === Your code here ==========================
# ---------------------------------------------

Q_ER_H = QLearning(W_ER, params={"LR": 0.5, "Gamma": 0.5, "Eps": 0.9, "Epochs":␣
 ↪1000, "MaxSteps": 200, "DrawInterval": 100})

# =============================================
```



Question 8: Explain your experiment and results, and why you choose this world (your answers should be based on the output of the cells above).

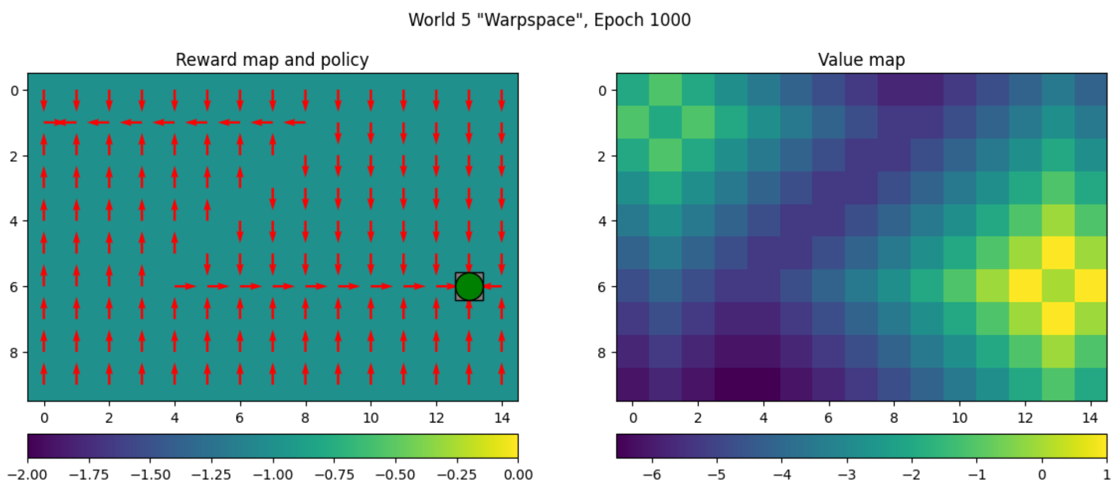
Answer: High eps will generate better result, compare 2 plots above, we find that high eps will make the optimal policy more reasonable and tidy, compare to the low eps version which is not so reasonable and tidy.

Meanwhile, high eps will make the agent explore the environment more and find the optimal policy and can help to avoid the agent get stuck in some local optimal.

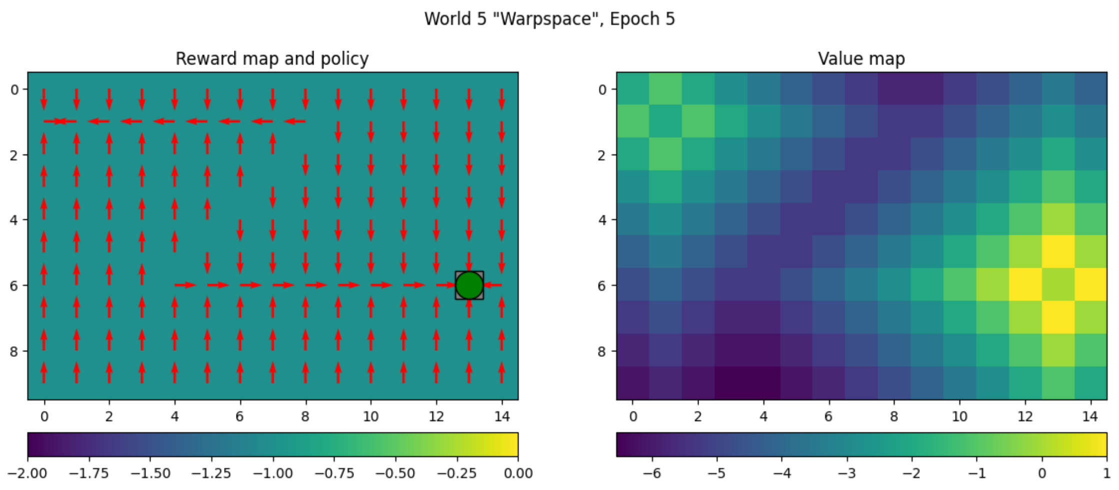
0.6 5. Optional worlds

