Yahtzee Agent

A project by Lucas Dunn, San Davran, and Danil Kryuchkov

Table of Contents

Initial Project Proposal

First, read the official Yahtzee rules: https://www.officialgamerules.org/yahtzee. For a hands-on experience of playing the game yourself and getting a feel for the rules visit Yahtzee | Play it online (cardgames.io). The rest of the problem description will follow assuming you know the rules of the game.

The game of Yahtzee is random in many ways, but there are a few crucial decisions a player can make to maximize their score. For example if a player is going for a large straight, and they roll [1, 2, 3, 5, 6], should they keep the 1 or the 6? The answer depends on what their backup plan is if they don't happen to roll that crucial 4. If they have their "Chance" category open, they should keep the six in order to maximize the number of points they get in that square. If the "Chance" category is full and their "Ones" category is open however, they might be better off keeping the one, and cutting their losses if they happen to roll a "1" instead of a "4". (The advantage to keeping the 1 in this case is that they would be putting two points in their "Ones" place rather than just the single point they would have if they had kept the 6 instead of the 1.) Another crucial example is if on your first turn, after your three rolls, you have [6 6 6 6 3]. Do you put it in your four-of-a-kind and take the 27 points? Or is it better in the long term to put it in your "Sixes", taking 24 points instead of 27 points, but setting yourself up for the future to be more likely to get the bonus of 35 points at the end of the game. (Note that another downside to putting the points into the "Sixes" is opportunity cost → you might not get another high scoring four-of-a-kind, but you will likely get at least three 6's to put in your "Sixes" category.)

There are many of these types of decisions, where it is not instantly obvious what the best statistical long term play is. This makes it an interesting problem, as given the closed nature of the game, and the limited (albeit stochastic) branches of possibilities, there is surely an optimal series of decisions a player can make to increase their **expected score**.

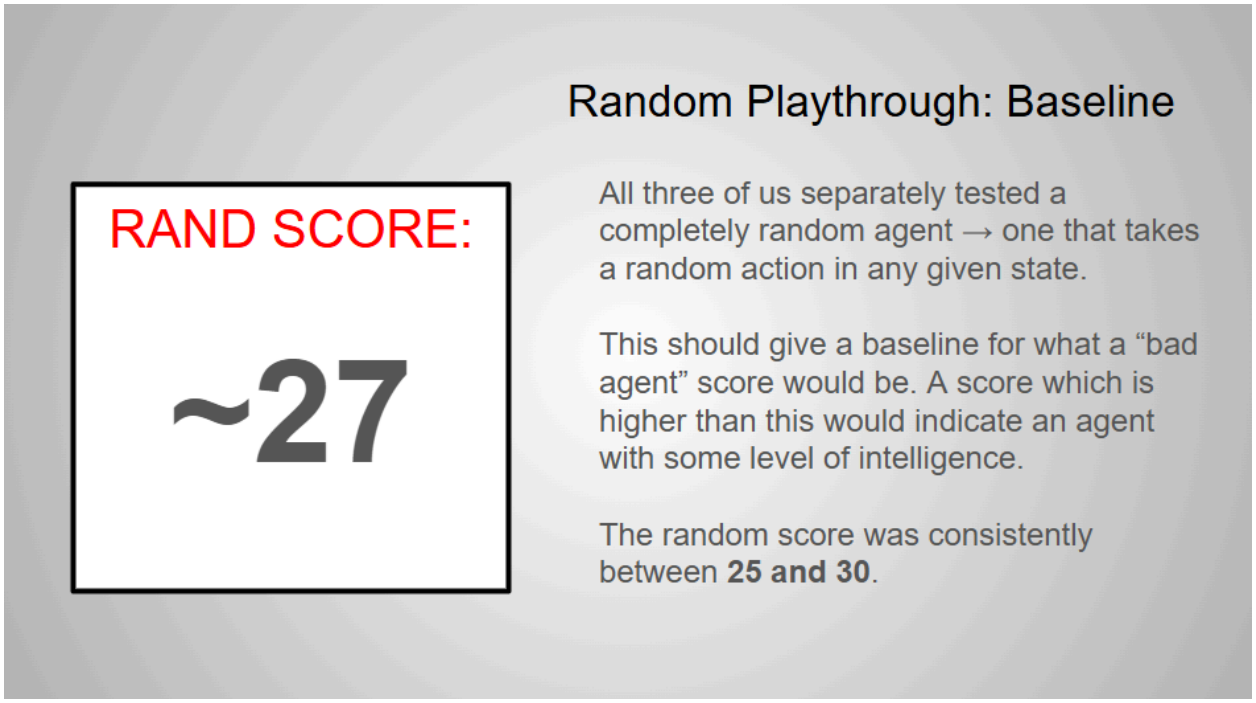
Our proposition is to write a program which chooses the optimal dice to store and the optimal category to put the points in, in order to maximize its expected score.

*Note that we will only be considering our own score, not our opponents. Where this comes into account is, for example, if you are very far behind your opponent, there may be a case where you should go for yahtzees exclusively, as it gives a non-zero chance to make a comeback, but at the cost of your own expected score. This would be a natural followup, but for the sake of simplicity, and setting an initial goal for the project, we propose only considering our own score.

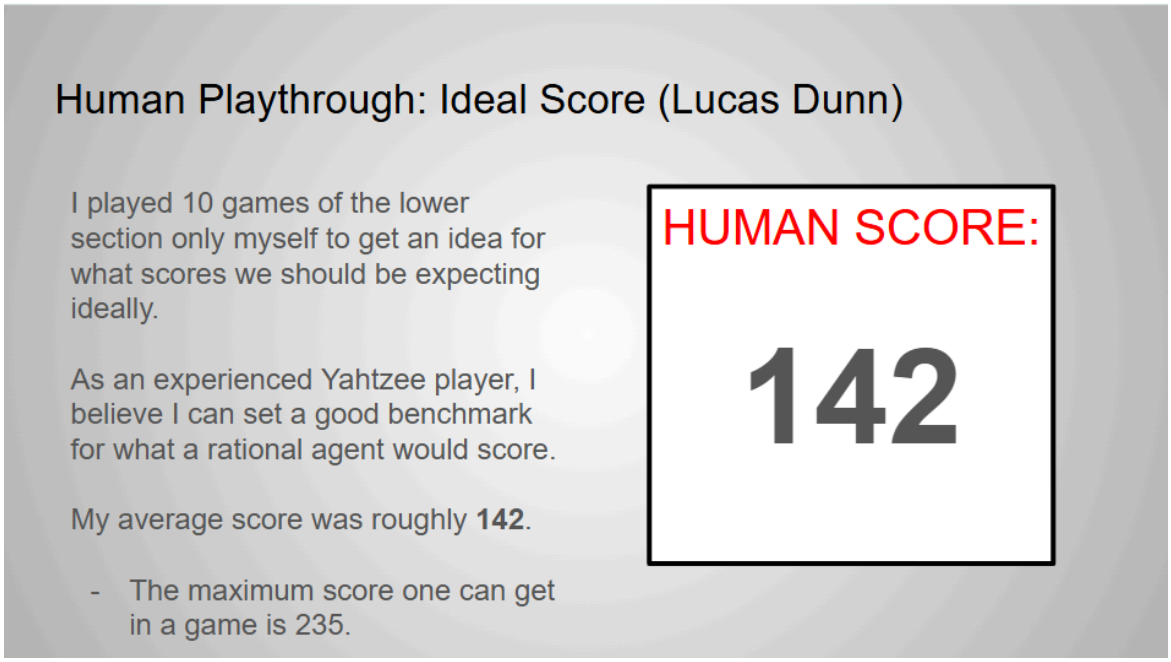
*Note that as we started the project, to reduce the state space, in some instances we only considered the lower section.

Baseline

Random Score



Human Score



Resources

Full Code on GitHub - dunnlu/yahtzeeAgent (github.com)

Yahtzee Online - Yahtzee | Play it online (cardgames.io)

Official Yahtzee Rules - https://www.officialgamerules.org/yahtzee

Markov Decision Process (MDP)

State Representation

Each state is represented as a vector of integers.
State vector consists of three distinct sections.

First section represents the values of the dice at hand.
There are 5 dice, each with 6 sides.
Therefore, first 5 integers of the vector are always ∈{1,2,3,4,5,6}
Dice values are sorted from low to high because we don't care about the order of the dice values, we only care about what they are. Using this trick, reduces the size of this part of the state space from 7776, (6^5) to 252.
Second section represents the number of rolls left in the current turn.
It is a single integer with value ∈{0,1,2}
This affects the possible actions in the state, as if it is 0, rerolls are impossible, but if it is 1 or 2, rerolling is possible.
Third section represents the availability of scoring boxes on the score sheet.
There are 13 integers (booleans) ∈{0,1}
They represent each of the possible scoring boxes, and whether they are already taken (1), or can still be used (0).

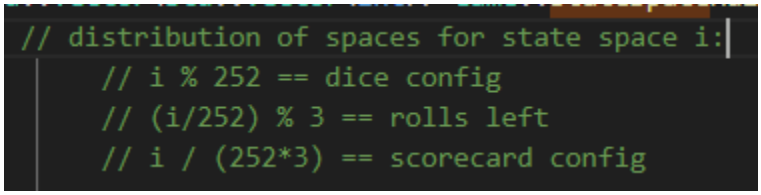
Note: In our project, we only used the lower 7 boxes of the score sheet (first 6 were always set to (1) aka. Used for generating episodes, etc.)

State Space Implementation: (Lucas Dunn)

The state space is generated as a 2D vector. The rows are different states, and the columns are features of the state.

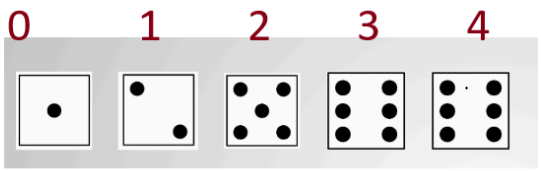
The state space table can be thought of as a hash map, where properties of the state determine what its index in the table is. For generation, properties are generated based on the index, not vice versa. This does however lead to every unique state being represented, as it is quite simple.

