AI BASED SNAKE GAME USING Q - LEARNING.

Snake game using deep learning is based on the reinforcement learning. In the snake game the snake eats the apple within the boundaries and not colliding with its own tail.

The rules for the snake game are:

--> The snake will try to eat the apple but the steps for the snake to eat the apple are limited.

--> The snake must not collide with its own tail and boundaries to keep itself alive.

--> The snake grows by 1 unit after eating the apple and then the games becomes even more complex to play.

--> There are 4 directions in the snake game which are left, right, top and bottom.

--> The boundaries are fixed and finite for the snake to move and achieve high score.

I have mounted my drive with the colab.

Install the pygame for the graphics and video libraries for the game which are already available in the pygame.

import videodriver for a window screen to popup for the snake game.

This is the code for the snake game and all the classes defined accordingly.

In [1]:
```
!pip install pygame
```

```
Requirement already satisfied: pygame in /Users/manasakilaru/opt/anaconda
3/lib/python3.9/site-packages (2.1.2)
```

In [2]:
```
#!/usr/bin/python
# -*-coding: utf-8 -*-

import contextlib
import random
import sys
import time
from operator import add, sub
from dataclasses import dataclass
from itertools import product
from typing import Tuple

with contextlib.redirect_stdout(None):
    import pygame
    from pygame.locals import *
from heapq import *

WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
RED = (255, 0, 0)
GREEN = (0, 255, 0)
BLUE = (0, 0, 255)
DARKGRAY = (40, 40, 40)


@dataclass
class Base:
    cell_size: int = 20
    cell_width: int = 12
    cell_height: int = 12
    window_width = cell_size * cell_width
    window_height = cell_size * cell_height

    @staticmethod
    def node_add(node_a: Tuple[int, int], node_b: Tuple[int, int]):
        result: Tuple[int, int] = tuple(map(add, node_a, node_b))
        return result

    @staticmethod
    def node_sub(node_a: Tuple[int, int], node_b: Tuple[int, int]):
        result: Tuple[int, int] = tuple(map(sub, node_a, node_b))
        return result

    @staticmethod
    def mean(l):
        return round(sum(l) / len(l), 4)


def heuristic(start, goal):
    return (start[0] - goal[0])**2 + (start[1] - goal[1])**2

# create the class for the apple base to have the random creation of apples
class Apple(Base):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.location = None
```

```
    def refresh(self, snake):
        """
        Generate a new apple
        """
        available_positions = set(product(range(self.cell_width - 1), range

        # If there's no available node for new apple, it reaches the perfec
        location = random.sample(available_positions, 1)[0] if available_po

        self.location = location

# create the class for the snake as well to have te blocks occupied by the
class Snake(Base):
    def __init__(self, initial_length: int = 3, body: list = None, **kwargs
        """
        :param initial_length: The initial length of the snake
        :param body: Optional. Specifying an initial snake body
        """
        super().__init__(**kwargs)
        self.initial_length = initial_length
        self.score = 0
        self.is_dead = False
        self.eaten = False

        # last_direction is only used for human player, giving it a default
        self.last_direction = (-1, 0)

        if body:
            self.body = body
        else:
            if not 0 < initial_length < self.cell_width:
                raise ValueError(f"Initial_length should fall in (0, {self.

            start_x = self.cell_width // 2
            start_y = self.cell_height // 2

            start_body_x = [start_x] * initial_length
            start_body_y = range(start_y, start_y - initial_length, -1)

            self.body = list(zip(start_body_x, start_body_y))

    def get_head(self):
        return self.body[-1]

    def dead_checking(self, head, check=False):
        """
        Check if the snake is dead
        :param check: if check is True, only return the checking result wit
        :return: Boolean
        """
        x, y = head
        if not 0 <= x < self.cell_width or not 0 <= y < self.cell_height or
            if not check:
                self.is_dead = True
            return True
