Advanced Machine Learning Lab 3

Mikael Montén

2024-10-22

Assignments

Q-Learning

The file RL Lab1.R in the course website contains a template of the Qlearning algorithm. You are asked to complete the implementation. We will work with a gridworld environment consisting of HxW 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: $S = \{(x, y)|x \in \{1, ..., H\}, y \in \{1, ..., W\}\}$
- 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 $(1 - \beta)$. 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.

Environment A

- Actions encoded

```
set.seed(12345)
# By Jose M. Peña and Joel Oskarsson.
# For teaching purposes.
# jose.m.pena@liu.se.

###########################################################################################################
# Q-learning
###########################################################################################################

library(ggplot2)
library(vctrs)

arrows <- c("^", ">", "v", "<")
action_deltas <- list(c(1,0), # up
                      c(0,1), # right
                      c(-1,0), # down
                      c(0,-1)) # left
```

- Function to visualize the environment and learned action values and policy. Run 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).

```
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 = 2.5,nudge_y = .35,na.rm = TRUE) +
          geom_text(aes(label = val2),size = 2.5,nudge_x = .35,na.rm = TRUE) +
          geom_text(aes(label = val3),size = 2.5,nudge_y = -.35,na.rm = TRUE) +
          geom_text(aes(label = val4),size = 2.5,nudge_x = -.35,na.rm = TRUE) +
          geom_text(aes(label = val5),size = 7.5) +
          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)))

}
```

- Implement the greedy and ϵ-greedy policies in the GreedyPolicy and EpsilonGreedyPolicy functions. The functions should break ties at random, i.e. they should sample uniformly from the set of actions with maximal Q-value.

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

```r
  # 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.
  maximum <- which(q_table[x,y,]==max(q_table[x,y,])) # find the maximum value for the next step
  # sample uniformly if several steps have same value
  action <- maximum[sample(seq_along(maximum),1)]

  return(action)
}
```

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

  # Your code here.
  threshold <- runif(1) # random probability

  greedyaction <- ifelse(threshold > 1-epsilon,GreedyPolicy(x,y),sample(1:4,1))

  # explore if random threshold is bigger than epsilon, otherwise exploit
  return(greedyaction)
}
```

- Transition model taking β as input.

```r
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)
}
```

- Implement the Q-learning in the function q-learning. 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.

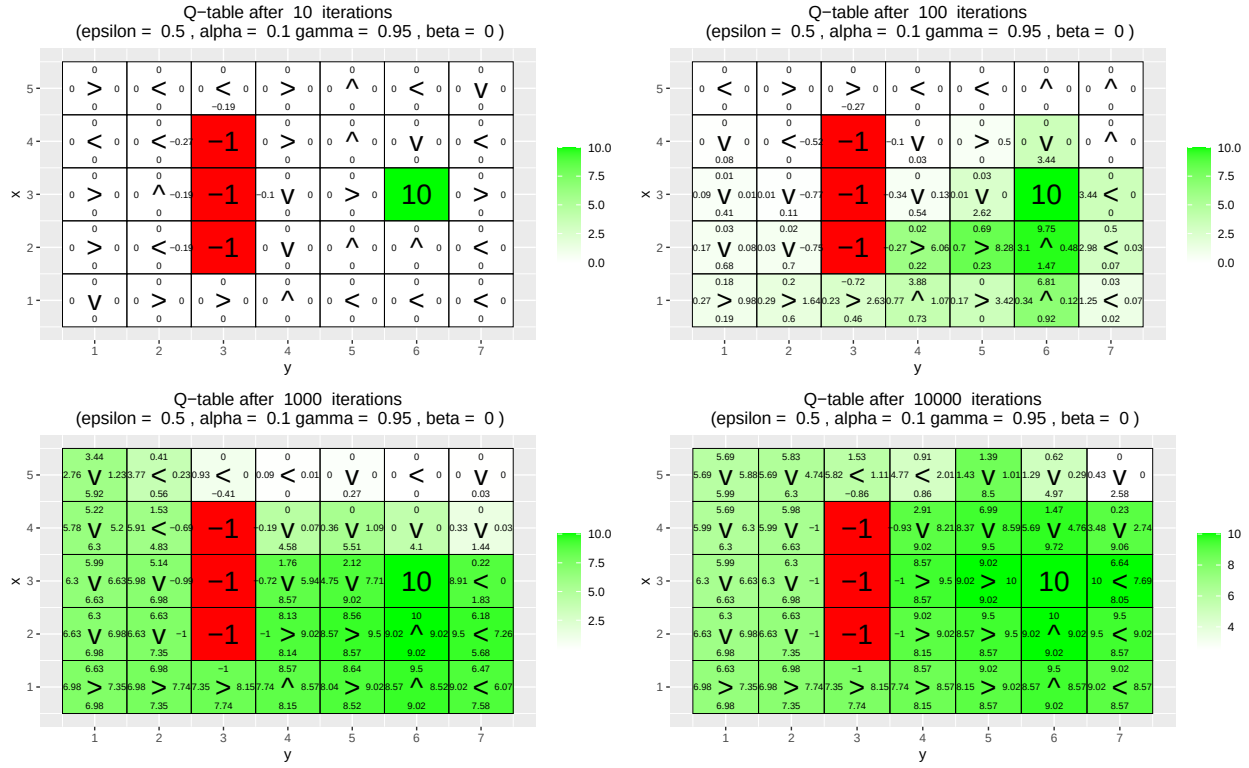
```
q_learning <- function(start_state, epsilon = 0.5, alpha = 0.1, gamma = 0.95,
                       beta = 0){

