SAPIENZA
UNIVERSITÀ DI ROMA

DEPARTMENT OF COMPUTER, CONTROL AND MANAGEMENT ENGINEERING

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

ELECTIVE IN ARTIFICIAL INTELLIGENCE:
REASONING ROBOTS

Professor:
Giuseppe De Giacomo

Students:
Ivan Bergonzani
Michele Cipriano
Armando Nania

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 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 set of states that can represent the environment, A is a finite set of actions that can be perform by an 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 6).

2.1 Q-Learning

Q-Learning is a 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:

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

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.

```
1  class QLearning(TDBrain):
2      def __init__(self, observation_space:Discrete,
           ↪ action_space, policy:Policy=EGreedy(),
3                  gamma=0.99, alpha=None, lambda_=0):
4          super().__init__(observation_space, action_space,
               ↪ policy, gamma, alpha, lambda_)
5
6      def update_Q(self, obs:AgentObservation):
7          state, action, reward, state2 = obs.unpack()
8
9          action2 = self.choose_action(state2)
10         Qa = np.max(self.Q[state2])
11         actions_star = np.argwhere(self.Q[state2] == Qa).
               ↪ flatten().tolist()
12
13         delta = reward + self.gamma * Qa - self.Q[state][
               ↪ action]
14         for (s, a) in set(self.eligibility.traces.keys()):
15             self.Q[s][a] += self.alpha.get(s,a) * delta *
                   ↪ self.eligibility.get(s, a)
16             if action2 in actions_star:
17                 self.eligibility.update(s, a)
18             else:
19                 self.eligibility.to_zero(s, a)
20
21         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:

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

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.

```
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 $\text{LTL}_f/\text{LDL}_f$ Non-Markovian Rewards

Recently, non-Markovian reward decision processes (NMRDPs) has attracted interest in the scientific community because of the possibility of specifying them as MDPs with $\text{LTL}_f/\text{LDL}_f$ non-Markovian rewards [4]. In particular, it is possible to model the problem with two separate representations of the world, one for the agent (low-level) and one for the goal (expressed in terms of high-level fluents).

This section presents the approach used in [4], where an efficient method has been developed in order to work with NMRDPs. The theory behind the main idea is quickly described and an example on a theoretical Breakout environment is discussed in order to be used in the following sections easily.

3.1 Theoretical Background

Before describing the problem, let's give a formal definition of NMRDP. A non-Markovian reward decision process is a tuple $M = \langle S, A, \delta, \bar{R} \rangle$, with S finite set of states that can represent the environment, A is a finite set of actions that can be performed by an agent in the environment, δ is a probability function modeling the transition from a state to another when performing a certain action and $\bar{R} : (S \times A)^* \to \mathbb{R}$ is a function from finite state-action sequences (traces) to real-values that represents the reward given by the environment when performing a certain state-action sequence. Specifying a non-Markovian reward function explicitely is difficult even when considering a finite number of traces. Luckily, the $\text{LTL}_f/\text{LDL}_f$ formalism allows to specify $\bar{R}$ implicitely using a set of pairs $\{(\phi_i, r_i)\}_{i=1}^m$ with ϕ_i boolean proposition over the components of the state vector and r_i such that, given a current trace $\pi = \langle s, a_1, \ldots, s_{n-1}, a_n \rangle$, the agent receives at s_n a reward r_i if ϕ_i is satisfied by π, hence:

$$\bar{R}(\pi) = \begin{cases} r_i & \text{if } \pi \vDash \phi_i \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

Since the NMRDP rewards are based on traces, instead of state-action pairs, typical learning algorithms like Q-learning or SARSA cannot be used. Nevertheless, it has been shown [4] that for any NMRDP $M = \langle S, A, \delta, \{(\phi_i, r_i)\}_{i=1}^m \rangle$ there exists an MDP $M' = \langle S', A', \delta' R' \rangle$ that is equivalent to M. The idea behind the proof consists in starting from an initial decision process $M_{ag}^{goal} = \langle S, A, R, \mathcal{L}, \delta_{ag}^g, \{(\phi_i, r_i)\}_{i=1}^m \rangle$ with $\text{LTL}_f/\text{LDL}_f$ goals (with $\mathcal{L}$ set of of configuration of the high-level features needed for expressing ϕ_i), transform it into a

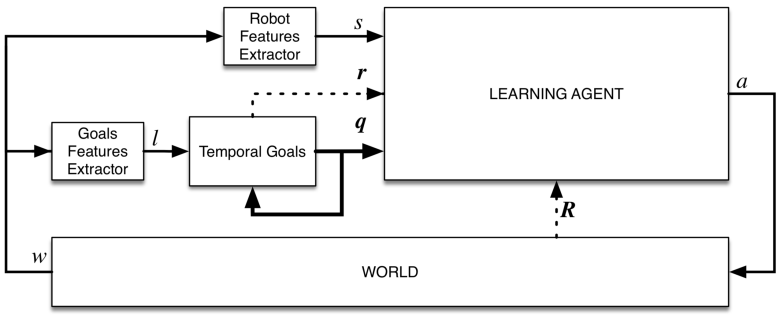

Figure 1: Pipeline describing how the agent is interacting with the world and how the robot features extractor and the goal features extractor are used in order to handle non-Markovian rewards.

