Homework 1: Building a Poker Agent

Arya Farahi

01/XX/2017

1 Characterizing the Problem

Note that in the following I do not assume being folded or not is an state of the environment. It is easy to assume that player who have folded are not part of the game anymore and they are not included in the counting of the number of possible states. Within this framework agents does no have an internal state need to be counted, or taken into account, so the number of agents is an variable which changes through out the game.

Specify a game of poker in terms of each element of the PEAS description. Assume that the agent just plays poker, and does not need to move between tables, socialize, order drinks, etc. Explain in 1 or 2 sentences each.

PEAS referrers to

P: Performance measure - In this case we can measure the performance of each agent by the amount of bet it wins more money the agent wins per game it has a better performance or if loos more money it would have the worst performance. Another performance measure would be whether the player (agent) win the game or loose it, so it would be a binary kind of performance 1, if wins, or 0, if looses.

E: Environment - the environment consist of tables, hands (2 cards in-front of each player), a deck of cards and chips. These are the elements that the agents interact with.

A: Actuators - this the function which tell the agent to call, raise, or fold. These are the only three option the agent has. In case of human in the real word it would be his hands and mouth. In case of a computer it can be a monitor which shows the decision.

S: Sensors - it would be a tool which allows the agent see the cards and chips on the table. In case of human agent in real world it would be his/her eyes. In case of computer it can be a camera which can see the table.

Characterize a poker game on each of the following axes, Justify each answer with one or two sentences:

Fully vs. Partially Observable : It is *partially observable*. Though all possible states of the game (environment) is known in advance, the agent do not know the state at the time of the game, what cards are in the opponents hands. Each agent is aware the cards in its own hand and the cards on the table, but not the hand of other agents. Though at the end of each hand the state become fully observable, but unfortunately at that point it is not going to help the agent to win the game.

Single vs. Multi Agent: *Multi Agent*, because there are say 5 agents are playing with one another. Each one of players would be 1 agent, so in case of 5 players there are 5 agents.

Stochastic vs. Deterministic: *Stochastic*, because the next state of the system is not known before the card revelation, and it can not be predicted by the action of agents.

Episodic vs. Sequential: It would be *Sequential*. Because the agent sees the action of other players before it plays and it play accordingly. As the dealer continue the card revelation the decision of agent may changes accordingly. It builds it decisions based on the other players played and the cards are shown so far.

Static vs. Dynamic: It is *statics*, because in each round of betting the cards and hands of each player does not change. Basically the environment state does not change based on players action. Note that even though one may hold, we can assume that his action would be hold and we assume the state of system did not change but based on the rules that agent can not win the bet.

Discrete vs. Continuous: *Discrete*, because there are 52 Discrete cards, and the cards on the table (states) and agent's cards consist of these 52 cards.

Known vs. Unknown: *known*, because the laws of the game is known in advance. Basically, the agents know who will win the bet at the end of each game. Because the states are not fully observable though they do not know who is the winner during each hand, until the very end. This is why the game is fun.

2 Rationality

Definition of Rational Agent: For each possible percept sequence, a rational agent should select an action that *maximizes* its *performance measure* (in expectation) given the evidence provided by the *percept sequence* and whatever *built-in knowledge* the agent has.

After observing a particular agent for a bit, we notice it sometimes seems to make the wrong decision. For example, it folded on a $[2\clubsuit, 4\clubsuit]$ but the communal cards turned out to be $[3\clubsuit, 9\diamondsuit, jack\spadesuit, 5\clubsuit, 6\clubsuit]$ giving our agent a straight flush and what would have been the win that round! What can we say about the rationality of our agent? Explain your answer.

The provided example is a very rare outcome. Still the agent may perform well in the expected sense, according to the designer's performance measure. For ensemble of random scenarios the expectation may be maximized, though in some occasions it may loose, as expected.

Due to the stochastic and partially observable environment, one can not design an agent which can win all possible scenarios. Though because all states of the environment and rules are known in advance one can calculate the probability of winning and play accordingly.

