# 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.

## Setup

In [4]:
```
# 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)
```

## *! 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.

---

# 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**.

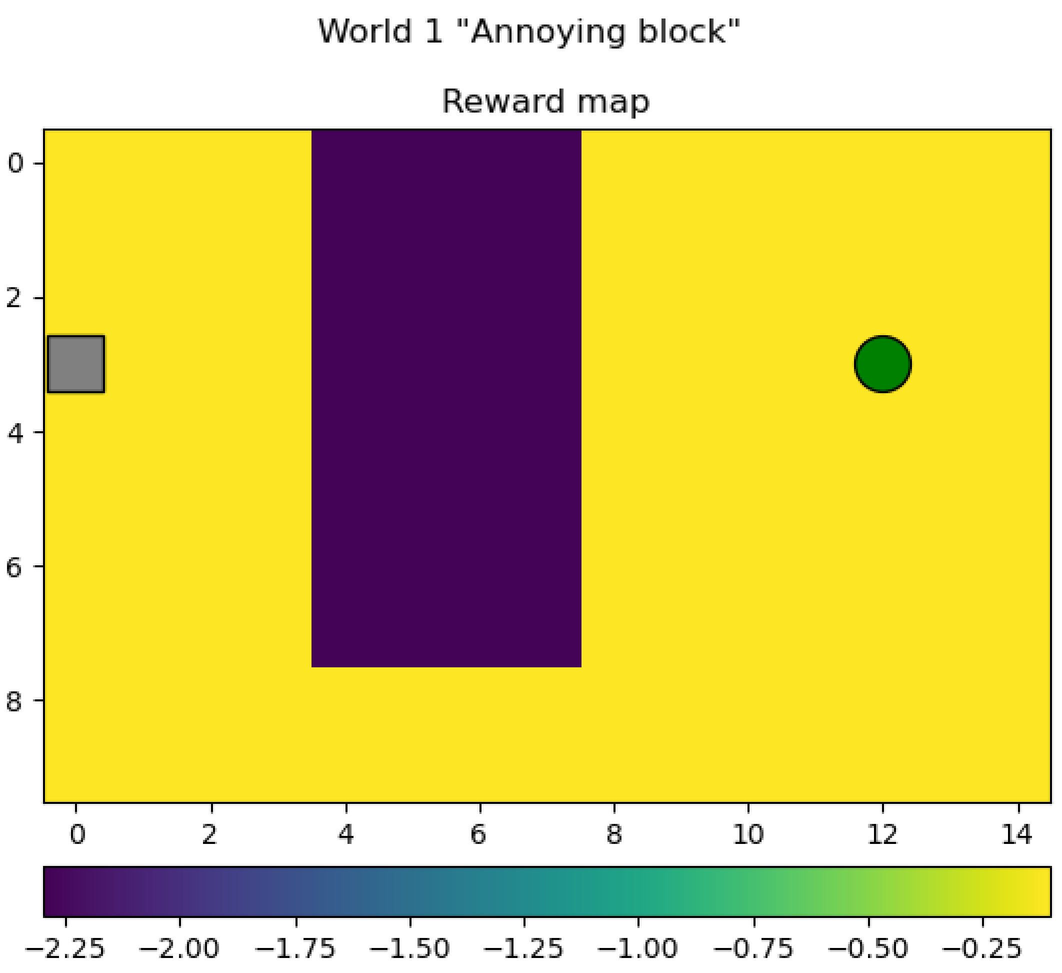
RL is a very broad concept and many different algorithms have been deviced based on these general concepts of actions and rewards. Perhaps the main advantage of RL over other machine learning techniques is that we do not explicitly tell the model what the right answer is (like we have done in the previous 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.

## 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.

In [5]:
```
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:

As the Reinforcement Learning (RL) is looking for the state-action pairs that maximize the reward. The negative rewards are used to act as a penalty or cost, so the learning system will try to avoid those state with larger negative reward. Also, negative reward can help learning system to avoid stucking in a loop, as in the loop the reward will keep on decreasing. In contrast, if the reward is postive, the system might stay with in the loop as the reward keep on going is what the system prefer.

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:

In [6]:
```python
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.

In [7]:
```python
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 {re

# 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 {re

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

```
State initialized to (1, 0).
Action 'Down' was valid and gave a reward of -0.1.
State is (2, 0) and is not terminal.
Action 'Right' was valid and gave a reward of -0.1.
State is (2, 1) and is not terminal.
```

---

## 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:

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

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 $\gamma$, the future value is weighted higher, which is why we say that this optimizes for long-term rewards.

### 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!

In [8]:
```python
def QLearning(World, params={}):

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

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

    #get parameters
    alpha = params["LR"]
    gamma = params["Gamma"]
    ExpRate = params["ExpRate"]

    # Initialize the Q-matrix (use the size variables above)
    Q = np.zeros((WSize[0], WSize[1],NA)) # the dimension will be the size of the w

    for i in range(params["Epochs"]):
        World.init()
        state = World.getState()[0] # get a starting state
        total_reward = 0


        # Limiting the number of steps in an epoch prevents getting stuck in infini
        for j in range(params["MaxSteps"]):
            #Epsilon greedy
            if np.random.random() < ExpRate:
                action = np.random.choice(A)
            else:
                action = A[np.argmax(Q[state[0], state[1]])]


            action_idx = A.index(action) # since the action is a string, we need to
            #print(action_to_idx)

            # Do the action
            valid, reward = World.doAction(action) # we actually don't need to chec
            state_next, end_check = World.getState()


            # Update Q, p28 lecture slide
            next_max_q = np.max(Q[state_next[0], state_next[1], :])
            Q[state[0], state[1], action_idx] = (1 - alpha) * Q[state[0], state[1]