You have now investigated the four most important GridWorlds in the lab, but we have also created some optional worlds (numbers 5 to 7) which you can try to solve. There is also World 8, but that is a special case, so scroll down a few cells if you are interested. Here is a brief description of World 5 to 7: - World 5, Warpspace: As the name suggests, in this world there is one tile in which the agent enters warpspace and imediatly moves to another specific location. How do you think this will affect the learning? - World 6, Torus: In this world, the opposite edges are connected together like a rolled-up paper. If you connect both the up-down and left-right edges, you get a mathematical shape called a torus which has no edges. This means that the closest path to the goal might not be obvious anymore. - World 7, Steps: This world is a staircase of increasing rewards (although still all negative). However, moving up the stairs towards higher rewards also puts the agent further from the goal. So what is the optimal choice, to go for the long path with higher rewards, or to sprint throught the low rewards towards the goal. This depends on the value of gamma.

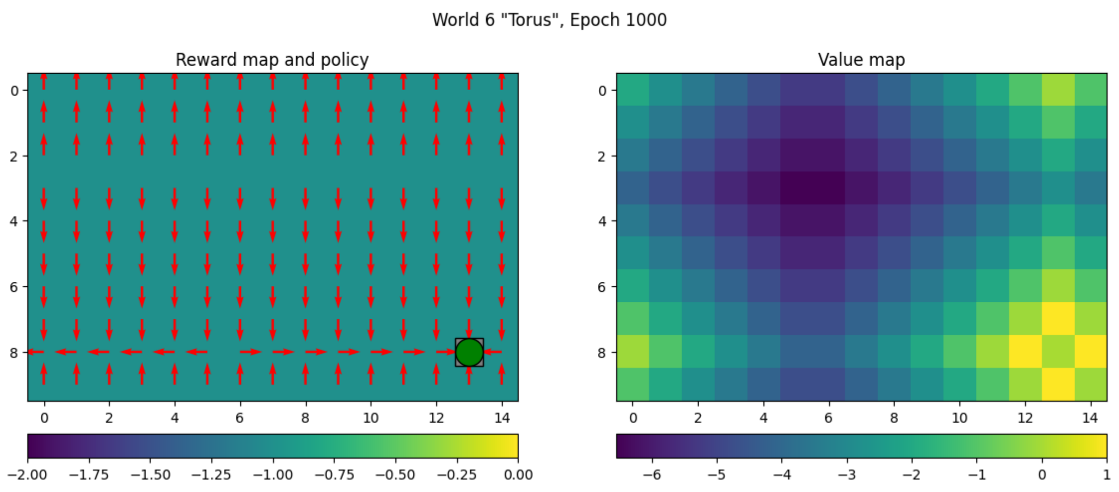
[57]:
```
WOpt = GridWorld( 5 )
QOpt = QLearning(WOpt, {"LR": 0.99, "Gamma": 0.9, "Eps": 0.9, "Epochs": 1000,␣
↪"MaxSteps": 200, "DrawInterval": 100})
```



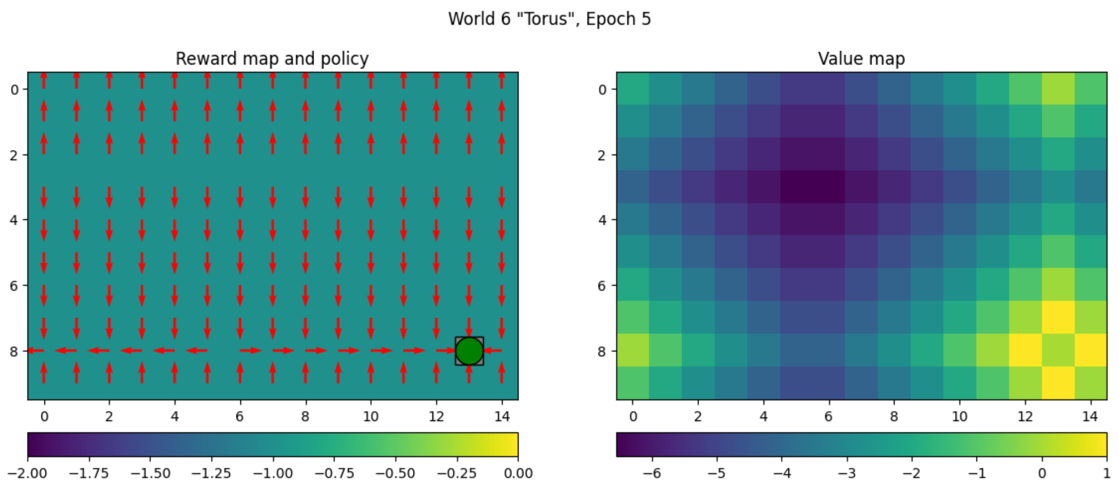
```
[58]: QLearningTest(W=WOpt, Q=QOpt, params={"Epochs": 5, "MaxSteps": 100})
```



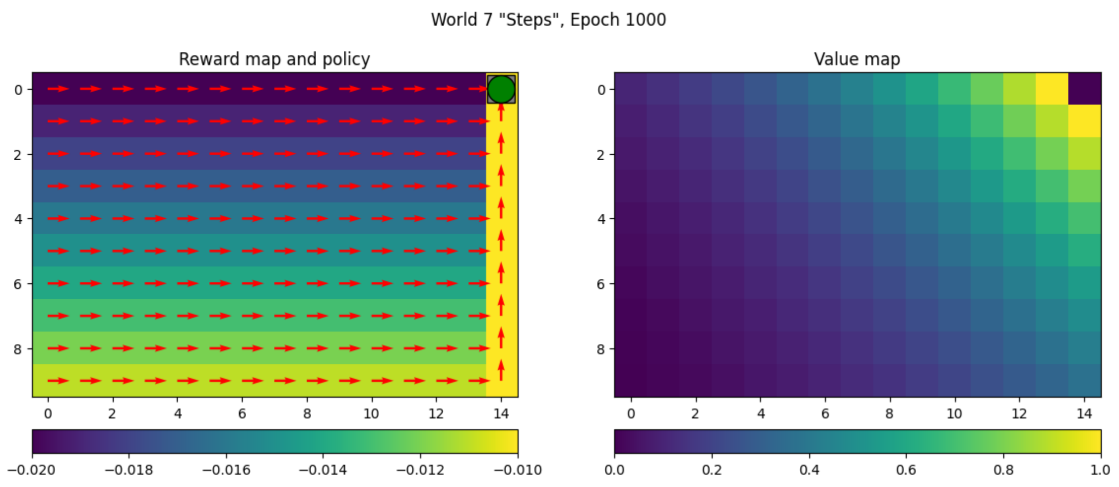
```
[59]: WOpt = GridWorld( 6 )
      QOpt = QLearning(WOpt, {"LR": 0.99, "Gamma": 0.9, "Eps": 0.9, "Epochs": 1000,␣
       ↪"MaxSteps": 200, "DrawInterval": 100})
```



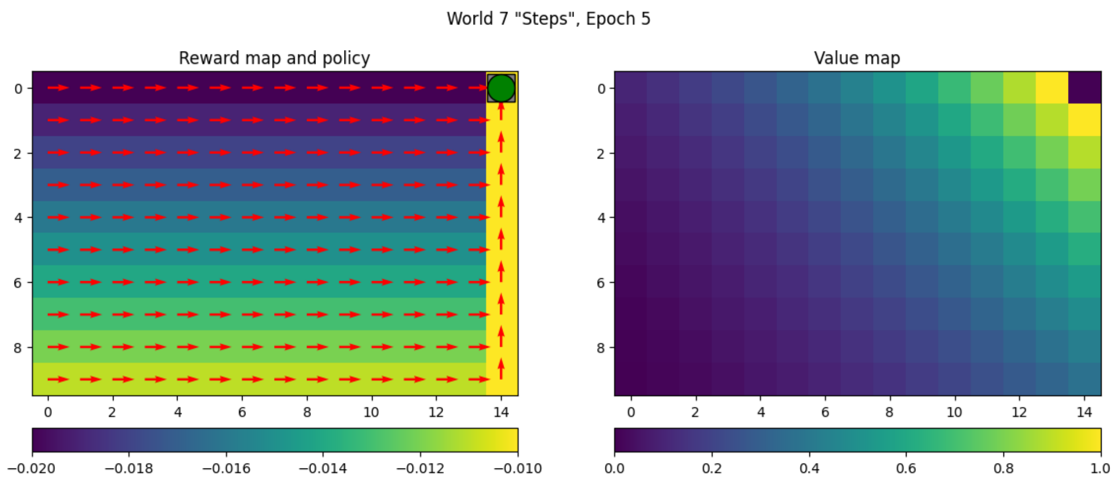
```
[60]: QLearningTest(W=WOpt, Q=QOpt, params={"Epochs": 5, "MaxSteps": 100})
```



```
[61]: WOpt = GridWorld( 7 )
      QOpt = QLearning(WOpt, {"LR": 0.99, "Gamma": 0.9, "Eps": 0.9, "Epochs": 1000,␣
