Yahtzee: Reinforcement Learning Techniques for Stochastic Combinatorial Games

Nick Pape
nap626
nickpape@utexas.edu

Abstract

Write your abstract here. This should be a concise summary of your work, including the problem you're addressing, your approach, and key results. Keep it to about 150-250 words.

1 Introduction

¡600-900 words¿

1.1 Yahtzee as a Reinforcement Learning Benchmark

While on the surface *Yahtzee* appears to be a trivial dice game (Hasbro, Inc., 2022), it is actually a complex stochastic optimization problem with combinatorial complexity.

Although there are methods for computing optimal play in *Yahtzee* using dynamic programming, these are computationally expensive and do not scale well to multiplayer settings. *Yahtzee* offers a rich environment for testing reinforcement learning (RL) solutions due to its combination of a large but manageable state space, randomness, ease of simulation, subtle strategic considerations, and easily identifiable subproblems. While there have been a small number of efforts to create RL agents for *Yahtzee*, a comprehensive approach using self-play has yet to be published. It remains an open question of whether deep RL methods can approach optimal performance in full-game *Yahtzee*, and which architectural and training choices most affect learning efficiency and final performance. Similarly, a robust solution for multiplayer *Yahtzee* using RL methods has yet to be demonstrated.

Yahtzee is an ideal candidate to serve as a bridge between simple toy problems such as *Lunar Lander* (Brockman et al., 2016) and extremely complex games like Go (Silver et al., 2016). Typical small benchmarks often offer low stochasticity and simple combinatorics whereas complex games have intractable state spaces and require massive computational resources and heavy engineering to solve. *Yahtzee* sits in a middle ground where an analytic optimum exists, but reaching it with RL methods is non-trivial. Its blend of stochasticity and large state space makes it a challenging yet feasible benchmark for RL research.

1.2 Objectives

In this paper we aim to methodically study whether a deep RL agent can achieve near DP-optimal performance in full-game solitaire *Yahtzee* using only self-play without any shaping rewards or expert demonstrations, and how architectural and training choices affect learning efficiency.

Concretely, we ask: (i) How does the trade-off between maximizing single-turn expected score and full-game performance behave? (ii) Can a shaping-free self-play agent can reach optimal performance under a fixed training budget? (iii) Which design choices (state and action encodings, variance controls, baselines, entropy, etc) most affect final performance? (iv) What failure modes exist in learned policies and how can they be addressed? (v) Can we find a successful architecture which could be adapted to multiplayer *Yahtzee*?

2 Related Work

¡600-800 words¿

2.1 Policy Gradient Methods and Variance Reduction

Policy-gradient methods are a family of algorithms which directly optimize a parameterized policy π_θ to follow an estimate of the performance gradient (Sutton et al., 2000). A simple formulation of this is the REINFORCE algorithm (Williams, 1992), which uses Monte-Carlo returns

G_t^{MC} on finite, episodic tasks; however, while unbiased, it suffers from high variance. Actor-critic methods (Konda and Tsitsiklis, 1999; Mnih et al., 2016) address this by learning a seperate value function, the critic V_ϕ, which serves a variance reducing baseline. A recent and popular algorithm, proximal policy optimization (PPO) (Schulman et al., 2017), utilizes a clipped objective $L^{CLIP}(\theta)$ and explicit Kullback-Leibler (KL) divergence control to dramatically reduce variance and ensure stable updates.

In long episodic games, the choice of return calculation affects sample efficiency, bias, and variance. REINFORCE Monte-Carlo (MC) returns G_t^{MC} use a summation over the full series of rewards until the end of the episode. This approach is unbiased but has high variance. In contrast, Temporal Difference methods use a "bootstraped estimate" of future rewards to reduce variance. Essentially, they only consider received rewards R in a specific time window, and use an estimate from the value function $V(S_{t+1})$ for future rewards beyond that window. The TD erorr is the difference between this sum and the value estimate for the current state $V(S_t)$; and can be used as the advantage estimate for policy gradient methods (Sutton, 1988). These estimates are biased (since they rely on future value estimates to be accurate), but they have much lower variance than full-episode returns. This often makes TD methods more sample efficient than REINFORCE, and provides the benefit of being able to learn online rather than waiting until the end of an episode.