            # Update state
            total_reward += reward
            state = state_next


            # Break if episode terminated
            if end_check:
                break


        # Update plots with regular intervals
        if ((i+1) % params["DrawInterval"] == 0) or (i == params["Epochs"]-1):
            World.draw(epoch=(i+1), Q=Q)

    # ============================================


    return Q
```

## 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.

In [9]:
```python
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

            q_vals = Q[s[0], s[1], :] #get the Q value of current state
            action_idx = np.argmax(q_vals) #compare the Q value of different actio
            action = A[action_idx]
            W.doAction(action)

            # ============================================

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

            # Check if goal
            if isTerm:
                break
```

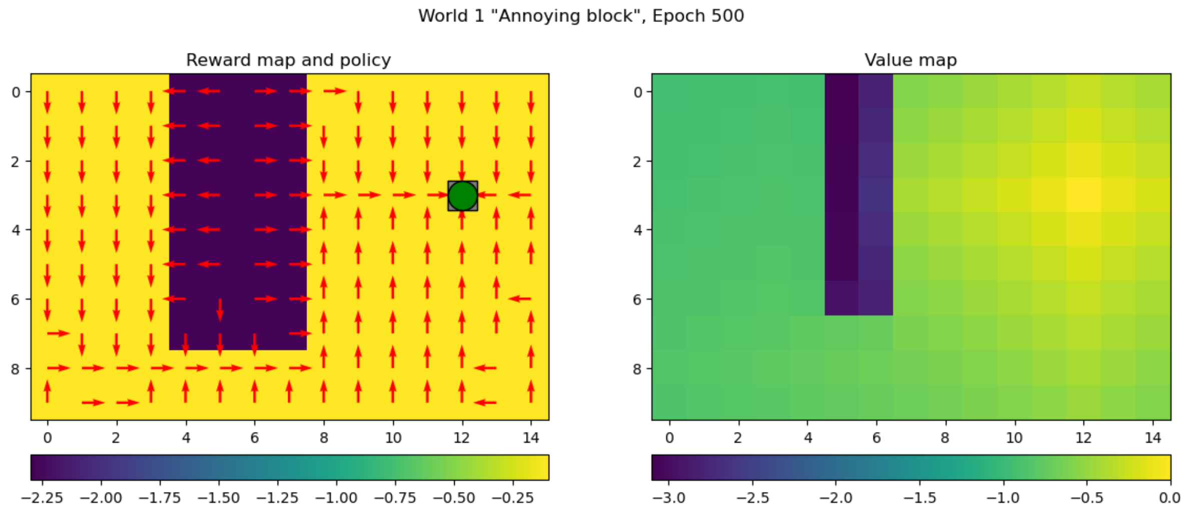
## 3. Optimizing the different worlds

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

### 3.1 GridWorld 1

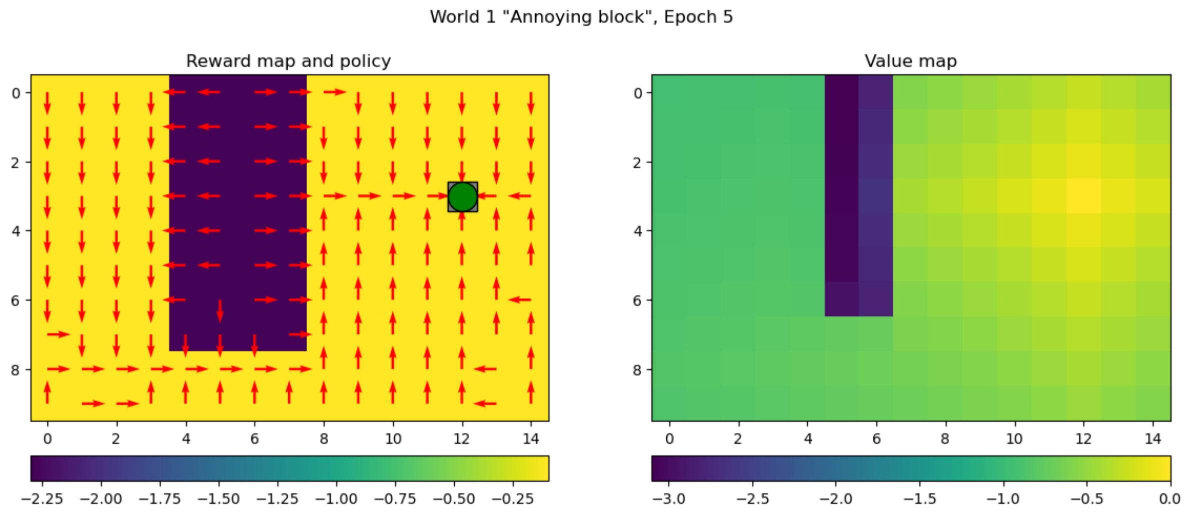
We start with the simplest of the worlds, "Annoying block". The policy should converge without much difficulty, so use this as a test to see if your implementaion is correct. If you use a good set of hyperparameters, you can expect a rather neat policy in about 1000 epochs.

In [21]:
```python
W1 = GridWorld(1)

# --------------------------------------------
# === Your code here =========================
# --------------------------------------------
params = {"Epochs": 500, "MaxSteps": 200, "LR": 0.5 ,"Gamma": 0.9, "ExpRate":  0.9
Q1 = QLearning(W1, params)

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



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

```
In [22]: QLearningTest(W=W1, Q=Q1, params={"Epochs": 5, "MaxSteps": 100})
```



