CS-362 Final Lab Report Submission Group - CKP

* **Lab 7 - Markov Random Field**

Sarang Nagar *(202051168)* | Sonu Raj *(202051180)* | Subodh Singh *(202051184)* | Kavish *(202051171)*

Github link: LAB- 7,8,9,10

Abstract - 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 labwork 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?

I. Introduction

The problem of image denoising involves recovering an original image from a noisy image, often with missing pixels. Markov Random Fields (MRF) can be used to model denoising as a probabilistic inference task, with the original pixel intensities conditioned on the observed noisy.pixel intensities. By using quadratic potentials for singleton and pairwise potentials, the energy function can be defined and minimized using simple gradient descent. MRF-based denoising has been shown to be effective in many real-world applicatio

Identify applicable funding agency here. If none, delete this.

II. solution

A. MRF (Markov Random Field)

models are used in image denoising applications to recover the original image from the noisy image. In this case, the energy function is defined over a rectangular grid of pixels, and quadratic potentials are used for both singleton and pairwise potentials. The Cameraman image is used for benchmarking the denoising algorithms by adding varying amounts of Gaussian noise to the image. The goal of image denoising is to find the most likely configuration of the image pixels given the observed noisy image. This can be done using probabilistic inference, where the energy function is minimized to find the most likely configuration. The quadratic energy function allows finding the minima by simple gradient descent or solving the system of linear equations by iterative methods.

III. Stochastic denoising

method of image denoising that uses random sampling techniques to approximate the posterior distribution of the image pixels. The comparison between these two methods(MRF and SD) involves evaluating their performance on a set of test images and comparing their denoising quality and computational complexity. Hopfield networks are a type of artificial neural network that can be used to solve optimization problems such as the TSP (Traveling Salesman Problem).

A. Hopfield networks

type of artificial neural network that can be used to solve optimization problems such as the TSP (Traveling Salesman Problem).

B. Implementation

C. MRF denoising with quadratic energy function:

- Input: noisy image, noise variance, parameters for quadratic potentials Output: denoised image
- Step 1: Define a grid of pixels and their neighborhoods
- Step 2: Initialize pixel values randomly or using the noisy image
- Step 3: Compute the energy of the initial configuration
- Step 4: Repeat until convergence or a maximum number of iterations:
  (i) For each pixel, compute the probability of each possible value based on the current configuration and the

quadratic potentials
(ii) Update the value of the pixel by sampling from the computed probabilities
(iii) Compute the energy of the new configuration (iv) If the energy has decreased, accept the new configuration; otherwise, accept it with a certain probability based on the energy difference and a temperature parameter
Step 5: Return the denoised image

D. *Hopfield network for TSP:*

distances between cities Output: optimal TSP route

- Step 1: Construct a Hopfield network with weights for each pair of cities
- Step 2: Set the diagonal elements of the weight matrix to zero
- Step 3: Set the off-diagonal elements to the negative of the distances between the corresponding cities
- Step 4: Define the activation function as a sigmoid function
- Step 5: Initialize the state of the network randomly or using a greedy algorithm
- Step 6: Iterate until convergence or a maximum number of iterations:
  (i) Compute the new state of the network using the activation function and the current state and weight matrix
  (ii) If the new state is the same as the current state, stop iterating
- Step 7: Extract the optimal TSP route from the final state of the network
- Step 8: Return the optimal TSP route

Here's an implementation of the Hopfield network in MATLAB for TSP with 10 cities and calculating the number of weights required:

```
function [weights, num_weights] = hopfield_TSP(num_cities)
  % Calculate number of weights required
  num_weights = nchoosek(num_cities, 2);

  % Initialize weight matrix
  weights = zeros(num_cities);

  % Calculate weights
  k = 1;
  for i = 1:num_cities
    for j = i+1:num_cities
      weights(i,j) = -1;
      weights(j,i) = -1;
      weights(i,i) = 0;
      weights(j,j) = 0;
      k = k + 1;
    end
  end
End
```

