Reinforcement Learning for Solving Yahtzee

Minhyung Kang
dankang@stanford.edu

Luca Schroeder
lucschr@stanford.edu

***Abstract*—This paper presents a reinforcement learning approach to the famous dice game Yahtzee. We outline the challenges with traditional model-based and online solution techniques given the massive state-action space, and instead implement global approximation and hierarchical reinforcement learning methods to solve the game. Our best agent is able to consistently outperform naïve benchmarks but remains suboptimal; we highlight difficulties and avenues for future improvement.**

I. Introduction

Yahtzee is a well-known dice game in which the goal is to achieve a high score by rolling certain dice combinations (e.g. 'three of a kind,' 'full house,' etc.) [17]. Though seemingly simple, the state-action space for Yahtzee is extremely large; a multi-player game can have quadrillions of states, rendering traditional model-based solvers ineffective. In this paper, we discuss the general challenges for solving Yahtzee and present a reinforcement learning approach. In particular, we implement a Yahtzee simulator and two reinforcement learning agents: the first employs perceptron Q-learning with eligibility traces, while the second uses hierarchical learning. The hierarchical learning agent is the most successful and is able to consistently outperform a random agent and a naïve greedy benchmark, although the achieved scores remain suboptimal relative to brute-force forward search online methods with limited look-ahead.

The remainder of the paper proceeds as follows:

- Section II reviews prior work, including previous approaches for solving single-player Yahtzee;
- Section III outlines the rules and structure of Yahtzee;
- Section IV discusses complexity of the game and the challenges with model-based or online techniques;
- Section V details our reinforcement learning approach;
- Sections VI and VII present our evaluation strategy and analyze our experimental results.

II. Related Work

Reinforcement learning has been successful in game playing in many instances. Examples range from backgammon and Atari hits such as Breakout and Pong [10] to more complicated games such as Starcraft [11] and Super Smash Brothers [4].

For Yahtzee in particular, prior work [5], [15] has identified an optimal strategy that maximizes the expected score. It is important to note that this strategy does not maximize the chance of winning. To maximize the chance of winning, you need to maximize the probability that your score is above your opponent's score; the final margin is irrelevant. In other words, rather than trying to blindly gain the highest point total possible, one might play a more conservative strategy (with a higher chance of success) to just barely beat the opponent's score.

One of the approaches of reinforcement learning that we have seen much progress in is hierarchical reinforcement learning, which discovers and exploits a hierarchical structure within a Markov decision problem. [1], [16]. Several methods utilize semi-Markov decision processes (SMDP) [6], in which time steps between decisions are of a fixed duration. The *Option* approach generalizes the idea of an action that results in state transitions of variable and extended duration [14]. This allows one to discretize an SMDP and understand it in relation to the underlying MDP. *Hierarchical Abstract Machines (HAM)* defines policies as hierarchies of stochastic finite-state machines, constraining the action space and allowing for application of prior knowledge about the environment [12]. *MAXQ Value decomposition* is another hierarchical method that we explore in this paper; it decomposes the core MDP into a hierarchy of subtasks and attempts to solve the multiple SMPDs simultaneously, rather than a single one as with options or HAM.

III. The Game of Yahtzee

Each game of Yahtzee is divided into thirteen rounds. During a round, each player rolls five dice; the player can roll the dice up to two more times and can choose which dice in particular to re-roll. After the player is done rolling their dice, they assign their ending roll to one of thirteen categories on their score-sheet. The numerical score is then calculated based on how well the roll matches to that category. Once a category has been chosen by a player, it cannot be chosen again by that player (each player has their own score-sheet). The game ends when all categories have score values, and the player with the highest total score wins.

To give a concrete example, let us imagine a player who initially rolls a 1,2,2,3,5. The player chooses to re-roll the 3 and 5 and obtains a 2 and a 4, and re-rolls the 4 to obtain another 2. The ending roll is 1,2,2,2,2. The player looks at the score-sheet and assigns the roll to the category 'Twos'; the score rule for this category is the sum of all the dice that rolled 2—in this case $2+2+2+2=8$. The player's round is concluded.

