Advanced Machine Learning
LAB 3: Reinforcement Learning

Mohammed Bakheet (mohba508)

07/10/2020

Contents

Question 1: Q-Learning **2**
Greedy Policy Function 2
Epsilon Greedy Function 2
Transition Model 3
Q Learning Function 4
Environment A: 4
Answers: 4
Environment B: 8
Answer: 8
Environment C 19
Answer: 19

Question 2: Reinforce: **23**
Visualizing the environment: 23
Transition Model: 23
Deep Policy Distribution: 23
Deep Policy 24
Deep Policy Training 24
Reinforcement Episode 25
Environment D: 26
Answer: 26
Environment E: 45
Answer: 45

References **58**

Appendix **58**

Question 1: Q-Learning

The file RL Lab1.R in the course website contains a template of the Q- learning algorithm. 1 You are asked to complete the implementation. We will work with a grid-world environment consisting of H × W tiles laid out in a 2-dimensional grid. An agent acts by moving up, down, left or right in the grid-world. This corresponds to the following Markov decision process:

- State space: S = {(x, y) | x ∈ {1, . . . , H}, y ∈ {1, . . . , W }}.
- Action space: A = {up, down, left, right}.

Additionally, we assume state space to be fully observable. The reward function is a deterministic function of the state and does not depend on the actions taken by the agent. We assume the agent gets the reward as soon as it moves to a state. The transition model is defined by the agent moving in the direction chosen with probability (1 - β). The agent might also slip and end up moving in the direction to the left or right of its chosen action, each with probability $\beta/2$. The transition model is unknown to the agent, forcing us to resort to model-free solutions. The environment is episodic and all states with a non-zero reward are terminal. Throughout this lab we use integer representations of the different actions: Up=1, right=2, down=3 and left=4.

Greedy Policy Function

```
GreedyPolicy <- function(x, y){

  # Get a greedy action for state (x,y) from q_table.
  #
  # Args:
  #   x, y: state coordinates.
  #   q_table (global variable): a HxWx4 array containing Q-values for each state-action pair.
  #
  # Returns:
  #   An action, i.e. integer in {1,2,3,4}.

  # Your code here.
  rewards = q_table[x,y,1:4]
  action = 0
  max_reward = which(rewards == max(rewards))

  if (length(max_reward)>1){
    action = sample(max_reward,1)
  }
  else
  {
    action = max_reward
  }
  return(action)
}
```

Epsilon Greedy Function

```
EpsilonGreedyPolicy <- function(x, y, epsilon){

  # Get an epsilon-greedy action for state (x,y) from q_table.
  #
  # Args:
  #   x, y: state coordinates.
  #   epsilon: probability of acting greedily.
  #
  # Returns:
  #   An action, i.e. integer in {1,2,3,4}.

  # Your code here.
  actions = c(1:4)
  action = 0
  rewards = q_table[x,y,1:4]

  max_reward = which(rewards == max(rewards))

  if (length(max_reward)>1){
    action = sample(max_reward,1)
  }
  else
  {
    action = max_reward
  }

  if (1-epsilon < runif(1)){
    return(sample(actions,1))
  }
  else{
    return(action)
  }
}
```

Transition Model

```
transition_model <- function(x, y, action, beta){

  # Computes the new state after given action is taken. The agent will follow the action
  # with probability (1-beta) and slip to the right or left with probability beta/2 each.
  #
  # Args:
  #   x, y: state coordinates.
  #   action: which action the agent takes (in {1,2,3,4}).
  #   beta: probability of the agent slipping to the side when trying to move.
  #   H, W (global variables): environment dimensions.
  #
  # Returns:
  #   The new state after the action has been taken.

  delta <- sample(-1:1, size = 1, prob = c(0.5*beta,1-beta,0.5*beta))
  final_action <- ((action + delta + 3) %% 4) + 1
  foo <- c(x,y) + unlist(action_deltas[final_action])
  foo <- pmax(c(1,1),pmin(foo,c(H,W)))

  return (foo)
}
```

Q Learning Function

Environment A:

For our first environment, we will use H = 5 and W = 7. This environment includes a reward of 10 in state (3,6) and a reward of -1 in states (2,3), (3,3) and (4,3).
We specify the rewards using a reward map in the form of a matrix with one entry for each state. States with no reward will simply have a matrix entry of 0. The agent starts each episode in the state (3,1). The function vis_environment in the file RL Lab1.R is used to visualize the environment and learned action values and policy. You will not have to modify this function, but read the comments in it to familiarize with how it can be used.
When implementing Q-learning, the estimated values of Q(S, A) are commonly stored in a data-structured called Q-table. This is nothing but a tensor with one entry for each state-action pair. Since we have a H × W environment with four actions, we can use a 3D-tensor of dimensions H × W × 4 to represent our Q-table. Initialize all Q-values to 0. Run the function vis_environment before proceeding further. Note that each non-terminal tile has four values. These represent the action values associated to the tile (state). Note also that each non-terminal tile has an arrow. This indicates the greedy policy for the tile (ties are broken at random).

Your are requested to carry out the following tasks:
- Implement the greedy and ϵ-greedy policies in the functions GreedyPolicy and EpsilonGreedyPolicy of the file RL Lab1.R. The functions should break ties at random, i.e. they should sample uniformly from the set of actions with maximal Q-value.

- Implement the Q-learning algorithm in the function q learning of the file RL Lab1.R.
  The function should run one episode of the agent acting in the environment and update the Q-table accordingly. The function should return the episode reward and the sum of the temporal-difference correction terms $R + \gamma * max_a Q(S‘, a) - Q(S, A)$ for all steps in the episode. Note that a transition model taking β as input is already implemented for you in the function transition model.

- Run 10000 episodes of Q-learning with $\epsilon = 0.5$, $\beta = 0$, $\alpha = 0.1$ and $\gamma = 0.95$. To do so, simply run the code provided in the file RL Lab1.R. The code visualizes the Q-table and a greedy policy derived from it after episodes 10, 100, 1000 and 10000. Answer the following questions:

– What has the agent learned after the first 10 episodes ?
– Is the final greedy policy (after 10000 episodes) optimal? Why / Why not ?
– Does the agent learn that there are multiple paths to get to the positive reward ?
If not, what could be done to make the agent learn this ?

Answers:

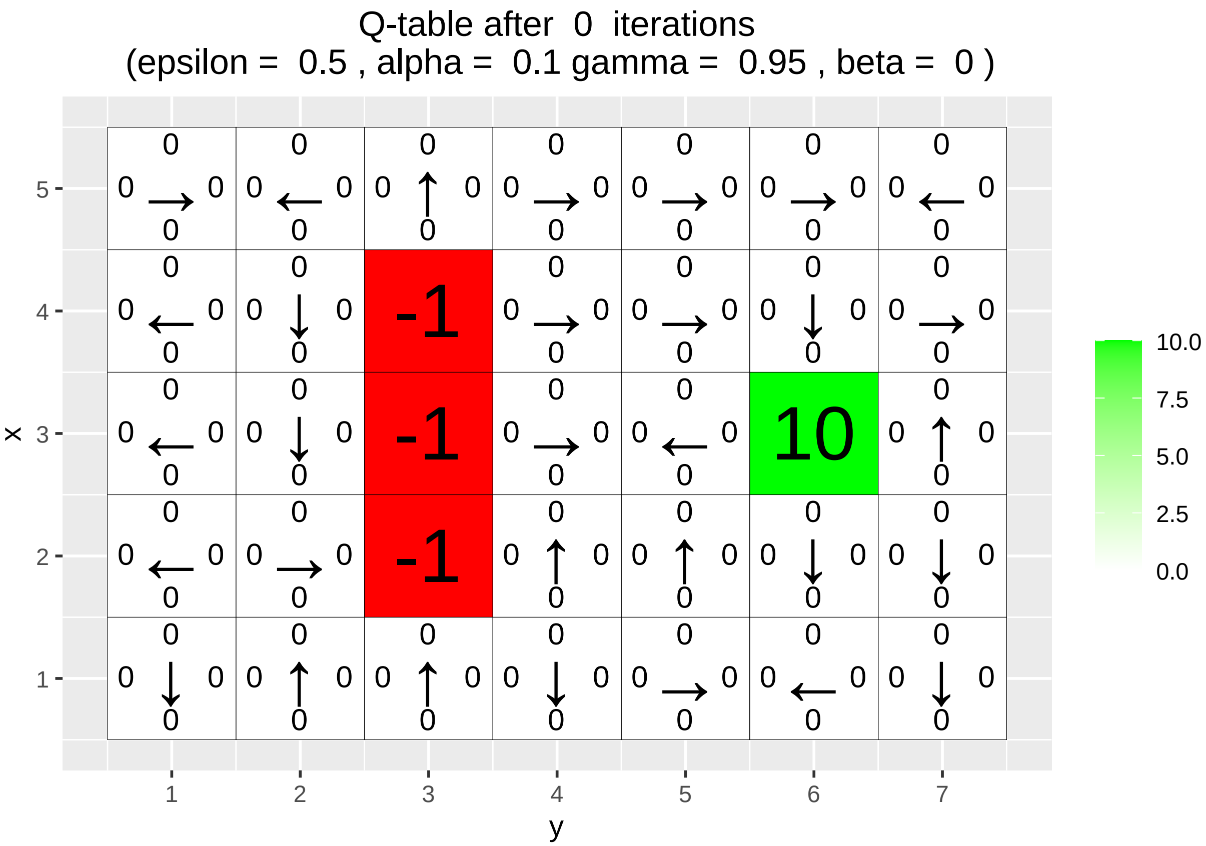
- After the first 10 episodes the agent hasn't learned much, it's got knowledge about avoiding the states with negative ones (-1).

- After 10000 iterations, even though, the agent's knowledge is very good and it's learned how to reach the goal, but the final greedy policy is not optimal, the agent still isn't sure about how good the actions are for some states i.e (1,3). This might be because of the value of epsilon that decides whether to explore the environment or exploit it, and it could also be because of the number of iterations is only 10000.

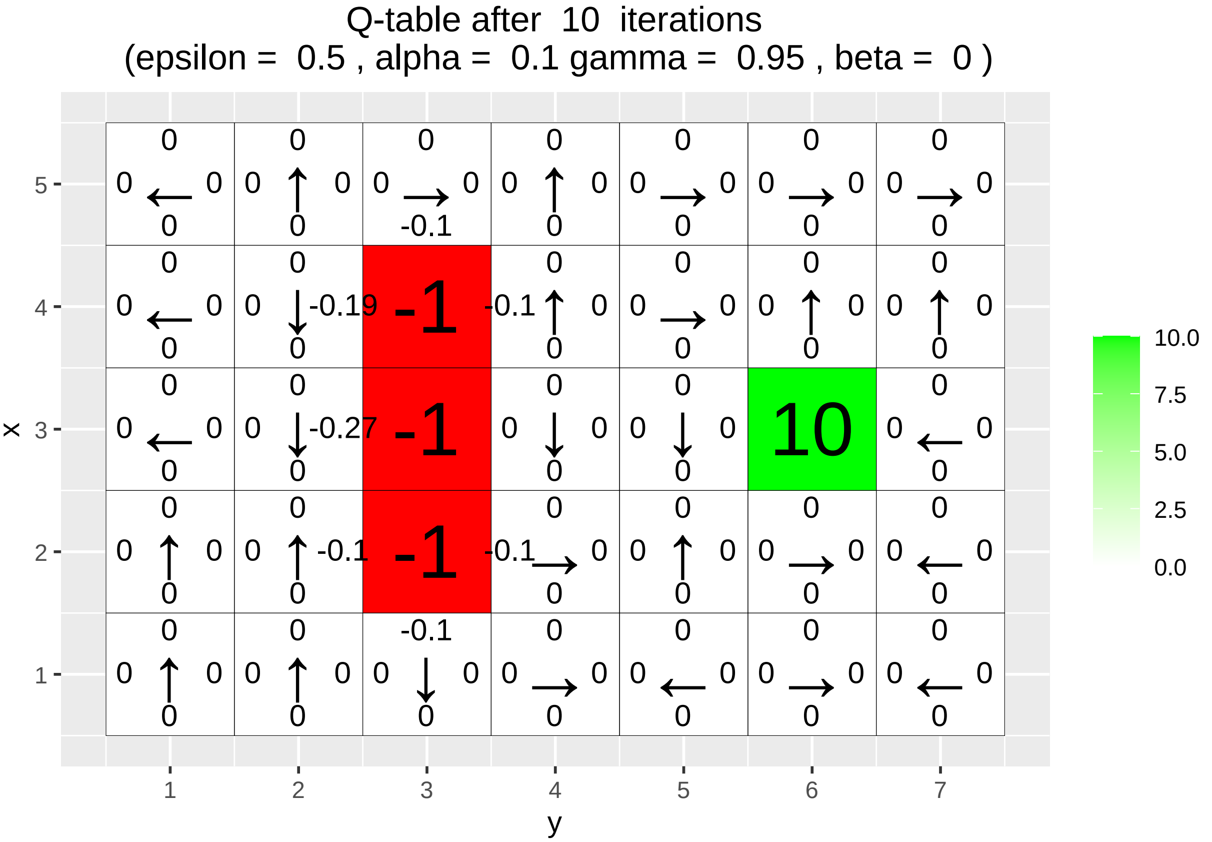
- The agent has learned how to reach to the goal from each state in our grid, it takes one path to the goal and it doesn't slip due the β value of 0 that doesn't allow the agent to slip from the path.

```
# Environment A (learning)

H <- 5
W <- 7
```

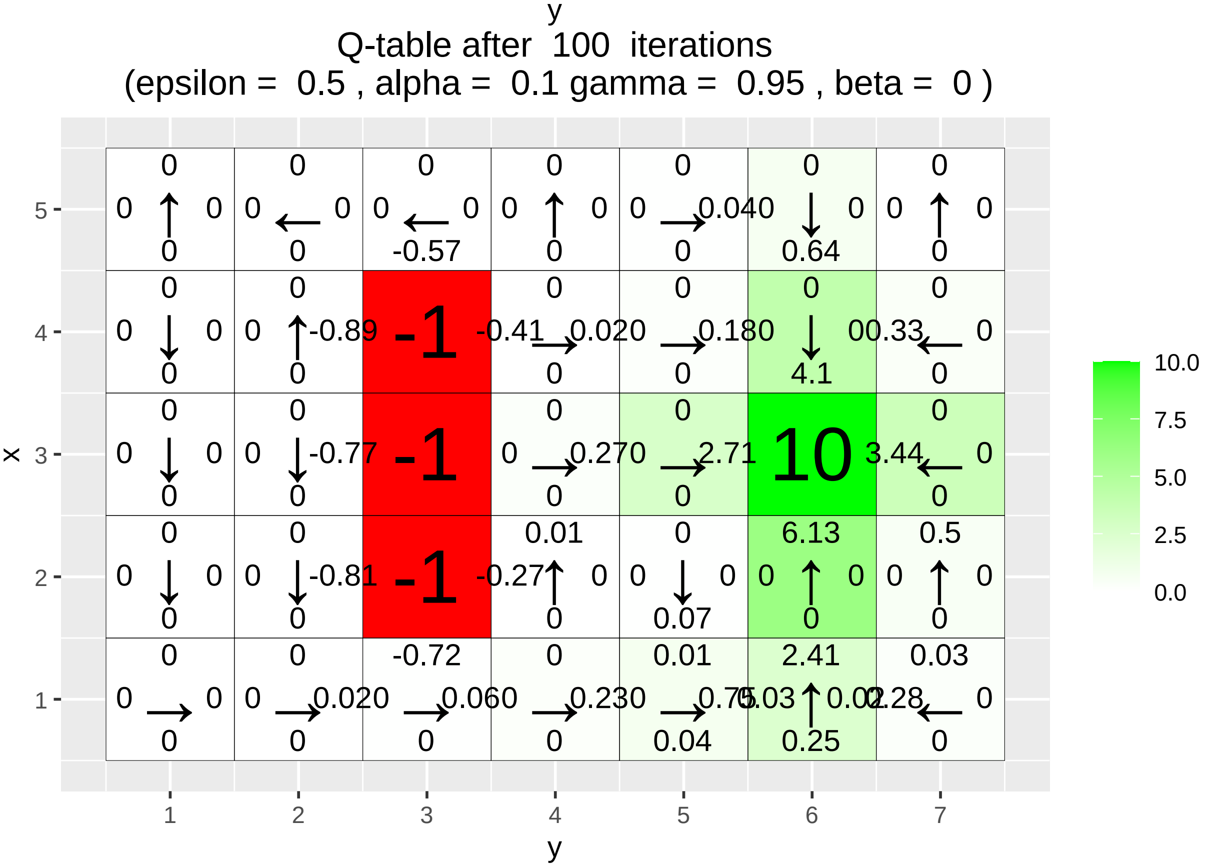
```r
reward_map <- matrix(0, nrow = H, ncol = W)
reward_map[3,6] <- 10
reward_map[2:4,3] <- -1

q_table <- array(0,dim = c(H,W,4))

vis_environment()
```



```r
for(i in 1:10000){
  foo <- q_learning(start_state = c(3,1))

  if(any(i==c(10,100,1000,10000)))
    vis_environment(i)
}
```

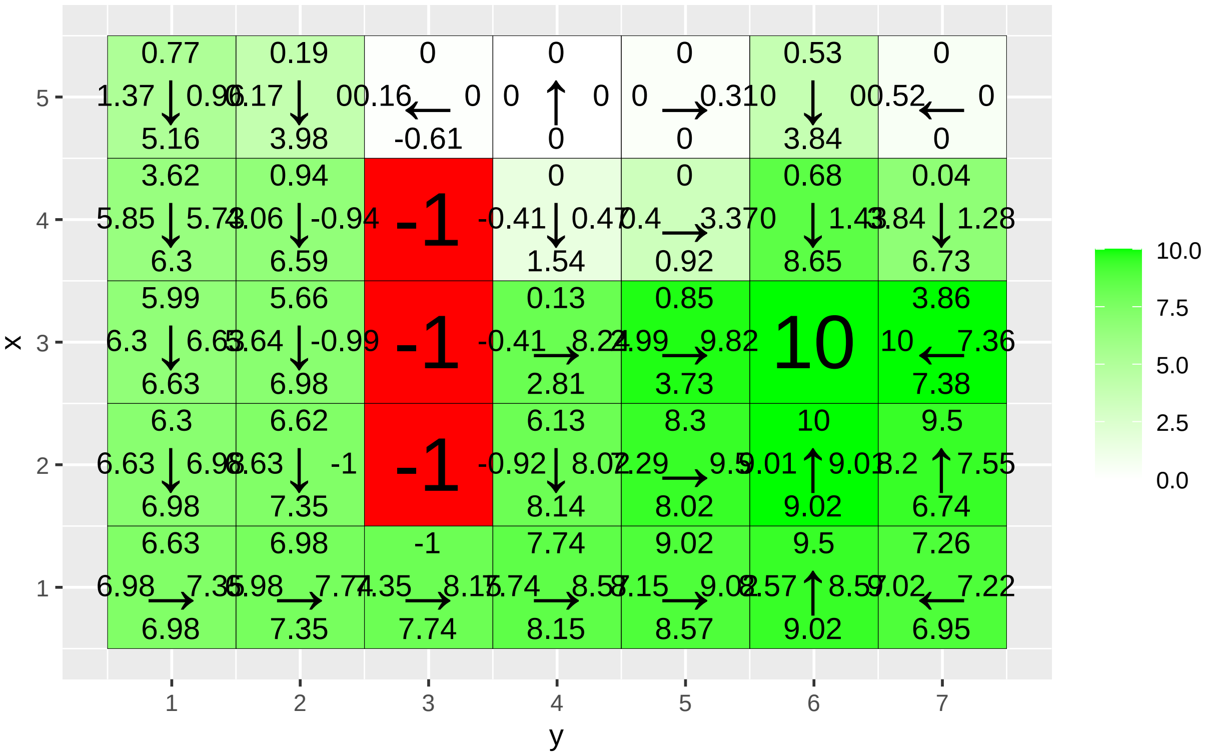


Q-table after 10 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.95 , beta = 0)

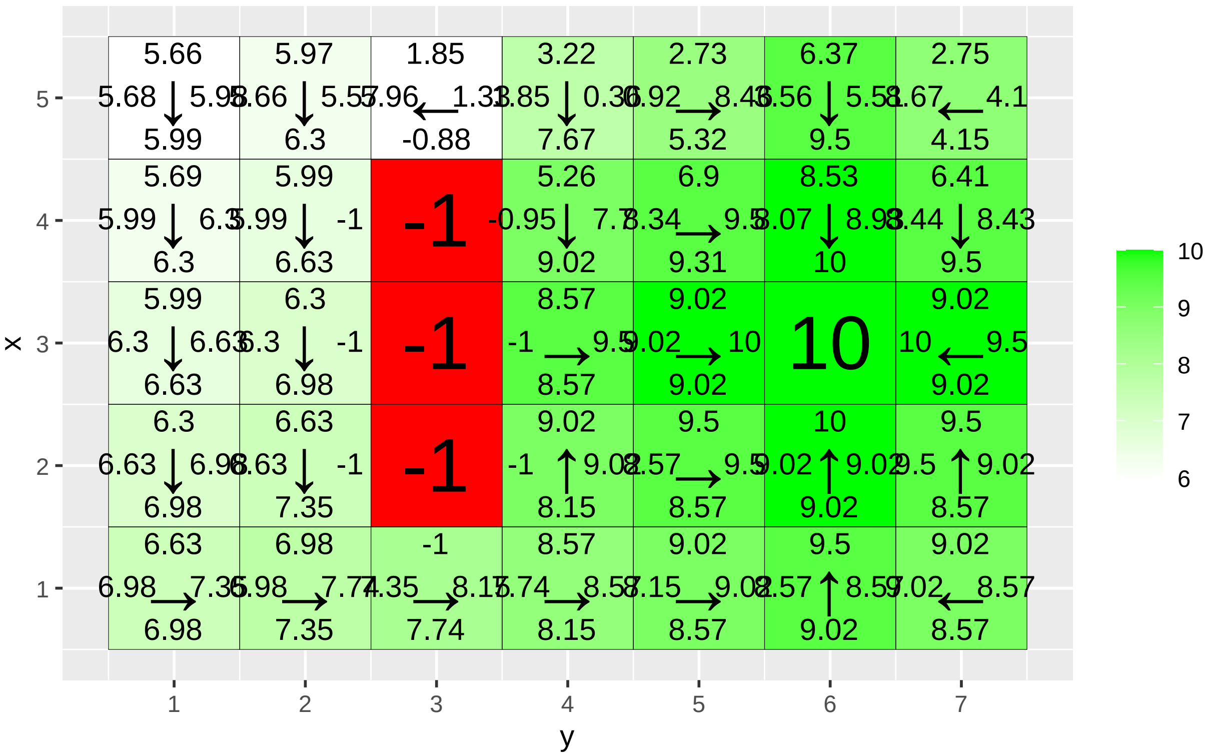


Q-table after 100 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.95 , beta = 0)

Q-table after 1000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.95 , beta = 0)



Q-table after 10000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.95 , beta = 0)



Environment B:

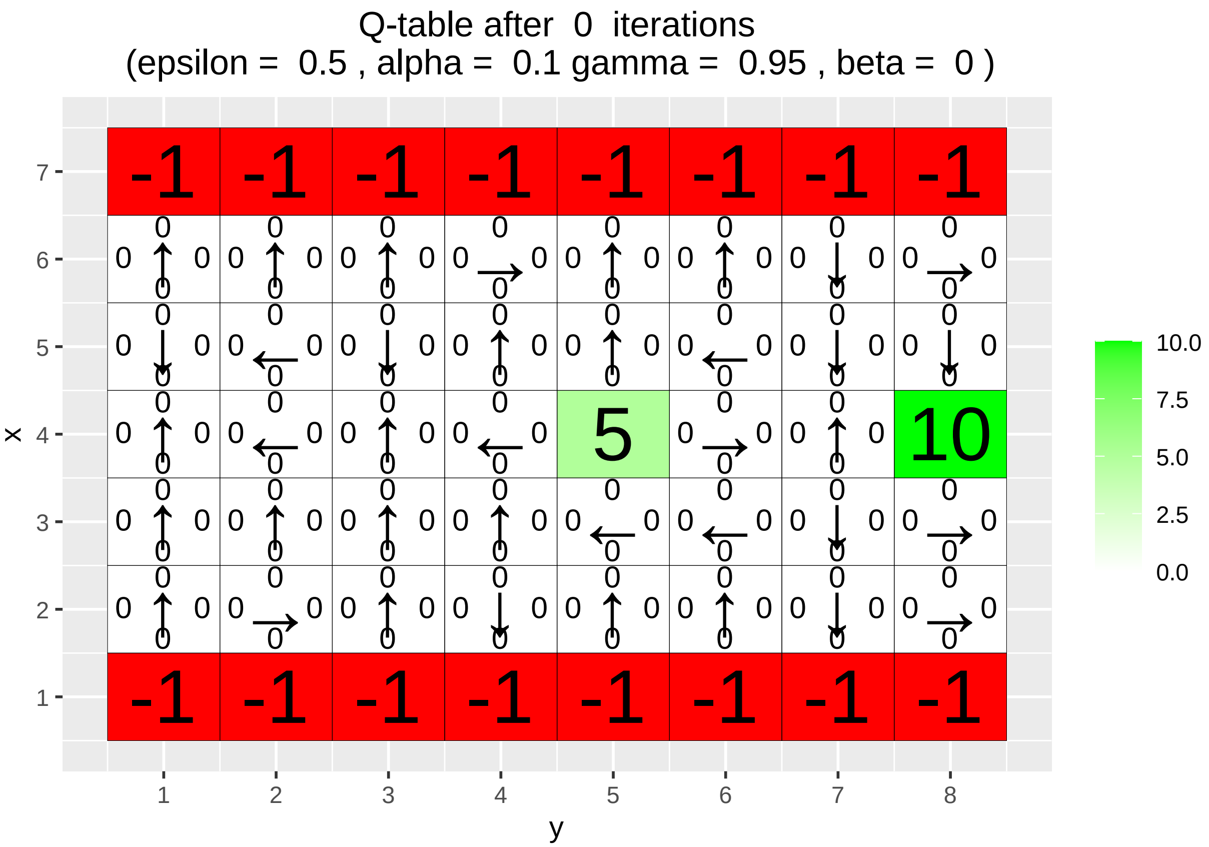
This is a 7×8 environment where the top and bottom rows have negative rewards. In this environment, the agent starts each episode in the state (4, 1). There are two positive rewards, of 5 and 10. The reward of 5 is easily reachable, but the agent has to navigate around the first reward in order to find the reward worth 10. Your task is to investigate how the ϵ and γ parameters affect the learned policy by running 30000 episodes of Q-learning with $\epsilon = 0.1, 0.5$, $\gamma = 0.5, 0.75, 0.95$, $\beta = 0$ and $\alpha = 0.1$. To do so, simply run the code provided in the file RL Lab1.R and explain your observations.

