Università di Roma
La Sapienza

Dipartimento di Ingegneria Informatica,
Automatica e Gestionale Antonio Ruberti

HRI system

Electives in Artificial Intelligence
Reasoning agents

Students:
Dat Nguyen Ngoc
Govardhan Chitrada

Professor
Giuseppe De Giacomo

June 2021

Contents

1 Introduction

The main goal is to use the Restraining bolts along with DQN Reinforcement algorithm to achieve a more robust control over the evironment. Here the snake environment is used as the platform for testing the proposed method. The infamous SNAKE game is used as the main environment and beside the goal of eating food we can add certain constraints to the states of the snake to follow. The environment consists of the two different types of food which act as the snakes food. To make the agent robust certain constraints have been introduced with the help of the reasoning agent where the snake has to continue eating in alternate sequence.

1.1 Requirements

To achieve a streamlined workflow for multiple people without the need to update one's existing system custom docker image has been built with all the required dependencies. To make it compatible with multiple vendors. Vendor specific scripts are included along with the project. The requirements are as follows.

- Docker
- Python3

1.2 Project Management

For every project related to Deep Learning must follow specific workflow design if it includes multiple people in that specific project. We followed a simple workspace implementation to deal with all the libraries and machine specific dependencies. In fig.1 the whole process has been containerized to achieve machine independent execution process.

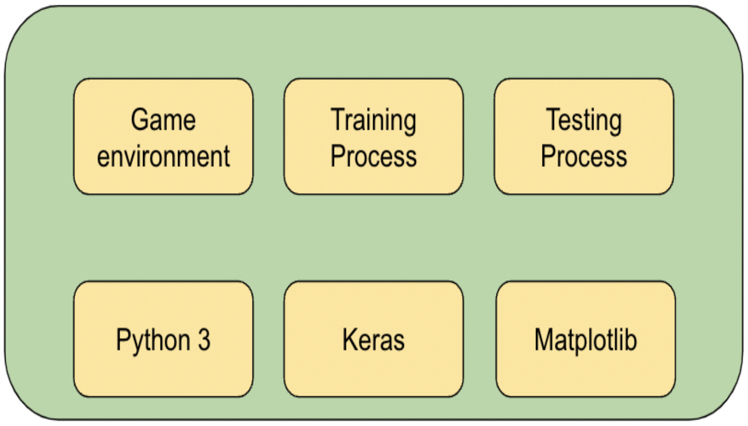

Figure 1: Docker Container

2 Retraining Bolt

Follow the main task: Snake eat meat interleave eating apple. We write the LTLf formulation for the task. Then we convert to Deterministic Finite state Automaton (DFA) using tool provide in [5]. We use the diagram to determine the reasoning agent 's states then combine them in program

