Lab3\_2

Andreas Hertin

2021-10-03

## REINFORCE

arrows <- c("^", ">", "v", "<")  
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.  
 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  
 }  
 }  
   
}

## REINFORCE Environments

## Environment D (training with random goal positions)

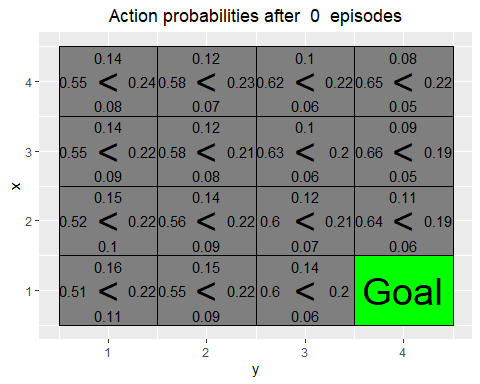
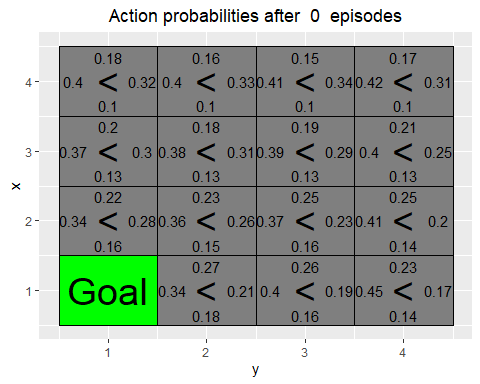
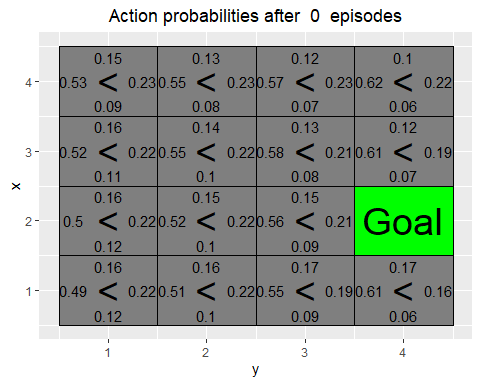
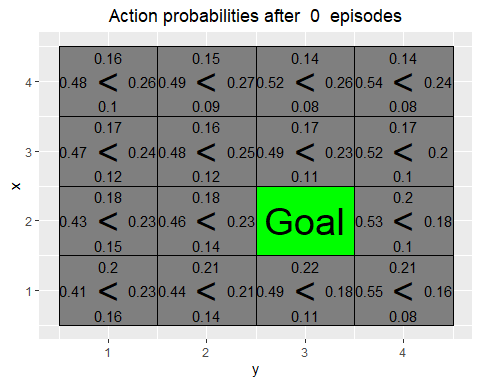
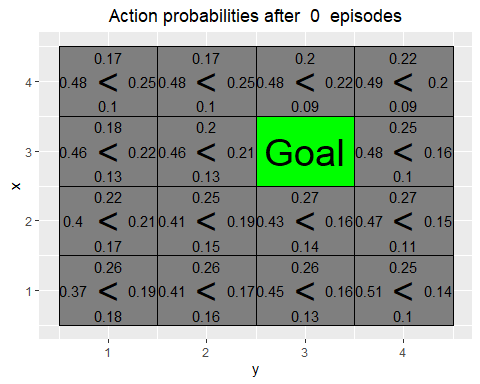
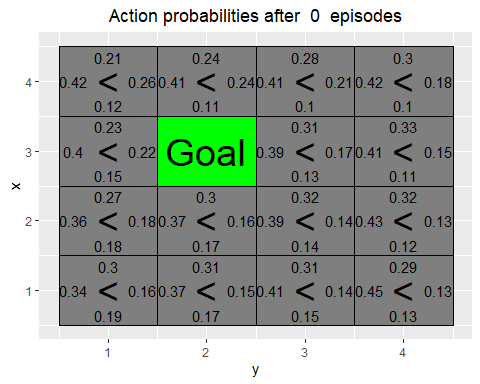
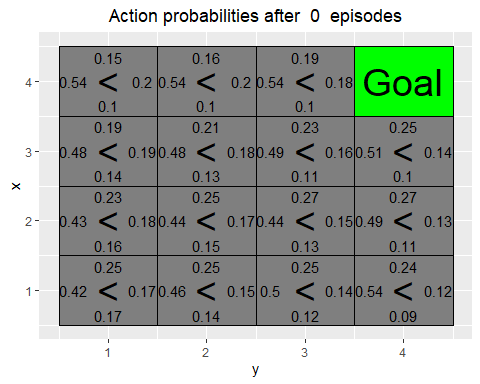
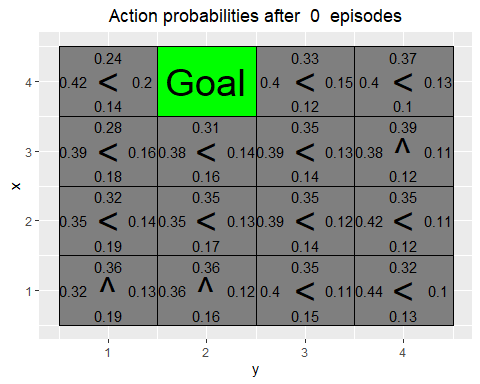
H <- 4  
W <- 4  
  