Answer:

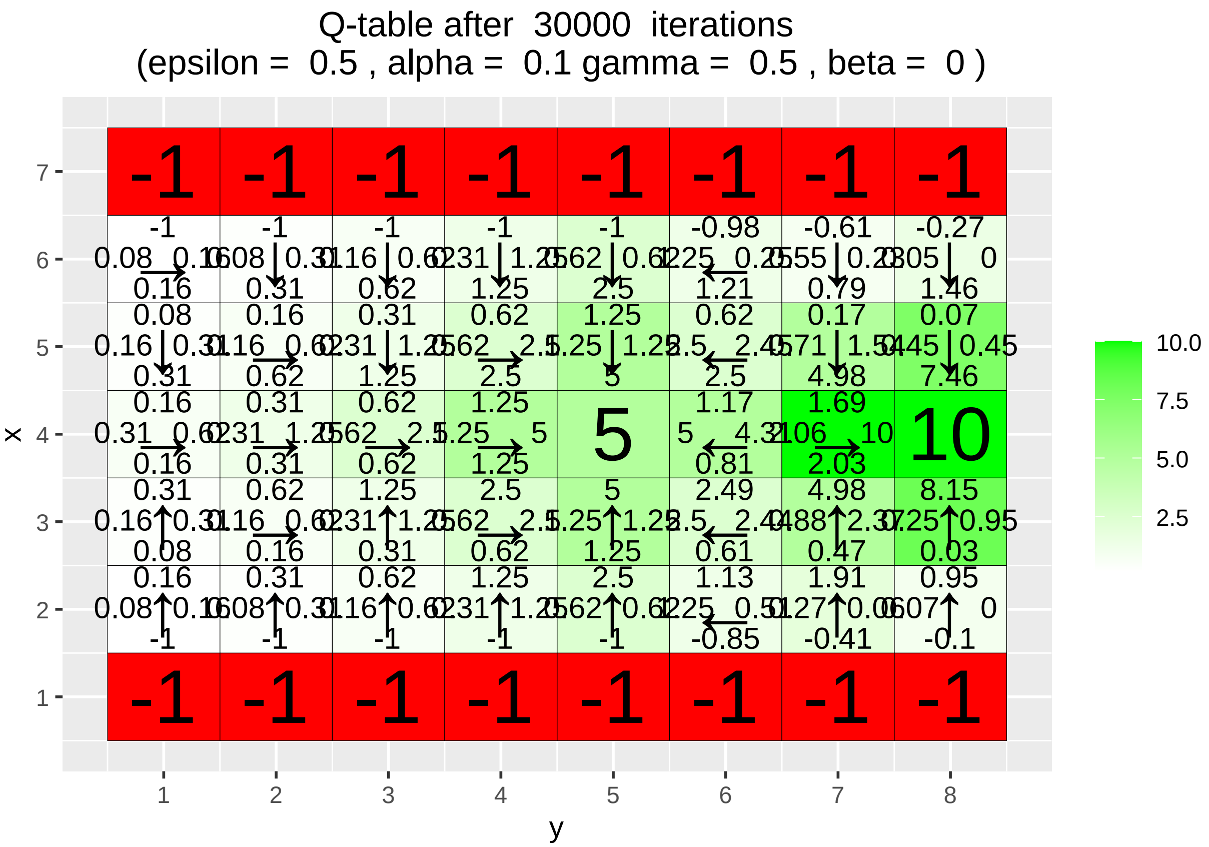
As we can see from the plot when:
($\epsilon = 0.1$, $\alpha = 0.1$ $\gamma = 0.5$, $\beta = 0$), vs ($\epsilon = 0.1$, $\alpha = 0.1$ $\gamma = 0.75$, $\beta = 0$) vs ($\epsilon = 0.1$, $\alpha = 0.1$ $\gamma = 0.95$, β $= 0$)
When we increased the value of γ from 0.5 to 0.75 with the same value of ϵ, the agent hasn't explored after reaching the state with 5 reward because we increased the discount factor that determines whether the agent should take the immediate reward or explore (considers rewards later in time). And when using an γ value of 0.95, the agent has learned more about how to reach the state with 5 rewards (if we have the same preferences, then γ should be 1), but still the agent has no idea about the state with 10 rewards because the value of ϵ is small which means not taking a greedy action exploring with the discount factor, and the number of iterations also effect the learning γ

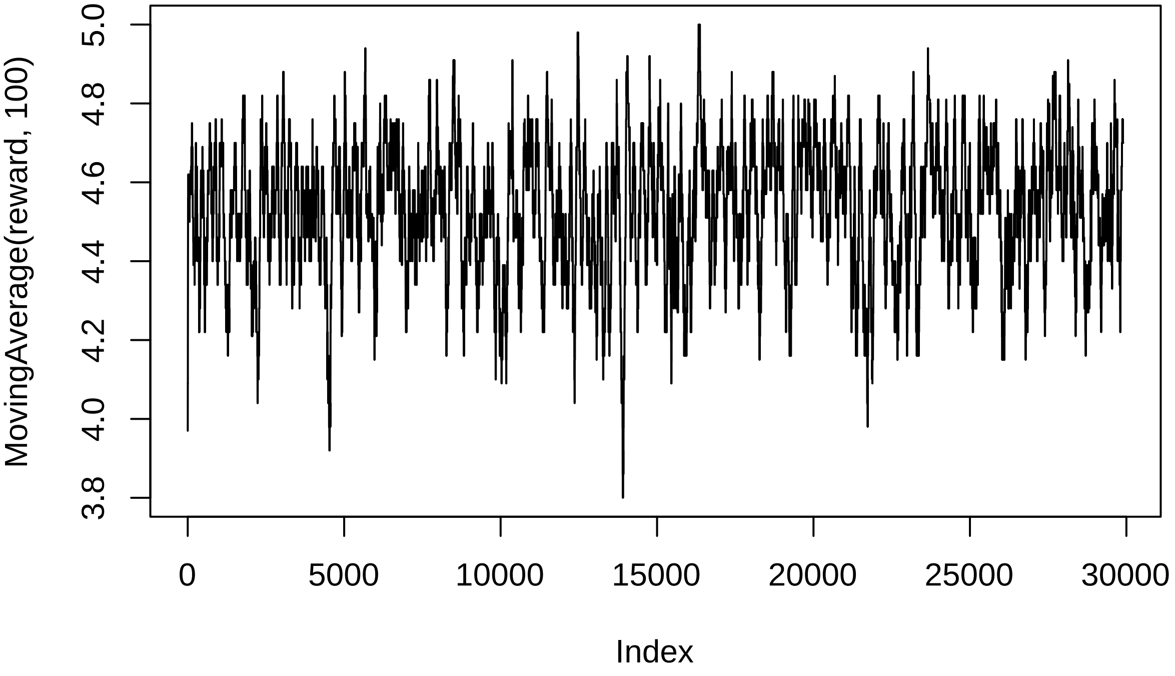
Nonetheless, when the value of ϵ increases to 5, then the agent learns about the state with 10 reward, and that is because a higher value of ϵ allows the agent to explore the environment more by not taking the action that maximizes the reward all the time, but with probability.

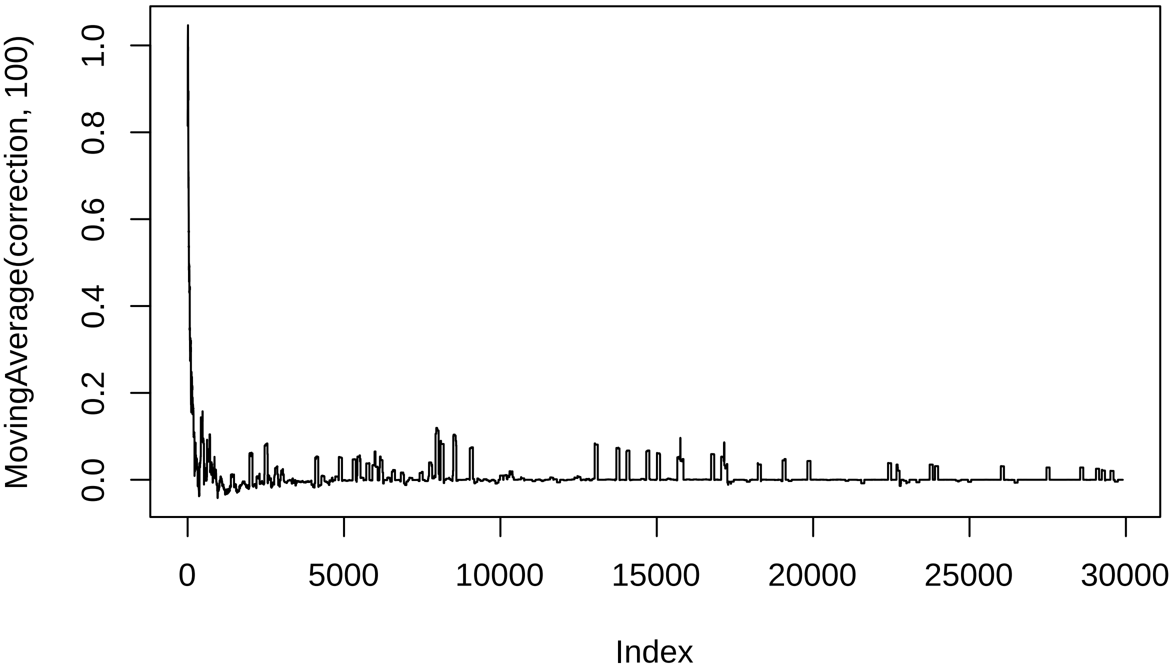
```r
# Environment B (the effect of epsilon and gamma)

H <- 7
W <- 8

reward_map <- matrix(0, nrow = H, ncol = W)
reward_map[1,] <- -1
reward_map[7,] <- -1
reward_map[4,5] <- 5
reward_map[4,8] <- 10

q_table <- array(0,dim = c(H,W,4))

vis_environment()
```

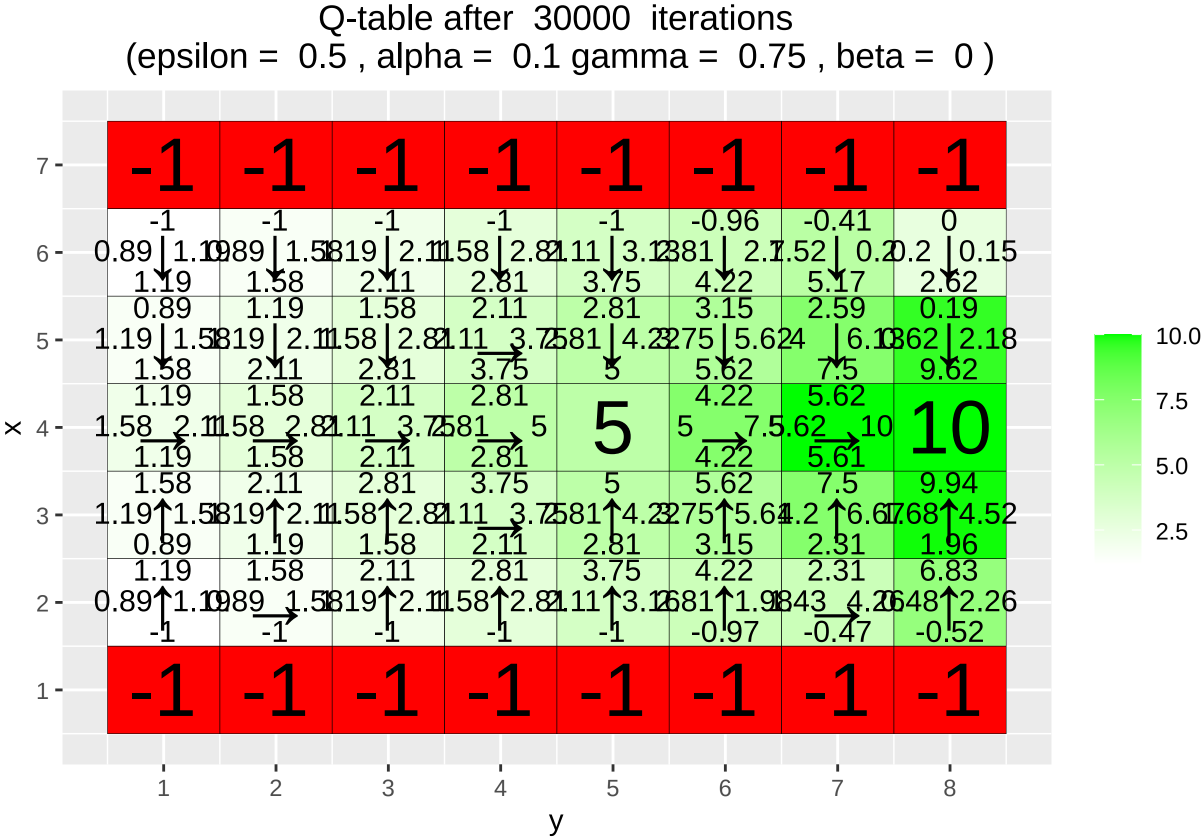


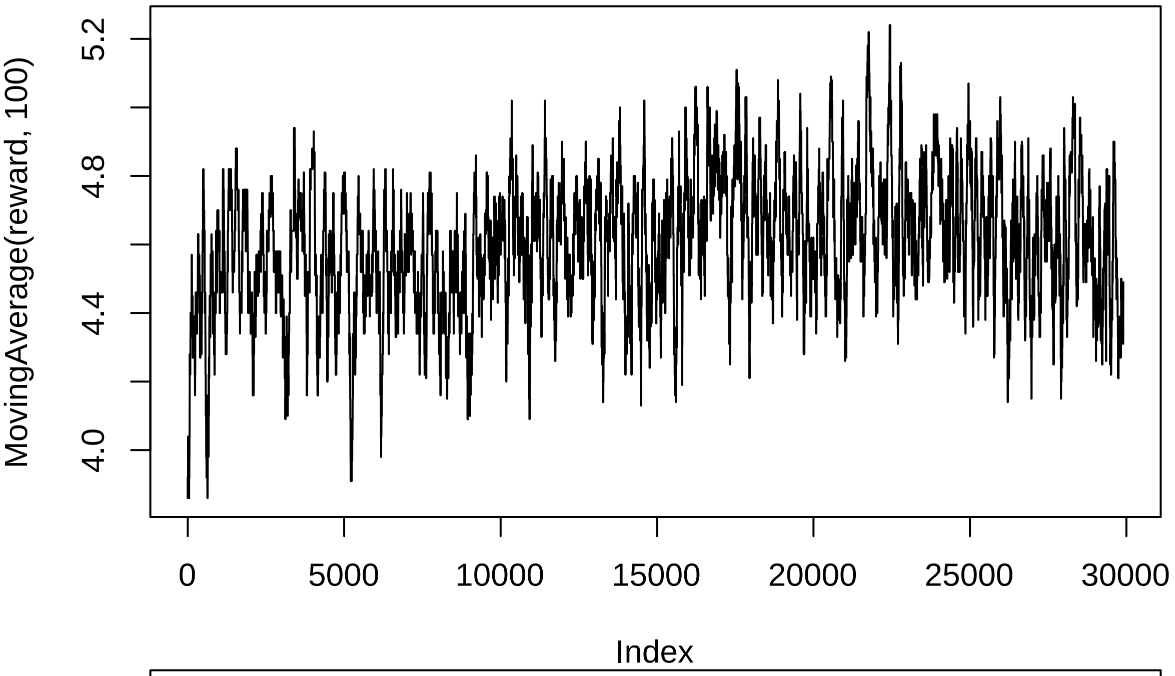
```
MovingAverage <- function(x, n){

  cx <- c(0,cumsum(x))
  rsum <- (cx[(n+1):length(cx)] - cx[1:(length(cx) - n)]) / n

  return (rsum)
}

for(j in c(0.5,0.75,0.95)){
  q_table <- array(0,dim = c(H,W,4))
  reward <- NULL
  correction <- NULL

  for(i in 1:30000){
    foo <- q_learning(gamma = j, start_state = c(4,1))
    reward <- c(reward,foo[1])
    correction <- c(correction,foo[2])
  }

  vis_environment(i, gamma = j)
  plot(MovingAverage(reward,100),type = "l")
  plot(MovingAverage(correction,100),type = "l")
}
```

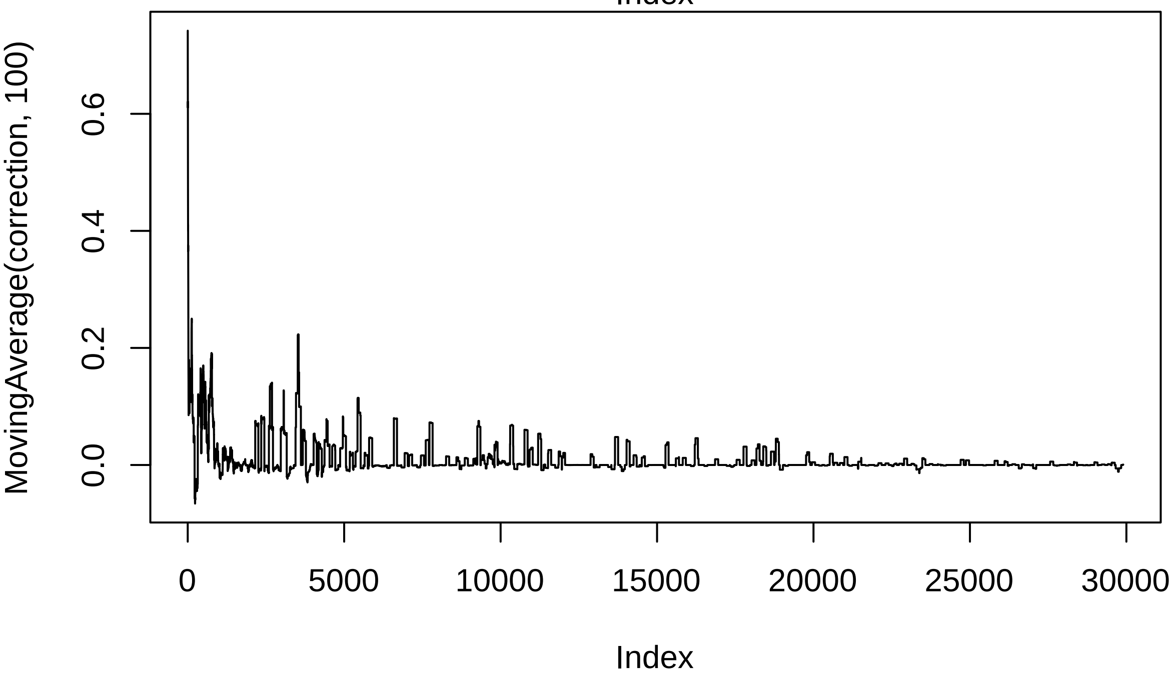

Q-table after 30000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.5 , beta = 0)
x
y
1
2
3
4
5
6
7
8
-1
5
10
10.0
7.5
5.0
2.5

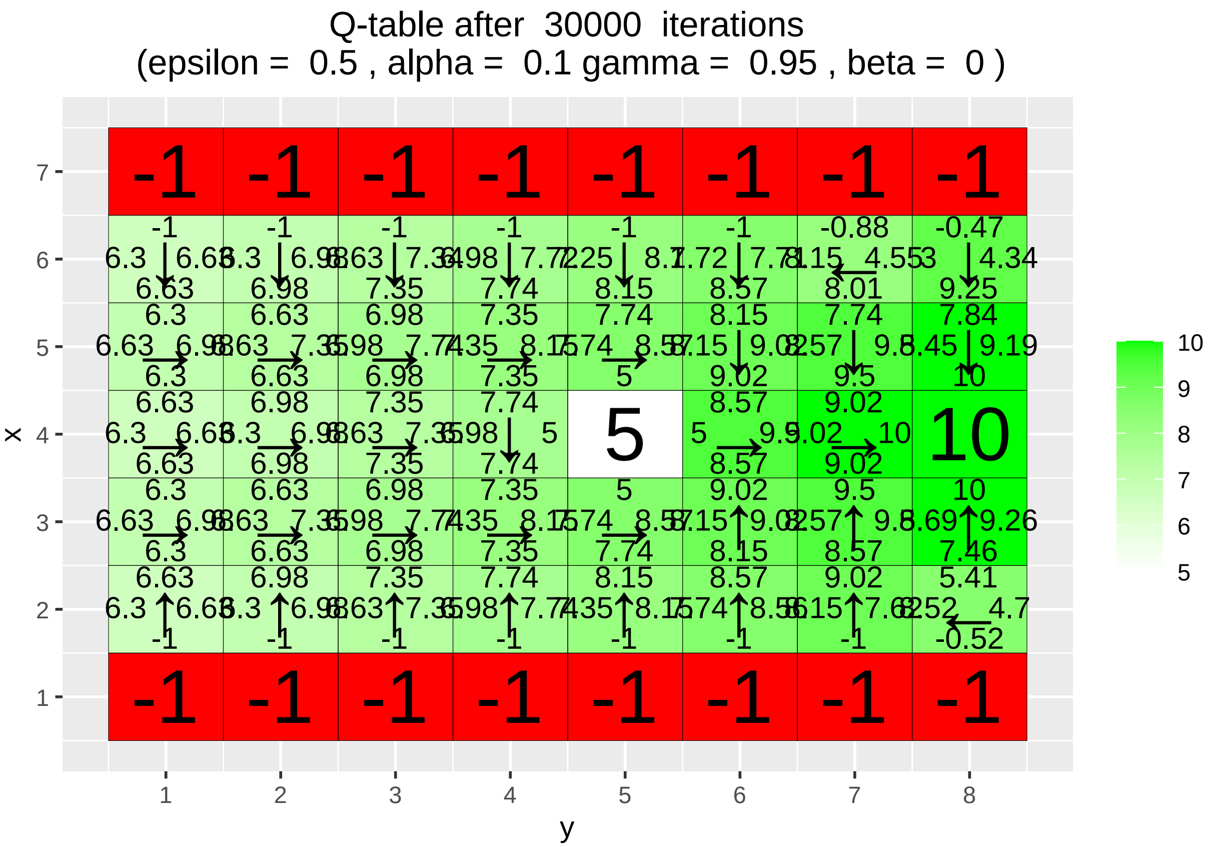

MovingAverage(reward, 100)
Index
3.8
4.0
4.2
4.4
4.6
4.8
5.0
0
5000
10000
15000
20000
25000
30000

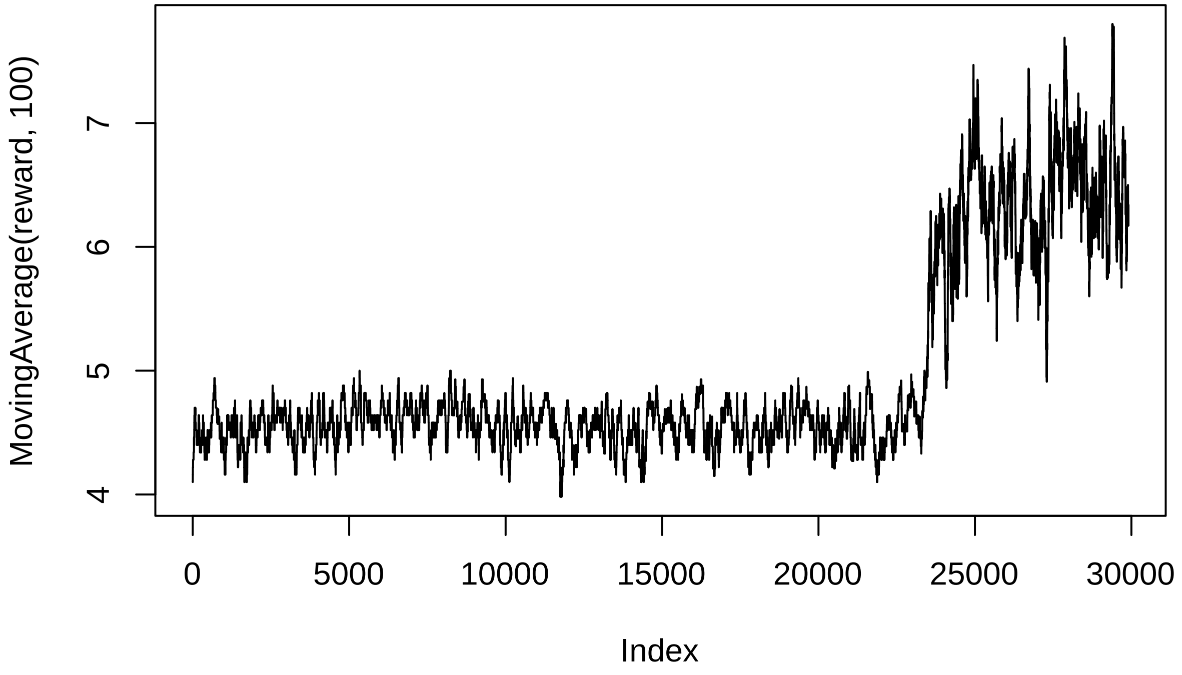

MovingAverage(correction, 100)
0.0
0.2
0.4
0.6
0.8
1.0
0
5000
10000
15000
20000
25000
30000
Index

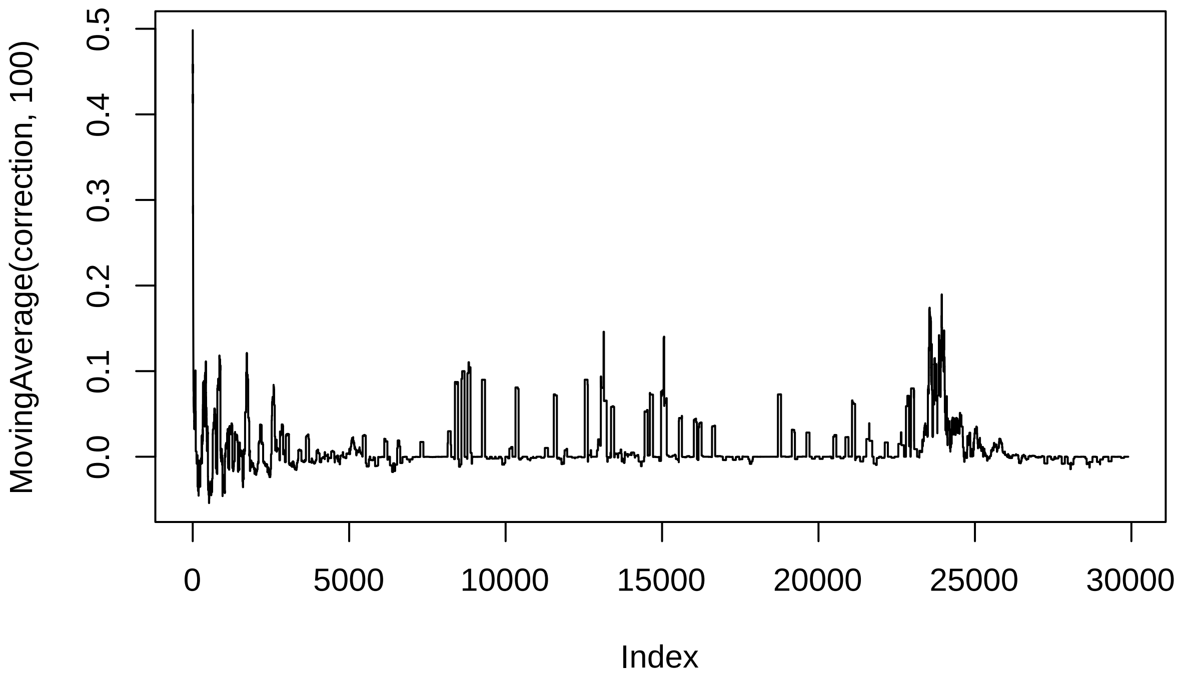

Q-table after 30000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.75 , beta = 0)
x
1
2
3
4
5
6
7
y
1
2
3
4
5
6
7
8
-1
5
10
10.0
7.5
5.0
2.5