This time window can also be adjusted, depending on the task. For example, n-step returns $G_t^{TD(n)}$ (Sutton and Barto, 2018) interpolate between MC and single-step TD returns, allowing us to define a time horizon n over which to sum rewards before bootstrapping, this lets us manually control the bias-variance tradeoff. An improvement on n-step is this is the $TD(\lambda)$ algorithm (Sutton and Barto, 2018), which uses an exponentially weighted average of n-step returns, effectively blending multiple time horizons intoa single estimate controlled by λ.

Actor-Critic (Konda and Tsitsiklis, 1999) and PPO (Schulman et al., 2017) can use any of these TD-style return estimation methods, including the generalized advantage estimate (GAE) (Schulman et al., 2016), which is a special formulation of $TD(\lambda)$.

Aside from return estimation, there is a host of other variance reduction techniques which can be employed for policy gradient methods. Subtracting a learned baseline from REINFORCE yields an advantage estimate that reduces variance without changing its expectation (Weaver and Tao, 2013; Greensmith et al., 2004). Normalizing advantages across a batch improves gradient conditioning and is common practice. Entropy regularization prevents early collapse to suboptimal policies by encouraging exploration via the addition of an explicit entropy bonus term in the loss function (Williams and Peng, 1991). Gradient clipping is frequently used alongside these techniques to stop rare, but large, gradient updates from destabilizing training (Pascanu et al., 2013). While high variance is unavoidable in deep reinforcement learning, poor performance can often be linked to numerical instability rather than inherent flaws in algorithmic design (Bjorck et al., 2022); simple tweaks like normalizing features before activations can dramatically improve stability.

2.2 Complex Games

Typical board and dice games have extreme state complexity or stochasticity; reinforcement learning methods are a natural fit for these problems. In a classic example, Tesauro (1995) utilized temporal difference learning to achieve superhuman performance in *Backgammon*, another game with a large state space and stochastic elements. Tetris, which is deterministic but combinatorial, has also been studied extensively; Bertsekas and Ioffe (1996) utilized approximate dynamic programming methods to learn effective policies for the game, while Gabillon et al. (2013) effectively tackled the game using reinforcement learning methods. Moravčík et al. (2017) demonstrated that *Texas Hold'em*, a stochastic game with hidden information, could be effectively learned. Many other stochastic games can be learned well, so long as methods which ensure better exploration are used (Osband et al., 2016). Lastly, RL methods can be used to reach superhuman performance on adversarial games, even despite their sparse reward structures. For example, the game of Go, which has a notoriously intractable state space was solved using Monte-Carlo Tree Search and deep value networks (Silver et al., 2016). Subsequent work showed Go could be learned without the use of expert data, purely through self-play

(Silver et al., 2017). In total, these works establish that RL methods can handle highly stochastic, combinatorial games, suggesting that *Yahtzee* is a natural but underexplored candidate in this family.

2.3 DP Methods for Yahtzee

Solitaire *Yahtzee* is a complex game with an upper bound of 7×10^{15} possible states in its state space. It has a high degree of stochasticity, as dice rolls are the primary driver of state transitions. Despite this, it has been analytically solved using dynamic programming techniques; Verhoeff (1999), calculated that the average score achieved during ideal play is 254.59 points, which serves as the gold-standard baselline for solitaire *Yahtzee*. Later work by Glenn (2006) optimized the DP approach via symmetries to propose a more efficient algorithm for computing the optimal policy, with a reachable state space of 5.3×10^8 states (Glenn, 2007).

However, adversarial *Yahtzee* remains an open problem. While Pawlewicz (2011) showed that DP techniques can be expanded to 2-player adversarial *Yahtzee*, they do not scale to more players due to the exponential growth of the state space. Approximation methods must be utilized for larger player counts.

