Group Spirits
First Lab Report CS362*

Submitted to Prof Pratik Shah

1st Devang Patel
202051062
202051062@iiitvadodara.ac.in

2nd Anik Bepari
202051024
202051024@iiitvadodara.ac.in

3rd Manoj kumar gautam
202051114
202051114@iiitvadodara.ac.in

4th Aditya tomar
202051012
202051012@iiitvadodara.ac.in

I. Introduction

1. lab assignment 8
2. lab assignment 9
3. lab assignment 10
4. lab assignment 6

These are the topics that we have made our reports on. We have performed our experiment to understand our results better on google collab which gave us the required output and is available on github accordingly.// This report is about theoritically understanding the problem and the process (like algorithm,diagram ,pseudocode etc) which will give us an idea about the topic ,and then we can approach to solve this via code(python etc) accordingly.

II. Lab Assignment 8

A. Objective

Basics of data structure needed for state-space search tasks and use of random numbers required for MDP and RL

B. Problem Statement

Read the reference on MENACE by Michie and check for its implementations. Pick the one that you like the most and go through the code carefully. Highlight the parts that you feel are crucial. If possible, try to code the MENACE in any programming language of your liking.

C. Explanation

MENACE (Matchbox Educable Noughts And Crosses Engine) is a machine learning algorithm developed by Donald Michie in the early 1960s to play the game of tic-tac-toe (also known as noughts and crosses). It uses the principles of reinforcement learning to improve its performance over time.

The MENACE algorithm consists of a set of matchboxes, each corresponding to a different board configuration in tic-tac-toe. Inside each matchbox, there are a number of colored beads corresponding to possible moves that can be played from that board configuration. At the start of the game, the matchboxes are filled with a random distribution of beads.

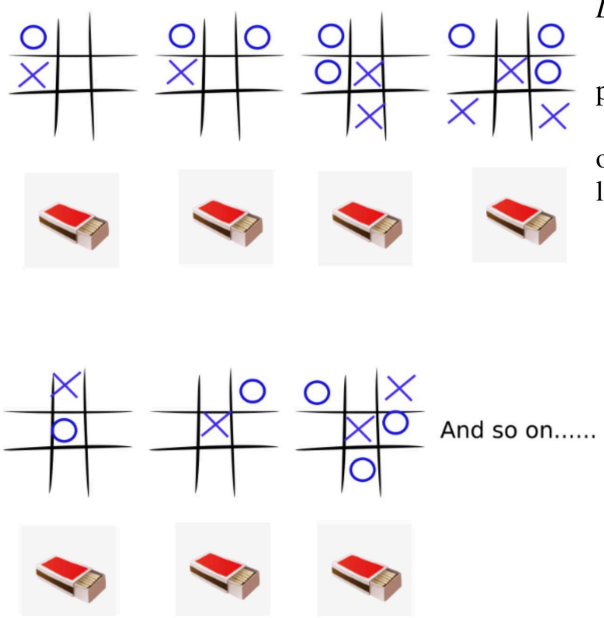
During the game, MENACE selects a matchbox based on the current board configuration and randomly selects a bead from that matchbox to determine its move. After each game, MENACE updates the bead distribution in the matchbox it used, increasing the frequency of beads that resulted in wins and decreasing the frequency of

beads that resulted in losses. This process is similar to the concept of reinforcement learning, where the algorithm learns by receiving feedback in the form of rewards or punishments.

Over time, as MENACE plays more games, the bead distribution in the matchboxes changes, with more successful moves becoming more likely to be selected in future games. This leads to improved performance, and eventually MENACE becomes a proficient player of tic-tac-toe.

MENACE is an early example of machine learning and has inspired further research and development of reinforcement learning algorithms for games and other applications.

Algorithm



Assuming you are referring to the MINACE (Minimum-Norm Adaptive Constant Modulus Algorithm), which is a signal processing algorithm used for blind equalization and channel estimation, here are the basic steps of the algorithm:

1) Initialize the weight vector to some initial values. Feed the received signal through a linear filter with the weight vector to get an estimate of the transmitted signal.