MovingAverage(reward, 100)
4.0
4.4
4.8
5.2
0
5000
10000
15000
20000
25000
30000
Index

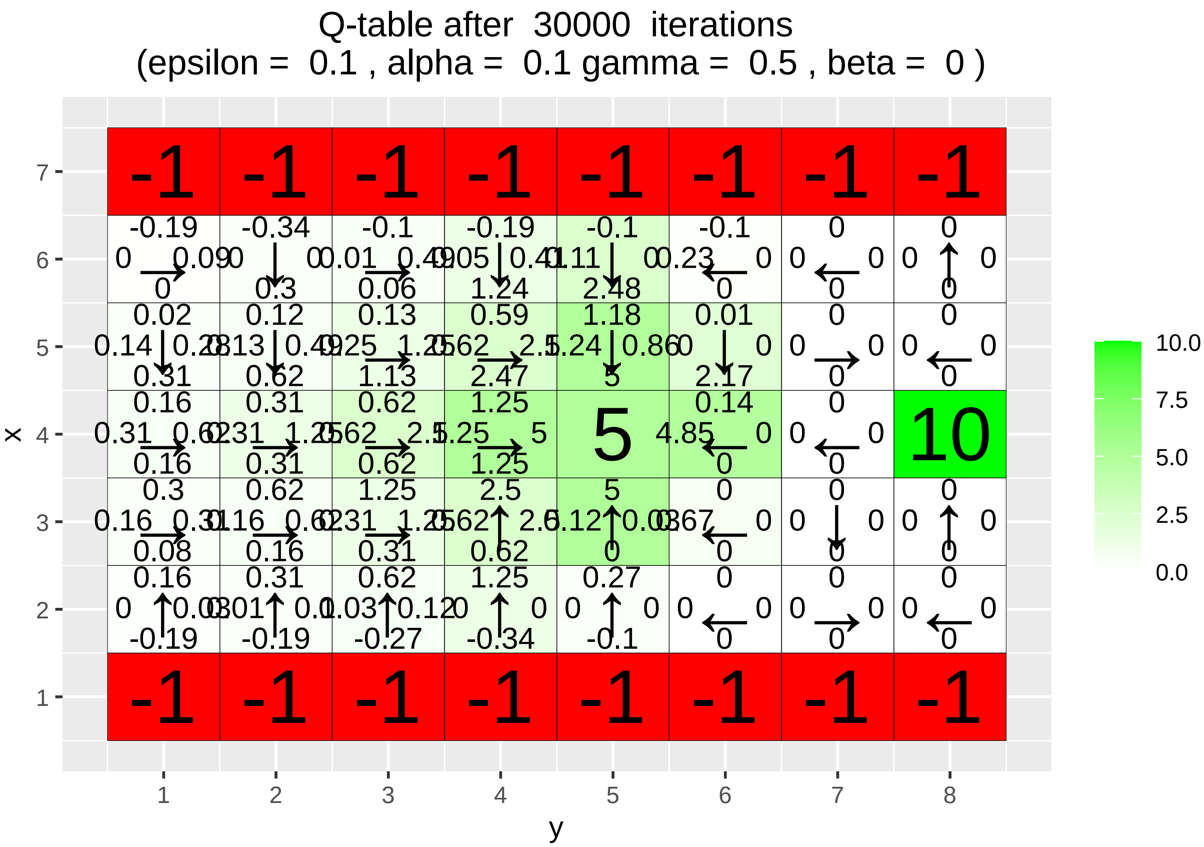

MovingAverage(correction, 100)
0.0
0.2
0.4
0.6
0
5000
10000
15000
20000
25000
30000
Index

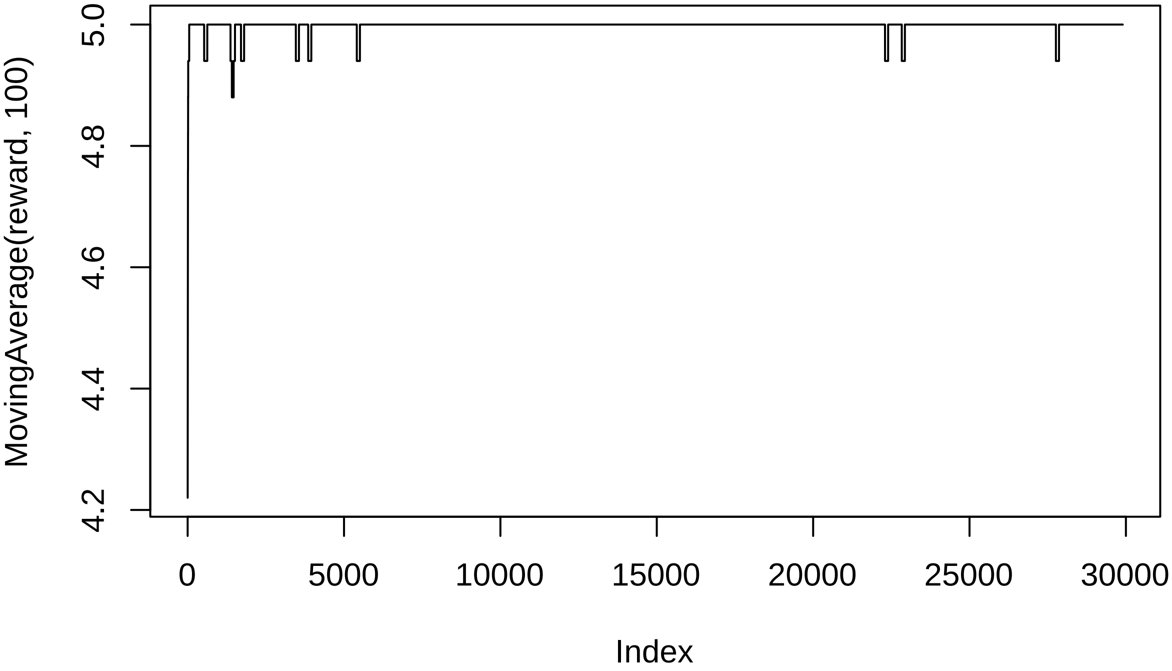

Q-table after 30000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.95 , beta = 0)
x
y
1
2
3
4
5
6
7
8
-1
5
10
10
9
8
7
6
5

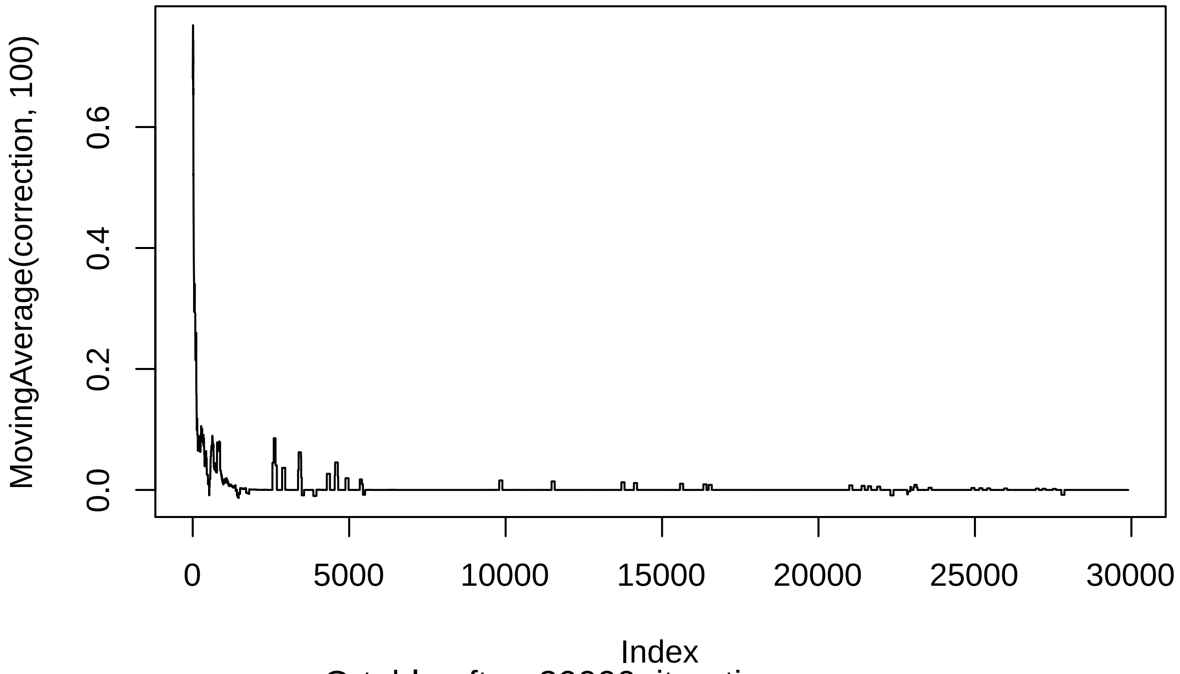

MovingAverage(reward, 100)
Index
4
5
6
7
0
5000
10000
15000
20000
25000
30000



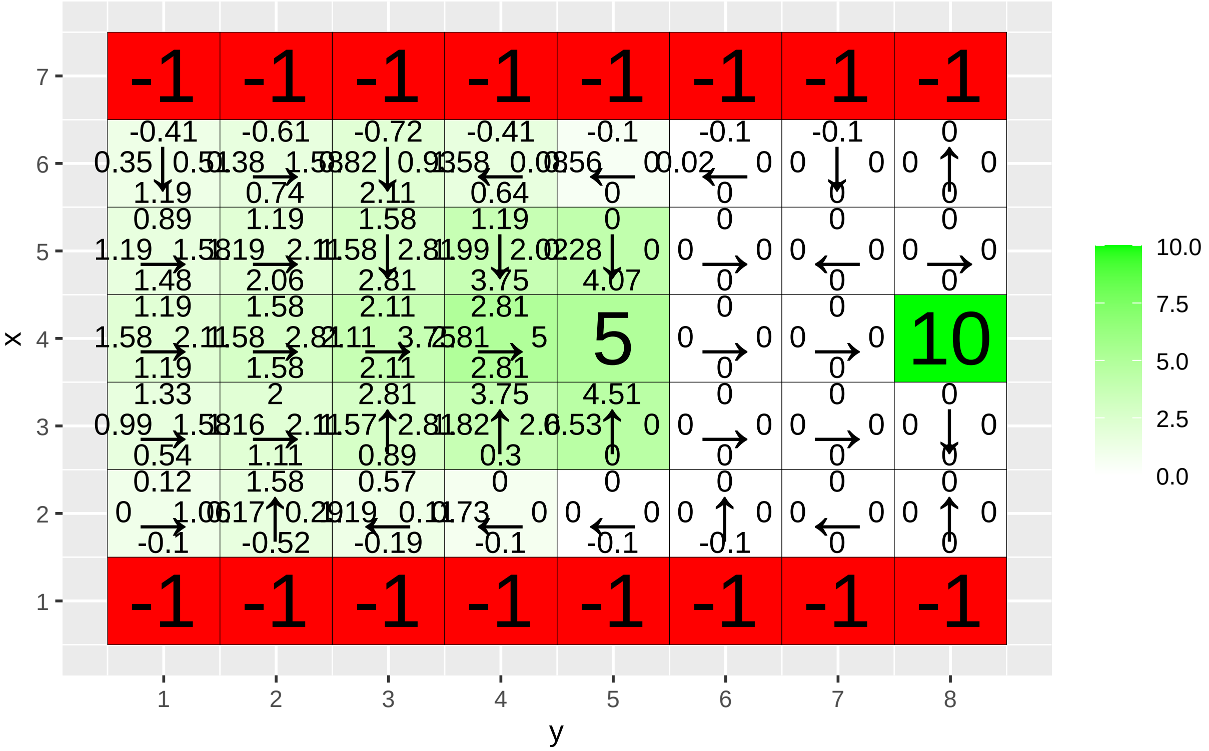
```r
for(j in c(0.5,0.75,0.95)){
  q_table <- array(0,dim = c(H,W,4))
  reward <- NULL
  correction <- NULL

  for(i in 1:30000){
    foo <- q_learning(epsilon = 0.1, gamma = j, start_state = c(4,1))
    reward <- c(reward,foo[1])
    correction <- c(correction,foo[2])
  }

  vis_environment(i, epsilon = 0.1, gamma = j)
  plot(MovingAverage(reward,100),type = "l")
  plot(MovingAverage(correction,100),type = "l")
}
```

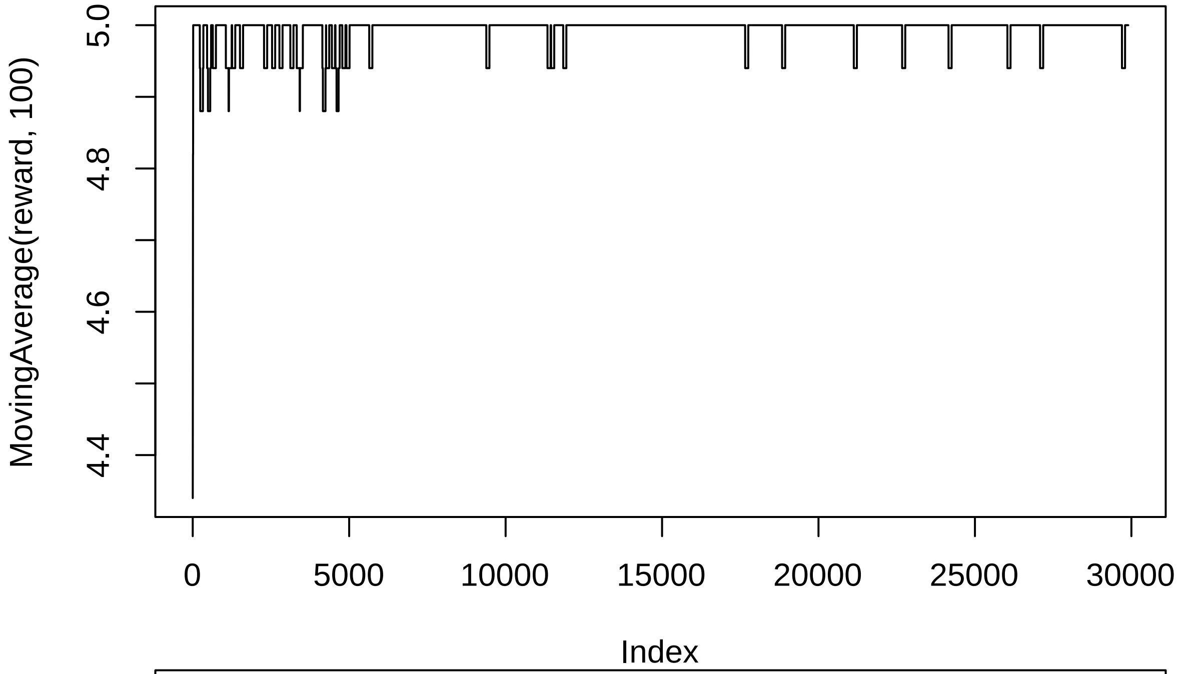

Q-table after 30000 iterations
(epsilon = 0.1 , alpha = 0.1 gamma = 0.5 , beta = 0)
x
y
1
2
3
4
5
6
7
8
-1
5
10
10.0
7.5
5.0
2.5
0.0

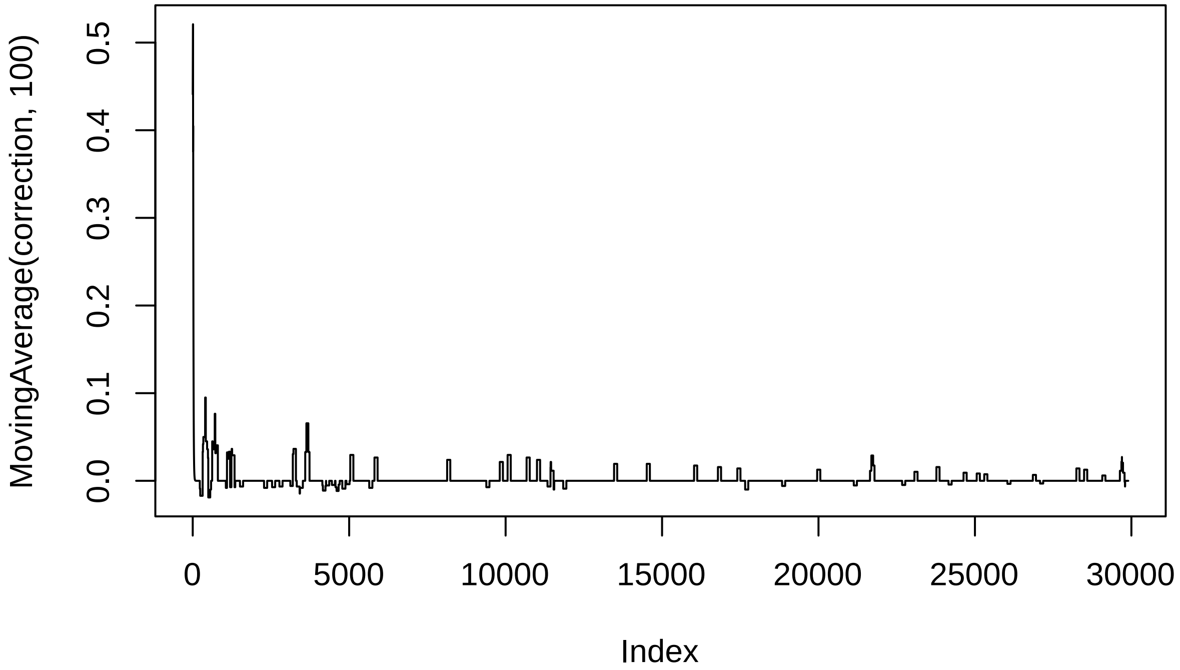

MovingAverage(reward, 100)
Index
4.2
4.4
4.6
4.8
5.0
0
5000
10000
15000
20000
25000
30000

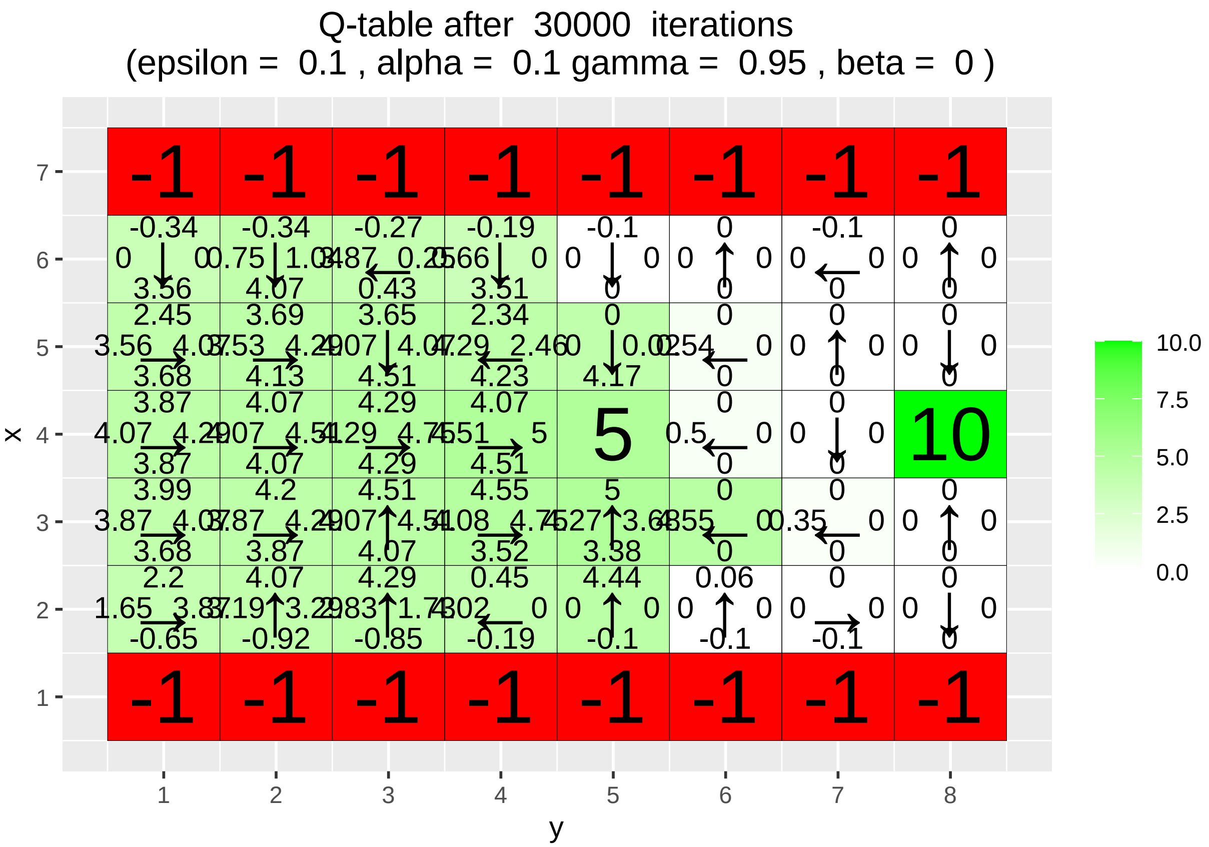


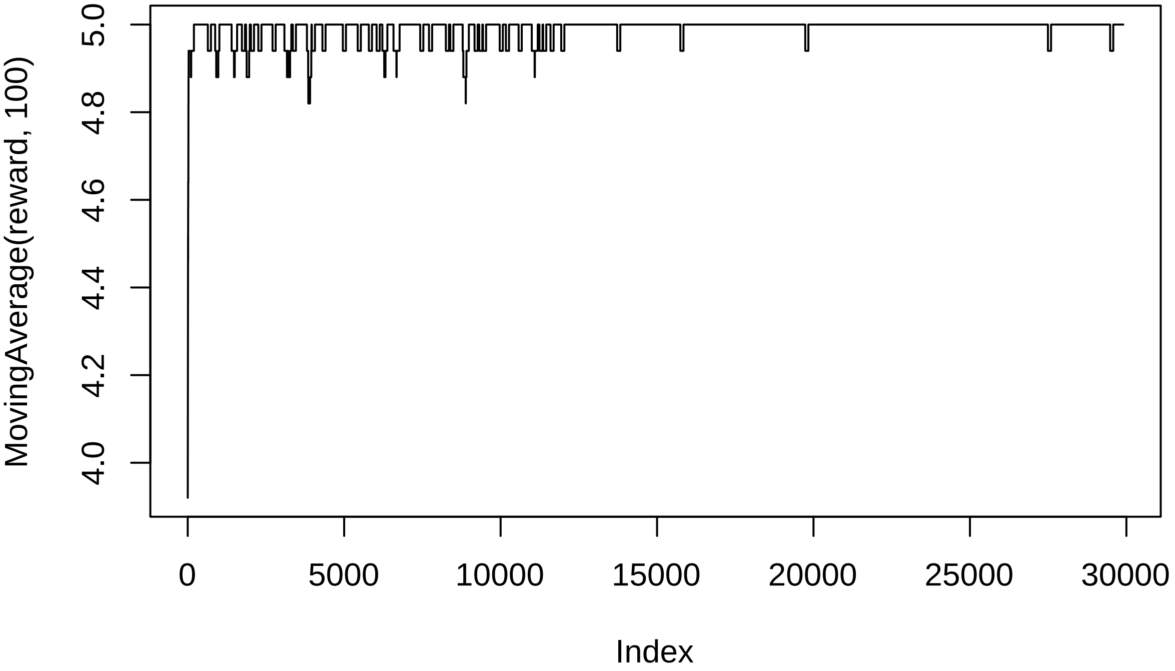
Q-table after 30000 iterations
(epsilon = 0.1 , alpha = 0.1 gamma = 0.75 , beta = 0)

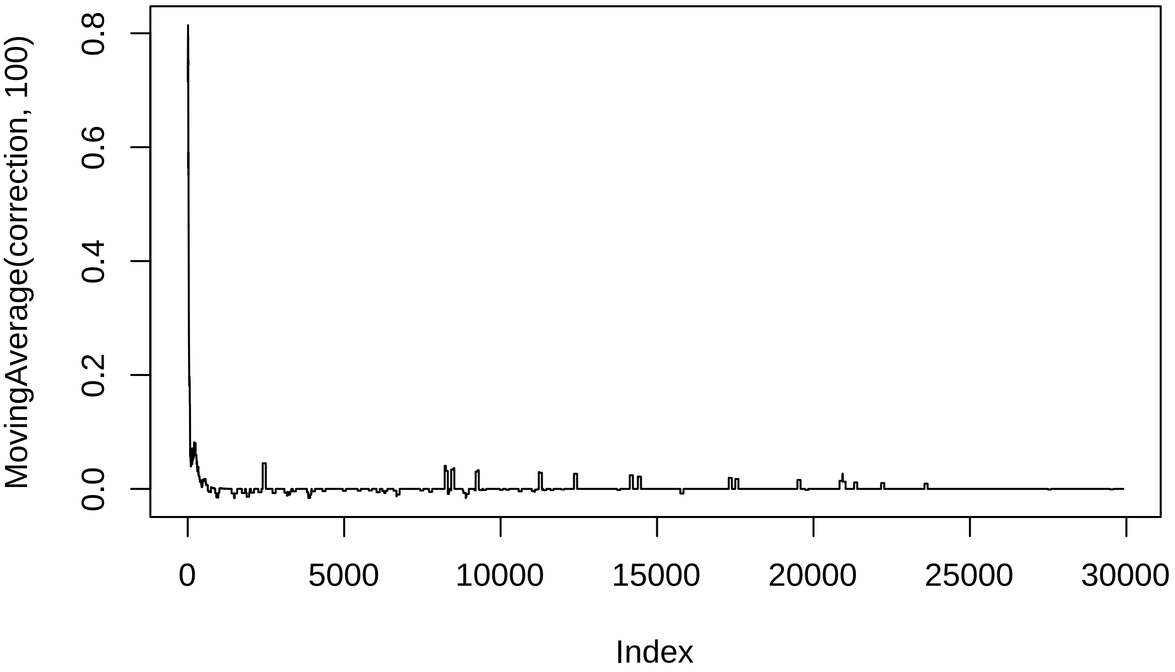



MovingAverage(reward, 100)
4.4
4.6
4.8
5.0
0
5000
10000
15000
20000
25000
30000
Index


MovingAverage(correction, 100)
0.0
0.1
0.2
0.3
0.4
0.5
0
5000
10000
15000
20000
25000
30000
Index


Q-table after 30000 iterations
(epsilon = 0.1 , alpha = 0.1 gamma = 0.95 , beta = 0)
-1 -1 -1 -1 -1 -1 -1 -1
-0.34 -0.34 -0.27 -0.19 -0.1 0 -0.1 0
3.56 4.07 0.43 3.51 0 0 0 0
2.45 3.69 3.65 2.34 0 0 0 0
3.68 4.13 4.51 4.23 4.17 0 0 0
3.87 4.07 4.29 4.07 0 0
3.87 4.07 4.29 4.51
5
0.5
10
3.99 4.2 4.51 4.55 5 0 0 0
3.68 3.87 4.07 3.52 3.38 0 0 0
2.2 4.07 4.29 0.45 4.44 0.06 0 0
-0.65 -0.92 -0.85 -0.19 -0.1 -0.1 -0.1 0
-1 -1 -1 -1 -1 -1 -1 -1
7
6
5
x 4
3
2
1
1 2 3 4 5 6 7 8
y
10.0
7.5
5.0
2.5
0.0


MovingAverage(reward, 100)
5.0
4.8
4.6
4.4
4.2
4.0
0
5000
10000
15000
20000
25000
30000
Index



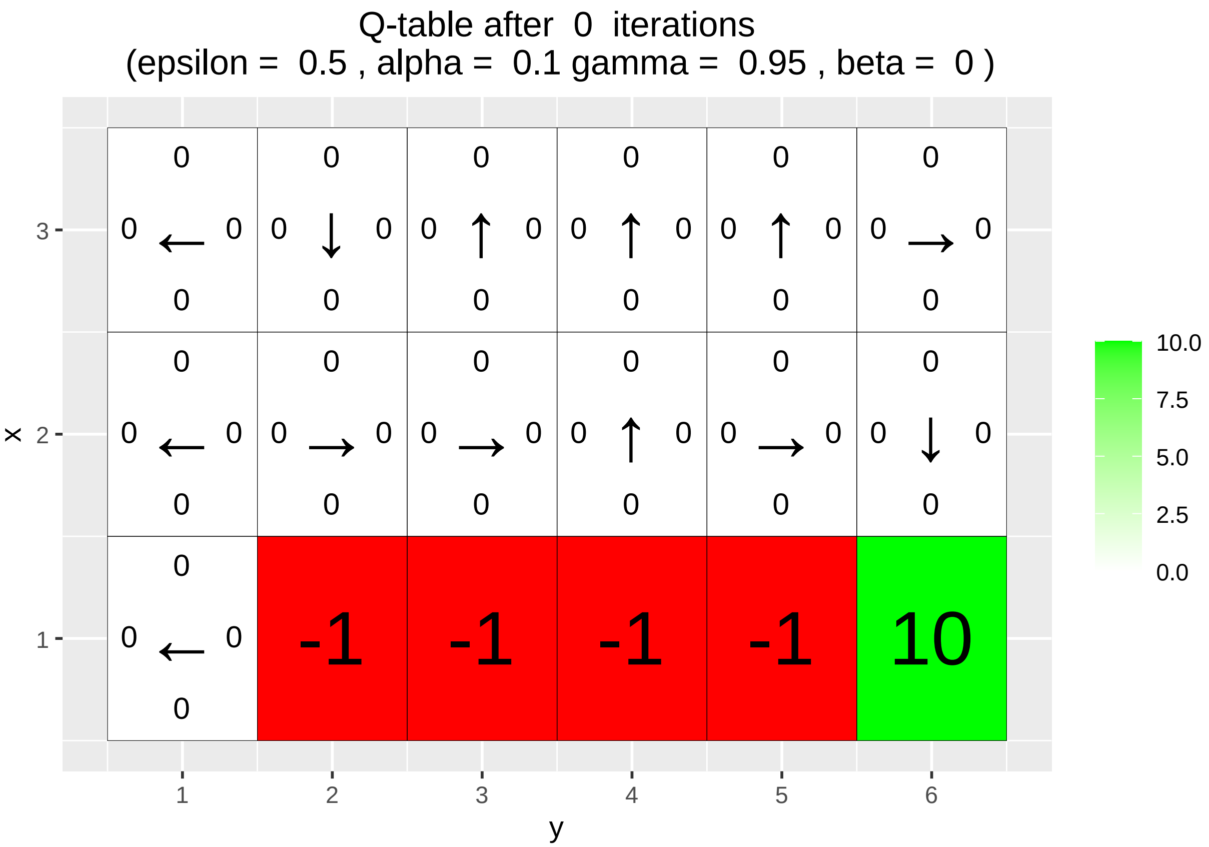
Environment C

This is a smaller 3 × 6 environment. Here the agent starts each episode in the state (1,1). Your task is to investigate how the β parameter affects the learned policy by running 10000 episodes of Q-learning with $\beta = 0, 0.2, 0.4, 0.66$, $\epsilon = 0.5$, $\gamma = 0.6$ and $\alpha = 0.1$. To do so, simply run the code provided in the file RL Lab1.R and explain your observations.

