Project Aim

The aim of this paper is to demonstrate how reinforcement learning can be used to train an agent to play and learn an optimal strategy for the game of snake. The reinforcement algorithm used in the application is Q-Learning. Moreover, the paper would highlight any shortcomings in the application and how improvements can be made.

Nature Of the Problem

Snake is a game, in which the aim of the agent is the eat the food pallets, while trying not to collide with any obstacles and itself. After eating a single food pallet, the length of the snake grows by 1 unit and a new food pallet spawn in a random location. The end goal is to try to eat as many food pallets as possible, which reflects the score of the player.

Why Q-Learning

Q-learninig was implemented in this scenario because of the stochastic nature of the game: the food pallet randomly appreas, meaning each game is differnt than the previous one. For this reason a model-free reinforcement algorithm was preferred as it would have been extremly hard to construct a mathematical model that would accuratley represent the stocasticity of the environmnet: the large search space would mean that a huge model would be needed. In a model-free approach, we let the agent explore the environmnet and develop numerous policies, which it can succesfully implement to reach its goal.

Another reason for applying this algorithm was the nature of the game, as it is similar to what Q-learning is about; reward the agent when he sucessfully complets a task or punish it otherwise. This commonality allowed for an easy implementation of a reward system.

Problem Analysis

Initial State:
The initial state of the snake would be to spawn at a pre-defined location with a length of 1 units (the least it can be) and try to find food pallets.

Goal State: The goal state would be to eat as many food pallets as possible and grow as large as possible, ideally filling the whole screen, after which the game ends because there is no space left for a food pallet to be placed.

Obstacles: In higher levels, when the length of the snake is large, one of the main obstacles would be to try not to surround its own body and block any exits, meaning it should always leave a path to exit.

Constraints: The snake cannot go through its own body or collide with the walls of the board, as in that case it would die.

Solution Specification

As mentioned above Q-learning was used to play the game of snake. There are two components to Q-learning, the Q-table and the Bellman equation.

Q-table

The Q-table is the data structure that is used to store the Q-values, which are the action-value function, also called Q-values, for each possible action in each possible state. The action-value function is a measure of reward that the action will produce in that particular state, for the agent. Thus the agent would prefer to take actions that produce a higher reward. For example, if the food pallet is infront of the snake, if instead of going straight it turns left, it would get a lower reward (punished for the action), but if it continues to move straight it would recieve a higher reward. After the agent plays numerous games, it updates its Q-values as it learn which actions are prefereable through trial and error. A robust Q-table is able to capture all the possible states that the agent can be in, and all the possible actions that it can take in that particular state. So when in the future the agent reaches that particular configuration, it knows which action would reap the highest reward. In my implementation the Q-table captures the following information:

- Horizontal Orientation of the snake with respect to the food pallet
- Vertical Orientation of the snake with respect to the food pallet
- Positioning of boundary walls and the snakes body with respect to its head

All the information in the Q-table is stored as numbers. As the state of my game is a 400 * 400 pixel box, where the size of the food pallet is 20 pixels, this means that there are 400 uniques states that the food pallet can be in. If we consider the moves that a snake can make in each orientation, which is 4, there would be 1600 unique action-state pairs. This right here is a pro and con of Q-learnning. In some cases the state space of the system is extremly large, and trying to capture it all is a very exhaustive process: numerous runs have to be made in order to capture the whole system and the long term action-reward values associated with each state. But after doing so, we have a robust agent that is able to identify the optimal action to take in each state.

Bellman Equation

The Q-values are updated using the Bellman equation, which is a mathematical formula that calculated the expected long-term reward for taking a particular action in a particular state. It takes into account the immediate reward recieved after taking an action and the expected reward of the subsequent stages that the agent would go to. The equation has the following form:

$$Q(s, a) = r + \gamma * max(Q(s', a'))$$

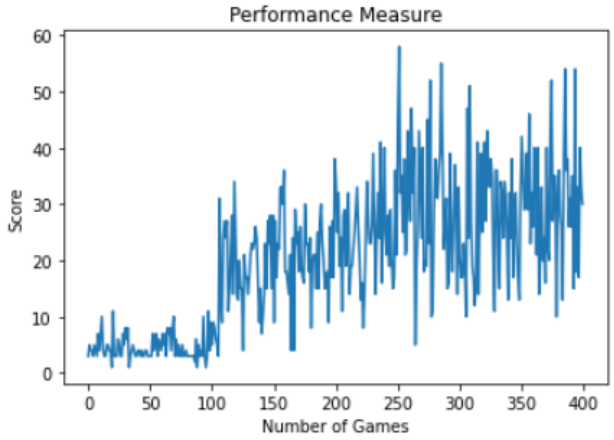
$Q(s, a)$ represents the expected reward for taking an action a in state s. r represents the immediate reward that the agent recieves after taking an action in the particular state. It is also called the learning rate, meaning a higher value would allow for a quicker change in Q-values. γ is the discount factor which described the importance of future rewards; a higher discount factor would mean that future rewards are being prefered. Finally $max(Q(s', a'))$ represents the maximum expected reward in the long run, that the agent can get for any action taken in the next state. By changing the learning and reward parameters, we are able to decide how the agent would be trained.