  # 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 randomly.
  #   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.

  # starting x and y coords
  x <- start_state[1]
  y <- start_state[2]

  episode_correction <- 0 # initialize correction

  repeat{
    # Follow policy, execute action, get reward.
    action <- EpsilonGreedyPolicy(x,y,epsilon) # follow policy

    transition <- transition_model(x,y,action,beta) # transition according to action returned

    # new x and y coords
    next_x <- transition[1]
    next_y <- transition[2]

    reward <- reward_map[next_x, next_y] # reward for move
```

```
    # Q-table update.

    # temporal-difference correction terms
    tempdiff <- reward+(gamma*max(q_table[next_x,next_y,]))-q_table[x,y,action]

    # sum temporal difference corrections
    episode_correction <- episode_correction + tempdiff

    q_table[x,y,action] <<- q_table[x,y,action] + alpha*tempdiff # update q-table


    # restart loop with next x and y as current values
    x <- next_x
    y <- next_y


    if(reward!=0)
      # End episode.
      return (c(reward,episode_correction))

  }

}
```

- 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). 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 x W environment with four actions, we can use a 3D-tensor of dimensions H x W x 4 to represent our Q-table. Initialize all Q-values to 0.
- Run 10000 episodes of Q-learning with $\epsilon = 0.5, \beta = 0, \alpha = 0.1, \gamma = 0.095$. The code visualizes the Q-table and a greedy policy derived from it after episodes 10, 100, 1000 and 10000.

```
##########################################################################################################
# Q-Learning Environments
##########################################################################################################

# Environment A (learning)

H <- 5
W <- 7

# create rewards in mentioned states
reward_map <- matrix(0, nrow = H, ncol = W)
reward_map[3,6] <- 10
reward_map[2:4,3] <- -1