The Yahtzee scoresheet [7] and its categories are broken down in Table I. If a total score of 63 or greater is achieved in the upper section, the player receives a bonus of 35 points. We simplify the game by ignoring multiple Yahtzee rules and Joker rules.

Upper Section		Lower Section	
Category	*Score*	*Category*	*Score*
Aces	Sum of 1s	3 of a Kind	Sum of all dice
Twos	Sum of 2s	4 of a Kind	Sum of all dice
Threes	Sum of 3s	Full House	25
Fours	Sum of 4s	Small Straight	30
Fives	Sum of 5s	Large Straight	40
Sixes	Sum of 6s	YAHTZEE	50
-	-	Chance	Sum of all dice

TABLE I: Scoresheet Categories

IV. Challenges

Yahtzee is deceptively simple: what seems to be a child's dice game is actually non-trivial to solve. A large part of this arises from the massive size of the state space. At a high level, a state in a multi-player Yahtzee game must contain the following information:

- For each player, binary values indicating which of the thirteen categories on the scoresheet are still available;
- The values on each of the five six-sided dice;
- The number of rerolls remaining (0, 1, or 2) for the current player;
- For each player, their score (to determine bonuses and winner).

The number of possible scores is on the order of a hundred values, so for an n-player Yahtzee game the number of states is $\approx (2^{13} \cdot 100)^n \cdot 6^5 \cdot 3$. For $n = 2$, the number of states is on the order of 15 quadrillion. Small optimizations can be made to reduce the size of the state space by a few orders of magnitude; for instance, prior work has highlighted that since the ordering of the dice doesn't matter, we can reduce the $6^5 = 7,776$ possible ordered dice rolls to $\binom{10}{5} = 252$ unordered dice rolls by the stars and bars combinatorial trick [5]. But in any case the reductions are only modest; although the dynamics of Yahtzee are well-known, it therefore is infeasible to materialize the model, such as a transition matrix $T(s, a, s')$, and solve for the optimal policy using policy or value iteration.

The space of possible actions is fortunately smaller than the state space: for each turn, there are up to 44 possible actions: at most 13 categories to assign a roll to and (if you have rerolls remaining) $2^5 - 1 = 31$ choices of dice to reroll. However, since a single player takes up to 39 actions per game, the policy space is still very large and direct policy search is not practical.

Nor, unfortunately, are online methods particularly useful for this problem. This is since if you have a reroll remaining you can choose to reroll all the dice which can bring you to any one of 6^5 dice configurations, so a large portion of the state space is reachable from most game states. Indeed, there are up to hundreds of millions of possible paths between states within a single round.

V. Approach

Although the model of Yahtzee is known, as argued in the previous section the problem's complexity makes model-free learning—where the transition and reward matrices do not have to be materialized—relatively attractive. Indeed, the principal approach of this paper is model-free reinforcement learning. However, the size of the state-action space rules out basic Q-learning and Sarsa, as it is still too memory-intensive to have a Q matrix for each possible (s, a). Even if the Q-matrix could comfortably fit in memory, given the fact that at most $39n$ state-action combinations are seen in one n-player Yahtzee game, it is necessary to either: run at least tens of millions of game simulations to properly fill out the Q-matrix; or devise an interpolation strategy to fill out zeros after a shorter training run. The latter is non-trivial for Yahtzee, as it is not at all clear how to define a reasonable distance metric between states. Consider for instance two states where the dice roll is $[5, 5, 5, 5, 5]$ in both. In the first state, all categories are taken except the 'Fives' category; in the second state, all categories are taken except the 'Sixes' category. The states are identical except for two binary values for the categories, yet for the first state $Q(s, a) = 25$ and for the second state $Q(s, a) = 0$ (assuming this is the last turn). Thus a naïve distance metric simply would not work.

