Program Development Laboratory

Term Project - Odd Semester 2022
December 13, 2022

A Vision Based Reinforcement Learning Control System

Viraj Shah (211080023)

as part of
B. Tech Information Technology, Semester III

Problem Statement

Implement a reinforcement environment consisting of a vision based, differential wheeled robot and points of interest to either navigate towards or avoid. The objective of the robot is to maximise the number of positive targets, and to minimize the number of negative targets visited.

Motivation

Humans rely heavily on vision to observe, identify, and devise appropriate courses of action to deal with various objects of interest. It is of key interest to emulate this behaviour in autonomous robots whose targets or points of interest are not confined to a fixed region, and are required to be seeked out and navigated to. While such a task pushes the limits of conventional control systems, it can be achieved through the use of a reinforcement learning based approach, which would be better equipped to combat the increasing scales of complexity. This project implements a modular environment suitable for the training and testing of such an agent, and proposes a simple temporal difference based tabular reinforcement learning approach to solve the aforementioned environment. This kind of environment uses a high level of generalization of any specific tasks to be accomplished by the robot, and hence this in conjunction with transfer learning can be used to generate a basic foundational policy which can be specialized to more specific tasks through further training.

Contents

1 Foundational Theory

Reinforcement learning (RL) can be described as a computational method to approach learning from interaction. RL is a goal-directed, interaction driven approach to learning and automated decision making. It separates itself from different techniques by using direct interaction with the environment as a source of information.

Reinforcement learning is different from other machine learning problems, in the way that it focuses on learning, not through evaluating the end result obtained by an agent (or model in case of other subcategories of machine learning), but rather by taking into consideration whatever the agent observes, the actions it takes, and their effects on the environment.

In order to practically implement and manage the many RL algorithms in a uniform manner, it is only natural that we require all kinds of environments and agents share a uniform, standardized interface such that the internal workings of the environment are abstracted away and agents, regardless of their policies can be made to interact with any environment with minimal adjustment (whether the agent will perform well or not perform at all is a separate consideration). To achieve this level of standardization, agents are implemented as to interact with environments through the OpenAI Gym API, and environments are written as to conform to its specifications.

The temporal difference based tabular method adopted by this project, Q learning, involves creation of a Q table which stores state-action values for all possible combinations. In a Monte Carlo like fashion, the agent is made to sample values and update the table accordingly. However, instead of waiting for the episode to be completed before integrating its new experiences into its decision process, it takes a page from dynamic programming's book and uses bootstrapping to turn it into an on-policy technique. The hybrid update is given by:

$$q(s,a) \longleftarrow q(s,a) + \alpha[r + \gamma \max_{a} q(s',a) - q(s,a)] \tag{1}$$

2 Solution

2.1 main.py

```python
from game import Environment
import numpy as np

from config import *

env = Environment('human')

observation, info = env.reset()

steps = 2 ** 12

alpha = 0.15
gamma = 0.995
eps = 0.1

try:
    q_table = np.load(r'weights.npy')
except OSError:
    q_table = np.zeros((3,) * SENSOR_COUNT + (2, 3), dtype=np.float64)

observation, info = env.reset()
state = tuple(observation)

for step in range(steps):
    if np.random.uniform(0, 1) < eps:
        action = tuple(env.action_space.sample())
    elif state == (1,) * SENSOR_COUNT:
        action = (0, 0)
    else:
        action = np.unravel_index(np.argmax(q_table[state]), (2, 3))

    prevState = state
    observation, reward, terminated, truncated, info = env.step(action)
    state = tuple(observation)

    q_table[prevState + action] = (1 - alpha) * q_table[prevState + action] + \
        alpha * (reward + (gamma * np.max(q_table[state]) if not terminated else 0))

    if terminated or truncated:
        observation, info = env.reset()
        state = tuple(observation)
    print(f'Step {step + 1} / {steps}', end=' ' * 32 + '\r')

env.close()

np.save(r'weights', q_table)
```