At lease the provided example is a very rare outcome with a very low probability so the agent may not expect such a rare outcome, and occasionally it may loose. Based on only this observation and without knowing the designer's performance measure we can *not* conclude about the rationality of the agent.

Consider a hypothetical agent that decides whether to fold or call based solely on the value of the largest card in its pair. Is this agent rational? Why or why not?

It is not rational, because it does not maximize its expected performance measure. Let's assume that the number of winning over the total number of game is the performance measure that the designer had in mind.

For example where there are two players and the cards are for example $[3\clubsuit, jack\diamondsuit]$ it call while it should fold, and for the case where its cards are $[5\clubsuit, 5\diamondsuit]$ it folds while it should call. So, this agent is not rational. We will examine this in the next section when we are looking at the performance measures.

Because the state of the environment is partially observable then random play would not be the best strategy under any scenario. This partial information is useful and the agent is not using this partial information.

Betting different amounts is an important part of playing poker and can be advantageous for a player. Our agent doesn't have actuators that allow it to do things like go "all-in" on a bluff to scare the other players into folding. With this in mind, could our agent still be considered rational? Explain your answer.

Yes, let's assume that the agent does not know the strategy of other agents and can not learn/model it, as will see why in the section 5. Then the agent can maximize the net winning according to the probability rules, which we will practice in section 4. If the performance that the designer had in mind is the net winning under this simplified model, then yes, the agent can be rational and we will see how we can achieve it.

Note that the expected value of the performance is maximized but it does not guarantee that the agent will win the game. Maybe one other agent go "all-in" and the our rational agent call but loose that hand. Then it ran out of money and can not play anymore. But it does not contradict the rationality of the agent.

3 Simple Reflex Player

Note that under this part of the problem I did not assume ante, or in another word I assumed it is zero, which is different from the next section.

Using functions in the homework, write condition-action rules as pseudo-code for the following simple reflex agent designs:

1. **Agent2 : An agent who calls on pairs or if the cards are the same suit and folds otherwise.**

   initialization;
   if *pair()* || *same_suit()* **then**
   | **return** call;
   else
   | **return** fold;
   end

   Algorithm 1: Agent 2

2. **Agent3 : An agent who calls if it has a jack or higher card in its hand (this would include aces) and folds otherwise.**

   initialization;
   if *high_card_value()* > *10* **then**
   | **return** call;
   else
   | **return** fold;
   end

   Algorithm 2: Agent 3

3. **Agent4 : An agent of your own design you feel would perform well. (Also describe in English the rationale behind your design.).**

   initialization;
   if *high_card_value()* > *10* && *pair()* **then**
   | **return** call;
   else
   | **return** fold;
   end

   Algorithm 3: Agent 4

   The idea is to increase the TP rate with the price of smaller number of games that the agent calls. Of course in term of the best TP rate this is not the best scenario, still the agent want to have a chance to play few games. We will describe it in more detail the following. Note that here we assumed that ante is zero, if ante is not zero maximizing TP rate is not the best strategy if the net winning matters.

Give a brief description of the process you used for converting the poker agents defined by condition-action rules to a tabulated format. Feel free to use pseudocode in your description, though this is not required.

The algorithm in 4 is showing how we make the realization of every possible cards (the for loops) and

Table 1: Performance Measurements

Agent	False Positive (FP)	False Negative (FN)	True Positive (TP)	True Negative (TN)	TP/(TP+FP)	TN/(TN+FN)	TP+FP	Designer Performance (3TP-FP)/(TP+FP)
1	338	2402	269	6991	0.44	0.74	607	0.77
2	2042	1706	889	5363	0.30	0.76	2931	0.21
3	3776	976	1557	3691	0.29	0.79	5333	0.16
4	72	2451	107	7370	0.6	0.75	179	1.39

the agent action call the rule and return "call" or "fold" based on the agent's rules.

```
initialization;
for i = 1 to 51 do
    card_1 = int_to_card(i);
    for j = i + 1 to 52 do
        card_2 = int_to_card(j);
        print card_1,card_2,agent.action(card_1, card_2);
        print card_2,card_1,agent.action(card_2, card_1);
    end
end
```

Algorithm 4: poker agents defined by condition-action rules to a tabulated format

Determine the number of functionally distinct pairs of hole cards. That is, count the number of ways in which two hole cards can be dealt from a 52 card deck such that no two ways are equally favorable for the player. Justify your answer.