NMRDP in order to further transform it into a MDP where it is possible to execute learning algorithms such as Q-learning. All the details are out of the scope of the project and are discussed in [4]. The set of states S is used to express low-level features of the agent.

Once the MDP has been determined it is possible to train a model that will be able to manage non-Markovian rewards as well. Fig. 1 shows a pipeline of the agent interacting with the world. In particular, a robot feature extractor will analyze the state of the world w in order to pass a representation s to the agent, a goal feature extractor will extract higher-level features of the world l that can be used to determine if a temporal goal has been achieved or not. In positive case a reward r is given to the agent. The evaluation q of the formula that specifies the temporal goals is given to the learning agent as well, in order to construct an extended state of the MDP as specified by the theory above. Each time the learning agent performs an action a, the environment changes and it gives to the agent a reward R.

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 6.

```
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 timesteps (lines 6-11) before closing the environment. Line 7 renders the current observation of the

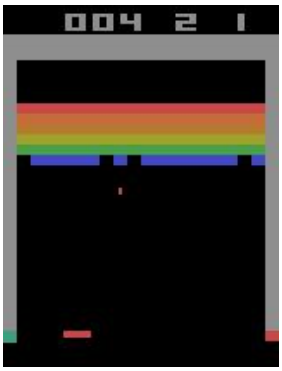

Figure 2: Observation of a frame of the environment `BreakoutNoFrameskip-v4`.

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. 2), 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` when 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

This section contains the main part of the project, describing in detail our starting point, how the program has been developed, the results achieved and the comparison with the original implementation, which uses a non-Atari version of the game Breakout [4] with the same number of bricks used in the Atari implementation of `gym`, already introduced in section 5.2.

Initially, the non-Atari version of Breakout (built on PyGame) is introduced, its implementation is discussed and the results on the training with a brick matrix of dimension 6×18 are presented. Then the `gym` environment `BreakoutNoFrameskip-v4` is presented and compared with the PyGame breakout. A detailed description of the implementation of the project is described, discussing in detail how the features have been extracted from the environment (both the robot and the goal features), how temporal goals have been used to evaluate the state of the bricks and how everything is connected together in order to make it work with `gym`. In the end, the main experiments performed on `BreakoutNoFrameskip-v4` are presented.

5.1 PyGame Breakout

As introduced above, in [4] a PyGame version of the Breakout game has been used in order to test the algorithms introduced in the paper. The implementation of this Breakout easily allows to determine the state of the environment, saving the status of the bricks (either in the scene or broken), the position of the ball, the direction of the ball and the position of the paddle. This makes it possible to reduce a lot the computational time of the implementation of the agent since the data it receives are already preprocessed in order to have a complete overview of the environment, allowing to focus on higher-level reasoning tasks.

Originally, the paper focused on a Breakout environment with a brick matrix of dimension 4×5. Since the Atari version of Breakout is dealing with a brick matrix of dimension 6×18, a new test has been performed to make the two environments comparable, in order to better understand the potentiality of $\text{LTL}_f/\text{LDL}_f$, the use of non-Markovian rewards and how to approach complex `gym` environments in the future.

The method managed to correctly break all the bricks in around one hour of training. TODO: add three figures with initial state, middle, final. TODO: add plot of rewards (shall this go to Experiments subsec?).

5.2 Arcade Learning Environment

This works aims at comparing the `gym` environment `BreakoutNoFrameskip-v4`, already introduced in section with the non-Atari Breakout used in [4]. In last years, the reinforcement learning community has grown a lot thanks to the introduction of `gym` and deep reinforcement learning algorithms [5] that managed to easily solve complex games that are considered difficult also for humans, often achieving better results than expert human gamers. More and more algorithms are introduced every year, exploiting GPU resources and managing to solve harder games like Montezuma Revenge [6]. The popularity of deep reinforcement learning begun with the introduction of Arcade Learning Environment (ALE) that includes most famous arcade Atari games [3].

The main characteristics of `gym` (or ALE) environments is that the world can be observed only from the pixels of the screen, putting the algorithms at the same level of the human, that can only observe the display while playing. This makes the game a lot more complex since a more abstract reasoning strategy is needed in order to solve the game. This hypothesis should make `gym` games a lot harder than the non-Atari version of Breakout that has been used to test algorithms that work with non-Markovian rewards.

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.