Our approach instead leverages global approximation, which has no dependence on a distance metric. In particular, we employ perceptron Q-learning, defining a set of m basis functions or features $\beta_1, \ldots, \beta_m$ over the state space, and learning parameter $\theta_a \in \mathbb{R}^m$ for each action a such that

$$Q(s, a) = \theta_a^T \beta(s)$$

So for each observation (s, a, r, s'), we perform update

$$\theta_a \leftarrow \theta_a + \alpha(r + \gamma \max_a \theta_a^T \beta(s') - \theta_a^T \beta(s))\beta(s)$$

for some learning rate α and discount factor γ. We anneal the learning rate α with step decay, decreasing α by a constant factor after every game during the learning phase.

A. Rewards and Eligibility Traces

Additionally, to speed up learning we used eligibility traces to assign credit to past states and actions; this is especially critical in game settings such as ours where rewards are typically sparsely distributed. We keep two sets of traces: local traces, which keep track of the actions taken within the current round; and global traces, which keep track of the actions taken throughout the entire game. At the end of a round, we give a reward proportional to the score achieved in that round; since this is only indirectly related to the actions taken in other rounds we propagate this reward backwards along the local trace. At the end of a game, we give a reward proportional to the score achieved in the entire game and (optionally) a bonus/penalty depending on whether a win or loss occurred. Since all actions taken in the game contribute to overall victory, this reward is propagated backwards along the global trace. More precisely, upon receiving a reward we calculate $\delta = r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)$ and update Q-values for all (s, a) in the relevant trace:

$$Q(s, a) \leftarrow Q(s, a) + \alpha\delta$$

Feature (binary)	Quantity
Is category _ free	13
Value of each dice	30
# of Dice with Value _	36
# of Distinct Values in Dice Roll	5
Sum of Dice (groups of 5)	6
# of Remaining Rerolls	3
# of Remaining Rounds (groups of 5)	3
Total	96

TABLE II: Atomic Single-Player Features

Feature	Quantity
Is opponent category _ free (binary)	$13n$
Player score (continuous)	1
Opponent score (continuous)	n
Total	$14n + 1$

TABLE III: Atomic n-Player Features

For our setting a state cannot be visited twice within a single game (since the state must encode information about which categories have been used), so there is no need to keep a $N(s, a)$ visit count matrix.

B. Features

Our strategy for defining features $\beta_i(s)$ for our global Q-function approximation was to first formulate a small set of basic or "atomic" features and then exhaustively add pairwise (or triple-wise, etc.) combinations of these atomic features to capture more advanced strategies. This approach was used to great success in a TD-learning agent for the card game Hearts and was able to beat advanced search-based programs [13]. Tables II and III list the atomic single-player and multi-player features we use. Most are simple binary variables that capture essential information about the state (the values of the dice, the status of categories on the score sheet, the number of rerolls and rounds left, etc.). It is fairly straightforward to see that combinations of these variables can represent reasonably sophisticated strategies: to take a simple example, an AND combination of the binary variable [# Distinct Values in Dice Roll = 1] and [Is category YAHTZEE free] would be able to capture the fact that 5 of a kind rolls are very valuable when it is possible to score them in the YAHTZEE category.

C. Benchmarks

To measure the success of our perceptron $Q(\lambda)$ agent, we measure it against several benchmarks. The first is a random agent, which chooses among the available actions uniformly at random each turn. The other agents are greedy agents. The greedy level-1 agent never rerolls the initial dice and simply chooses the category that maximizes its score this round; this is obviously sub-optimal as it may be worth saving a category for later if your roll is not ideal. The greedy level-2 and level-3 agents reroll the dice at most one and two times (respectively), and choose the actions that maximize the expected score this round given this constraint. The greedy agents are comparable to forward search with limited look-ahead. As previously discussed, since much of the state space is reachable at each state it is infeasible to look ahead beyond the current turn; the level-3 agent is already prohibitively slow to train against.