For my implementation I use a learning rate of 0.7 and a discount rate of 0.5. I tried varying values but the most optimal results were produced using these values. Moreover, after 200 iterations, I also reduced the epsilon values, which is the randomization factor that allows the agent to perform a random action so that it is able to explore the space further, from 0.01 to 0, meaning that I now wanted my agent to take optimal actions rather than random one.

Initially all the values in the Q-table are zero, meaning that the snake doesnt know what to do. After taking random actions, it is able to realize that moving away from the food pallet is not recommended and moving towards it is more beneficial. In the scenario where it eats the pallet, it realizes the reward associated with that action in that particular state is the highest. It now updates its q-values using the response that it just recieved. So now whenever the agent would be in that particular state it would chose the action that would lead it to the food pallet, which is realized from its previous experience. Every now and then, the agent takes random actions to explore the environmnet and prevent the agent from being trapped in a local maxima: there can be a more effecient method to reach the goal state. This process keeps going, as long as one wants, and all the q-values for each state-action pair are being updated using a feedback response. Over a large number of iterations, the agent would be able to reach a state where the Q-table represnt the optimal policy for each state.

Analysis of Solution

The agent would be judged on the number of food pallets that it was able to eat before dying, and the trend that it would pose after a given number of games. The plot below shows how for the first 100 games the maximum score that the agent could get was almost 10, but as the number of games increased the agent was able to learn from its experience and get higher and higher scores: shown by the upward trend of the plot.



Parameters used: Learning rate = 0.7 Discount Rate = 0.5

It can be seen that over time the agent is able to make better decision, which is clearly refected by the trend of increasing scores in latter games. Initially it is in the exploration phase, where it tries to explore the environment and learn which actions in each state are more favourable. Over time it is able to update its Q-table and get closer to an optimal policy. We can see that even though the number of iterations increases the agent still sometimes dies with a low score. We can attribute this to the fact that the search space is extremly large, thus the agent might be in a state that it hasnt explored before so it is unable to choose the optimal action.

Limitations

One of the things to notice here is the limitation of the agent in higher and more complicated levels: when the length of the snake is considerably larger. Even though the snake has learned that taking actions that lead to eating the food pallet are more favourable, it still hasnt figured out scenarios where doing so would be ineffecient. For example the food pallet that the snake is currently going after, might be in an area which has no exit: if the agent eats the pallet there would be no way for it to escape, if may eat itself or collide with the wall. The agent might be able to learn this policy if we train it enough times.

Imrpovements

One of the improvements that can be made would be to change the Q-table, allowing it to capture a larger picture of the environment. It can include all the possible states that the snake can be in, and in every state all the positions the food pallet can be in. This is different from our current application because now we would also want to account for all possible lengths of the snake: we said all the possible states!! Even though this would capture the whole state space, it would be computationally exhausting: the Q-table would be extremly large and the state space itself would be just too huge to effectivly explore.

To fully optimize the game we can use A* search using the Hamiltonian cycle. This would be now a search problem rather than a reinforcement learning one, but the aim here is to get the maximum number of scores so we can go ahead with this. In this implementation, the snake would try to find the shortest possible route to the food pallet but it would also consider if there is a path for it to escape. This is where Hamiltonian cycles come into play. The Hamiltonian cycle is path in a graph such that each node is exactly visited one. This would mean that we can divide the whole board into a Hamiltonian cycle, and the snake would only take the shortest route to the food if it is able to establish that an exit path would be possible. If not, then it would disregard the current route and try to find a different one. This would allow the snake to not ever be in a position where it is boxed in. A disadvantage of such an approach might be the time that the algorithm takes to run because in most scenarios instead of taking the shortest route it would opt for a larger on, but in the long-run this is a beneficial strategy as it would ensure that the snake is being kept alive.

LOs to be Grades:

#Aicoding

#Aiconcepts

#Search: Only to be graded if the application is deemed good.

