The Cat and The Mouse

Alfonso D'Acunzo
0000907058
Autonomous and Adaptive Systems Course
Department of Computer Science
Bologna University

September 10, 2020

Abstract

In this implementation I tried to connect a 2D video-game World with Artificial Intelligent system.This little application is inspired by Tom and Jerry movies where the mouse tries to take the food without being discovered by the Cat. In this situation the mouse should find the best way to get cheese on the opposite side of the room. The algorithm that manages his movements is a Deep Q Learning. The following report aims to explain all the steps taken and the design choices made to arrive at a working version of the software.

1 Introduction

The goal is to provide an overview of existing RL methods on an intuitive level by avoiding any deep dive into the models or the math behind it. Reinforcement Learning is a more complex and challenging method to be realized, but basically, it deals with learning via interaction and feedback, or in other words acting in an environment and receiving rewards for it. Essentially an agent (or several) is built that can perceive and interpret the environment in which is placed, furthermore, it can take actions and interact with it.

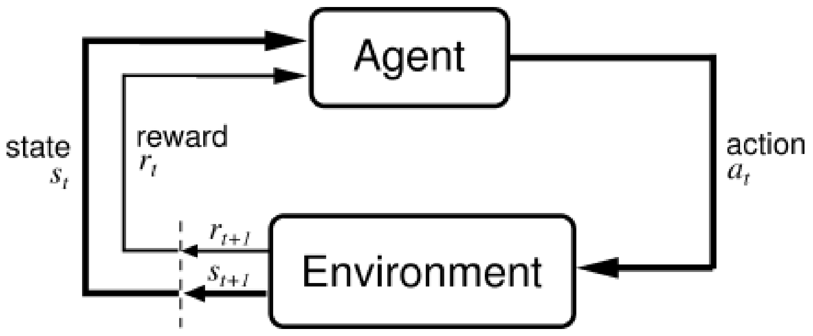

Figure 1: Agent Environment Interaction

In this Project we develop a simple case of interaction between a mere video game, and Reinforcement Learning Algorithm, in particular we use Deep Q Network method for interaction e movement. DQN is Q-learning with Neural Networks . The motivation behind is simply related to big state space environments where defining a Q-table would be a very complex, challenging and time-consuming task. Instead of a Q-table Neural Networks approximate Q-values for each action based on the state.

2 Environment

The game environment chosen in this project is made up by a 7x7 grid, which can be readjusted at though configuration parameters on the top of the environment file, build this is allowed by means of Python suite: TKinter. Tkinter is Python's defacto standard GUI (Graphical User Interface) package. It is a thin object-oriented layer on top of Tcl/Tk. Through it we can create the basic grid of movements, while the PIL library allows us to insert the elements of interaction within the scene, which in this case are three:

- mouse: main actor within the environment, it must reach his goal;
- cat: "enemy" of the game, who must prevent that mouse to reach the cheese;
- cheese: this is the agent goal, where the episode is done and the system compute the value reward.

For the movement of the characters in the game, it was decided to give the mouse total freedom of movement, it can always move in all directions with a single step for each interval. While the cat exhibits a periodic horizontal shift of unit pitch. In the end, the cheese is kept fixed in opposite side of the mouse starts. The characters are positioned in specific positions with an fixed assignment within environment algorithm, and restored in his beginning position at the end of each episodes with reset() function. The check_reward() function is used to verify the achievement of the goal, we have also added 2 types of rewards:

- positive: assigned upon reaching the GOAL and assigned once per episode;
- negative: attributed to single overlap by the mouse with the cat, in the same cell in that interval.

Moving on empty cells does not involve any kind of negative or positive reward. In the end all the negative and positive rewards are computed for the final score, the score interval goes from -∞ to 1.

3 Neural Network

Given the nature of the application implemented, the choice of using a Deep Q Network as function approximator within the current project seemed very valid, allowing the extension to considerably more complex problems that cannot be addressed with a tabular approach. The primary focus of the work has been to identify and implement a number of changes to the original version of DQN, with the aim of achieving better results in the case of games with sparse rewards.