This is the formula for determining the properties of the state space given index i:



*Note that '/' is integer division

State Space Example (Danil)

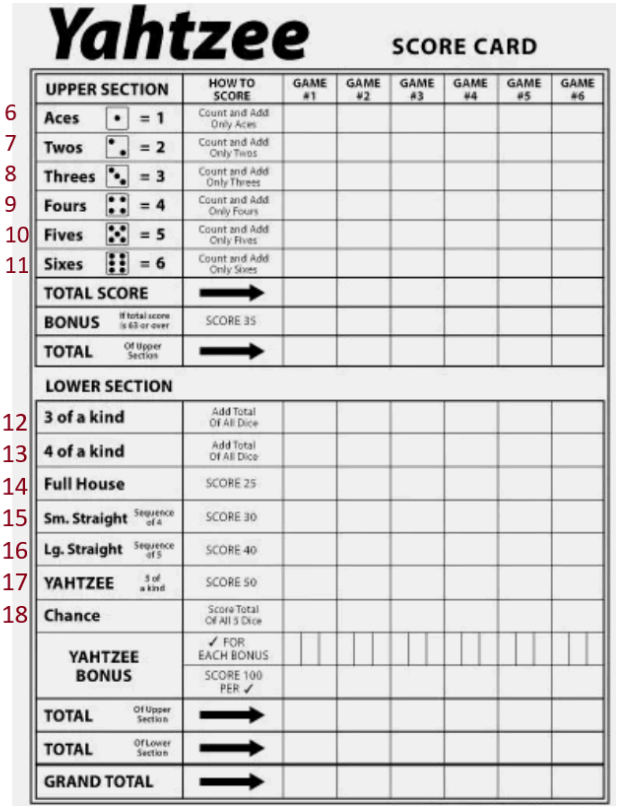


5

Rerolls left: 2

idx: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18

state: [1, 2, 5, 6, 6, 2, 0, 0, 0, 0, 0 , 0, 0, 0, 0, 0, 0, 0, 0]



Action Representation

Each action is represented as a vector of integers.
Action vector consists of two district sections.

First section represents the action on every die.

It is 5 integers (booleans) $\in\{0,1\}$ corresponding to every one of 5 dice.

They only play a part if the second section has a value of (-1) aka. reroll.

If the second section is = (-1) aka. reroll, dice with a corresponding integer value of 1 (in the action vector) will be “kept” and not rerolled.

Those with a corresponding integer value of 0 (in the action vector) will be “rerolled”.

Ex: current dice is [1, 2, 5, 6 ,6], first section of the action vector is [1,1,1,0,0] will result in the next state having the following dice: [1,2,5,X,X] where X is unknown, newly rolled dice. We can see that 1,2,5 were “kept” and 6,6 were rerolled. They can ofcourse roll to be 6, 6 again, but they can also be any other dice value upon reroll.

Second section represents an action on the score sheet or reroll.

It is a single signed integer $\in\{-1,0, 1, 2\ldots13\}$

-1 stands for “reroll”, no score is saved on the sheet, “unkempt” dice are rerolled.

[0:13] stands for the index of the scoresheet box that is to be filled. It cannot be a box that is already used.

Note: if this section is anything but -1, the first section is disregarded, and the present dice are used to score.

Action Implementation (Lucas)

In each state, there are only a few legal moves, as described above. For example, if you have 0 rolls left, you *must* score. Additionally, you cannot score in a box which is already filled. The ‘possibleActions’ function simply returns a 2D vector, where the rows are the actions and the columns are features of the action (which dice to keep, where to score).

The actual implementation is as simple as possible, and can be thought of as:

possibleActions(state) = [<action>$_1$, <action>$_2$...]

where <action> follows the format described above.

When creating a policy, you only need to specify the state, and the *index of the action in possibleActions*. This is because possibleActions is deterministic based on the state, and storing an index to the action rather than the action itself is simple and efficient.

Reward Functions (Value Iteration Only)

For the purpose of this report, the function “game.score(<box>)” is the score you would get in yahtzee from taking an action. For example if you have five 1’s as the dice, and the action was to score in your yahtzee box, it would return “50”. If the action was instead to score in your three of a kind box, it would return “5”. It can be thought of as a black box for scoring in the specified box.

There were two reward functions used. They are as follows:

1. Minimalistic R(s,a)

    - Reward for rerolling → 0
    - Reward for scoring → game.score(<box>)

    This reward function is as simple as possible. This theoretically leads to optimal behavior, as it is the exact same reward one gets for simply playing yahtzee. However it may take a while to converge, as it theoretically takes a long time to encode learning that you should store in four of a kind before three of a kind, for example.

2. Common Human Strategy R(s,a)

    - Reward for rerolling → 0
    - Reward for scoring zero in a box:
        - Three of a Kind → -10
        - Four of a Kind → -7
        - Full House → -10
        - Small Straight → -10
        - Large Straight → -7
        - Yahtzee → -4
    - Reward for scoring nonzero points in a box
        - Three of a kind → game.score(three_of_a_kind) - 1
        - Chance → -1
        - Other → game.score(<box>)

    This reward function is meant to mimic human behavior when scoring zero in a box → that is, set a preference order for which boxes to score zero in. Additionally, it encodes favoring scoring in four of a kind rather than three of a kind, and ONLY scoring in chance if you can’t score points anywhere else, and it does not return any information about how good your chance is. This means it is theoretically nonoptimal, as it leads to random behavior when the only box you have left is chance. However, it should converge quickly to standard human behavior, as it will try to maximize points in every square but chance, and score in certain boxes in a certain order.

Transition Function (Value Iteration Only)

This report describes a function called “transition” which generates a vector of tuples → resulting state, and odds of being in that state. It can be thought of as the following:

$T(s,a) = [(s'_1, T(s,a,s'_1)), (s'_2, T(s,a,s'_2)) \dots]$

This was implemented in conjunction with state space generation. As a reminder, the state space can be thought of as a hash map, where properties of the state determine its position in the state space.

The implementation for adding a single resulting_state-odds pair is as follows:

```
// add the tuple with the enumerated position and the corresponding odds to the transition vector
transition.push_back(std::make_tuple(252*3*scPos + 252*(state[5]-1) + dicePosition(roll), it.value()));
```

- The first element of the tuple is the hash function for the index of the resulting state within ‘state_space’
  - scPos is an integer in the range [0, 2^7] corresponding to the scorecard configuration of the resulting state
  - (state[5]-1) is the amount of rolls remaining in the resulting state. If the rolls left in the current state is 0, this is 2 instead.
  - dicePosition(roll) is an integer from [0,251] corresponding to the dice roll in the resulting state
- The second element is the odds for being in the resulting state
  - This is entirely dependent on the dice roll
  - This is determined by a json, where the key is the roll and the value is the odds of getting that roll
  - So “it.value()” just returns this iteration’s dice odds from the json

Example:
Let's say you kept 4 dice and decided to reroll, with an empty scorecard, and 2 rolls left:

- The transition function would return 6 states, one for each possible roll of the only remaining dice
- Each resulting state would have 1 roll left, and an empty scorecard
- Each resulting state would be equally likely, as there is only one dice being rolled

Let’s say you score in your chance, with only your yahtzee box filled

- The transition function would calculate scorecard position with yahtzee AND chance filled
- Each resulting state would have 2 rolls left, as it is the start of a new turn
- There would be 252 resulting states, one for each possible roll of 5 dice
- Some states would be more likely than others, for example there are 120 different permutations of [1,2,3,4,5] but only one permutation of [1,1,1,1,1]

Value Iteration

Algorithm

Value Iteration:

- space = stateSpace #
- for all states s in space:
    - value[state] = 0
    - policy[state] = array.resize(6)
- temp_val = array.resize(states) # temporary value array
- temp_state_value
- for i in range(22)
    - for state in space:
        - best_value = 0
        - for action in possibleActions(state)
            - temp_state_value = 0
            - temp_state_value+=reward(state,action)
            - for resulting_state in transition(state,action)
                - temp_state_value += resulting_state[odds] * value[resulting_state]
            - if temp_state_value > best_value
                - best_value = temp_state_value
                - best_action = action
        - temp_val[state] = best_value
        - policy[state] = best_action
    - value = temp_val
- return (value, policy)

This is the pseudo code for the value iteration function. It is worth noting that after each iteration, the resulting value and policy iteration vectors are stored in text files.

IMPORTANT: In the final project presentation, I reported suboptimal results. This was due to an error in the transition function. Since then, I have fixed the transition function, and the results are more than 50% better. They are actually quite close to human play, as will be presented later in the report.

Training

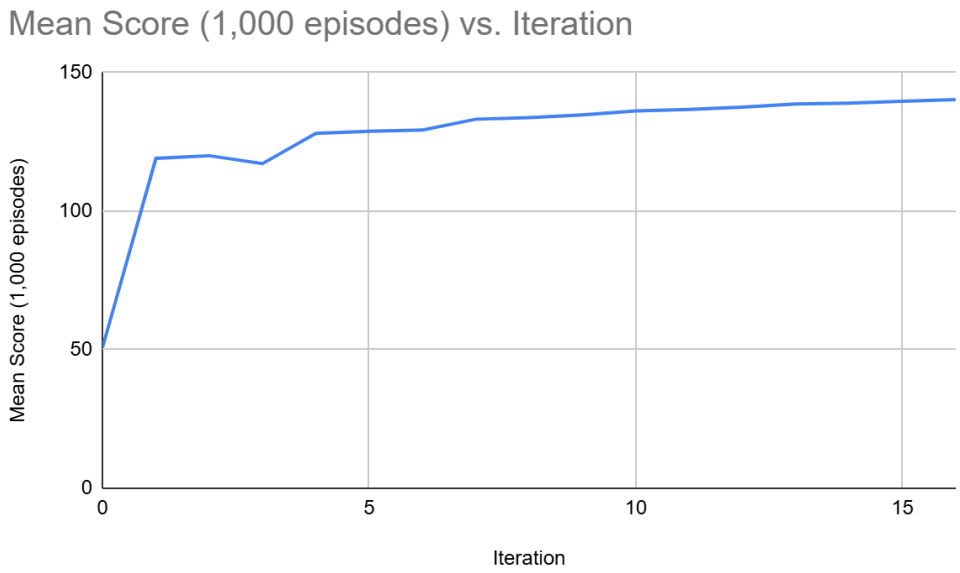
Value Iteration was run on OSU's High Performance Computing Cluster. Each iteration took roughly 2 hours to complete.