2.4 Reinforcement Learning for Yahtzee

Some prior attempts have been made to apply reinforcement learning to *Yahtzee*. YAMS attempted to use Q-learning and SARSA to attempt to learn *Yahtzee*, but was not able to surpass 120 points median (Belaich, 2024). Likewise, Kang and Schroeder (2018) applied hierarchical MAX-Q, achieving an average score of 129.58 and a 67% win-rate over a 1-turn expectimax agent baseline. Vasseur (2019) explored strategy ladders for multiplayer *Yahtzee*, to understand how sensitive Deep-Q networks were to the upper-bonus threshold. Later, (Yuan, 2023) applied Deep-Q networks to the adversarial setting.

Additionally, some recent informal work has reported success using RL methods for *Yahtzee*. For example, Yahtzotron used heavy supervised pretraining and A2C to achieve an average of 236 points (Häfner, 2021). Although it is unclear if the results are repeatable, Dutschke reports a deep-Q agent achieving a score of 241.6 ± 40.7 after just 8,000 games.

No prior work systematically explores policy gradient methods with variance reduction tricks on full-game solitaire *Yahtzee*, attempts transfer learning from single-turn optimization, provides a detailed ablation and failure mode analysis, or offers an architecture that is theoretically transferrable to multiplayer settings.

3 Problem Formulation

¡400-600 words¿

3.1 Game Description

3.1.1 Rules of Yahtzee

Yahtzee is played with five standard six-sided dice and a shared scorecard containing 13 categories. Turns are rotated among players. A turn starts with a player rolling all five dice. They may then choose to keep some dice, and re-roll the remaining ones. This process can be repeated up to two more times, for a total of three rolls. After the final roll, the player must select one of the 13 scoring categories to apply to their current dice. Each category has specific scoring rules, and each can only be used once per game.

3.1.2 Mathematical Representation of Yahtzee

The space of all possible dice configurations is:

$$\mathcal{D} \in \{1,2,3,4,5,6\}^5 \tag{1}$$

and the current state of the dice is represented as:

$$\mathbf{d} \in \mathcal{D} \tag{2}$$

In addition, we can represent the score card as a vector of length 13, where each element corresponds to a scoring category:

$$\mathbf{c} = (c_1, c_2, \ldots, c_{13}) \text{ where } c_i \in \mathcal{D}_i \cup \{\varnothing\} \tag{3}$$

and $\varnothing$ indicates an unused category.

Let us also define a dice face counting function which we can use to simplify score calculations:

$$n_v(\mathbf{d}) = \sum_{i=1}^{5} \mathbb{I}(d_i = v), \quad v \in \{1,\ldots,6\} \tag{4}$$

$$\mathbf{n}(\mathbf{d}) = \big(n_1(\mathbf{d}), \ldots, n_6(\mathbf{d})\big) \tag{5}$$

Let the potential score for each category be defined as follows (where detailed scoring rules can be found in Appendix B):

$$\mathbf{f}(\mathbf{d}) = \big(f_1(\mathbf{d}), f_2(\mathbf{d}), \ldots, f_{13}(\mathbf{d})\big) \tag{6}$$

The current turn number can be represented as:

$$t \in \{1, 2, \ldots, 13\}, \quad t = \sum_{i=1}^{13} \mathbb{I}(c_i \neq \varnothing)$$

A single turn is composed of an initial dice roll, two optional re-rolls, and a final scoring decision. Let $r = 0$, with $r \in \{0, 1, 2\}$ which is the number of rolls taken so far. Prior to the first roll, the dice are randomized:

$$\mathbf{d}_{r=0} \sim U(\mathcal{D})$$

The player must decide which dice to keep and which to re-roll. Let the player define a keep vector:

$$\mathbf{k} \in \{0, 1\}^5$$

where $\mathbf{k}_i = 1$ indicates that die i is kept, otherwise it is re-rolled.