2) Compute the error between the estimated and actual transmitted signals.

3) Update the weight vector using the error and the received signal, such that it minimizes the mean square error and maintains a constant modulus constraint.

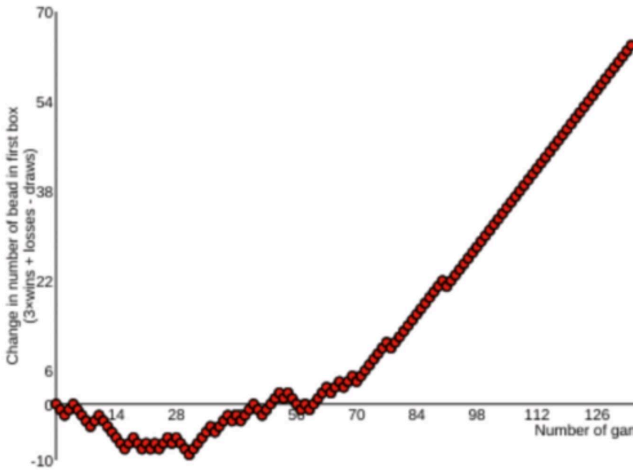
4) Repeat steps 2-4 until the weight vector converges to some final value.

1) Symmetric Layouts of Tic Tac Toe: The tic-tac-toe game board is symmetric, so many of the states are actually similar states if rotated or taken mirror image of the board. So we remove all the duplicate states that we generated. After this we are left with only around 900 states out of 255,168 states.

D. RESULTS

When MENACE plays a perfect-playing computer, the results look like this:

When MENACE played with a random picking opponent, the result is a near-perfect positive correlation:



E. GITHUB LINK

Click here to visit the Lab8 GitHub repository.

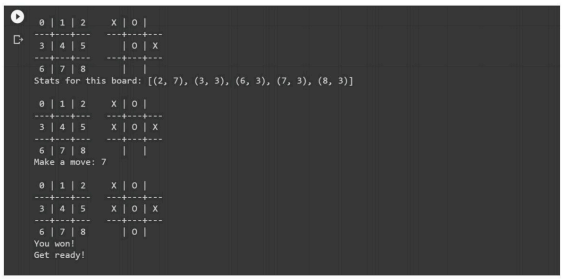


Fig. 1. Output of Code

F. References

1) Matchbox Educable Naughts and Crosses Engine:

III. Lab Assignment 9

A. OBJECTIVES

Understanding Exploitation - Exploration in simple n-arm bandit reinforcement learning task, epsilon-greedy algorithm.

B. Problem Statement

1) Consider a binary bandit with two rewards 1-success, 0-failure. The bandit returns 1 or 0 for the action that you select, i.e. 1 or 2. The rewards are stochastic (but stationary). Use an epsilon-greedy algorithm discussed in class and decide upon the action to take for maximizing the expected reward. There are two binary bandits named binaryBanditA.m and binaryBanditB.m are waiting for you.
2) Develop a 10-armed bandit in which all ten mean-rewards start out equal and then take independent random walks (by adding a normally distributed increment with mean zero and standard deviation 0.01 to all mean-rewards on each time step).
{function [value] = bandit_nonstat(action)}
3) The 10-armed bandit that you developed (bandit_nonstat) is difficult to crack with a standard epsilon-greedy algorithm since the rewards are non-stationary. We did discuss how to track non-stationary rewards in class. Write a modified epsilon-greedy agent and show whether it is able to latch onto correct actions or not. (Try at least 10000 time steps before commenting on results)

C. Explanation

1) Problem 1: The epsilon greedy function takes as input the epsilon value (the probability of choosing a random action), two instances of BinaryBandit representing the two bandits, and the number of pulls to make. It initializes the estimated values and counts for each bandit to zero, then loops over the specified number of pulls. On each iteration, it selects an action using epsilon-greedy selection, pulls the corresponding bandit arm, and updates the estimated value and count for the selected action.