For value iteration, I trained it using both reward functions mentioned. For the human-strategy reward function, I trained it for 22 iterations. For the minimalistic reward function, I trained it for 17 iterations – as many as I could before the report was due.

Evaluation

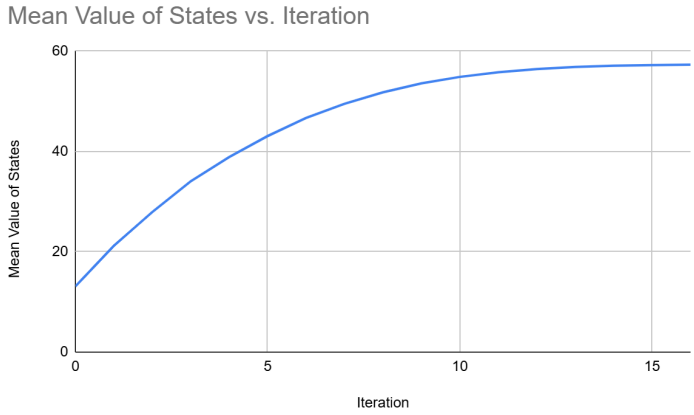
1. Minimalistic Reward Function

   Mean Score (17th iteration): 140



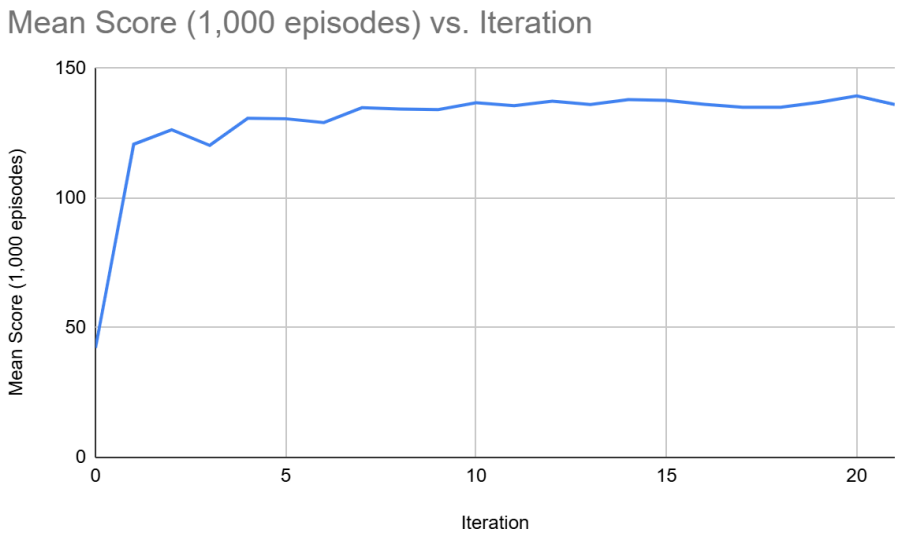
As predicted, the score continued to iterate even after 10, and continues to improve up until the most recent iteration.

It took until the 5th iteration to reach 120, and the 8th iteration to reach 130, confirming the hypothesis that it would improve slower. What is also true however is that it seems to have a higher ceiling, as it is not finished running yet, but it has already reached an average score of 140 by the 17th iteration.



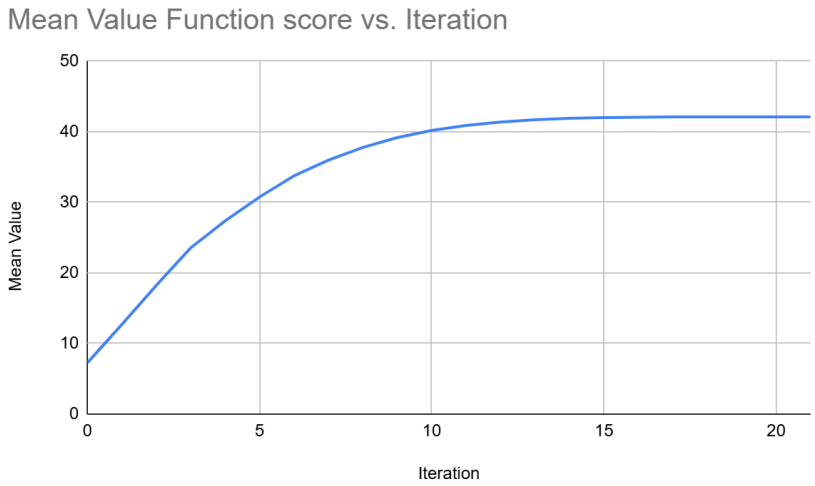
The mean perceived value of states reaches around 60. This is because the value of a state can be thought of as "*expected score starting from this position*", so the later in the game a state is, the lower the perceived value. Terminal states have a value of 0.

2. Common Human Strategy Reward Function
   Mean Score (21st iteration): 139



After 10 iterations, the score plateaued to an average score of roughly 137.

This reward function was able to converge to a high average score relatively quickly, reaching 120 on the second iteration, and reaching 130 on the fourth iteration.



IMPORTANT: Additionally, the function converged at 21 iterations, as the 22nd iteration had the exact same value function as the 21st. This was as predicted, and checks out as there are only 21 possible decisions (7 turns, 3 or less decisions per turn).

Game Play (Minimalistic Reward Function)

Score: 0
State:
Dice: 1 2 2 6 6
Rolls Left: 2
Scorecard: [Ones 1 | Twos 1 | Threes 1 | Fours 1 | Fives 1 | Sixes 1 | Three of A Kind 0 | Four of a Kind 0 | Full House 0 | Small Straight 0 | Large Straight 0 | Yahtzee 0 | Chance 0 |]

ACTION
0 | 0 | 0 | 1 | 1

Score: 0
State:
Dice: 2 3 3 6 6
Rolls Left: 1
Scorecard: [Ones 1 | Twos 1 | Threes 1 | Fours 1 | Fives 1 | Sixes 1 | Three of A Kind 0 | Four of a Kind 0 | Full House 0 | Small Straight 0 | Large Straight 0 | Yahtzee 0 | Chance 0 |]

ACTION
0 | 0 | 0 | 1 | 1

Score: 0
State:
Dice: 2 3 6 6 6
Rolls Left: 0
Scorecard: [Ones 1 | Twos 1 | Threes 1 | Fours 1 | Fives 1 | Sixes 1 | Three of A Kind 0 | Four of a Kind 0 | Full House 0 | Small Straight 0 | Large Straight 0 | Yahtzee 0 | Chance 0 |]

ACTION
Three of A Kind

Score: 23
State:
Dice: 1 1 2 3 6
Rolls Left: 2
Scorecard: [Ones 1 | Twos 1 | Threes 1 | Fours 1 | Fives 1 | Sixes 1 | Three of A Kind 1 | Four of a Kind 0 | Full House 0 | Small Straight 0 | Large Straight 0 | Yahtzee 0 | Chance 0 |]

ACTION
0 | 0 | 0 | 1 | 0

Score: 23
State:
Dice: 1 3 3 4 6
Rolls Left: 1
Scorecard: [Ones 1 | Twos 1 | Threes 1 | Fours 1 | Fives 1 | Sixes 1 | Three of A Kind 1 | Four of a Kind 0 | Full House 0 | Small Straight 0 | Large Straight 0 | Yahtzee 0 | Chance 0 |]

ACTION
0 | 1 | 0 | 1 | 1

Score: 23
State:
Dice: 3 3 4 6 6
Rolls Left: 0
Scorecard: [Ones 1 | Twos 1 | Threes 1 | Fours 1 | Fives 1 | Sixes 1 | Three of A Kind 1 | Four of a Kind 0 | Full House 0 | Small Straight 0 | Large Straight 0 | Yahtzee 0 | Chance 0 |]

ACTION
Chance

Score: 45
State:
Dice: 1 2 2 5 6
Rolls Left: 2
Scorecard: [Ones 1 | Twos 1 | Threes 1 | Fours 1 | Fives 1 | Sixes 1 | Three of A Kind 1 | Four of a Kind 0 | Full House 0 | Small Straight 0 | Large Straight 0 | Yahtzee 0 | Chance 1 |]

ACTION
0 | 0 | 0 | 1 | 0

Score: 45
State:
Dice: 2 3 5 5 5
Rolls Left: 1
Scorecard: [Ones 1 | Twos 1 | Threes 1 | Fours 1 | Fives 1 | Sixes 1 | Three of A Kind 1 | Four of a Kind 0 | Full House 0 | Small Straight 0 | Large Straight 0 | Yahtzee 0 | Chance 1 |]

ACTION
0 | 0 | 1 | 1 | 1

Score: 45
State:
Dice: 2 4 5 5 5
Rolls Left: 0
Scorecard: [Ones 1 | Twos 1 | Threes 1 | Fours 1 | Fives 1 | Sixes 1 | Three of A Kind 1 | Four of a Kind 0 | Full House 0 | Small Straight 0 | Large Straight 0 | Yahtzee 0 | Chance 1 |]

ACTION
Yahtzee

Score: 45
State:
Dice: 1 3 3 5 5
Rolls Left: 2
Scorecard: [Ones 1 | Twos 1 | Threes 1 | Fours 1 | Fives 1 | Sixes 1 | Three of A Kind 1 | Four of a Kind 0 | Full House 0 | Small Straight 0 | Large Straight 0 | Yahtzee 1 | Chance 1 |]

ACTION
0 | 1 | 1 | 1 | 1

Score: 45
State:
Dice: 1 3 3 5 5
Rolls Left: 1
Scorecard: [Ones 1 | Twos 1 | Threes 1 | Fours 1 | Fives 1 | Sixes 1 | Three of A Kind 1 | Four of a Kind 0 | Full House 0 | Small Straight 0 | Large Straight 0 | Yahtzee 1 | Chance 1 |]