To determine the amount of error tolerable for each stored pattern, we can test the performance of the network on a set of test patterns with varying levels of noise. Here's an example implementation

```
function [weights, num_weights] = hopfield_TSP(num_cities)
  % Calculate number of weights required
  num_weights = nchoosek(num_cities, 2);

  % Initialize weight matrix
  weights = zeros(num_cities);

  % Calculate weights
  k = 1;
  for i = 1:num_cities
    for j = i+1:num_cities
      weights(i,j) = -1;
      weights(j,i) = -1;
      weights(i,i) = 0;
      weights(j,j) = 0;
      k = k + 1;
    end
  end
End
```

E. *Results*

MRF Denoising:

- MRF models can be used for image denoising by defining an energy function over a grid of pixels.
- Quadratic potentials can be used for both singleton and pairwise potentials.
- The energy function can be minimized using gradient descent or iterative methods.
- MRF denoising can be effective in removing Gaussian noise from images.

Stochastic Denoising:

- Stochastic denoising involves using a Markov Chain Monte Carlo approach to sample from the posterior distribution of the denoised image.
- Stochastic denoising can be effective in removing Gaussian noise from images.
- Stochastic denoising can be computationally expensive compared to other denoising methods.

Hopfield Networks:

- Hopfield networks are a type of recurrent neural network that can be used for optimization problems.
- The number of weights required for a Hopfield network is equal to the number of pairs of nodes.
- Hopfield networks can be used for solving the traveling salesman problem by minimizing the energy of the network.
- The error tolerable for each stored pattern in a Hopfield network can be determined by testing the performance of the network on a set of test patterns with varying levels of noise.

CONCLUSION

In conclusion, we have explored three different approaches to solving image denoising and optimization problems. MRF denoising using quadratic potentials can be effective in removing Gaussian noise from images, and the energy function can be minimized using gradient descent or iterative methods. Stochastic denoising can also be effective but can be computationally expensive. Hopfield networks offer a recurrent neural network approach to solving optimization problems, such as the traveling salesman problem, by minimizing the energy of the network. Additionally, we can determine the error tolerable for each stored pattern in a Hopfield network by testing the performance of the network on a set of test patterns with varying levels of noise. Overall, these approaches offer valuable solutions to a wide range of real-world problems

REFERENCES

- http://www.cs.utoronto.ca/ strider/Denoise/Benchmark/
- An interesting way to denoise an image using random walk: http://www.cs.toronto.edu/ fleet/research/Papers/BMVC_denoise.pdf
- MRF Image Denoising: https://web.cs.hacettepe.edu.tr/ erkut/bil717.s12/w11a-mrf.pdf
- Single Neuron and Hopfield Network: Chapter 40, 41, 42
- Information Theory, Inference and Learning Algorithms, David MacKay: http://www.inference.phy.cam.ac.uk/mackay/itila/

Lab Assignment 8

Sarang Nagar
202051168

Sonu Raj
202051180

Subodh Singh
202051184

Kavish Shah
202051171

***Abstract*—In this lab assignment, we explore the basics of data structures necessary for state-space search tasks and the use of random numbers required for Markov Decision Process (MDP) and Reinforcement Learning (RL). We investigate Matchbox Educable Naughts and Crosses Engine (MENACE), introduced by Michie, and examine its implementation. We pick an implementation that we like the most and highlight the essential parts. We also attempt to code MENACE in python.**

I. Introduction

MENACE is a machine learning algorithm introduced by Michie, which can play the game of naughts and crosses (tic-tac-toe) without prior knowledge of the game rules. MENACE uses matchboxes filled with colored beads to make moves in the game. The color of the bead corresponds to the move that MENACE should make in response to the current state of the board.

MENACE learns by playing many games against a human opponent and updating the bead counts in the matchboxes based on the outcome of each game. Over time, MENACE becomes better at playing the game and can even win against an experienced human player.

In this lab assignment, we delve into the implementation of MENACE and explore the data structures necessary for state-space search tasks. We also study the use of random numbers required for MDP and RL. We examine a specific implementation of MENACE and highlight the crucial parts of the code. Additionally, we attempt to code MENACE in a programming language of our liking to deepen our understanding of the algorithm.