        return False
```

```
    def cut_tail(self):
        self.body.pop(0)

    def move(self, new_head: tuple, apple: Apple):
        """
        Given the location of apple, decide if the apple is eaten (same loc
        :param new_head: (new_head_x, new_head_y)
        :param apple: Apple instance
        :return: Boolean. Whether the apple is eaten.
        """
        if new_head is None:
            self.is_dead = True
            return

        if self.dead_checking(head=new_head):
            return

        self.last_direction = self.node_sub(new_head, self.get_head())

        # make the move
        self.body.append(new_head)

        # if the snake eats the apple, score adds 1
        if self.get_head() == apple.location:
            self.eaten = True
            self.score += 1
        # Otherwise, cut the tail so that snake moves forward without growi
        else:
            self.eaten = False
            self.cut_tail()

# create the class for the player for the prediction of the moves for the s
class Player(Base):
    def __init__(self, snake: Snake, apple: Apple, **kwargs):
        """
        :param snake: Snake instance
        :param apple: Apple instance
        """
        super().__init__(**kwargs)
        self.snake = snake
        self.apple = apple

    def _get_neighbors(self, node):
        """
        fetch and yield the four neighbours of a node
        :param node: (node_x, node_y)
        """
        for diff in ((0, 1), (0, -1), (1, 0), (-1, 0)):
            yield self.node_add(node, diff)

    @staticmethod
    def is_node_in_queue(node: tuple, queue: iter):
        """
        Check if element is in a nested list
        """
        return any(node in sublist for sublist in queue)
```

```
    def is_invalid_move(self, node: tuple, snake: Snake):
        """
        Similar to dead_checking, this method checks if a given node is a v
        :return: Boolean
        """
        x, y = node
        if not 0 <= x < self.cell_width or not 0 <= y < self.cell_height or
            return True
        return False


class BFS(Player):
    def __init__(self, snake: Snake, apple: Apple, **kwargs):
        """
        :param snake: Snake instance
        :param apple: Apple instance
        """
        super().__init__(snake=snake, apple=apple, **kwargs)

    def run_bfs(self):
        """
        Run BFS searching and return the full path of best way to apple fro
        """
        queue = [[self.snake.get_head()]]

        while queue:
            path = queue[0]
            future_head = path[-1]

            # If snake eats the apple, return the next move after snake's h
            if future_head == self.apple.location:
                return path

            for next_node in self._get_neighbors(future_head):
                if (
                    self.is_invalid_move(node=next_node, snake=self.snake)
                    or self.is_node_in_queue(node=next_node, queue=queue)
                ):
                    continue
                new_path = list(path)
                new_path.append(next_node)
                queue.append(new_path)

            queue.pop(0)

    def next_node(self):
        """
        Run the BFS searching and return the next move in this path
        """
        path = self.run_bfs()
        return path[1]


class LongestPath(BFS):
    """
    Given shortest path, change it to the longest path
```

```
    """

    def __init__(self, snake: Snake, apple: Apple, **kwargs):
        """
        :param snake: Snake instance
        :param apple: Apple instance
        """
        super().__init__(snake=snake, apple=apple, **kwargs)
        self.kwargs = kwargs

    def run_longest(self):
        """
        For every move, check if it could be replace with three equivalent
        For example, for snake moving one step left, check if moving up, le
        move with equivalent longer move. Start this over until no move can
        """
        path = self.run_bfs()

        # print(f'longest path initial result: {path}')

        if path is None:
            # print(f"Has no Longest path")
            return

        i = 0
        while True:
            try:
                direction = self.node_sub(path[i], path[i + 1])
            except IndexError:
                break

            # Build a dummy snake with body and longest path for checking i