ACTION
0 | 1 | 1 | 1 | 1

Score: 45
State:
Dice: 3 3 5 5 6
Rolls Left: 0
Scorecard: [Ones 1 | Twos 1 | Threes 1 | Fours 1 | Fives 1 | Sixes 1 | Three of A Kind 1 | Four of a Kind 0 | Full House 0 | Small Straight 0 | Large Straight 0 | Yahtzee 1 | Chance 1 |]

ACTION
Four of a Kind

Score: 45
State:
Dice: 2 3 4 5 6
Rolls Left: 2
Scorecard: [Ones 1 | Twos 1 | Threes 1 | Fours 1 | Fives 1 | Sixes 1 |
Three of A Kind 1 | Four of a Kind 1 | Full House 0 | Small Straight 0 | Large Straight 0 | Yahtzee 1 | Chance 1 |]

ACTION
Large Straight

Score: 85
State:
Dice: 1 2 3 3 6
Rolls Left: 2
Scorecard: [Ones 1 | Twos 1 | Threes 1 | Fours 1 | Fives 1 | Sixes 1 |
Three of A Kind 1 | Four of a Kind 1 | Full House 0 | Small Straight 0 | Large Straight 1 | Yahtzee 1 | Chance 1 |]

ACTION
0 | 1 | 1 | 0 | 0

Score: 85
State:
Dice: 2 2 3 3 5
Rolls Left: 1
Scorecard: [Ones 1 | Twos 1 | Threes 1 | Fours 1 | Fives 1 | Sixes 1 |
Three of A Kind 1 | Four of a Kind 1 | Full House 0 | Small Straight 0 | Large Straight 1 | Yahtzee 1 | Chance 1 |]

ACTION
1 | 1 | 1 | 1 | 0

Score: 85
State:
Dice: 2 2 3 3 3
Rolls Left: 0
Scorecard: [Ones 1 | Twos 1 | Threes 1 | Fours 1 | Fives 1 | Sixes 1 |
Three of A Kind 1 | Four of a Kind 1 | Full House 0 | Small Straight 0 | Large Straight 1 | Yahtzee 1 | Chance 1 |]

ACTION
Full House

Score: 110
State:
Dice: 1 1 3 3 4
Rolls Left: 2
Scorecard: [Ones 1 | Twos 1 | Threes 1 | Fours 1 | Fives 1 | Sixes 1 |
Three of A Kind 1 | Four of a Kind 1 | Full House 1 | Small Straight 0 | Large Straight 1 | Yahtzee 1 | Chance 1 |]

ACTION
0 | 0 | 1 | 0 | 1

Score: 110
State:
Dice: 1 2 3 3 4
Rolls Left: 1
Scorecard: [Ones 1 | Twos 1 | Threes 1 | Fours 1 | Fives 1 | Sixes 1 |
Three of A Kind 1 | Four of a Kind 1 | Full House 1 | Small Straight 0 | Large Straight 1 | Yahtzee 1 | Chance 1 |]

ACTION
1 | 1 | 1 | 0 | 1

Score: 110
State:
Dice: 1 2 3 4 4
Rolls Left: 0
Scorecard: [Ones 1 | Twos 1 | Threes 1 | Fours 1 | Fives 1 | Sixes 1 |
Three of A Kind 1 | Four of a Kind 1 | Full House 1 | Small Straight 0 | Large Straight 1 | Yahtzee 1 | Chance 1 |]

ACTION
Small Straight

Final Score: 140

Note that this is an exactly average game for the agent.

Game Play Analysis (Minimalistic Reward Function)

The agent has a definitive and recognizable playstyle. I can say as a yahtzee player that it is not what I expected, which is exciting to me. Some things to note about its playstyle:

- It values yahtzee and four of a kind very low, and tends to score zero in those boxes quite often
- It values large straight the most, and full house the second most
- It sees small straight and three of a kind as a given, it will often leave these to go for last if it can
- It views scoring a high chance as very important → every single game I analyzed it explicitly goes for a high chance
    - This is by far the most surprising result

Some known and obvious optimal behavior it developed naturally:

- Scoring 6's is better than scoring 1's
    - this one is obvious
- Scoring in the four of a kind is better than scoring in the three of a kind if both are available
    - this one is less obvious and shows a clear understanding of long term strategy
- Storing two pairs is the best way to get the full house
    - This is not completely obvious, but interesting that it developed this strategy

Conclusion: This version of the yahtzee bot is very close to optimal → I believe wholeheartedly that when it gets to 21 iterations it will converge to the optimal play style. One interesting behavior that took a long time for it to learn is when it has a small straight already and a roll left, it often just scores in the small straight outright rather than going for the large straight. I have to imagine this will be fixed in the final iterations, but it is an interesting behavior for it to take this long for it to be developed as a strategy.

Game Play (Common Human Strategy Reward Function)

Score: 0
Turn 1
Dice: 1 1 3 4 5
Rolls Left: 2
Scorecard: [Ones 1 | Twos 1 | Threes 1 | Fours 1 | Fives 1 | Sixes 1 | Three of A Kind 0 | Four of a Kind 0 | Full House 0 | Small Straight 0 | Large Straight 0 | Yahtzee 0 | Chance 0 |]

ACTION
0 | 0 | 1 | 1 | 1

Score: 0
Turn 1
Dice: 1 3 3 4 5
Rolls Left: 1
Scorecard: [Ones 1 | Twos 1 | Threes 1 | Fours 1 | Fives 1 | Sixes 1 | Three of A Kind 0 | Four of a Kind 0 | Full House 0 | Small Straight 0 | Large Straight 0 | Yahtzee 0 | Chance 0 |]

ACTION
0 | 1 | 0 | 1 | 1

Score: 0
Turn 1
Dice: 2 3 4 4 5
Rolls Left: 0
Scorecard: [Ones 1 | Twos 1 | Threes 1 | Fours 1 | Fives 1 | Sixes 1 | Three of A Kind 0 | Four of a Kind 0 | Full House 0 | Small Straight 0 | Large Straight 0 | Yahtzee 0 | Chance 0 |]

ACTION
Small Straight

Score: 30
Turn 2
Dice: 1 3 4 5 5
Rolls Left: 2
Scorecard: [Ones 1 | Twos 1 | Threes 1 | Fours 1 | Fives 1 | Sixes 1 | Three of A Kind 0 | Four of a Kind 0 | Full House 0 | Small Straight 1 | Large Straight 0 | Yahtzee 0 | Chance 0 |]

ACTION
0 | 0 | 0 | 1 | 1

Score: 30
Turn 2
Dice: 3 4 5 5 6
Rolls Left: 1
Scorecard: [Ones 1 | Twos 1 | Threes 1 | Fours 1 | Fives 1 | Sixes 1 | Three of A Kind 0 | Four of a Kind 0 | Full House 0 | Small Straight 1 | Large Straight 0 | Yahtzee 0 | Chance 0 |]

ACTION
0 | 0 | 1 | 1 | 0

Score: 30
Turn 2
Dice: 1 2 5 5 5
Rolls Left: 0
Scorecard: [Ones 1 | Twos 1 | Threes 1 | Fours 1 | Fives 1 | Sixes 1 | Three of A Kind 0 | Four of a Kind 0 | Full House 0 | Small Straight 1 | Large Straight 0 | Yahtzee 0 | Chance 0 |]

ACTION
Three of A Kind

Score: 48
Turn 3
Dice: 1 1 2 3 3
Rolls Left: 2
Scorecard: [Ones 1 | Twos 1 | Threes 1 | Fours 1 | Fives 1 | Sixes 1 | Three of A Kind 1 | Four of a Kind 0 | Full House 0 | Small Straight 1 | Large Straight 0 | Yahtzee 0 | Chance 0 |]

ACTION
1 | 1 | 0 | 1 | 1

Score: 48
Turn 3
Dice: 1 1 3 3 5
Rolls Left: 1
Scorecard: [Ones 1 | Twos 1 | Threes 1 | Fours 1 | Fives 1 | Sixes 1 | Three of A Kind 1 | Four of a Kind 0 | Full House 0 | Small Straight 1 | Large Straight 0 | Yahtzee 0 | Chance 0 |]

ACTION
1 | 1 | 1 | 1 | 0

Score: 48
Turn 3
Dice: 1 1 3 3 6
Rolls Left: 0
Scorecard: [Ones 1 | Twos 1 | Threes 1 | Fours 1 | Fives 1 | Sixes 1 | Three of A Kind 1 | Four of a Kind 0 | Full House 0 | Small Straight 1 | Large Straight 0 | Yahtzee 0 | Chance 0 |]

ACTION
Chance

Score: 62
Turn 4
Dice: 1 2 3 5 5
Rolls Left: 2
Scorecard: [Ones 1 | Twos 1 | Threes 1 | Fours 1 | Fives 1 | Sixes 1 | Three of A Kind 1 | Four of a Kind 0 | Full House 0 | Small Straight 1 | Large Straight 0 | Yahtzee 0 | Chance 1 |]

ACTION
0 | 0 | 0 | 1 | 1

Score: 62
Turn 4
Dice: 4 5 5 5 5
Rolls Left: 1
Scorecard: [Ones 1 | Twos 1 | Threes 1 | Fours 1 | Fives 1 | Sixes 1 | Three of A Kind 1 | Four of a Kind 0 | Full House 0 | Small Straight 1 | Large Straight 0 | Yahtzee 0 | Chance 1 |]

ACTION
0 | 1 | 1 | 1 | 1

Score: 62
Turn 4
Dice: 4 5 5 5 5
Rolls Left: 0
Scorecard: [Ones 1 | Twos 1 | Threes 1 | Fours 1 | Fives 1 | Sixes 1 | Three of A Kind 1 | Four of a Kind 0 | Full House 0 | Small Straight 1 | Large Straight 0 | Yahtzee 0 | Chance 1 |]

ACTION
Four of a Kind

Score: 86
Turn 5
Dice: 1 2 4 4 5
Rolls Left: 2
Scorecard: [Ones 1 | Twos 1 | Threes 1 | Fours 1 | Fives 1 | Sixes 1 | Three of A Kind 1 | Four of a Kind 1 | Full House 0 | Small Straight 1 | Large Straight 0 | Yahtzee 0 | Chance 1 |]

