Abdullah_Exercise0

May 18, 2024

0.1 Course: APS1080: Introduction to Reinforcement Learning

Student Name: Abdullah Rasul

Student ID: 1011328243

Assignment: Exercise 0: Preliminaries

The goal of this exercise is to start you thinking about reinforcement learning, and to get you started with python programming in an AI context.

A. Context: AI, Autonomy, Search, Data Science, Control

AI is ultimately the science behind the engineering of autonomous machines. "AI" includes a variety of disparate tools, each of which have different kinds of scope.

1 1

1. What is an autonomous machine, and what does autonomy mean?

Ans:

Autonomous Machine:

A system that is capable of performing tasks and making decisions without the intervention and help of humans. This system is a combination of various senosrs, artifical intelligence, and actuators to percieve the surrounding world and make decisions accordingly without taking help from humans.

Autonomy:

Autonomy means the capacity to operate independently, making decisions and taking actions without external control. It encompasses various degrees, from assistance (where a system helps a human operator) to full control (where the system operates entirely on its own). Autonomy involves self-sufficiency in performing tasks and solving problems.

---

2 2a

2a. Classical AI techniques employ trees of decision constructs (if statements, etc.) to achieve some form of "intelligent"-like behavior. An example of this is the famous Eliza AI-based "psychotherapist". **What do you feel are the strengths and weaknesses of such an approach? State and explain three such strengths and three weaknesses.**

Ans:

Classical AI techniques laid the groundwork for modern AI, offering clear benefits in terms of simplicity, determinism, and computational efficiency. However, their limitations in adaptability, robustness, and scalability highlight the need for more advanced and flexible approaches in contemporary AI applications.

3 Strengths of Classical AI Techniques

1 - Simplicity:

Decision trees (like Eliza) and rule-based systems are straightforward to design and implement. Their logical structure is similar to if-else problems and is clear and easy to understand, and making debugging simpler.

2 - Consistent Behavior:

Classical AI techniques provide predictable and repeatable outcomes for given inputs. This determinism is crucial in applications where consistency and reliability are necessary.

3 - Low Computational Requirements:

These systems often require less computational power compared to modern AI techniques like deep learning. They are suitable for applications with limited processing capabilities.

4 Weaknesses of Classical AI Techniques

1 - Limited Adaptability and Scalability:

Trees based systems are rigid and do not adapt to new situations without manual updates. This lack of adaptability makes them impractical for dynamic and complex environments.

2 - Fragile Systems:

These systems can fail catastrophically when encountering scenarios not covered by their rules. They lack robustness and flexibility, often breaking when dealing with noisy or incomplete data.

3 - Intensive Development:

Creating and maintaining extensive rule-based systems is time-consuming and requires significant domain expertise. The process of manually encoding knowledge is both tedious and error-prone.

---

5 2b

2b. Data Science includes signal processing, adaptive filters, supervised and unsupervised machine learning. These techniques solve the modeling problem from the perspective of the data. That is, they serve to classify signals (data), interpolate signals (data), or extract signal from noise. **Comment on how these functions relate to artificial intelligence?**

Ans:

Data science techniques are integral to artificial intelligence (AI) as they provide the foundational methods for analyzing and interpreting data, which is essential for AI to function effectively. Classification helps AI systems categorize data into meaningful groups, enabling tasks such as image recognition, language processing, and predictive analytics. Interpolation allows AI models to estimate unknown values and make predictions based on existing data, enhancing their ability to perform tasks like forecasting and recommendation.

Additionally, noise reduction through signal processing and adaptive filters ensures that AI models work with clean, high-quality data, which is crucial for accuracy and reliability. By extracting relevant signals from noisy data, AI systems can make more precise decisions and insights. Together, these data science functions enable AI to learn from data, adapt to new information, and perform complex tasks across various domains.

- **Signal Processing**: Extracts signals from noise to enhance data quality for AI applications.
- **Adaptive Filters**: Dynamically adjust to filter out noise, improving the accuracy of AI models.
- **Supervised Machine Learning**: Classifies signals (data) based on labeled training data for predictive tasks.
- **Unsupervised Machine Learning**: Identifies patterns and groups in data without predefined labels, aiding in data exploration and clustering.