            snake_path = Snake(body=self.snake.body + path[1:], **self.kwar

            # up -> left, up, right
            # down -> right, down, left
            # left -> up, left, down
            # right -> down, right, up
            for neibhour in ((0, 1), (0, -1), (1, 0), (-1, 0)):
                if direction == neibhour:
                    x, y = neibhour
                    diff = (y, x) if x != 0 else (-y, x)

                    extra_node_1 = self.node_add(path[i], diff)
                    extra_node_2 = self.node_add(path[i + 1], diff)

                    if snake_path.dead_checking(head=extra_node_1) or snake
                        i += 1
                    else:
                        # Add replacement nodes
                        path[i + 1:i + 1] = [extra_node_1, extra_node_2]
                    break

        # Exclude the first node, which is same to snake's head
        return path[1:]
```

```
class Fowardcheck(Player):
    def __init__(self, snake: Snake, apple: Apple, **kwargs):
        """
        :param snake: Snake instance
        :param apple: Apple instance
        """
        super().__init__(snake=snake, apple=apple, **kwargs)
        self.kwargs = kwargs

    def run_forwardcheck(self):
        bfs = BFS(snake=self.snake, apple=self.apple, **self.kwargs)

        path = bfs.run_bfs()

        print("trying BFS")

        if path is None:
            snake_tail = Apple()
            snake_tail.location = self.snake.body[0]
            snake = Snake(body=self.snake.body[1:])
            longest_path = LongestPath(snake=snake, apple=snake_tail, **sel
            next_node = longest_path[0]
            # print("BFS not reachable, trying head to tail")
            # print(next_node)
            return next_node

        length = len(self.snake.body)
        virtual_snake_body = (self.snake.body + path[1:])[-length:]
        virtual_snake_tail = Apple()
        virtual_snake_tail.location = (self.snake.body + path[1:])[-length
        virtual_snake = Snake(body=virtual_snake_body)
        virtual_snake_longest = LongestPath(snake=virtual_snake, apple=virt
        virtual_snake_longest_path = virtual_snake_longest.run_longest()
        if virtual_snake_longest_path is None:
            snake_tail = Apple()
            snake_tail.location = self.snake.body[0]
            snake = Snake(body=self.snake.body[1:])
            longest_path = LongestPath(snake=snake, apple=snake_tail, **sel
            next_node = longest_path[0]
            # print("virtual snake not reachable, trying head to tail")
            # print(next_node)
            return next_node
        else:
            # print("BFS accepted")
            return path[1]


class Mixed(Player):
    def __init__(self, snake: Snake, apple: Apple, **kwargs):
        """
        :param snake: Snake instance
        :param apple: Apple instance
        """
        super().__init__(snake=snake, apple=apple, **kwargs)
        self.kwargs = kwargs

    def escape(self):
```

```
        head = self.snake.get_head()
        largest_neibhour_apple_distance = 0
        newhead = None
        for diff in ((0, 1), (0, -1), (1, 0), (-1, 0)):
            neibhour = self.node_add(head, diff)

            if self.snake.dead_checking(head=neibhour, check=True):
                continue

            neibhour_apple_distance = (
                abs(neibhour[0] - self.apple.location[0]) + abs(neibhour[1]
            )
            # Find the neibhour which has greatest Manhattan distance to ap
            if largest_neibhour_apple_distance < neibhour_apple_distance:
                snake_tail = Apple()
                snake_tail.location = self.snake.body[1]
                # Create a virtual snake with a neibhour as head, to see if
                # thus remove two nodes from body: one for moving one step
                snake = Snake(body=self.snake.body[2:] + [neibhour])
                bfs = BFS(snake=snake, apple=snake_tail, **self.kwargs)
                path = bfs.run_bfs()
                if path is None:
                    continue
                largest_neibhour_apple_distance = neibhour_apple_distance
                newhead = neibhour
        return newhead

    def run_mixed(self):
        """
        Mixed strategy
        """
        bfs = BFS(snake=self.snake, apple=self.apple, **self.kwargs)

        path = bfs.run_bfs()

        # If the snake does not have the path to apple, try to follow its t
        if path is None:
            return self.escape()

        # Send a virtual snake to see when it reaches the apple, does it st
        # alive
        length = len(self.snake.body)
        virtual_snake_body = (self.snake.body + path[1:])[-length:]
        virtual_snake_tail = Apple()
        virtual_snake_tail.location = (self.snake.body + path[1:])[-length
        virtual_snake = Snake(body=virtual_snake_body)
        virtual_snake_longest = BFS(snake=virtual_snake, apple=virtual_snak
        virtual_snake_longest_path = virtual_snake_longest.run_bfs()
        if virtual_snake_longest_path is None:
            return self.escape()
        else:
            return path[1]


class Astar(Player):
    def __init__(self, snake: Snake, apple: Apple, **kwargs):
        """
```

```
        :param snake: Snake instance
        :param apple: Apple instance
        """
        super().__init__(snake=snake, apple=apple, **kwargs)
        self.kwargs = kwargs

    def run_astar(self):
        came_from = {}
        close_list = set()
        goal = self.apple.location
        start = self.snake.get_head()
        dummy_snake = Snake(body=self.snake.body)
        neighbors = [(1, 0), (-1, 0), (0, 1), (0, -1), (-1, -1), (-1, 1), (
        gscore = {start: 0}
        fscore = {start: heuristic(start, goal)}
        open_list = [(fscore[start], start)]
        print(start, goal, open_list)
        while open_list:
            current = min(open_list, key=lambda x: x[0])[1]
            open_list.pop(0)
            print(current)
            if current == goal:
                data = []
                while current in came_from:
                    data.append(current)
                    current = came_from[current]
                    print(data)
                return data[-1]

            close_list.add(current)

            for neighbor in neighbors:
                neighbor_node = self.node_add(current, neighbor)

                if dummy_snake.dead_checking(head=neighbor_node) or neighbo
                    continue
                if sum(map(abs, self.node_sub(current, neighbor_node))) ==
                    diff = self.node_sub(current, neighbor_node)
                    if dummy_snake.dead_checking(head=self.node_add(neighbo
                                                 ) or self.node_add(neighbo
                        continue
                    elif dummy_snake.dead_checking(head=self.node_add(neigh
                                                   ) or self.node_add(neigh
                        continue
                tentative_gscore = gscore[current] + heuristic(current, nei
                if tentative_gscore < gscore.get(neighbor_node, 0) or neigh
                    gscore[neighbor_node] = tentative_gscore
                    fscore[neighbor_node] = tentative_gscore + heuristic(ne
                    open_list.append((fscore[neighbor_node], neighbor_node)
                    came_from[neighbor_node] = current


class Human(Player):
    def __init__(self, snake: Snake, apple: Apple, **kwargs):
        """
        :param snake: Snake instance
        :param apple: Apple instance
```

```
        """
        super().__init__(snake=snake, apple=apple, **kwargs)

    def run(self):
        for event in pygame.event.get():  # event handling loop
            if event.type == KEYDOWN:
                if (event.key == K_LEFT or event.key == K_a) and self.snake
                    diff = (-1, 0)  # left
                elif (event.key == K_RIGHT or event.key == K_d) and self.sn
                    diff = (1, 0)  # right
                elif (event.key == K_UP or event.key == K_w) and self.snake
                    diff = (0, -1)  # up
                elif (event.key == K_DOWN or event.key == K_s) and self.sna
                    diff = (0, 1)  # down
                else:
                    break
                return self.node_add(self.snake.get_head(), diff)
        # If no button is pressed down, follow previou direction
        return self.node_add(self.snake.get_head(), self.snake.last_directi


@dataclass
class SnakeGame(Base):
    fps: int = 60

    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.kwargs = kwargs

        pygame.init()
        self.clock = pygame.time.Clock()
        self.display = pygame.display.set_mode((self.window_width, self.win