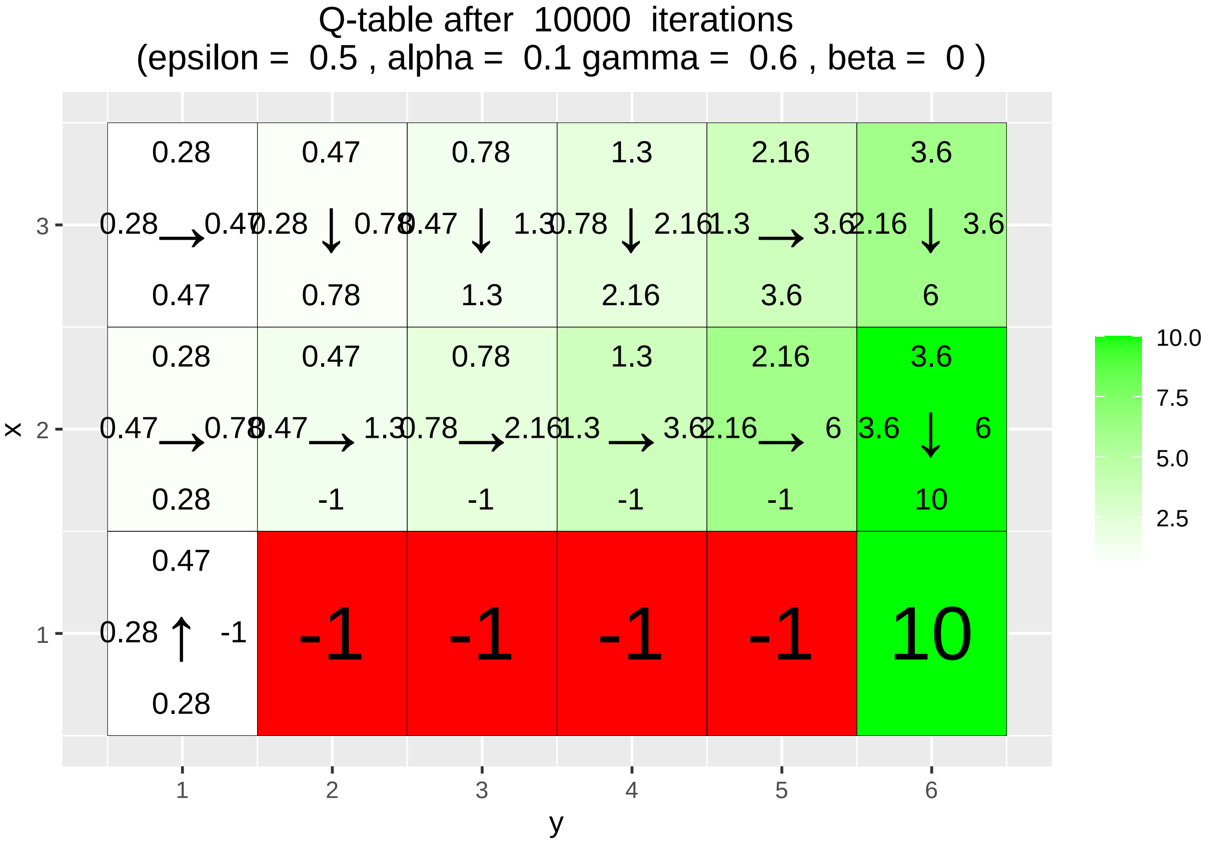
Answer:

As it's seen in the plots when we increase the value of β, the frequency of the agent slipping from the path to the goal increases, for the highest values of β (0.66) it goes in a loop in some states.

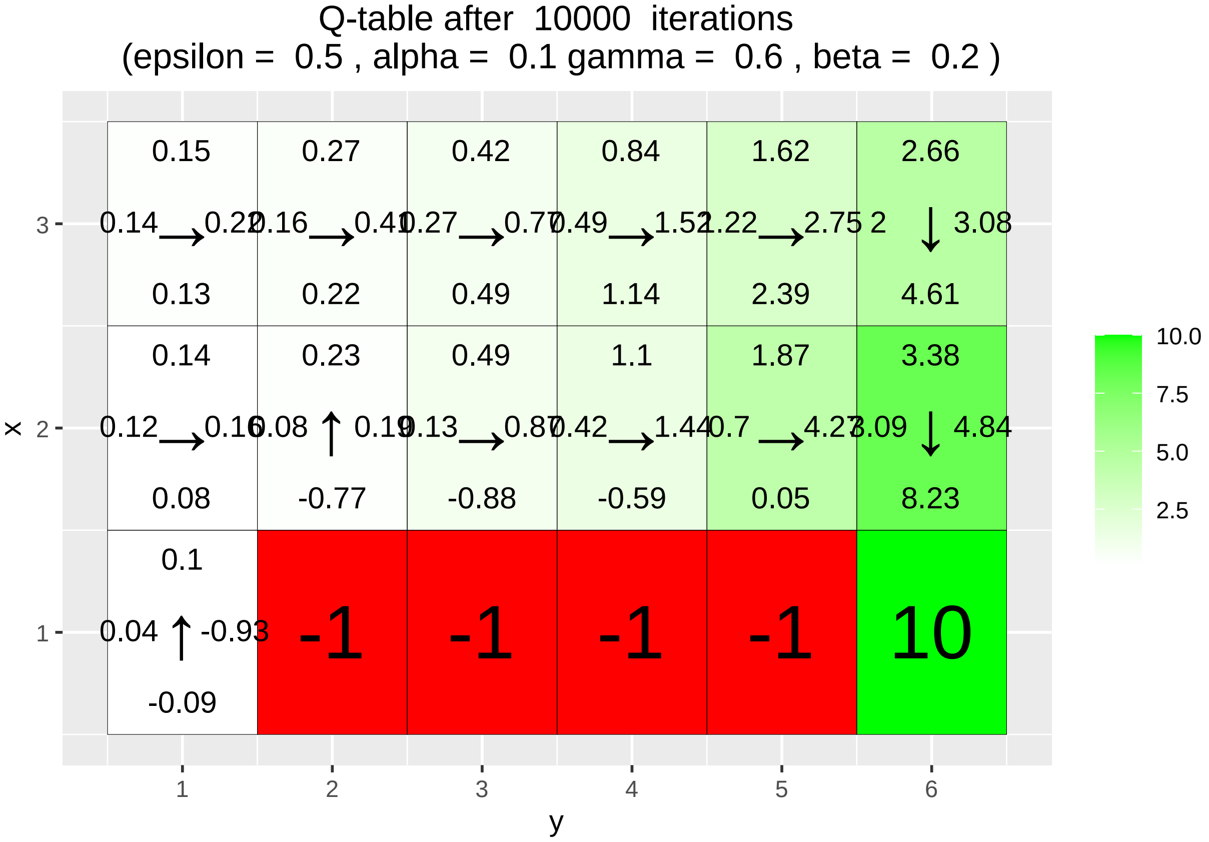
```
# Environment C (the effect of beta).

H <- 3
W <- 6

reward_map <- matrix(0, nrow = H, ncol = W)
reward_map[1,2:5] <- -1
reward_map[1,6] <- 10

q_table <- array(0,dim = c(H,W,4))

vis_environment()
```



```
for(j in c(0,0.2,0.4,0.66)){
  q_table <- array(0,dim = c(H,W,4))

  for(i in 1:10000)
    foo <- q_learning(gamma = 0.6, beta = j, start_state = c(1,1))

  vis_environment(i, gamma = 0.6, beta = j)
}
```

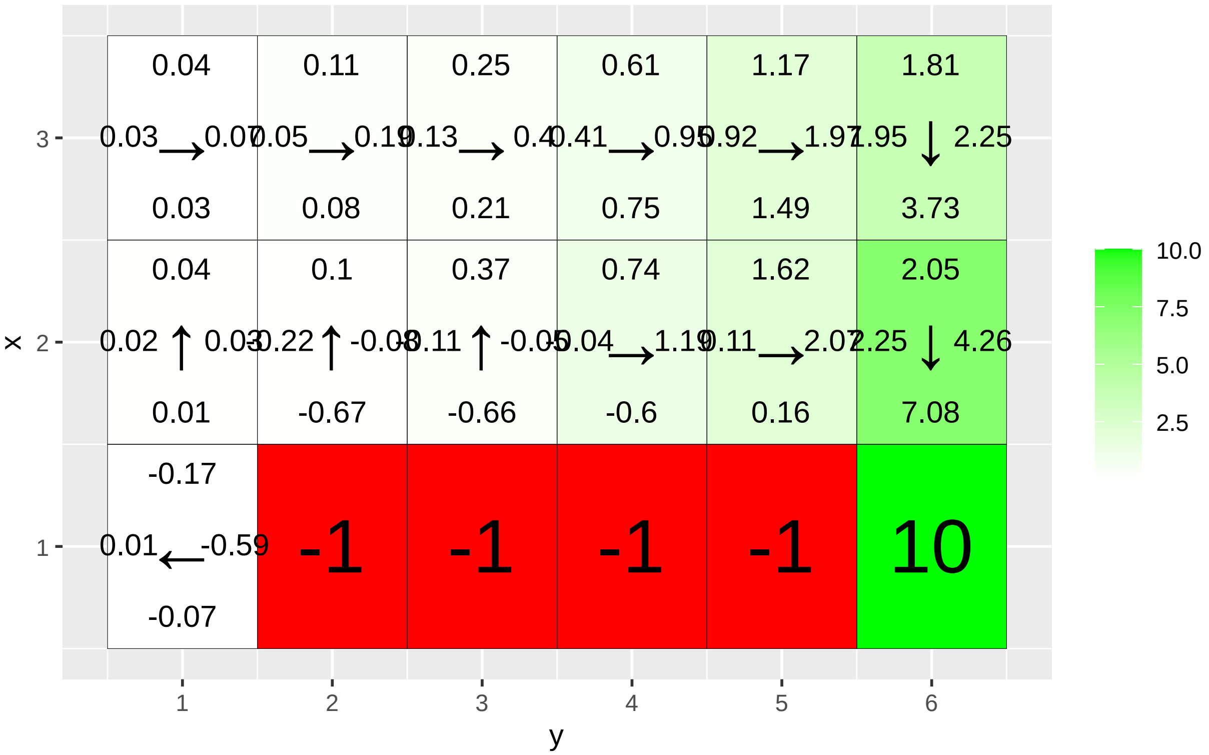


Q-table after 10000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.6 , beta = 0)

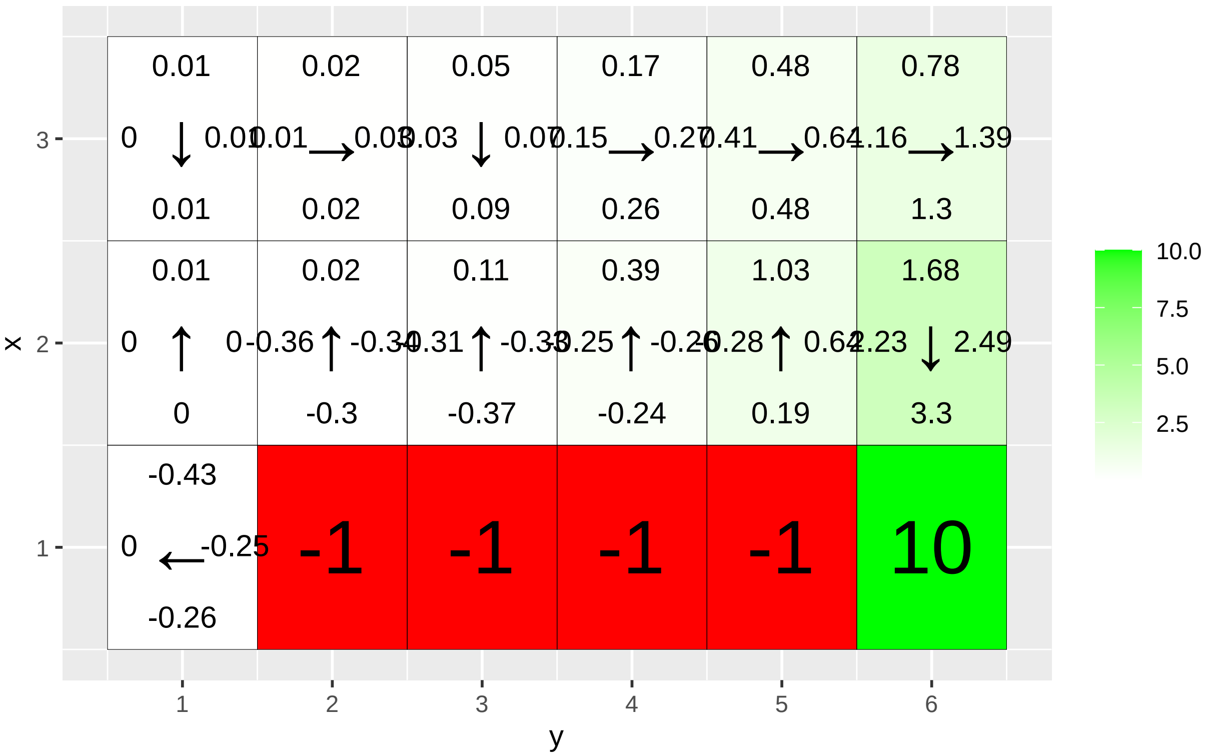


Q-table after 10000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.6 , beta = 0.2)

Q-table after 10000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.6 , beta = 0.4)



Q-table after 10000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.6 , beta = 0.66)



Question 2: Reinforce:

The file RL Lab2.R in the course website contains an implementation of the REINFORCE algorithm. 2 Your task is to run the code provided and answer some questions. Although you do not have to modify the code, you are advised to check it out to familiarize with it. The code uses the R package keras to manipulate neural networks.

We will work with a 4 × 4 grid. We want the agent to learn to navigate to a random goal position in the grid. The agent will start in a random position and it will be told the goal position.
The agent receives a reward of 5 when it reaches the goal. Since the goal position can be any position, we need a way to tell the agent where the goal is. Since our agent does not have any memory mechanism, we provide the goal coordinates as part of the state at every time step, i.e. a state consists now of four coordinates: Two for the position of the agent, and two for the goal position. The actions of the agent can however only impact its own position, i.e. the actions do not modify the goal position. Note that the agent initially does not know that the last two coordinates of a state indicate the position with maximal reward, i.e. the goal position. It has to learn it. It also has to learn a policy to reach the goal position from the initial position. Moreover, the policy has to depend on the goal position, because it is chosen at random in each episode. Since we only have a single non-zero reward, we do not specify a reward map. Instead, the goal coordinates are passed to the functions that need to access the reward function.

Visualizing the environment:

Transition Model:

```
transition_model <- function(x, y, action, beta){

  # Computes the new state after given action is taken. The agent will follow the action
  # with probability (1-beta) and slip to the right or left with probability beta/2 each.
  #
  # Args:
  #   x, y: state coordinates.
  #   action: which action the agent takes (in {1,2,3,4}).
  #   beta: probability of the agent slipping to the side when trying to move.
  #   H, W (global variables): environment dimensions.
  #
  # Returns:
  #   The new state after the action has been taken.

  delta <- sample(-1:1, size = 1, prob = c(0.5*beta,1-beta,0.5*beta))
  final_action <- ((action + delta + 3) %% 4) + 1
  foo <- c(x,y) + unlist(action_deltas[final_action])
  foo <- pmax(c(1,1),pmin(foo,c(H,W)))

  return (foo)
}
```

Deep Policy Distribution:

```
DeepPolicy_dist <- function(x, y, goal_x, goal_y){

  # Get distribution over actions for state (x,y) and goal (goal_x,goal_y) from the deep policy.
  #
  # Args:
  #   x, y: state coordinates.
  #   goal_x, goal_y: goal coordinates.
  #   model (global variable): NN encoding the policy.
  #
  # Returns:
  #   A distribution over actions.

  foo <- matrix(data = c(x,y,goal_x,goal_y), nrow = 1)

  # return (predict_proba(model, x = foo))
  return (predict_on_batch(model, x = foo)) # Faster.

}
```

Deep Policy

```
DeepPolicy <- function(x, y, goal_x, goal_y){

  # Get an action for state (x,y) and goal (goal_x,goal_y) from the deep policy.
  #
  # Args:
  #   x, y: state coordinates.
  #   goal_x, goal_y: goal coordinates.
  #   model (global variable): NN encoding the policy.
  #
  # Returns:
  #   An action, i.e. integer in {1,2,3,4}.

  foo <- DeepPolicy_dist(x,y,goal_x,goal_y)

  return (sample(1:4, size = 1, prob = foo))

}
```

Deep Policy Training

```
DeepPolicy_train <- function(states, actions, goal, gamma){

  # Train the policy network on a rolled out trajectory.
  #
  # Args:
  #   states: array of states visited throughout the trajectory.
  #   actions: array of actions taken throughout the trajectory.
  #   goal: goal coordinates, array with 2 entries.
  #   gamma: discount factor.

  # Construct batch for training.
  inputs <- matrix(data = states, ncol = 2, byrow = TRUE)
  inputs <- cbind(inputs,rep(goal[1],nrow(inputs)))
  inputs <- cbind(inputs,rep(goal[2],nrow(inputs)))

  targets <- array(data = actions, dim = nrow(inputs))
  targets <- to_categorical(targets-1, num_classes = 4)
  # Sample weights. Reward of 5 for reaching the goal.
  weights <- array(data = 5*(gamma^(nrow(inputs)-1)), dim = nrow(inputs))

  # Train on batch. Note that this runs a SINGLE gradient update.
  train_on_batch(model, x = inputs, y = targets, sample_weight = weights)

}
```

Reinforcement Episode

```
reinforce_episode <- function(goal, gamma = 0.95, beta = 0){

  # Rolls out a trajectory in the environment until the goal is reached.
  # Then trains the policy using the collected states, actions and rewards.
  #
  # Args:
  #   goal: goal coordinates, array with 2 entries.
  #   gamma (optional): discount factor.
  #   beta (optional): probability of slipping in the transition model.

  # Randomize starting position.
  cur_pos <- goal
  while(all(cur_pos == goal))
    cur_pos <- c(sample(1:H, size = 1),sample(1:W, size = 1))

  states <- NULL
  actions <- NULL

  steps <- 0 # To avoid getting stuck and/or training on unnecessarily long episodes.
  while(steps < 20){
    steps <- steps+1

    # Follow policy and execute action.
    action <- DeepPolicy(cur_pos[1], cur_pos[2], goal[1], goal[2])
    new_pos <- transition_model(cur_pos[1], cur_pos[2], action, beta)

    # Store states and actions.
    states <- c(states,cur_pos)
    actions <- c(actions,action)
    cur_pos <- new_pos

    if(all(new_pos == goal)){
      # Train network.
      DeepPolicy_train(states,actions,goal,gamma)
      break
    }
  }

}
```

Environment D:

2.6. Environment D. In this task, we will use eight goal positions for training and, then, validate the learned policy on the remaining eight possible goal positions. The training and validation goal positions are stored in the lists train goals and val goals in the code in the file RL Lab2.R. You are requested to run the code provided, which runs the REINFORCE algorithm for 5000 episodes with $\beta = 0$ and $\gamma = 0.95$. 3 Each training episode uses a random goal position from train goals. The initial position for the episode is also chosen at random. When training is completed, the code validates the learned policy for the goal positions in val goals. This is done by with the help of the function vis prob, which shows the grid, goal position and learned policy. Note that each non-terminal tile has four values. These represent the action probabilities associated to the tile (state). Note also that each non-terminal tile has an arrow. This indicates the action with the largest probability for the tile (ties are broken at random). Finally, answer the following questions:

- Has the agent learned a good policy? Why / Why not ?

- Could you have used the Q-learning algorithm to solve this task ?

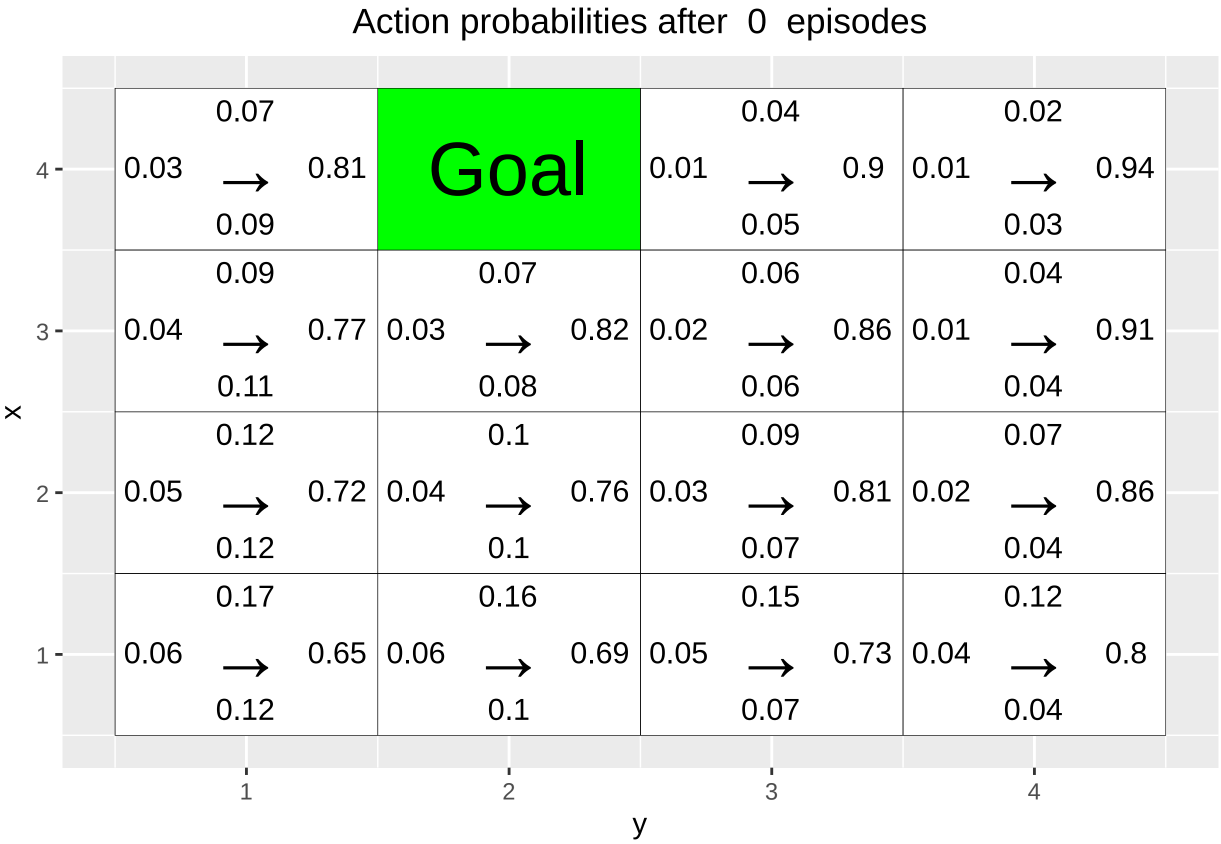
Answer:

Given a training goals: train_goals <- (4,1), (4,3), (3,1), (3,4), (2,1), (2,2), (1,2), (1,3)
and validation goals: val_goals <- (4,2), (4,4), (3,2), (3,3), (2,3), (2,4), (1,1), (1,4)

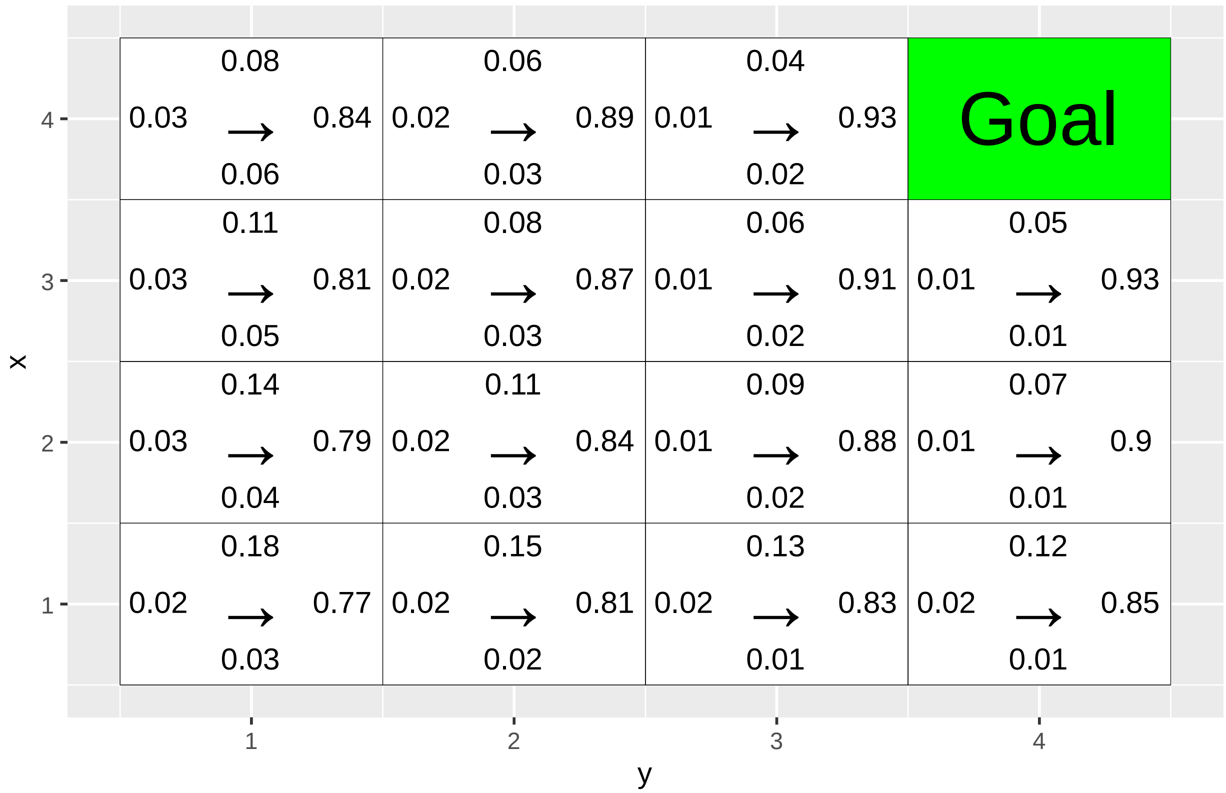
The agent could find its way to the validation goals because it was trained on a training goals that are similar to the validation goals. The training goals gave the agent a good view of the environment from the first, second, third, and fourth rows, and that's why the agent was able to find its way to the validation goals with high probability. In other words, the agent was able to generalize the policy for similar states.

