Solving cartpole using RL

Bolivar Enrique Solarte Pardo

ID: 106061858

June 12, 2019

1. Introduction

The present report refers to homework 4 for EE5510 System Theory course, 2019 spring semester. This assignment aims to solve a classic control problem, the Cartpole environment from OpenAi. This report is presented by the following sections: In section 2 environment details are introduced as well as the metric used for comparison. Then, three different solutions for the Cartpole environment are presented in section 3, 4, and 5. The algorithms used in this report are Tabular Q-learning, Policy Gradient and Actor-critic approaches. In section 6, a comparison between the proposed algorithms are presented, Experiments and Analysis. Finally, Conclusions are detailed in section 7.

2. Cartpole environment

The cartPole environment used in this report was deployed from OpenAi Gym[1] library. This environment presents a mobile agent with a joint pendulum. The agent moves along a frictionless track with the pendulum in balance. For each episode, the pendulum (pole) starts upright, then the goal is to prevent it from falling by taking action on the mobile agent. Such actions over the agent can be either ***pushing cart to the left or to the right.*** The environment responses with a state-observation. Such observation represents the cinematic variable-states of the agent. Table 1 shows these state obeservations associated with the Cartpole environment.

Table 1. Variables associated with the Cartpole environment

Tag Variable	Descriptions	Range
cart_pos	Horizontal Cart position	-2.4 to 2.4
cart_vel	Horizontal Cart velocity	-inf to inf
pole_pos	Angular pole position respect to a vertical ax	-41.8° to 41.8°
pole_vel	Angular pole velocity	-inf to inf
Action	Vector action [left, right]	1 or 0

2.1 Metric Qualifications

Each time that an action is commanded on the agent, the environment responds with a state observation which describes the position and velocity of the cart and the pole as well ([cart_pose, cart_vel, pole_pos, pole_vel] described in table 1). Additionally, the environment outputs a termination state. The termination state is triggered when either the pole angle is

[1] https://github.com/openai/gym/wiki/CartPole-v0.

over than ±12° or the car position is over than ±4.8. For each action where the terminal state has not been triggered, a reward of +1 is counted.
In order to evaluate the performances between the proposed algorithms, an average reward is evaluated. Additionally, each experiment is run by several episodes, thus each episode contents an average reward of this performance. Table 2 shows the metrics used for evaluation.

Table 2. Metric Evaluation

Metric	Tag variables
Average reward	Av_reward
Max. reward	max_reward
Min. reward	min_reward
Std reward	std_reward
Number of episodes	episodes

3. Q-Learning

In order to apply the Q-learning algorithm to the Cartpole environment is necessary to discretize the state observations introduced in table 1. Additionally, we need to create our state-action space, where the rewards associated with a particular state-action are stored.

In order to discretize the state observation of the environment, a subrange of each state is considered. Furthermore, each new range is also split into an specific number of partitions. Table 3 presents the number of discrete partitions and the subrange for each state observation.

Table 2. Discrete state observations

Metric	Subrange	Number of partitions
cart_pose	[-4.8 to 4.8]	100
cart_vel	[-1000 to 1000]	10000
pole_pos	[-10 to 10]	20
pole_vel	[-1000 to 1000]	10000

The implementation of this algorithm considers two classes, ***Class* QLearning()** and ***Class* CartPole()**. A general overview of these classes is presented in figures 3 and 4**.** For each state observation, a pre-computed array is created. These arrays will evaluate each state in order to discrete it. The pre-computed arrays are created in the initialization of the class Cartpole by **self.setting_discrete_observation().** For each observation, a state **S** is built by **self.build_state(current_observation)**, which is also a method for the same class Cartpole

```
def setting_discrete_observation(self, bins_pos=100, bins_vel=10000):
        '''
        Precomputed state observations
        '''
        self.discrete_cart_pos = np.linspace(-4.8, 4.8, bins_pos)
        self.discrete_cart_vel = np.linspace(-1000, -1000, bins_vel)
        self.discrete_pole_pos = np.linspace(-10, 10, 20)
        self.discrete_pole_vel = np.linspace(-1000, 1000, bins_vel)
```

Figure 1. Precomputed discrete state-observation

```
self.env = Carpole()...
...
current_observation = self.env.reset()
...
state = self.env.build_state(current_observation)


def build_state(self, observation):
    """
    Given an observation a state-code is built
    """
    cart_pos, cart_vel, pole_pos, pole_vel = observation

    cart_pos = self.__discretize(cart_pos, self.discrete_cart_pos)
    cart_vel = self.__discretize(cart_vel, self.discrete_cart_vel)
    pole_pos = self.__discretize(pole_pos, self.discrete_pole_pos)
    pole_vel = self.__discretize(pole_vel, self.discrete_pole_vel)

    features = [cart_pos, pole_pos, cart_vel, pole_vel]
    return int("".join(map(lambda feature: str(int(feature)), features)))

def __discretize(self, value, discrete_array):
    return np.digitize(x=[value], bins=discrete_array)[0]
```

