```
import numpy as np
import helper
import random
from board import Snake

#   This class has all the functions and variables necessary to implement snake 
game
#   We will be using Q learning to do this

class SnakeAgent:

    #   This is the constructor for the SnakeAgent class
    #   It initializes the actions that can be made,
    #   Ne which is a parameter helpful to perform exploration before deciding next 
action,
    #   LPC which ia parameter helpful in calculating learning rate (lr)
    #   gamma which is another parameter helpful in calculating next move, in other 
words
    #            gamma is used to blalance immediate and future reward
    #   Q is the q-table used in Q-learning
    #   N is the next state used to explore possible moves and decide the best one 
before updating
    #           the q-table
    def __init__(self, actions, Ne, LPC, gamma):
        self.actions = actions
        self.Ne = Ne
        self.LPC = LPC
        self.gamma = gamma
        self.reset()

        # Create the Q and N Table to work with
        self.Q = helper.initialize_q_as_zeros()
        self.N = helper.initialize_q_as_zeros()


    #   This function sets if the program is in training mode or testing mode.
    def set_train(self):
        self._train = True

     #   This function sets if the program is in training mode or testing mode.
    def set_eval(self):
        self._train = False

    #   Calls the helper function to save the q-table after training
    def save_model(self):
        helper.save(self.Q)

    #   Calls the helper function to load the q-table when testing
    def load_model(self):
        self.Q = helper.load()

    #   resets the game state
    def reset(self):
        #self.Q = helper.initialize_q_as_zeros()
        self.N = helper.initialize_q_as_zeros()

        self.points = 0
        self.s = None
        self.a = None
```

```
#   This is a function you should write.
#   Function Helper:IT gets the current state, and based on the
#   current snake head location, body and food location,
#   determines which move(s) it can make by also using the
#   board variables to see if its near a wall or if  the
#   moves it can make lead it into the snake body and so on.
#   This can return a list of variables that help you keep track of
#   conditions mentioned above.
def helper_func(self, state):
    #print("IN helper_func")
    """ Get possible moves given state  """
    actions = [a for a in self.actions]
    current_snake_head_x = state[0]
    current_snake_head_y = state[1]
    snake_body = state[2]
    food_x = state[3]
    food_y = state[4]
    possible_actions = []
    boundaries_dict = {"wall": [0, 0, 0, 0], "body": [0, 0, 0, 0]}

    # YOUR CODE HERE
    # YOUR CODE HERE
    # YOUR CODE HERE
    # YOUR CODE HERE
    # YOUR CODE HERE

    snake = Snake(current_snake_head_x, current_snake_head_y, food_x, food_y)
    for action in actions:
        result = snake.move(action)
        if not(result): # snake lived on action
            possible_actions.append(action)
        else:
            key = "wall" if snake.did_hit_wall else "body"
            boundaries_dict[key][action] = 1
        snake.reset()
    return possible_actions, boundaries_dict



# Computing the reward, need not be changed.
def compute_reward(self, points, dead):
    if dead:
        return -1
    elif points > self.points:
        return 2
    else:
        return -0.1

def exploration_transormation(self, q_value_list, n_value_list):
    return q_value_list + self.Ne / (n_value_list + 1.0)
    #return q_value_list



#   This is the code you need to write.
#   This is the reinforcement learning agent
#   use the helper_func you need to write above to
#   decide which move is the best move that the snake needs to make
```

```
    #   using the compute reward function defined above.
    #   This function also keeps track of the fact that we are in
    #   training state or testing state so that it can decide if it needs
    #   to update the Q variable. It can use the N variable to test outcomes
    #   of possible moves it can make.
    #   the LPC variable can be used to determine the learning rate (lr), but if
    #   you're stuck on how to do this, just use a learning rate of 0.7 first,
    #   get your code to work then work on this.
    #   gamma is another useful parameter to determine the learning rate.
    #   based on the lr, reward, and gamma values you can update the q-table.
    #   If you're not in training mode, use the q-table loaded (already done)
    #   to make moves based on that.
    #   the only thing this function should return is the best action to take
    #   ie. (0 or 1 or 2 or 3) respectively.
    #   The parameters defined should be enough. If you want to describe more
elaborate
    #   states as mentioned in helper_func, use the state variable to contain all
that.
    def agent_action(self, state, points, dead):

        #print("IN AGENT_ACTION")

        # YOUR CODE HERE
        # YOUR CODE HERE
        # YOUR CODE HERE
        # YOUR CODE HERE
        # YOUR CODE HERE
        # YOUR CODE HERE
        # YOUR CODE HERE

        possible_actions, boundaries_dict = self.helper_func(state)


        LEFT = 2
        RIGHT = 3
        TOP = 1
        BOTTOM = 0

        def get_position(value_one, value_two):
            if value_one == 1:
                return 0
            elif value_two == 1:
                return 2
            else:
                return 1

        def get_state_features(state, possible_actions, boundaries_dict):
            current_snake_body = state[2]
            current_food_x = state[3]
            current_food_y = state[4]
            current_snake_head_x = state[0]
            current_snake_head_y = state[1]

            # walls
            is_left_wall = boundaries_dict["wall"][LEFT]
            is_right_wall = boundaries_dict["wall"][RIGHT]
            is_top_wall = boundaries_dict["wall"][TOP]
            is_bottom_wall = boundaries_dict["wall"][BOTTOM]
```

```
            #food
            is_food_left = 1 if current_food_y == current_snake_head_y and
current_snake_head_x > current_food_x else 0
            is_food_right = 1 if current_food_y == current_snake_head_y and
current_snake_head_x < current_food_x else 0
            is_food_top = 1 if current_food_x == current_snake_head_x and
current_snake_head_y < current_food_y else 0
            is_food_bottom = 1 if current_food_x == current_snake_head_x and
current_snake_head_y > current_food_y else 0

            # body
            is_left_body = boundaries_dict["body"][LEFT]
            is_right_body = boundaries_dict["body"][RIGHT]
            is_top_body = boundaries_dict["body"][TOP]
            is_bottom_body = boundaries_dict["body"][BOTTOM]

            return (is_left_wall, is_right_wall, is_top_wall, is_bottom_wall,
is_food_left, is_food_right, is_food_top, is_food_bottom, is_left_body,
is_right_body, is_top_body, is_bottom_body)


        if self._train:
            # update q-values
            for action in possible_actions:


                # if no action found, then if new position in snake body no wall
else wall
                is_left_wall, is_right_wall, is_top_wall, is_bottom_wall,
is_food_left, is_food_right, is_food_top, is_food_bottom, is_left_body,
is_right_body, is_top_body, is_bottom_body = get_state_features(state,
possible_actions, boundaries_dict)

                # print(get_state_features(state, possible_actions,
boundaries_dict))
                current_q_value = self.Q[get_position(is_left_wall, is_right_wall),
get_position(is_top_wall, is_bottom_wall), get_position(is_food_left,
is_food_right), get_position(is_food_top, is_food_bottom), is_top_body,
is_bottom_body, is_left_body, is_right_body, action]
                #print("Current ", current_q_value)

                snake = Snake(state[0], state[1], state[3], state[4])
                new_state, new_points, new_is_dead = snake.step(action)
                new_possible_actions, new_boundaries_dict =
self.helper_func(new_state)


                # if no action found, then if new position in snake body no wall
else wall
                is_new_left_wall, is_new_right_wall, is_new_top_wall,
is_new_bottom_wall, is_new_food_left, is_new_food_right, is_new_food_top,
is_new_food_bottom, is_new_left_body, is_new_right_body, is_new_top_body,
is_new_bottom_body = get_state_features(new_state, new_possible_actions,
new_boundaries_dict)


                new_max_state_q_value =
np.max(self.exploration_transormation(self.Q[get_position(is_new_left_wall,
is_new_right_wall), get_position(is_new_top_wall, is_new_bottom_wall),
```

```
get_position(is_new_food_left, is_new_food_right), get_position(is_new_food_top,
is_new_food_bottom), is_new_top_body, is_new_bottom_body, is_new_left_body,
is_new_right_body, :], self.N[get_position(is_new_left_wall, is_new_right_wall),
get_position(is_new_top_wall, is_new_bottom_wall), get_position(is_new_food_left,
is_new_food_right), get_position(is_new_food_top, is_new_food_bottom),
is_new_top_body, is_new_bottom_body, is_new_left_body, is_new_right_body, :]))

                updated_q_value = current_q_value + self.LPC *
(self.compute_reward(new_points + points, new_is_dead) + self.gamma *
new_max_state_q_value - current_q_value)


                self.Q[get_position(is_left_wall, is_right_wall),
get_position(is_top_wall, is_bottom_wall), get_position(is_food_left,
is_food_right), get_position(is_food_top, is_food_bottom), is_top_body,
is_bottom_body, is_left_body, is_right_body, action] = updated_q_value

                # update visited count

                self.N[get_position(is_left_wall, is_right_wall),
get_position(is_top_wall, is_bottom_wall), get_position(is_food_left,
is_food_right), get_position(is_food_top, is_food_bottom), is_top_body,
is_bottom_body, is_left_body, is_right_body, action] += 1



        # Inference

        is_left_wall, is_right_wall, is_top_wall, is_bottom_wall, is_food_left,
is_food_right, is_food_top, is_food_bottom, is_left_body, is_right_body,
is_top_body, is_bottom_body = get_state_features(state, possible_actions,
boundaries_dict)


        action = np.argmax(self.Q[get_position(is_left_wall, is_right_wall),
get_position(is_top_wall, is_bottom_wall), get_position(is_food_left,
is_food_right), get_position(is_food_top, is_food_bottom), is_top_body,
is_bottom_body, is_left_body, is_right_body, :])


        return action
```