       ↪"MaxSteps": 200, "DrawInterval": 100})
```



```
[62]: QLearningTest(W=WOpt, Q=QOpt, params={"Epochs": 5, "MaxSteps": 100})
```



0.6.1 5.1: World 8

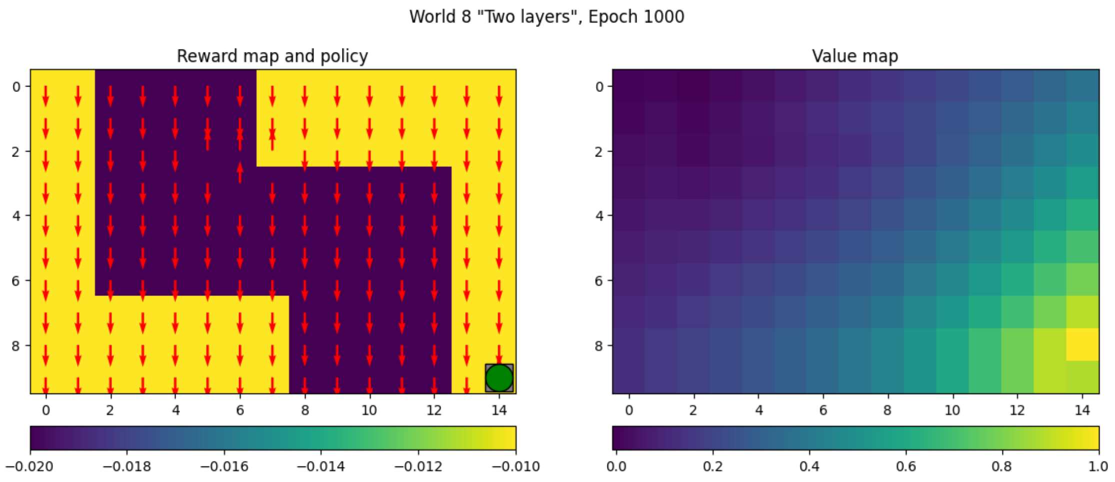
So far, every world has been a 2D-grid (y and x dimensions), and the four actions have been the same in every world. It has therefore been possible to write the code with this in mind, probably resulting in code where you index Q for example with `Q[s[0], s[1], a]` for a given state `s` and action `a`. However, it is possible to slightly rewrite the code to be independent of the number of dimensions in the state space, which means that we can then explore much more interesting worlds. It is also a nice excercise in how to write code that is general and modular. The way to do this is to index Q in the following way: `Q[(*s,a)]`. It's perfectly fine if you want to consider this as "python magic", but for the interested here is an explaination.