## 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:

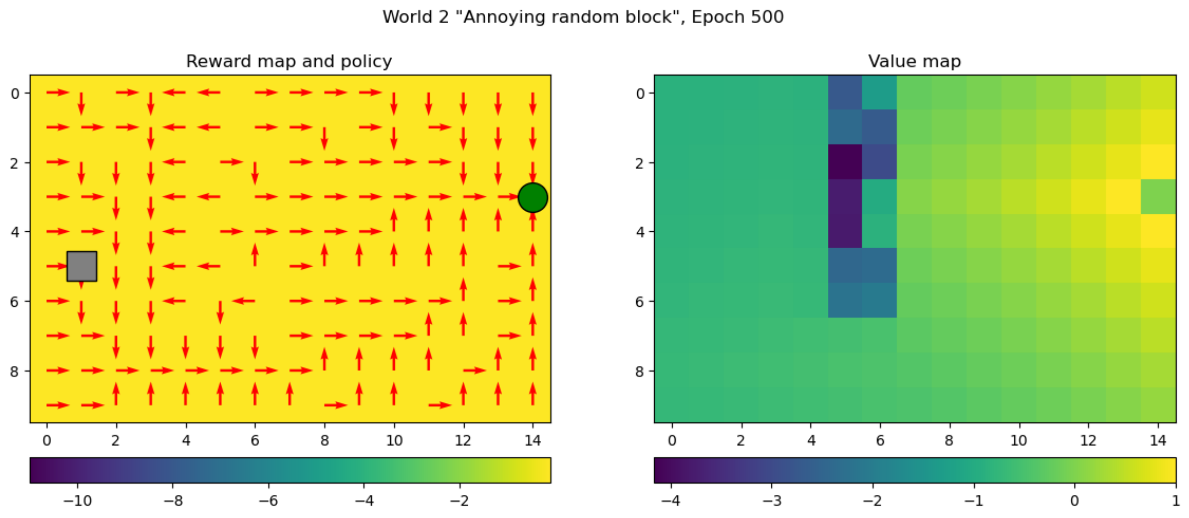
The World 1 has a fixed obstacle in it, the agent's goal is to avoid this block and get to the terminal point with shortest distance.
We set the learning rate = 0.5. Generally speaking, this world is rather simple, so using learning rate ranges from 0.1 to 0.5 all works well. The choice of learning rate 0.5 is the output policy looks more uniform.

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

## 3.2 GridWorld 2

In [355…

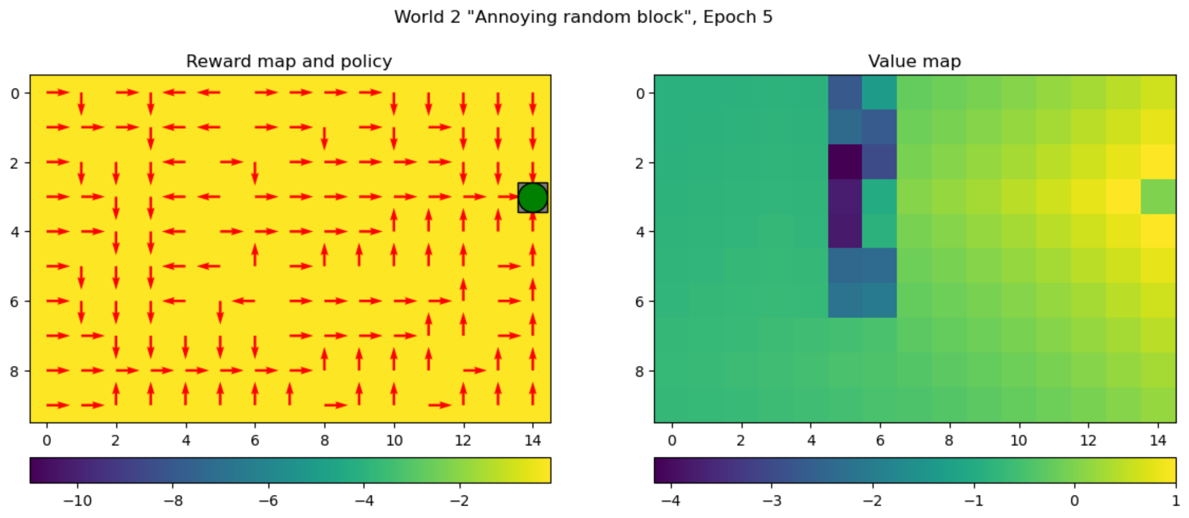
```
W2 = GridWorld(2)

# ------------------------------------------
# === Your code here ========================
# ------------------------------------------
params = {"Epochs": 500, "MaxSteps": 200, "LR": 0.3 ,"Gamma": 0.9, "ExpRate":  0.9
Q2 = QLearning(W2, params)

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



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

In [357…

```
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:

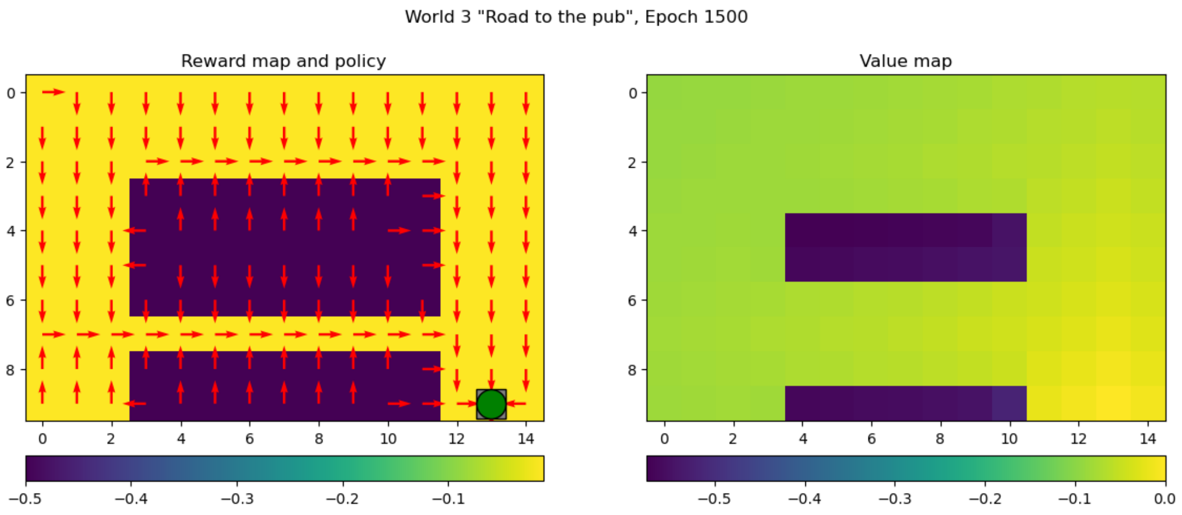
World2 is similar to World 1 but the obstacle will randomly appeared, the goal for the agent is try to reach the terminal point but now need to consider if it want to go thru the obstacle area.
The reinforcement learning is good for this task since it can learn from the previous experiance and can be used to learn how to get the long term reward. This behavior can help the agent to avoid the randomlly appeared block as it will "memorized" where the random block is. As we want the model to learn the randomly appeared obstacle, we decrease the learning rate so the model puts more emphasis on already learned experience, but we don't set it too low so it can still take the risk of going thru the obstacle.