D. Hierarchical Learning

Hierarchical approaches have been used to decompose complex problems into a hierarchy of sub-problems. [16] As the dimensionality of state space increases, it becomes intractable to solve the problem with not only direct methods such as value iteration but also online learning methods. Fortunately, real-life problems typically have innate structure in them. For example, a task of opening a door can be broken down to locating the handle, moving the hand to the handle, grabbing it, turning it, pulling the hand back, and finally releasing it. Some actions are inherently different, so it might make sense to use different value functions to represent each task, such as learning how to locate the handle and learning to grab it. On the other hand, moving the hand to the handle as well as pulling the hand back, while different actions, revolve around a similar concept of moving the hand; in this case, they might be considered as actions of similar nature, and could be learned together.

One can see how Yahtzee could be formulated as a hierarchical problem with interesting characteristics. A game consists of 13 rounds, and each round can take up to 3 turns. Each turn consists of either re-rolling the dice or assigning a category, and each of these has well-defined options, such as re-rolling particular dice or assigning scores to a particular category. This hierarchy is depicted in Figure 1. Some of the hierarchy here is trivial in the sense the subdivision does not really give us an advantage. For instance, while we are playing games, we are bound in our order of rounds and turns; we cannot play turn 2 before turn 1. The structure which is actually useful is noted with the green box in the figure. For each turn, we can decide between *Re-roll* and *Category*. Then we can choose which particular action to actually take, i.e. choosing the particular dice to re-roll or the particular category to assign a roll to. The aim is that this formulation of the problem will allow the agent to balance these two "types" of actions.

Out of many hierarchical approaches, we explore the method of MAXQ-Q learning [2], [3]. In MAXQ-Q learning we decompose the tasks into a hierarchy, and then decompose value function into two parts:

$$Q(p, s, a) = V(s, a) + C(p, s, a)$$

Here, $V(s, a)$ is the value function for subtask a, the expected total reward while carrying out subtask a from state s. The second part of above equation, $C(p, s, a)$, is the completion function. It denotes the expected total reward of completing s's parent task p after a has completed. In other words, this value estimates the reward of finishing p if one follows the optimal policy after performing a. As one can see, Q now represents the expected utility one can gain by performing action a from state s while performing parent task

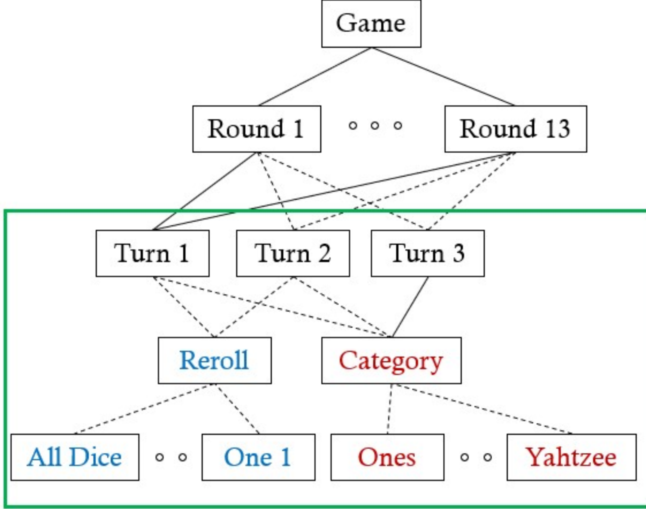

Fig. 1: Hierarchy of the Game of Yahtzee

of p. We can recursively define $V(s,x)$ for subtask x by setting it as maximum Q value of its descendents.

$$V(s,x) = \max_a [V(s,a) + C(x,s,a)]$$

The base case for this recursive definition is the primitive actions, for which $V(s,a)$ is defined as expected one-step reward.