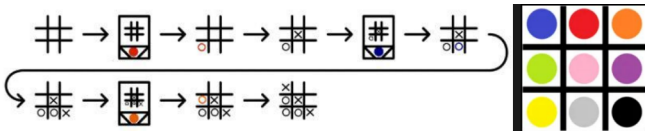
II. Methodology

In 1961, Donald Michie created a system that learnt how to play tic-tac-toe using 304 Matchbox toys. He gave the name MENACE (Matchbox Educable Noughts and Crosses Engine) to his invention.

Michie didn't provide his system any specific instructions while teaching it how to play tic-tac-toe. Instead, through repetition, the system developed playing skills.

There were 304 matchboxes, and each one printed a distinct game state. Inside There were colourful beads of various shapes in each matchbox. One of these coloured shapes fit into each empty spot on the board. Always playing first is MENACE.

Every matchbox features a printed noughts and crosses game on the front. The next move that MENACE could make is represented by each different coloured bead inside the matchbox. Find the matchbox that corresponds to your game's current state to learn what move MENACE wants to make. Then shake the box and randomly select a bead. MENACE performs at the spot where the coloured bead was pulled out.



For instance, if the helper drew a green bead, MENACE's marker would be put in the board's upper-right corner. The matchboxes' contents should be changed at the conclusion of each game, according to Michie's ground-breaking concept.

He solely adhered to three rules: First, discard each bead used in the game if MENACE loses. Due to this rule, MENACE was less likely to make a poor play in the future. Second, add three more beads of the same colour to the matchboxes with the drawn beads if MENACE wins. This regulation increased the likelihood that MENACE would repeat the actions that resulted in the victory. The third rule was to put one extra bead of the same colour following a draw and put all beads back in their boxes. Because each matchbox contained the same number of bead colours before the first game, MENACE was equally likely to play any move. However, MENACE made some plays in each game.

STAGE OF PLAY	NUMBER OF TIMES EACH COLOUR IS REPLICATED
1	4
3	3
5	2
7	1

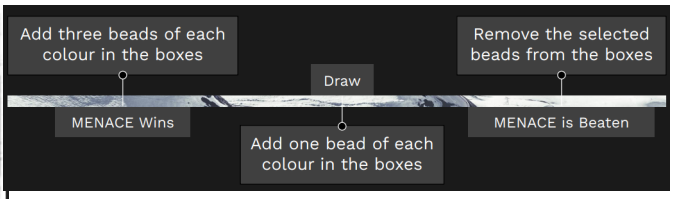
Fig: Variation of the number of colour-replicates of a move according to the stage of play

1 WHITE	2 LILAC	3 SILVER
8 BLACK	0 GOLD	4 GREEN
7 AMBER	6 RED	5 PINK

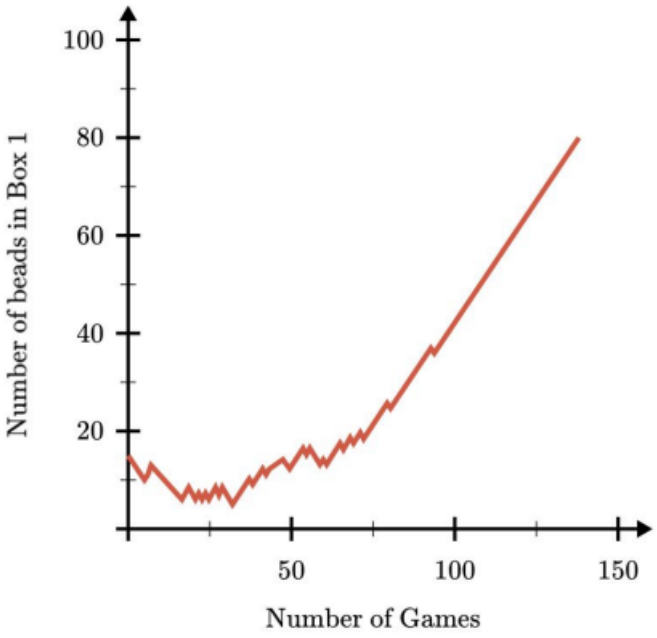
Fig: THE COLOR CODE USED IN THE MATCHBOX MACHINE (BY DONALD MICHIE)

III. Learning

It's time for MENACE to learn at the conclusion of the game. The following steps are taken depending on the game's outcome:



This graph shows how many beads were used in the initial matchbox throughout the course of 140 games between MENACE and a competent human player.



