Sapienza Università di Roma

Department of Computer, Control and Management Engineering

Atari Breakout with $\text{LTL}_f/\text{LDL}_f$ Goals

Elective in Artificial Intelligence: Reasoning Robots

Students:

Ivan Bergonzani

Michele Cipriano

Armando Nania

Professor:

Giuseppe De Giacomo

Academic Year 2017/2018

Contents

1 Introduction

Introduction to the whole project, structure of the report and summary of the work.

2 Reinforcement Learning

Reinforcement learning [1] is an area of machine learning which aims at studying how to to develop agents that can interact with their environment maximing a cumulative reward. The environment can be formally defined as a Markov Decision Process (MDP), which is a tuple $\langle S, A, \delta, R \rangle$ where S is a finite state of states that can represent the environment, A is a finite state of actions that can be perform by the agent in the environment, δ is a probability function modeling the transition from a state to another when performing a certain action and R is a reward function which models the reward received by the environment when performing a certain action which makes the agent move from a state to another.

An interesting property of the MDP is that it satisfies the Markov property, hence, the future states that will be reached by the agent do not depend on the past interaction of the environment, but just on the current state. This makes it possbile to define the transition and the reward function depending only on the current state (and of course the action and the future state of interest).

This section considers two common reinforcement learning algorithm, namely Q-Learning and SARSA, which have been used in our experiments in order to train an agent interacting with an Atari Breakout environment (section 5.4).

2.1 Q-Learning

Q-Learning is an temporal difference (TD) algorithm that directly approximates the optimal action-value function. This method guarantees to find an optimal behaviour under the assumption that the all state-action pairs are updated infinitely many times. It is defined [1] by the following equation:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \Big] \tag{1}$$

Let's briefly discuss the implementation used in our project by studying the Python implementation (Algorithm 1). The algorithm is defined by the class `QLearning` that extends the abstract class `TDBrain`. The constructor of the class (lines 2-4) simply calls its parent constructor that will initialize the parameters of the object, hence, the observation space and the action space (`gym` objects), the strategy used by the policy function, which is ε-greedy by default and the hyperparameters γ, α and λ of the upper class. The abstract method inherited from `TDBrain` is `update_Q`, which should be implemented in order to define how to update the action-state table. The method (lines 6-21) simply follows Eq. 1.

TODO: eligibility.

Algorithm 1: Q-Learning algorithm Python implementation.

```
class QLearning(TDBrain):
    def __init__(self, observation_space:Discrete, action_space, policy:Policy=EGreedy(),
                gamma=0.99, alpha=None, lambda_=0):
        super().__init__(observation_space, action_space, policy, gamma, alpha, lambda_)

    def update_Q(self, obs:AgentObservation):
        state, action, reward, state2 = obs.unpack()

        action2 = self.choose_action(state2)
        Qa = np.max(self.Q[state2])
        actions_star = np.argwhere(self.Q[state2] == Qa).flatten().tolist()

        delta = reward + self.gamma * Qa - self.Q[state][action]
        for (s, a) in set(self.eligibility.traces.keys()):
            self.Q[s][a] += self.alpha.get(s,a) * delta * self.eligibility.get(s, a)
            if action2 in actions_star:
                self.eligibility.update(s, a)
            else:
                self.eligibility.to_zero(s, a)

        return action2
```

2.2 SARSA

A similar TD algorithm is the SARSA algorithm, which name comes from the fact that at each timestep a quintuple $\langle S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}\rangle$ is considered. As before, SARSA converges to an optimal action-value function under the assumption that all state-action pairs are updated infinitely many times. It is defined [1] by the following equation:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\Big] \qquad (2)$$

Let's briefly discuss the implementation used in our project by studying the implementation (Algorithm 2), as done before with the Q-Learning algorithm. Again, the algorithm is defined by the class `Sarsa` that extends the abstract class `TDBrain`. The constructor of the class (lines 2-4) calls its parent constructor initializing the parameters of the upper class exactly in the same way as the class `QLearning`. The class implements the inherited method `update_Q` by following Eq. 2 (lines 6-17). TODO: eligibility.