$$V(s,a) = \sum_a T(s'|s,a)R(s,a,s')$$

This value can be initialized as 0 and the estimate can be updated as we observe each action and its corresponding reward. In our case, we only explore two levels of hierarchy, so the value function is rather simple. First, we define our Q values for our root task, which is decision between re-roll and assigning a category. Let $\alpha \in [\textit{Re-roll}, \textit{Category}]$

$$Q(s,\alpha) = V(s,\alpha) + C(root,s,\alpha)$$

For each parent task, we have defined actions we can take. One thing to note is that we do not define C for primitive actions, as the parent action terminates at the same time as the child action, and hence there is no expected completion function. Let $\beta \in [\text{All re-rolls}], \gamma \in [\text{All categories}]$

$$V(s, \textit{Re-roll}) = \max_\beta V(s,\beta)$$
$$V(s, \textit{Category}) = \max_\gamma V(s,\gamma)$$

Choosing the best action comes down to following two steps. First, we choose α_i that maximizes $Q(s_i,\alpha_i)$ for a given state s_i. Then, depending on α_i, we choose $a_i \in \beta,\gamma$ that maximizes $V(s,a_i)$.

$$\alpha_i = \max_{\alpha \in [\textit{Re-roll}, \textit{Category}]} Q(s_i,\alpha)$$
$$a_i = \max_{x \in \alpha_i} V(s,x)$$

Agent	Avg. Score	Avg. Time per Turn
Random	45.635	.026 ms
Greedy Level-1	112.541	.057 ms
Greedy Level-2	171.166	8.54 ms
Greedy Level-3	203.882	3169.38 ms
Perceptron $Q(\lambda)$	77.772	0.57 ms
HRL	120.299	1.28 ms
HRL-$G1$	129.580	0.38 ms

TABLE IV: Average Score of Different Agents

	Rand.	G-1	G-2
Random	50%	2%	0%
Greedy Level-1 (G-1)	98%	50%	7%
Greedy Level-2 (G-2)	100%	93%	50%
Perceptron $Q(\lambda)$	87%	17%	1%
HRL	91%	56%	11%
HRL-$G1$	99%	68%	17%

TABLE V: Relative Winrates

When we train the agent, we allow it to perform random exploration as much as possible to allow it to visit as many states and actions. Of course, as discussed in previous sections, it is infeasible to experience all possible combinations. Our update process is as follows: after we finish a round, we trace back the actions we took and update our V and C. Assume that in a round we performed actions a_1, a_2, a_3 from state s_1, s_2, s_3, received reward r in the end and resulted in state s_4. Let t denote the number of steps from last action, which is $2-i$ in this case for a_i. l and d are some learning rate and decay rate, respectively.

$$st = \begin{cases} \textit{Re-roll} & a_i \text{ is re-roll} \\ \textit{Category} & a_i \text{ is category} \end{cases}$$
$$V(s_i,a_i) = (1-l) * V(s_i,a_i) + l * r * d^t$$
$$V(s_i,st) = \max_{a \in st} V(s_i,a)$$
$$C(root,s_i,st) = (1-l) * C(root,s_i,st) + l * \max_{a \in root} [V(s_{i+1},a) + C(root,s_{i+1},a)]$$

Here we are again using the idea of eligibility traces to ensure that re-roll actions, which do not directly produce any reward, are awarded discounted rewards. Note that we update this in a reversed order, from a_3 to a_1, for two reasons. First, it allows us to discount the values accordingly. Secondly, and more importantly, it allows us for better updates for the re-roll actions, as V and C values for next state are already computed.

VI. Experiments

A. Simulator

We developed from scratch a Yahtzee simulator for this project. It is written in Python and allows us to specify the number of players, query the possible actions a player may take, order a player to make an action, and get the current scoresheets. The simulator allows us to organize “tournaments” between different agents to analyze their performance.

To measure performance we played the different agents against each other and recorded the win rates. Table V gives the win rates of agents against the three main benchmarks—the random agent as well as the greedy level-1 and level-2 agents. Each win rate is obtained from a 1,000 game tournament between the two agents; naturally the tournaments are after the learning phase for the RL agents.

Table IV also gives the average score and average time per turn for each of the agents. These results are also averaged over 1,000 games. Among the benchmark agents, as expected the greedy level-3 agent is far superior to the greedy level-2 agent, which in turn is far superior to the greedy level-1 agent, but each level of look-ahead adds two orders of magnitude to the running time due to the exponential growth of possible future paths. The greedy level-3 agent takes more than 3 seconds per turn and is prohibitively slow to train or play against so we exclude it from the tournaments in Table V.