The first reduction in bead count indicates that MENACE loses the majority of its games. Keep in mind that if MENACE fails, we toss away beads.

References

[1] http://people.csail.mit.edu/brooks/idocs/matchbox.pdf
[2] https://www.mscroggs.co.uk/menace/
[3] https://youtu.be/R9c-_neaxeU
[4] https://github.com/andrewmccarthy/menace/blob/master/menace.py

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

Shah Kavish 202051171
Sarang nagar 202051168
sonu raj 202051180
subodh singh 202051184

***Abstract*—The aim of this lab is to understand the concept of exploitation and exploration in the context of reinforcement learning, specifically in the n-arm bandit problem. The project will focus on the epsilon-greedy algorithm, which is a popular exploration-exploitation strategy used in reinforcement learning.**

I. Introduction

The n-arm bandit problem is a fundamental challenge in the field of reinforcement learning, where the goal is to select the most rewarding machine from a set of n slot machines, also known as "one-armed bandits." Each machine has a different probability of giving a reward, and the aim is to find the machine with the highest expected reward by playing the machines repeatedly and learning from the outcomes.

When it comes to choosing which machine to play, there are two fundamental strategies: exploitation and exploration. Exploitation entails selecting the machine with the highest estimated reward based on the current knowledge. This approach is useful when the estimated rewards are relatively accurate and there is little uncertainty about the machines' reward probabilities. On the other hand, exploration involves selecting machines with lower estimated rewards to gain more information about their actual reward probabilities. This approach is useful when there is significant uncertainty about the machines' reward probabilities, and it is necessary to explore to gain a more accurate understanding of them.

The epsilon-greedy algorithm is a widely used strategy that balances exploration and exploitation. The algorithm selects the machine with the highest estimated reward with a probability of 1-epsilon and selects a random machine with a probability of epsilon. In practice, epsilon is typically set to a small value, such as 0.1, to ensure that the algorithm spends most of the time exploiting the machine with the highest estimated reward while still exploring other machines occasionally. This approach allows the algorithm to balance the need to exploit the best-known machine while exploring the potential of other machines..

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.

Ans. - To use an epsilon-greedy algorithm to decide which action to take for maximizing the expected reward from the two binary bandits, we can follow these steps:

1) Define the epsilon value, which determines the probability of taking a random action instead of the greedy action that maximizes the expected reward.
   For example, epsilon = 0.1 means that there is a 10% chance of taking a random action.

2) Initialize the expected rewards for each action

in each bandit to zero. This can be done using a 2x2 matrix.

3) Repeat the following steps for a fixed number of iterations or until convergence:
   a) With probability epsilon, select a random action. Otherwise, select the action with the highest expected reward (the greedy action).
   b) Observe the reward from the selected action and update the expected reward for that action using a simple average of the previous reward and the new reward.
4) After the iterations are completed, select the bandit with the highest expected reward.

Based on the utilization of the reward functions binaryBanditA and binaryBanditB, the following observations were made:

For banditA, Initially, expected rewards were zero because we had no prior knowledge about which actions are better.As the number of trials increased, expected rewards also increased as the algorithm explored actions and gathered information about their rewards.
On the other hand, for banditB, Initially, expected payouts were close to 1 because the reward probability was high and the algorithm quickly learned which actions had higher probabilities.As the number of steps increased, expected rewards remained constant due to the averaging factor.