ACTION
1 | 1 | 1 | 0 | 1

Score: 86
Turn 5
Dice: 1 2 3 4 5
Rolls Left: 1
Scorecard: [Ones 1 | Twos 1 | Threes 1 | Fours 1 | Fives 1 | Sixes 1 | Three of A Kind 1 | Four of a Kind 1 | Full House 0 | Small Straight 1 | Large Straight 0 | Yahtzee 0 | Chance 1 |]

ACTION
Large Straight

Score: 126
Turn 6
Dice: 3 4 6 6 6
Rolls Left: 2
Scorecard: [Ones 1 | Twos 1 | Threes 1 | Fours 1 | Fives 1 | Sixes 1 | Three of A Kind 1 | Four of a Kind 1 | Full House 0 | Small Straight 1 | Large Straight 1 | Yahtzee 0 | Chance 1 |]

ACTION
0 | 0 | 1 | 1 | 1

Score: 126
Turn 6
Dice: 2 4 6 6 6
Rolls Left: 1
Scorecard: [Ones 1 | Twos 1 | Threes 1 | Fours 1 | Fives 1 | Sixes 1 | Three of A Kind 1 | Four of a Kind 1 | Full House 0 | Small Straight 1 | Large Straight 1 | Yahtzee 0 | Chance 1 |]

ACTION
0 | 0 | 1 | 1 | 1

Score: 126
Turn 6
Dice: 2 4 6 6 6
Rolls Left: 0
Scorecard: [Ones 1 | Twos 1 | Threes 1 | Fours 1 | Fives 1 | Sixes 1 | Three of A Kind 1 | Four of a Kind 1 | Full House 0 | Small Straight 1 | Large Straight 1 | Yahtzee 0 | Chance 1 |]

ACTION
Yahtzee

Score: 126
Turn 7
Dice: 2 2 2 5 6
Rolls Left: 2
Scorecard: [Ones 1 | Twos 1 | Threes 1 | Fours 1 | Fives 1 | Sixes 1 |
Three of A Kind 1 | Four of a Kind 1 | Full House 0 | Small Straight 1 | Large Straight 1 | Yahtzee 1 | Chance 1 |]

ACTION
1 | 1 | 1 | 1 | 0

Score: 126
Turn 7
Dice: 2 2 2 5 6
Rolls Left: 1
Scorecard: [Ones 1 | Twos 1 | Threes 1 | Fours 1 | Fives 1 | Sixes 1 |
Three of A Kind 1 | Four of a Kind 1 | Full House 0 | Small Straight 1 | Large Straight 1 | Yahtzee 1 | Chance 1 |]

ACTION
1 | 1 | 1 | 1 | 0

Score: 126
Turn 7
Dice: 2 2 2 2 5
Rolls Left: 0
Scorecard: [Ones 1 | Twos 1 | Threes 1 | Fours 1 | Fives 1 | Sixes 1 |
Three of A Kind 1 | Four of a Kind 1 | Full House 0 | Small Straight 1 | Large Straight 1 | Yahtzee 1 | Chance 1 |]

ACTION
Full House

Final Score: 126

Note that this is a slightly below average game for the agent.

Game Play Analysis (Common Human Strategy)

As you can see, it makes very intelligent decisions. In the first turn, it stores ‘3,4,5’ multiple times, as it goes for the large straight. Also, it does not store the 1, even though that would bring it slightly closer to a large straight, it is riskier.

Another example is on turn 2, it keeps the fives, and nothing else, as this leads to the highest chance of scoring in three of a kind, four of a kind, full house, or yahtzee. In the end it has to settle for three of a kind.

On turn 3, it gets very close to a full house on the first roll. It decides to keep the 2 pairs and go for it on the following rolls but doesn't get it, storing in its chance instead.

CONCLUSION: In general, I can say as an avid yahtzee player, this seems very close to optimal behavior, given the strategy outlined by the reward function. At the time of writing this report, value iteration is being run with the minimalistic reward function, and I am very interested to see the results after 21 iterations. If you are interested in the results of value iteration with minimalistic reward, shoot me an email at dunnlu@oregonstate.edu, and I should have the full results by Thursday, June 13th.

Q-Learning. (San Davran)

Algorithm.

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  until S is terminal

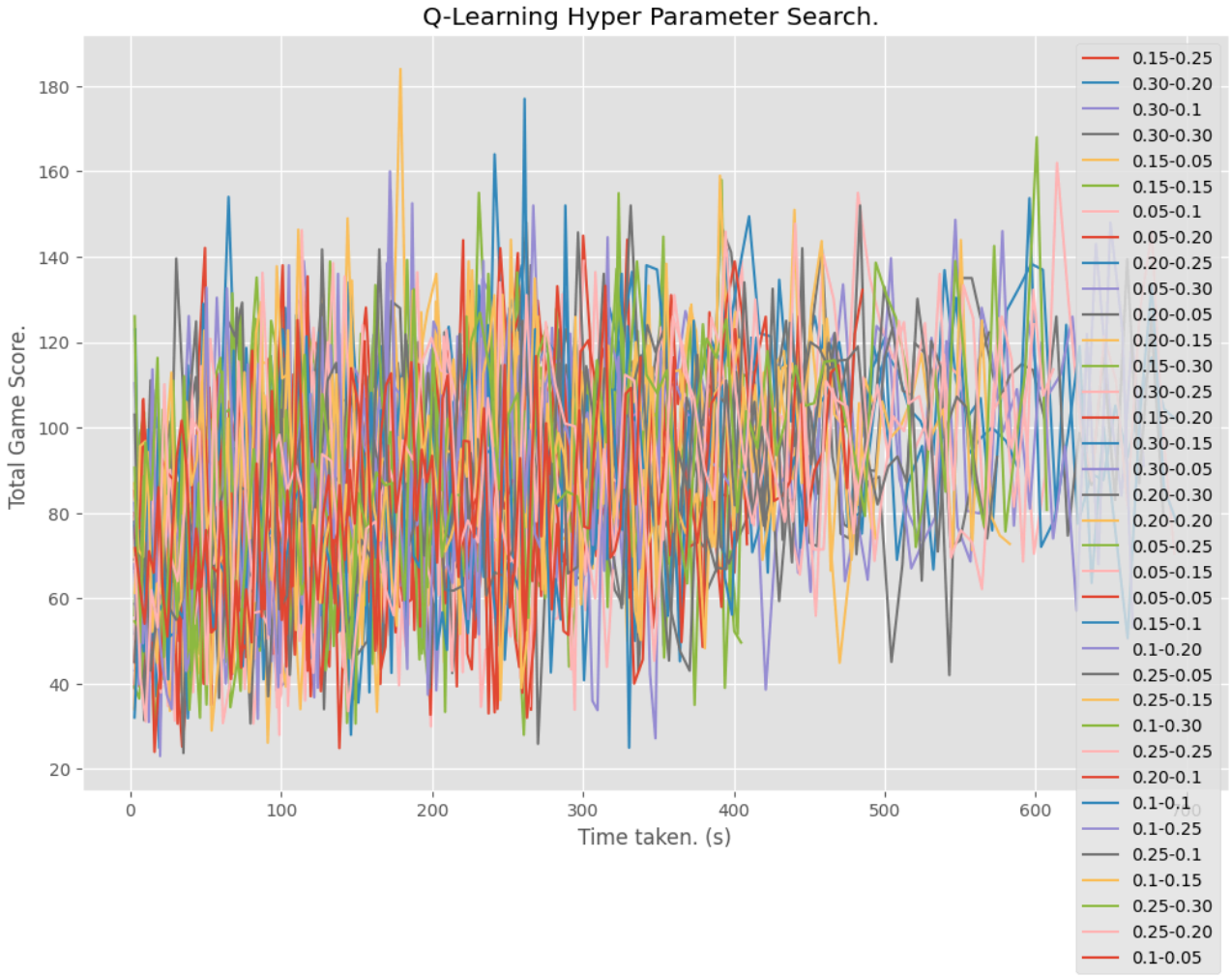
Methods.

Hyper Parameter Search.

- I have conducted a hyper parameter search, in order to determine the best epsilon and alpha values. I have tried all of the combinations using the list of possible epsilon values and the list of possible alpha values. These are the 2 lists of hyper parameter values:
  - Epsilon: 0.05, 0.1, 0.15, 0.2, 0.25, 0.3.
  - Alpha: 0.05, 0.1, 0.15, 0.2, 0.25, 0.3.
- For each hyper parameter combination, a policy is trained with the following structure:
  - For 1000 iterations:
    - Train the policy using 1000 episodes.
    - Evaluate the policy using 1000 episodes, average the results, return the result.
  - All of the algorithms are stopped after around 10 minutes, in order to evaluate the results. (this allows for quick iteration, since anything above this time frame, I would need to use the CoE DGX machines or the interactive desktop, so that the process is not terminated when I log off) (I had a lot of issues using the bindings in different machines)
  - In all of the plots for the hyper parameter search, the first value is the epsilon value, and the second value is the alpha value.
- The hyperparameter search was completed using the whole scorecard, I then stopped using the whole scorecard and just used the lower section of the scorecard, since the value iteration is trained this way, so that the comparison is fair.