---

6 2c

2c. Control theory concerns the development of feedback laws (i.e., control laws) that strive to push a system (the "plant") to a desired state. It does this based on observation of essential characteristics of the plant (the state, or some possibly noisy or lossy observation of the state), and construction of an appropriate feedback signal. How does this tool compare with the prior two, relating to the development of an autonomous system? **Explain the similarities and differences from a structural (high level) point of view.**

Ans:

Control theory, unlike data science techniques and classical AI approaches, focuses on regulating the behavior of dynamic systems, such as autonomous vehicles or robots, to achieve desired outcomes.

6.0.1 Similarities:

Like data science and classical AI, control theory utilizes feedback mechanisms. It continuously monitors the system's state and adjusts control signals accordingly to maintain or achieve desired performance.

All three tools aim to achieve specific objectives. While data science focuses on analyzing data to make predictions or classifications, control theory and AI strive to control or optimize system behavior to achieve desired states or outcomes.

6.0.2 Differences:

Control theory specifically addresses dynamic systems, where the system's behavior evolves over time due to internal dynamics or external influences. In contrast, data science and classical AI techniques are more general and can be applied to static as well as dynamic systems.

Control theory often involves real-time adaptation of control signals to respond to dynamic changes in the system. This differs from data science and classical AI techniques, which typically involve offline training and batch processing of data.

In control theory, the system's behavior is directly manipulated through control signals based on observations of the system's state. In contrast, data science and AI techniques primarily involve analyzing data to infer patterns and make decisions, without directly controlling the underlying system.

---

7 B. Practice

Consider the game of tic tac toe (TTT). Two individuals (i.e., an X-player, and an O-player) play on a 9-position board, alternately placing "X" or "O" markers on the board. A player wins when they have achieved three markers of their type, in a row (vertical, horizontal, diagonal).

Code a TTT player in the following manner:

1. Create a python class maintains the TTT board. It should have a reset method (to clear the TTT board), methods for setting the position of the board for each player, and a method to indicate whether the game has been won (returning who has won), or whether the game is at a draw.
2. Test this python class, by playing a two-human game of TTT.

[ ]:
```python
class TicTacToe:
    def __init__(self):
        self.board = [' ' for _ in range(9)]
        self.current_winner = None

    def print_board(self):
        for row in [self.board[i*3:(i+1)*3] for i in range(3)]:
            print('| ' + ' | '.join(row) + ' |')

    def print_board_nums():
        number_board = [[str(i) for i in range(j*3, (j+1)*3)] for j in range(3)]
        for row in number_board:
            print('| ' + ' | '.join(row) + ' |')
```

```
    def available_moves(self):
        return [i for i, spot in enumerate(self.board) if spot == ' ']

    def empty_squares(self):
        return ' ' in self.board

    def num_empty_squares(self):
        return self.board.count(' ')

    def make_move(self, square, letter):
        if self.board[square] == ' ':
            self.board[square] = letter
            if self.winner(square, letter):
                self.current_winner = letter
            return True
        return False

    def winner(self, square, letter):
        # check row
        row_ind = square // 3
        row = self.board[row_ind*3:(row_ind+1)*3]
        if all([spot == letter for spot in row]):
            return True
        # check column
        col_ind = square % 3
        column = [self.board[col_ind+i*3] for i in range(3)]
        if all([spot == letter for spot in column]):
            return True
        # check diagonal
        if square % 2 == 0:
            diagonal1 = [self.board[i] for i in [0, 4, 8]]
            if all([spot == letter for spot in diagonal1]):
                return True
            diagonal2 = [self.board[i] for i in [2, 4, 6]]
            if all([spot == letter for spot in diagonal2]):
                return True
        return False

def play_game():
    game = TicTacToe()
    x_player = 'X'
    o_player = 'O'
    current_player = x_player

    while game.empty_squares():
        game.print_board_nums()
```

```
        print()
        game.print_board()
        print(f"\nPlayer {current_player}'s turn.")
        available_moves = game.available_moves()
        move = None
        while move not in available_moves:
            try:
                move = int(input("Choose a position from 0-8: "))
            except ValueError:
                print("Invalid input. Please enter a number from 0 to 8.")
        game.make_move(move, current_player)

        if game.current_winner:
            print("\nCongratulations! Player", current_player, "wins!")
            game.print_board()
            break
        if game.num_empty_squares() == 0:
            print("\nIt's a draw!")
            game.print_board()
            break
        current_player = o_player if current_player == x_player else x_player

if __name__ == "__main__":
    play_game()
```

```
| 0 | 1 | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |

|   |   |   |
|   |   |   |
|   |   |   |

Player X's turn.
Choose a position from 0-8: 2
| 0 | 1 | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |

|   |   | X |
|   |   |   |
|   |   |   |

Player O's turn.
Choose a position from 0-8: 3
| 0 | 1 | 2 |
| 3 | 4 | 5 |
```

```
| 6 | 7 | 8 |

|   |   | X |
| O |   |   |
|   |   |   |

Player X's turn.
Choose a position from 0-8: 4
| 0 | 1 | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |

|   |   | X |
| O | X |   |
|   |   |   |

Player O's turn.
Choose a position from 0-8: 6
| 0 | 1 | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |

|   |   | X |
| O | X |   |
| O |   |   |

Player X's turn.
Choose a position from 0-8: 5
| 0 | 1 | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |

|   |   | X |
| O | X | X |
| O |   |   |

Player O's turn.
Choose a position from 0-8: 1
| 0 | 1 | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |

|   | O | X |
| O | X | X |
| O |   |   |

Player X's turn.
Choose a position from 0-8: 8
```

```
Congratulations! Player X wins!
|   | O | X |
| O | X | X |
| O |   | X |
```