2.2 game.py

```
from math import exp
import numpy as np

import pygame
from pygame.locals import *

import gym
from gym import spaces

from sprites import Player, Sensor, Bit
from utils import Color
from config import *



class Environment(gym.Env):
    metadata = {'render_modes': ['human'], 'render_fps': 60}

    def __init__(self, render_mode=None):
        assert render_mode is None or render_mode in Environment.metadata['render_modes']
        self.render_mode = render_mode

        self.observation_space = spaces.MultiDiscrete(SENSOR_COUNT * [3], dtype=np.int32)
        self.action_space = spaces.MultiDiscrete((2, 3), dtype=np.int32)

        pygame.init()
        self.clock = None
        self.screen = None

        self.UPDATEBITS = pygame.USEREVENT + 1
        pygame.time.set_timer(self.UPDATEBITS, 1000)


    def reset(self, seed=None, options=None):
        super().reset(seed=seed)

        self.player = Player()
        self.bits = pygame.sprite.Group()
        self.sensors = pygame.sprite.Group(
            [
                Sensor(self.player, t) for t in
                np.linspace(-SENSOR_FOV / 2, SENSOR_FOV / 2, SENSOR_COUNT)
            ]
        )
        self.all_sprites = pygame.sprite.Group(self.player, self.sensors)

        observation = np.array(
            [sensor.update(self.bits) for sensor in self.sensors], dtype=np.int32
        ) + 1
        info = self.player.score

        self._render_frame()

        return (observation, info)

    def step(self, action):
        for event in pygame.event.get():
            if event.type == self.UPDATEBITS:
                p = 1 / (1 + exp(12 - len(self.bits)))

                if np.random.random() > p:
                    new_bit = Bit()
                    self.bits.add(new_bit)
                    self.all_sprites.add(new_bit)
                if np.random.random() < p:
                    old_bit = self.bits.sprites()[0]
                    old_bit.kill()

        keys = {K_w: False, K_a: False, K_d: False}

        if action[0] == 1: keys[K_w] = True

        if action[1] == 0: keys[K_a] = True
        elif action[1] == 2: keys[K_d] = True

        reward = self.player.update(keys, self.bits)
        observation = np.array([sensor.update(self.bits) for sensor in self.sensors], dtype=np.int32)
        info = self.player.score

        self._render_frame()

        return observation, reward, False, False, info
```

```
def render(self):
    pass


def _render_frame(self):
    if self.screen is None and self.render_mode == 'human':
        pygame.init()
        pygame.display.init()
        self.screen = pygame.display.set_mode((SCREEN_WIDTH, SCREEN_HEIGHT))
    if self.clock == None and self.render_mode == 'human':
        self.clock = pygame.time.Clock()

    for sprite in self.all_sprites:
        sprite.draw()

    if self.render_mode == 'human':
        self.screen.fill(Color.WHITE)
        for e in self.all_sprites:
            self.screen.blit(e.surf, e.rect)
        pygame.display.flip()
        self.clock.tick(Environment.metadata['render_fps'])


def close(self):
    if self.screen is not None:
        pygame.display.quit()
        pygame.quit()


def play(self):
    self.reset()

    running = True
    while running:
        for event in pygame.event.get():
            if event.type == KEYDOWN:
                if event.key == K_ESCAPE:
                    running = False

            elif event.type == self.UPDATEBITS:
                p = 1 / (1 + exp(12 - len(self.bits)))

                if np.random.random() > p:
                    new_bit = Bit()
                    self.bits.add(new_bit)
                    self.all_sprites.add(new_bit)
                    if np.random.random() < p:
                        old_bit = self.bits.sprites()[0]
                        old_bit.kill()

                elif event.type == QUIT:
                    running = False

            keys = pygame.key.get_pressed()
            self.player.update(keys, self.bits)
            self.sensors.update(self.bits)

            for sprite in self.all_sprites:
                sprite.draw()

            self.screen.fill(Color.WHITE)
            for e in self.all_sprites:
                self.screen.blit(e.surf, e.rect)

            pygame.display.flip()
            print(f'Score: {self.player.score}', end=' ' * 8 + '\r')
            self.clock.tick(Environment.metadata['render_fps'])



if __name__ == '__main__':
    env = Environment('human')
    env.play()
    env.close()
```