We can't use Q-Learning because it doesn't have the feature of generalization, for instance, if we set a different goal than the one Q-Learning was trained to find, then Q-learning won't be able to reach that goal. Reinforce has the generalization advantage which enables it to determine the action for states that it has not visited before given a similar state. So, if Reinforce visited a state that's similar to the state in interest, then it will give a similarly accurate result for that state.

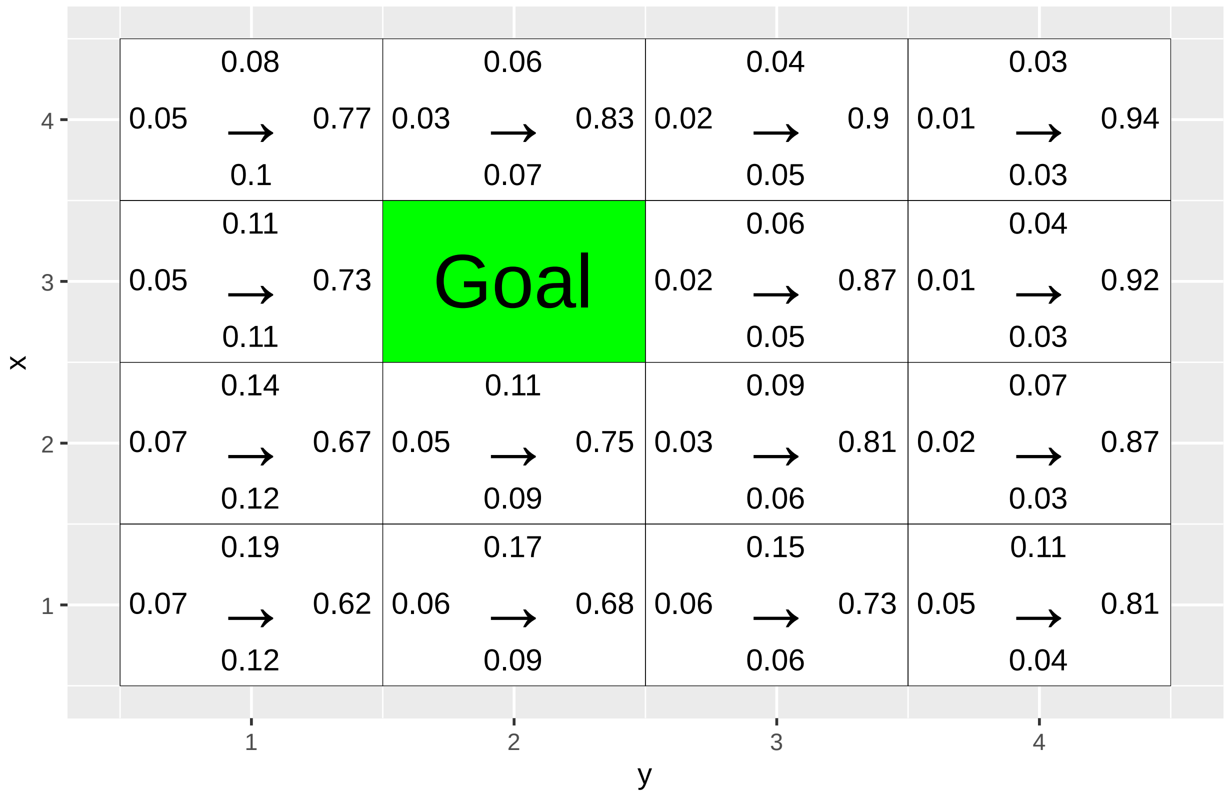
```r
# Environment D (training with random goal positions)

H <- 4
W <- 4

# Define the neural network (two hidden layers of 32 units each).
model <- keras_model_sequential()
model %>%
  layer_dense(units = 32, input_shape = c(4), activation = 'relu') %>%
  layer_dense(units = 32, activation = 'relu') %>%
  layer_dense(units = 4, activation = 'softmax')

compile(model, loss = "categorical_crossentropy", optimizer = optimizer_sgd(lr=0.001))

initial_weights <- get_weights(model)

train_goals <- list(c(4,1), c(4,3), c(3,1), c(3,4), c(2,1), c(2,2), c(1,2), c(1,3))
val_goals <- list(c(4,2), c(4,4), c(3,2), c(3,3), c(2,3), c(2,4), c(1,1), c(1,4))

show_validation <- function(episodes){

  for(goal in val_goals)
    vis_prob(goal, episodes)
}

set_weights(model,initial_weights)

show_validation(0)
```



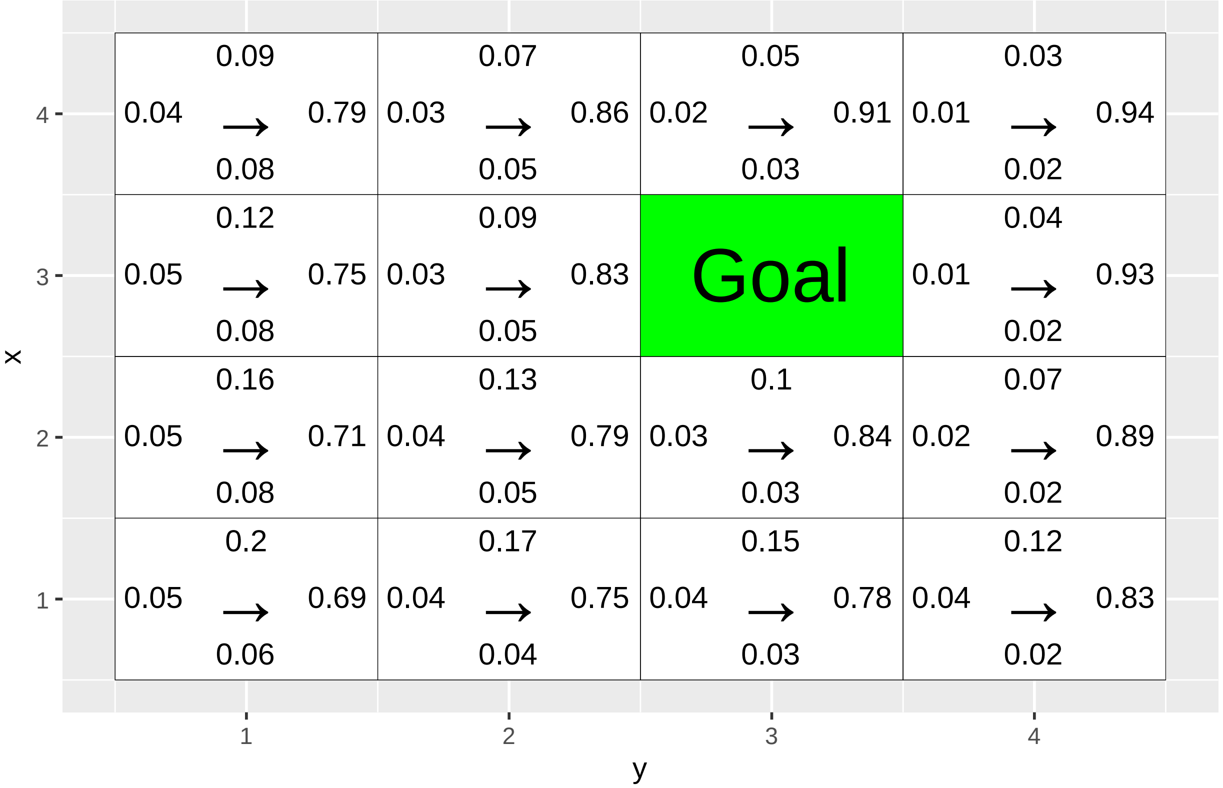
Action probabilities after 0 episodes



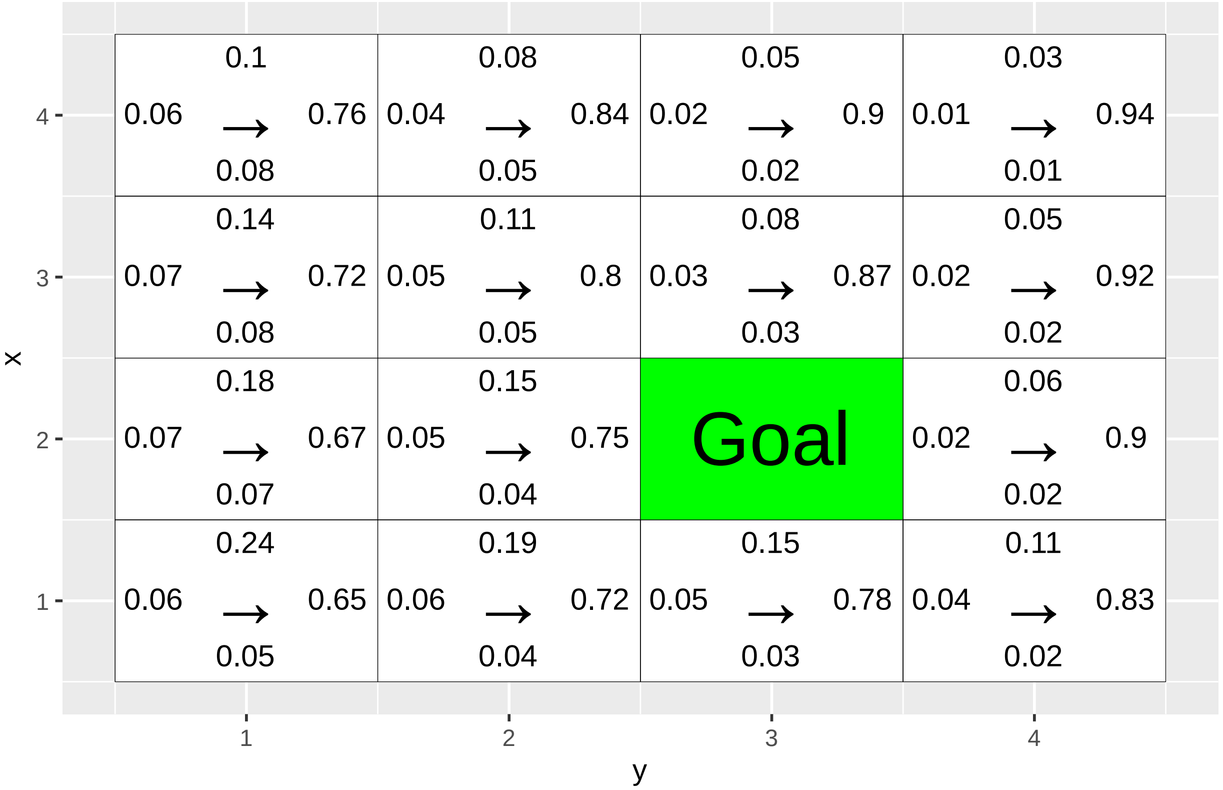
Action probabilities after 0 episodes

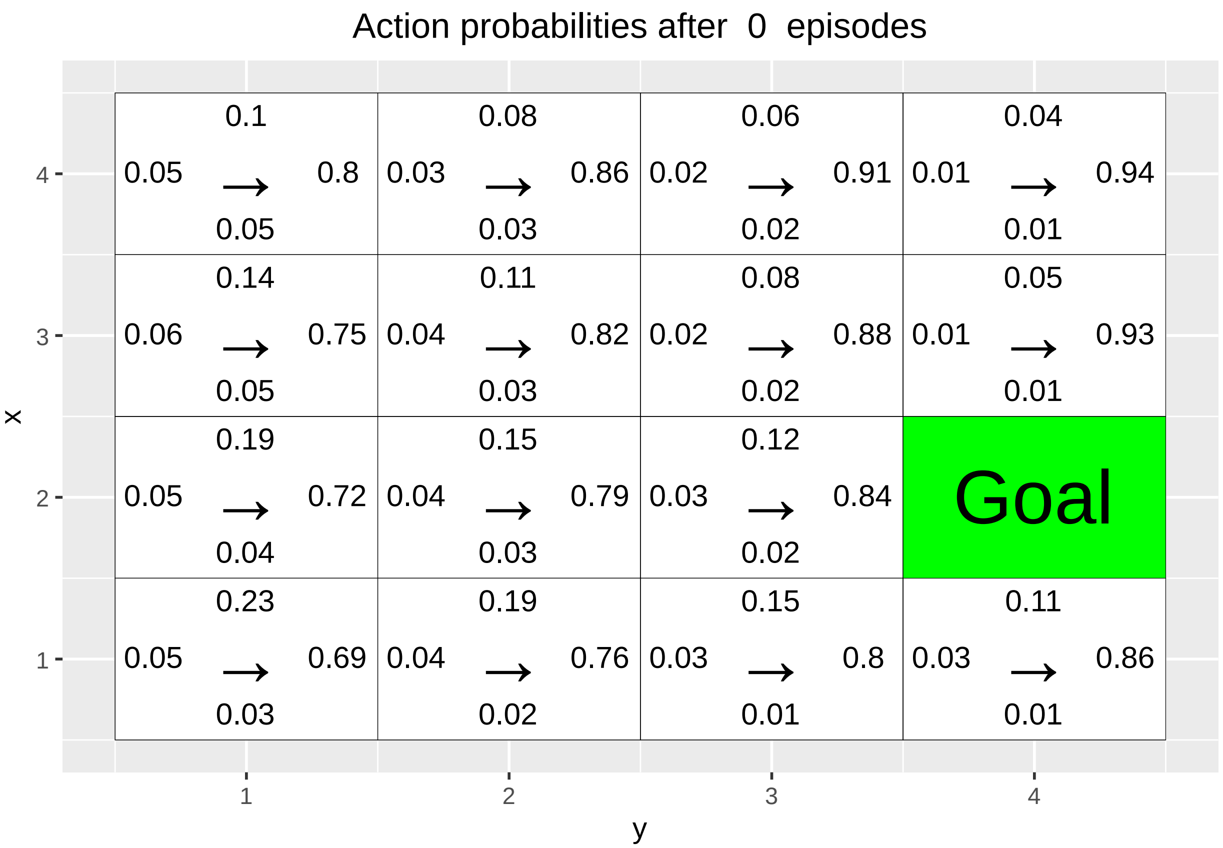


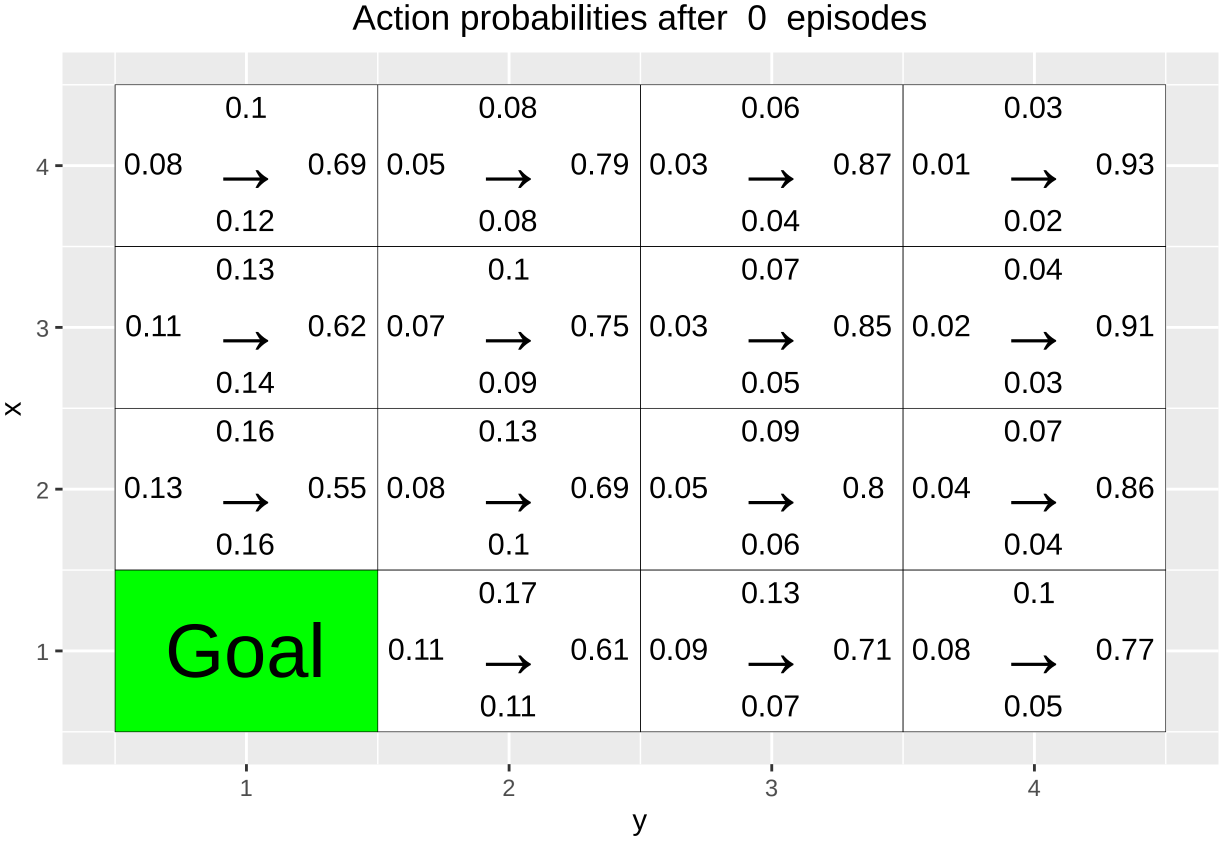
Action probabilities after 0 episodes

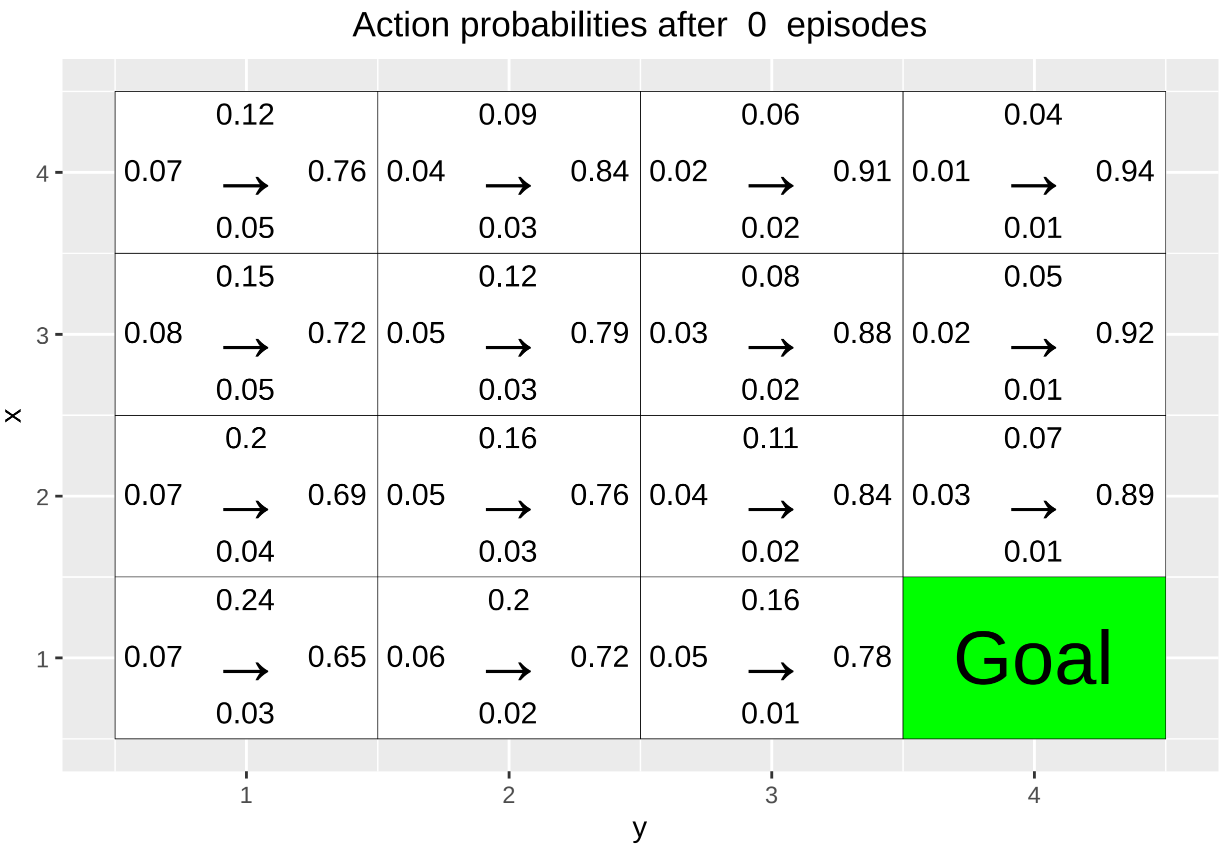


Action probabilities after 0 episodes




Action probabilities after 0 episodes
0.1
0.05 → 0.8
0.05
0.08
0.03 → 0.86
0.03
0.06
0.02 → 0.91
0.02
0.04
0.01 → 0.94
0.01
0.14
0.06 → 0.75
0.05
0.11
0.04 → 0.82
0.03
0.08
0.02 → 0.88
0.02
0.05
0.01 → 0.93
0.01
0.19
0.05 → 0.72
0.04
0.15
0.04 → 0.79
0.03
0.12
0.03 → 0.84
0.02
Goal
0.23
0.05 → 0.69
0.03
0.19
0.04 → 0.76
0.02
0.15
0.03 → 0.8
0.01
0.11
0.03 → 0.86
0.01
4
3
2
1
x
1
2
3
4
y

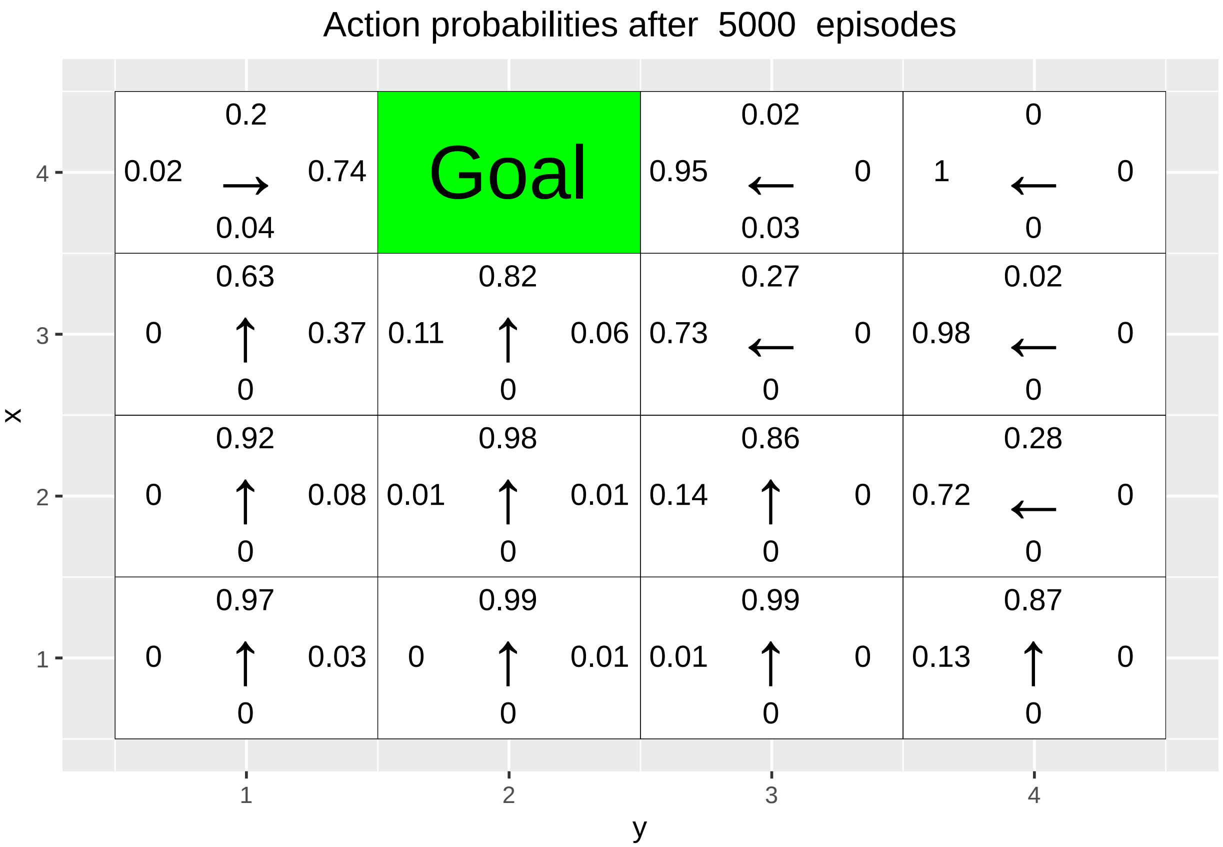

Action probabilities after 0 episodes
0.1
0.08 → 0.69
0.12
0.08
0.05 → 0.79
0.08
0.06
0.03 → 0.87
0.04
0.03
0.01 → 0.93
0.02
0.13
0.11 → 0.62
0.14
0.1
0.07 → 0.75
0.09
0.07
0.03 → 0.85
0.05
0.04
0.02 → 0.91
0.03
0.16
0.13 → 0.55
0.16
0.13
0.08 → 0.69
0.1
0.09
0.05 → 0.8
0.06
0.07
0.04 → 0.86
0.04
Goal
0.17
0.11 → 0.61
0.11
0.13
0.09 → 0.71
0.07
0.1
0.08 → 0.77
0.05
4
3
2
1
x
1
2
3
4
y