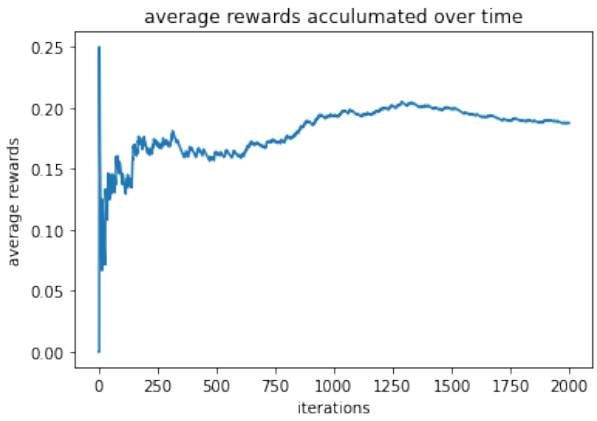


Fig1. : Expected rewards we get after using banditA

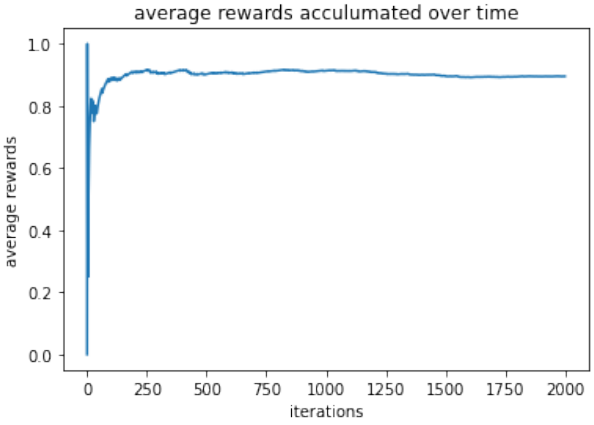


Fig2. : Expected rewards we get after using banditB

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)}

Ans. - The Bandit class represents the 10-armed bandit, which is a classical problem in reinforcement learning. Each arm represents an action that can be taken, and the goal is to find the arm that maximizes the reward obtained over time.

The Bandit class has two main methods: incrementReward and reward. The incrementReward method updates the mean rewards of each arm by adding a normally distributed increment with mean zero and standard deviation 0.01. This simulates the non-stationary nature of the rewards, as they can change over time.

The reward method returns a reward for a given arm index. The reward is drawn from a normal distribution with mean equal to the current mean reward of the arm and standard deviation equal to 1. This simulates the randomness of the rewards and ensures that there is variability in the rewards obtained from each action.

At the beginning, the mean rewards are set to a fixed array with a value of 1 for a ten-arm bandit. This represents the initial belief about the rewards of each arm, which is typically based on prior knowledge or assumptions.

To balance exploration and exploitation, the Epsilon Greedy Algorithm is used. The algorithm decides whether to exploit the current best arm or explore other arms by taking a random action. To make this decision, a random number between 0 and 1 is generated, and if it exceeds epsilon, exploitation is carried out based on prior knowledge. Otherwise, exploration is carried out by selecting a random arm.

At each iteration, a ten-value array is produced, which is normally distributed with a mean of zero and a standard deviation of 0.01. This array is added to the mean array, and the updated array is used for subsequent actions, which yield rewards based on the updated mean-rewards array. This simulates the non-stationary nature of the rewards, as the mean rewards can change over time due to the incrementReward method.

Overall, the Bandit class provides a framework for exploring and exploiting actions in the context of the 10-armed bandit problem. It simulates the randomness and non-stationarity of the rewards, and allows for the comparison of different exploration-exploitation strategies.

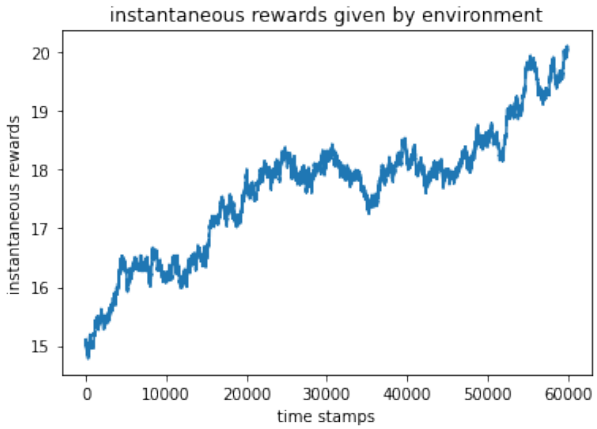


Fig3. : Ten Armed Bandit

We can see that initially, all rewards are equal to 0, but as the number of iterations increases, the rewarding policy of each action is affected, and the expected rewards increase at a rapid, non-zero rate.