2.3 sprites.py

```
import numpy as np

import pygame
from pygame.locals import *

from utils import Color, rotate
from config import *



class Player(pygame.sprite.Sprite):
    def __init__(self):
        super(Player, self).__init__()

        self.v = np.array([PLAYER_SPEED, 0], dtype=np.float64)
        self.theta = 0
        self.score = 0

        self.surf = pygame.Surface(2 * (PLAYER_SIZE,), pygame.SRCALPHA)
        self.mask = pygame.mask.from_surface(self.surf)
        self.rect = self.surf.get_rect(center=(SCREEN_WIDTH / 2, SCREEN_HEIGHT / 2))


    def update(self, keys, bits):
        if keys[K_w]:
            self.rect.move_ip(rotate(self.v, self.theta))
        if keys[K_a]:
            self.theta -= PLAYER_TORQUE
        if keys[K_d]:
            self.theta += PLAYER_TORQUE

        if self.rect.left <= 0:
            self.rect.left = 0
        elif self.rect.right >= SCREEN_WIDTH:
            self.rect.right = SCREEN_WIDTH
        if self.rect.top <= 0:
            self.rect.top = 0
        elif self.rect.bottom >= SCREEN_HEIGHT:
            self.rect.bottom = SCREEN_HEIGHT

        delta = 0
        collided_bits = pygame.sprite.spritecollide(self, bits, True)
        for bit in collided_bits:
            delta += bit.point
            bit.kill()

        self.score += delta
        return delta


    def draw(self):
        arrow = rotate(
            np.array([
                [ 10, -5, -5],
                [  0,  5, -5]
            ], dtype=np.int32),
            self.theta
        ) + np.array([2 * [PLAYER_SIZE / 2]], dtype=np.int32).T

        pygame.draw.circle(self.surf, Color.CYAN, 2 * (PLAYER_SIZE / 2,), PLAYER_SIZE / 2)
        pygame.draw.circle(self.surf, Color.BLUE, 2 * (PLAYER_SIZE / 2,), PLAYER_SIZE / 2, 3)
        pygame.draw.polygon(self.surf, Color.BLACK, list(zip(arrow[0], arrow[1])))
```

```
class Sensor(pygame.sprite.Sprite):
    def __init__(self, player, theta_offset):
        super(Sensor, self).__init__()

        self.player = player
        self.init_los_a = rotate(np.array([PLAYER_SIZE / 2, 0], dtype=np.int32), theta_offset)
        self.init_los_b = rotate(np.array([SENSOR_LOS_LEN, 0], dtype=np.int32), theta_offset)
        self.los_a = self.init_los_a
        self.los_b = self.init_los_b
        self.reading = 0

        self.surf = pygame.Surface(2 * (SENSOR_LOS_LEN * 2,), pygame.SRCALPHA)
        self.mask = pygame.mask.from_surface(self.surf)
        self.rect = self.surf.get_rect(
            center=self.player.rect.center
        )


    def update(self, bits):
        self.los_a = rotate(self.init_los_a, self.player.theta)
        self.los_b = rotate(self.init_los_b, self.player.theta)
        self.rect.clamp_ip(self.player.rect)

        self.reading = 0
        for bit in filter(lambda b: pygame.sprite.collide_mask(self, b), bits):
            self.reading += bit.point
        self.reading = np.clip(self.reading, -1, 1)

        return self.reading + 1


    def draw(self):
        color = {-1: Color.RED, 0: Color.WHITE, 1: Color.GREEN}[self.reading]

        self.surf.fill(pygame.SRCALPHA)
        pygame.draw.line(
            self.surf, color,
            2 * (SENSOR_LOS_LEN,) + self.los_a,
            2 * (SENSOR_LOS_LEN,) + self.los_b,
        )

        self.mask = pygame.mask.from_surface(self.surf)
```

```python
class Bit(pygame.sprite.Sprite):
    def __init__(self):
        super(Bit, self).__init__()

        self.point = -1 if np.random.random() < 0.25 else 1

        self.surf = pygame.Surface(2 * (BIT_SIZE,), pygame.SRCALPHA)
        self.mask = pygame.mask.from_surface(self.surf)
        self.rect = self.surf.get_rect(
            center=(
                np.random.randint(10, SCREEN_WIDTH - 10),
                np.random.randint(10, SCREEN_HEIGHT - 10)
            )
        )


    def update(self):
        # The bits just stay where they are, for now atleast.
        pass


    def draw(self):
        pygame.draw.circle(self.surf,
            Color.GREEN if self.point > 0 else
            Color.RED, 2 * (BIT_SIZE / 2,), BIT_SIZE / 2
        )
        pygame.draw.circle(self.surf,
            Color.BLACK, 2 * (BIT_SIZE / 2,), BIT_SIZE / 2, 2
        )

        self.mask = pygame.mask.from_surface(self.surf)
```