2) Problem 2: This is the case of 10 arm-bandit where the mean-rewards are initially taken as a constant valued array initialised by 1. We performed exploration and exploitation based on the Epsilon Greedy Algorithm. A random number between 0 and 1 is generated and if it comes out to be greater than epsilon, we perform exploitation, which is based on the prior knowledge otherwise exploration. For every iteration an array of ten values is generated such that they are normally distributed with mean value zero and standard deviation of 0.01. This array is added to the mean array and this updated array is used every time. For every action a reward is given from this updated mean-rewards array.The rewards given in this case are non-stationary.

3) Problem 3: This is same as problem 2 except the calculation of action rewards. In this case, instead of using averaging method to update estimation of action reward, we are assigning more weights to the current reward earned by using alpha parameter having value 0.7

D. RESULTS

1) First Problem: For bandit1 we kept the probability of rewards low(0.3 and 0.4) and for bandit2 we kept if higher(0,7 and 0.8).

We can see that the epsilon greedy methods works same for both of them as we kept epsilon same.
Since bandit2 had higher probabilites of getting reward, average reward of him is also higher than bandit 1.

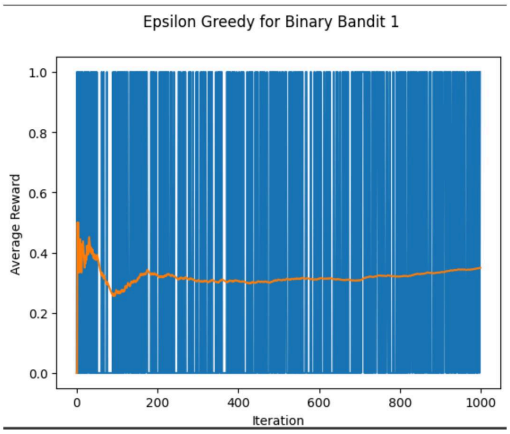


Fig. 2. Epsilon greedy for binary bandit 1

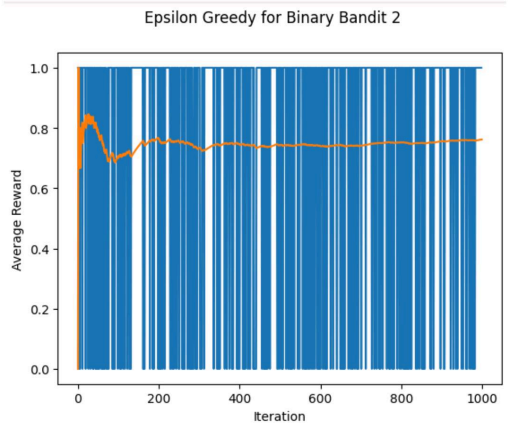


Fig. 3. Epsilon greedy for binary bandit 2

2) Second Problem: We observed that as initially all the rewards were equal, we get a constant reward of say like 1 but as the steps increases , it affects the rewarding policy of every action and thus we see that the expected rewards also start increasing. We also observe that the increase decreases over steps.

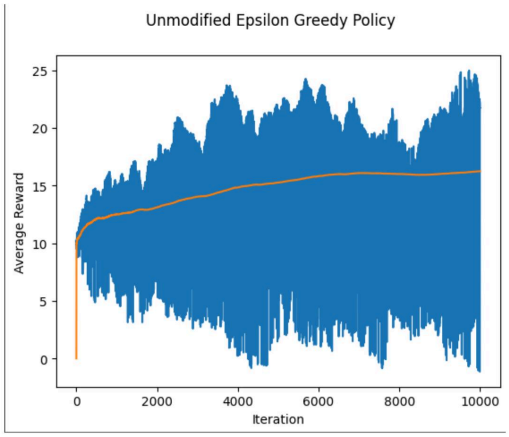


Fig. 4. Unmodified epsilon greedy policy

3) Third Problem: Here we can see that as compared to problem 2, in which the expected rewards were low , we saw that when instead of averaging the profit percentage of state when we gave higher weightage to the to the current value at every step (a normally distributed array is added to the rewards array), the expected rewards we get were much higher as in the case 2 and the slope of the graph was too steeper when compared with case 2.