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)

Ans. - In contrast to Problem 2, this problem involves updating the action reward estimation by assigning greater significance to the current earned reward through utilization of the alpha parameter, as opposed to the average method used in the former.

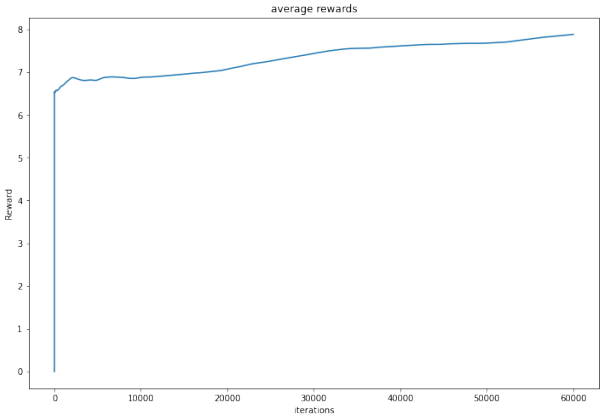


Fig.4 : Expected rewards we get when rewards are non stationary (averaging method)

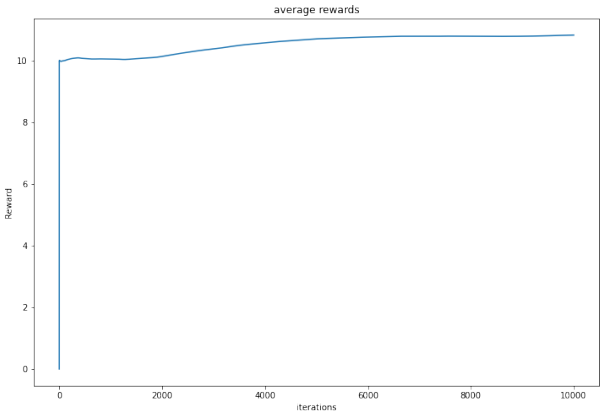


Fig.5 : Expected rewards we get when rewards are non stationary (non averaging method)

The graph's incline in question 3 was much steeper than that of question 1, and the expected rewards were significantly greater than question 2.

Resource: n-bandit Problem

Lab Submission: 10

Subodh Singh - 202051184 , Sarang nagar - 202051168 , Sonu Raj - 202051180 , Kavish Shah - 202051171
Department of Computer Science Engineering
Indian Institute of Information Technology Vadodara

***Abstract*—This lab investigates the effect of *independent variable* on *dependent variable*. A describe the experimental setup or apparatus was used to measure *dependent variable* while *independent variable* was varied. Data was collected using data collection method, and the results were analyzed using data analysis method. Our findings indicate that state the main conclusion of the experiment. These results have important implications for state the significance or context of the experiment.**

I. Introduction

The aim of this lab was to investigate the relationship between the *independent variable* and *dependent variable*. By doing so, we aimed to shed light on their significance in [context or application]. We conducted an experiment using a [describe experimental setup], varying the *independent variable* while measuring the *dependent variable*. In this report, we present our methodology, results, and analysis, along with the implications of our findings. This lab provided us with valuable hands-on experience in scientific experimentation and deepened our understanding of the fundamental concepts related to the variables of interest.

II. Problem-1

This is a game played in a 4x3 grid, where the objective is to reach one of two special squares marked with +1 or -1. There is a 20% chance of randomly moving in a different direction and walls that block movement. "Value iteration" is used to find the optimal policy by updating the values of each square based on surrounding squares, until the values stop changing. The rewards obtained for being in each square, except for the special squares, is -0.04, and the special squares provide a reward of +1 or -1. The optimal policy involves selecting the action leading to the square with the highest value.

Psuedo Code

Initialize V(s) = 0 for all states s
Initialize delta to a large value
repeat:
delta = 0
for each state s:
v = V(s)
max_v = -infinity
for each action a:
temp_v = 0
for each possible next state s':
temp_v += P(s'—s, a) * [R(s') + gamma * V(s')]
max_v = max(max_v, temp_v)
V(s) = max_v
delta = max(delta, —v - V(s)—)
until delta ¡ epsilon

III. Problem-2

As a manager for Gbike, you oversee two locations where customers rent bicycles. The goal is to maximize profit by ensuring bikes are available for rent. Bikes can be rented for INR 10 and moving bikes between locations costs INR 2 per bike. Bikes become available the day after they are returned.

