Human-centered Assistive Robotics Technische Universität München Prof. Dr.-Ing. Dongheui Lee	**MACHINE LEARNING IN ROBOTICS** Assignment 2

Exercise 1: *Learning dataset using Gaussian mixture model*

Given $dataGMM.mat$: Write a Matlab code that loads the data set and trains a Gaussian Mixture model. The number of components used in GMM is equal to 4.

a) Initialize the GMM parameters with the k-means algorithm (from Matlab Toolbox or from the previous assignment).

b) Implement Expectation–Maximization estimation of GMM parameters.

Plot the density values for inputs arranged in a grid of 100x100 with values in the range [-0.1,0.1] for each variable (hint: you can use the Matlab function $surf$).

Exercise 2: *Human gesture recognition using hidden Markov model*

Some data collected with Microsoft Kinect sensor are given. The file $Test.txt$ contains the result of a segmentation of some sequences with the k-means clustering. $Test.txt$ is a 60×10 matrix $(10$ observation sequences, each of length $60)$ in which each element is an integer in the interval $1, ..., 8$.
The files $A.txt$, $B.txt$ and $pi.txt$ represent:

- the transition probability matrix,
- the observation probability matrix,
- the initial state probability vector,

of a discrete $left - to - right$ HMM. The number of discrete observations is $M = 8$, the number of states in the model is $N = 12$. So the matrix in $B.txt$ is a $N \times M$ matrix, where the sum of each row equals to 1.

- Using these data, classify each sequence in the file $Test.txt$ as follows:

$$\begin{cases} gesture1 & if\ log - likelihood > -115 \\ gesture2 & otherwise \end{cases}$$

NOTE: It is possible that all the sequences in $Test.txt$ belong to $gesture1$ or to $gesture2$.

Exercise 3: *Learning gait pattern for a humanoid robot using Reinforcement Learning*

Robot Description

The aim of this exercise is to control a mobile robot so that *it moves forward*. For simplification, let's assume a discrete state-space in which each leg can be in one of the four positions: up&forward, up&back, down&forward and down&back. Since the two legs can be positioned independently, the system can be in any of the 16 states as shown in Figure 1. The control system can only choose one of the four actions from Table 1. Whether the leg moves forward or backward for action 2 or 4 is determined by its current state. If it is forward then it will move backward and if it is backward then it will move forward. Up-down is handled correspondingly. The system is deterministic, which means that with probability equal to 1, the state transitions according to the commanded action.

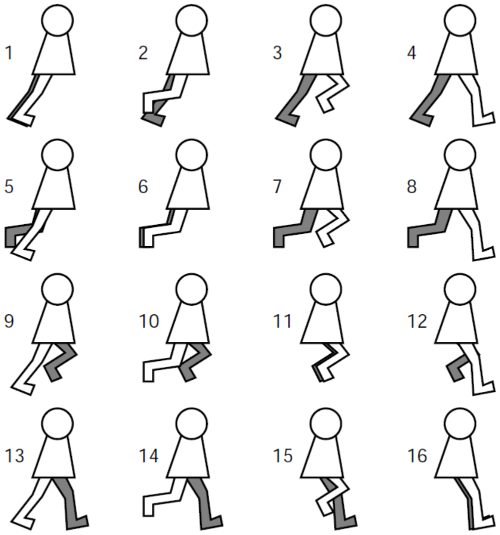

Figure 1: The humanoid-robot can be in 16 different states.

Action	Effect
1	Move right leg up or down
2	Move right leg back or forward
3	Move left leg up or down
4	Move left leg back or forward

Table 1: Actions and their effect on humanoid robot.

Tasks:

Task 1: Defining reward function

Your first task is to define a state-action reward function $r(s, a)$. Make reasonable guesses of what actions to reward and what to penalize in order to get a reasonable walking behavior. Only give reward to actions that actually move the robot forward and not all the intermediate actions necessary such as moving the leg forward when lifted. The idea is that the learning algorithm should do the actual planning of how to move the legs. Therefore, try *not to guide it by rewarding all steps throughout the whole step cycle*.

The reward matrix rew should be 16×4 matrix where each row correspond to a state and each column corresponds to an action. A useful way of defining reward matrix is to start with zeros throughout the reward matrix. Now enter positive values for the state-action pair that move the robot forward. Also enter negative values for the state-action pair that should be avoided i.e. moving the robot backward or raising one leg while one is already in the air. Try to keep the reward matrix as simple as possible while still achieving the desired behaviour.

Task 2: Applying policy iteration

After defining the reward function, apply $Policy\ Iteration$ for learning the gait sequence. Since all states and actions are discrete we can define $\pi(s)$ (policy) and $V(s)$ (value function) as vectors. While $s' = \delta(s, a)$ (state transition function) and $r(s, a)$ (state-action reward) will be both matrices.