Algorithm 2: SARSA algorithm Python implementation.

```python
class Sarsa(TDBrain):
    def __init__(self, observation_space:Discrete, action_space, policy:Policy=EGreedy(),
                 gamma=0.99, alpha=None, lambda_=0.0):
        super().__init__(observation_space, action_space, policy, gamma, alpha, lambda_)

    def update_Q(self, obs:AgentObservation):
        state, action, reward, state2 = obs.unpack()

        action2 = self.choose_action(state2)
        Qa = self.Q[state2][action2]

        delta = reward + self.gamma * Qa - self.Q[state][action]
        for (s, a) in set(self.eligibility.traces.keys()):
            self.Q[s][a] += self.alpha.get(s,a) * delta * self.eligibility.get(s, a)
            self.eligibility.update(s, a)

        return action2
```

3 LTL$_f$/LDL$_f$ Non-Markovian Rewards

Intro.

3.1 Theoretical Background

Introduction to the research paper.

3.2 Examples

How it can be used to train a RL model.

4 OpenAI Gym

OpenAI gym [2] is a toolkit for developing and comparing reinforcement learning algorithms, without making assumptions about the structure of the agent interacting with the environment, in order to keep development flexible to updates on both sides.

4.1 Framework

The framework of gym allows to interact easily with an environment, giving the developers to tools they need to perform actions and to observe the state of the environment itself. In this way it is possible to focus more on the development of the agent without spending time on the structure of the world.

gym makes it possible to interact with multiple kinds of environments. Among these, the authors of the framework developed the support for Arcade Learning Environment [3], which includes all the classing Atari games, including Breakout, which has been used in this project.

4.2 Examples

Let's consider a simple example to understand how gym works and how the framework can be used to interact with an environment. The description will follow Algorithm 3.

Algorithm 3: Example of a random interaction with the gym environment BreakoutNoFrameskip-v4, used also in our experiments of subsection 5.4.

```
1  import gym
2
3  env = gym.make("BreakoutNoFrameskip-v4")
4  env.reset()
5
6  for _ in range(1000):
7    env.render()
8    action = env.action_space.sample() # takes random actions
9    observation, reward, done, info = env.step(action)
10   if done == True:
11     env.reset()
12
13 env.close()
```

Initially (line 1) the framework is imported. Then (line 3-4) an environment is created specifying its name and initializing it. The program makes a random agent interact randomly with the environment for 1000 episodes (lines 6-11) before closing the environment. Line 7 renders the current observation of the

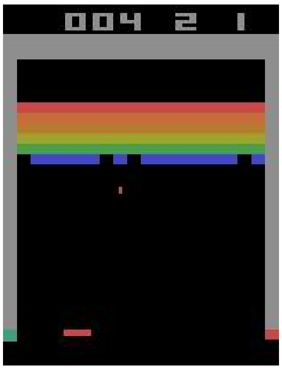

Figure 1: Observation of a frame of the environment `BreakoutNoFrameskip-v4`.

environment on screen, line 8-9 performs a random action between those available in this Brekout version, note that the method `step` return an `observation` (shown in Fig. 1), which is an array of pixels that represent the current state of the environment, a `reward`, which is a value return by the game after performing the specified action `action`, a boolean value `done`, which is `True` is the game is over, `False` otherwise, and `info` which contains extra information about the game. Lines 10-11 handles the case when the game is over, resetting the environment.

5 Atari Breakout

Intro.

5.1 PyGame Breakout

Original implementation of the paper (non-ATARI).

5.2 Arcade Learning Environment

ATARI Breakout (from ALE) and differences from the other one.

5.3 Implementation

Intro to implementation.

5.3.1 Robot Features Extractor

`RobotFeatureExtractor` (OpenCV). Extracts features of the robot (robot and ball positions).