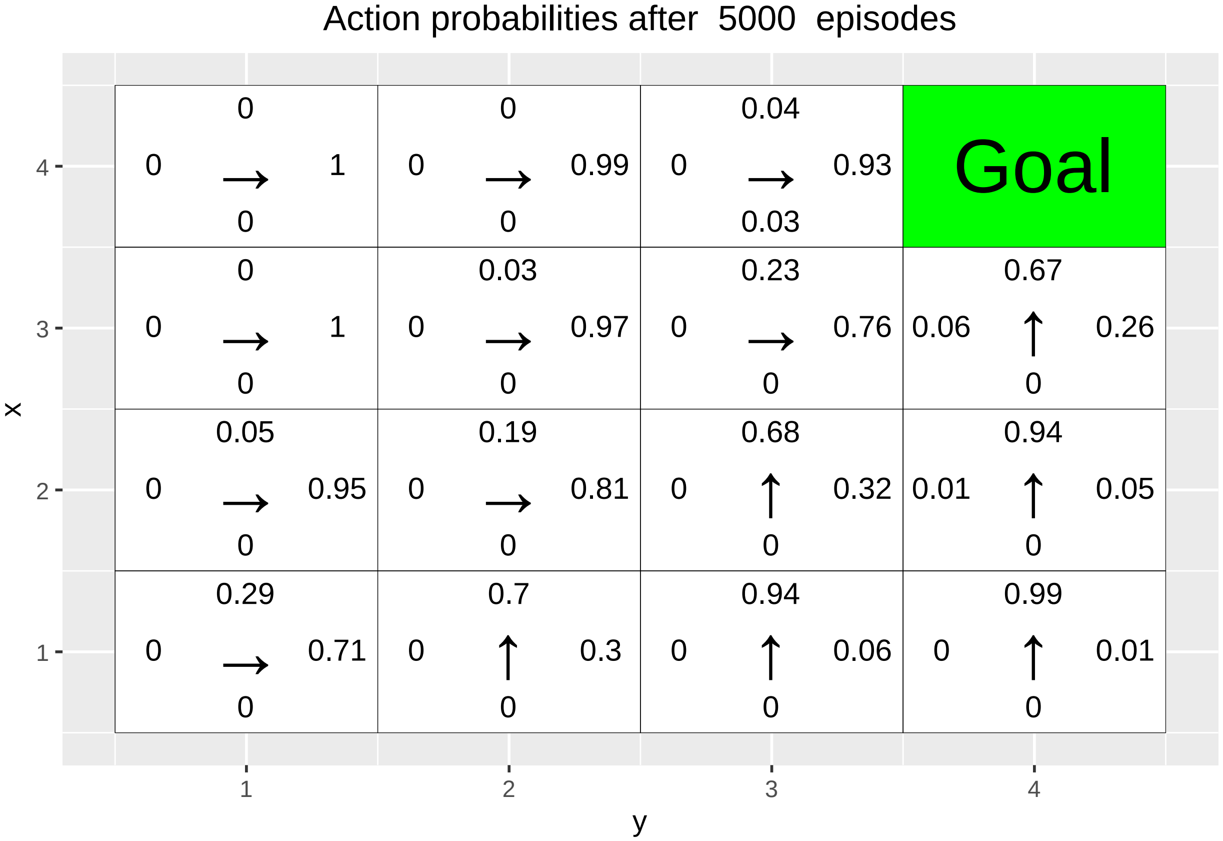
```
for(i in 1:5000){
  if(i%%10==0) cat("episode",i,"\n")
  goal <- sample(train_goals, size = 1)
  reinforce_episode(unlist(goal))
}
```

```
## episode 10
## episode 20
## episode 30
## episode 40
## episode 50
## episode 60
## episode 70
## episode 80
## episode 90
## episode 100
## episode 110
## episode 120
## episode 130
## episode 140
## episode 150
## episode 160
## episode 170
## episode 180
## episode 190
## episode 200
## episode 210
## episode 220
## episode 230
## episode 240
## episode 250
## episode 260
## episode 270
## episode 280
## episode 290
## episode 300
## episode 310
## episode 320
## episode 330
## episode 340
## episode 350
## episode 360
## episode 370
## episode 380
## episode 390
## episode 400
## episode 410
## episode 420
## episode 430
## episode 440
## episode 450
## episode 460
## episode 470
## episode 480
## episode 490
## episode 500
## episode 510
## episode 520
## episode 530
## episode 540
## episode 550
## episode 560
## episode 570
## episode 580
## episode 590
## episode 600
## episode 610
## episode 620
## episode 630
## episode 640
## episode 650
## episode 660
## episode 670
## episode 680
## episode 690
## episode 700
## episode 710
## episode 720
## episode 730
## episode 740
## episode 750
## episode 760
## episode 770
## episode 780
## episode 790
## episode 800
## episode 810
## episode 820
## episode 830
## episode 840
## episode 850
## episode 860
## episode 870
## episode 880
## episode 890
## episode 900
## episode 910
## episode 920
## episode 930
## episode 940
## episode 950
## episode 960
## episode 970
## episode 980
## episode 990
## episode 1000
## episode 1010
## episode 1020
## episode 1030
## episode 1040
## episode 1050
## episode 1060
## episode 1070
## episode 1080
## episode 1090
## episode 1100
## episode 1110
## episode 1120
## episode 1130
## episode 1140
## episode 1150
## episode 1160
## episode 1170
## episode 1180
## episode 1190
## episode 1200
## episode 1210
## episode 1220
## episode 1230
## episode 1240
## episode 1250
## episode 1260
## episode 1270
## episode 1280
## episode 1290
## episode 1300
## episode 1310
## episode 1320
## episode 1330
## episode 1340
## episode 1350
## episode 1360
## episode 1370
## episode 1380
## episode 1390
## episode 1400
## episode 1410
## episode 1420
## episode 1430
## episode 1440
## episode 1450
## episode 1460
## episode 1470
## episode 1480
## episode 1490
## episode 1500
## episode 1510
## episode 1520
## episode 1530
## episode 1540
## episode 1550
## episode 1560
## episode 1570
## episode 1580
## episode 1590
## episode 1600
## episode 1610
## episode 1620
## episode 1630
## episode 1640
## episode 1650
## episode 1660
## episode 1670
## episode 1680
## episode 1690
## episode 1700
## episode 1710
## episode 1720
## episode 1730
## episode 1740
## episode 1750
## episode 1760
## episode 1770
## episode 1780
## episode 1790
## episode 1800
## episode 1810
## episode 1820
## episode 1830
## episode 1840
## episode 1850
## episode 1860
## episode 1870
## episode 1880
## episode 1890
## episode 1900
## episode 1910
## episode 1920
## episode 1930
## episode 1940
## episode 1950
## episode 1960
## episode 1970
## episode 1980
## episode 1990
## episode 2000
## episode 2010
## episode 2020
## episode 2030
## episode 2040
## episode 2050
## episode 2060
## episode 2070
## episode 2080
## episode 2090
## episode 2100
## episode 2110
## episode 2120
## episode 2130
## episode 2140
## episode 2150
## episode 2160
## episode 2170
## episode 2180
## episode 2190
## episode 2200
## episode 2210
## episode 2220
## episode 2230
## episode 2240
## episode 2250
## episode 2260
## episode 2270
## episode 2280
## episode 2290
## episode 2300
## episode 2310
## episode 2320
## episode 2330
## episode 2340
## episode 2350
## episode 2360
## episode 2370
## episode 2380
## episode 2390
## episode 2400
## episode 2410
## episode 2420
## episode 2430
## episode 2440
## episode 2450
## episode 2460
## episode 2470
## episode 2480
## episode 2490
## episode 2500
## episode 2510
## episode 2520
## episode 2530
## episode 2540
## episode 2550
## episode 2560
## episode 2570
## episode 2580
## episode 2590
## episode 2600
## episode 2610
## episode 2620
## episode 2630
## episode 2640
## episode 2650
## episode 2660
## episode 2670
## episode 2680
## episode 2690
## episode 2700
## episode 2710
## episode 2720
## episode 2730
## episode 2740
## episode 2750
## episode 2760
## episode 2770
## episode 2780
## episode 2790
## episode 2800
## episode 2810
## episode 2820
## episode 2830
## episode 2840
## episode 2850
## episode 2860
## episode 2870
## episode 2880
## episode 2890
## episode 2900
## episode 2910
## episode 2920
## episode 2930
## episode 2940
## episode 2950
## episode 2960
## episode 2970
## episode 2980
## episode 2990
## episode 3000
## episode 3010
## episode 3020
## episode 3030
## episode 3040
## episode 3050
## episode 3060
## episode 3070
## episode 3080
## episode 3090
## episode 3100
## episode 3110
## episode 3120
## episode 3130
## episode 3140
## episode 3150
## episode 3160
## episode 3170
## episode 3180
## episode 3190
## episode 3200
## episode 3210
## episode 3220
## episode 3230
## episode 3240
## episode 3250
## episode 3260
## episode 3270
## episode 3280
## episode 3290
## episode 3300
## episode 3310
## episode 3320
## episode 3330
## episode 3340
## episode 3350
## episode 3360
## episode 3370
## episode 3380
## episode 3390
## episode 3400
## episode 3410
## episode 3420
## episode 3430
## episode 3440
## episode 3450
## episode 3460
## episode 3470
## episode 3480
## episode 3490
## episode 3500
## episode 3510
## episode 3520
## episode 3530
## episode 3540
## episode 3550
## episode 3560
## episode 3570
## episode 3580
## episode 3590
## episode 3600
## episode 3610
## episode 3620
## episode 3630
## episode 3640
## episode 3650
## episode 3660
## episode 3670
## episode 3680
## episode 3690
## episode 3700
## episode 3710
## episode 3720
## episode 3730
## episode 3740
## episode 3750
## episode 3760
## episode 3770
## episode 3780
## episode 3790
## episode 3800
## episode 3810
## episode 3820
## episode 3830
## episode 3840
## episode 3850
## episode 3860
## episode 3870
## episode 3880
## episode 3890
## episode 3900
## episode 3910
## episode 3920
## episode 3930
## episode 3940
## episode 3950
## episode 3960
## episode 3970
## episode 3980
## episode 3990
## episode 4000
## episode 4010
## episode 4020
## episode 4030
## episode 4040
## episode 4050
## episode 4060
## episode 4070
## episode 4080
## episode 4090
## episode 4100
## episode 4110
## episode 4120
## episode 4130
## episode 4140
## episode 4150
## episode 4160
## episode 4170
## episode 4180
## episode 4190
## episode 4200
## episode 4210
## episode 4220
## episode 4230
## episode 4240
## episode 4250
## episode 4260
## episode 4270
## episode 4280
## episode 4290
## episode 4300
## episode 4310
## episode 4320
## episode 4330
## episode 4340
## episode 4350
## episode 4360
## episode 4370
## episode 4380
## episode 4390
## episode 4400
## episode 4410
## episode 4420
## episode 4430
## episode 4440
## episode 4450
## episode 4460
## episode 4470
## episode 4480
## episode 4490
## episode 4500
## episode 4510
## episode 4520
## episode 4530
## episode 4540
## episode 4550
## episode 4560
## episode 4570
## episode 4580
## episode 4590
## episode 4600
## episode 4610
## episode 4620
## episode 4630
## episode 4640
## episode 4650
## episode 4660
## episode 4670
## episode 4680
## episode 4690
## episode 4700
## episode 4710
## episode 4720
## episode 4730
## episode 4740
## episode 4750
## episode 4760
## episode 4770
## episode 4780
## episode 4790
## episode 4800
## episode 4810
## episode 4820
## episode 4830
## episode 4840
## episode 4850
## episode 4860
## episode 4870
## episode 4880
## episode 4890
## episode 4900
## episode 4910
## episode 4920
## episode 4930
## episode 4940
## episode 4950
## episode 4960
## episode 4970
## episode 4980
## episode 4990
## episode 5000
```

```r
show_validation(5000)
```

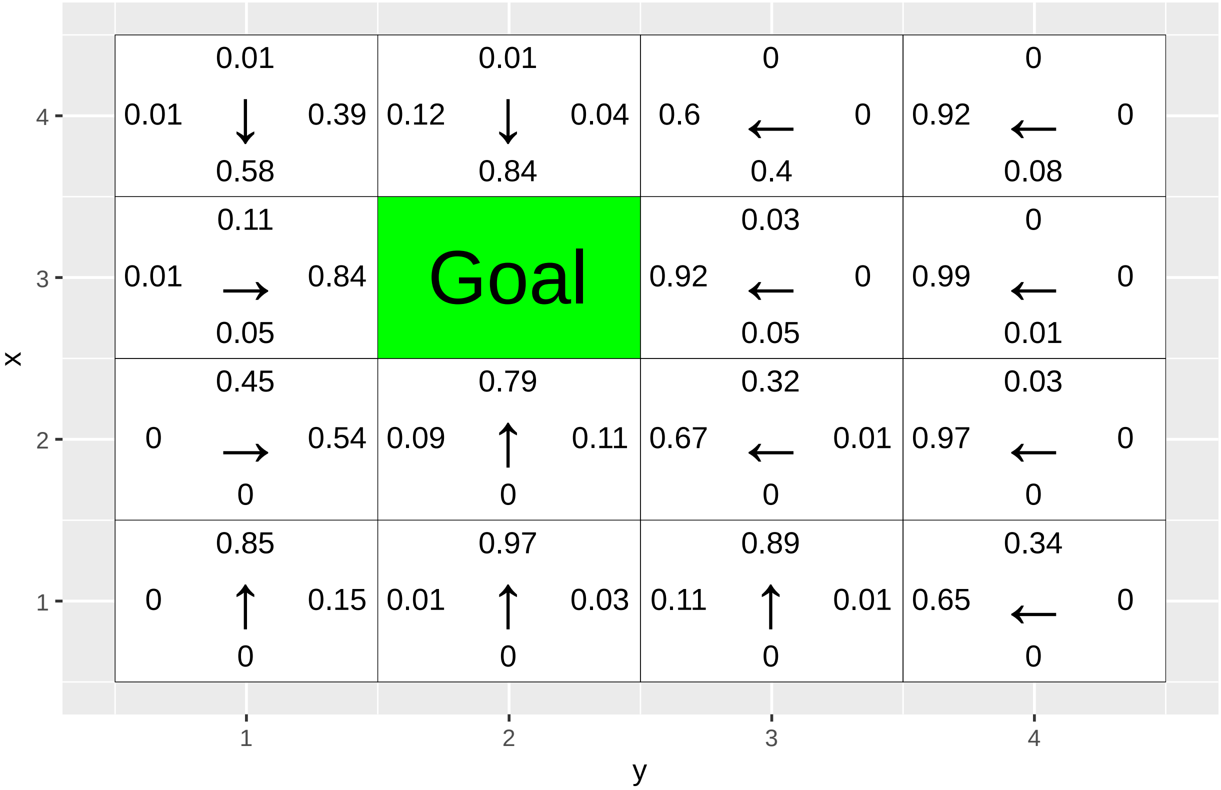


Action probabilities after 5000 episodes

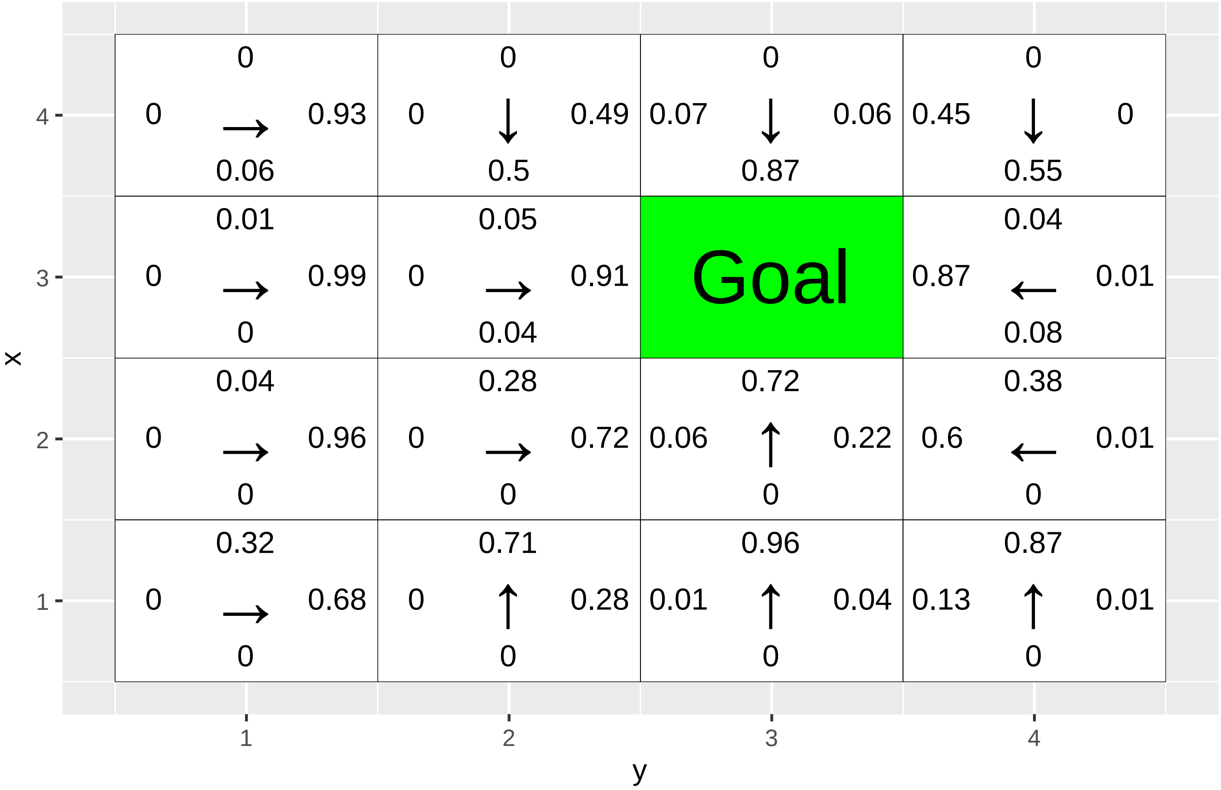


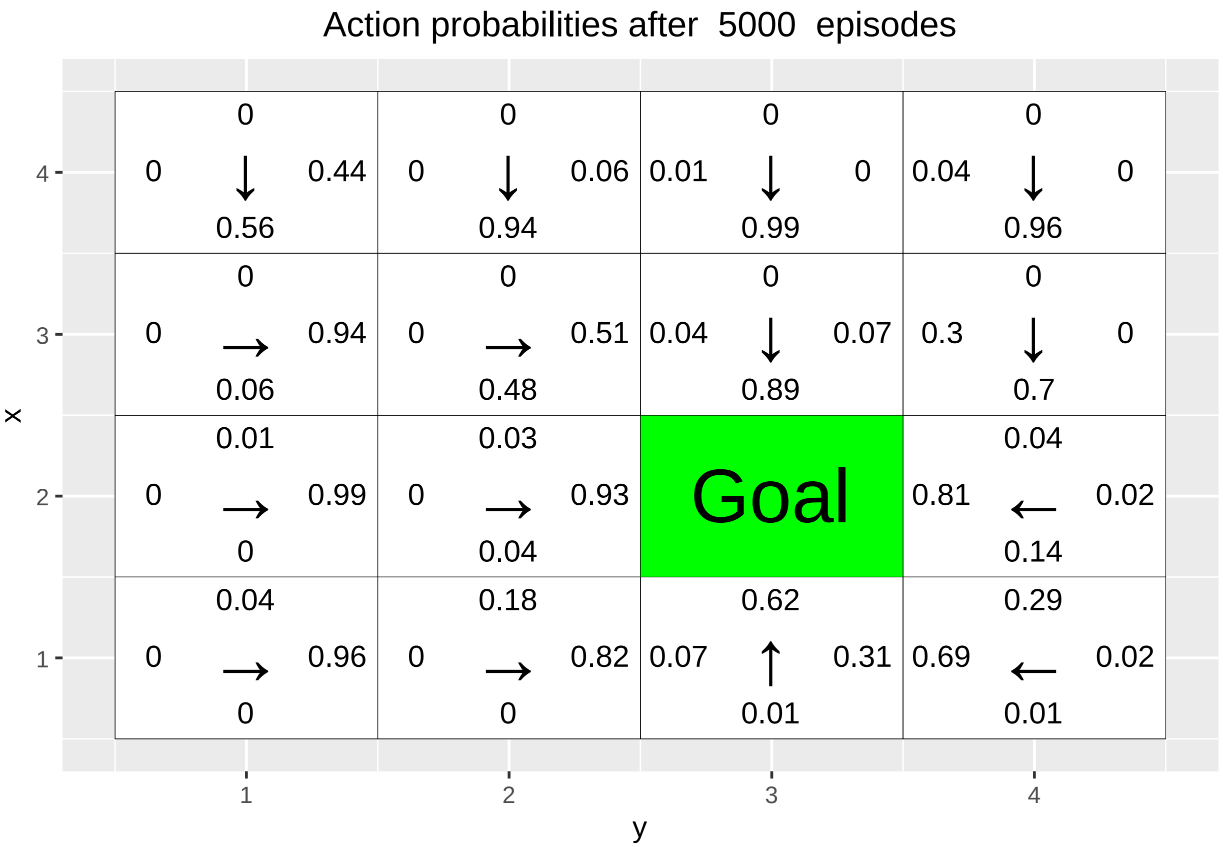
Action probabilities after 5000 episodes

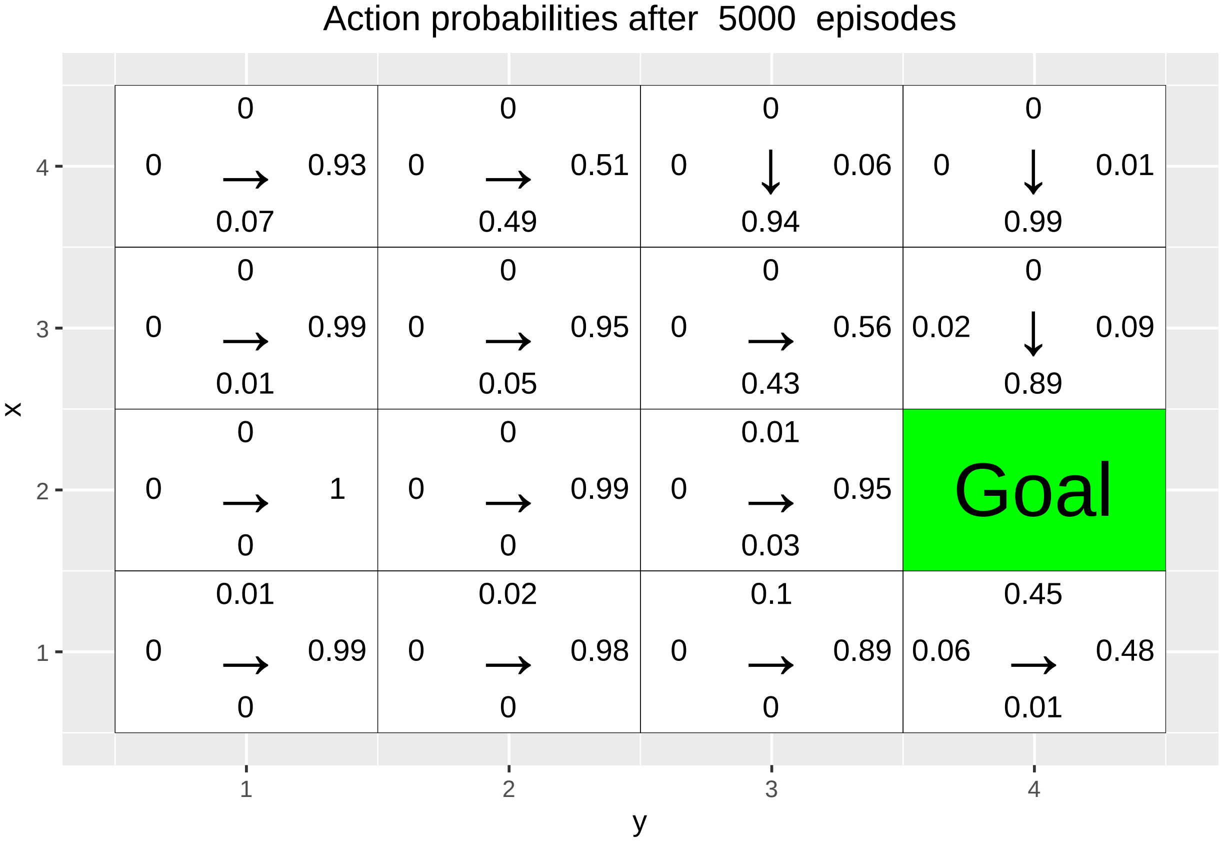
Action probabilities after 5000 episodes