The state transition matrix is defined like this:

$$\delta(s,a) = \begin{bmatrix} 2 & 4 & 5 & 13 \\ 1 & 3 & 6 & 14 \\ 4 & 2 & 7 & 15 \\ 3 & 1 & 8 & 16 \\ 6 & 8 & 1 & 9 \\ 5 & 7 & 2 & 10 \\ 8 & 6 & 3 & 11 \\ 7 & 5 & 4 & 12 \\ 10 & 12 & 13 & 5 \\ 9 & 11 & 14 & 6 \\ 12 & 10 & 15 & 7 \\ 11 & 9 & 16 & 8 \\ 14 & 16 & 9 & 1 \\ 13 & 15 & 10 & 2 \\ 16 & 14 & 11 & 3 \\ 15 & 13 & 12 & 4 \end{bmatrix}$$

where each row corresponds to a state and each column corresponds to an action. In the beginning we don't have any knowledge about the policy ($\pi(s)$) so it can be initialized randomly.

```
policy=ceil(rand(16,1)*4);
```

Now use policy iteration to learn a policy that moves the robot forward.

```
Policy iteration for deterministic system
Initialize $\pi$ randomly.
Repeat until convergence
{
(a) Let $V := V^\pi$
(b) For each state s, let $\pi(s) := \arg\max_{a \in A}(r(s,a) + \gamma V(s'))$
}
```

For step (a) you can easily calculate the value of each state for fixed policy by writing the Bellman equation for our deterministic system.

$$V^\pi(s) = r(s,a) + \gamma V^\pi(\delta(s, \pi(s))) \tag{1}$$

With this you will get 16 equations with 16 unknowns. Now the linear system of equations can be easily solved to get the value of each state. You can also use the iterative policy evaluation algorithm to approximate the values $V^\pi(s)$. After this, step (b) greedily updates the policy using the current value function.

In order to test the resulting policy, make a short simulation by starting in an arbitrary state and successively making actions according to the policy. Verify that the behavior is reasonable, i.e. that it looks like a good walking pattern. Use the provided matlab function *walkshow.m* (see walkshow.m) to verify your results, where the function walkshow(state list) takes the sequence of states as input and displays a graphical "cartoon" of the walking robot. This makes it easier to visualize that whether you are getting a desirable behavior or not. A sample output of learned policy when starting from state 8 can be visualized in Figure 2.

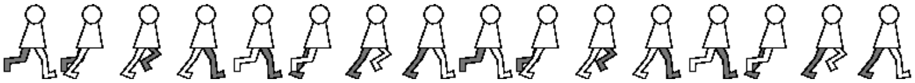

Figure 2: Learned policy with policy iteration.

Now write a Matlab function

$WalkPolicyIteration(s)$

that takes the state s as input and then produces a result like as in Figure 2. All the learning should be performed in this function. You are not allowed to use any matlab function/toolbox which solves Policy Iteration.

After completing this task answer the following questions:

1. Report your reward matrix.
2. What value of γ have you used and what is the result of increasing or decreasing γ?
3. Approximately how many iterations are required for the policy iteration to converge?
4. Attach the result of *WalkPolicyIteration(s)* when starting from state 10 and 3.

Task 3: Applying Q-learning

Policy iteration is only useful for the known model of the environment i.e. when $r(s,a)$ and $\delta(s,a)$ are known. In many practical problems, these are not known and have to be estimated from the experience. $Temporal\ Difference$ (TD) methods improve the estimate at each time step. Now you will again learn the policy for making the robot to move forward but now with Q-$learning$. Similar to $V^\pi(s)$, the action value function $Q^\pi(s,a)$ is defined as the expected return of taking action a in state s and thereafter following policy π.

```
Initialize $Q(s,a)$ arbitrarily $\forall\ s,a$
Initialize $s$:
Repeat:
   Choose a from s using $\epsilon$-greedy policy based on $Q(s,a)$
   Take action $a$ acording to $\epsilon$-greedy policy and observe $r$ and $s'$
   Update $Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right]$
   $s \leftarrow s'$
Until T steps.
```

Now write a Matlab function

[newstate reward]=SimulateRobot(state,action)

which uses the transition and reward matrices and returns next state (s') and reward (r) for given state and action.

Since we don't know the values of Q-function in advance, it can be initialized randomly or filled with zeros.

Q=zeros(16,4);

Now you will use ϵ-greedy policy for learning. This means most of the time with probability $(1-\epsilon)$ the robot will act greedily by picking the optimal action according to:

$\pi(s) = \arg\max_a Q(s,a)$

but with small probability ϵ it takes a random action (exploration step). As the agent collects more and more evidence, the policy can be shifted towards a deterministic greedy policy. A sample output of learned policy when starting from state 16 should look like Figure 3.

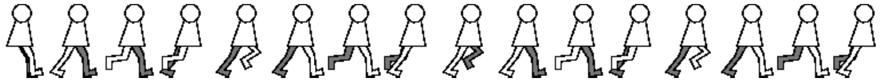

Figure 3: Learned policy with Q-learning.

Now write a Matlab function

WalkQLearning(s)

that takes a state s as input. Starting in an arbitrary state, the algorithm should follow its current policy (ϵ-greedy) to generate actions which are sent to the *SimulateRobot.m* to move around in the state space. For each move. the Q-values should be updated appropriately. After learning the robot should shift to greedy policy and then it should produce a result like as in Figure 3. All the learning should be performed in this function. You are not allowed to use any matlab function/toolbox which solves Q-learning.

After completing this task answer the following questions:

1. Report the values of ϵ and α that you have used.
2. What happens if a pure greedy policy is used? Implement and compare with the ϵ-greedy policy. Does it matter what value of ϵ you use?
3. Approximately how many steps are necessary for the Q-learning algorithm to find an optimal policy?
4. Attach the result of *WalkQLearning(s)* when starting from states 5 and 12.