# initialize q-table
q_table <- array(0,dim = c(H,W,4))

# shows initial environment
#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)
}
```



- What has the agent learned after the first 10 episodes?

The agent has started to discover the negative rewards. The Q-values for states near these rewards have begun to update, but the policy is still very random.

- Is the final greedy policy (after 10 000 episodes) optimal for all states, i.e. not only for the initial state? Why/not?

The final greedy policy seems to be optimal for many states but not all. The states that are quite close to negative/positive rewards or starting position are frequented often, which results in converged Q-values. The lower right corner has a correct pointing arrow but not a good score, probably due to the algorithm not having visited that corner often. The final policy is not optimal for all states after 10 000 episodes due to Q-learning being an exploration-explotation algorithm. The aim is to learn the optimal policy, but it might not visist every state-action pair sufficiently, especially those further away from the reward or starting state.

- Do the learned values in the Q-table reflect the fact that there are multiple paths (above and below the negative rewards) to get to the positive reward? If not, what could be done to make it happen?

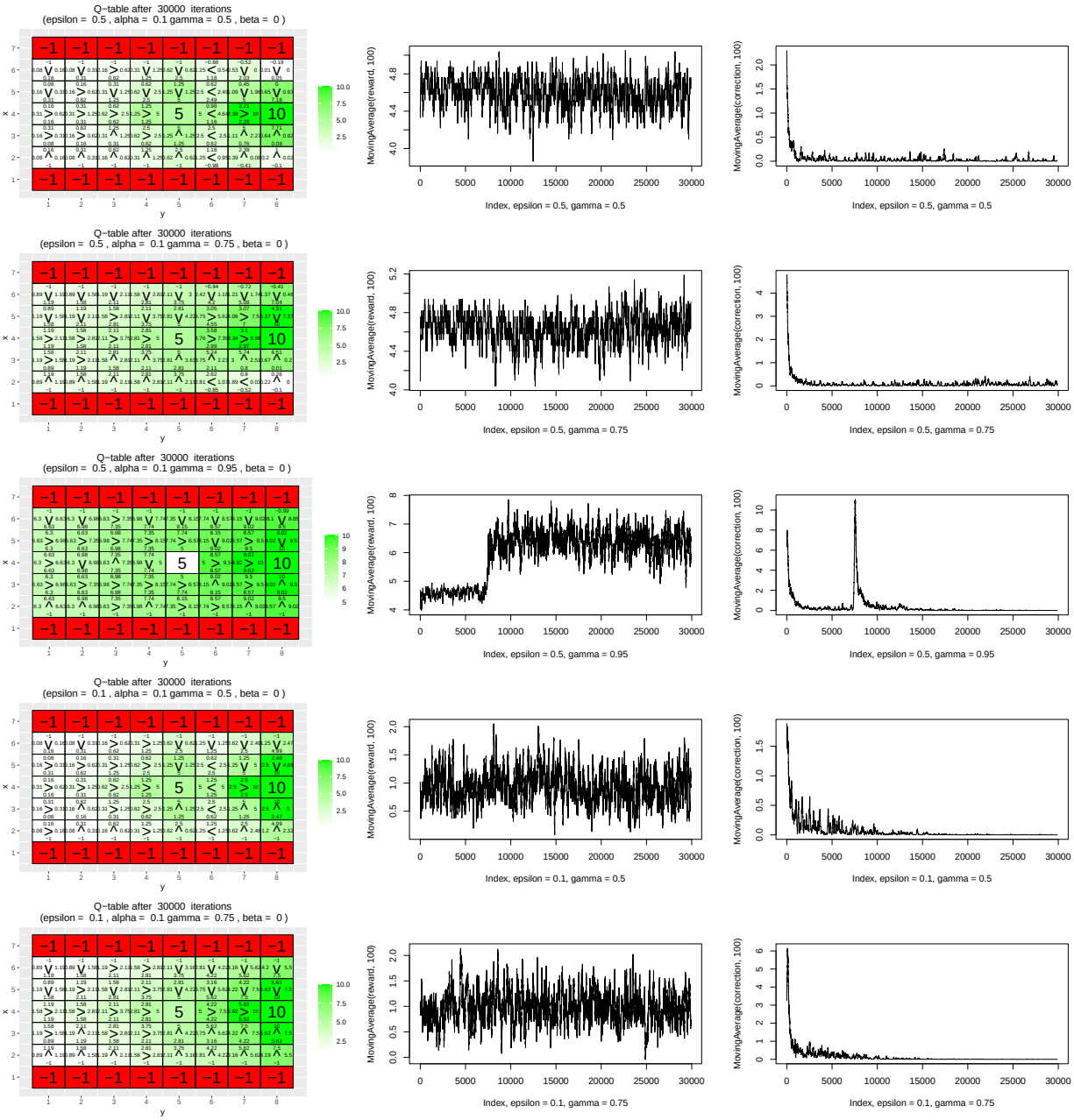
The learned Q-values does not reflect the existence of multiple paths to the positive reward, it avoids going underneath the negative rewards and instead only moves above it. This is because Q-learning updates Q-values based on experienced rewards and estimates of future rewards, and if the agent consistently finds one path more rewarding during exploration, the Q-values for less explored paths might not be as accuratelt updated. More exploration would be required to learn the path below the negative rewards as well, which could be solved by having a larger epsilon or running more episodes.

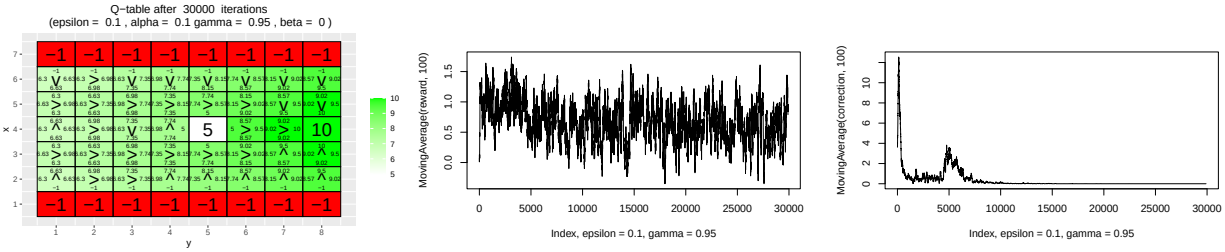
Environment B

This is a 7x8 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 $\epsilon = (0.1, 0.5), \gamma = (0.5, 0.75, 0.95), \beta = 0, \alpha = 0.1$.