Algorithm 4: Robot feature extractor Python implementation.

```python
class BreakoutNRobotFeatureExtractor(BreakoutRobotFeatureExtractor):

    def __init__(self, obs_space):
        robot_feature_space = Tuple((
            Discrete(287),
            Discrete(157),
        ))

        self.prev_ballX = 0
        self.prev_ballY = 0
        self.prev_paddleX = 0
        self.still_image = True

        super().__init__(obs_space, robot_feature_space)

    def _extract(self, input, **kwargs):
        self.still_image = not self.still_image
        if self.still_image:
            return (self.prev_ballX-self.prev_paddleX+143, self.prev_ballY)
        # Extract position of the paddle:
        paddle_img = input[189:193,8:152,:]
        gray = cv2.cvtColor(paddle_img, cv2.COLOR_RGB2GRAY)
        thresh = cv2.threshold(gray, 60, 255, cv2.THRESH_BINARY)[1]
```

```
        cnts = cv2.findContours(thresh.copy(), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
        cnts = cnts[0] if imutils.is_cv2() else cnts[1]
        min_distance = np.inf
        paddleX = self.prev_paddleX
        for c in cnts:
            M = cv2.moments(c)
            if M["m00"] == 0:
                continue
            pX = int(M["m10"] / M["m00"])
            if abs(self.prev_paddleX - pX) < min_distance:
                min_distance = abs(self.prev_paddleX - pX)
                paddleX = pX

        # Extract position of the ball:
        ballX = self.prev_ballX
        ballY = self.prev_ballY
        ballspace_img = input[32:189,8:152,:]
        lower = np.array([200, 72, 72], dtype=np.uint8)
        upper = np.array([200, 72, 72], dtype=np.uint8)
        mask = cv2.inRange(ballspace_img, lower, upper)
        cnts = cv2.findContours(mask.copy(), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
        cnts = cnts[0] if imutils.is_cv2() else cnts[1]
        for c in cnts:
            M = cv2.moments(c)
            # Avoid to compute position of the ball if M["m00"] is zero:
            if M["m00"] == 0:
                continue
            # Calculate the centroid
            cX = int(M["m10"] / M["m00"])
            cY = int(M["m01"] / M["m00"])
            # Check that the centroid is actually part of the ball:
            left_black = False
            right_black = False
            if cX > 3:
                if ballspace_img[cY][cX-3][0] != 200 or \
                    ballspace_img[cY][cX-3][1] != 72 or \
                    ballspace_img[cY][cX-3][2] != 72:
                    left_black = True
            else:
                 if ballspace_img[cY][cX+3][0] != 200 or \
                     ballspace_img[cY][cX+3][1] != 72 or \
                     ballspace_img[cY][cX+3][2] != 72:
                     right_black = True
            if left_black or right_black:
                ballX = cX
                ballY = cY

        self.prev_ballX = ballX
        self.prev_ballY = ballY
        self.prev_paddleX = paddleX
```

```python

        return (self.prev_ballX - self.prev_paddleX + 143, self.prev_ballY)
```

5.3.2 Goal Features Extractor

`GoalFeatureExtractor` (OpenCV). Extracts 6x18 table representation of the bricks in order to evaluate a formula.

Algorithm 5: Goal feature extractor Python implementation.

```python
class BreakoutGoalFeatureExtractor(FeatureExtractor):
    def __init__(self, obs_space, bricks_rows=6, bricks_cols=18):
        self.bricks_rows = bricks_rows
        self.bricks_cols = bricks_cols
        output_space = Box(low=0, high=1, shape=(bricks_cols, bricks_rows), dtype=np.uint8)
        super().__init__(obs_space, output_space)

    def _extract(self, input, **kwargs):
        bricks_features = np.ones((self.bricks_cols, self.bricks_rows))
        for row, col in itertools.product(range(self.bricks_rows), range(self.bricks_cols)):
            # Pixel of the observation to check:
            px_upper_left  = int( 8 + 8 * col)
            py_upper_left  = int(57 + 6 * row)
            px_upper_right = int(15 + 8 * col)
            py_upper_right = int(57 + 6 * row)

            # Checking max because the input has 3 channels:
            if max(input[py_upper_left][px_upper_left]) == 0 or \
                max(input[py_upper_right][px_upper_right]) == 0:
                bricks_features[col][row] = 0

        return bricks_features
```

*Ext used to improve implementation.