1. # Pairs: 13 There are only 13 distict set of pairs $[1, 2, \cdots, \mathrm{K}, A]$

2. # Suited hands: $13 \times 12/2 = 78$, assuming both card have the same suits then there are 13 possibilities for the first card and 12 possibilities for the second card and because the ordering does not matter then we divide it by two.

3. # Offsuit hands: The same as # Suited hands. Because the suits are not an issue though the numbers would be an issue. With the same argument there are 78 of distinct value hole cards.

In total we have $13 + 78 + 78 = 169$ of distinct value hole cards

Record for each of agents 1-4 the outcomes in a table with the following format:

1. **Compute the percentage of true positives out of total positives and the percentage of true negatives out of total negatives for each agent. (TP/(TP+FP) and TN / (TN+ FN))**

   Look at the Table 1.

2. **Order the agents by how many total positives each had (TP+FP). Based on this, and the ratios you computed in (1), do you notice any trends among agents 1-3? Explain why this makes sense based on how each agent plays.**

   Total positive cases are all cases where the agent call. It is increasing for agent 1 and 2 and 3 because the total number of states which the call function is called increased. For agent 1 total number positive states are: $13 \times 12 = 156$ (out of 2652). For agent 2 total number of positive states are: $13 \times 12 = 156 + 4 \times 13 \times 12 = 780$ (out of 2652). For agent 3 total number of positive states are: $52 \times 51 - 36 \times 35 = 1392$ (out of 2652).

3. **Compare your agent (agent 4) to the others in terms of the ratios from (1) and ranking from (2). Why do you think your agent performed the way it did?**

   The number of states that the agent call decreased, and it decreased in a way that maximizes the TP rate. Not only it should be a pair but also it should be jack or larger in order to agent call it. The the total number of available states (in order to call), are smaller than agent 1 so the FP + TP goes down, though it selects the best set of pairs, then the TP/(TP + FP) goes up. TN rate (TN/(TN + FN)) goes down because there are more cases that the agent fold and it increases the chance of making a mistake.

4. **Describe what criteria or additional information you would need to determine which agent has the best performance (i.e. which agent makes a less severe errors than another). Which agent do you think performed the best overall? State your assumptions and explain your reasoning.**

   From a point of view, the expected value of (TP + TN) / All would determine the overall performance, if the designer care about the average number of correct predictions. It indicates the number of the time the agent did not loose any bet or win the bet, or in another word, which agent makes a less severe errors, larger the number means more correct prediction. Though one can include the cost in this equation and see how much money one expect to win/loos for given strategy. In that case the overall performance would be (3 × TP - FP) / (TP + FP). TP(FP) is in unit of bet. If it is negative one looses in average if it is positive one in average wins. I would choose (3 × TP - FP) / (TP + FP) as designer performance measure, which is shown in table 1. For sure I prefer the second one, one can fold many times and do not loose any money (here I assumed ante is zero), and have ∞ time to spend at a casino. The idea it to win as much as possible and the number of games which one can fold does not matter in this case, because the player has ∞ time to play. Note that the factor of 3 is due to the fact that there are 3 other players and once the agent win the game, win all the money.

   Though I want to note that, this is a dangerous algorithm in a real world champion because after few hands, the opponents can learn the agent's algorithm easily and bit it and there would be ante in a real game. So this algorithm is not practical and optimized. One could still make it more efficient in term of our performance measures, by making the agent only plays if it sees a pair of aces. In that case it get the highest TP rate with the price of almost not playing for a finite and small number of games.

   Under this assumption the game become boring and never-ending.