```
# 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])
  }

  xaxis05 = paste0("Index, epsilon = 0.5, gamma = ",j)

  vis_environment(i, gamma = j)
  plot(MovingAverage(reward,100),type = "l",xlab=xaxis05)
  plot(MovingAverage(correction,100),type = "l",xlab=xaxis05)
}

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])
  }

  xaxis01 = paste0("Index, epsilon = 0.1, gamma = ",j)

  vis_environment(i, epsilon = 0.1, gamma = j)
  plot(MovingAverage(reward,100),type = "l",xlab=xaxis01)
  plot(MovingAverage(correction,100),type = "l",xlab=xaxis01)
}
```





- Explain your observations.

Exploration (ϵ) in the ϵ-greedy policy controls the balance between exploration and exploitation. A higher ϵ encourages more exploration, potentially leading to the discovery of better paths, even if they initially seem less rewarding.

Discount factory (γ) determines the importance of future rewards. A higher γ values future rewards more significantly, making the agent more likely to choose paths that lead to higher rewards in the long run.

For $\gamma = \{0.5, 0.75, 0.95\}$, as γ increases the agent becomes more "far-sighted". At 0.5 the agent prefers the closer reward of 5, as it discounts future rewards more heavily. At 0.75, there is more of a balance between immediate and future rewards, meaning it learns both paths. At 0.95, the agent learns the optimal path to the reward of 10 as it values future rewards more.

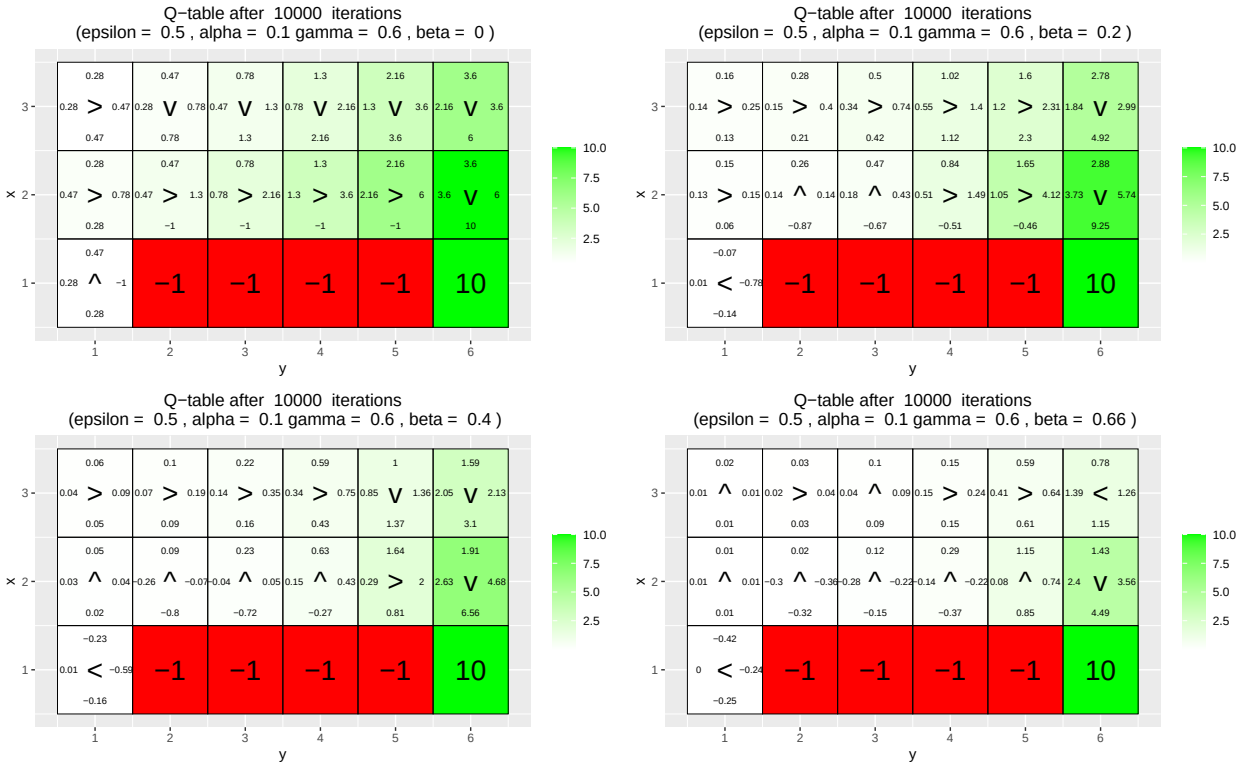
For $\epsilon = \{0.1, 0.5\}$, as ϵ increases the agent explores more. At 0.1, the agent exploits more, potentially getting stuck in paths to local optima (i.e. easier reward of 5). At 0.5, there is more exploration which increases the chances of finding the optimal path to the reward of 10.

The Q-table reflects this, i.e. a the highest *gamma* with the highest ϵ, results in an algorithm that finds the 10 reward often. This is because it explores 50% of actions taken and has a high enough discount factor for it to be able to consider in a substantial part of the environment before stopping.

Environment C

This is a smaller 3 x 6 environment. Here the agent starts each episode in the state (1,1). Investigate how the β parameter affects the learned policy by running 10000 episodes of Q-learning with $\beta = \{0, 0.2, 0.4, 0.66\}, \epsilon = 0.5, \gamma = 0.6, \alpha = 0.1$