```
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)
```

```
15
16      def _extract(self, input, **kwargs):
17          self.still_image = not self.still_image
18          if self.still_image:
19              return (self.prev_ballX-self.prev_paddleX+143,
                    ↪ self.prev_ballY)
20          # Extract position of the paddle:
21          paddle_img = input[189:193,8:152,:]
22          gray = cv2.cvtColor(paddle_img, cv2.COLOR_RGB2GRAY)
23          thresh = cv2.threshold(gray, 60, 255, cv2.
                ↪ THRESH_BINARY)[1]
24          cnts = cv2.findContours(thresh.copy(), cv2.
                ↪ RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
25          cnts = cnts[0] if imutils.is_cv2() else cnts[1]
26          min_distance = np.inf
27          paddleX = self.prev_paddleX
28          for c in cnts:
29              M = cv2.moments(c)
30              if M["m00"] == 0:
31                  continue
32              pX = int(M["m10"] / M["m00"])
33              if abs(self.prev_paddleX - pX) < min_distance:
34                  min_distance = abs(self.prev_paddleX - pX)
35                  paddleX = pX
36
37          # Extract position of the ball:
38          ballX = self.prev_ballX
39          ballY = self.prev_ballY
40          ballspace_img = input[32:189,8:152,:]
41          lower = np.array([200, 72, 72], dtype=np.uint8)
42          upper = np.array([200, 72, 72], dtype=np.uint8)
43          mask = cv2.inRange(ballspace_img, lower, upper)
44          cnts = cv2.findContours(mask.copy(), cv2.
                ↪ RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
45          cnts = cnts[0] if imutils.is_cv2() else cnts[1]
46          for c in cnts:
47              M = cv2.moments(c)
48              # Avoid to compute position of the ball if M["
                    ↪ m00"] is zero:
49              if M["m00"] == 0:
50                  continue
51              # Calculate the centroid
52              cX = int(M["m10"] / M["m00"])
53              cY = int(M["m01"] / M["m00"])
54              # Check that the centroid is actually part of
                    ↪ the ball:
55              left_black = False
56              right_black = False
57              if cX > 3:
58                  if ballspace_img[cY][cX-3][0] != 200 or \
59                      ballspace_img[cY][cX-3][1] != 72 or \
60                      ballspace_img[cY][cX-3][2] != 72:
61                      left_black = True
62              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

        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.

```
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.

```
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)
```

```
39
40      @abstractmethod
41      def fromFeaturesToPropositional(self, features, action,
            ↪  *args, **kwargs):
42          """map the matrix bricks status to a propositional
                ↪ formula
43          first dimension: columns
44          second dimension: row
45          """
46          matrix = features
47          lines_status = np.all(matrix == 0.0, axis=kwargs["
                ↪ axis"])
48          result = set()
49          sorted_symbols = reversed(self.line_symbols) if
                ↪ kwargs["is_reversed"] else self.line_symbols
50          for rs, sym in zip(lines_status, sorted_symbols):
51              if rs:
52                  result.add(sym)
53
54          return frozenset(result)
55
56  class BreakoutCompleteRowsTemporalEvaluator(
        ↪ BreakoutCompleteLinesTemporalEvaluator):
57      """Temporal evaluator for complete rows in order"""
58
59      def __init__(self, input_space, bricks_cols=3,
            ↪ bricks_rows=3, bottom_up=True, gamma=0.99,
            ↪ on_the_fly=False):
60          super().__init__(input_space, bricks_cols=
                ↪ bricks_cols, bricks_rows=bricks_rows,
                ↪ lines_num=bricks_rows, gamma=gamma,
                ↪ on_the_fly=on_the_fly)
61          self.bottom_up = bottom_up
62
63      def fromFeaturesToPropositional(self, features, action,
            ↪  *args, **kwargs):
64          """complete rows from bottom-to-up or top-to-down,
                ↪ depending on self.bottom_up"""
65          return super().fromFeaturesToPropositional(features
                ↪ , action, axis=0, is_reversed=self.bottom_up
                ↪ )
```

Atari wrappers (OpenAI).

6 Experiments

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

7 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.

[4] G. De Giacomo, L. Iocchi, M. Favorito, and F. Patrizi, "Reinforcement Learning for LTLf/LDLf Goals," *CoRR*, vol. abs/1807.06333, 2018.

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, Feb. 2015.

[6] "Montezuma's Revenge Solved by Go-Explore, a New Algorithm for Hard-Exploration Problems (Sets Records on Pitfall, Too)." https://eng.uber.com/go-explore/.