2.4 config.py

```python
from math import pi

SCREEN_WIDTH = 540
SCREEN_HEIGHT = 540

PLAYER_SIZE = 54
PLAYER_SPEED = 12
PLAYER_TORQUE = 0.1
PLAYER_BOUND = 120
```

```
SENSOR_LOS_LEN = 210
SENSOR_COUNT = 3
SENSOR_FOV = pi / 6

BIT_SIZE = 27
```

2.5 utils.py

```
from math import sin, cos
import numpy as np

class Color:
    RED     = (0xFF, 0x00, 0x00)
    GREEN   = (0x00, 0xFF, 0x00)
    BLUE    = (0x00, 0x00, 0xFF)
    CYAN    = (0x00, 0xFF, 0xFF)
    YELLOW  = (0xFF, 0x00, 0xFF)
    MAGENTA = (0xFF, 0xFF, 0x00)
    BLACK   = (0x00, 0x00, 0x00)
    GRAY    = (0x96, 0x96, 0x96)
    WHITE   = (0xFF, 0xFF, 0xFF)

def rotate(v, theta):
    s = sin(theta)
    c = cos(theta)

    m = np.array([
        [c, -s],
        [s,  c]
    ], dtype=np.float64)

    return np.matmul(m, v)
```

2.6 test.py

```
from game import Environment
import numpy as np

from config import *

env = Environment('human')

observation, info = env.reset()

steps = 2 ** 10
```

```
q_table = np.load(r'weights.npy')

observation, info = env.reset()
state = tuple(observation)

for step in range(steps):
    if state == (1,) * SENSOR_COUNT:
        action = (0, 0)
    else:
        action = np.unravel_index(np.argmax(q_table[state]), (2, 3))

    observation, reward, terminated, truncated, info = env.step(action)
    state = tuple(observation)

    if terminated or truncated:
        observation, info = env.reset()
        state = tuple(observation)

    # print(*state, end=' ' * 32 + '\r')

env.close()
```

2.7 read_q.py

```
from itertools import product
import numpy as np

weights = np.load(r'weights.npy')

for i in product(range(3), range(3), range(3)):
    print(i)
    print(weights[i])
    print()
```