```
# 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)
}
```



As β increases from 0 to 0.66, randomness is introduces into the environment which affects the learned policy and represents the probability of the agent's action not having the intended effect. A higher β makes the

environment less predictable, forcing the agent to learn a policy that accounts for possible slips. At 0, the policy is deterministic and optimal for all states to find the reward. At 0.2, the policy starts to account for an occasional slip, but remains very similar to 0. At 0.4, the policy is more cautious than before, and trying to avoid edges close to the negative rewards more. Here the policy has been noticably worse. At 0.66, the policy is very conservative and basically only prioritize staying away from negative rewards over reaching the positive rewards quickly.

REINFORCE

We will work with a 4x4 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.

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 lists train_goals and val_goals. The results provided in the file correspond to running the REINFORCE algorithm for 5000 episodes with $\beta = 0, \gamma = 0.95$. 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 nonterminal 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:

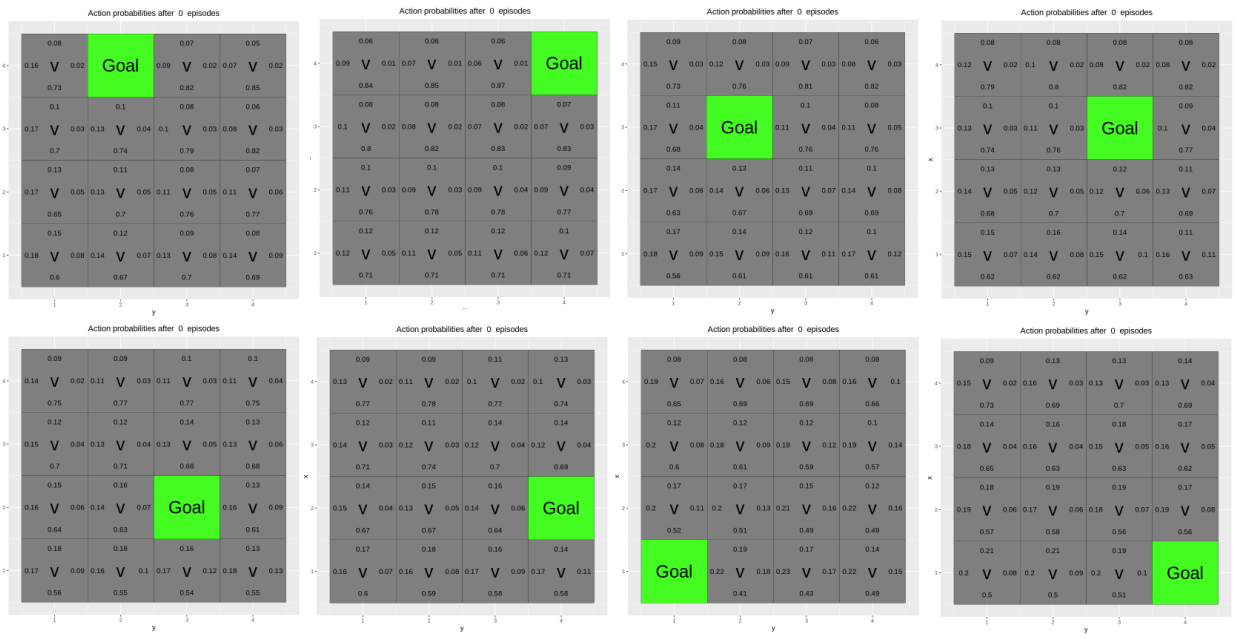