Compared the policy of World 2 to World 1, we would say the policy will be slightly different. As the obstacle appears only 20% of the time(according to the class), the model might take account for the randomness of it and take the risk for going stright towards the terminal point. Therefore, if the agent start on the left, it will not take the risk to go thru the obstacle area, but if the agent start within the obstacle area, it might decide to go striaght toward's the terminal point.

## 3.3 GridWorld 3

In [359…

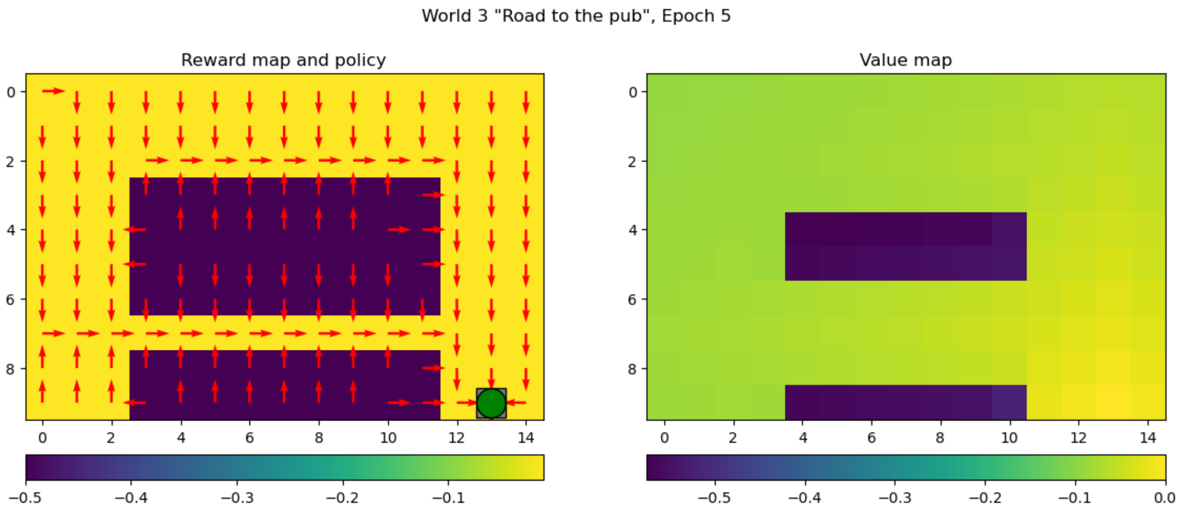
```
W3 = GridWorld(3)

# ---------------------------------------------
# === Your code here ==========================
# ---------------------------------------------
params = {"Epochs": 1500, "MaxSteps": 200, "LR": 0.3 ,"Gamma": 0.9, "ExpRate":  0.
Q3 = QLearning(W3, params)

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



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

In [360…

```
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?

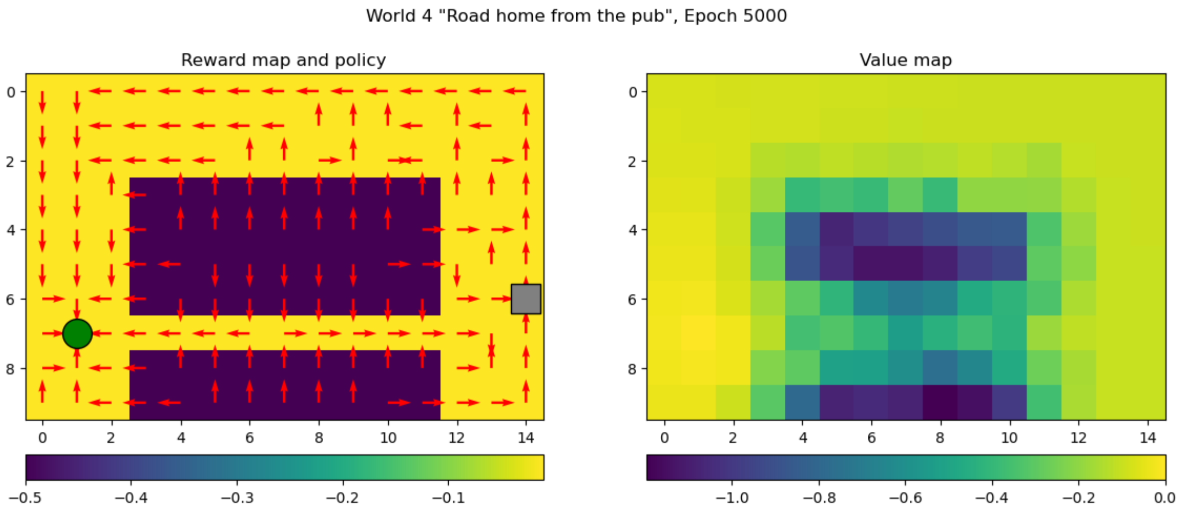
## Answer:

World 3 is also a rather simple world like world 1, but there is a pathway inside the obsticle, and the goal of the agent is to reach to the termianl point using pathway or get around the obsticle from the top side. Compare to world 1, the learning algorithm is simlar, but the model for world 3 should learn to use the pathway as short cut to get higher overall reward. It is possible to get a good policy in this world. we will say the exploration rate is important for this world, as it encourage the agent to explore uncharted area. The policy with low exploration rate will lead to the agent avoiding the obsticle but not moving towards to terminal point.