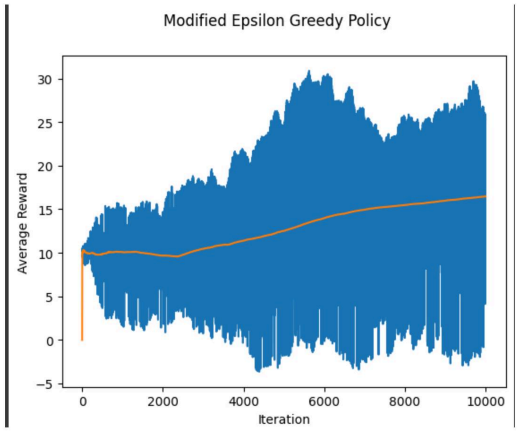


Fig. 5. Modified epsilon greedy policy

E. GITHUB LINK

Click here to visit the Lab9 GitHub repository.

F. References

Reinforcement Learning: an introduction by R Sutton and A Barto (Second Edition) (Chapter 1-2)

IV.
Lab Assignment 10

A. OBJECTIVES

Understand the process of sequential decision making (stochastic environment) and the connection with reinforcement learning

B. Problem Statement

1) Suppose that an agent is situated in the 4x3 environment as shown in Figure 1. Beginning in the start state, it must choose an action at each time step. The interaction with the environment terminates when the agent reaches one of the goal states, marked +1 or -1. We assume that the environment is fully observable, so that the agent always knows where it is. You may decide to take the following four actions in every state: Up, Down, Left and Right. However, the environment is stochastic, that means the action that you take may not lead you to the desired state. Each action achieves the intended effect with probability 0.8, but the rest of the time, the action moves the agent at right angles to the intended direction with equal probabilities. Furthermore, if the agent bumps into a wall, it stays in the same square. The immediate reward for moving to any state (s) except for the terminal states S+ is r(s)= -0.04. And the reward for moving to terminal states is +1 and -1 respectively. Find the value function corresponding to the optimal policy using value iteration.
2) You are managing two locations for Gbike. Each day, some number of customers arrive at each location to rent bicycles. If you have a bike available, you rent it out and earn INR 10 from Gbike. If you are out of bikes at that location, then the business is lost. Bikes become available for renting the day after they are returned. To help ensure that bicycles are available where they are needed, you can move them between the two locations overnight, at a cost of INR 2 per bike moved. Assumptions: Assume that the number of bikes requested and returned at each location are Poisson random variables. Expected numbers of rental requests are 3 and 4 and returns are 3 and 2 at the first and second locations respectively. No more than 20 bikes can be parked at either of the locations. You may move a maximum of 5 bikes from one location to the other in one night. Consider the discount rate to be 0.9. Formulate the continuing finite MDP, where time steps are days, the state is the number of bikes at each location at the end of the day, and the actions are the net number of bikes moved between the two locations overnight.
3) Write a program for policy iteration and resolve the Gbike bicycle rental problem with the following changes. One of your employees at the first location rides a bus home each night and lives near the second location. She is happy to shuttle one bike to the second location for free. Each additional bike still costs INR 2, as do all bikes moved in the other direction. In addition, you have limited parking space at each location. If more than 10 bikes are kept overnight at a location (after any moving of cars), then an additional cost of INR 4 must be incurred to use a second parking lot (independent of how many cars are kept there).

C. Explanation

An MDP consists of a set of states, a set of actions, and a set of probabilities that describe the outcomes of actions taken in each state. The state of the system is assumed to evolve according to a Markov process, which means that the future state depends only on the current state and the action taken, and not on the past history of the system.