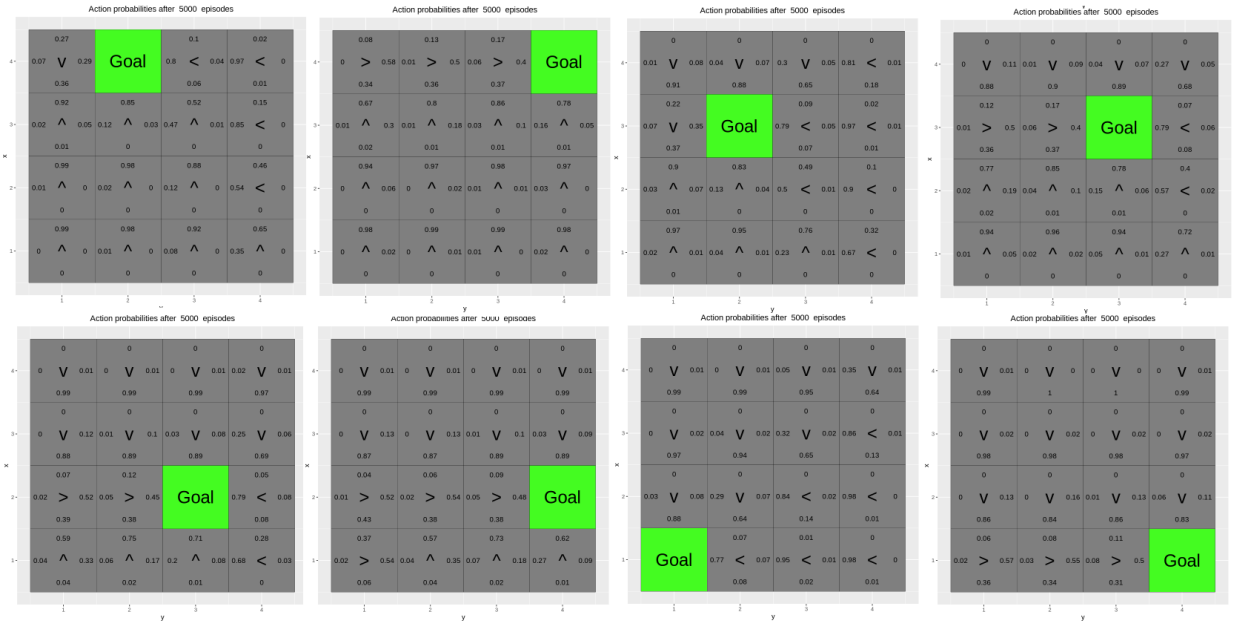


Figure 1: “0 episodes”


Action probabilities after 5000 episodes
Goal
x
y

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

REINFORCE directly learns the policy without explicitly estimating a value function. It updates the policy parameters based on the rewards received at the end of an episode, using gradient ascent to maximize expected reward. This provides REINFORCE with a strong advantage, which is its ability to generalise to unseen states. This is because the policy is a parameterized function that can interpolate between observed state-action pairs. This has resulted in finding optimal policy for almost all paths, albeit not all of them. For all goals situated in the bottom of the environment the algorithm has converged to the optimal path for all states. However, when the goals are situated at the top, it has found optimal paths for many but not all. Specifically, there are two environments where the arrow and maximum score points downward while it is next to the goal. This is likely due to the high gamma and the agent finds it way to the reward eventually, combined with there being some form of bias for going downwards from the environments after 0 episodes.

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