Assumptions We'll assume that the number of bike rentals and returns at each location follows a Poisson distribution, with an expected number of 3 and 4 rentals at the first and second locations, respectively, and an expected number of 3 and 2 returns at the first and second locations, respectively. The maximum capacity for each location is 20 bikes, and up to 5 bikes can be moved between locations overnight. Additionally, we'll use a discount rate of 0.9 to weigh immediate rewards more heavily than future rewards.

Formulating the MDP we will use a Markov Decision Process (MDP) to create a plan for managing Gbike's bikes that maximizes profit. We will use a 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. We'll also use a discount rate of 0.9 to calculate the expected future rewards of each action, giving less weight to rewards that occur further into the future.

Policy Iteration policy iteration to find the best set of actions to take in each state to maximize Gbike's profit. This involves two steps: policy evaluation, where we calculate the expected future rewards of each action for each state, and policy improvement, where we update our policy based on the value function calculated during policy evaluation. We repeat these steps until we converge to the optimal policy.

Algorithm

1) Set the maximum number of bikes that can be parked at each location to 20.
2) Define the expected number of rental requests and returns for each location.
3) Define the cost of moving a bike from one location to the other.
4) Define the rental reward.
5) Define the discount rate.

6) Define the Poisson probability function.
7) Initialize the state-value function for all possible states to 0.
8) Initialize the policy function for all possible states to (0, 0).
9) Define a function to calculate the expected reward for a given state and action.
10) Define a function to perform one step of policy evaluation.
11) Define a function to perform one step of policy improvement.
12) Define a function to perform policy iteration.
13) Call policy_iteration function to obtain the optimal policy and state-value function.
14) Print the final policy for each state.

IV. Problem-3

The Gbike bicycle rental problem has a new change where an employee who works at the first rental location is willing to shuttle one bike to the second location for free. Also, if we park more than 10 bikes at a location overnight, we need to pay an extra INR 4 for a second parking lot.

Approach Start by initializing some variables, including the maximum number of bikes at each location, the cost of moving bikes, and the cost of parking bikes.

2) Initialize a value function and a policy.

3) Use a loop to do policy iteration. In each iteration, we evaluate the current policy by calculating the expected profit of moving bikes according to the policy. We use this to update the value function.

4) Improve the policy by choosing the action that leads to the highest expected profit, given the current value function.

5) Keep doing this until the policy stops changing. Then, we have found the best policy for the Gbike bicycle rental problem with the given changes.

In simple terms, policy iteration is a technique for finding the best strategy for moving bikes between rental locations to make the most money. By using it, we can take into account the changes in the problem such as the employee shuttle, additional parking costs, and limited parking space. Th

V. Conclusion

1st Conclusion

This lab has demonstrated the application of reinforcement learning in solving complex decision-making problems, using the value iteration algorithm to find the optimal policy for an agent moving through a stochastic environment. This approach has potential in various domains such as robotics, game AI, and autonomous systems, and we can expect even greater successes with the development of more advanced algorithms and techniques in the field of reinforcement learning.

2nd Conclusion

We have applied the policy iteration algorithm to solve a finite MDP problem of managing two bike rental locations to determine the optimal policy for moving bikes between the locations to maximize expected profit. The results show that the policy iteration algorithm is an effective method for solving finite MDP problems, and the code implementation and explanations provided can serve as a reference for future work in similar MDP problems.

References

[1] Reinforcement Learning: An Introduction by Richard S. Sutton and Andrew G. Barto (Free Online Book): http://incompleteideas.net/book/RLbook2020.pdf
[2] OpenAI Gym: https://gym.openai.com/
[3] Policy Iteration Algorithm in Reinforcement Learning: https://towardsdatascience.com/policy-iteration-in-reinforcement-learning-70e7af71b3e8
[4] The TensorFlow library, The PyTorch library, The RL-Toolkit.
[5] https://www.google.com/
[6] https://chat.openai.com/