5. **Suppose you have a fifth agent that chooses to fold or call randomly, with a 50% chance of each. What are the expected values of the tables entries for this agent? (i.e., what is the expected number of false positives, false negatives, etc?)**

   For a random game in case of five agent game, and if no other agent except agent 5 folds, we expect each agent wins 20% of the games randomly, if everyone call. So FN+TP=$0.2 \times \#$ of games. We know that agent 5 would call randomly 50% of the time. As a result we have TN=TP=$0.1 \times \#$ of games. The same argument holds for FN+FP. One can conclude that TN=FP=$0.4 \times \#$ of games. # of games in case of our simulation program is 10,000. Or simply for the rate of mentioned performance measurements, we have:

   $\langle TP \rangle = \langle FN \rangle = 0.1$ and $\langle TN \rangle = \langle FP \rangle = 0.4$

   Above numbers are normalized by the number of games.

Table 2: Performance Measurements for Utility-Based Player

Agent	False Positive (FP)	False Negative (FN)	True Positive (TP)	True Negative (TN)	TP/(TP+FP)	TN/(TN+FN)	TP+FP
Prob > 18%	5684	311	2263	1742	0.28	0.85	7947
Prob > 22%	3858	813	1771	3458	0.31	0.81	5629
Prob > 26%	2479	1278	1302	4941	0.34	0.79	3781

Table 3: Net Winnings for Utility-Based Player

Agent	Net Winnings
Prob > 18%	4734
Prob > 22%	2898
Prob > 26%	-1022

4 Utility-Based Player

Explain a way these probabilities could be estimated.

Agent 1 knows its own state. His card can be subtracted from the deck and one can build all possible outcomes and count the number of outcomes that his cards may win that hand assuming there are 3 other players on the table, and they always would play that hand.

Note that the probabilities are changing if the agent know a player is already folded.

Run the simulation for 10000 hands for each threshold by selecting the radio button for the appropriate threshold strategy. Record the results from all thresholds in a table like the one in problem 3D.

Look at table 2.

Based on your numbers from part b, determine the net winnings (or losings) of each threshold strategy over the 10000 hands. Which threshold (0.18, 0.22, or 0.26) worked the best?

The net winnings is calculated according to the following formula, NW = 32 × TP - 6 × (FP + TP) - 2 × 10,000, which assumes 10,000 hands. The results is presented in table 3. Interestingly, Agent with Prob > 18% has the highest net winnings.

What is the expected net gain for one hand if the agent folds?

$$\langle \text{net gain} \rangle_{\text{fold}} = -2 \tag{1}$$

What is the expected net gain for one hand if the agent calls? (Express this in terms of the probability p of winning)

$$\langle \text{net gain} \rangle_{\text{call}} = 24p - 8(1-p) = 32p - 8 \tag{2}$$

Use parts D and E to determine the optimal threshold value. That is, determine for which values of p it is better to call than to fold.

We are looking for the point where the $\langle \text{net gain} \rangle_{\text{fold}} = \langle \text{net gain} \rangle_{\text{call}}$. At that point folding or calling would not affect the expected net gain. If the probability become larger it would be better to call rather than folding because the the expected net gain would increases. On the other hand, if the probability be below that threshold the expect net gain for fold would be larger. At the equality point it does not matter which strategy we take and it defines the threshold. We have,

$$\langle \text{net gain} \rangle_{\text{fold}} = \langle \text{net gain} \rangle_{\text{call}} \tag{3}$$

after plugin the numerical values from part D and E we get,

$$32p - 8 = -2 \tag{4}$$

solving above equation gives us, $p = 0.1875$.

5 Learning

Now, let us return to the full game of Texas Hold'em, as played in the World Series of Poker. Remember that, unlike the training program we've used, each player only sees his or her own hole cards until the hand ends. We will only consider a single table of players. At the table, there will be many hands of poker played, with each hand proceeding according to the full rules (as linked to in the introduction). Note that players can only leave the table when they either lose all their money, or else take the money from all others at the table, and no new players can join the table during the game. An important aspect of poker is to spend some time getting a feel for the strategies of your opponents. Suppose you were associating the opponent's pair of hole cards (for simplicity assume these are shown after the hand even if they folded) with his/her decision about whether to stay or fold. The player could use one of the following three methods above to model the opponent: Table-lookup, Reflex rule, Threshold

For each method, describe in a few sentences how easy/difficult it would be to infer the model. Briefly sketch a systematic method for building these models while the game progresses.