B. Perceptron $Q(\lambda)$

For all reported results involving Perceptron $Q(\lambda)$, the training phase consisted of 10,000 games versus a random agent. Rewards were given out for each round score and at the end for the overall game score; we ultimately disabled rewards/penalties for winning or losing a game as we found this resulted in reduced quality and speed of learning. Only atomic, single-player features were used. We experimented extensively with multi-player features, different reward schemes, and pair-wise combinations of atomic features but found that each resulted in the identical (or slightly worse) performance. To make matters worse, some of these modifications—in particular pair-wise feature combinations—dramatically increased the time required for each turn. The next section gives possible reasons for these findings.

Even with the constrained set-up described above, however, Perceptron $Q(\lambda)$ was able to achieve some limited success. After the 10,000 game learning phase the Perceptron $Q(\lambda)$ agent was able to achieve a 87% win rate against the random agent and a 68% higher average score relative to the random agent. However, as Tables IV and V show, the Perceptron $Q(\lambda)$ agent was not very competitive with any of the greedy agents. Only a 17% win rate was achieved against the basic greedy level-1 agent, for instance.

Figures 2 and 3 (see Appendix) show the quality of the learned θ over time, by plotting the average score and average win rate versus a random agent as a function of the number of training games. To generate these plots, the 10,000-game learning phase was broken up into one hundred 100-game segments. After each segment, learning was disabled and a 100-game tournament was played between the random agent and the RL agent to measure performance. As the number of games in each post-segment tournament is small there is quite a bit of noise but there is a clear upward trajectory in performance; notable also is that the rate of learning is very fast, likely thanks to our use of local and global eligibility traces.

C. Hierarchical Learning

We test two exploration strategies for hierarchical learning. For HRL, we take random actions all the time, whether it is a re-roll or category assignment. On the other hand, we provide a one-level greedy guide for the HRL-G agent. The agent randomly re-rolls the dice twice, and on the last turn makes a category assignment that maximizes score with ending dice. This allows the agent to see more examples that help it learn, for a round with 0 reward is not informative and does not lead to parameter updates. As we enforce the agent to randomly re-roll, it will be able to explore different re-rolls from different states in shorter rounds compared to naïve random exploration.

For both cases, we simplify our state space to that of 15 dimensions: first 13 are binary variables that indicate whether a category can be assigned or not. 14th variable is a defined dice configuration value among 252 possibilities, and 15th variable indicates how many re-rolls we have left. This gives us around 6 million possible states, which is still huge but not as large as when we take into account category scores and the opponent's scoresheet. The action space depends on the subtask. For the root task, we have 2: *Re-roll* and *Category*. Here, we define a re-roll action to be in terms of the unordered set of dice values rather than the set of dice indices we reroll. That is, action of re-rolling first and third dice of 3,1,3,2,4 is regarded as 're-rolling two 3s'. There are 461 such possible actions for re-roll and 13 for categories.

We again played $1,000$ games with each benchmark agent, and the results are given in table IV and V. HRL was trained for 2×10^6 rounds for a total of 5 hours with initial $\alpha = 1 \times 10^6$ and discounted over time. It achieves average score higher than naïve greedy agent, beating it 56% of the time and beating the uniform random agent 91% of the time. HRL-$G1$ was trained for 1×10^6 rounds for a total of 5 hours with initial $\alpha = 5 \times 10^5$ and discounted over time. It achieves 99% win rate against random agent and 68% against greedy level-1 agent. We can see that both agents have learned how to greedily assign scores given dice rolls. However, both HRL and HRL-$G1$ performed poorly against level-2 greedy agent, each achieving only 11% and 16% win rate, respectively.

