Simulation and Optimal Decision-Generating of *HPJY* Gashapon Machine based on Value Iteration

Chuhuan Huang
March 2021

Preface

HPJY/ Game for Peace, a phenomenal mobile FPS-like game, unveiled a multi-party coop with a new-established girl idol group, BonBon Girls 303, Aston Martin and AIRBUS in the eve of Chinese New Year.

In this coop, customers could “gamble” with probability through a mechanism similar to Gashapon machine for several rewards such as fashioned and shinning outfits for role in game, new pinky appearance for M416 automatic rifle, Aston Martin DBS skin for the cars in battlefield, and AIRBUS helicopter for presentation.

(picture from Game for Peace official account of Wechat)

As a showing-off person, I immerged into this “gamble” without hesitation. Luckily, I managed to achieve the jackpot, 3 pieces of token redeemable for the goldened Aston Martin, with merely several rounds of trial. Trials cost me merely 150RMB while the goldened Aston Martin worth of at least 2500RMB at that time. Standing before this humongous profit rate, I was thinking of the chance of profiting if my “luck” is reproducible.

What drives me more determined in the chance of profiting was I managed to acquire another 2 tokens with less than 300RMB, right after the day I hit that 3 pieces. Chance becomes pattern if it is reproducible. I believed there is a pattern, a decision-making mechanism, that will at last lead me to a profit end.

Given finite situations, solid probability distribution, and finite possible operations, I transfer this game-contexted Gashapon machine into a context-free finite markov decision process and our story starts here.

1. ***Define the elements in M.D.P.***: *State Space, Action Space*

Firstly and the most obviously is to define the operations or *actions*. There 4 actions during each round of trial, namely *Start*, *Unprotected Trial*, *Protected Trial*, and *Collect&Exit*, and for the purpose of facilitating programing, they are named action 1, 2, 3, and 4 respectively.

That is, Action Space $\boldsymbol{A} = \{1, 2, 3, 4\}$

Then, the States and State Space is what we need to address. Apparently, there are 8 stages, signified by the number of stars showed in user interface, namely 0, 1, 2, 3, 4, 5, 6, 7.

In fact, by sticking to the rules of this Gashapon Machine, it is possible to step-up from "6" to not only "7" but also "8" and "9" by transition step sized 2 and 3 respectively. Thus, 8 and 9 must be contained in State Space. Similar, a 2-sized step-down could happen at "1", so "-1" should be taken into consideration.

We call -1, 0, ……, 9 the *norm* of the states, call the transition from lower- normed states to higher-normed states a *step-up*, and a *step-down* in the other direction, with *transition step* sized 1,2,3 upward and -1, -2 downwards.

From the rules, we know there is a difference in treatment if we apply protected trial 3 times in a row. Therefore, we need to count the times of applying protected trial, either 0, 1, or 2 as the counts will be recalibrated to zero by rules.

We call 0, 1, 2 the *count* of the states.

Intuitively, we need separate the step-up states and step-down states as, by rule, step-up can do action either 2, 3 and 4 while the step-down can do merely collect&exit. For the purpose of facilitating programing, we use 0, 1 as the *sign* or signature of the states, with 0 representing a step-up originating and 1 a step-down originating.

Therefore, our states are 3-tuple: (norm, count, sign) and State Space $\boldsymbol{S}$ is as following

(in hpjy.py)

```
# State Space S: s = (norm, count, sign)
S = {(-1, 0, 1),
     (0, 0, 0), (0, 0, 1),
     (1, 0, 0), (1, 1, 0), (1, 2, 0), (1, 0, 1),
     (2, 0, 0), (2, 1, 0), (2, 2, 0), (2, 0, 1),
     (3, 0, 0), (3, 1, 0), (3, 2, 0), (3, 0, 1),
     (4, 0, 0), (4, 1, 0), (4, 2, 0), (4, 0, 1),
     (5, 0, 0), (5, 1, 0), (5, 2, 0), (5, 0, 1),
     (6, 0, 0), (6, 1, 0), (6, 2, 0),
     (7, 0, 0),
     (8, 0, 0),
     (9, 0, 0)}
```

2. ***Formal Translation of the Rules***

With State Space and Action Space defined, let us translate the rules into formal language.