1. Table Look up:

If table look up is not build based on a rule, then one need to see all possible non-identical cases to be able to map out the table look up. There are total of $51 \times 52 = 2652$ possibilities where many of these cases are identical. Like ([3♣, 9♢] and [9♢, 3♣]) or ([9♣, 9♠] and [9♢, 9♣]) .

2. Reflex rule:

It is based on how many rules exists. In the worst scenario the maximum number of unique case which one needs to see before learning the rules would be like table look up. If combination of rules are allowed (e.g. *pair*() **and** *high_card_value*()), then one can generate very specific rules in order to for example include [9♣, 9♠] but not [10♣, 10♠]. So in practice space of search would be similar to table look up.

In practice though, note that one can uses his/her intuition to rule out/in some cases. For example if the agent plays based on rules it would be very unlikely to play [9♣, 9♠] but not [10♣, 10♠]. Logically it does not make sense, except the agent want to confuse other agents which are trying to learn the rules.

3. Threshold:

Learning this strategy would be very easy. There are finite number of probabilities. As the game goes on the player can estimate the threshold based on whether the agent called or folded. The estimation gets better and better as the game goes on.

In the above question, I assumed that the strategy of each player is not changing a the card revelation occurs, though one may changes the strategy at each card revelation.

Given your answers from part A, which of these methods would you use to model your opponents? Why?

Well, in reality probably none of them works, because the opponent may try to bluff or change the strategy as the game goes on.

But logically starting with the threshold would be the best, as along as I assume I know all the probabilities. Because even after the first hand I can have an estimation of the threshold, after few hands the estimation get better and better. Because the only thing which I have to memorize is a single number, it is easier much easier than a complicated rule to a table look up. For sure I can not remember the table look up.

How would an agent use these opponent models to its advantage?

Based on whether the opponents called or folded, the agent can model opponents cards, and it can estimate by how much the chance of winning for its cards goes up or down. If one knows the opponents table look up (for example) then if the opponent call, the agent can rule out some of the states which it

knows that the opponent would fold if he/she has those cards. This is an extra information which the agent can use to increase its chance of winning.

If an opponent thinks that the agent is trying to model it, what might the opponent do to hinder those efforts?.

The opponent can do three things,

1. It can changes the rules (look up table), or the model completely at a random point in the game.

2. It may make a bluff to confuse the other players, and mess up the learning process.

3. Occasionally it may play randomly, which confuses the other player.

4. It can decide about N different strategies and change from one strategy to another with some designed pattern in advance.

Maybe the outcome many of above suggestions is similar for an outsider, but from modeling perspective it is very different whether the player bluff according to some rules, or play randomly for example, or changes the strategy completely.