The initial 10,000 rounds of training results are shown Figures 2 and 3 (see Appendix). We note HRL's deceptively poor performance is due to its training nature; as we randomly make actions there are much more rounds with 0 reward, which means it plays more rounds in a given time yet does not learn fast per round. On the other hand, we see how HRL-$G1$, with its greedy guide, effectively learns information from each round, outperforming other agents.

VII. Discussion

A. Perceptron $Q(\lambda)$

We were at first puzzled why taking into account the multi-player aspects (whether by giving rewards for winning a game or adding the n-player atomic features in Table III) of a multi-player Yahtzee game *decreased* the winrate of the Perceptron $Q(\lambda)$ agent against the benchmarks, as reported in the previous

section. But some investigation suggested that this is primarily due to the heavy role of pure luck in the game of Yahtzee. The policies corresponding to the learned θ_a are poor for much of the learning phase, and wins are attributed mostly to the opponent's bad luck rather than any good decision on the agent's part; thus a win is a highly unreliable signal of good action choices and we found that being rewarded for early wins hurt both learning speed and quality. Future work may consider somehow initializing θ_a values based on a greedy level-1 policy to allow for a better starting point after which being rewarded for wins makes sense. As for why pair-wise feature combinations did not add any improvement either, the only plausible explanation is that basic strategies are sufficiently captured by the atomic features and that advanced strategies require more than just pairwise combinations, which we were unable to explore due to hardware limitations. The poor performance versus greedy agents suggests there is in general room for improvement in feature selection.

B. Hierarchical Learning

A disadvantage of hierarchical learning is that it is memory intensive. We are not only trying to assign values for each state action pair, but we are keeping 3 dictionaries (C, V for parent action and V for primitive actions). For example, the total amount of space of dictionaries for HRL agent was around 2.5GB. A relevant note is that one strength of hierarchical learning is achieved by performing state abstraction, which could take form of removing irrelevant variables, performing structural constraints, or funnel abstraction, where we generalize a large number of sites by small number of resulting states. [2] The only advantage we explored here is structural constraints, limiting the action space depending on the state. As we wanted to utilize the information about the assigned categories, dice, and number of re-rolls remaining to decide our actions, we could not reduce our state space by hierarchy.

VIII. Conclusion

In this paper, we presented a reinforcement learning approach to the famous dice game Yahtzee. Our hierarchical learning agent in particular was able to achieve reasonable success against our benchmark agents, with 99% win rate against the random agent and 68% win rate against the simplest greedy agent. Both our hierarchical learning approach and the less successful global approximation method are ultimately suboptimal, however, and there remains much room for further experimentation and development.

A lot of recent progress in reinforcement learning in a variety of domains has been made using deep reinforcement learning [8]–[10]. Equipped with enough computational resources, it would be interesting to see if deep neural networks can capture the probabilisitc as well as structural properties of Yahtzee. Other Hierarchical learning methods described before, such as *Options* or *HAM*, would be a similar yet different approach to this problem. Another possible extension is dual-learning. During the training phrase we use an agent of interest and play with a random agent; while this may allow for faster computation, we could instead make use of both players' turns by replacing the random agent with our agent of interest. With parameter sharing and adequate exploration, more efficient learning might be possible.

IX. Contributions

Minhyung's work focused on the hierarchical reinforcement learning agent and the initial perceptron $Q(\lambda)$ agent, as well as surveying existing literature for best practices and insights. Luca developed the initial simulator, experimental framework, and benchmark agents and iterated further on the perceptron $Q(\lambda)$ agent. We have contributed equally to the creation of this report and previous written submissions.