---

3. Now you are to create a computer player. The goal of the computer player is to win if possible, or at worst, not lose. First think about the strategy that you need to codify, and then codify it.

4. Test your python class, by playing a human-vs-computer game of TTT.

[ ]:
```python
import random

class TicTacToe:
    def __init__(self):
        self.board = [' ' for _ in range(9)]
        self.current_winner = None

    def print_board(self):
        for row in [self.board[i*3:(i+1)*3] for i in range(3)]:
            print('| ' + ' | '.join(row) + ' |')

    def print_board_nums():
        number_board = [[str(i) for i in range(j*3, (j+1)*3)] for j in range(3)]
        for row in number_board:
            print('| ' + ' | '.join(row) + ' |')

    def available_moves(self):
        return [i for i, spot in enumerate(self.board) if spot == ' ']

    def empty_squares(self):
        return ' ' in self.board

    def num_empty_squares(self):
        return self.board.count(' ')

    def make_move(self, square, letter):
        if self.board[square] == ' ':
            self.board[square] = letter
            if self.winner(square, letter):
                self.current_winner = letter
            return True
        return False

    def winner(self, square, letter):
        # check row
```

```
        row_ind = square // 3
        row = self.board[row_ind*3:(row_ind+1)*3]
        if all([spot == letter for spot in row]):
            return True
        # check column
        col_ind = square % 3
        column = [self.board[col_ind+i*3] for i in range(3)]
        if all([spot == letter for spot in column]):
            return True
        # check diagonal
        if square % 2 == 0:
            diagonal1 = [self.board[i] for i in [0, 4, 8]]  # top-left to␣
↪bottom-right diagonal
            if all([spot == letter for spot in diagonal1]):
                return True
            diagonal2 = [self.board[i] for i in [2, 4, 6]]  # top-right to␣
↪bottom-left diagonal
            if all([spot == letter for spot in diagonal2]):
                return True
        return False

    def minimax(self, board, depth, maximizing_player):
        if self.current_winner == 'O':
            return {'position': None, 'score': 1}
        elif self.current_winner == 'X':
            return {'position': None, 'score': -1}
        elif not self.empty_squares():
            return {'position': None, 'score': 0}

        if maximizing_player:
            best = {'position': None, 'score': -float('inf')}
            for possible_move in self.available_moves():
                board_copy = board[:]
                board_copy[possible_move] = 'O'
                self.board = board_copy
                sim_score = self.minimax(board_copy, depth + 1, False)
                self.board[possible_move] = ' '
                sim_score['position'] = possible_move
                if sim_score['score'] > best['score']:
                    best = sim_score
        else:
            best = {'position': None, 'score': float('inf')}
            for possible_move in self.available_moves():
                board_copy = board[:]
                board_copy[possible_move] = 'X'
                self.board = board_copy
                sim_score = self.minimax(board_copy, depth + 1, True)
```

```
                self.board[possible_move] = ' '
                sim_score['position'] = possible_move
                if sim_score['score'] < best['score']:
                    best = sim_score
        return best

def play_game():
    game = TicTacToe()
    x_player = 'X'
    o_player = 'O'
    current_player = x_player

    while game.empty_squares():
        TicTacToe.print_board_nums()  # Adjusted this line
        print()
        game.print_board()