The resulting Q-table would be very large. We have 4x4 states for current x and y, as well as 4x4 states for reward x and reward y. Combine this with the 4 entries of up, down, left, right, results in a Q-table of size 4x4x4x4x4 = 1024 entries. Q-learning would need to learn separate policies for each goal position and cant generalize to unseen states, which means for goal positions not encountered during training the Q-values would be random. Deep Q-learning áddresses this limiation by using a neural network to approximate the Q-function, enabling generalisation and handling of large state spaces.

Environment E

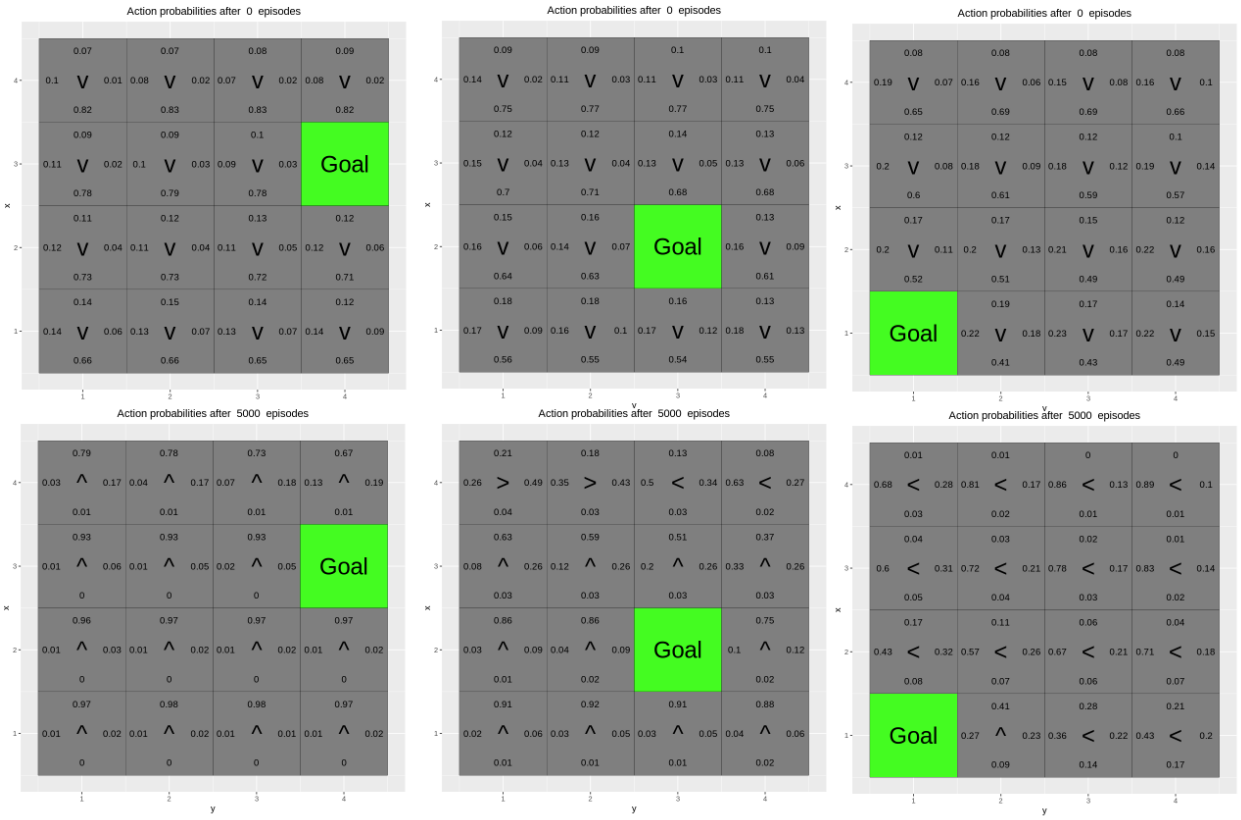


Figure 2: "0 and 5k episodes"

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

The agent has not a learned a good policy at all. This is due to the agent never having encountered the rewards at the position for which they are situated in the validation data. Training data only has rewards in a certain space of the environment thus reducing generalizability.

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

The agents in D & E differ due to the fact that D has training data where reward is situated in all states, which means the algorithm has learned paths to every coordinate and can generalize to arbitrary paths in the environment. This is not the case for E, where it only knows paths to a certain part of the environment.

Appendix