1) Algorithm:

$$U(s) = \max_{a \in A(s)} \sum_{s'} P(s' \mid s,a)[R(s,a,s') + \gamma U(s')]$$

1) Bellman Equation: Here, U(s) represents the value of state s, a is an action, s' represents the next state, P(s'|s,a) is the probability of transitioning to state s' from state s when action a is taken, R(s,a,s') is the reward received from transitioning from state s to state s' when action a is taken, and is a discount factor that represents the importance of future rewards.
2) Policy Iteration: Policy Iteration is an algorithm in 'Reinforcement Learning', which helps in learning the optimal policy which maximizes the long term discounted reward. These techniques are often useful, when there are multiple options to chose from, and each option has its own rewards and risks.
3) Poisson distribution: A Poisson Distribution is a statistical distribution used to express the probability of a given number of events occurring within a fixed interval of time or space. Additionally, the events must occur independently of each other and with a known constant rate.
4) We can then use policy iteration to determine the optimal policy.

D. Code

function VALUE-ITERATION(mdp, ϵ) **returns** a utility function
  inputs: mdp, an MDP with states S, actions $A(s)$, transition model $P(s' \mid s,a)$,
    rewards $R(s,a,s')$, discount γ
    ϵ, the maximum error allowed in the utility of any state
  local variables: U, U', vectors of utilities for states in S, initially zero
    δ, the maximum relative change in the utility of any state

  repeat
    $U \leftarrow U'$; $\delta \leftarrow 0$
    for each state s **in** S **do**
      $U'[s] \leftarrow \max_{a \in A(s)}$ Q-VALUE(mdp, s, a, U)
      if $|U'[s] - U[s]| > \delta$ **then** $\delta \leftarrow |U'[s] - U[s]|$
  until $\delta \leq \epsilon(1-\gamma)/\gamma$
  return U

$$Q(s,a) = \sum_{s'} P(s' \mid s,a)[R(s,a,s') + \gamma U(s')]$$

```
sequential_decision_environment = GridMDP([[0.1, 0.1, 0.1, +1],
                                           [0.1, None, 0.1, -1],
                                           [0.1, 0.1, 0.1, 0.1]],
                                          terminals=[(3, 2), (3, 1)])
value_iteration(sequential_decision_environment, epsilon=0.001)
```

```
{(0, 1): 0.9999899925828089,
 (1, 2): 0.999999331397728,
 (2, 1): 0.9999577777706711,
 (0, 0): 0.9999875459240295,
 (3, 1): -1.0,
 (2, 0): 0.9999787337320681,
 (3, 0): 0.9994791585041776,
 (0, 2): 0.9999917011356055,
 (2, 2): 0.9999944696344899,
 (1, 0): 0.999984982429827,
 (3, 2): 1.0}
```

E. GITHUB LINK

Click here to visit the lab10 GitHub repository.

F. References

1) Matchbox Educable Naughts and Crosses Engine:

Lab Assignment 7

I. OBJECTIVES

To model the low level image processing tasks in the framework of Markov Random Field and Conditional Random Field. To understand the working of Hopfield network and use it for solving some interesting combinatorial problems.

II. PROBLEM STATEMENT

1) Many low level vision and image processing problems are posed as minimization of energy function defined over a rectangular grid of pixels. We have seen one such problem, image segmentation, in class. The objective of image denoising is to recover an original image from a given noisy image, sometimes with missing pixels also. MRF models denoising as a probabilistic inference task. Since we are conditioning the original pixel intensities with respect to the observed noisy pixel intensities, it usually is referred to as a conditional Markov random field. Refer to (3) above. It describes the energy function based on data and prior (smoothness). Use quadratic potentials for both singleton and pairwise potentials. Assume that there are no missing pixels. Cameraman is a standard test image for benchmarking denoising algorithms. Add varying amounts of Gaussian noise to the image for testing the MRF based denoising approach. Since the energy function is quadratic, it is possible to find the minima by simple gradient descent. If the image size is small (100x100) you may use any iterative method for solving the system of linear equations that you arrive at by equating the gradient to zero. Extra Credit Challenge: Implement and compare MRF denoising with Stochastic denoising (reference 2).
2) For the sample code hopfield.m supplied in the lab-work folder, find out the amount of error (in bits) tolerable for each of the stored patterns.
3) Solve a TSP (traveling salesman problem) of 10 cities with a Hopfield network. How many weights do you need for the network?