A The Python code

Only the relevant of the code is provided here. For the full code the grader may want to check the git repository for this homework: https://github.com/afarahi/AI_HW1/tree/master/scr/HW1.

```
import sys
from HW1 import agent1_class, agent2_class, agent3_class, agent4_class, card_class

def print_tabular_rules():

    try:
        agent_id = int(sys.argv[2])
    except IndexError:
        print "Error, Please provide agent_id"
        sys.exit(-1)

    if agent_id == 1:
        agent = agent1_class()
    if agent_id == 2:
        agent = agent2_class()
    if agent_id == 3:
        agent = agent3_class()
    if agent_id == 4:
        agent = agent4_class()
    card = card_class()

    counter = 0

    for i in range(51):

        card_1 = card.int_to_card(i)

        for j in range(i+1, 52):

            card_2 = card.int_to_card(j)

            agent.assign(card_1, card_2)

            # counter += 1
            # print counter, card.card_to_name(card_1), card.card_to_name(card_2), agent.action()
            # counter += 1
            # print counter, card.card_to_name(card_2), card.card_to_name(card_1), agent.action()

            counter += 1
            print "%s,%s,%s"%(card.card_to_name(card_1), card.card_to_name(card_2), agent.action())
            counter += 1
            print "%s,%s,%s"%(card.card_to_name(card_2), card.card_to_name(card_1), agent.action())
class card_class():

    def __init__(self):
        pass

    def int_to_card(self, i):
        suit = i/13 + 1
        number = i%13 + 2
```

```
        return suit, number

    def card_to_int(self, card):
        return (card[0]-1)*13 + card[1] - 2

    def card_to_name(self, card):
        if card[0] == 1:
            return "c%i"%card[1]
        elif card[0] == 2:
            return "d%i"%card[1]
        elif card[0] == 3:
            return "h%i"%card[1]
        elif card[0] == 4:
            return "s%i"%card[1]




class texas_hold_em_agent_class():

    def __init__(self, card_1, card_2):
        """
        Jack=11, Queen=12, King=13, Ace=14
        Initialize the agent
        :param card_1: two int value, the second value is the card number
         the first value is the suit
        :param card_2: two int value, the second value is the card number
         the first value is the suit
        """

        self.card_1 = card_1
        self.card_2 = card_2

    def pair(self):
        """
        true if the hole cards are a pair
        :return: boolean
        """
        return self.card_1[1] == self.card_2[1]

    def same_suit(self):
        """
        true if the hole cards have the same suit
        :return: boolean
        """
        return self.card_1[0] == self.card_2[0]

    def distance_between(self):
        """
        returns the distance between the cards
        useful for detecting possible straights
        :return: integer
        """
        return abs(self.card_1[1] - self.card_2[1])

    def high_card_value(self):
```

```
        """
        returns the value of the high card
        :return: integer
        """
        return max(self.card_1[1], self.card_2[1])

    def call(self):
        return "call"

    def fold(self):
        return "fold"


class agent1_class():
    """
    agent that only calls on pairs
    """

    def __init__(self):
        pass

    def assign(self, card_1, card_2):
        """
        Jack=11, Queen=12, King=13, Ace=14
        Initialize the agent
        :param card_1: two int value, the first value is the card number
         the second value is the suit
        :param card_2: two int value, the first value is the card number
         the second value is the suit
        """
        self.agent = texas_hold_em_agent_class(card_1, card_2)

    def action(self):

        if self.agent.pair():
            return self.agent.call()
        else:
            return self.agent.fold()


class agent2_class():
    """
    agent that only calls on pairs
    """

    def __init__(self):
        pass

    def assign(self, card_1, card_2):
        """
        Jack=11, Queen=12, King=13, Ace=14
        Initialize the agent
        :param card_1: two int value, the first value is the card number
         the second value is the suit
        :param card_2: two int value, the first value is the card number
         the second value is the suit
```

```
        """
        self.agent = texas_hold_em_agent_class(card_1, card_2)

    def action(self):

        if self.agent.pair():
            return self.agent.call()
        elif self.agent.same_suit():
            return self.agent.call()
        else:
            return self.agent.fold()


class agent3_class():
    """
    agent that only calls on pairs
    """

    def __init__(self):
        pass

    def assign(self, card_1, card_2):
        """
        Jack=11, Queen=12, King=13, Ace=14
        Initialize the agent
        :param card_1: two int value, the first value is the card number
         the second value is the suit
        :param card_2: two int value, the first value is the card number
         the second value is the suit
        """
        self.agent = texas_hold_em_agent_class(card_1, card_2)

    def action(self):

        if self.agent.high_card_value() > 10:
            return self.agent.call()
        else:
            return self.agent.fold()


class agent4_class():
    """
    agent that only calls on pairs
    """

    def __init__(self):
        pass

    def assign(self, card_1, card_2):
        """
        Jack=11, Queen=12, King=13, Ace=14
        Initialize the agent
        :param card_1: two int value, the first value is the card number
         the second value is the suit
        :param card_2: two int value, the first value is the card number
         the second value is the suit
```

```
    """
    self.agent = texas_hold_em_agent_class(card_1, card_2)

def action(self):

    if self.agent.pair() and self.agent.high_card_value() > 10:
        return self.agent.call()
    # elif (self.agent.same_suit() and self.agent.distance_between() < 2
    #       and self.agent.high_card_value() > 9):
    #    return self.agent.call()
    else:
        return self.agent.fold()
```