1) At start state, Action 1 and only Action 1 can lead to a new state with higher norm, or a step-up. Other actions result in nothing.

2) Every step-up has transition step sized in 1, 2, or 3, with probability 0.82, 0.17, and 0.01 respectively. While every step-down sized in -1, -2 with probability 0.75 and 0.25 respectively.

3) Step-up has probability 0.2 and step-down has probability 0.8 under the action 2.

4) Under protected trial/ action 3, there are two cases.

i) when the count of state is less than 2, step-down has probability 0. Instead, *translation* has probability 0.8 under action 3. A state transition is called a translation if the norm does not change during transition, e.g. (3,0,0) → (3,1,0). Still, Step-up has probability 0.2.

ii) when the count of state is 2, a step-up is secured, namely with probability 1, and its step size follows 2)

the cost of protected trial/ action 3 refers to COST_ARR

5) action 4 operated on any state leads to immediate reward and the terminal state, which coincides with the start state (0, 0, 0). And the rewards distribution refers to REWARD_ARR and the norm of the state. Specially, (0, 0, 1) and (-1, 0, 1) is applied with subsistence allowance.

3. ***State Transition Function and Rewards Function***

Based on State Space, Action Space and the rules, we defined our state-transition probability function and reward function. (in hpjy.py)

```python
# state transition probability function p(s'|s, a), defined in S x S x A ---> [0,1]
# the probability of state s transit to s' following the action a, sp - s prime state, s - state, a - action
def prob(sp, s, a):
    """..."""
    if sp not in S or s not in S or a not in A:
        # filter out invalid input variables
        return 0.0
    # starting position has only one acceptable action : start / action a = 1
    if s == (0, 0, 0):
        return STEP_UP_DIST[sp[0]] * int(0 <= sp[0] <= 3 and sp[1] == 0 == sp[2] and a == 1)
    # pre-terminal state has only one acceptable action : collect&exit / action a = 4
    elif s == (7, 0, 0) or s == (8, 0, 0) or s == (9, 0, 0) or s[2] == 1:
        return int(sp == (0, 0, 0) and a == 4)
    ...
    else:
        ...
        d1 = sp[0] - s[0]
        d2 = sp[1] - s[1]
        # 平A in s state
        if a == 2:
            return Be_DIST[0] * STEP_UP_DIST[d1] * int(0 < d1 <= 3 and sp[1] == 0 == sp[2]) \
                   + Be_DIST[1] * STEP_DOWN_DIST[-d1] * int(d1 < 0 and sp[1] == 0 and sp[2] == 1)

        ...
        elif a == 3:
            return STEP_UP_DIST[d1] * int(s[1] == 2 and 0 < d1 <= 3 and sp[1] == 0 == sp[2]) + \
                   Be_DIST[1] * int(0 <= s[1] < 2 and d1 == 0 == sp[2] and d2 == 1) + \
                   Be_DIST[0] * STEP_UP_DIST[d1] * int(0 <= s[1] < 2 and 0 < d1 <= 3 and sp[1] == 0 == sp[2])
        # 直接领取
        elif a == 4:
            return int(sp == (0, 0, 0))
        else:
            return 0.00
```

```python
# reward function: reward of action a from state s to state sp, defined in S x S x A ---> R (set of Real numbers)
# notice that there is no instant reward for step-up. that is,prize are postponed till exit/action 4
# while step-down is followed by a solely-acceptable state represented in third dimension,
# immediate reward (action 4) and going-back to origin
# e.g. (6, 1, 0) -> (5, 0, 1) -> (0, 0, 0)
def rewards(sp, s, a):
    """

    :rtype: float
    """
    # in this function by using filter above we only need to consider possible rewards
    # impossible ones have been filtered
    if prob(sp, s, a) == 0:
        return 0.0
    # start the game with an entry fee
    if s == (0, 0, 0):
        return - COST_ARR[0]
    # subsistence allowance case/ 碎星吃低保
    # the only two cases we need back trace
    elif s == (0, 0, 1):
        return 4 * unit
    elif s == (-1, 0, 1):
        return 2 * unit
    else:
        # a == 2 平A free of charge
        # a == 3 保护追加 cost is defined in COST_ARR
        # a == 4 直接领取 回到起点
        return 0 * unit * int(a == 2) - COST_ARR[s[0]] * int(a == 3) + REWARD_ARR[s[0]] * unit * int(a == 4)
```