$$(F(eat\_red)\&G((eat\_red- > F(eat\_green)))) \tag{1}$$

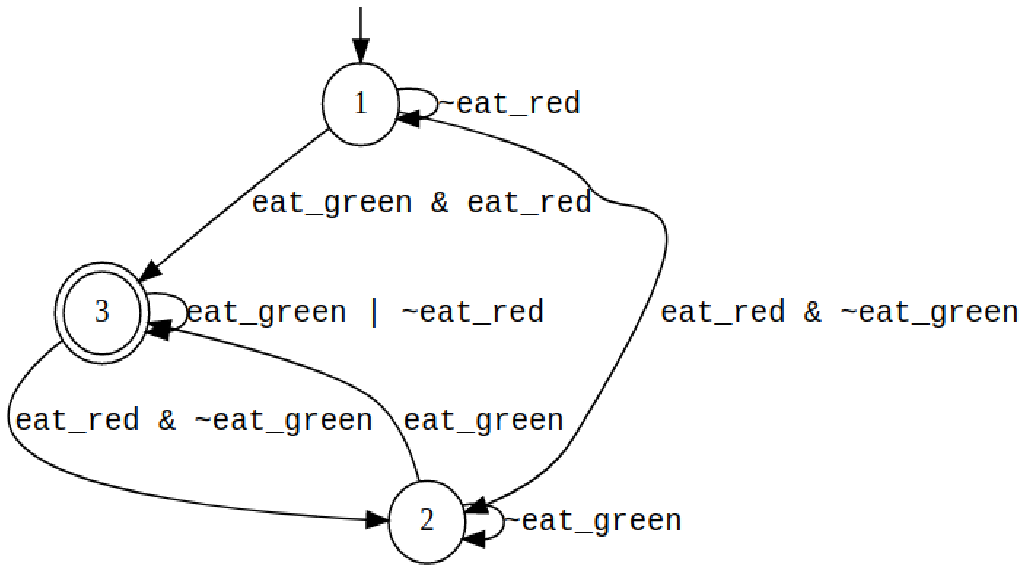

Figure 2: Every time eating red food is executed, eating green food must be executed afterwards

Following DFA content, we build the structure for learning snake agent so that combine restraining bolts in reinforcement algorithm. Rewards were provide from reasoning agent follow table 1

q1− >q2	30
q2− >q2	-30
q2− >q3	50
q3− >q2	50
q3− >q3	-30

Table 1: Restraining bolts rewards

The snake agent and reasoning agent with receive states and fluents that were prorvied by environment.

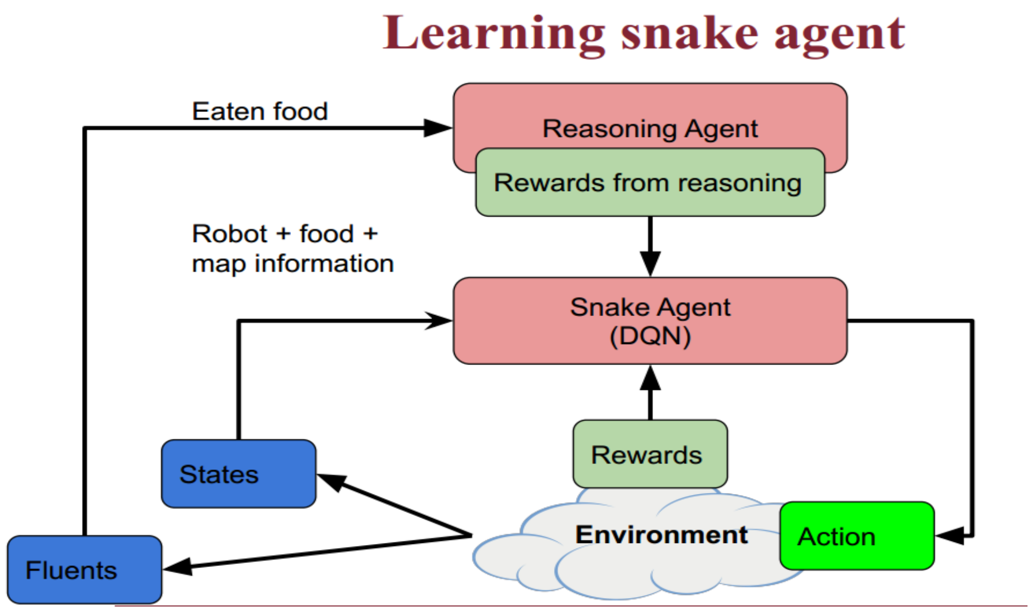

Figure 3: Learning snake agent using restraining bolts and reinforcement learning

In Figure 3, there are 2 main parts include: snake agent and reasoning agent.
Snake agent:

- Features: 18 features have information about eaten food, location of foods (meat and apple), and robot position. (detail in table 4)
- Actions: Move up,move down,move right, move left
- Reward: Negative reward if robot die (hit wall or hit tail) and moving cost per step.Positive reward when snake eat food.

Restraining Bolts

- Fluents: Type of food snake eat : “no food” ; “apple” ; “meat”
- Reward: Eating meat interleave with eating apple

3 Reinforcement Learning

Reinforcement Learning is a type of machine learning that uses a reward function to guide the agent in deciding which action to take. The agent is rewarded for taking the action that maximizes the reward function. In general, a Reinforcement learning agent is able to perceive and interpret the environment and learn from it. By adding a method of rewarding desired behaviors and punishing undesired behaviors. This programs the agent to seek long term and maximum overall reward to achieve an optimal solution.

While reinforcement learning has been a topic of much interest in the field of AI, its widespread, real-world adoption and application remain limited. Noting this, however, research papers abound on theoretical applications, and there have been some successful use cases.