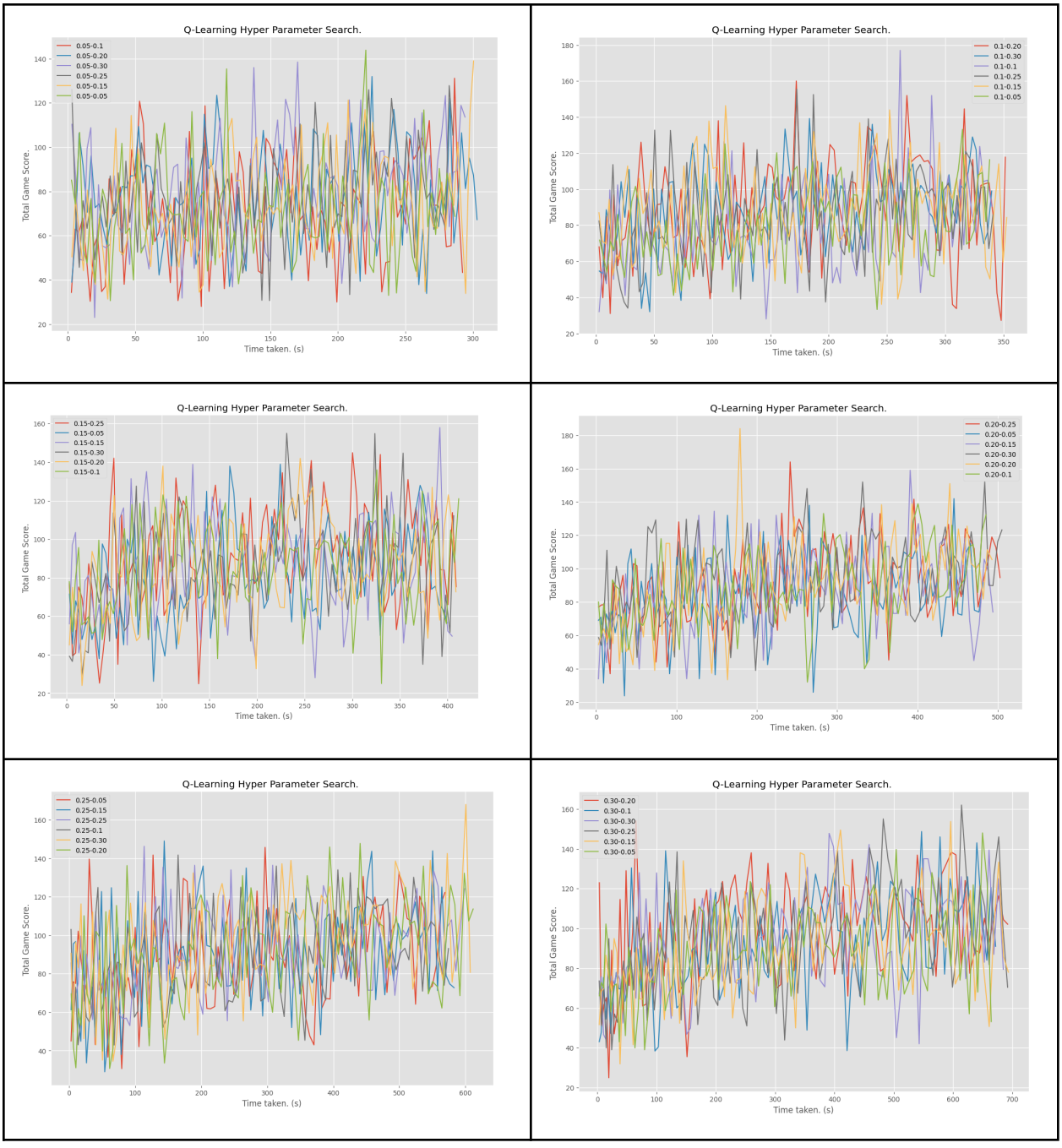
HPS - General.

- Here is a graph of all of the algorithms used for the hyper parameter search.



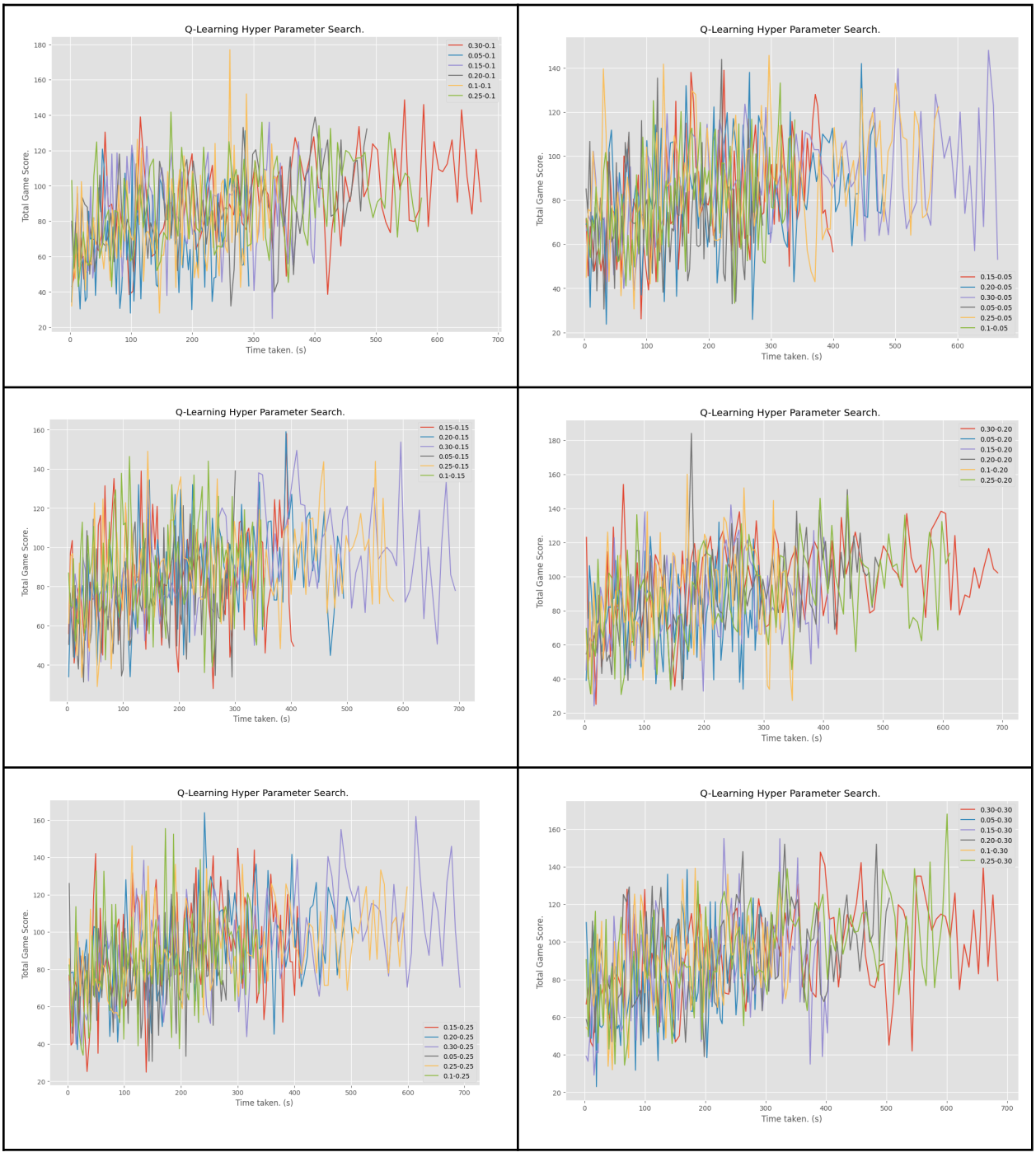
HPS - Fix epsilon.

- For each epsilon value, I have plotted the graphs of policies with that certain epsilon value.



HPS - Fix alpha.

- For each alpha value, I have plotted the graphs of the policies with that certain alpha value.



HPS - Insights.

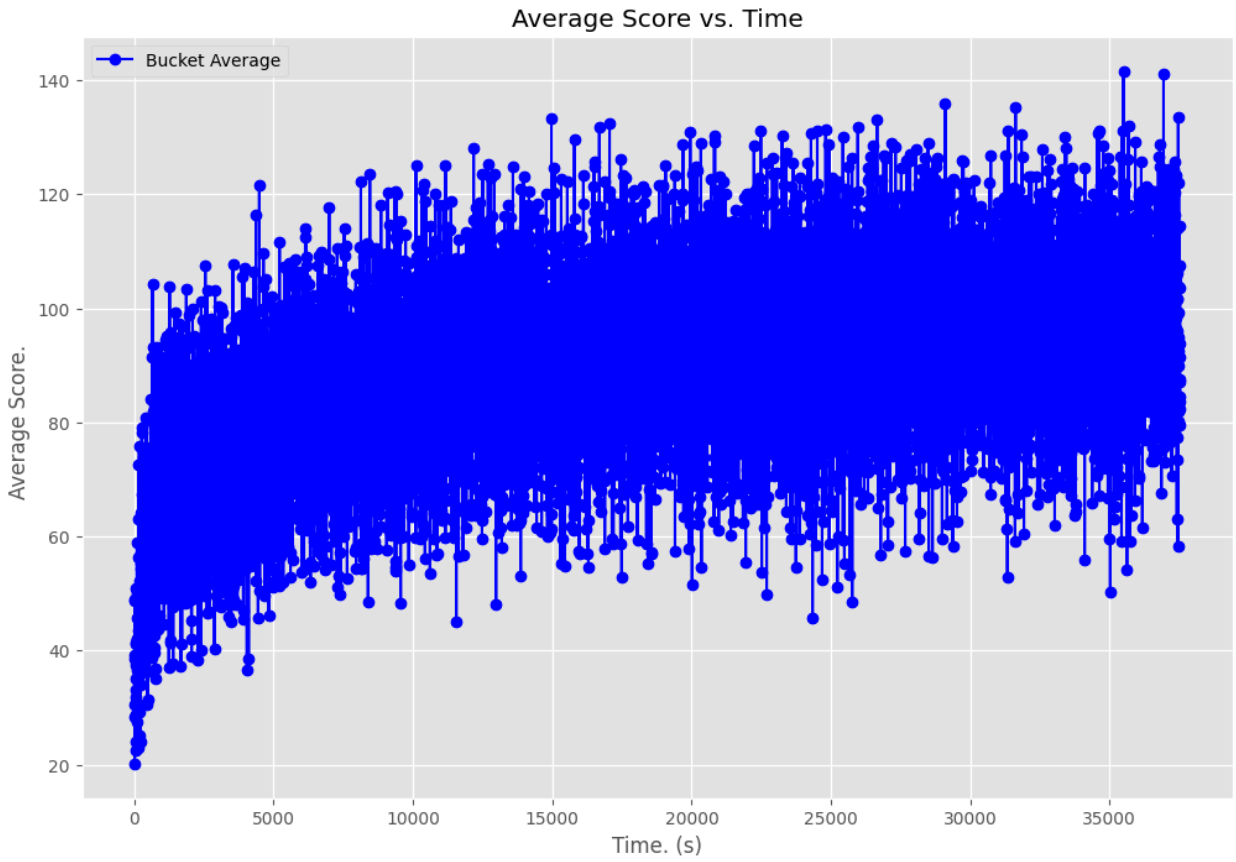
- In the first 10 minutes, the policy is mostly random, therefore there is a lot of variance in the results. This is using a trial count of 1000 for each evaluation run. There aren't any significant trends, however, the policies with low epsilon and/or alpha values, have a constant average, while policies with high epsilon and/or alpha values, have an increasing average. The trend is slight, however it is possible to understand.
    - Therefore, I have decided to use the epsilon value of 0.3, and the alpha value of 0.3.