4. ***Optimal Policy learning and generating***: *Value Iteration*

Recall that our goal is to find a decision-making mechanism, or a *policy*, to maximize our play profits in a round. A round, or a *trajectory*, is a series of states and actions alternating, starting from start state (0, 0, 0) and ending in the terminal state (0, 0, 0). Our profit is measured in expected reward collected through the trajectory, and we will use Value Iteration to generate the policy by which maximum expected return is achieved.

Firstly, we setup the *State Value function* of the state s, the expected reward from state s, and initialize this function with all zero output. In codes, we used dictionary to simulate this functionality: key is the state and value is the value. Value[s] returns the Value of the state s.
(in hpjy.py)

```
Value = dict()
# the dictionary that represent the value function
```

```
# initialize the value function, which here is implemented as dictionary
def value_function_init():
    for s in S:
        Value[s] = 0.0
    return
```

Besides, we define our state-action value function Q/q, representing the expected rewards from state s by action a. (in hpjy.py)

```
# state-action value function/ q function
# represents the expected reward from state s, by action a
def q(s, a):
    M = 0.0
    for sp in S:
        M = M + prob(sp, s, a) * (rewards(sp, s, a) + Value[sp])
    return M
```

Then we deploy the Value Iteration, in which we apply *Bellman optimality equation* for update the Value[s], where threshold is a constant we set for the accuracy of the policy. (in hpjy.py)

```
# Value Iteration
# Reinforcement Learning: An introduction, 2nd edition, Richard S. Sutton and Andrew G. Barto
# Page 83
def value_iteration():
    delta = 10
    while delta >= threshold:
        delta = 0
        for s in S - {(0, 0, 0)}:
            v = Value[s]
            Value[s] = max_q(s)
            delta = max(delta, abs(Value[s] - v))
    return
```

Then we calculate for any state s, policy(s) = argmax $_a$ q(s, a) as V*(s) = max $_a$ Q*(s, a)

Concrete implementation following (in hpjy.py)

```
# Optimized Policy generation using Value Iteration
# policy is a function defined as S ---> A
# here we applied Value Iteration to acquire a deterministic policy that optimize the total reward
# we will consider stochastic policy generation after this part is finished
def policy(state):
    value_function_init()
    value_iteration()
    action = 0
    if state == (0, 0, 0):
        return 1
    for a in A:
        if q(state, a) == max_q(state):
            action = a
    return action
```

Note: we postpone the reason of setting policy((0, 0, 0)) = 1until the end of simulation explanation.

5. ***Simulation, Contrast, and Reflection***

As I said in preface, I dreamed of high expectation. Thus, when I calculated the q((0, 0, 0),1) to see the expected reward (we not using Value[(0, 0, 0)]) as we set it as zero for the termination of the value iteration) the result, `-1.44428327969`,surprises me because it indicates each round of play we expect a loss of 1.44 RMB, even by the optimal policy.

To levitate my shock, I set 3 contrast policies and see if the learned policy is indeed optimal and my code does work. (in test.py)

```
def all_random_policy(s):
    if s not in S:
        return
    if s == (0, 0, 0):
        return 1
    elif s[0] >= 7 or s[2] == 1:
        return 4
    else:
        return np.random.choice([2, 3, 4], 1)[0]
```

```
def chh_policy_1(s):
    if s not in S:
        return
    if s == (0, 0, 0):
        return 1
    elif s[0] >= 6 or s[2] == 1:
        return 4
    elif s[0] == 2 or s[0] == 3:
        return 3
    else:
        return 2
```

```
def chh_policy_2(s):
    if s not in S:
        return
    if s == (0, 0, 0):
        return 1
    elif s[0] >= 6 or s[2] == 1 or s[0] == 4:
        return 4
    elif s[0] == 2 or s[0] == 3:
        return 3
    else:
        return 2
```