Current use cases include, but are not limited to, the following:

- Reinforcement learning for autonomous driving
- Reinforcement learning for autonomous navigation
- Reinforcement learning for autonomous speech recognition
- Reinforcement learning for robotic manipulation

3.1 Deep Reinforcement Learning

Deep Learning uses artificial neural networks to map inputs to outputs. Deep Learning is powerful, because it can approximate any function with only one hidden layer. How does it work? The network exists of layers with nodes. The first layer is the input layer. Then the hidden layers transform the data with weights and activation functions. The last layer is the output layer, where the target is predicted. By adjusting the weights the network can learn patterns and improve its predictions. As the name suggests, Deep Reinforcement Learning is a combination of Deep Learning and Reinforcement Learning. By using the states as the input, values for actions as the output and the rewards for adjusting the weights in the right direction, the agent learns to predict the best action for a given state.

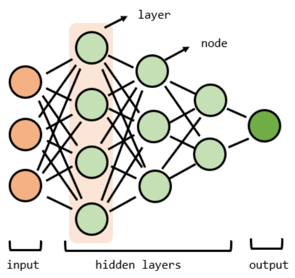

Figure 4: Simple neural network

3.2 Deep Q Network - DQN

Reinforcement learning can be sufficiently applicable to the environment where the all achievable states can be manged (iterated) and stored in standard computer RAM memory. However, the environment where the number of states overwhelms the capacity of contemporary computers (for Atari games there are 12833600 states) the standard Reinforcement Learning approach is not very applicable. Furthermore, in real environment, the Agent has to face with continuous states (not discrete), continuous variables and continuous control (action) problems.

Bearing in mind the complexity of environment the Agent has to operate in (number of states, continuous control) the standard well defined Reinforcement Learning Q — table is replaced by Deep Neural Network (Q — Network) which maps (non — linear approximation) environment states to Agent actions. Network architecture, choice of network hyper parameters and learning is performed during training phase (learning of Q — Network weight). DQN allows the Agent to explore unstructured environment and acquire knowledge which over time makes them possible for imitating human behavior.

3.3 DQN - Algorithm

The below Fig 5 shows the DQN algorithm during the training process. where Q — network proceeds as a as nonlinear approximation which maps both state into an action value. During the training process, the Agent, interacts with the environment and receives data, which is used during the learning the Q — network. The Agent explores the environment to build a complete picture of transitions and action outcomes. At the beginning the Agent decides about the actions randomly which over time becomes insufficient. While exploring the environment the Agent tries to look on Q — network (approximation) in order to decide how to act. We called this approach (combination of random behavior and according to Q — network) as an epsilon — greedy method (Epsilon -greedy action selection block), which just means changing between random and Q policy using the probability hyper parameter epsilon.

The core of presented Q-learning algorithm is derived from the supervised learning. Here as it was mention above, the goal is to approximate a complex, nonlinear function Q(S, A) with a deep neural network.