Figure 2. Evaluation of a discrete state-observation

Figures 1 and 2 show the precomputed discrete states and the evaluation of a new state respectively. Lastly, a dictionary ***q{}*** is used to store each discrete state-action and the reward associated with it. This dictionary is also an instance of the ***class CartPole()*** and is evaluated in the methods ***def choose_action(self, state, epsilon)*** and ***def evaluate_state(self, state, action).*** The first one evaluates a state **S** in ***q{}*** and return the max action following an epsilon-greedy policy. The second one evaluates an specific state and action and returns the reward associated with them. The dictionary ***q{}*** is also evaluated in the method ***def update(self, state, action, reward, next_state)*** in the ***class QLearning()***, where the Q-learning algorithm computes its rewards and either store or update the dictionary ***q{}.*** The details implementation for the Q-Learning algorithm has not been included in this report. For details about Q-Learning implementation please refer to Github SystemTheory[2]

[2] https://github.com/EnriqueSolarte/SystemTheory

```
class CartPole:
    def __init__(self):

    def setting_discrete_observation(self, bins_pos=100, bins_vel=10000):

    def build_state(self, observation):

    def __discretize(self, value, discrete_array):

    def choose_action(self, state, epsilon):

    def evaluate_state(self, state, action):

    def step(self, action):

    def reset(self):

    def render(self):
```

Figure 3. A general overview of the class CartPole

```
class QLearning:
    def __init__(self, actions, epsilon, alpha, gamma, env):

    def run(self, args):

    def update(self, state, action, reward, next_state):
```

Figure 4. A general overview of the class QLearning

4. Policy Gradient (ANN)

In this section, a policy gradient method for an optimal policy is implemented. The method used for this purpose is a ***Monte Carlo Policy Gradient method using an*** Artificial Neural Network **ANN**. The idea behind this approach is to use a parametrized policy following a Monte Carlo exploration in the environment Cartpole.

The ANN architecture for this implementation has three fully connected hidden layers with a relu activation function. The neurons per layer are described as follows ***5-5-2***. The latter layer instead of a relu activation function a softmax function is evaluated. It is important to note the output of this **ANN** is a softmax function which later will be evaluated.

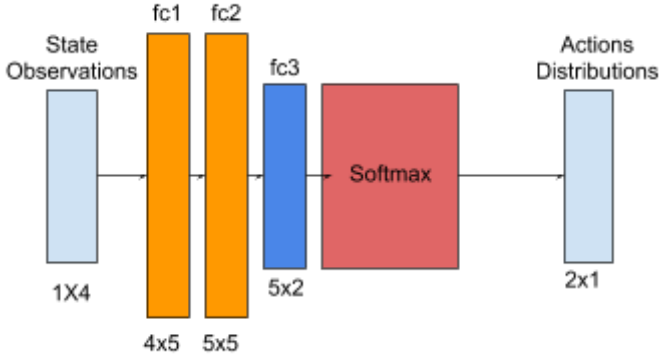

Figure 5. ANN architecture

```
Input_ = 4
Action_size =2

with tf.name_scope("fc1"):
    fc1 = tf.contrib.layers.fully_connected(inputs=input_,
                                            num_outputs=5,
                                            activation_fn=tf.nn.relu,
                                            weights_initializer=tf.contrib.layers.xavier_initializer())

with tf.name_scope("fc2"):
    fc2 = tf.contrib.layers.fully_connected(inputs=fc1,
                                            num_outputs=5,
                                            activation_fn=tf.nn.relu,
                                            weights_initializer=tf.contrib.layers.xavier_initializer())

with tf.name_scope("fc3"):
    fc3 = tf.contrib.layers.fully_connected(inputs=fc2,
                                            num_outputs=action_size,
                                            activation_fn=None,
                                            weights_initializer=tf.contrib.layers.xavier_initializer())

with tf.name_scope("softmax"):
    action_distribution = tf.nn.softmax(fc3)
```

Figure 6. Implementation of ANN architecture

This solution for the Cartpole environment begins with the architecture setting described in figure 6, which was also introduced in figure 5 graphically. Additionally, both ***loss function*** and the ***optimizer*** are also defined after the architecture.

```
with tf.name_scope("loss"):
    neg_log_prob = tf.nn.softmax_cross_entropy_with_logits_v2(logits=fc3, labels=actions)
    loss = tf.reduce_mean(neg_log_prob * discounted_episode_rewards_)

with tf.name_scope("train"):
    train_opt = tf.train.AdamOptimizer(learning_rate).minimize(loss)
```

Figure 7. Loss function and optimizer implementations

From figure 7 we can appreciate the ***neg_log_prob,*** which describe the evaluation for our policy output distribution is weighted by a ***discounted_episode_rewards_*** which is defined by the **Monte Carlo exploration.** This discounted episode rewards are evaluated by the function described in figure 8.

```
def discount_and_normalize_rewards(episode_rewards):
        discounted_episode_rewards = np.zeros_like(episode_rewards)
        cumulative = 0.0
        for i in reversed(range(len(episode_rewards))):
                cumulative = cumulative * gamma + episode_rewards[i]
                discounted_episode_rewards[i] = cumulative

        mean = np.mean(discounted_episode_rewards)
        std = np.std(discounted_episode_rewards)
        discounted_episode_rewards = (discounted_episode_rewards - mean) / (std)

        return discounted_episode_rewards
```

Figure 8. Loss function and optimizer implementations

5. Actor-Critic method

The idea behind the Actor-Critic method is to use the same previuos approach, policy gradient method but adding a state-action approximation as well. For such purpose, In this implementation two ANN are defined, one for the policy estimation (***Actor***), and another for the state-action function estimation (***Critic***).

This implementation defines a central class called **A2C().** Where the **ANNs** for the actor and the critic function are initialized with the class by the own methods ***def build_actor() and def build_critic().*** These methods are described in figure 10.

```
class A2C:
    def __init__(self, state_size, action_size):

    def build_actor(self):

    def build_critic(self):

    def get_action(self, state):

    def train_model(self, state, action, reward, next_state, done):
```

Figure 9. Central Actor-Critic class implementation

```
    def build_actor(self):
        actor = Sequential()
        actor.add(Dense(24, input_dim=self.state_size, activation='relu',
                        kernel_initializer='he_uniform'))
        actor.add(Dense(self.action_size, activation='softmax',
                        kernel_initializer='he_uniform'))
        actor.summary()
        actor.compile(loss='categorical_crossentropy',
                      optimizer=Adam(lr=self.actor_lr))
        return actor

    def build_critic(self):
        critic = Sequential()
        critic.add(Dense(24, input_dim=self.state_size, activation='relu',
                         kernel_initializer='he_uniform'))
        critic.add(Dense(self.value_size, activation='linear',
                         kernel_initializer='he_uniform'))
        critic.summary()
        critic.compile(loss="mse", optimizer=Adam(lr=self.critic_lr))
        return critic
```

Figure 10. Actor and Critic Architectures

After to build the architectures for the Actor and Critic agent, the training for such ANNs is trigger by the following sequence. (1) After either reset the environment (at the beginning) or take an action, a state vector (see table 1) is defined. This state vector is evaluated by ***def get_action(state)*** method. Where an action is sampled from the current distribution output of the Actor. (2) Then, the action is evaluated in the environment. Thus the environment outputs both a new state and a reward associated with such action. (3) With the previous

state, the new_state, and the reward, the method ***def train_model()*** is called. In this method, the **state and new_state** are evaluated in by the state-action model (**Critic**)**.** Then the correspondent reward estimations are used to compute a ***error and target value*** using the reward predicted by the environment. Figure 11 details the implementation of this procedure. Furthermore, figure 12 shows the implementation of the ***def train_model()*** method.

```
agent = A2C()

state = env.reset()                                                    (1)
While True:
        action = agent.get_action(state)                               (2)
        next_state, reward, done, info = env.step(action)
        agent.train_model(state, action, reward, next_state, done)     (3)
        If done:
                break
        state = next_state
```

Figure 11. General Actor and Critic algorithm

```
def train_model(self, state, action, reward, next_state, done):
    target = np.zeros((1, self.value_size))
    error = np.zeros((1, self.action_size))

    value = self.critic.predict(state)[0]
    next_value = self.critic.predict(next_state)[0]

    if done:
            error[0][action] = reward - value
            target[0][0] = reward
    else:
            error[0][action] = reward + self.discount_factor * (next_value) - value
            target[0][0] = reward + self.discount_factor * next_value

    self.actor.fit(state, error, epochs=1, verbose=0)
    self.critic.fit(state, target, epochs=1, verbose=0)
```

Figure 12. Train_model Method which feeds the inputs to the Actor and Critic ANN