# Define the neural network (two hidden layers of 32 units each).  
model <- keras\_model\_sequential()

## Warning in normalizePath(path.expand(path), winslash, mustWork): path[1]="C:  
## \Users\Andreas\.conda\envs\.conda\_venv/python.exe": Det går inte att hitta filen  
  
## Warning in normalizePath(path.expand(path), winslash, mustWork): path[1]="C:  
## \Users\Andreas\.conda\envs\.conda\_venv/python.exe": Det går inte att hitta filen

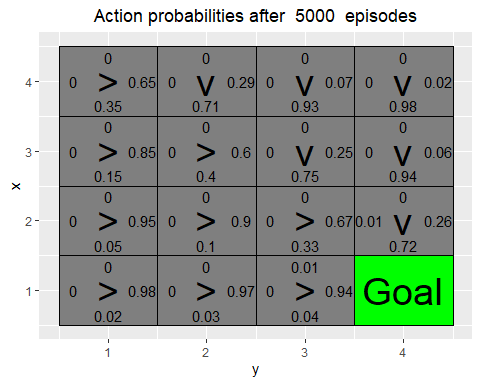
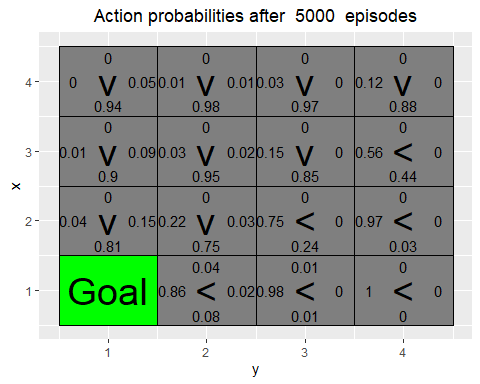
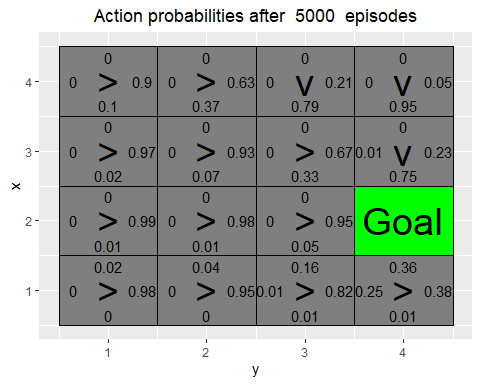
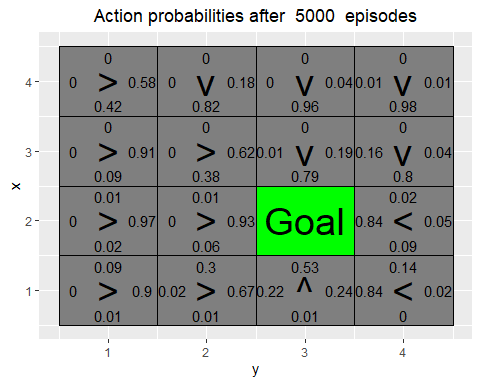