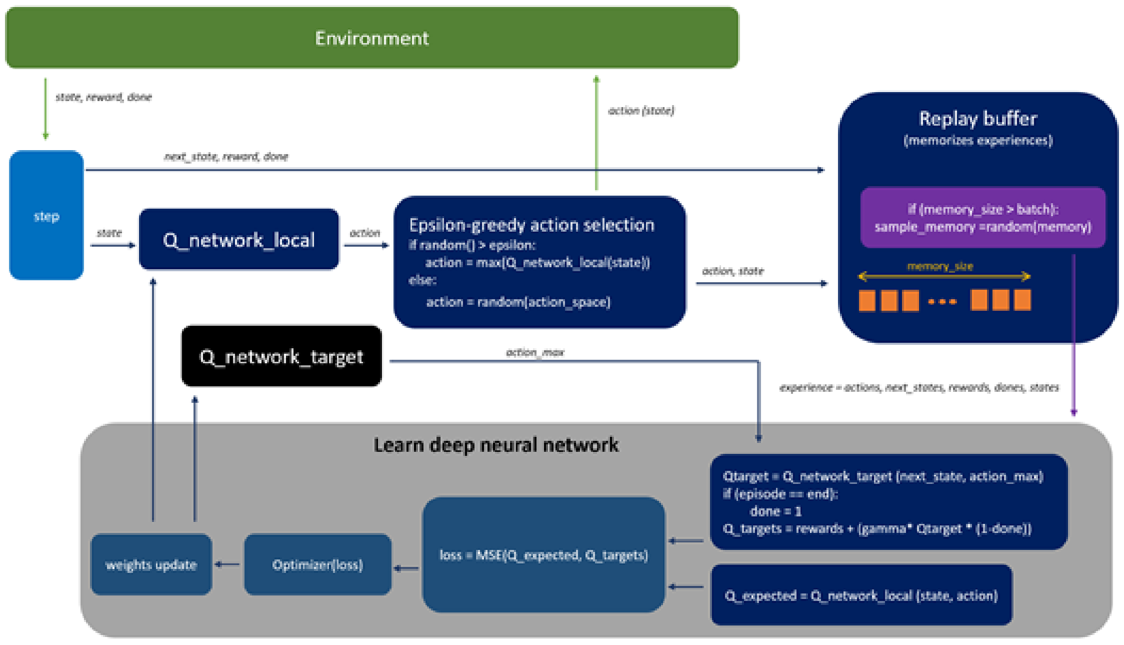
Environment
state, reward, done
action (state)
step
next_state, reward, done
state
Q_network_local
action
Epsilon-greedy action selection
if random() > epsilon:
action = max(Q_network_local(state))
else:
action = random(action_space)
action, state
Replay buffer
(memorizes experiences)
if (memory_size > batch):
sample_memory =random(memory)
memory_size
Q_network_target
action_max
experience = actions, next_states, rewards, dones, states
Learn deep neural network
Qtarget = Q_network_target (next_state, action_max)
if (episode == end):
done = 1
Q_targets = rewards + (gamma* Qtarget * (1-done))
Q_expected = Q_network_local (state, action)
loss = MSE(Q_expected, Q_targets)
Optimizer(loss)
weights update

Figure 5: DQN Algortihm

During the process we use two seperate networks Q — network and target Q — network. The Q — network is used to approximate the Q function. The target Q — network is used to approximate the target Q function. The target network is frozen for several time steps and then the target network weights are updated by copying the weights from the actual Q network. Freezing the target Q — network for a while and then updating its weights with the actual Q network weights stabilizes the training.

In short the algorithm can be described as follows:

1. Initialize the Q — network with random weights.
2. Initialize replay buffer
3. Pre-process the environment and store the initial state in the replay buffer.
4. Initialize the target Q — network with random weights.
5. For each episode:
    (a) Choose an action at random.
    (b) Take action and observe the reward and the next state.
    (c) Store the transition in the replay buffer.
    (d) Sample a random minibatch from the replay buffer.
    (e) Update the weights of the Q — network using the minibatch.
    (f) Update the weights of the target Q — network using the minibatch.
6. After the training is finished, the target Q — network weights are copied to the Q — network weights.

3.4 DQN Agent

The methods *reset(self)*, *step(self,action)*, and *get_state(self)*.. It is also necessary to calculate the reward every time the agent takes a step.

The agent learns to play snake (with expreience replay) and to avoid the obstacles. The following four state spaces are used:

Param name	optimized values
epsilon	0.9
epsilon_min	0.01
gamma	0.95
batch_size	500
learning_rate	0.00025
layer_sizes	[128,128,128]

Table 2: Optimized Hyper-parameters for **DQN**.

- *observation* - the state of the environment.
- *action* - the action taken by the agent.
- *reward* - the reward received by the agent.
- *next_observation* - the next state of the environment.

4 Environment - Snake

In this specific project the game ”Snake” is modified to fit certain requirements. The game is played on a grid of size 20x20. The snake is controlled by the agent. The agent is rewarded for eating food and for avoiding the walls and itself. The agent is penalized for running into itself. To this we also added two different food types such as meat and apple. In general, to stay healthy in real life people should balance their diet with fruits and meat. In the same way here the snake has to eat maximum amount of food and in the same way it has to alternate the food (We call it as pairs).