References

[1] Barto, A. G. and Mahadevan, S. 2003. Recent Advances in Hierarchical Reinforcement Learning. *Discrete Event Dynamic Systems: Theory and Applications, 13, 41-77.*
[2] Dietterich, T. 2000. An Overview of MAXQ Hierarchical Reinforcement Learning *Abstraction, Reformulation, and Approximation, 26-44.*
[3] Dietterich, T. Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition. *arXiv:cs/9905014 [cs.LG].*
[4] Firoiu, V., Whitney, W., Tenenbaum, J. Beating the World's Best at Super Smash Bros. with Deep Reinforcement Learning. *arXiv:1702.06230 [cs.LG].*
[5] Glenn, J. 2006. An optimal strategy for Yahtzee. *Loyola College in Maryland, Tech. Rep. CS-TR-0002.*
[6] Gosavi, A. 2014. Simulation-Based Optimization: Parametric Optimization Techniques and Reinforcement *Learning, Springer, New York, NY, Second edition.*
[7] Hasbro. https://www.hasbro.com/common/instruct/Yahtzee.pdf. Retrieved December 6, 2018.
[8] Hessel, M. et al. 2018 Rainbow: Combining Improvements in Deep Reinforcement Learning. *AAAI Conference on Artificial Intelligence, 32.*
[9] Mnih, V. et al. 2016 Asynchronous Methods for Deep Reinforcement Learning. *International Conference on Machine Learning 48.*
[10] Mnih, V. et al. 2013. Playing Atari with Deep Reinforcement Learning. *arXiv:1312.5602 [cs.LG].*
[11] Pang, Z., Liu, R., Meng, Z., Zhang, Y., Yu, Y., Lu, T. On Reinforcement Learning for Full-length Game of Starcraft. *arXiv:1809.09095 [cs.LG].*
[12] Parr, R. and Russell, S. 1997. Reinforcement Learning with Hierarchies of Machines. *Neural Information Processing Systems 10.*
[13] Sturtevant, N. and White, A. M. 2006. Feature Construction for Reinforcement Learning in Hearts. *International Conference on Computers and Games, 122-134.*
[14] Sutton, R.S. and Singh, S. 1999. Between MDPs and Semi-MDPs: A Framework for Temporal Abstraction in Reinforcement Learning. *Artificial Intelligence 112, 181-211.*
[15] Vancura, O. 2001. *Advantage Yahtzee*. Las Vegas: Huntington Press.
[16] Wiering, M. and Otterlom, M. v. 2012. Reinforcement Learning (State of the Art). Springer.
[17] The Yahtzee! Page. http://www.yahtzee.org.uk. Retrieved October 4, 2018.

Appendix

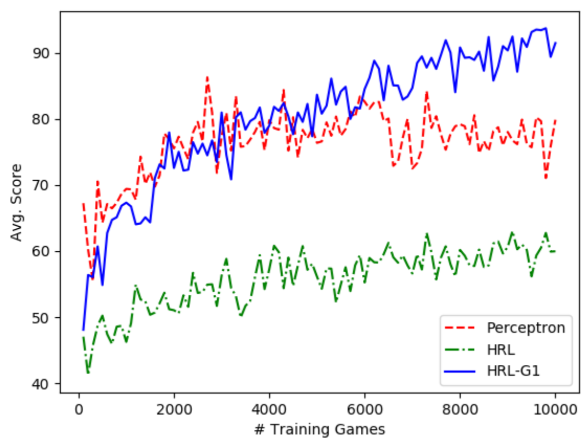

Fig. 2: Avg. Score of Agents Over Time

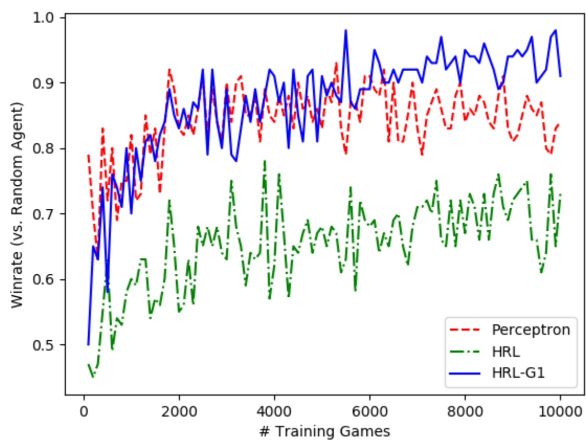

Fig. 3: Avg. Winrate of Agents Over Time