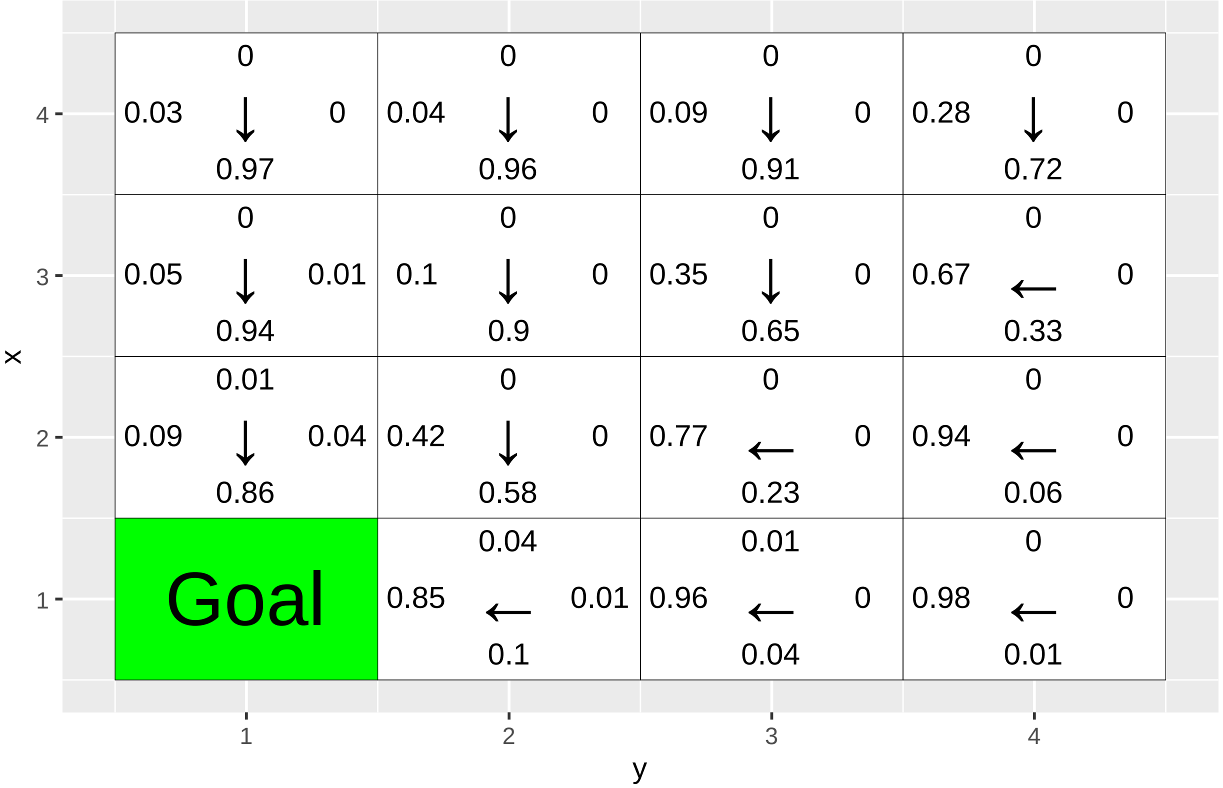
Action probabilities after 5000 episodes



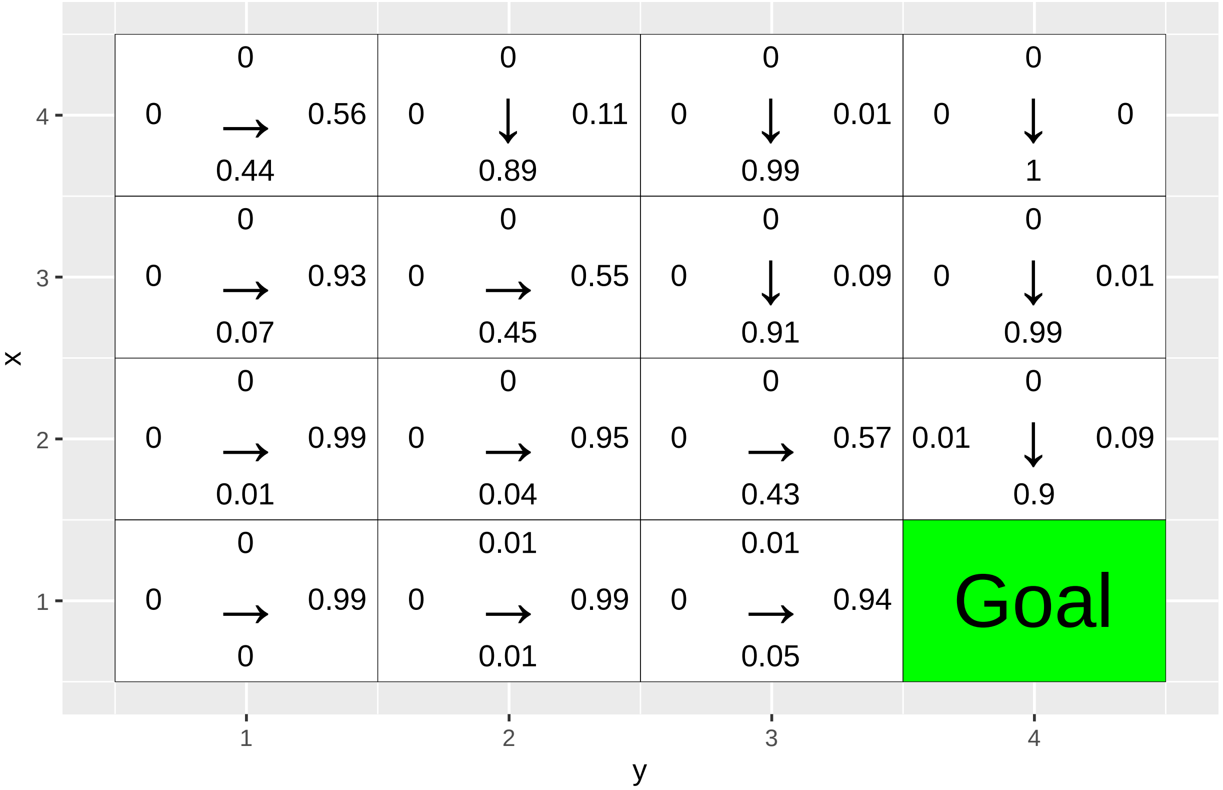

Action probabilities after 5000 episodes


Action probabilities after 5000 episodes

Action probabilities after 5000 episodes



Action probabilities after 5000 episodes



Environment E:

2.7. Environment E. You are now asked to repeat the previous experiments but this time the goals for training are all from the top row of the grid. The validation goals are three positions from the rows below. To solve this task, simply run the code provided in the file RL Lab2.R and answer the following questions:

- Has the agent learned a good policy? Why / Why not ?

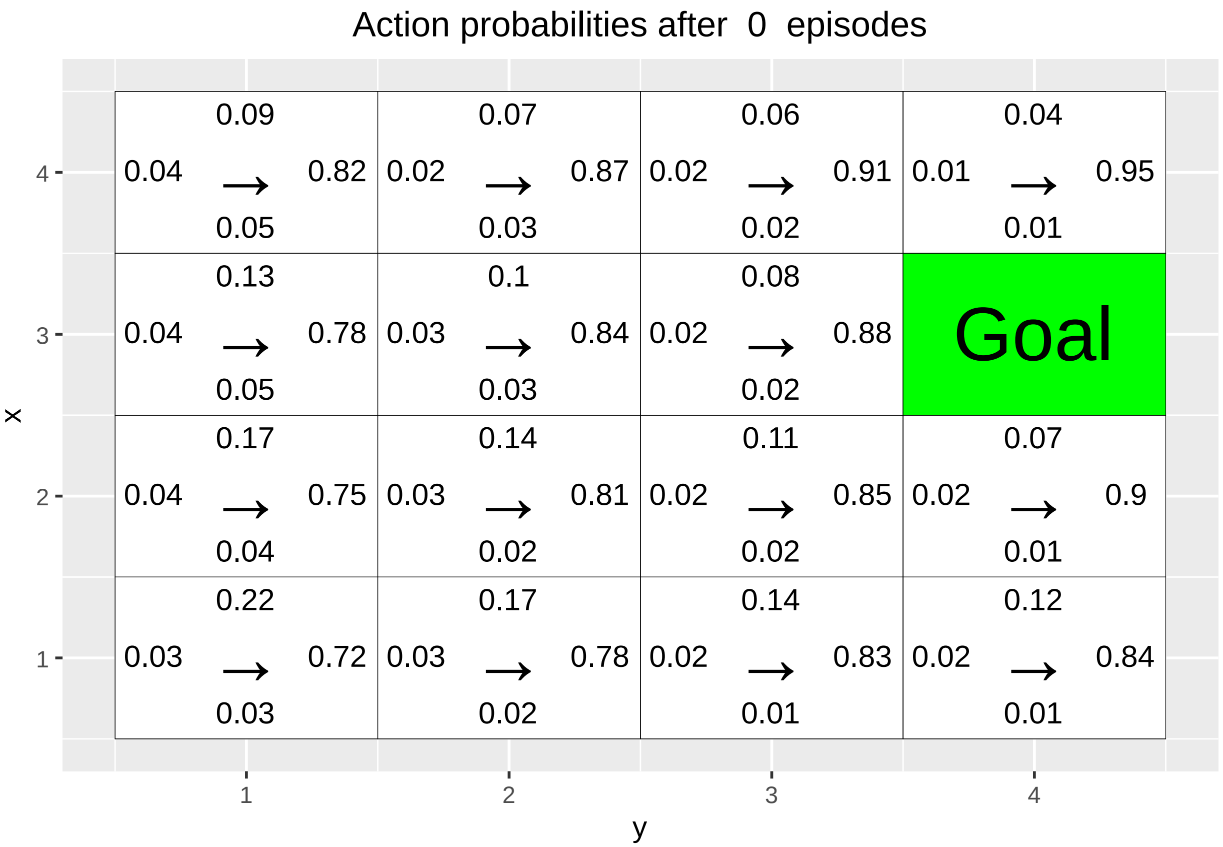
- If the results obtained for environments D and E differ, explain why.

Answer:

The agent couldn't find its way to the validation goals from most of the states because the training was done on the first row and the agent is unable to generalize over a very different goals from the ones it was trained on. The closer the goal to the first row, the more accurately the generalization works. Even though, the agent was able to reach the goal from some of the states, but in most cases (the majority of states), it was unable to find its way to the validation goal.

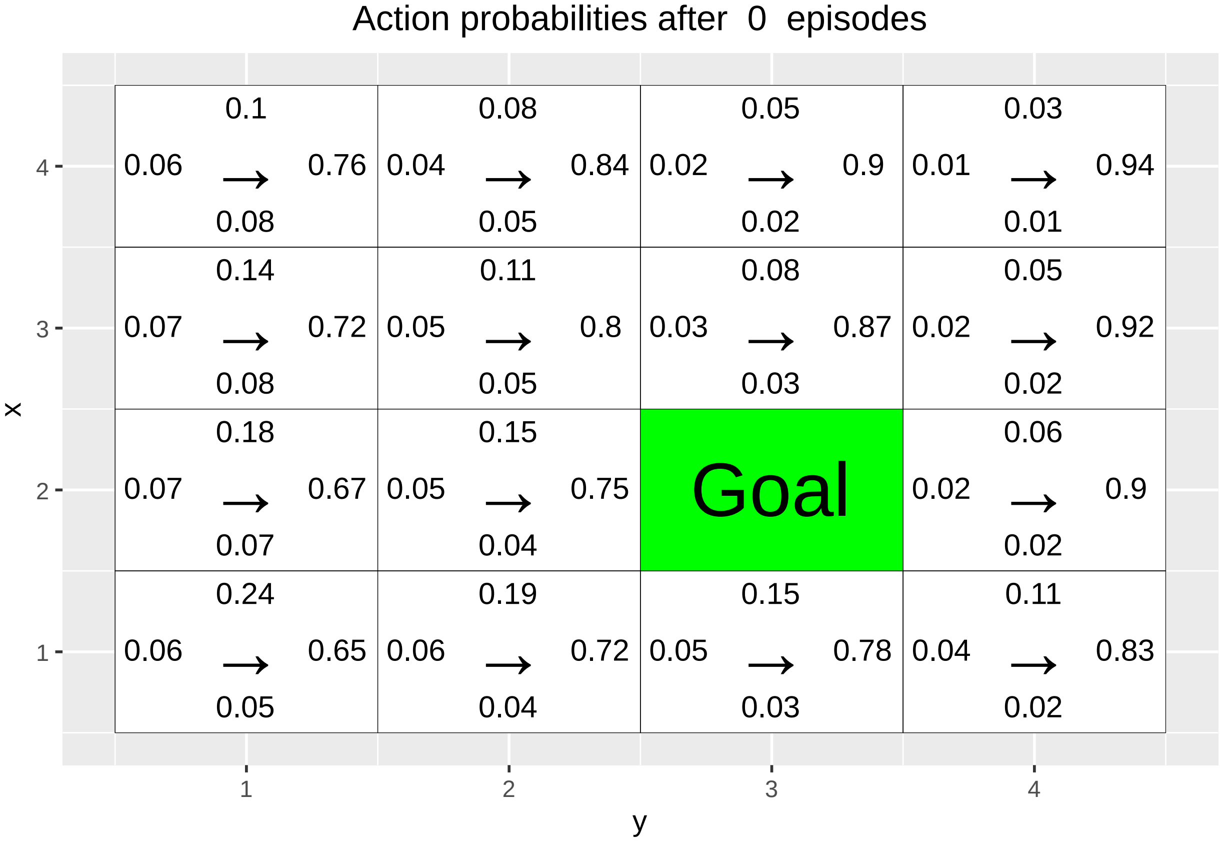
The result of environment E is different from that of D because the training goals are different and the validation goals are different. In D, the agent was trained on random goals from the environment with good exploration of different states, however, in E the agent was trained to reach goals on the first row only, which has made it difficult for it to reach validation goals that are far from the first row (unsimilar to the training goals).

```r
# Environment E (training with top row goal positions)

train_goals <- list(c(4,1), c(4,2), c(4,3), c(4,4))
val_goals <- list(c(3,4), c(2,3), c(1,1))

set_weights(model,initial_weights)

show_validation(0)
```



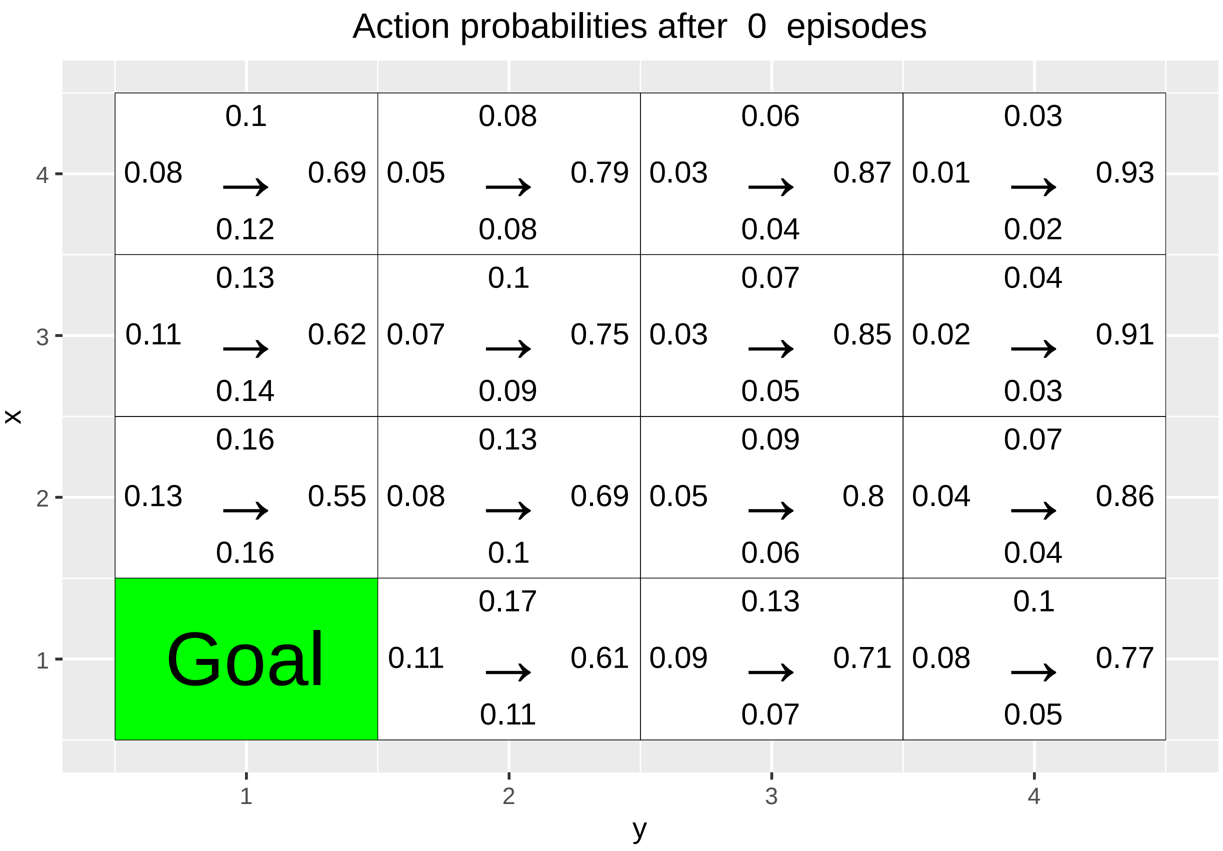
Action probabilities after 0 episodes

	1	2	3	4
4	0.09 0.04 → 0.82 0.05	0.07 0.02 → 0.87 0.03	0.06 0.02 → 0.91 0.02	0.04 0.01 → 0.95 0.01
3	0.13 0.04 → 0.78 0.05	0.1 0.03 → 0.84 0.03	0.08 0.02 → 0.88 0.02	Goal
2	0.17 0.04 → 0.75 0.04	0.14 0.03 → 0.81 0.02	0.11 0.02 → 0.85 0.02	0.07 0.02 → 0.9 0.01
1	0.22 0.03 → 0.72 0.03	0.17 0.03 → 0.78 0.02	0.14 0.02 → 0.83 0.01	0.12 0.02 → 0.84 0.01



Action probabilities after 0 episodes

	1	2	3	4
4	0.1 0.06 → 0.76 0.08	0.08 0.04 → 0.84 0.05	0.05 0.02 → 0.9 0.02	0.03 0.01 → 0.94 0.01
3	0.14 0.07 → 0.72 0.08	0.11 0.05 → 0.8 0.05	0.08 0.03 → 0.87 0.03	0.05 0.02 → 0.92 0.02
2	0.18 0.07 → 0.67 0.07	0.15 0.05 → 0.75 0.04	Goal	0.06 0.02 → 0.9 0.02
1	0.24 0.06 → 0.65 0.05	0.19 0.06 → 0.72 0.04	0.15 0.05 → 0.78 0.03	0.11 0.04 → 0.83 0.02

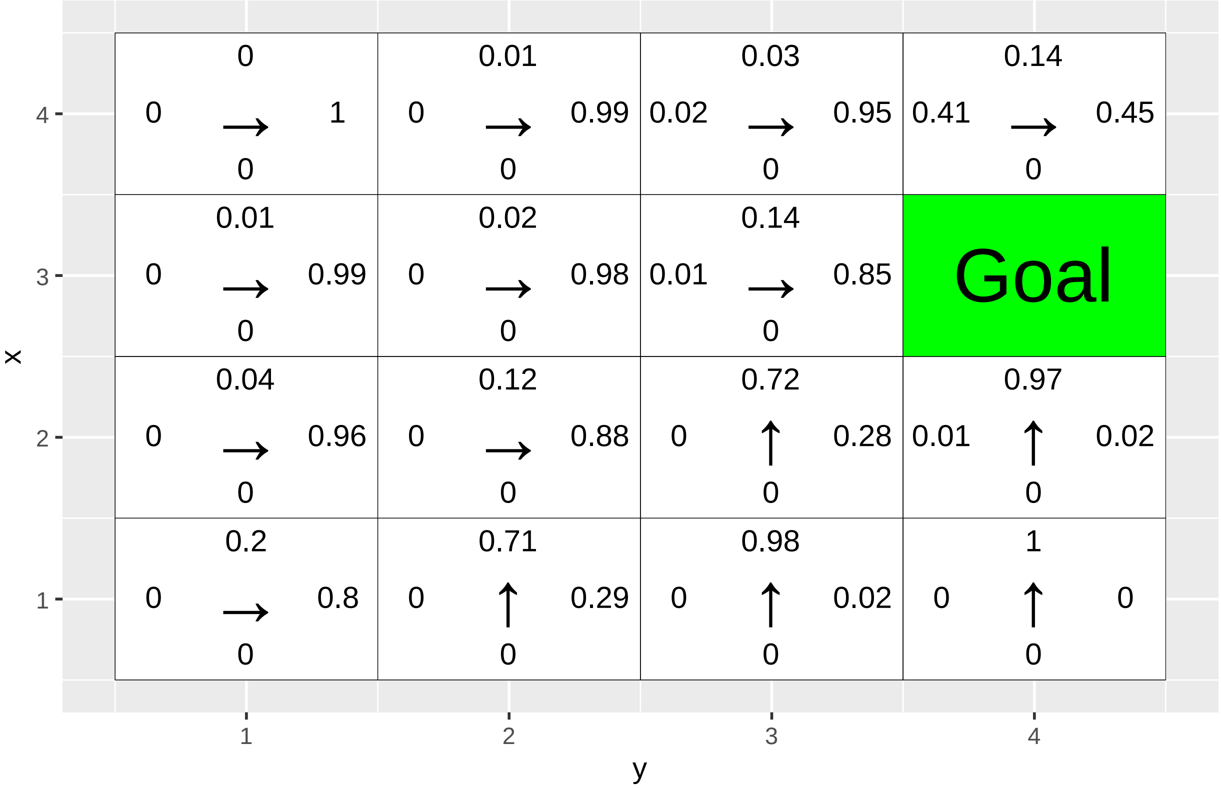


```r
for(i in 1:5000){
  if(i%%10==0) cat("episode", i,"\n")
  goal <- sample(train_goals, size = 1)
  reinforce_episode(unlist(goal))
}
```

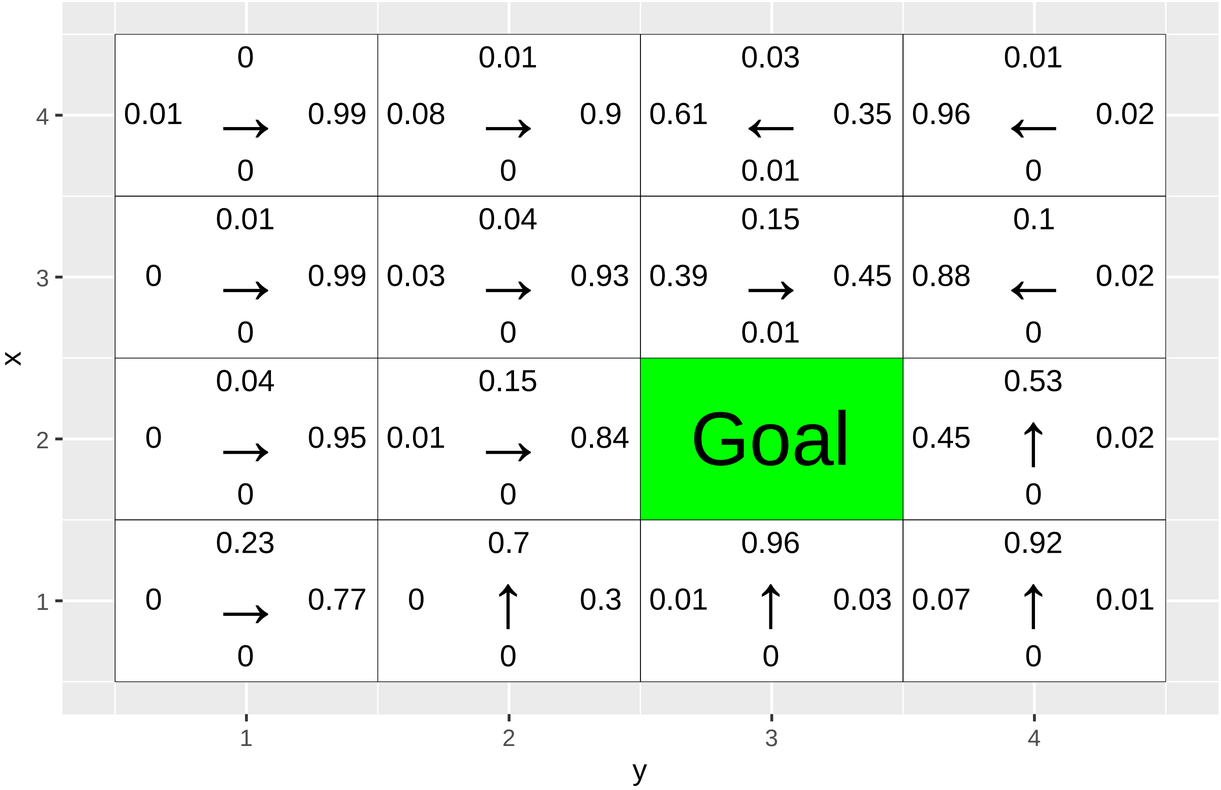
```
## episode 10
## episode 20
## episode 30
## episode 40
## episode 50
## episode 60
## episode 70
## episode 80
## episode 90
## episode 100
## episode 110
## episode 120
## episode 130
## episode 140
## episode 150
## episode 160
## episode 170
## episode 180
## episode 190
## episode 200
## episode 210
## episode 220
## episode 230
## episode 240
## episode 250
## episode 260
## episode 270
## episode 280
## episode 290
## episode 300
## episode 310
## episode 320
## episode 330
## episode 340
## episode 350
## episode 360
## episode 370
## episode 380
## episode 390
## episode 400
## episode 410
## episode 420
## episode 430
## episode 440
## episode 450
## episode 460
## episode 470
## episode 480
## episode 490
## episode 500
## episode 510
## episode 520
## episode 530
## episode 540
## episode 550
## episode 560
## episode 570
## episode 580
## episode 590
## episode 600
## episode 610
## episode 620
## episode 630
## episode 640
## episode 650
## episode 660
## episode 670
## episode 680
## episode 690
## episode 700
## episode 710
## episode 720
## episode 730
## episode 740
## episode 750
## episode 760
## episode 770
## episode 780
## episode 790
## episode 800
## episode 810
## episode 820
## episode 830
## episode 840
## episode 850
## episode 860
## episode 870
## episode 880
## episode 890
## episode 900
## episode 910
## episode 920
## episode 930
## episode 940
## episode 950
## episode 960
## episode 970
## episode 980
## episode 990
## episode 1000
## episode 1010
## episode 1020
## episode 1030
## episode 1040
## episode 1050
## episode 1060
## episode 1070
## episode 1080
## episode 1090
## episode 1100
## episode 1110
## episode 1120
## episode 1130
## episode 1140
## episode 1150
## episode 1160
## episode 1170
## episode 1180
## episode 1190
## episode 1200
## episode 1210
## episode 1220
## episode 1230
## episode 1240
## episode 1250
## episode 1260
## episode 1270
## episode 1280
## episode 1290
## episode 1300
## episode 1310
## episode 1320
## episode 1330
## episode 1340
## episode 1350
## episode 1360
## episode 1370
## episode 1380
## episode 1390
## episode 1400
## episode 1410
## episode 1420
## episode 1430
## episode 1440
## episode 1450
## episode 1460
## episode 1470
## episode 1480
## episode 1490
## episode 1500
## episode 1510
## episode 1520
## episode 1530
## episode 1540
## episode 1550
## episode 1560
## episode 1570
## episode 1580
## episode 1590
## episode 1600
## episode 1610
## episode 1620
## episode 1630
## episode 1640
## episode 1650
## episode 1660
## episode 1670
## episode 1680
## episode 1690
## episode 1700
## episode 1710
## episode 1720
## episode 1730
## episode 1740
## episode 1750
## episode 1760
## episode 1770
## episode 1780
## episode 1790
## episode 1800
## episode 1810
## episode 1820
## episode 1830
## episode 1840
## episode 1850
## episode 1860
## episode 1870
## episode 1880
## episode 1890
## episode 1900
## episode 1910
## episode 1920
## episode 1930
## episode 1940
## episode 1950
## episode 1960
## episode 1970
## episode 1980
## episode 1990
## episode 2000
## episode 2010
## episode 2020
## episode 2030
## episode 2040
## episode 2050
## episode 2060
## episode 2070
## episode 2080
## episode 2090
## episode 2100
## episode 2110
## episode 2120
## episode 2130
## episode 2140
## episode 2150
## episode 2160
## episode 2170
## episode 2180
## episode 2190
## episode 2200
## episode 2210
## episode 2220
## episode 2230
## episode 2240
## episode 2250
## episode 2260
## episode 2270
## episode 2280
## episode 2290
## episode 2300
## episode 2310
## episode 2320
## episode 2330
## episode 2340
## episode 2350
## episode 2360
## episode 2370
## episode 2380
## episode 2390
## episode 2400
## episode 2410
## episode 2420
## episode 2430
## episode 2440
## episode 2450
## episode 2460
## episode 2470
## episode 2480
## episode 2490
## episode 2500
## episode 2510
## episode 2520
## episode 2530
## episode 2540
## episode 2550
## episode 2560
## episode 2570
## episode 2580
## episode 2590
## episode 2600
## episode 2610
## episode 2620
## episode 2630
## episode 2640
## episode 2650
## episode 2660
## episode 2670
## episode 2680
## episode 2690
## episode 2700
## episode 2710
## episode 2720
## episode 2730
## episode 2740
## episode 2750
## episode 2760
## episode 2770
## episode 2780
## episode 2790
## episode 2800
## episode 2810
## episode 2820
## episode 2830
## episode 2840
## episode 2850
## episode 2860
## episode 2870
## episode 2880
## episode 2890
## episode 2900
## episode 2910
## episode 2920
## episode 2930
## episode 2940
## episode 2950
## episode 2960
## episode 2970
## episode 2980
## episode 2990
## episode 3000
## episode 3010
## episode 3020
## episode 3030
## episode 3040
## episode 3050
## episode 3060
## episode 3070
## episode 3080
## episode 3090
## episode 3100
## episode 3110
## episode 3120
## episode 3130
## episode 3140
## episode 3150
## episode 3160
## episode 3170
## episode 3180
## episode 3190
## episode 3200
## episode 3210
## episode 3220
## episode 3230
## episode 3240
## episode 3250
## episode 3260
## episode 3270
## episode 3280
## episode 3290
## episode 3300
## episode 3310
## episode 3320
## episode 3330
## episode 3340
## episode 3350
## episode 3360
## episode 3370
## episode 3380
## episode 3390
## episode 3400
## episode 3410
## episode 3420
## episode 3430
## episode 3440
## episode 3450
## episode 3460
## episode 3470
## episode 3480
## episode 3490
## episode 3500
## episode 3510
## episode 3520
## episode 3530
## episode 3540
## episode 3550
## episode 3560
## episode 3570
## episode 3580
## episode 3590
## episode 3600
## episode 3610
## episode 3620
## episode 3630
## episode 3640
## episode 3650
## episode 3660
## episode 3670
## episode 3680
## episode 3690
## episode 3700
## episode 3710
## episode 3720
## episode 3730
## episode 3740
## episode 3750
## episode 3760
## episode 3770
## episode 3780
## episode 3790
## episode 3800
## episode 3810
## episode 3820
## episode 3830
## episode 3840
## episode 3850
## episode 3860
## episode 3870
## episode 3880
## episode 3890
## episode 3900
## episode 3910
## episode 3920
## episode 3930
## episode 3940
## episode 3950
## episode 3960
## episode 3970
## episode 3980
## episode 3990
## episode 4000
## episode 4010
## episode 4020
## episode 4030
## episode 4040
## episode 4050
## episode 4060
## episode 4070
## episode 4080
## episode 4090
## episode 4100
## episode 4110
## episode 4120
## episode 4130
## episode 4140
## episode 4150
## episode 4160
## episode 4170
## episode 4180
## episode 4190
## episode 4200
## episode 4210
## episode 4220
## episode 4230
## episode 4240
## episode 4250
## episode 4260
## episode 4270
## episode 4280
## episode 4290
## episode 4300
## episode 4310
## episode 4320
## episode 4330
## episode 4340
## episode 4350
## episode 4360
## episode 4370
## episode 4380
## episode 4390
## episode 4400
## episode 4410
## episode 4420
## episode 4430
## episode 4440
## episode 4450
## episode 4460
## episode 4470
## episode 4480
## episode 4490
## episode 4500
## episode 4510
## episode 4520
## episode 4530
## episode 4540
## episode 4550
## episode 4560
## episode 4570
## episode 4580
## episode 4590
## episode 4600
## episode 4610
## episode 4620
## episode 4630
## episode 4640
## episode 4650
## episode 4660
## episode 4670
## episode 4680
## episode 4690
## episode 4700
## episode 4710
## episode 4720
## episode 4730
## episode 4740
## episode 4750
## episode 4760
## episode 4770
## episode 4780
## episode 4790
## episode 4800
## episode 4810
## episode 4820
## episode 4830
## episode 4840
## episode 4850
## episode 4860
## episode 4870
## episode 4880
## episode 4890
## episode 4900
## episode 4910
## episode 4920
## episode 4930
## episode 4940
## episode 4950
## episode 4960
## episode 4970
## episode 4980
## episode 4990
## episode 5000
```