The rewards for eating the food are assigned by the Reinforcement learning agent and the rewards for eating the food in alternative are assigned by the reasoning agent. The reasoning agent is a simple agent which tries to find the optimal solution for the agent.

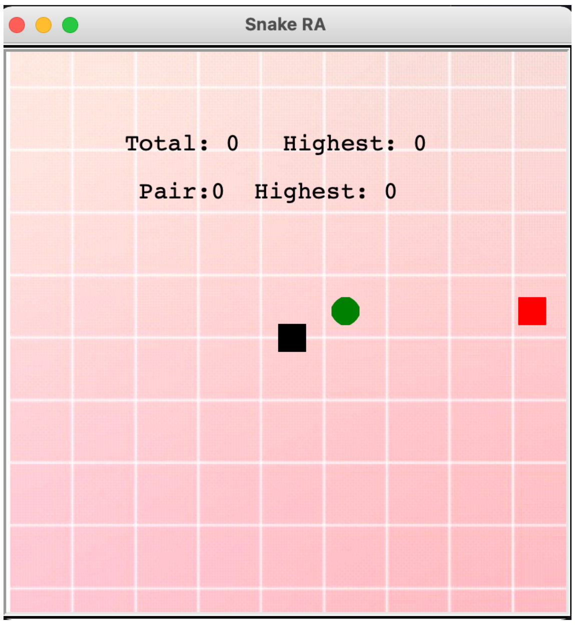

Figure 6: Snake Environment

In the above Fig.6 the environment is represented. The snake is represented by a black square. The food is represented by a green circle (apple) and red square (meat).

4.1 Actions, rewards and states

The agent can choose between going up, right, down or left. The rewards and state space are a bit harder. There are multiple solutions, and one will work better than the other. For now, let's try the following. If the snake grabs an apple, give a reward of 20. If the snake dies, the reward is -500. To help the agent, give a reward of 1 if the snake comes closer to the foods, and a reward of -1 if the snake moves away from the apple. And coming to the state, where scaled coordinates of the snake and apple are the same, the state is 1. Adding the location of obstacles so the agent learns to avoid them. Here is the set of **actions**:

- *UP* - the snake moves up.
- *RIGHT* - the snake moves right.
- *DOWN* - the snake moves down.
- *LEFT* - the snake moves left.

The tables below show the state space and the reward space. These are the same for the agent and the reasoning agent.

For eating an apple	20
For coming closer to foods	0
For going away from foods	-1
For hitting the wall or itself	-500

Table 3: Set of *rewards*.

Eat meat in previous step	0/1
Eat apple in previous step	0/1
Top view of snake	0/1
Bottom view of snake	0/1
Left view of snake	0/1
Right view of snake	0/1
Food above the snake	0/1
Food below the snake	0/1
Food on right side of the snake	0/1
Food on left side of the snake	0/1
Wall above the snake	0/1
Wall on the right	0/1
Wall below the snake	0/1
Wall on the left	0/1
Snake direction is up	0/1
Snake direction is down	0/1
Snake direction is left	0/1
Snake direction is right	0/1

Table 4: Set of *states*.

Tab 3 and 4 are used to show the set of rewards and states of the environment. Two first elements in set of states connect with restraining bolts state. For example: value of 2 components are (0,0) (1,0) (0,1) correspond with 3 reasoning agent state q1,q2,q3 as we discuss in section 2. The reason agent state does not change until snake eat food (meat or apple) , by the reason the policy can know the current state of restraining bolts and can learn correctly through optimize agent's state to action.

5 Results

From Fig.7 Training the agent with the reasoning agent for 160 episodes with 10000 steps per episode. Approximately at 60 episodes the agent achieved a maximum reward and the learning curve shows that the agent is learning rapidly in first fifty episodes.

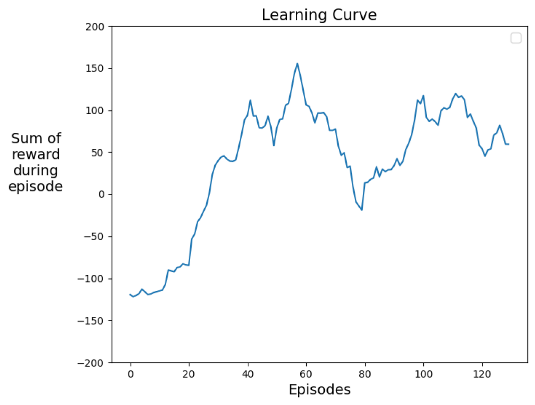

Figure 7: Represents the **learning curve** and the maximum **reward** without adding the states to the model.