Appendix

In [28]:
```
#Importing necessary dependencies
import itertools
import json
import random
import dataclasses
import pygame
import matplotlib.pyplot as plt
# Initializing the Q-tables. It doesnt contain all the states yet, they will be added when they are explored.
sequence = [''.join(s) for s in list(itertools.product(*[['0','1']] * 4))]
widths = ['0','1','NA']
heights = ['2','3','NA']

qtable = {}
for i in widths:
    for j in heights:
        for k in sqs:
            qtable[str((i,j,k))] = [0,0,0,0]

with open("qtable.json", "w") as f:
    json.dump(states, f)
```

In [38]:

```
@dataclasses.dataclass
class GameState:
    distance: tuple
    position: tuple
    surroundings: str
    food: tuple

# The class represents the main brain of the agent
class Brain(object):
    def __init__(self, width, height, block_size):

        # Represent the pararmeters for the game board
        self.width = width
        self.height = height
        self.block_size = block_size

        # Represent the learning parameters
        self.epsilon = 0.1
        self.learning_rate = 0.7
        self.discount = 0.5

        # Contains the log of the State/Action pairs and values
        self.qvalues = self.loadqvalues()
        self.log = []


        # Actions that the agent can take
        self.actions = {
            0:'left',
            1:'right',
            2:'up',
            3:'down'
        }

    #Resets the log to an empty list
    def reset(self):
        self.log = []

    #Loads the q-values
    def loadqvalues(self, path="qvalues.json"):
        with open(path, "r") as f:
            qvalues = json.load(f)
        return qvalues

    #This function decides how the agent should act in a given state and also remebers them.
    def act(self, snake, food):
        state = self.getstate(snake, food)

        # Chooses a random action
        rand = random.uniform(0,1)
        if rand < self.epsilon:
            action_key = random.choices(list(self.actions.keys()))[0]
        else:
            state_scores = self.qvalues[self.getstateStr(state)]
            action_key = state_scores.index(max(state_scores))
        action_val = self.actions[action_key]
```

```
        # Remember the actions it took at each state
        self.log.append({
            'state': state,
            'action': action_key
            })
        return action_val
    #Using the Bellman Equation, updates the q-values
    def updateqvalues(self, reason):
        log = self.log[::-1]
        for i, h in enumerate(log[:-1]):
            #If the snake ate itself then we would reward it negatively.
            #The bellman equation doesnt contain the future reward as the game is over
            if reason == 'Tail':
                _state = log[0]['state']
                _action = log[0]['action']
                state_str = self.getstateStr(_state)
                reward = -5
                self.qvalues[state_str][_action] = (1-self.learning_rate) * self.qvalues[state_str][_action] +self.learning_rate * reward
                reason = None
            #If the snake gets killed by going off screen
            #The bellman equation doesnt contain the future reward as the game is over
            elif reason == 'Screen':
                _state = log[0]['state']
                _action = log[0]['action']
                state_str = self.getstateStr(_state)
                reward = -1
                self.qvalues[state_str][_action] = (1-self.learning_rate) * self.qvalues[state_str][_action] + self.learning_rate * reward
                reason = None
            #Case where the snake survives
            else:
                # Current state of the snake
                s1 = h['state']

                #Previous state of the snake
                s0 = log[i+1]['state']
                #Action taken at previous state
                a0 = log[i+1]['action']

                #Location in current state
                x1 = s0.distance[0]
                y1 = s0.distance[1]
                #Location in previous state
                x2 = s1.distance[0]
                y2 = s1.distance[1]

                # Case where the snake would eat the food
                if s0.food != s1.food:
                    reward = 10
                #Case where the snake would move closer to the food pallet instead of eating it
                elif (abs(x1) > abs(x2) or abs(y1) > abs(y2)): # Snake is closer to the food, positive reward
                    reward = 1
                #Case where the snake starts to move away from the food pallet
                else:
                    reward = -1
                #The q-values would be updated accordingly
                state_str = self.getstateStr(s0)
                new_state_str = self.getstateStr(s1)
                # Bellman equation
                self.qvalues[state_str][a0]=(1-self.learning_rate)*(self.qvalues[state_str][a0])+self.learning_rate*(reward+self.discount*max(self.qvalues[new_st
    #This function helps in defining the positiong of the snake in the environmnet
```

```
def getstate(self, snake, food):
    head = snake[-1]
    #Distance described with respect to the food pallet
    dist_x = food[0] - head[0]
    dist_y = food[1] - head[1]

    if dist_x > 0:
        #Food it to right of the snake
        pos_x = '1'
    elif dist_x < 0:
        #Food it to left of the snak
        pos_x = '0'
    else:
        #Snake is in the same direction of the food
        pos_x = 'NA'

    if dist_y > 0:
        #Food is below snake
        pos_y = '3'
    elif dist_y < 0:
        #Food is above the snake
        pos_y = '2'
    else:
        #Snake is in the same direction of the food
        pos_y = 'NA'
    #Sequence of positions of the snake
    sequences = [
        (head[0]-self.block_size, head[1]),
        (head[0]+self.block_size, head[1]),
        (head[0],head[1]-self.block_size),
        (head[0],head[1]+self.block_size),
    ]

    surrounding_list = []
    #Describes the environment in relation to the snake. Like where is the screen and where is its tail.
    #Value of 1 would be given for obstacles, and 0 in case nothing dangerous is there.
    for sequence in sequences:
        if sequence[0] < 0 or sequence[1] < 0:
            surrounding_list.append('1')
        elif sequence[0] >= self.width or sequence[1] >= self.height:
            surrounding_list.append('1')
        elif sequence in snake[:-1]:
            surrounding_list.append('1')
        else:
            surrounding_list.append('0')
    surroundings = ''.join(surrounding_list)

    return GameState((dist_x, dist_y), (pos_x, pos_y), surroundings, food)

def getstateStr(self, state):
    return str((state.position[0],state.position[1],state.surroundings))
```