The contrast policies consist of an all-random policy, random based on availability of action, and two policies by which I indeed made huge profit. The simulation plays 100000 rounds of the game and print an average reward for each policy. (in test.py)

```
def simulation(iterations):
    R = 0
    # for i in range(1, 1001):
    for i in range(iterations):
        s = (0, 0, 0)
        a = policy(s)
        sp = random_state_generator(s, a)
        R = R + rewards(sp, s, a)
        # print(sp, s, a)
        while sp != (0, 0, 0):
            s = sp
            a = policy(s)
            sp = random_state_generator(s, a)
            # print(sp, s, a)
            R = R + rewards(sp, s, a)
    print(R / k, "learned_policy")
    return
```

```
def simulation_all_random_policy(iterations):
    R = 0
    # for i in range(1, 1001):
    for i in range(iterations):
        s = (0, 0, 0)
        a = all_random_policy(s)
        sp = random_state_generator(s, a)
        R = R + rewards(sp, s, a)
        # print(sp, s, a)
        while sp != (0, 0, 0):
            s = sp
            a = all_random_policy(s)
            sp = random_state_generator(s, a)
            # print(sp, s, a)
            R = R + rewards(sp, s, a)
        # print(R)
    print(R / k, "all_random_policy")
    return
```

```
def simulation_chh_policy_1(iterations):
    R = 0
    # for i in range(1, 1001):
    for i in range(iterations):
        s = (0, 0, 0)
        a = chh_policy_1(s)
        sp = random_state_generator(s, a)
        R = R + rewards(sp, s, a)
        # print(sp, s, a)
        while sp != (0, 0, 0):
            s = sp
            a = chh_policy_1(s)
            sp = random_state_generator(s, a)
            # print(sp, s, a)
            R = R + rewards(sp, s, a)
        # print(R)
    print(R / k, "chh_policy_1")
    return
```

```
def simulation_chh_policy_2(iterations):
    R = 0
    # for i in range(1, 1001):
    for i in range(iterations):
        s = (0, 0, 0)
        a = chh_policy_2(s)
        sp = random_state_generator(s, a)
        R = R + rewards(sp, s, a)
        # print(sp, s, a)
        while sp != (0, 0, 0):
            s = sp
            a = chh_policy_2(s)
            sp = random_state_generator(s, a)
            # print(sp, s, a)
            R = R + rewards(sp, s, a)
        # print(R)
    print(R / k, "chh_policy_2")
    return
```

And the results are also surprising:

```
(-8.558000277777577, 'all_random_policy')
(-17.196806111114594, 'chh_policy_1')
(-19.094634861108233, 'chh_policy_2')
```

None of them makes positive profit. This situation reminds me an ancient saying goes that "gambling for long time leads to definite loss".

Now we trace back to the policy() where I set policy((0, 0, 0)) = 1, because indeed q((0, 0, 0), a) = 0 for a = 2, 3, 4 as those action will do nothing on start state and no transition will happen. However, 0 is greater than that -1.44 = q((0, 0, 0), 1) and the optimal policy shall choose a that max q(s, a). That is, the learned optimal learned to do nothing facing the start state. "You lose nothing if you don't gamble" strikes to my mind.

Epilogue

This simulation is just a simple application of the model-based reinforcement learning algorithm, but it indeed inspired me some thought regarding dancing with the probability and the potential of reinforcement learning. Hopefully, in the future, there will be a series of powerful self-learning machine able to sort the human knowledge base and expand it in inhuman rate. At that day, the question whether the God throw the dice or not should be answered.

Reference

1. Sutton, Richard S., and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press Ltd, 2018.

2. Xipeng Qiu, *Neutral Networks and Deep Learning*, 2020, https://nndl.github.io/

3. Game for Peace, BonBon Girls 303, 2021,

https://gp.qq.com/act/a20210211five/five.html?orientation=1

Appendix

1. https://github.com/chh172/hpjy-gambling-simulation