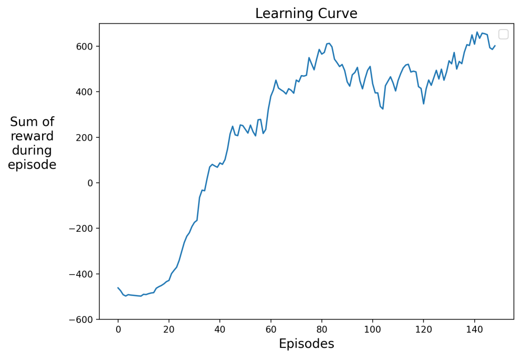

Figure 8: Represents the **learning curve** and the maximum **reward** with adding the states to the model.

Fig.7 and Fig.8 represents the comparison between the different approaches. One without including the state knowledge from reasoning agent and one with the state knowledge where the information of different states is passed along with the respective rewards. The difference in the performance is massive which proves that adding more knowledge to the reinforcement learning algorithm along with environment rewards is the best approach when dealing with complex conditions.

From Fig.9 we can say that the agent has learned to eat the food in way which avoids the wall and hitting itself. Then the reward is given to the DQN agent and in the same way the reasoning agent is rewarded when the snake eats food in pairs(red, green) As we can see in the figure the agent is able to archive a pair score of 15 correspond 30 point and total food score of 33 point and out algorithm is 91% successful rate.

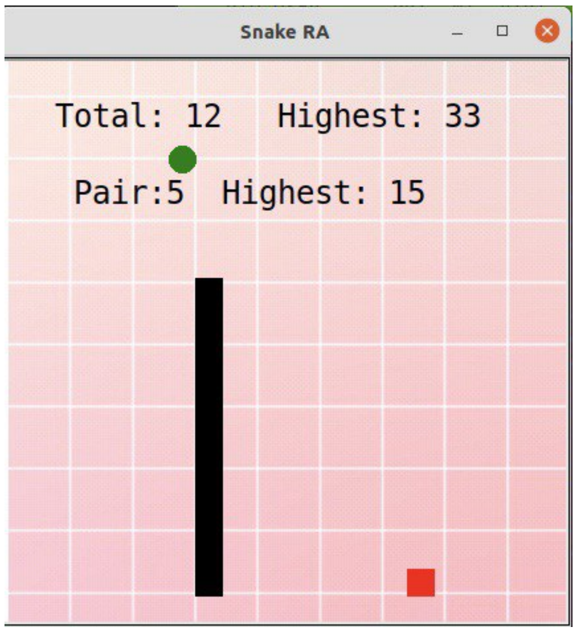

Figure 9: The snake represents the trained **agent**.

6 Conclusion and future works

The main motive of this project is to prove that complex conditions as restraining bolts can be as used as effective module for fast implementation into the already present environment without redoing the whole network

- Using Restraining Bolts can help snake agent eating meat interleave eating apple.
- The convergence of DQN need more episodes when using reasoning agent.
- The simulation environment is light computation so author does not need GPU.
- In future work, we can put some obstacle in environment. Or extend more complex task for snake.

References

[1] *Ronen I. Brafman, Giuseppe De Giacomo, and Fabio Patrizi. "LTLf/LDLf Non-Markovian Rewards". In: AAAI. 2018.*

[2] *Alberto Camacho et al. "LTL and Beyond: Formal Languages for Reward Function Specification in Reinforcement Learning". In: Aug. 2019, pp. 6065–6073. DOI: 10.24963/ijcai.2019/840.*

[3] *Giuseppe De Giacomo et al. Reinforcement Learning for LTLf/LDLf Goals. July 2018*

[4] *Giuseppe De Giacomo et al. "Foundations for Restraining Bolts: Reinforcement Learning with LTLf/LDLf Restraining Specifications". In:ICAPS. 2019*

[5] *https://github.com/whitemech/LTLf2DFA*