We can then define the transition of the dice state after a re-roll as:

$$\begin{aligned} \mathbf{d}' &\sim U(\mathcal{D}), \\ \mathbf{d}_{r+1} &= (\mathbf{1} - \mathbf{k}) \odot \mathbf{d}' + \mathbf{k} \odot \mathbf{d} \end{aligned}$$

When $r = 2$, the player must choose a scoring category to apply their current dice to. Define a scoring choice mask as a one-hot vector:

$$\mathbf{s} \in \{0, 1\}^{13}, \quad \|\mathbf{s}\|_1 = 1$$

For the purposes of calculating the final (or current) score, any field that has not been scored yet can be counted as zero. We can define a mask vector for this:

$$\begin{aligned} &\mathbf{u}(\mathbf{c}) \in \{0, 1\}^{13} \\ &\mathbf{u}(\mathbf{c})_i = \mathbb{I}\big(c_i \neq \varnothing\big), \quad \forall i = \{1, \ldots 13\} \end{aligned}$$

If a player achieves a total score of 63 or more in the upper section (categories 1-6), they receive a bonus of 35 points:

$$B(\mathbf{c}) = \begin{cases} 35, & \sum_{i=1}^{6} \mathbf{u}(\mathbf{c})_i \cdot \mathbf{c}_i \geq 63 \\ 0, & \text{otherwise} \end{cases}$$

The player's score can thus be calculated as:

$$\text{score}(\mathbf{c}) = B(\mathbf{c}) + \big\langle \mathbf{u}(\mathbf{c}), \mathbf{c} \big\rangle$$

3.2 MDP Formulation

We model *Yahtzee* as a Markov Decision Process $(\mathcal{S}, \mathcal{A}, P, R, \gamma)$ (Puterman, 1994).

A state is represented as $\mathbf{s} = (\mathbf{d}, \mathbf{c}, r, t)$, where $\mathbf{d}$ is the current dice configuration, $\mathbf{c}$ the scorecard, and r the roll index, and t the current turn index (see Section 3.1.2).

For simplicity, we define the action $\mathbf{a} = (\mathbf{k}, \mathbf{s})$, where $\mathbf{k}$ is the keep vector and s is the score category choice. We can define the action as a parameterization of the policy: $\pi_\theta(\mathbf{a}|\mathbf{s}) = \pi_\theta(\phi(\mathbf{s}))$, where $\phi(\mathbf{s})$ is a feature representation of the state $\mathbf{s}$.

The transition function P is is specified in Appendix C. Note that when $r < 2$, the $\mathbf{k}$ is used by P, otherwise $\mathbf{s}$ is used.

The reward is the change in total score between steps $R_t = \text{score}(c_{t+1}) - \text{score}(c_t)$, and since we desire to maximize total score at the end of the game, we set $\gamma = 1$.

3.3 Single-Turn Optimization Task

In the single-turn optimization task, the agent is trained to maximize the expected score over a single turn. This task has 3 steps total; after being initialized with a random dice roll, the agent chooses which dice to keep and which to re-roll twice, and then selects a scoring category. A single reward is given at the end of the turn.

This is a useful subproblem to study, as it isolates the decision-making process in a single turn, allowing us to analyze the network architecture and training regime in a low-variance setting.

3.4 Full-Game Optimization Task

In the full-game optimization task, 13-turn episodes are played to completion. The objective again is to maximize the total score at the end of the game. This task is more challenging due to the longer horizon, increased variance, and the requirement for the network to learn explicit long-term strategies, such as planning for the upper bonus.

4 Methodology

¡1400-1700 words¿

4.1 State Representation & Input Features

The design of $\phi(\mathbf{s}) \to \mathbf{x}$ is one of the most critical components to the performance of a model (Sutton and Barto, 2018).

Formally, we define the state representation function as

$$\mathbf{x} = \phi(\mathbf{s}) \tag{7}$$

where $\mathbf{s}$ is the raw MDP state (e.g., dice configuration, scorecard, roll index, turn index), and $\mathbf{x}$ is the feature vector or tensor provided as input to the model. The choice of ϕ determines how information from the environment is encoded for learning and inference. As such, several different representations were tested to evaluate their impact on learning efficiency and final performance.