3 Results

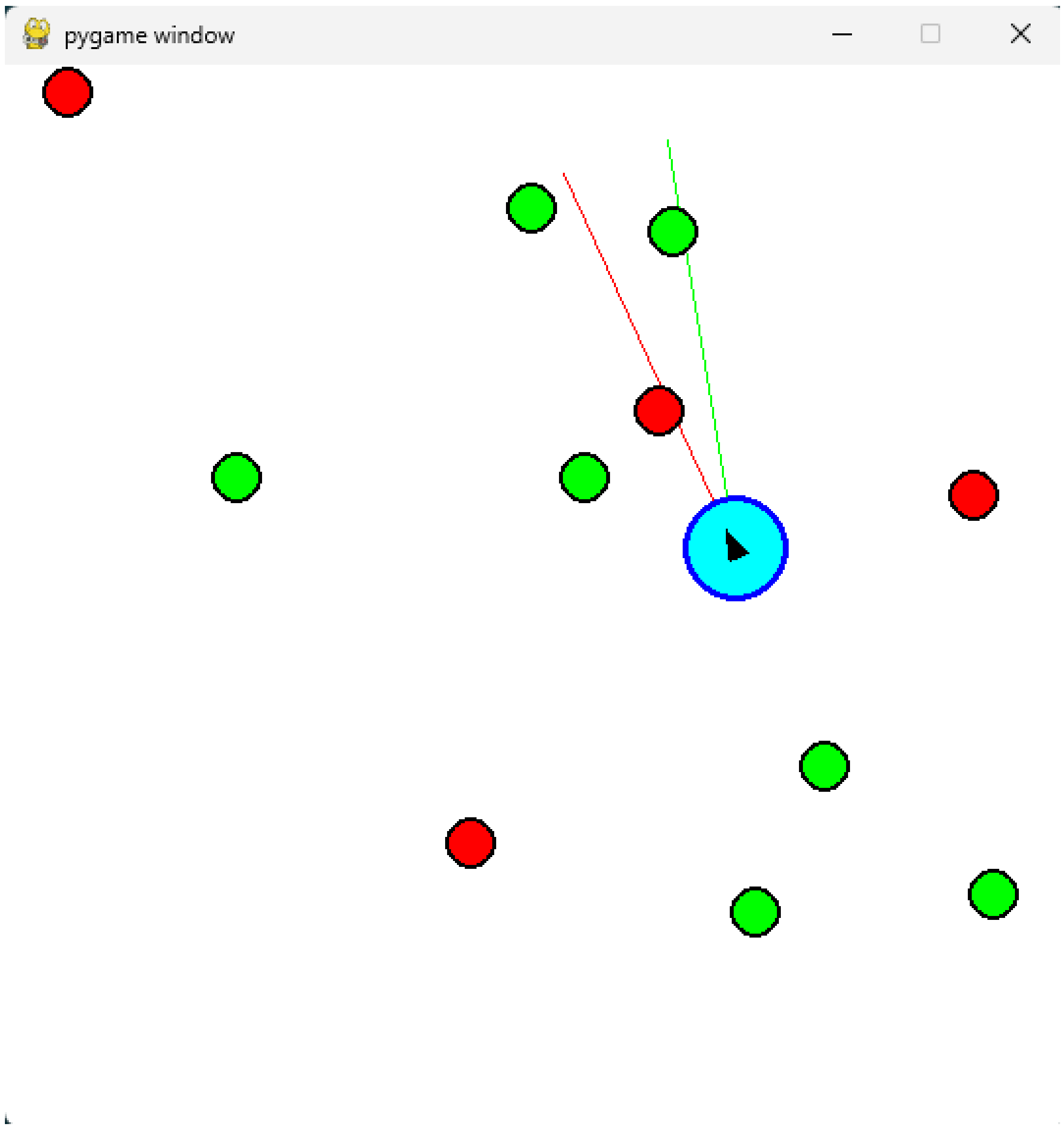

Figure 1: The simulated environment.

4 Discussion

Through this project, I worked on various scripts cumulating into a highly generalized and extensible representation of a vision based control system environment and a computationally inexpensive solution to it.

The environment is designed to conform to and extend the OpenAI Gym API for reinforcement learning methods, providing a library-like interface to facilitate the design of new learning methods or model architectures.

Some improvement can be made to the learning algorithm, as by inspection of the learnt Q table, it is observed that a large number of possible states are meaningless and do not require to be held in memory. Approaches like Q learning are widely applicable, but invariably struggle when the state-action space become so large that it is no longer feasible to represent and update the learnt Q function tabularly, which forces us to replace our Q function with an approximation, such as a simple linear approximator or more effectively, a neural network.

An example of a possible specific scenario covered by this generalization would be autonomous driving systems. The environment can be effectively extended to assign different priories to the targets and hence treat them as obstacles (negative) or navigational waypoints (positive). Various papers have proposed Deep Reinforcement Learning for autonomous driving. In self-driving cars, there are various aspects to consider, such as speed limits at various places, drivable zones, avoiding collisions-just to mention a few, represented by the objectives. AWS DeepRacer is an autonomous racing car that has been designed to test out RL in a physical track. It uses cameras to visualize the runway and a reinforcement learning model to control the throttle and direction, as is with our environment.

5 Conclusion

Reinforcement Learning problems have many possible solution methods, each more suitable for a given type of problem, with more state of the art methods generally coming ever so close to effectively solving the entire reinforcement learning problem with its broadest definition. This project is a journey towards such a method, with each technique bringing new improvements over the last.