```
knitr::opts_chunk$set(echo = TRUE,fig.height = 4)
set.seed(12345)
# By Jose M. Peña and Joel Oskarsson.
# For teaching purposes.
# jose.m.pena@liu.se.

#########################################################################################################
# Q-learning
#########################################################################################################

library(ggplot2)
library(vctrs)

arrows <- c("^", ">", "v", "<")
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 = 2.5,nudge_y = .35,na.rm = TRUE) +
          geom_text(aes(label = val2),size = 2.5,nudge_x = .35,na.rm = TRUE) +
```

```
          geom_text(aes(label = val3),size = 2.5,nudge_y = -.35,na.rm = TRUE) +
          geom_text(aes(label = val4),size = 2.5,nudge_x = -.35,na.rm = TRUE) +
          geom_text(aes(label = val5),size = 7.5) +
          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.
  maximum <- which(q_table[x,y,]==max(q_table[x,y,])) # find the maximum value for the next step
  # sample uniformly if several steps have same value
  action <- maximum[sample(seq_along(maximum),1)]

  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 randomly.
  #
  # Returns:
  #   An action, i.e. integer in {1,2,3,4}.

  # Your code here.
  threshold <- runif(1) # random probability

  greedyaction <- ifelse(threshold > 1-epsilon,GreedyPolicy(x,y),sample(1:4,1))

  # explore if random threshold is bigger than epsilon, otherwise exploit
  return(greedyaction)
}
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 randomly.
  #   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.

  # starting x and y coords
  x <- start_state[1]
  y <- start_state[2]

  episode_correction <- 0 # initialize correction

  repeat{
    # Follow policy, execute action, get reward.
    action <- EpsilonGreedyPolicy(x,y,epsilon) # follow policy

    transition <- transition_model(x,y,action,beta) # transition according to action returned

    # new x and y coords
```

```
    next_x <- transition[1]
    next_y <- transition[2]

    reward <- reward_map[next_x, next_y] # reward for move

    # Q-table update.

    # temporal-difference correction terms
    tempdiff <- reward+(gamma*max(q_table[next_x,next_y,]))-q_table[x,y,action]

    # sum temporal difference corrections
    episode_correction <- episode_correction + tempdiff

    q_table[x,y,action] <<- q_table[x,y,action] + alpha*tempdiff # update q-table


    # restart loop with next x and y as current values
    x <- next_x
    y <- next_y


    if(reward!=0)
      # End episode.
      return (c(reward,episode_correction))

  }

}
################################################################################################################
# Q-Learning Environments
################################################################################################################

# Environment A (learning)

H <- 5
W <- 7

# create rewards in mentioned states
reward_map <- matrix(0, nrow = H, ncol = W)
reward_map[3,6] <- 10
reward_map[2:4,3] <- -1

# initialize q-table
q_table <- array(0,dim = c(H,W,4))

# shows initial environment
#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
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])
  }

  xaxis05 = paste0("Index, epsilon = 0.5, gamma = ",j)

  vis_environment(i, gamma = j)
  plot(MovingAverage(reward,100),type = "l",xlab=xaxis05)
  plot(MovingAverage(correction,100),type = "l",xlab=xaxis05)
}

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])
  }
```

```
  xaxis01 = paste0("Index, epsilon = 0.1, gamma = ",j)

  vis_environment(i, epsilon = 0.1, gamma = j)
  plot(MovingAverage(reward,100),type = "l",xlab=xaxis01)
  plot(MovingAverage(correction,100),type = "l",xlab=xaxis01)
}
# 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)
}
```