Together with the DQN, the SARSA algorithm has been implemented, it is used to maximize the Q-Value on the state s' from the target model. After defining the

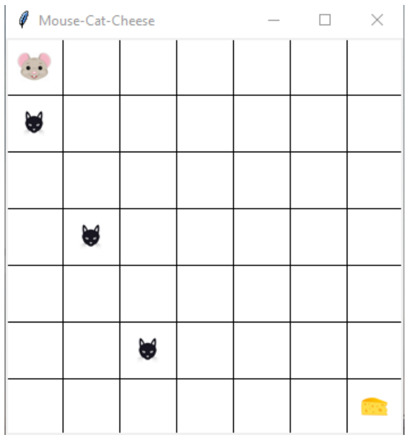

Figure 2: Environment with 7x7 grid world

target function, we make mini-batch which includes target Q-Value and predicted Q-value and do the model fit!

The SARSA features are made up of tuples:

- state: situation of the system at the time t with him state space;
- action: move executed in the state s at time t;
- reward: identifies the reward at the end of the step;
- next_state: system state at time t+1, as a consequence of action a(t);
- next_action: system action at time t+1.

For reasons of efficiency, in the implementation of the DQN used, this function contains an additional value ”end”, it is a Boolean variable that indicates whether the state is terminal, in this case when the mouse reaches cheese in the opposite side if the room.

```
action = agent.get_action(state)
next_state, reward, end = environment.step(action)
next_state = np.reshape(next_state, [1, 15])
next_action = agent.get_action(next_state)
agent.train_model(state, action, reward, next_state,
        next_action, end)
```

The neural network model employed involves the use of state information as input, processed to derive the output from the Q-Value of each action. The construction of the neural network involves the use of three levels of learning: all of the ”dense” type, which are very efficient with this type of problems. Each level has a different exit status format, which is then taken as input by the next, the use of different sizes is useful for improving the availability of entire network.

```
model = tf.keras.Sequential([
        tf.keras.layers.Dense(64, input_dim=self.state_size,
                activation='relu'),
        tf.keras.layers.Dense(30, activation='relu'),
        tf.keras.layers.Dense(self.action_size,
                    activation='linear')
])
model.summary()
model.compile(loss='mse',
        optimizer=Adam(lr=self.learning_rate))
```

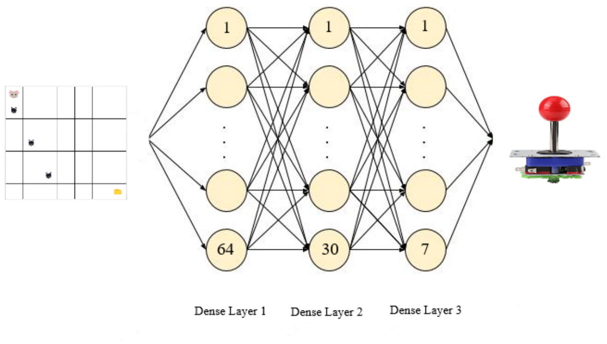

Figure 3: Neural Network Model Implemented

Another important feature concerns the application of the epsilon-greedy policy, the agent's action can occur randomly or it is possible to predict the reward based on the given status. The application provides that epsilon-greedy parameter is strictly monotonous decreasing as the increase number of steps performed during the training. It can be considered as network uncertainty parameter, which decreases when the exploration of the environment increases.

```
if np.random.rand() <= self.epsilon:
        return random.randrange(self.action_size)
else:
        state = np.float32(state)
        q_values = self.model.predict(state)
        return np.argmax(q_values[0])
```

It is planned to train the neural network with 1000 episodes, they are considered sufficient for a good result from the point of view of performance and learning time.

4 Results

At the end of the training it is possible to analyze the results obtained, from the point of view of scores, epsilon, step. The three graphs that will be generated show the real improvement of the network as the number of episodes passes.

