Solving Grid-World Problem with Reinforcement Learning

Jean Lucien Randrianantenaina
Applied Mathematics, Machine Learning and Artificial Intelligence
Stellenbosch University

***Abstract*—**

I. Introduction

II. Problem statement and modelling

Our goal is to build an agent that solve a an empty 6×6 grid world, e.g., reaching the green cell as illustrated in the Figure 1. The agent (red triangle) interact with the environment through a sequence of observation, actions and rewards. We use the `mini-grid` API from the Farama foundation to simulate the environment.

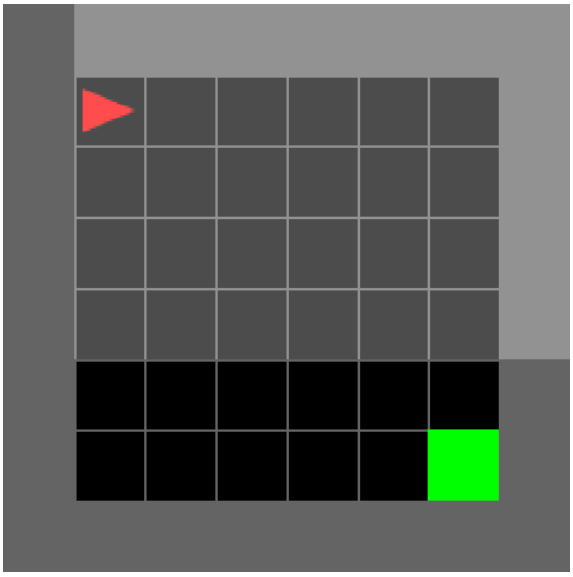

Fig. 1: The `MiniGrid-Empty-8x8-v0` environment

Depending on our configuration, the state can be a tensors, or an image which is returned after each actions. The agent have in total eight possible actions, but we only use three: turn left and right, and move forward. By this setting, we deal with an episodic task, and for each episode the agent can perform at most 256 steps.

If the agent reach the target, it receive a reward r defined by:

$$r = 1 - 0.9 \times \frac{\text{Number of steps}}{\text{Maximum number of steps}} \tag{1}$$

otherwise the reward is zero.

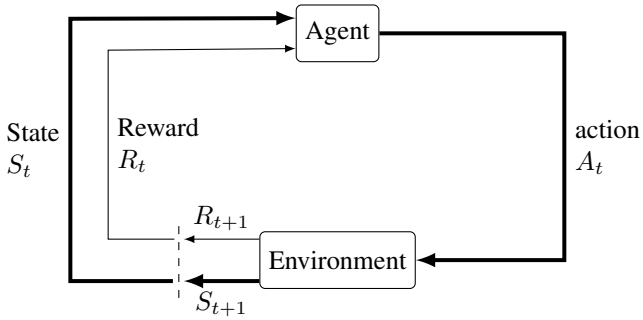

Fig. 2: Reinforcement learning Model

Therefore our task is to make the agent learns how to move in these grid in such a way that it maximize its reward. For this particular environments the agent can get at most a reward of 0.9613 with 11 steps. But for this works we will consider 0.9578 as optimal reward corresponding to 12 steps.

III. Reinforcement Learning technique

In this era, reinforcement learning is widely used to solve the problem of programming intelligent agents that learn how optimise a specific task such as playing game, autonomous vehicle. This is done by learning a policies for sequential decision problems that maximize the discounted cumulative future reward:

$$G_t \triangleq \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \tag{2}$$

Where R is the reward, $\gamma \in [0, 1]$ the discount factor that control the importance fro immediate reward (near to 0) or future reward (near to 1). The agent learn the optimal policy without being explicitly told if its action are good or bad using the reinforcement learning model depicted in the Figure 2.

Several algorithms/technique can be used to solve our problem with reinforcement leaning, in the next section we will look:

- Q-Learning
- SARSA (State-Action-Reward-State-Action)
- Deep Q-Network
- Deep Q-Network with RGB Image techniques.

The algorithm 1 show the skeleton shared by these RL algorithm. In addition to that, we also use the ϵ-greedy strategy to select the action to be performed by the agent. The agent

Parameters:...
foreach *episode* **do**
  (Re)Initialize the environment
  foreach *step* **do**
    Perform an action and consider the obsrvation and reward
    Steps specific to each methods
    if *done or truncated* **then**
      Some step for metric and monitoring
    end
  end
end

Algorithm 1: RL Algorithm Skeleton

uniformly sample an action from the possible action for the given state with a probability ϵ, and use the optimal action for the given state with a probability $1-\epsilon$. In the first case we say that the agent is exploring, and it the second case we say that the agent is exploiting.

For this works we will start with a higher value of ϵ to allow to the agent to explore all the environment, then we decrease it slowly until we reach a small value, this will be controlled by :

$$\epsilon_t = \epsilon_{min} + (\epsilon_{max} - \epsilon_{min}) \exp\left\{-\frac{t}{\Delta}\right\} \tag{3}$$

Where Δ is the decay rate, a higher value correspond to a slow decrease, and a small value correspond to fast decrease.

IV. Tabular methods: Q-Learning and SARSA

Tabular method, refer to a creation of a table called Q-table, containing the value of $Q(S_t = s, A_t = a)$. This value indicate how good is performing the action a on the state s.

For the two algorithm that we present in this section, the observation (a $7 \times 3 \times 3$ array) will be converted into a string to create a MD5 hash to represent the given state.

A. SARSA

The SARSA algorithm is an on-policy methods, that means it iteratively refine a single policy, that also generates controls action within the environments. The update rule is defined by:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \tag{4}$$

Form the equation (4), we obtain the Algorithm 2 fo the SARSA learning.

B. Q-Learning

In contrast of SARSA, the Q-Learning algorithm is an off-policy algorithm, that mean we optimise the target policy using different policy. The update rule is given by

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a(S_{t+1}, a) - Q(S_t, A_t)] \tag{5}$$

Since, we have the Algorithm 3 for Q-learning methods.

These two methods are powerful to deal with few states and action, but not very efficient to deal with a high number of states and action. We can replace the table by a function approximator as we will see in the next section.

Parameters: Step size $\alpha \in]0, 1]$, small $\epsilon > 0$
Initialize $Q(s, a)$ for all $s \in \mathcal{S}^+$ and $a \in \mathcal{A}(s)$ arbitrarily, except that $Q(terminal - state, \cdot) = 0$
foreach *episode* **do**
  Initialize S
  Choose A from S using policy derived from Q (ϵ-greedy)
  foreach *step until S is a terminal state* **do**
    Take the action A, observe R, S'
    Choose A' from S' using policy derived from Q (ϵ-greedy)
    $Q(S, A) \leftarrow Q(S, A) + \alpha[R_{t+1} + \gamma(S', A') - Q(S_t, A_t)]$
    $S \leftarrow S'$
    $A \leftarrow A'$
  end
end

Algorithm 2: SARSA: on-policy learners to estimate the optimal Q-table

Parameters: Step size $\alpha \in (0, 1]$, small $\epsilon > 0$
Initialize $Q(s, a)$ for all $s \in \mathcal{S}^+$ and $a \in \mathcal{A}(s)$ arbitrarily, except that $Q(terminal - state, \cdot) = 0$
foreach *episode* **do**
  Initialize S
  Choose A from S using ϵ-greedy
  foreach *step until S is a terminal state* **do**
    Take the action A, observe R, S'
    Choose A' from S' using ϵ-greedy
    $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a(S_{t+1}, a) - Q(S_t, A_t)]$
    $S \leftarrow S'$
  end
end

Algorithm 3: Q-learning algorithm to estimate the optimal Q-table

V. Function approximation: Deep Q-network

The idea of function approximation is to replace the value function by its approximation instead of using a look-up table. The function can take

- the state and action as input, and $q(s, a)$ as output,
- or the state as input and $q(s, a)$ as output.

This is illustrated in the figure 3.

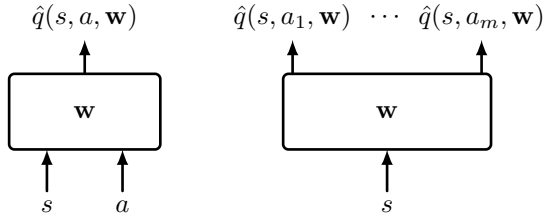

Fig. 3: Illustration of function approximation

In our case, we will use neural network as function approximation. Thus we will consider two types.

A feed forward neural network (MLP) which take a vector $\mathbf{u} \in \mathbb{R}^{49\times 1}$ (state) as input and $q(s,a_1), q(s,a_2), q(s,a_3)$ as output.

Similarly, a Convolutional Neural networks (CNN), which take a stack of four frames (images) at four successive time steps, e.g. a $4 \times 56 \times 56$ tensor as input and same output as the MLP,. The architecture of these neural networks are described in respectively in REF-MLP and REF-CNN.

As we do not have any labels to train the networks, we use two networks: the `policy_net` and `target_net`. The first one is optimised with the `Adam` optimizer, while the second one is fixed, and used to generate a sort of label for the `policy_net`. We also synchronize the weight of the two network in aregular period of the training, to move toward into the optimal values.

To train these networks, we need a dequeue (double end-queue) that store a given number of the experience of the agent formed by the current state, action, next state, and reward, we call this a replay memory $\mathcal{D}$. As new experience are added the oldest data is pushed out of the dequeue. Once we reach have batch size (enough number) of experience, we start to sample from $\mathcal{D}$ and optimize the policy_net parameter by minimizing the Mean Squared Error (MSE) given by:

$$\mathcal{L} = \frac{1}{N} \sum i = 1^N \left[R_i + \gamma \max_{a'} \hat{Q}(s'_i, a', \mathbf{w}^-) - \hat{Q}(s_i, a, \mathbf{w}) \right]^2 \tag{6}$$

where $R_i + \gamma \max_{a'} \hat{Q}(s'_i, a', \mathbf{w}^-)$ is computed using `target_net`, and $\hat{Q}(s'_i, a', \mathbf{w}^-)$ is 0 if s'_i is terminal state, while $\hat{Q}(s_i, a, \mathbf{w})$ is computed with `policy_net`.

Note that, the input of these network (observation) are normalized by rescaling their value between $[-1, 1]$, particularly we convert the RGB image into agrey-scale image for the CNN architecture. We can summarize the whole process in the algorithm 4.

Initialize `policy_net` and `target_net`
Initialize the environments
Set the decay rate for the epsilon decreasing
Set the updating period of `target_net`
Set the total step to 0
Create a replay memory $\mathcal{D}$
foreach *episode* **do**
  Set step to 0
  Make new episode
  Observe the first state
  while *not(done or truncated)* **do**
    Choose A from S using policy derived from Q (ϵ-greedy)
    Increment the total training step
    Execute A, observe R and the new state S'
    Store Transition $< S, A, S', R >$ in $\mathcal{D}$
    Compute the $\mathcal{L}$ and do a gradient descent step
    if *updating period* **then**
      Copy the `policy_net` parameter to `target_net`
    end
  end
end

Algorithm 4: Training a DQN to estimate the optimal policy

VI. Hyper-parameters tuning and Evaluation

A. Hyper-parameters

like all machine learning problem we have several parameter to tune to get the optima results. Grid search is one of the best methods, it iterate though all the possible combination of predefined parameters set. The drawback of this method is the time complexity which can explode considerably when we hyper-parameter space is big and the algorithm is quite slow.

So, instead of using that approach we will use a intelligent trial an error to find the good hyper-parameter such as γ, α, and the number of episodes. To do that we start we some values, and we adjust these parameter according to the obtained results.

B. Model evaluation

To evaluate the model, we run the the agents in the environment for 1000 episodes, then compute the completion rate, the reward average, and the averaged steps for each methods.

We also investigate these value for the training process to asses the speed and efficiency of each methods, in addition of the plot of the loss and accumulated rewards.

VII. Results and discussion

VIII. Conclusion