## 3.4 GridWorld 4

In [289…

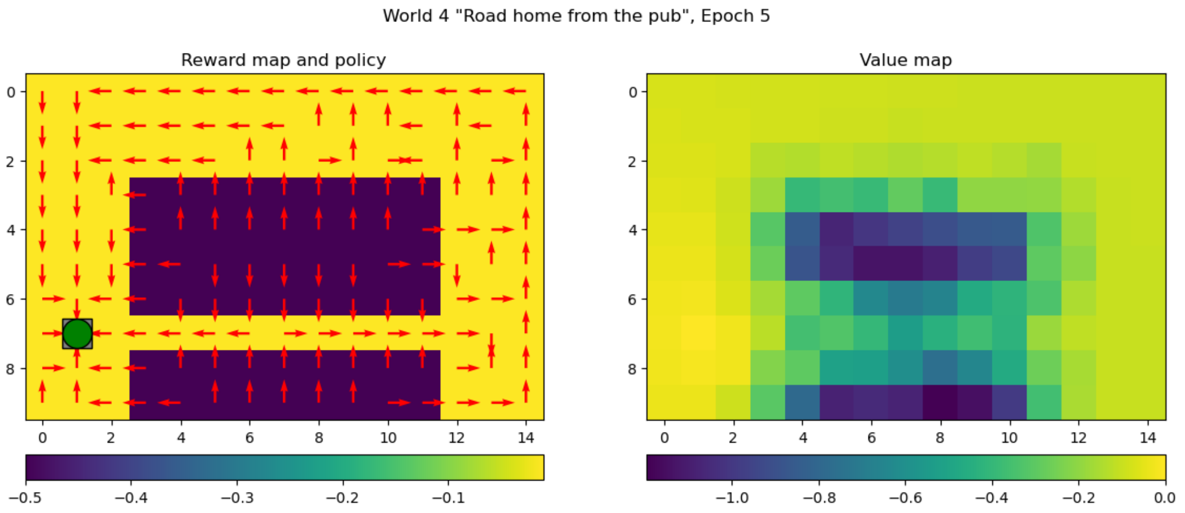
```
W4 = GridWorld(4)

# ---------------------------------------------
# === Your code here ========================
# ---------------------------------------------
params = {"Epochs": 5000, "MaxSteps": 200, "LR": 0.1 ,"Gamma": 0.9, "ExpRate":  0.
Q4 = QLearning(W4, params)

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



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.

In [290…

```
QLearningTest(W=W4, Q=Q4, params={"Epochs": 5, "MaxSteps": 150})
```



## 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:

The world 4 is similar to world 3. However, it contains the trick of randomly walk (drunk) after the action has been made. The agent's goal is to learn the way to get to the termianl point despite the random walk behavior.

The learning rate should be low, since it is important for the model to memorize the path to reach the terminal point even the agent is "drunk".

We expect the optimal route to be similar as World 3, the agent will take the path way in between the obstacle since it is the shorter route. However, the actual optimal policy looks like the agent will try to avoid the path way. This can be explain by the randomness will cause the agent hit the wall within the path way and get more penalties, so the model learn that it will be better to avoid the pathway instead.

---

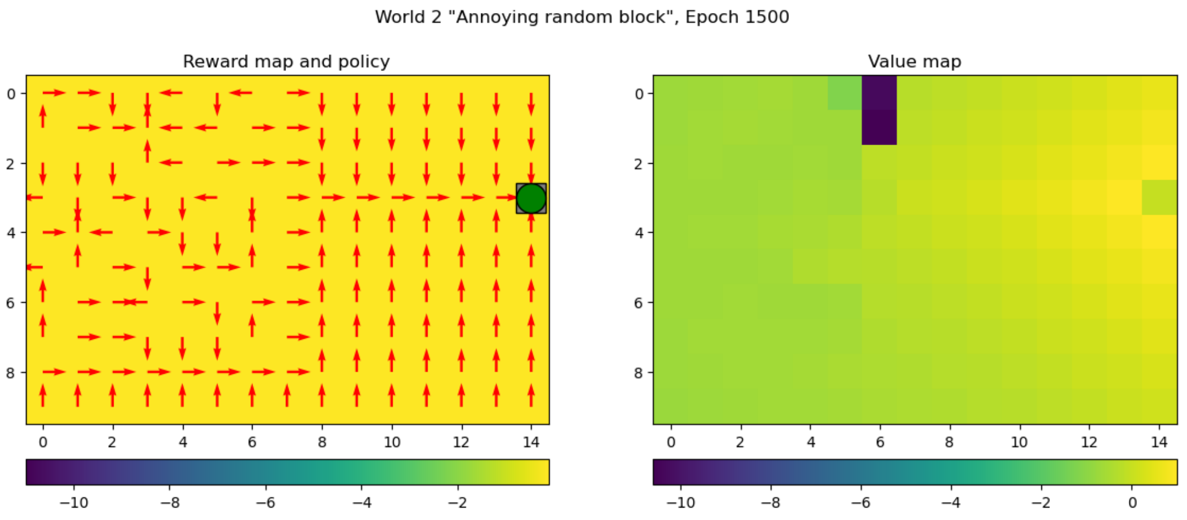
## 4. Investigating the effects of hyperparameters

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.

### 4.1 Learning rate

In [336…

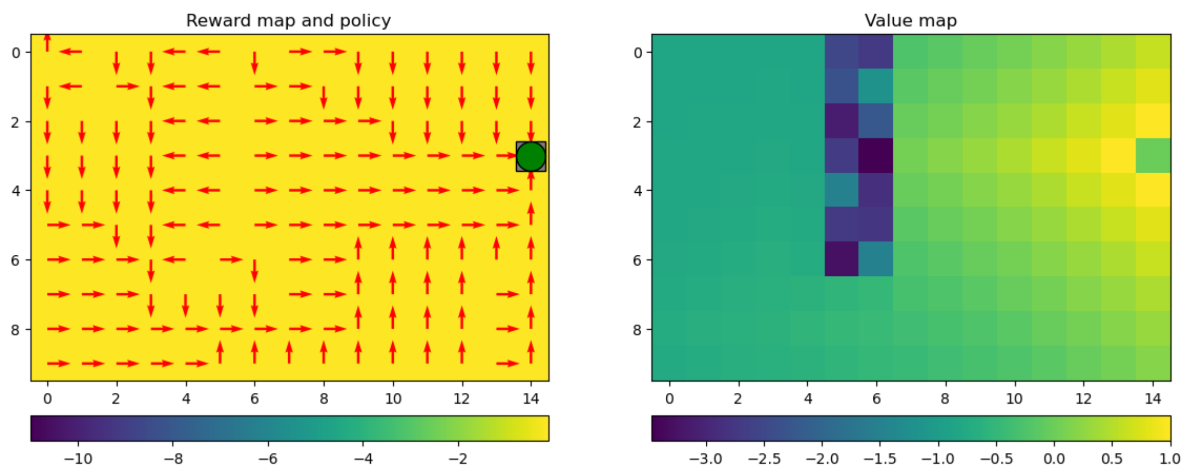
```
# --------------------------------------------
# === Your code here ========================
# --------------------------------------------