As a first important result we can observe the reward trend at each episode, we notice that already after 500 episodes it achieves the maximum reward and good stability, in which we can observe that for most of cases the reward achieved is defined in the interval [-1, 1].

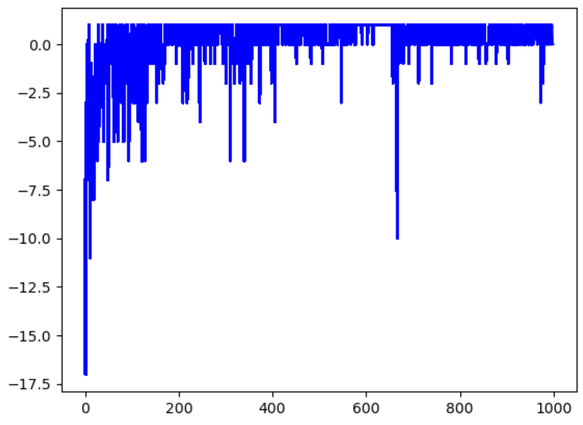

Figure 4: Reward - Episode Graph

Further results that we can identify is the decreasing number of steps taken by the agent to achieve the goal. we note that already after about 400 episodes the neural network is very close to the minimum number of steps allowed, 13. This means that the network can find the correct path already after a few episodes and that it can also be used on more complex and large systems.

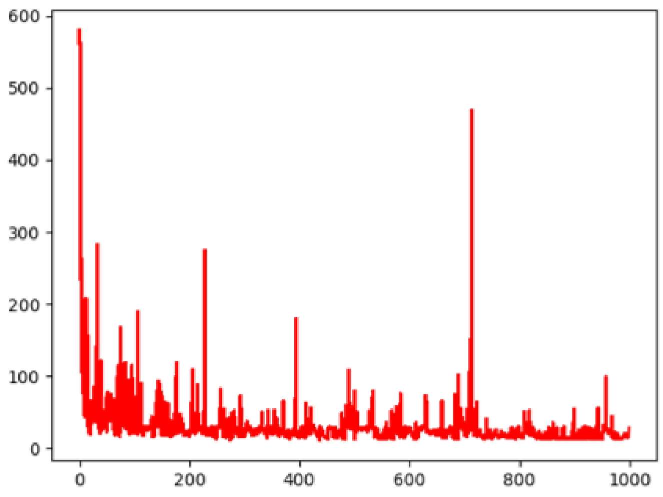

Figure 5: Step - Episode Graph

The last important result that we can observe concerns the gradual lowering of the epsilon parameter, it will tend to zero as the increasing episodes number and steps done during the learn.

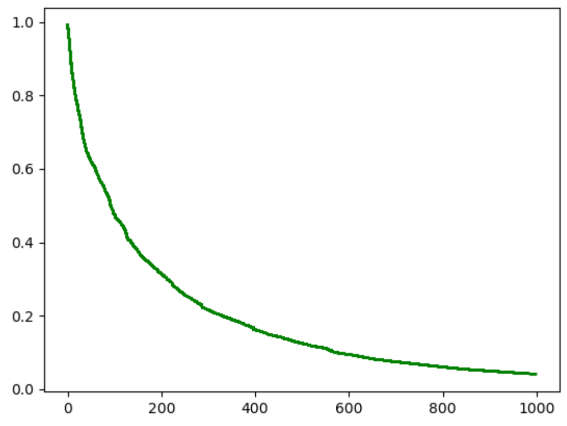

Figure 6: Epsilon - Episode Graph

5 Conclusions

Before the implementation of the network, I means that the union between DNQ algorithm and SARSA features was very strange, but analyzing the overall results obtained, we can see that this neural network, in this particular case, works very well even after about 500 training episodes. Finally we can say that the whole system performs are very well, it achieves a good optimization and stability in the last stages of learning.

One of the major issue meets concerns the advancement training time, due by the uncertainty of the environment in first stages. In fact, the training operation takes several hours considering the use of 1000-episode cycle. A possible improvement to reduce the computation time could be to exploit the computational power of the GPU of ours computer (if a GPU is available).