Training.

- The policy is trained with an epsilon value of 0.3, alpha value of 0.3. In order to produce better graphs, I have decided to decrease the batch size from 1000 to 100, (evaluate more frequently, so that the evaluated policies are similar to each other) and increase the batch count from 1000 to 10000 so that the total amount of episodes used for just training is 1M. I have also increased the trial count from 1000 to 10000 because of the huge variance in the results.
- Therefore, the training structure is this:
    - For 10000 iterations:
        - Train the policy using 100 episodes.
        - Evaluate the policy using 10000 episodes, average the results, return the result.

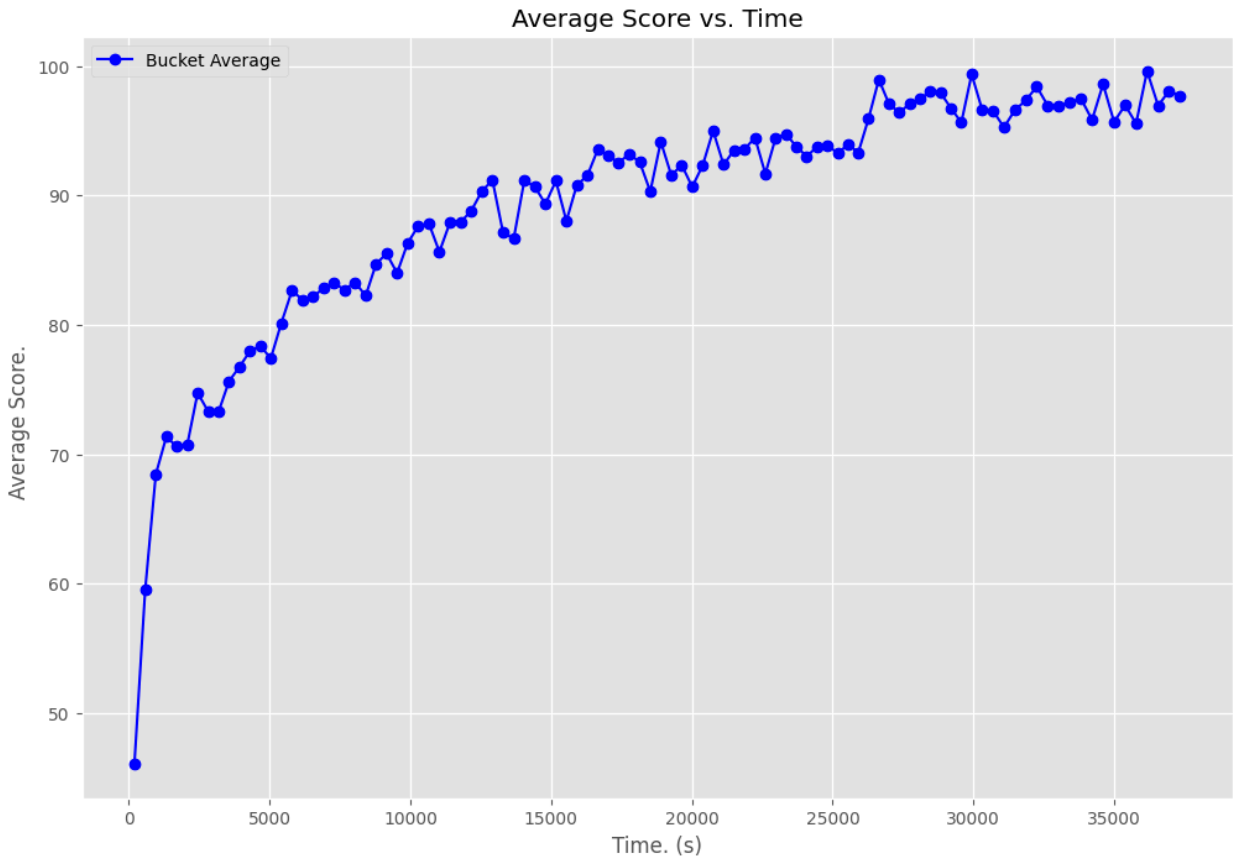
Evaluation.

- Here is the plot of training:



- We are aware that the maximum score for the version of Yahtzee that just uses the lower part of the score-card is 235.
- Even with trial count 10000, the variance of the score that the agent got was quite a lot. It looks like the algorithm is actually not improving. Therefore, I have decided to use a method to make sure there are less fluctuations in the graph.

The bucket size of 100:



- Without a decreasing epsilon value, the policy is improving, the score peaks around a value of 95.

Game Play.

- I have printed an episode that both of the policies were tested after the training was completed.
- Here is the just_lower:

```
state=[1, 3, 4, 5, 6, 2, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0].
action=[0, 0, 1, 1, 1, -1].
Re-roll the dice 1 and 3.
reward=0.
state=[2, 4, 5, 5, 6, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0].
action=[0, 1, 1, 1, 1, -1].
Re-roll the dice 2.
reward=0.
state=[3, 4, 5, 5, 6, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0].
action=[1, 1, 1, 1, 1, 12].
Score a chance.
reward=23.
state=[1, 2, 3, 4, 5, 2, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1].
action=[1, 1, 1, 1, 1, 10].
Score a large straight.
reward=40.
state=[3, 4, 4, 5, 6, 2, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 1].
action=[0, 1, 0, 0, 0, -1].
Re-roll the dice 3, 4, 5, and 6.
reward=0.
state=[2, 3, 4, 4, 6, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 1].
action=[0, 0, 1, 0, 1, -1].
Re-roll the dice 2, 3, and 4.
reward=0.
state=[3, 4, 5, 6, 6, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 1].
action=[1, 1, 1, 1, 1, 9].
Score a small straight.
reward=30.
state=[2, 3, 3, 5, 6, 2, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1].
action=[1, 1, 0, 1, 0, -1].
Re-roll the dice 3, and 6.
reward=0.
state=[2, 2, 3, 5, 6, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1].
action=[0, 0, 0, 0, 1, -1].
Re-roll the dice 2, 2, 3, and 5.
reward=0.
state=[2, 2, 6, 6, 6, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1].
action=[1, 1, 1, 1, 1, 6].
```

Score a 3 of a kind.
reward=22.
state=[2, 2, 3, 6, 6, 2, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1].
action=[1, 1, 0, 1, 1, -1].
Re-roll the dice 3.
reward=0.
state=[2, 2, 6, 6, 6, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1].
action=[1, 1, 1, 1, 1, 8].
Score a full house.
reward=25.
state=[2, 4, 5, 5, 6, 2, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1].
action=[0, 0, 0, 1, 0, -1].
Re-roll the dice 2, 4, 5, and 6.
reward=0.
state=[2, 3, 4, 4, 5, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1].
action=[1, 1, 0, 0, 1, -1].
Re-roll the dice 4.
reward=0.
state=[2, 3, 4, 5, 5, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1].
action=[1, 1, 1, 1, 1, 11].
Score a Yahtzee.
reward=0.
state=[2, 2, 3, 3, 4, 2, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1].
action=[1, 1, 0, 0, 0, -1].
Re-roll the dice 3, 3, and 4.
reward=0.
state=[2, 2, 2, 4, 4, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1].
action=[1, 1, 1, 1, 0, -1].
Re-roll the dice 4.
reward=0.
state=[2, 2, 2, 2, 4, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1].
action=[1, 1, 1, 1, 1, 7].
Score a 4 of a kind.
reward=12.
The score of the agent: 152.

*Note that this is a well-above average game for the agent.

- After the training is complete, the policy is evaluated for 1M episodes and the average is taken.
- Average result: (just_lower) 95.426672.

Every Visit Monte Carlo

Note: EVMC: Every Visit Monte Carlo, FVMC: First Visit Monte Carlo

Introduction

To start, it is important to mention that due to the chosen state representation, any state can be visited only once per episode. Only being able to visit any state once per episode, effectively converts EVMC algorithm into a FVMC algorithm.

In this implementation, an epsilon-greedy policy will be used to augment the FVMC algorithm with an Exploration-Exploitation Tradeoff ability.

Pseudocode

Input: epsilon (ϵ), gamma (γ), number of episodes
Output: Q(s, a) (can be turned into a policy by selecting the highest valued action for each state)

Initialize: Q(s, a) to all 0s for each (s, a) pair. Returns(s, a) to empty lists for each (s, a)
Repeat for “number of episodes”:

1) Use the epsilon-greedy policy to generate an episode
   $[(S_1, A_1, R_1), (S_2, A_2, R_2), \dots, (S_T, A_T, R_T)]$
2) For each (s, a) appearing in the episode:
   a) t = time of first occurrence of (s, a) in the episode
   b) $G = \sum_{k=0}^{T} \gamma^k R_{t+k}$
   c) Append G to Returns(s,a)
   d) Q(s,a) = Average(Returns(s,a))

As a result, we get an action-value function (Q).

Note: epsilon greedy policy pseudocode is shown below.