5.3.3 Temporal Goals

$\text{LTL}_f/\text{LDL}_f$ implementation (with Marco Favorito libraries).

Algorithm 6: $\text{LTL}_f/\text{LDL}_f$ formulas Python implementation.

```python
def get_breakout_lines_formula(lines_symbols):
    # Generate the formula string
    # E.g. for 3 line symbols:
```

```
    # "<(!l0 & !l1 & !l2)*;(l0 & !l1 & !l2);(l0 & !l1 & !l2)*;(l0 & l1 & !l2); (l0 & l1 & !l2)*; l0 & l1 & l2>tt"
    pos = list(map(str, lines_symbols))
    neg = list(map(lambda x: "!" + str(x), lines_symbols))

    s = "(%s)*" % " & ".join(neg)
    for idx in range(len(lines_symbols)-1):
        step = " & ".join(pos[:idx + 1]) + " & " + " & ".join(neg[idx + 1:])
        s += ";({0});({0})*".format(step)
    s += ";(%s)" % " & ".join(pos)
    s = "<%s>tt" % s

    return s

class BreakoutCompleteLinesTemporalEvaluator(TemporalEvaluator):
    """Breakout temporal evaluator for delete columns from left to right"""

    def __init__(self, input_space, bricks_cols=3, bricks_rows=3, lines_num=3, gamma=0.99, on_the_fly=False):
        assert lines_num == bricks_cols or lines_num == bricks_rows
        self.line_symbols = [Symbol("l%s" % i) for i in range(lines_num)]
        lines = self.line_symbols

        parser = LDLfParser()


        string_formula = get_breakout_lines_formula(lines)
        print(string_formula)
        f = parser(string_formula)
        reward = 10000

        super().__init__(BreakoutGoalFeatureExtractor(input_space, bricks_cols=bricks_cols, bricks_rows=bricks_rows),
                         set(lines),
                         f,
                         reward,
                         gamma=gamma,
                         on_the_fly=on_the_fly)

    @abstractmethod
    def fromFeaturesToPropositional(self, features, action, *args, **kwargs):
        """map the matrix bricks status to a propositional formula
        first dimension: columns
        second dimension: row
```

```python
        """
        matrix = features
        lines_status = np.all(matrix == 0.0, axis=kwargs["axis"])
        result = set()
        sorted_symbols = reversed(self.line_symbols) if kwargs["is_reversed"] else self.line_symbols
        for rs, sym in zip(lines_status, sorted_symbols):
            if rs:
                result.add(sym)

        return frozenset(result)

class BreakoutCompleteRowsTemporalEvaluator(BreakoutCompleteLinesTemporalEvaluator):
    """Temporal evaluator for complete rows in order"""

    def __init__(self, input_space, bricks_cols=3, bricks_rows=3, bottom_up=True, gamma=0.99, on_the_fly=False):
        super().__init__(input_space, bricks_cols=bricks_cols, bricks_rows=bricks_rows, lines_num=bricks_rows, gamma=gamma, on_the_fly=on_the_fly)
        self.bottom_up = bottom_up

    def fromFeaturesToPropositional(self, features, action, *args, **kwargs):
        """complete rows from bottom-to-up or top-to-down, depending on self.bottom_up"""
        return super().fromFeaturesToPropositional(features, action, axis=0, is_reversed=self.bottom_up)
```

Atari wrappers (OpenAI).

5.4 Experiments

Results with 6x18 non-ATARI Breakout (+CODE).

Results with our experiments (+CODE).

6 Conclusion

Why it does not work.

Summary + differences between the two environments.

Future works (neural networks and parallel computation).

References

[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction.* The MIT Press, second ed., 2018.

[2] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "OpenAI Gym," 2016.

[3] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The Arcade Learning Environment: An Evaluation Platform for General Agents," *Journal of Artificial Intelligence Research*, vol. 47, pp. 253–279, jun 2013.