        print(f"\nPlayer {current_player}'s turn.")

        if current_player == x_player:
            move = None
            while move not in game.available_moves():
                try:
                    move = int(input("Choose a position from 0-8: "))
                except ValueError:
                    print("Invalid input. Please enter a number from 0 to 8.")
        else:
            if game.num_empty_squares() == 9:
                move = random.choice(game.available_moves())
            else:
                move = game.minimax(game.board, 0, True)['position']

        game.make_move(move, current_player)

        if game.current_winner:
            print("\nCongratulations! Player", current_player, "wins!")
            game.print_board()
            break
        if game.num_empty_squares() == 0:
            print("\nIt's a draw!")
            game.print_board()
            break
        current_player = o_player if current_player == x_player else x_player

if __name__ == "__main__":
    play_game()
```

```
| 0 | 1 | 2 |
```

```
| 3 | 4 | 5 |
| 6 | 7 | 8 |

|   |   |   |
|   |   |   |
|   |   |   |

Player X's turn.
Choose a position from 0-8: 9
Choose a position from 0-8: 8
| 0 | 1 | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |

|   |   |   |
|   |   |   |
|   |   | X |

Player O's turn.
| 0 | 1 | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |

| O |   |   |
|   |   |   |
|   |   | X |

Player X's turn.
Choose a position from 0-8: 7
| 0 | 1 | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |

| O |   |   |
|   |   |   |
|   | X | X |

Player O's turn.
| 0 | 1 | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |

| O | O |   |
|   |   |   |
|   | X | X |

Player X's turn.
Choose a position from 0-8: 3
```

```
| 0 | 1 | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |

| O | O |   |
| X |   |   |
|   | X | X |

Player O's turn.

Congratulations! Player O wins!
| O | O | O |
| X |   |   |
|   | X | X |
```

What helper methods do you need to create to make this easier? What feedback is needed? What is your space of actions?

Ans:

Feedback needed during the game includes indicating whose turn it is, providing prompts for player input, informing players of game outcomes (win, lose, draw), and displaying the current state of the board.

The space of actions for the computer player includes selecting an available move on the board, which can be determined using the Minimax algorithm or other strategies such as heuristic approaches.

`print_board`: A method to print the current state of the board to the console, making it easier for players to visualize the game.

`print_board_nums`: A static method to print the numbers corresponding to each position on the board, helping players identify which positions they can choose.

`available_moves`: A method to return a list of available moves (empty positions) on the board, aiding in checking for valid player moves and assisting the computer player in selecting its move.

`empty_squares`: A method to check if there are any empty squares remaining on the board, helping determine when the game ends in a draw.

`num_empty_squares`: A method to count the number of empty squares remaining on the board, facilitating decision-making by the computer player.

`make_move`: A method to update the board with a player's move, allowing players to interact with the game and assisting in simulating moves for the computer player.

`winner`: A method to determine if a player has won the game after making a move, aiding in detecting game-winning conditions and ending the game accordingly.

`minimax`: A method to implement the Minimax algorithm for the computer player, enabling it to make optimal moves by recursively evaluating potential moves and selecting the one with the highest score.