- $A^* = \arg\max_a Q(s, a)$
- For all $a \in A$:
  - $\pi(s, a) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|A|} & \text{if a} = A^* \\ \frac{\epsilon}{|A|} & \text{if a} \neq A^* \end{cases}$

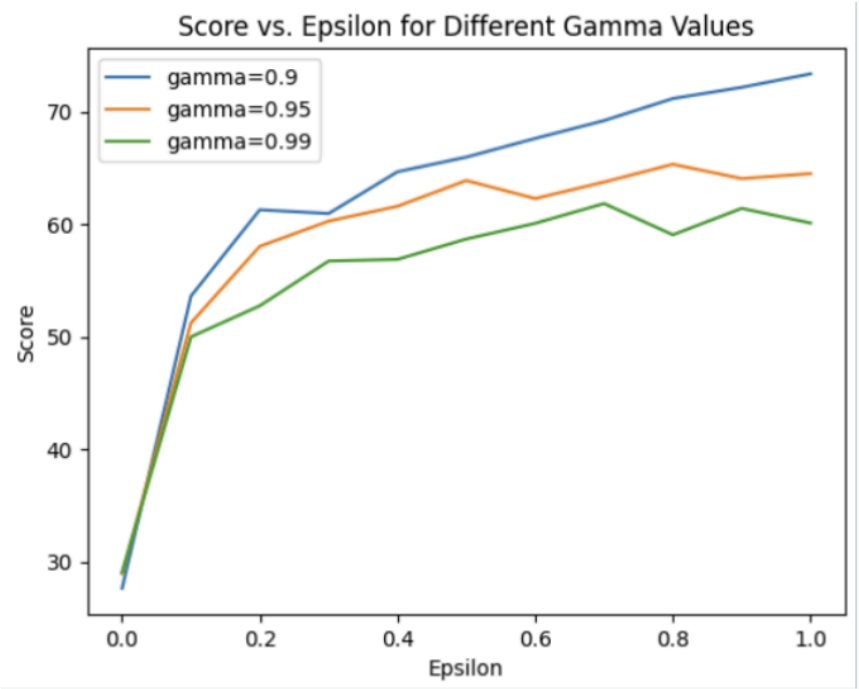
To implement it, when choosing an action at a state during training, we can look at our current Q(s, a) to find the A*, and compute the probabilities for other possible actions without saving them to some policy. The probabilities for each (s, a) for a given state are re-calculated instead of being saved somewhere.

Hyperparameter Search

Epsilon: [0, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5, 0.55, 0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95, 1]
Gamma: [0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95, 0.99]
Number of episodes: [500_000,1_000_000, …, 80_000_000]

Epsilon

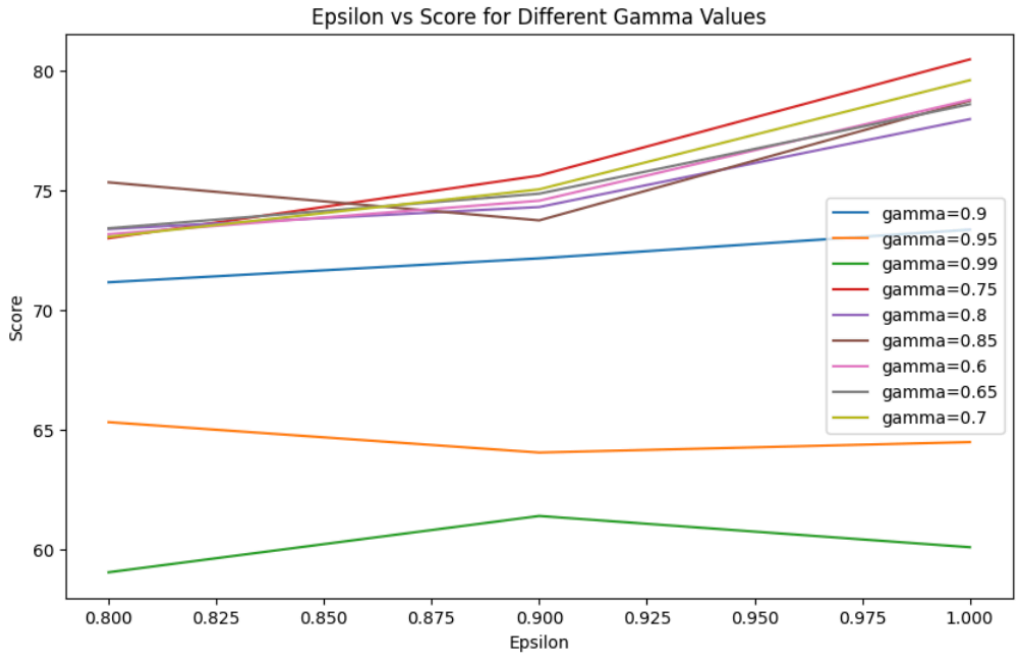
Initially, only a few values of Gamma were tested along with all values of epsilon



The figure above shows the average performance (score, averaged over 1000 runs) of the trained models with the following parameters.
Number of episodes: 10_000_000
Gamma: [0.9, 0.95, 0.99]
Epsilon: [0, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5, 0.55, 0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95, 1]
A pattern of increasing performance (score) as the epsilon is increased is clear.
Also, a consistent pattern of lower gamma leading to better returns is present.

In later stages, only values [0.8, 0.85, 0.9, 0.95, 1] of epsilon will be used to explore lower gamma values. This is done to use the time/compute more efficiently. This choice relies on the assumption that the hypothesis “performance (score) increases as the epsilon is increased” is generally true. There is still some buffer ([0.8, 0.85, 0.9, 0.95, 1] instead of just using 0.99) to account for possibly changing behavior with different values of gamma.

Gamma



The figure above shows the average performance (score, averaged over 1000 runs) of the trained models with the following parameters.
Number of episodes: 10_000_000
Gamma: [0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95, 0.99]
Epsilon: [0.8, 0.85, 0.9, 0.95, 1]

A pattern of higher values of gamma (>0.9) performing worse than others is evident.
0.99 = worst, 0.95 = second worst, 0.9 = third worst.
The rest of the values seem to have very similar performances, with no clear linear pattern of increasing performance as gamma is increased/decreased.
The best performance was observed with gamma = 0.75, epsilon = 1.

Number of episodes

The search for the best number of episodes consisted of running a few hand-picked epsilon and gamma combinations and running them for different number of episodes.
Number of episodes: [500_000,1_000_000, …, 80_000_000] (in increments of 500_000)
No notable/stable improvement has occurred after ~10_000_000 episodes, with some not showing improvement after 1_000_000 episodes. So a value of 10_000_000 was used as a “safe” choice for the rest of the parameter search.

Hyperparameter Search Results

Gamma: 0.75

Considering the chosen state representation, transition, and reward function, a discount factor that can propagate back a future reward upon rerolling the dice (potential reward is always 1, 2, or 3 actions away) is needed, so very low values would not work well as the agent would not see a meaningful reward for rerolling the dice when needed and over appreciate the immediate reward of scoring even if sub-optimally.

On the other hand, a high value of gamma would perform poorly as this game's turns are highly independent of each other and are highly unpredictable (hundreds of successor states), so learning from a reward that the agent gets from more than 3 steps away is not useful, and can give a false evaluation of the current action, in case an agent "gets lucky" a few rolls down the line after making a factually bad choice (and the rewards propagates back and makes it seem like the current action was good).

Epsilon: 1

This was a bit of a surprise for me. Given that most (state, action) pairs have many hundreds of potential successor states, exploring as many of them as possible will end up giving a better idea about the environment in general.

In contrast, if we set epsilon to 0, we will be taking random actions for every new state, and repeating the one action that we already performed I(also at random) in the states that we have already visited, which will be equivalent to the performance of taking fully random actions all the time.

Setting epsilon higher, allows us to try new actions (39 unique actions in most states, each leading to ~852 successor states), and at least get some idea of what actions with immediate reward are better. For example, if the dice is (3, 3, 3, 3, 3) and the "Yahtzee" box is available, only 1 out of 39 possible actions will lead to 50 point reward, and all others lead to 0 immediate reward. The agent would need to do a lot of exploration to be able to try taking that action, see how good it is, and remember to do it again when given a chance. But taking the same action twice to get a more accurate evaluation of that (s, a) pair is not as valuable as getting a rough estimate of the new unique one.

Ultimately, I believe that the sheer number of possible actions and possible successor states is what makes the exploration (epsilon: 1) as valuable as it has shown itself to be.

Every Visit Monte Carlo + Exploration/Exploitation Tradeoff for Yahtzee

Strengths: No need for a transition function (can simulate rolling the dice, and generate episodes that way), sampling the (state, action) pairs and averaging the observed cumulative rewards can estimate the value of that (state, action) pair well, ability to utilize the epsilon-greedy policy to focus on exploration showed to be effective.

Weaknesses: Unique structure of “turns” versus “states” is hard to represent accurately in the reward function. Ex. turns can be ((dice, reroll) →(dice, reroll) →(dice, reroll) →(dice, score) = reward), or ((dice, reroll) →(dice, reroll) →(dice, score) = reward), or ((dice, reroll) →(dice, score) = reward), or ((dice, score) = reward). Each of these sequences is a single turn in Yahtzee, but the nature of this structure is hard to represent accurately as one continuous list of states and actions. So an agent can see 3 rewards in a row after scoring 3 times, or a single reward in 3 consecutive (state, action) pairs, but that one reward might have been more optimal. But the gamma-discounted reward makes it seem like actions that are closer to scoring are better no matter what (which is not the case). Also, the high variance of Monte Carlo approaches in general, combined with vast state and action spaces, makes this algorithm need a lot more time to be able to visit all of the states enough times to have meaningful averaged-out values for its actions.

Contributions

- Danil Kryuchkov:
  - Every Visit Monte Carlo (Report and Code)
  - MDP: State Representation (report only)
  - MDP: Action Representation (report only)
- San Davran:
  - Q-Learning (report and code)
  - MDP (large part of the design)
  - Python Bindings (used by EVMC and Q-Learning)
  - Game (a small part of it, the step function that q-learning and evmc is using)
- Lucas Dunn:
  - Value Iteration (Report and Code)
  - MDP: State Implementation (Code)
  - MDP: Action Implementation (Code)
  - MDP Reward (report and code)
  - MDP Transition (report and code)
  - Game (majority of code, including entire MDP and basic environment structure)
  - Project Proposal
  - Baseline