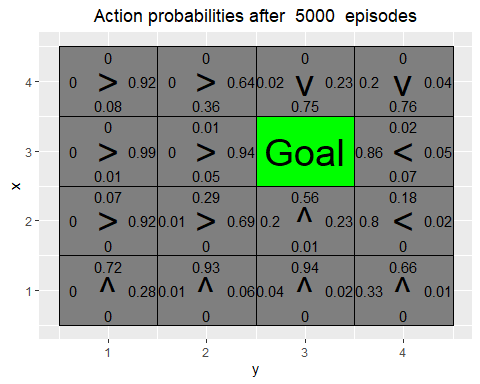
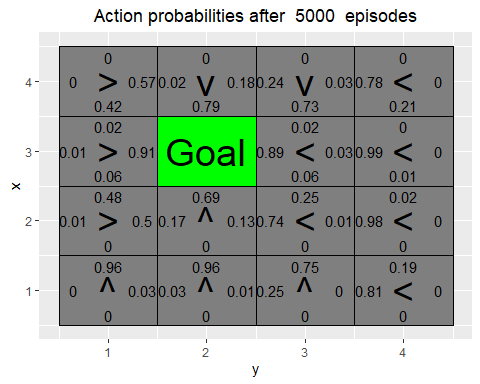
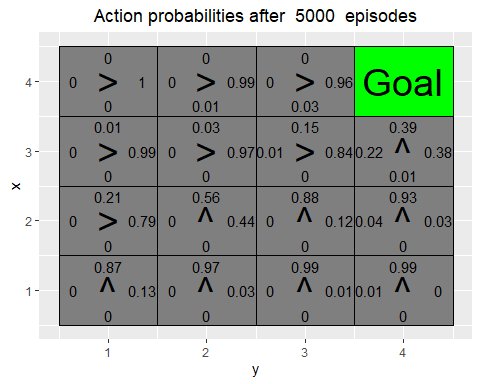
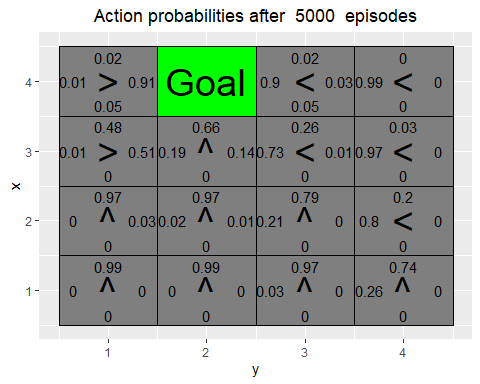
model %>%   
 layer\_dense(units = 32, input\_shape = c(4), activation = 'relu') %>%   
 layer\_dense(units = 32, activation = 'relu') %>%   
 layer\_dense(units = 4, activation = 'softmax')  
  
compile(model, loss = "categorical\_crossentropy", optimizer = optimizer\_sgd(lr=0.001))

## Warning in backcompat\_fix\_rename\_lr\_to\_learning\_rate(...): the `lr` argument has  
## been renamed to `learning\_rate`.