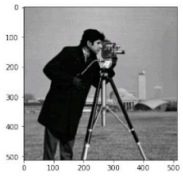
III. EXPLANATION

A. Problem 1

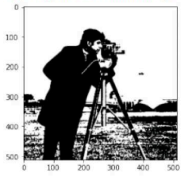
This is a 512x512 grayscale image. We normalize the pixel values to be between 0 and 1, by dividing all values by 255, and then 'binarizing' it for the Markov Random Field by converting all the normalized pixel values below 0.5 to 0 and the rest to 1 Markov random fields sue a quadratic potential function to measure the energy potential of the image when changing a particular pixel, with respect to the neighbouring pixels. The quadratic potential function is given by:

$$E(U) = \Sigma_{n=1}^{N}(u_n - v_n)^2 + \lambda\Sigma_{n=1}^{N-1}(u_{n+1} - u_n)^2 \quad (1)$$

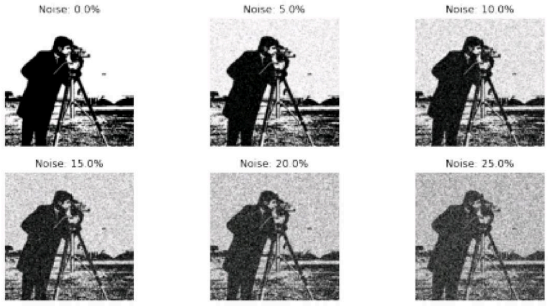
where v is the smooth ID signal, u is the IID and E is the energy function. We ran the algorithm for 5*512*512 = 1310720 iterations.

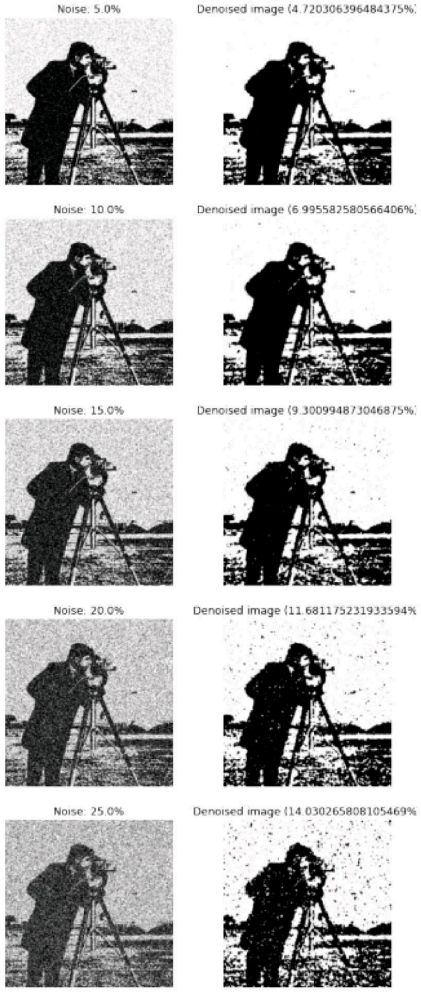


(a) Original Cameraman Image



(b) 'Binarized' Cameraman Image

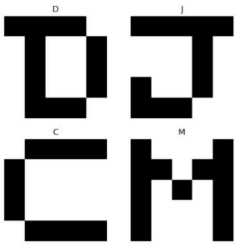




B. Problem 2

For this task, we converted the hopfield.m MATLAB codes to Python so that we could do it in the same Notebook as the other parts. The network was trained using Hebb's rule, as in the file and then they were noised by changing some random pixel values. At max, 15 pixel values were noised. We can see that suprisingly, the network can correct up to max 8 errors.

C. Problem 3



This is the usual famous NP-hard problem of Travelling Salesman, that is done using the Hopfield Networks. Since in a Hopfield Network, each node is connected to the other node, we needed a total of 10x10 = 100 weights. We first generate 10 cities randomly. And then let the hopfield network predict an optimal least path cost.