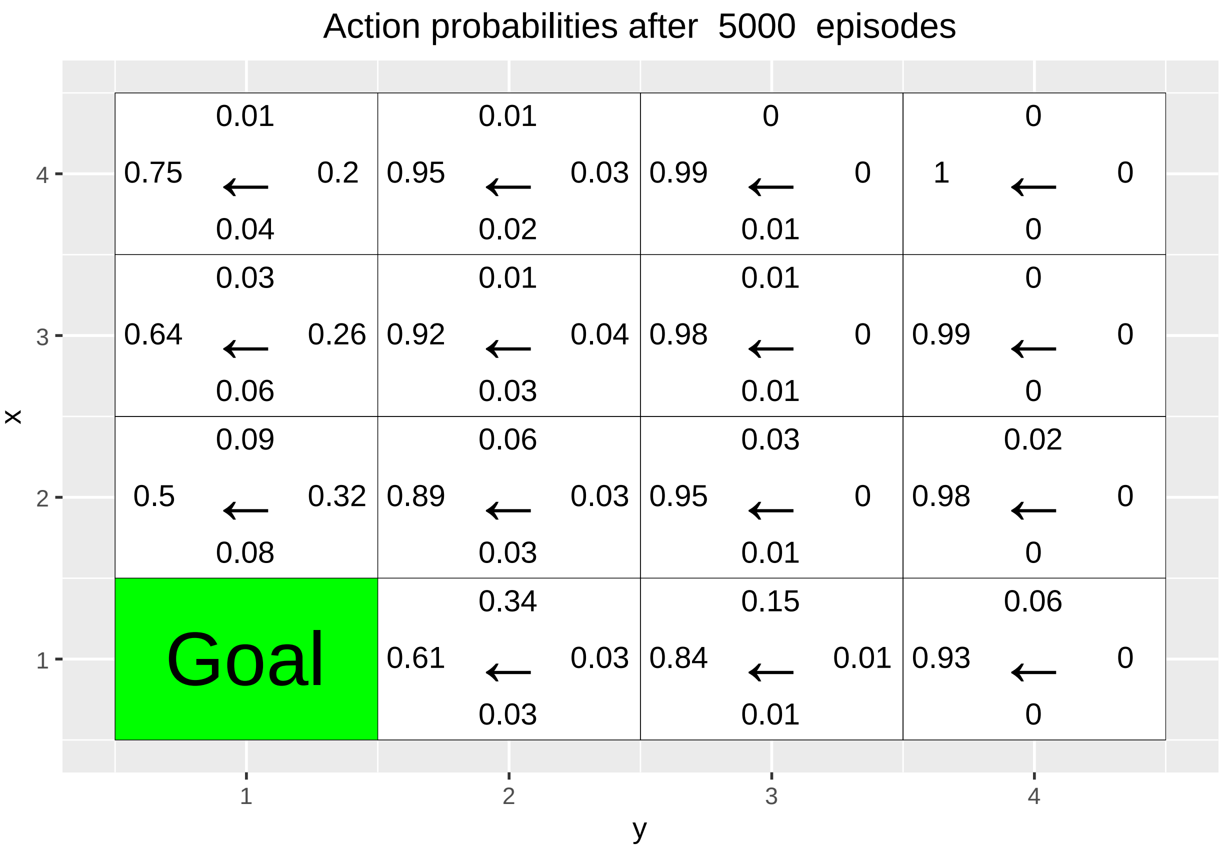
```r
show_validation(5000)
```

Action probabilities after 5000 episodes



Action probabilities after 5000 episodes





References

Course Documents
https://stackoverflow.com/

Appendix

```
RNGversion('3.5.1')
knitr::opts_chunk$set(echo = TRUE)
library(HMM)
library(entropy)
set.seed(12345)
library(ggplot2)

# If you do not see four arrows in line 16, then do the following:
# File/Reopen with Encoding/UTF-8

arrows <- c("↑", "→", "↓", "←")
action_deltas <- list(c(1,0), # up
                      c(0,1), # right
                      c(-1,0), # down
                      c(0,-1)) # left

vis_environment <- function(iterations=0, epsilon = 0.5, alpha = 0.1, gamma = 0.95, beta = 0){
  # Visualize an environment with rewards.
  # Q-values for all actions are displayed on the edges of each tile.
  # The (greedy) policy for each state is also displayed.
  #
  # Args:
  #   iterations, epsilon, alpha, gamma, beta (optional): for the figure title.
  #   reward_map (global variable): a HxW array containing the reward given at each state.
  #   q_table (global variable): a HxWx4 array containing Q-values for each state-action pair.
  #   H, W (global variables): environment dimensions.

  df <- expand.grid(x=1:H,y=1:W)
  foo <- mapply(function(x,y) ifelse(reward_map[x,y] == 0,q_table[x,y,1],NA),df$x,df$y)
  df$val1 <- as.vector(round(foo, 2))
  foo <- mapply(function(x,y) ifelse(reward_map[x,y] == 0,q_table[x,y,2],NA),df$x,df$y)
  df$val2 <- as.vector(round(foo, 2))
  foo <- mapply(function(x,y) ifelse(reward_map[x,y] == 0,q_table[x,y,3],NA),df$x,df$y)
  df$val3 <- as.vector(round(foo, 2))
  foo <- mapply(function(x,y) ifelse(reward_map[x,y] == 0,q_table[x,y,4],NA),df$x,df$y)
  df$val4 <- as.vector(round(foo, 2))
  foo <- mapply(function(x,y)
    ifelse(reward_map[x,y] == 0,arrows[GreedyPolicy(x,y)],reward_map[x,y]),df$x,df$y)
  df$val5 <- as.vector(foo)
  foo <- mapply(function(x,y) ifelse(reward_map[x,y] == 0,max(q_table[x,y,]),
                                     ifelse(reward_map[x,y]<0,NA,reward_map[x,y])),df$x,df$y)
  df$val6 <- as.vector(foo)

  print(ggplot(df,aes(x = y,y = x)) +
          scale_fill_gradient(low = "white", high = "green", na.value = "red", name = "") +
          geom_tile(aes(fill=val6)) +
          geom_text(aes(label = val1),size = 4,nudge_y = .35,na.rm = TRUE) +
          geom_text(aes(label = val2),size = 4,nudge_x = .35,na.rm = TRUE) +
          geom_text(aes(label = val3),size = 4,nudge_y = -.35,na.rm = TRUE) +
          geom_text(aes(label = val4),size = 4,nudge_x = -.35,na.rm = TRUE) +
          geom_text(aes(label = val5),size = 10) +
          geom_tile(fill = 'transparent', colour = 'black') +
          ggtitle(paste("Q-table after ",iterations," iterations\n",
                        "(epsilon = ",epsilon,", alpha = ",alpha,"gamma = ",gamma,", beta = ",beta,")")
          theme(plot.title = element_text(hjust = 0.5)) +
          scale_x_continuous(breaks = c(1:W),labels = c(1:W)) +
          scale_y_continuous(breaks = c(1:H),labels = c(1:H)))

}


GreedyPolicy <- function(x, y){

  # Get a greedy action for state (x,y) from q_table.
  #
  # Args:
  #   x, y: state coordinates.
  #   q_table (global variable): a HxWx4 array containing Q-values for each state-action pair.
  #
  # Returns:
```

```
  #   An action, i.e. integer in {1,2,3,4}.

  # Your code here.
  rewards = q_table[x,y,1:4]
  action = 0
  max_reward = which(rewards == max(rewards))

  if (length(max_reward)>1){
    action = sample(max_reward,1)
  }
  else
  {
    action = max_reward
  }
  return(action)
}


EpsilonGreedyPolicy <- function(x, y, epsilon){

  # Get an epsilon-greedy action for state (x,y) from q_table.
  #
  # Args:
  #   x, y: state coordinates.
  #   epsilon: probability of acting greedily.
  #
  # Returns:
  #   An action, i.e. integer in {1,2,3,4}.

  # Your code here.
  actions = c(1:4)
  action = 0
  rewards = q_table[x,y,1:4]

  max_reward = which(rewards == max(rewards))

  if (length(max_reward)>1){
    action = sample(max_reward,1)
  }
  else
  {
    action = max_reward
  }

  if (1-epsilon < runif(1)){
    return(sample(actions,1))
  }
  else{
    return(action)
  }
}

transition_model <- function(x, y, action, beta){
  # Computes the new state after given action is taken. The agent will follow the action
  # with probability (1-beta) and slip to the right or left with probability beta/2 each.
  #
  # Args:
  #   x, y: state coordinates.
  #   action: which action the agent takes (in {1,2,3,4}).
  #   beta: probability of the agent slipping to the side when trying to move.
  #   H, W (global variables): environment dimensions.
  #
  # Returns:
  #   The new state after the action has been taken.

  delta <- sample(-1:1, size = 1, prob = c(0.5*beta,1-beta,0.5*beta))
  final_action <- ((action + delta + 3) %% 4) + 1
  foo <- c(x,y) + unlist(action_deltas[final_action])
  foo <- pmax(c(1,1),pmin(foo,c(H,W)))

  return (foo)
}
q_learning <- function(start_state, epsilon = 0.5, alpha = 0.1, gamma = 0.95,
                       beta = 0){

  # Perform one episode of Q-learning. The agent should move around in the
  # environment using the given transition model and update the Q-table.
  # The episode ends when the agent reaches a terminal state.
  #
  # Args:
  #   start_state: array with two entries, describing the starting position of the agent.
  #   epsilon (optional): probability of acting greedily.
  #   alpha (optional): learning rate.
  #   gamma (optional): discount factor.
  #   beta (optional): slipping factor.
  #   reward_map (global variable): a HxW array containing the reward given at each state.
  #   q_table (global variable): a HxWx4 array containing Q-values for each state-action pair.
  #
  # Returns:
  #   reward: reward received in the episode.
  #   correction: sum of the temporal difference correction terms over the episode.
  #   q_table (global variable): Recall that R passes arguments by value. So, q_table being
  #   a global variable can be modified with the superassigment operator <<-.

  # Your code here.
  reward = 0
  episode_correction = 0

  repeat{
    # Follow policy, execute action, get reward.

    #Taking an action
    old_x = start_state[1]
    old_y = start_state[2]

    action= EpsilonGreedyPolicy(old_x, old_y, epsilon)
    #Computing the new state after the action is taken
    new_state = transition_model(old_x, old_y, action, beta)
    new_x = new_state[1]
    new_y = new_state[2]

    #Computing the current reward
    current_reward = reward_map[new_x, new_y]

    # Q-table update.
    #Getting a greedy action
    maximum_action = GreedyPolicy(new_x,new_y)

    #Calculating the final reward
    reward = reward+current_reward

    #Calculating the epsilon_correction
    epsilon_correction = current_reward + gamma*q_table[new_x,new_y,maximum_action]-q_table[old_x,old_y

    q_table[old_x,old_y,action] <<- q_table[old_x,old_y,action]+ alpha*epsilon_correction

    start_state = new_state
    if(reward!=0)
      # End episode.
      return (c(reward,epsilon_correction))
  }

}


# Environment A (learning)

H <- 5
W <- 7

reward_map <- matrix(0, nrow = H, ncol = W)
reward_map[3,6] <- 10
reward_map[2:4,3] <- -1

q_table <- array(0,dim = c(H,W,4))

vis_environment()

for(i in 1:10000){
  foo <- q_learning(start_state = c(3,1))

  if(any(i==c(10,100,1000,10000)))
    vis_environment(i)
}


# Environment B (the effect of epsilon and gamma)

H <- 7
```

```r
W <- 8

reward_map <- matrix(0, nrow = H, ncol = W)
reward_map[1,] <- -1
reward_map[7,] <- -1
reward_map[4,5] <- 5
reward_map[4,8] <- 10

q_table <- array(0,dim = c(H,W,4))

vis_environment()

MovingAverage <- function(x, n){

  cx <- c(0,cumsum(x))
  rsum <- (cx[(n+1):length(cx)] - cx[1:(length(cx) - n)]) / n

  return (rsum)
}

for(j in c(0.5,0.75,0.95)){
  q_table <- array(0,dim = c(H,W,4))
  reward <- NULL
  correction <- NULL

  for(i in 1:30000){
    foo <- q_learning(gamma = j, start_state = c(4,1))
    reward <- c(reward,foo[1])
    correction <- c(correction,foo[2])
  }

  vis_environment(i, gamma = j)
  plot(MovingAverage(reward,100),type = "l")
  plot(MovingAverage(correction,100),type = "l")
}

for(j in c(0.5,0.75,0.95)){
  q_table <- array(0,dim = c(H,W,4))
  reward <- NULL
  correction <- NULL

  for(i in 1:30000){
    foo <- q_learning(epsilon = 0.1, gamma = j, start_state = c(4,1))
    reward <- c(reward,foo[1])
    correction <- c(correction,foo[2])
  }

  vis_environment(i, epsilon = 0.1, gamma = j)
  plot(MovingAverage(reward,100),type = "l")
  plot(MovingAverage(correction,100),type = "l")
}

# Environment C (the effect of beta).
```

```r
H <- 3
W <- 6

reward_map <- matrix(0, nrow = H, ncol = W)
reward_map[1,2:5] <- -1
reward_map[1,6] <- 10

q_table <- array(0,dim = c(H,W,4))

vis_environment()

for(j in c(0,0.2,0.4,0.66)){
  q_table <- array(0,dim = c(H,W,4))

  for(i in 1:10000)
    foo <- q_learning(gamma = 0.6, beta = j, start_state = c(1,1))

  vis_environment(i, gamma = 0.6, beta = j)
}
library(keras)

# install.packages("ggplot2")
# install.packages("vctrs")
library(ggplot2)

# If you do not see four arrows in line 19, then do the following:
# File/Reopen with Encoding/UTF-8

arrows <- c("↑", "→", "↓", "←")
action_deltas <- list(c(1,0), # up
                      c(0,1), # right
                      c(-1,0), # down
                      c(0,-1)) # left

vis_prob <- function(goal, episodes = 0){

  # Visualize an environment with rewards.
  # Probabilities for all actions are displayed on the edges of each tile.
  # The (greedy) policy for each state is also displayed.
  #
  # Args:
  #   goal: goal coordinates, array with 2 entries.
  #   episodes, epsilon, alpha, gamma, beta (optional): for the figure title.
  #   H, W (global variables): environment dimensions.

  df <- expand.grid(x=1:H,y=1:W)
  dist <- array(data = NA, dim = c(H,W,4))
  class <- array(data = NA, dim = c(H,W))
  for(i in 1:H)
    for(j in 1:W){
      dist[i,j,] <- DeepPolicy_dist(i,j,goal[1],goal[2])
      foo <- which(dist[i,j,]==max(dist[i,j,]))
      class[i,j] <- ifelse(length(foo)>1,sample(foo, size = 1),foo)
    }

  foo <- mapply(function(x,y) ifelse(all(c(x,y) == goal),NA,dist[x,y,1]),df$x,df$y)
  df$val1 <- as.vector(round(foo, 2))
  foo <- mapply(function(x,y) ifelse(all(c(x,y) == goal),NA,dist[x,y,2]),df$x,df$y)
  df$val2 <- as.vector(round(foo, 2))
  foo <- mapply(function(x,y) ifelse(all(c(x,y) == goal),NA,dist[x,y,3]),df$x,df$y)
  df$val3 <- as.vector(round(foo, 2))
  foo <- mapply(function(x,y) ifelse(all(c(x,y) == goal),NA,dist[x,y,4]),df$x,df$y)
  df$val4 <- as.vector(round(foo, 2))
  foo <- mapply(function(x,y) ifelse(all(c(x,y) == goal),NA,class[x,y]),df$x,df$y)
  df$val5 <- as.vector(arrows[foo])
  foo <- mapply(function(x,y) ifelse(all(c(x,y) == goal),"Goal",NA),df$x,df$y)
  df$val6 <- as.vector(foo)

  print(ggplot(df,aes(x = y,y = x)) +
          geom_tile(fill = 'white', colour = 'black') +
          scale_fill_manual(values = c('green')) +
          geom_tile(aes(fill=val6), show.legend = FALSE, colour = 'black') +
          geom_text(aes(label = val1),size = 4,nudge_y = .35,na.rm = TRUE) +
          geom_text(aes(label = val2),size = 4,nudge_x = .35,na.rm = TRUE) +
          geom_text(aes(label = val3),size = 4,nudge_y = -.35,na.rm = TRUE) +
          geom_text(aes(label = val4),size = 4,nudge_x = -.35,na.rm = TRUE) +
          geom_text(aes(label = val5),size = 10,na.rm = TRUE) +
          geom_text(aes(label = val6),size = 10,na.rm = TRUE) +
          ggtitle(paste("Action probabilities after ",episodes," episodes")) +
          theme(plot.title = element_text(hjust = 0.5)) +
          scale_x_continuous(breaks = c(1:W),labels = c(1:W)) +
          scale_y_continuous(breaks = c(1:H),labels = c(1:H)))

}
transition_model <- function(x, y, action, beta){

  # Computes the new state after given action is taken. The agent will follow the action
  # with probability (1-beta) and slip to the right or left with probability beta/2 each.
  #
  # Args:
  #   x, y: state coordinates.
  #   action: which action the agent takes (in {1,2,3,4}).
  #   beta: probability of the agent slipping to the side when trying to move.
  #   H, W (global variables): environment dimensions.
  #
  # Returns:
  #   The new state after the action has been taken.

  delta <- sample(-1:1, size = 1, prob = c(0.5*beta,1-beta,0.5*beta))
  final_action <- ((action + delta + 3) %% 4) + 1
  foo <- c(x,y) + unlist(action_deltas[final_action])
  foo <- pmax(c(1,1),pmin(foo,c(H,W)))

  return (foo)
}
DeepPolicy_dist <- function(x, y, goal_x, goal_y){
  # Get distribution over actions for state (x,y) and goal (goal_x,goal_y) from the deep policy.
  #
  # Args:
  #   x, y: state coordinates.
  #   goal_x, goal_y: goal coordinates.
  #   model (global variable): NN encoding the policy.
  #
  # Returns:
  #   A distribution over actions.

  foo <- matrix(data = c(x,y,goal_x,goal_y), nrow = 1)

  # return (predict_proba(model, x = foo))
  return (predict_on_batch(model, x = foo)) # Faster.

}
DeepPolicy <- function(x, y, goal_x, goal_y){

  # Get an action for state (x,y) and goal (goal_x,goal_y) from the deep policy.
  #
  # Args:
  #   x, y: state coordinates.
  #   goal_x, goal_y: goal coordinates.
  #   model (global variable): NN encoding the policy.
  #
  # Returns:
  #   An action, i.e. integer in {1,2,3,4}.

  foo <- DeepPolicy_dist(x,y,goal_x,goal_y)

  return (sample(1:4, size = 1, prob = foo))

}

DeepPolicy_train <- function(states, actions, goal, gamma){

  # Train the policy network on a rolled out trajectory.
  #
  # Args:
  #   states: array of states visited throughout the trajectory.
  #   actions: array of actions taken throughout the trajectory.
  #   goal: goal coordinates, array with 2 entries.
  #   gamma: discount factor.

  # Construct batch for training.
  inputs <- matrix(data = states, ncol = 2, byrow = TRUE)
  inputs <- cbind(inputs,rep(goal[1],nrow(inputs)))
  inputs <- cbind(inputs,rep(goal[2],nrow(inputs)))

  targets <- array(data = actions, dim = nrow(inputs))
  targets <- to_categorical(targets-1, num_classes = 4)

  # Sample weights. Reward of 5 for reaching the goal.
```

```r
  weights <- array(data = 5*(gamma^(nrow(inputs)-1)), dim = nrow(inputs))

  # Train on batch. Note that this runs a SINGLE gradient update.
  train_on_batch(model, x = inputs, y = targets, sample_weight = weights)

}

reinforce_episode <- function(goal, gamma = 0.95, beta = 0){

  # Rolls out a trajectory in the environment until the goal is reached.
  # Then trains the policy using the collected states, actions and rewards.
  #
  # Args:
  #   goal: goal coordinates, array with 2 entries.
  #   gamma (optional): discount factor.
  #   beta (optional): probability of slipping in the transition model.

  # Randomize starting position.
  cur_pos <- goal
  while(all(cur_pos == goal))
    cur_pos <- c(sample(1:H, size = 1),sample(1:W, size = 1))

  states <- NULL
  actions <- NULL

  steps <- 0 # To avoid getting stuck and/or training on unnecessarily long episodes.
  while(steps < 20){
    steps <- steps+1

    # Follow policy and execute action.
    action <- DeepPolicy(cur_pos[1], cur_pos[2], goal[1], goal[2])
    new_pos <- transition_model(cur_pos[1], cur_pos[2], action, beta)

    # Store states and actions.
    states <- c(states,cur_pos)
    actions <- c(actions,action)
    cur_pos <- new_pos

    if(all(new_pos == goal)){
      # Train network.
      DeepPolicy_train(states,actions,goal,gamma)
      break
    }
  }

}

# Environment D (training with random goal positions)

H <- 4
W <- 4

# Define the neural network (two hidden layers of 32 units each).
```

```r
model <- keras_model_sequential()
model %>%
  layer_dense(units = 32, input_shape = c(4), activation = 'relu') %>%
  layer_dense(units = 32, activation = 'relu') %>%
  layer_dense(units = 4, activation = 'softmax')

compile(model, loss = "categorical_crossentropy", optimizer = optimizer_sgd(lr=0.001))

initial_weights <- get_weights(model)

train_goals <- list(c(4,1), c(4,3), c(3,1), c(3,4), c(2,1), c(2,2), c(1,2), c(1,3))
val_goals <- list(c(4,2), c(4,4), c(3,2), c(3,3), c(2,3), c(2,4), c(1,1), c(1,4))

show_validation <- function(episodes){

  for(goal in val_goals)
    vis_prob(goal, episodes)

}

set_weights(model,initial_weights)

show_validation(0)

for(i in 1:5000){
  if(i%%10==0) cat("episode",i,"\n")
  goal <- sample(train_goals, size = 1)
  reinforce_episode(unlist(goal))
}

show_validation(5000)

# Environment E (training with top row goal positions)

train_goals <- list(c(4,1), c(4,2), c(4,3), c(4,4))
val_goals <- list(c(3,4), c(2,3), c(1,1))

set_weights(model,initial_weights)

show_validation(0)

for(i in 1:5000){
  if(i%%10==0) cat("episode", i,"\n")
  goal <- sample(train_goals, size = 1)
  reinforce_episode(unlist(goal))
}

show_validation(5000)
```