In [ ]:
```
#Game code adopted from the following website: https://www.edureka.co/blog/snake-game-with-pygame/

#Represents the game environmnet


pygame.init()

#used the make the game environmnet
YELLOW = (255, 255, 102)
BLACK = (0, 0, 0)
GREEN = (0, 255, 0)
BLUE = (50, 153, 213)

BLOCK_SIZE = 20
DIS_WIDTH = 400
DIS_HEIGHT = 400

FRAMESPEED = 99999



def GameLoop():
    global dis

    dis = pygame.display.set_mode((DIS_WIDTH, DIS_HEIGHT))
    pygame.display.set_caption('Snake')
    clock = pygame.time.Clock()

    # Starting position of snake
    x1 = DIS_WIDTH / 2
    y1 = DIS_HEIGHT / 2
    x1_change = 0
    y1_change = 0
    snake_list = [(x1,y1)]
    length_of_snake = 1

    # Create first food pallet at a random location
    food_posx = round(random.randrange(0, DIS_WIDTH - BLOCK_SIZE) / 20.0) * 20.0
    food_posy = round(random.randrange(0, DIS_HEIGHT - BLOCK_SIZE) / 20.0) * 20.0

    dead = False
    reason = None
    while not dead:
        # Get valid actions and implement them
        action = brain.act(snake_list, (food_posx,food_posy))
        if action == "left":
            x1_change = -BLOCK_SIZE
            y1_change = 0
        elif action == "right":
            x1_change = BLOCK_SIZE
            y1_change = 0
        elif action == "up":
            y1_change = -BLOCK_SIZE
            x1_change = 0
        elif action == "down":
            y1_change = BLOCK_SIZE
            x1_change = 0

        # Move the snake
```