W_LR = GridWorld(2)
Q_41_H = QLearning(W_LR, params={"LR": 0.95 ,"Gamma": 0.9, "ExpRate": 0.9, "Epochs

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



In [337…

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

Q_LR_L = QLearning(W_LR, params={"LR": 0.05 ,"Gamma": 0.9, "ExpRate": 0.9, "Epochs

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

World 2 "Annoying random block", Epoch 1500



## 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:

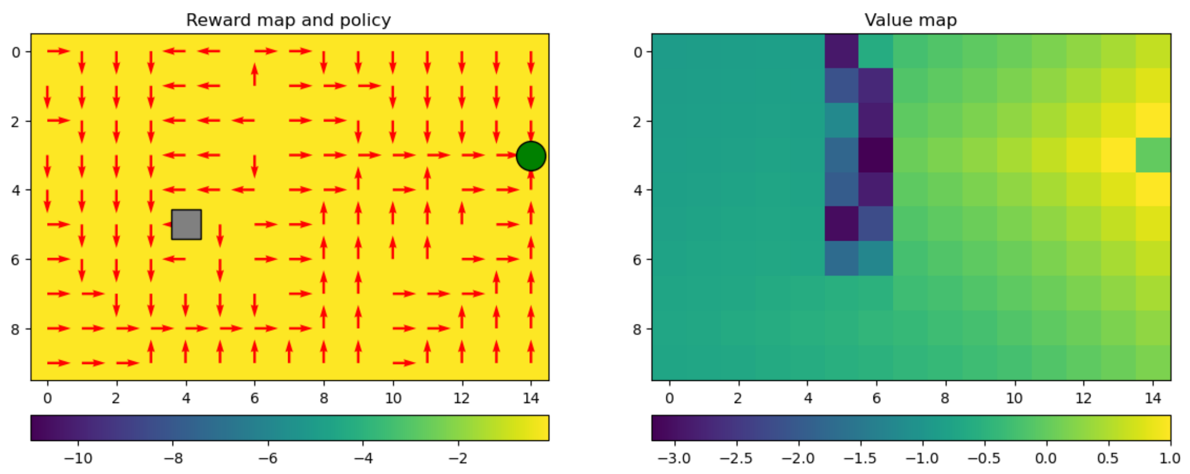
We choose World2 for this experiment, since the nature of the randomness requires the model to learn where the obstacle is. As observed from the above cell, it can be seen that the policy produce by large learning rate is tend to ignore the obstacle and the lower learning rate will try to avoid it like World1 doess.

## 4.2 Discount factor (gamma)

In [338…

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

W_DF = GridWorld(2)
Q_DF_H = QLearning(W_DF, params={"LR": 0.1 ,"Gamma": 0.9, "ExpRate": 0.9, "Epochs"

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

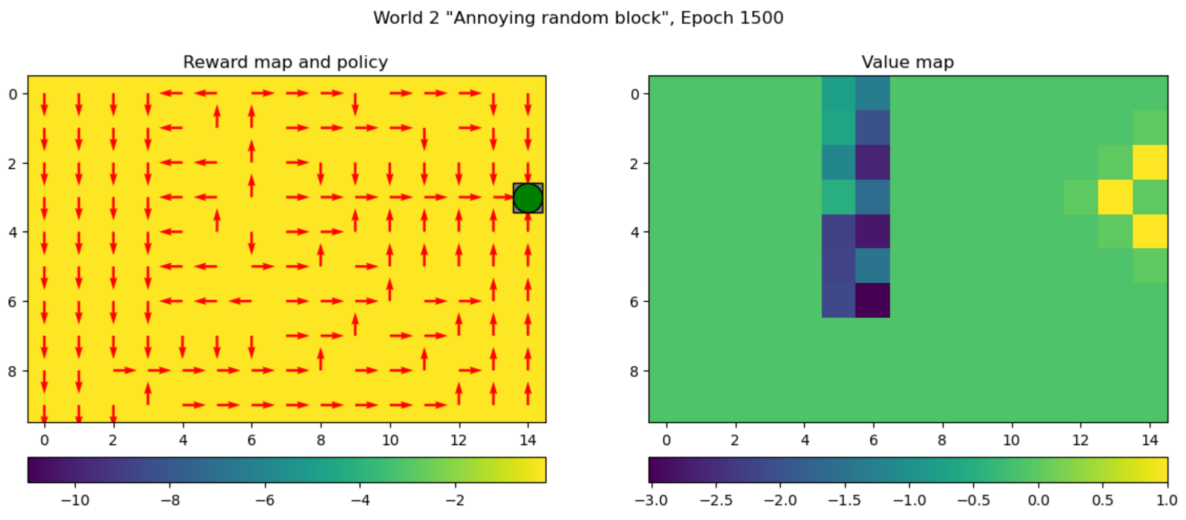
World 2 "Annoying random block", Epoch 1500