        pygame.display.set_caption('Perfect Snake')

    def launch(self):
        while True:
            self.game()
            # self.showGameOverScreen()
            self.pause_game()

    def game(self):
        snake = Snake(**self.kwargs)

        apple = Apple(**self.kwargs)
        apple.refresh(snake=snake)

        step_time = []

        longgest_path_cache = []

        while True:
            # Human Player
            # new_head = Human(snake=snake, apple=apple, **self.kwargs).run

            # AI Player
            for event in pygame.event.get():  # event handling loop
                if event.type == QUIT or (event.type == KEYDOWN and event.k
```

```
                self.terminate()

        start_time = time.time()

        # BFS Solver
        # new_head = BFS(snake=snake, apple=apple, **self.kwargs).next_

        # Longest Path Solver
        # this solver is calculated per apple, not per move
        # if not longgest_path_cache:
        #     longgest_path_cache = LongestPath(snake=snake, apple=appl
        # new_head = longgest_path_cache.pop(0)

        # A star Solver
        # new_head = Astar(snake=snake, apple=apple, **self.kwargs).run

        # FORWARD CHECKING
        # new_head = Fowardcheck(snake=snake, apple=apple, **self.kwarg
        new_head = Mixed(snake=snake, apple=apple, **self.kwargs).run_m
        print(new_head)

        end_time = time.time()
        move_time = end_time - start_time
        # print(move_time)
        step_time.append(move_time)

        snake.move(new_head=new_head, apple=apple)

        if snake.is_dead:
            print(snake.body)
            print("Dead")
            break
        elif snake.eaten:
            apple.refresh(snake=snake)

        if snake.score + snake.initial_length >= self.cell_width * self
            break

        self.display.fill(BLACK)
        self.draw_panel()
        self.draw_snake(snake.body)

        self.draw_apple(apple.location)
        pygame.display.update()
        self.clock.tick(self.fps)

    print(f"Score: {snake.score}")
    print(f"Mean step time: {self.mean(step_time)}")

@staticmethod
def terminate():
    pygame.quit()
    sys.exit()

def pause_game(self):
    while True:
        time.sleep(0.2)
```

```
            for event in pygame.event.get():  # event handling loop
                if event.type == QUIT:
                    self.terminate()
                if event.type == KEYUP:
                    if event.key == K_ESCAPE:
                        self.terminate()
                    else:
                        return

    def draw_snake(self, snake_body):
        for snake_block_x, snake_block_y in snake_body:
            x = snake_block_x * self.cell_size
            y = snake_block_y * self.cell_size
            snake_block = pygame.Rect(x, y, self.cell_size - 1, self.cell_s
            pygame.draw.rect(self.display, WHITE, snake_block)

        # Draw snake's head
        x = snake_body[-1][0] * self.cell_size
        y = snake_body[-1][1] * self.cell_size
        snake_block = pygame.Rect(x, y, self.cell_size - 1, self.cell_size
        pygame.draw.rect(self.display, GREEN, snake_block)

        # Draw snake's tail
        x = snake_body[0][0] * self.cell_size
        y = snake_body[0][1] * self.cell_size
        snake_block = pygame.Rect(x, y, self.cell_size - 1, self.cell_size
        pygame.draw.rect(self.display, BLUE, snake_block)

    def draw_apple(self, apple_location):
        apple_x, apple_y = apple_location
        apple_block = pygame.Rect(apple_x * self.cell_size, apple_y * self.
        pygame.draw.rect(self.display, RED, apple_block)

    def draw_panel(self):
        for x in range(0, self.window_width, self.cell_size):  # draw verti
            pygame.draw.line(self.display, DARKGRAY, (x, 0), (x, self.windo
        for y in range(0, self.window_height, self.cell_size):  # draw hori
            pygame.draw.line(self.display, DARKGRAY, (0, y), (self.window_w


if __name__ == '__main__':
    SnakeGame().launch()
```

```
/var/folders/m1/4lxnd66915dm01hy7zh942c0000gn/T/ipykernel_4320/182599293
1.py:65: DeprecationWarning: Sampling from a set deprecated
since Python 3.9 and will be removed in a subsequent version.
  location = random.sample(available_positions, 1)[0] if available_positi
ons else (-1, -1)
```

```
(5, 4)
(5, 5)
(5, 6)
(4, 6)
```

In [ ]:

In [ ]: