Sipna College of Engineering & Technology, Amravati.
Department of Computer Science & Engineering
Session 2022-2023

Branch :- Computer Sci. & Engg. **Class :- Final Year**
Subject :-Artificial Intelligence and Machine Learning **Sem :- VIII**

Teacher Manual

PRACTICAL NO 4

AIM: To Understand and implement the concept of reinforcement learning

S/W REQUIRED: Python
Source - Keras.io

Reinforcement Learning:-

Reinforcement Learning is the science of making optimal decisions using experience. It is categorized with supervised learning and unsupervised learning, rather than categorizing it with Machine Learning and Deep Learning. Reinforcement learning is the methodology that deals with the interaction between agent and environment through actions and has got nothing to do with labeled and unlabeled data, although there is another category called **Semi-supervised learning**, which is indeed a hybrid of supervised and unsupervised learning.

The word "reinforce" means strengthen or support (an object or substance), especially with additional material.
But what we are strengthening here, and what is our support which strengthen?
We are trying to strengthen the learning ability of an **agent** to understand the environment. But that also happens in machine learning and deep learning, where the model is trained and the model learns a pattern from the trained data while minimizing the loss and improving the accuracy. The factor that strengthens the learning ability in reinforcement learning is **Reward**. A high positive reward is awarded to the agent for making a correct decision, and the agent should be penalized for making a wrong decision. The agent should get a slight negative reward for not making a correct decision after every time-step. "Slight" negative because we would prefer our agent to take more time in taking a decision rather than making the wrong decision.

What is Actor-Critic agent?

Now, before jumping into the concept of Actor-Critic agent, I would recommend you to have some basic knowledge about Q-Learning, followed by deep Q-Learning because without these two you won't understand the significance and necessity Actor-Critic agent.
In sort,
As an agent takes actions and moves through an environment, it learns to map the observed state of the environment to two possible outputs:

- Recommended action: A probabiltiy value for each action in the action space. The part of the agent responsible for this output is called the **actor**.
- Estimated rewards in the future: Sum of all rewards it expects to receive in the future. The part of the agent responsible for this output is the **critic**.

Agent and Critic learn to perform their tasks, such that the recommended actions from the actor maximize the rewards.

Implementation:

```
!pip install kaggle-environments --upgrade
```

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import os
import sys
import PIL.Image

import tensorflow as tf
import logging

from sklearn import preprocessing
import random
import matplotlib.pyplot as plt
import seaborn as sns

from kaggle_environments import evaluate, make
from kaggle_environments.envs.halite.helpers import *
```

```
seed=123
tf.compat.v1.set_random_seed(seed)
session_conf = tf.compat.v1.ConfigProto(intra_op_parallelism_threads=1, inter_op_parallelism_threads=1)
sess = tf.compat.v1.Session(graph=tf.compat.v1.get_default_graph(), config=session_conf)
tf.compat.v1.keras.backend.set_session(sess)
logging.disable(sys.maxsize)
global ship_
```

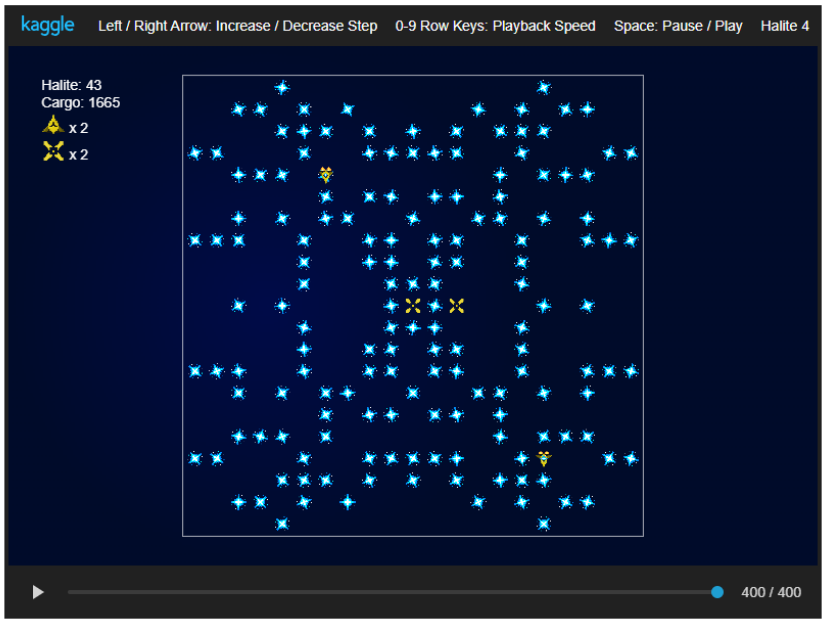
Analyzing the environment

Lets take a tour of our environment and its settings first.

In [4]:

linkcode

```
env = make("halite", debug=True)
env.run(["random"])
env.render(mode="ipython",width=800, height=600)
```

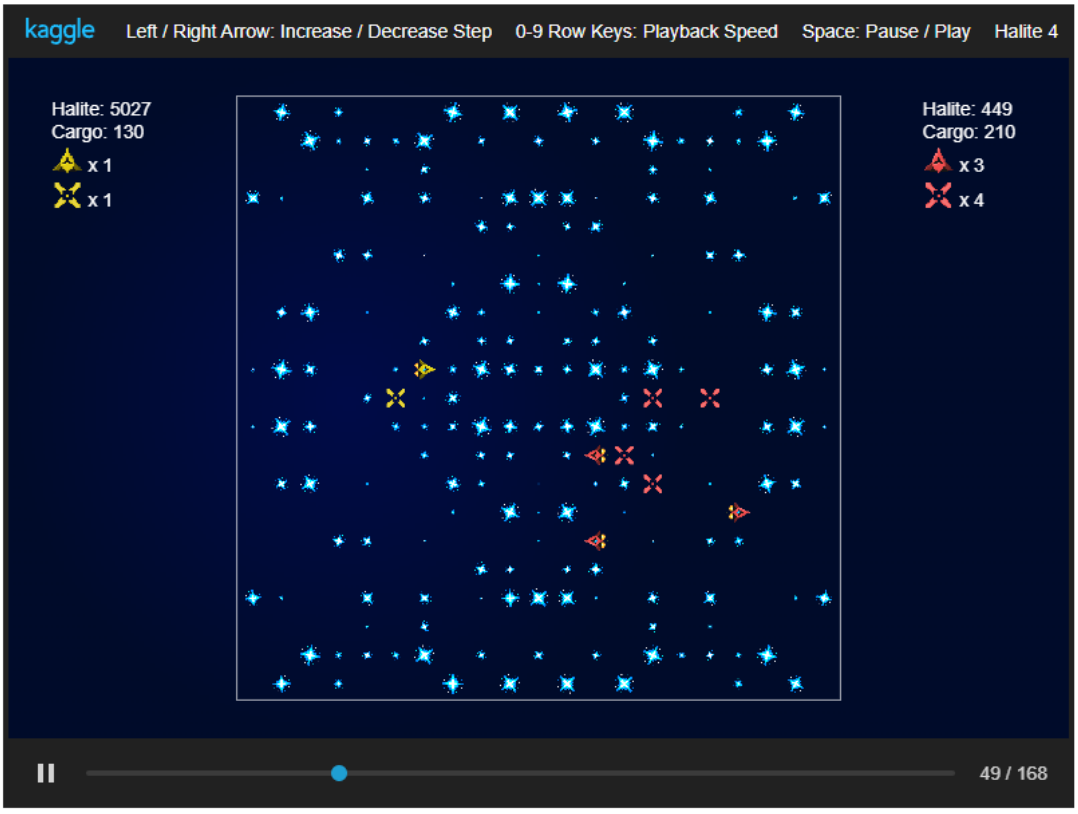


The game begins

So lets train our model with respect to random actions and see what happens...

```
def getDirTo(fromPos, toPos, size):
    fromX, fromY = divmod(fromPos[0],size), divmod(fromPos[1],size)
    toX, toY = divmod(toPos[0],size), divmod(toPos[1],size)
    if fromY < toY: return ShipAction.NORTH
    if fromY > toY: return ShipAction.SOUTH
    if fromX < toX: return ShipAction.EAST
    if fromX > toX: return ShipAction.WEST

# Directions a ship can move
directions = [ShipAction.NORTH, ShipAction.EAST, ShipAction.SOUTH, ShipAction.WEST]

# Will keep track of whether a ship is collecting halite or carrying cargo to a shipyard
ship_states = {}

# Returns the commands we send to our ships and shipyards
def simple_agent(obs, config):
    size = config.size
    board = Board(obs, config)
    me = board.current_player
    # If there are no ships, use first shipyard to spawn a ship.
    if len(me.ships) == 0 and len(me.shipyards) > 0:
        me.shipyards[0].next_action = ShipyardAction.SPAWN

    # If there are no shipyards, convert first ship into shipyard.
    if len(me.shipyards) == 0 and len(me.ships) > 0:
        me.ships[0].next_action = ShipAction.CONVERT

    for ship in me.ships:
        if ship.next_action == None:
### Part 1: Set the ship's state
            if ship.halite < 200: # If cargo is too low, collect halite
                ship_states[ship.id] = "COLLECT"
            if ship.halite > 500: # If cargo gets very big, deposit halite
                ship_states[ship.id] = "DEPOSIT"
```

```
        ### Part 2: Use the ship's state to select an action
        if ship_states[ship.id] == "COLLECT":
            # If halite at current location running low,
            # move to the adjacent square containing the most halite
            if ship.cell.halite < 100:
                neighbors = [ship.cell.north.halite, ship.cell.east.halite,
                             ship.cell.south.halite, ship.cell.west.halite]
                best = max(range(len(neighbors)), key=neighbors.__getitem__)
                ship.next_action = directions[best]
        if ship_states[ship.id] == "DEPOSIT":
            # Move towards shipyard to deposit cargo
            direction = getDirTo(ship.position, me.shipyards[0].position, size)
            if direction: ship.next_action = direction

    return me.next_actions

trainer = env.train([None, "random"])
observation = trainer.reset()
while not env.done:
    my_action = simple_agent(observation, env.configuration)
    print("My Action", my_action)
    observation = trainer.step(my_action)[0]
    print("Reward gained",observation.players[0][0])


env.render(mode="ipython",width=800, height=600)
```



CONCLUSION: Thus we have implemented the concept of reinforcement learning.

(our ship is collecting halites, transforming into shipyards and also spawning if now ship is available in the most efficient way possible. In simple words, we have successfully trained our agent to direct the ship to collect halites in the most efficient way possible. I wish if I could have found a way to track the nearest shipyard and deposits collected halites or find a way to control multiple agents through reinforcement learning. Although there are research papers which I found explaining Multi-Reinforcement learning and Multi-goal Reinforcement learning, which I think could be more useful to solve this problem. Anyways my knowledge is currently limited to Actor-Critic agent, and I would study more to find a better solution than this.)