In [339…

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

```
Q_DF_L = QLearning(W_DF, params={"LR": 0.1 ,"Gamma": 0.1, "ExpRate": 0.9, "Epochs"

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



## 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).

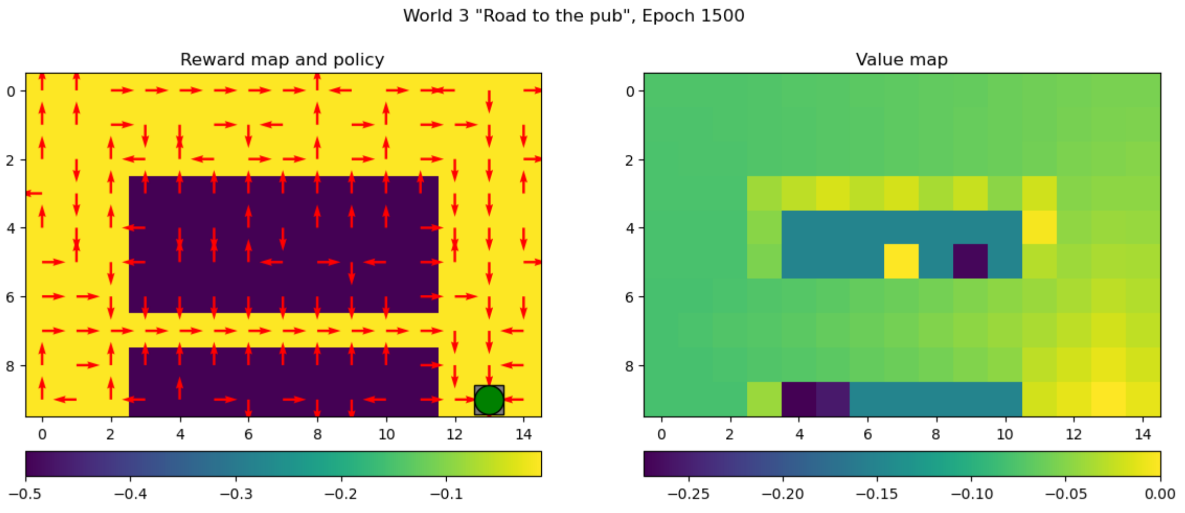
## Answer:

We choose world2 for this experiment as well, as we can see in the value map. The lower gamma will result in similar value across the non-obstacle area. Moreover, according to the policy the agent seems to form a unnecessary path toward top within the obstacle area, and on the lefthand side the agent try to move towards to the lower-left cornor. The reason is the low gamma model doesn't seek the long-term reward so it will consider lower left cornor being a good end result. Also, the combination of low gamma and low learning rate make the model being confused with the upper side of the obstacle and therefore form the uunnecessary path.

## 4.3 Exploration rate (epsilon)

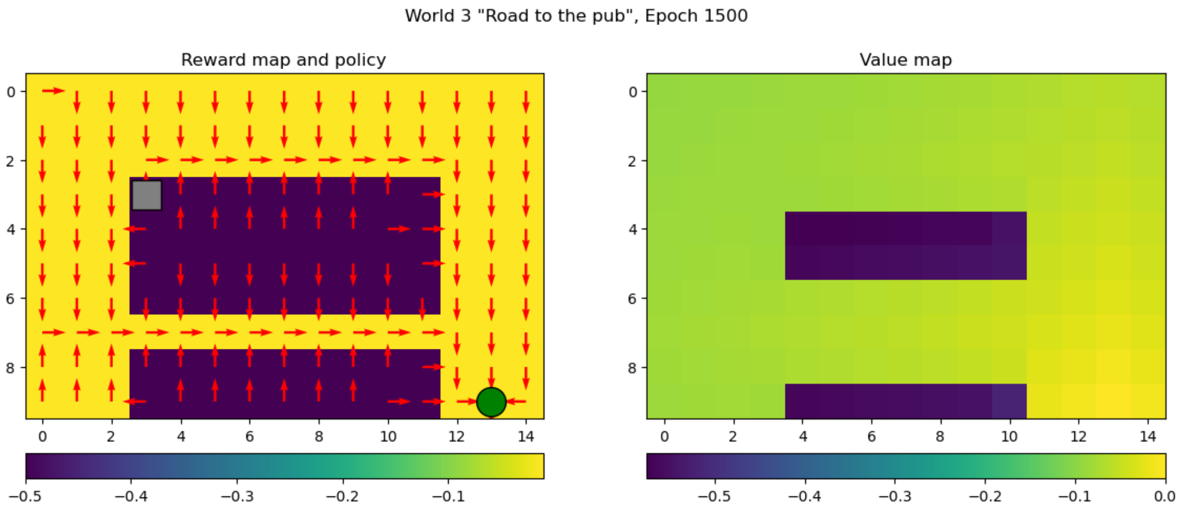
In [345…

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

W_ER = GridWorld(3)
params = {"Epochs": 1500, "MaxSteps": 200, "LR": 0.3 ,"Gamma": 0.9, "ExpRate":  0.
Q_ER_L = QLearning(W_ER, params)

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



In [346…

```python
# --------------------------------------------
# === Your code here =========================
# --------------------------------------------

params = {"Epochs": 1500, "MaxSteps": 200, "LR": 0.3 ,"Gamma": 0.9, "ExpRate":  0.
Q_ER_H = QLearning(W_ER, params)

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



## 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:

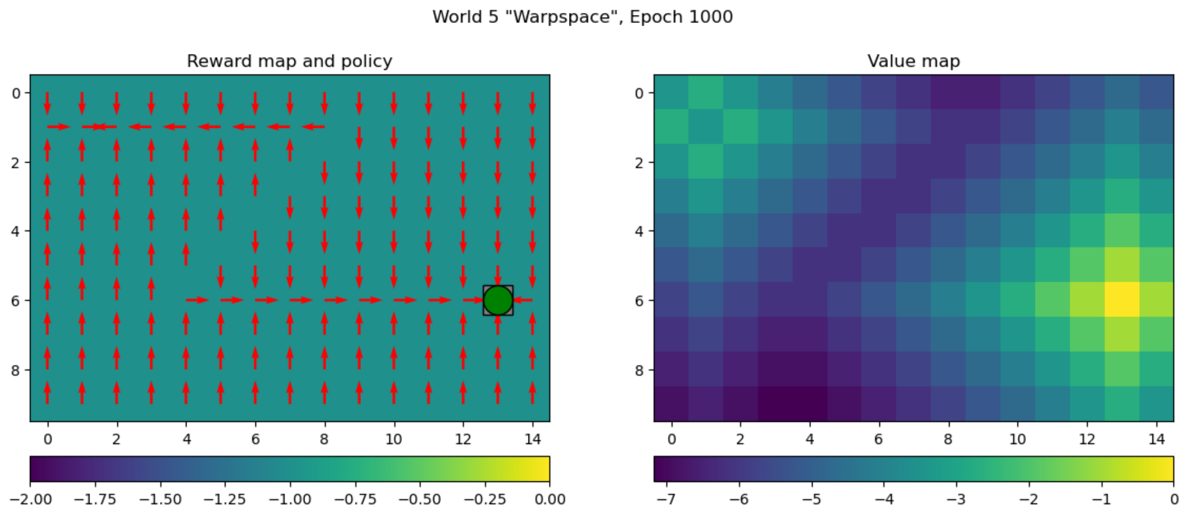
We choose world 3 for this test as this world need the agent to explore more to find the way toward terminal point, and also there is no randomness will affect it. As we can see in the low-epsilon output, the model did not converge in to optimal policy, the policy seems only try to avoid the obstacle and some of it move twards to edge. This is because with low-epsilon, the model will not try to explore unchartted part of the world and take the same route it has learn for each episode. This end up with non-optimal policy.

## 5. Optional worlds

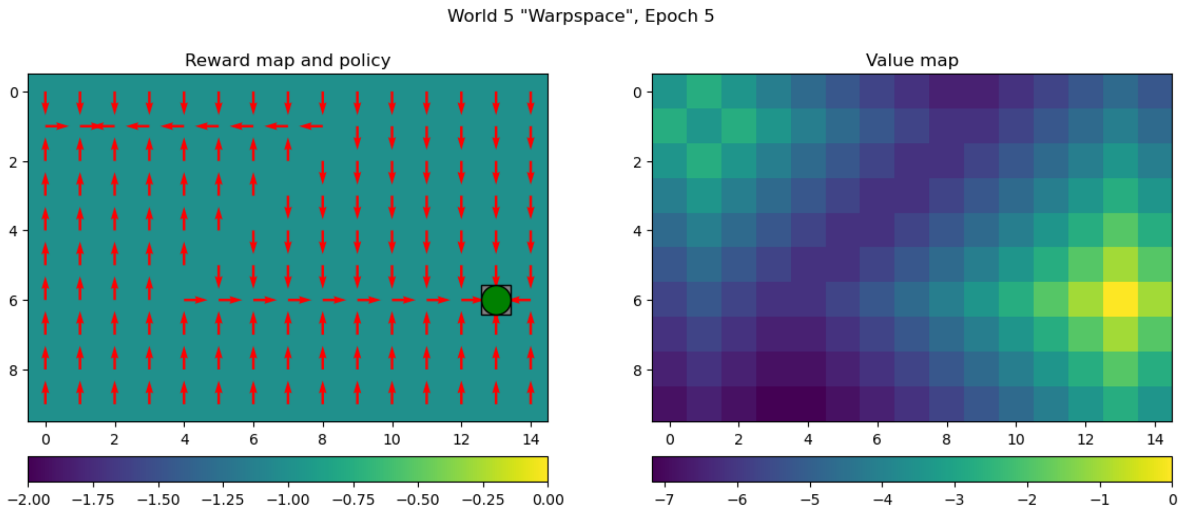
You have now investigated the four most important GridWorlds in the lab, but we have also created some optional worlds (numbers 5 to 7) which you can try to solve. There is also World 8, but that is a special case, so scroll down a few cells if you are interested. Here is a brief description of World 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.

In [348…
```
WOpt = GridWorld( 5 )
QOpt = QLearning(WOpt, {"LR": 0.99, "Gamma": 0.9, "ExpRate": 0.9, "Epochs": 1000,
```



In [349…
```
QLearningTest(W=WOpt, Q=QOpt, params={"Epochs": 5, "MaxSteps": 100})
```



## 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.

In [ ]:
```
W8 = GridWorld(8)
Q8 = QLearning(W8, {"LR": 0.99, "Gamma": 0.9, "ExpRate": 0.9, "Epochs": 1000, "Max
```

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