The state `s` is a tuple, for example `(3,6)`. A quirk in python is that tuples can be used to index into arrays, with each value in the tuple indexing separate dimensions in the array. For example, if

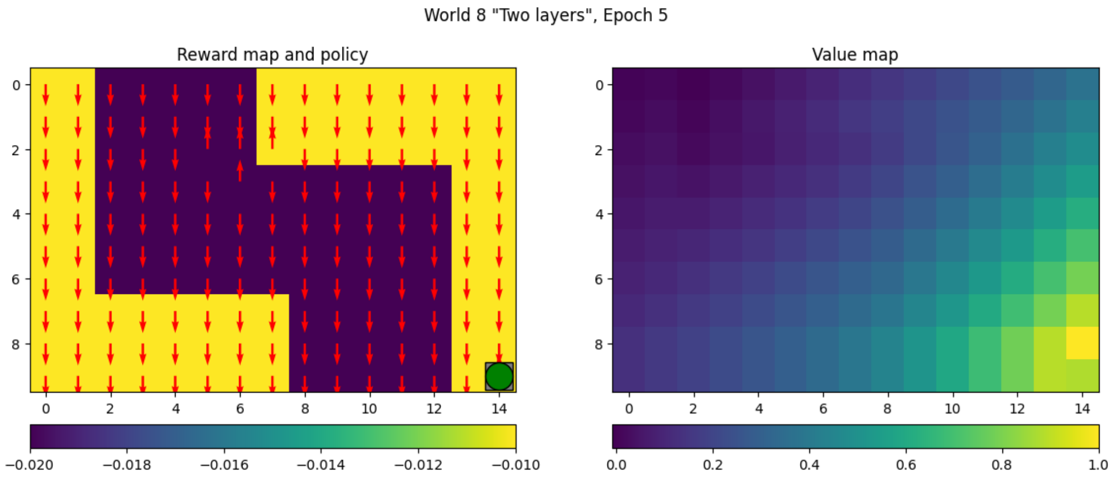
Q is a 10x15x4 array, then `Q[(3,6)]` will return the vector of four values in Q that are in the 3rd row and 6th column (i.e. all the action values for state `s = (3,6)`). The problem is that we want to access the Q-value of a specific action when updating with a new reward. One might assume that `Q[s,a]` would work, but this now works differently since we explicitly index Q with not only a tuple. The solution is to remake a tuple that contains both `s` and `a`, and then index Q with this. We can do this by first unpacking the state tuple by calling `*s`, then creating a new tuple with `(*s,a)`, containing both the state and action. For example, if `s = (3,6)` and `a = 2`, then `(*s,a) = (3,6,2)`. We then use this tuple to index into Q as `Q[(*s,a)]`.

With this change to the implementation, we can for example extend the world to a 3D-grid, and your code should work the same. Let's try it in World 8, where the agent has the choice of moving between two floors of the map. This is shown as diagonal up or diagonal down arrows.

[63]:
```
W8 = GridWorld(8)
Q8 = QLearning(W8, {"LR": 0.99, "Gamma": 0.9, "Eps": 0.9, "Epochs": 1000,␣
↪"MaxSteps": 200, "DrawInterval": 100})
```



[64]:
```
QLearningTest(W=W8, Q=Q8, params={"Epochs": 5, "MaxSteps": 100})
```


World 8 "Two layers", Epoch 5
Reward map and policy
Value map
0
2
4
6
8
10
12
14
−0.020
−0.018
−0.016
−0.014
−0.012
−0.010
0.0
0.2
0.4
0.6
0.8
1.0