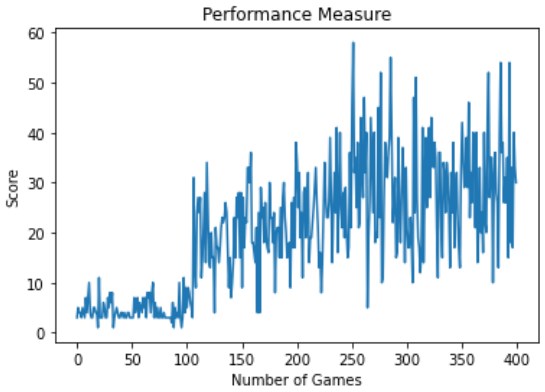
```
        x1 += x1_change
        y1 += y1_change
        head = (x1,y1)
        snake_list.append(head)

        # Check if snake is off screen, in which case it would be dead
        if x1 >= DIS_WIDTH or x1 < 0 or y1 >= DIS_HEIGHT or y1 < 0:
            reason = 'Screen'
            dead = True

        # Check if snake hit tail, in which case it would be dead

        if len(snake_list) >4:
            if head in snake_list[:-1]:
                reason = 'Tail'
                dead = True

        # Check if snake ate food. If it did then generate a different food pallet
        if x1 == food_posx and y1 == food_posy:
            food_posx = round(random.randrange(0, DIS_WIDTH - BLOCK_SIZE) / 20.0) * 20.0
            food_posy = round(random.randrange(0, DIS_HEIGHT - BLOCK_SIZE) / 20.0) * 20.0
            length_of_snake += 1

        # Delete the last cell since we just added a head for moving, unless we ate a food
        if len(snake_list) > length_of_snake:
            del snake_list[0]

        # Draw food, snake and update score
        dis.fill(BLUE)
        DrawFood(food_posx, food_posy)
        DrawSnake(snake_list)
        DrawScore(length_of_snake - 1)
        pygame.display.update()

        # Update Q Table
        brain.updateqvalues(reason)

        # Next Frame
        clock.tick(FRAMESPEED)

    return length_of_snake - 1, reason

#Function that draws the food pallets
def DrawFood(food_posx, food_posy):
    pygame.draw.rect(dis, GREEN, [food_posx, food_posy, BLOCK_SIZE, BLOCK_SIZE])

#Function that shows the score
def DrawScore(score):
    font = pygame.font.SysFont("comicsansms", 35)
    value = font.render(f"Score: {score}", True, YELLOW)
    dis.blit(value, [0, 0])

#Renders the snake
def DrawSnake(snake_list):
    head = snake_list[0]
    for x in snake_list:
        #This allows us to have one different block to differentiate which part is the head
        if x == head:
            pygame.draw.rect(dis, YELLOW, [x[0], x[1], BLOCK_SIZE, BLOCK_SIZE])
        else:
```

```
            pygame.draw.rect(dis, BLACK, [x[0], x[1], BLOCK_SIZE, BLOCK_SIZE])



game_count = 0

brain = Brain(DIS_WIDTH, DIS_HEIGHT, BLOCK_SIZE)
results = []

while game_count <400:

    brain.reset()
    #We stop it from making random moves
    if game_count > 200:
        brain.epsilon = 0
    else:
        brain.epsilon = 0.1
    score, reason = GameLoop()
    results.append(score)
    #Outputs the game number along with the score and reason of death
    print(f"Games number: {game_count}; Final Score: {score}; Died because of: {reason}")
    game_count += 1

pygame.quit()
```

Plots

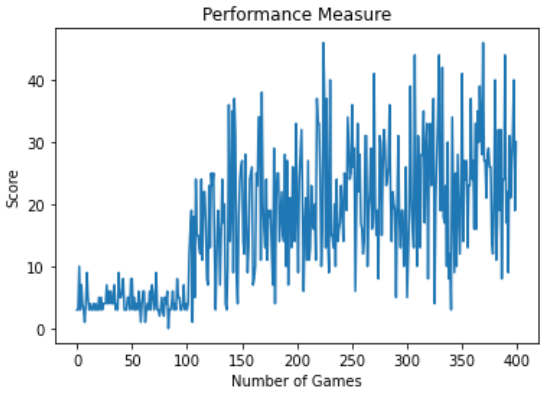
In [34]:

```
plt.plot(results)
plt.xlabel("Number of Games")
plt.ylabel("Score")
plt.title('Performance Measure')
plt.show()
print('Parameters used: Learning rate = 0.7  Discount Rate = 0.5')
```



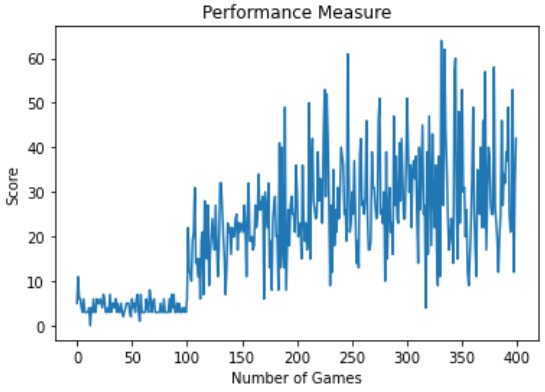

```
Parameters used: Learning rate = 0.7  Discount Rate = 0.5
```

In [37]:
```
plt.plot(results)
plt.xlabel("Number of Games")
plt.ylabel("Score")
plt.title('Performance Measure')
plt.show()
print('Parameters used: Learning rate = 0.1  Discount Rate = 0.5')
```




```
Parameters used: Learning rate = 0.1  Discount Rate = 0.5
```

In [40]:
```
plt.plot(results)
plt.xlabel("Number of Games")
plt.ylabel("Score")
plt.title('Performance Measure')
plt.show()
print('Parameters used: Learning rate = 1  Discount Rate = 0.5')
```




```
Parameters used: Learning rate = 1  Discount Rate = 0.5
```

In [ ]: