Experimental Report

Introduction

Atari 2600 Pong is a game environment provided on the OpenAI “Gym” platform. Pong is a two-dimensional sport game that simulates table tennis which released it in 1972 by Atari. The player controls an in-game paddle by moving it vertically across the left or right side of the screen. They can compete against another player controlling a second paddle on the opposing side. Players use the paddles to hit a ball back and forth and have three action (“stay”, “down”, and “up”). The goal is for each player to reach 21 points before the opponent, points are earned when one fails to return the ball to the other. In this experiment , we use Policy Gradient method to train an agent to play the Pong-v0 game.

Data Set and Feature

In this environment, the observation is an RGB image of the screen, which is an array of shape $(210, 160, 3)$ Each action is repeatedly performed for a duration of k frames, where k is uniformly sampled from $\{2, 3, 4\}$. To simplify the train, we preprocesses the raw observation by cropping, downsampling, making it grayscale, erasing the background, and flattening the image to a one-dimensional vector.

Algorithm

There are 3 components in reinforcement learning: agent, environment and reward function. In reinforcement learning, the environment and reward function are not under our control. The environment and reward function are given in advance before we start learning. The only thing we can do is to adjust the policy inside the agent so that the agent can get the maximum reward. There is a policy in the agent, and this policy determines the actor's behavior. Given an input, the policy outputs the action the agent should perform.

Policy π is a function(In our code, it is a network) with parameter θ, the input is the observation of machine represented as a vector or a matrix, and the output is each action corresponds to a neuron in output layer.

A game is called an episode. Adding up all the rewards obtained in this game is the $totalreward$, we call it $return$, and denote it by R.
The agent has to figure out a way to maximize the reward it can get.

In a game, we put the s output by the environment and the behavior a output by the agent, and set the Trajectory.

$$Trajectory\tau = \{\mathrm{s}_1, \mathrm{a}_1, \ \mathrm{s}_2, \mathrm{a}_2, \cdots, \mathrm{s}_\mathrm{t}, \mathrm{a}_\mathrm{t}\}$$

so the possibility is

$$\begin{aligned} \mathrm{p}_\theta(\tau) &= \mathrm{p}\left(\mathrm{s}_1\right)\mathrm{p}_\theta\left(\mathrm{a}_1 \mid \mathrm{s}_1\right)\mathrm{p}\left(\mathrm{s}_2 \mid \mathrm{s}_1, \mathrm{a}_1\right)\mathrm{p}_\theta\left(\mathrm{a}_2 \mid \mathrm{s}_2\right)\mathrm{p}\left(\mathrm{s}_3 \mid \mathrm{s}_2, \mathrm{a}_2\right)\cdots \\ &= \mathrm{p}\left(\mathrm{s}_1\right)\prod_{\mathrm{t}=1}^{\mathrm{T}}\mathrm{p}_\theta\left(\mathrm{a}_\mathrm{t} \mid \mathrm{s}_\mathrm{t}\right)\mathrm{p}\left(\mathrm{s}_{\mathrm{t}+1} \mid \mathrm{s}_\mathrm{t}, \mathrm{a}_\mathrm{t}\right) \end{aligned}$$

The Expected Reward function is
$\bar{R}_\theta = \sum_\tau R(\tau)p_\theta(\tau) = E_{\tau\sim p_\theta(\tau)}[R(\tau)]$

In a certain game, in a certain round, we will get reward. All we have to do is to adjust the parameter θ inside the agent. The larger the value of reward, the better the agent is.

We're using gradient ascent method, because we want to make it as big as possible. Gradient ascent is added when updating parameters. To do gradient ascent, we first compute the gradient of the expected reward.

$$\begin{aligned}\nabla \bar{R}_\theta &= \sum R(\tau)\nabla p_\theta(\tau) = \sum R(\tau)p_\theta(\tau)\frac{\nabla p_\theta(\tau)}{p_\theta(\tau)} \\ &= \sum R(\tau)p_\theta(\tau)\nabla \log p_\theta(\tau) = E_{\tau\sim p_\theta(\tau)}\left[R(\tau)\nabla \log p_\theta(\tau)\right]\end{aligned}$$

Next we use sample method to estimate expectations

$$\begin{aligned}\mathrm{E}_{\tau\sim \mathrm{p}\theta(\tau)}\left[\mathrm{R}(\tau)\nabla \log \mathrm{p}_\theta(\tau)\right] &\approx \frac{1}{\mathrm{N}}\sum_{\mathrm{n}=1}^{\mathrm{N}}\mathrm{R}\left(\tau^{\mathrm{n}}\right)\nabla \log \mathrm{p}_\theta\left(\tau^{\mathrm{n}}\right) \\ &= \frac{1}{\mathrm{N}}\sum_{\mathrm{n}=1}^{\mathrm{N}}\sum_{\mathrm{t}=1}^{\mathrm{T_n}}\mathrm{R}\left(\tau^{\mathrm{n}}\right)\nabla \log \mathrm{p}_\theta\left(\mathrm{a}_{\mathrm{t}}^{\mathrm{n}} \mid \mathrm{s}_{\mathrm{t}}^{\mathrm{n}}\right)\end{aligned}$$

$$\begin{aligned}\nabla \log \mathrm{p}_\theta(\tau) =& \nabla\left(\log \mathrm{p}\left(\mathrm{s}_1\right)+\sum_{\mathrm{t}=1}^{\mathrm{T}}\log \mathrm{p}_\theta\left(\mathrm{a}_{\mathrm{t}} \mid \mathrm{s}_{\mathrm{t}}\right)+\sum_{\mathrm{t}=1}^{\mathrm{T}}\log \mathrm{p}\left(\mathrm{s}_{\mathrm{t}+1} \mid \mathrm{s}_{\mathrm{t}}, \mathrm{a}_{\mathrm{t}}\right)\right) \\ =& \nabla \log \mathrm{p}\left(\mathrm{s}_1\right)+\nabla\sum_{\mathrm{t}=1}^{\mathrm{T}}\log \mathrm{p}_\theta\left(\mathrm{a}_{\mathrm{t}} \mid \mathrm{s}_{\mathrm{t}}\right)+\nabla\sum_{\mathrm{t}=1}^{\mathrm{T}}\log \mathrm{p}\left(\mathrm{s}_{\mathrm{t}+1} \mid \mathrm{s}_{\mathrm{t}}, \mathrm{a}_{\mathrm{t}}\right) \\ =& \nabla\sum_{\mathrm{t}=1}^{\mathrm{T}}\log \mathrm{p}_\theta\left(\mathrm{a}_{\mathrm{t}} \mid \mathrm{s}_{\mathrm{t}}\right) \\ =& \sum_{\mathrm{t}=1}^{\mathrm{T}}\nabla \log \mathrm{p}_\theta\left(\mathrm{a}_{\mathrm{t}} \mid \mathrm{s}_{\mathrm{t}}\right)\end{aligned}$$

Suppose we do s_t at a_t, and the final reward is positive, then we have to increase the probability of this item , and if we do s_t at a_t, and the final reward is negative, then we have to decrease the probability of this item.

In a word , the implementation is

$$\begin{aligned}&\theta \leftarrow \theta+\eta \nabla \bar{R}_\theta \\ &\nabla \bar{R}_\theta=\frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T_n}R\left(\tau^n\right)\nabla \log p_\theta\left(a_t^n \mid s_t^n\right)\end{aligned}$$

Code

In our code, we will create a policy network that will take raw pixels of our observation from the Pong-v0 in OpenAi Gym as the input. The policy network π_θ is a single hidden layer neural network fully connected to the raw pixels of Pong-v0 at the input layer and also to the output layer containing a single node returning the probability of the board going up.

We initialize a random policy.For the given current policy, we will collect different sample trajectories by running a single episode of the Pong game and store the trajectories in that episode.

Now we just need to maximize $\sum_i A_i * \log p\left(y_i \mid x_i\right)$. And We call A_i the advantage, it's a number, like $+1$ or -1 based on how this action eventually turned out in the Pong-v0 game reward.

The Details of File

1. main.py

- import the important dependencies
- model parameters
- model train function
- user-interface

2. func.py
   - sigmoid
   - pre_process_img :simplify the image from Pong-v0
   - discount_rewards : compute the discount rewards
3. my_model.py
   - our_pong_model : model forward, back_propagation ,save_model

The Details of Code

1. Parameters

```
INPUT_SIZE = 75*80  # input picture size
HD_N = 200  # hiden layer size
BATCH_SIZE = 10 # train batch size
LR = 1e-3  # learning rate
GAMMA = 0.99 # discount of reward
DECAY_RATE = 0.99 # deep_learning
TIME_TO_SAVE = 10 # save model
FROM_FILE = True  # load model
FILE_NAME = None
SHOW = False
CHECKPOINTS_PATH = 'checkpoints/'  # the path of checkpoints
REWARD_RECORD = 'Reward_record.csv' # the reward record
TIME_RECORD = 'Time_record.csv' # the time record
```

2. Build environment

```
env = gym.make('Pong-v0')
# play vedio
env = wrappers.Monitor(env, 'tmp/pong-base', force=True)
```

3. Initialize train functioin

```
def train:
    observation = env.reset()

    # prev: in order to compute the difference between current and previous
frame.
    # So we can capture the motion.
    # At first, prev = None ,because there's no previous frame at the beginning
of the game.

    prev_states = None

    episode_hidden_layer_values, episode_observations, episode_gradient_log_ps,
episode_rewards = [], [], [], []
    running_reward = None

    state_list, hidden_list, action_weight_list, reward_list = [], [], [], []
    accumulated_reward = None
```

```
    reward_sum = 0
    print('start training:\n')

    # start time
    start_time = time.time()
```

4. Preprocess image

```
# input: the raw image pixels
# preprocesses by cropping, downsampling,
# making it grayscale, erasing the background,
# and flattening the image to a one-dimensional vector.
def pre_process_img(img):
```

5. Play the Pong-v0 game

```
while True:
    if SHOW:
        env.render()

    cur_states = func.pre_process_img(observation)

    # input the motion of the Pong
    x = cur_states - prev_states if prev_states is not None else
np.zeros(INPUT_SIZE)
    prev_states = cur_states

    # Forward the network and get the action
    prob, hidden = model.forward(x)

    # At the start, randomly pick a step
    action = 2 if np.random.uniform() < prob else 3

    state_list.append(x)
    hidden_list.append(hidden)

    # fake label
    y = 1 if action == 2 else 0

    # action_weight_list : derivative of the loss
    action_weight_list.append(y - prob)

    # Step in the environment:
    observation, reward, done, info = env.step(action)
    reward_sum += reward
    reward_list.append(reward)

    # one game is over
    if done:
        episode += 1

        states_matrix = np.vstack(state_list)
        hidden_matrix = np.vstack(hidden_list)
        action_weight_matrix = np.vstack(action_weight_list)
```

```
        reward_matrix = np.vstack(reward_list)

        state_list, hidden_list, action_weight_list, reward_list = [], [], [],
[]  # empty the arrays

        # Compute the discounted reward and standalize it
        discounted_rewards = func.discount_rewards(reward_matrix, GAMMA)
        discounted_rewards -= np.mean(discounted_rewards)
        discounted_rewards /= np.std(discounted_rewards)

        action_weight_matrix *= discounted_rewards
        model.store_gradient(hidden_matrix, states_matrix, action_weight_matrix)

        # every batch size, we apply back_propagation
        if episode % BATCH_SIZE == 0:
            model.back_propagation()

        accumulated_reward = reward_sum if accumulated_reward is None else
accumulated_reward * 0.99 + reward_sum * 0.01

        # save train time
        now = time.time()
        gap = now - start_time

        print('episode:', episode)
        print('resetting env. episode reward total was %f. running mean: %f and
time spent:%f' % (reward_sum, accumulated_reward, gap))
        with open(REWARD_RECORD, 'a', encoding='UTF-8') as f:
            f.write(str(episode)+','+str(reward_sum)+'\n')

        with open(TIME_RECORD, 'a', encoding='UTF-8') as f:
            f.write(str(episode)+','+str(gap)+'\n')


        # Reset the game , and replay
        reward_sum = 0
        observation = env.reset()  # reset env
        prev_states = None # reset prev_frames

        # save model
        if episode % TIME_TO_SAVE == 0:

 model.save_model(CHECKPOINTS_PATH+'save_'+str(int(episode/TIME_TO_SAVE))+'0.p')
```

5. Neural Network

```
#  sigmoid function
def sigmoid(x):
    return 1.0 / (1.0 + np.exp(-x))

#  Network forward
def forward(self, x):
    # input data, with type of input_shape
    # output : the probability of taking up action, the state of the first layer
```

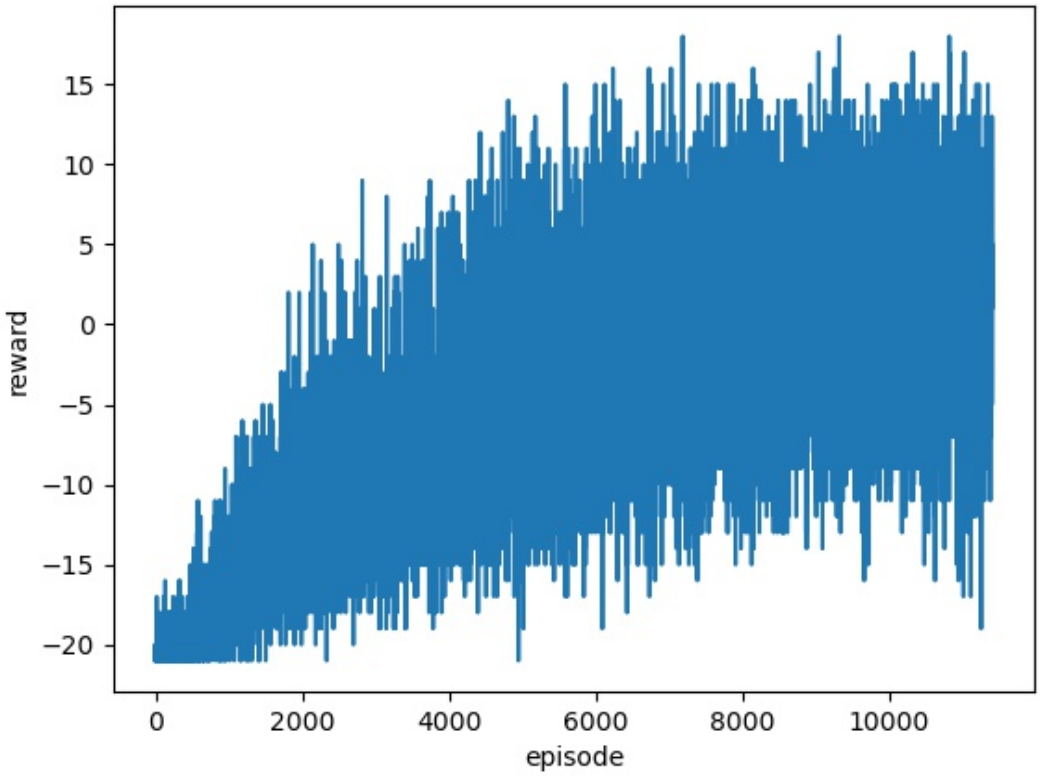
```
    y_1 = np.dot(self.param['w1'], x)
    y_1[y_1 < 0] = 0  # Here we manually call ReLU
    y_2 = np.dot(self.param['w2'], y_1)
    p = func.sigmoid(y_2)
    return p, y_1

#  Network backward
def store_gradient(self, ep_y1, ep_x, ep_dy2):
    dw2 = np.dot(ep_y1.T, ep_dy2).ravel()
    d_y1 = np.outer(ep_dy2, self.param['w2'])
    d_y1[ep_y1 <= 0] = 0  # bp PReLU
    dw1 = np.dot(d_y1.T, ep_x)

    self.grad_buffer['w1'] += dw1
    self.grad_buffer['w2'] += dw2

def back_propagation(self):
    for k, v in self.param.items():
        grad = self.grad_buffer[k]
        self.rmsprop_cache[k] = self.decay_rate * self.rmsprop_cache[k] + (1 -
self.decay_rate) * grad ** 2
        self.param[k] += self.learning_rate * grad /
(np.sqrt(self.rmsprop_cache[k]) + 1e-5)
        self.grad_buffer[k] = np.zeros_like(v)
```

6. Discount Rewards

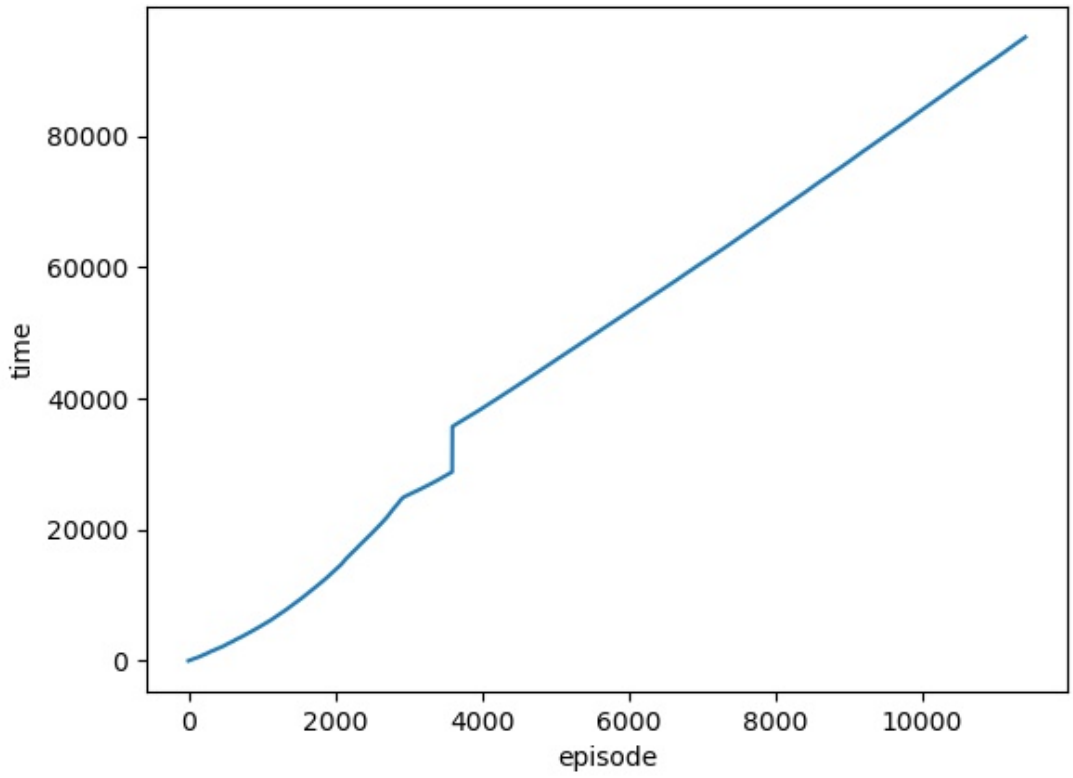
```
def discount_rewards(r, gamma):
    # by discount factor , we compute the discount rewards by orign reward
    discounted_r = np.zeros_like(r)
    sum_up = 0

    for t in reversed(range(0, r.size)):
        if r[t] != 0:
            sum_up = 0
        sum_up = sum_up * gamma + r[t]
        discounted_r[t] = sum_up

    return discounted_r
```

The Analysis of The Results

Learning Curve Plot



As the number of training episode increases, the agent's score continue to increase.



This is the training time figure.

Model train

The final agent that played this game was trained for 31000 episodes on a Y7000 with 2.6GHz i7 9750H.

We tested 10 games with the final model , the final score is fine.

金小龙 邓一川 李卓