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)



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

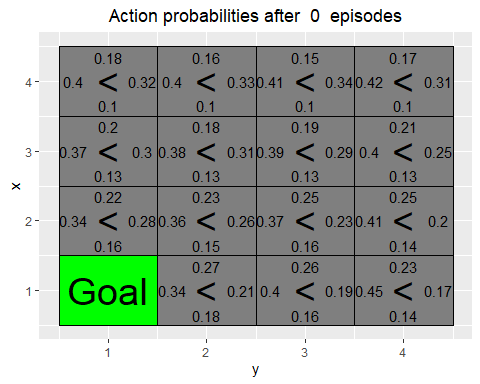
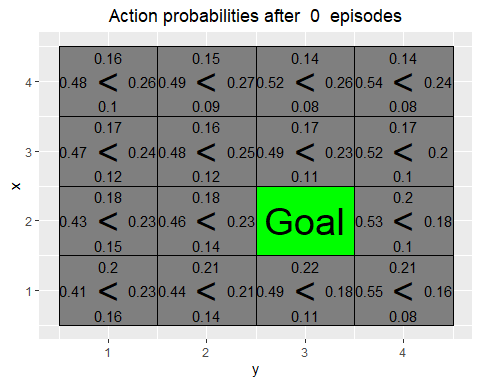
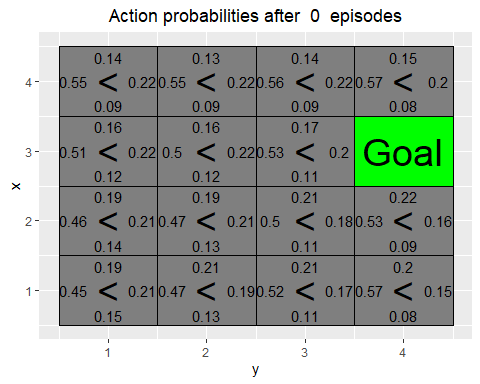
Yes, all states leads to the goal with their highest probabilities.

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

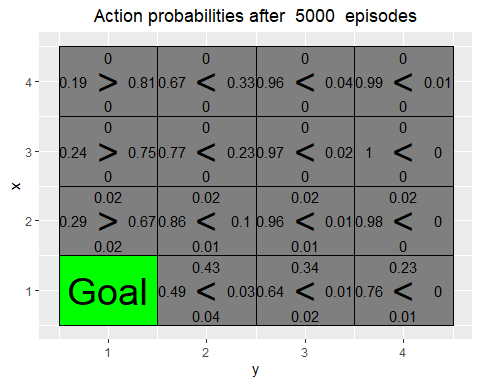
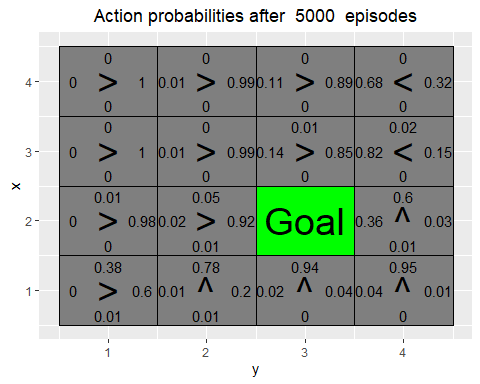
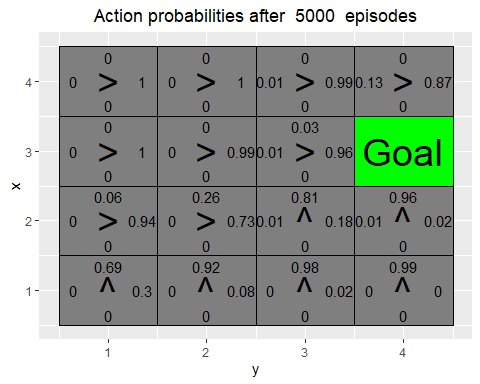
No, the goal moves which Q-learning can not handle and the algorithm has to visit a state to learn what Q-value it has. Not all states are set as a goal in the learning so it cannot learn to go to those.

## 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)



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

No, a lot of states does not lead to the goal. All training goals are in the top row so it is to specifically trained towards those.

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

In env. D the agent learns with spread out goals meaning it knows how to move in all directions. In env. E it does not learn how to move downwards leading to the validation goals only being reached when the state is on the same or rows beneath it.