4.1.1 Dice Representation

The dice representation can be encoded in several ways, depending on if we want to preserve permutation invariance or not. Preserving ordering information (and implicitly, ranking) gives the model the benefit of being able to directly output actions corresponding to dice indices, however, it comes at the cost of implicitly biasing the model to specific dice orderings; in other words, towards a local optima of keeping the highest ranking dice. However, eliminating ordering information requires the model to either waste capacity learning permutation invariance or be inherently supportive of invariance (e.g. with self-attention). It also requires a different action representation, since actions can no longer correspond to specific dice indices. We attempted 5 different dice representations:

- **One-hot encoding of ordered dice:** Each die is represented as a one-hot vector of length 6, and the 5 dice are concatenated in order.
- **Bin representation:** Here we pass through the dice face count vector $\mathbf{n}(\mathbf{d})$.
- **Combined representation:** Both ordered one-hot and bin representations are concatenated.
- **Learnable encoding of dice:** Each die is represented as a learnable embedding vector of length d, and self-attention is applied.
- **Learnable encoding of dice with positional encodings:** Similar to above, but with sinusoidal positional encodings added to each die embedding.

$$\phi_{\text{dice}}(\mathbf{d}) = \text{(varies by representation)}$$

4.1.2 Scorecard Representation

There are two important pieces of information ϕ must encode about the scorecard: whether a category is open or closed, and some form of progress towards the upper bonus.

$$\phi_{\text{cat}}(\mathbf{c}) = \mathbf{u}(\mathbf{c})$$

We experimented with several ways of encoding the bonus progress, but settled on a simple normalized, clamped sum of the upper section scores:

$$\phi_{\text{bonus}}(\mathbf{c}) = \min\left(\frac{1}{63}\sum_{i=1}^{6} c_i, 1\right)$$

4.1.3 Computed Features

There are some key features that can be computed from the raw state, providing these can allow the model to focus on higher-level patterns.

$$\begin{aligned}
\phi_{\text{progress}}(t) &= \frac{t}{12} \\
\phi_{\text{rolls}}(r) &\in \{0,1\}^3, \quad \|\phi_{\text{rolls}}(r)\|_1 = 1 \\
\phi_{\text{joker}}(\mathbf{c}) &\in \{0,1\}, \quad \text{(Joker rule active)} \\
\phi_{\text{potential}}(\mathbf{d},\mathbf{c}) &= \big(\mathbf{u}(\mathbf{c}) \odot \mathbf{f}(\mathbf{d})\big) \oslash \mathbf{m},
\end{aligned}$$

where $m_k = \max_{\mathbf{d}'} f_k(\mathbf{d}')$ is the maximum possible score for category k.

4.2 Action Representation

4.2.1 Rolling Action

We experiment with two different rolling action representations. The first is a Bernoulli representation, where each die has an individual binary decision to be re-rolled or held. The second is a categorical representation, where each of the 32 possible combination of dice to keep is represented as a unique action.

$$a_{\text{roll}} \sim \begin{cases} \text{Bernoulli}(\sigma(f_\theta(\phi(x)))) \\ \text{Categorical}(\text{softmax}(f_\theta(\phi(x)))) \end{cases}$$

4.2.2 Scoring Action

The scoring action is always a categorical distribution over the 13 scoring categories.

$$a_{\text{score}} \sim \text{Categorical}(\text{softmax}(f_\theta(\phi(x))))$$

4.2.3 Rewards

We intentionally do not use any reward shaping or extra rewards beyond the change in score after each action. While this poses a greater challenge for the model, it ensures that the learned policy is not biased by hand-crafted rewards, and allows us to better focus on other aspects of the learning process.

4.3 Neural Network Architecture

The neural network uses a unique architecture designed to handle the specific challenges of *Yahtzee*. The architecture consists of a trunk, followed by heads for the policy and value functions.

4.3.1 Trunk

The trunk of the network is a standard feedforward architecture with 2-3 fully connected hidden layers. The width of each layer is 386-512 neurons, found through empirical hyperparameter tuning, but aligning our model capacity with theoretical maximums (Horne and Hush, 1994) and minimums (Hanin, 2017). We utilize layer normalization for improved training stability (Ba et al., 2016; Bjorck et al., 2022) and Swish activations (Ramachandran et al., 2017) to introduce stable non-linearities.

4.3.2 Policy and Value Heads

We utilize two distinct heads for the rolling and scoring actions, allowing the model to specialize in each task (Tavakoli et al., 2018; Hausknecht and Stone, 2016). We also implement a value head. This outputs a scalar which is used as the baseline for REINFORCE and the value estimate for actor-critic methods. Each of these networks has a fully connected hidden layer before the final output layer.

In the rolling action head, we use either 5 outputs with sigmoid activations for Bernoulli representation, or 32 outputs with softmax activations for the categorical representation.

In the scoring action head, we use 13 outputs with softmax activations for the categorical distribution over scoring categories.

For the value head, we use a single linear output, constrained with ELU activation to clamp negative value estimates (Clevert et al., 2016), since negative rewards are not possible in *Yahtzee*.

4.3.3 Weight Initialization

¡KAIMING OR XAVIER?¿

4.3.4 Optimization & Schedules

We utilize the Adam optimizer (Kingma and Ba, 2014) with maximum learning rate between 1×10^{-4} and 5×10^{-4}. To improve training stability, we utilize a warmup schedule over the first 5% of training (Kalra and Barkeshli, 2024), plateau for 70% of training, and then linearly decay over the final 25% of training steps (Defazio et al., 2023; Lyle et al., 2024).

4.4 Entropy

We also inject an entropy bonus into the loss function to encourage exploration (Williams and Peng, 1991), these are held constant at the start of training then linearly decayed to a final value near the end of training. Different entropy bonuses were used for rolling and scoring actions, as rolling actions had a tendency to collapse early in training.

Exploration is particularly important for Yahtzee, there are many stable suboptimal policies (e.g., exclusively going for the upper bonus, always going for Yahtzees, etc). Once the model has figured out how to play the game, it quickly converges on one of these, and won't explore other strategies since they often trade off short-term rewards for long-term gains.

We can define the entropy bonus as:

$$\begin{aligned}\mathcal{L}_{\text{entropy}}(\theta) = &-\underbrace{\overbrace{\beta_{\text{roll}}}^{\text{weight}}\;\mathcal{H}\big[\pi_{\theta,\text{roll}}(\cdot \mid s_t)\big]}_{\text{rolling action entropy}} \\ &-\underbrace{\overbrace{\beta_{\text{score}}}^{\text{weight}}\;\mathcal{H}\big[\pi_{\theta,\text{score}}(\cdot \mid s_t)\big]}_{\text{scoring action entropy}}\end{aligned}$$

4.4.1 Training Metrics

To better understand training dynamics, we log several metrics during training. To monitor the quality of the value network, we monitor explained variance (Schulman, 2016; Schulman et al., 2016). To monitor for policy collapse, we track the policy entropy and KL divergence between policy updates (Schulman, 2016; Schulman et al., 2017), mask diversity (Hubara et al., 2021), and the top-k action frequency (Sun et al., 2025). To monitor for learning stability, we track gradient norms and clip rate (Pascanu et al., 2013; Engstrom et al., 2020). To ensure advantages are well-conditioned, we track advantage mean and standard deviation (Achiam, 2018). We also monitor standard training metrics such as average reward and loss values.

4.5 Reinforcement Learning Algorithms

4.5.1 REINFORCE

We first implement the REINFORCE algorithm (Williams, 1992) with baseline for single-turn optimization, then attempt to extend it to full-game optimization. The baseline is the output of the value head, $V_\phi(\mathbf{s})$. Thus, the loss function is:

$$\begin{aligned}\mathcal{L}(\theta,\phi) = & \underbrace{\overbrace{-\log\big(\pi_\theta(a_t \mid s_t)\big)}^{\text{negative log likelihood}}\ \overbrace{\big(\hat{R}_t - V_\phi(s_t)\big)}^{\text{advantage}}}_{\text{policy loss}} \\ & + \underbrace{\overbrace{\lambda_V}^{\text{weight}}\ \big\|V_\phi(s_t) - \hat{R}_t\big\|_2}_{\text{value loss}} \\ & - \underbrace{\overbrace{\beta}^{\text{weight}}\ \mathcal{H}\big[\pi_\theta(\cdot \mid s_t)\big]}_{\text{entropy bonus}}\end{aligned}$$

4.5.2 Advantage Actor-Critic (A2C)

For *Yahtzee*, we utilize an episodic, one-step TD(0) Advantage Actor-Critic (A2C) method (Sutton and Barto, 2018).

$$\delta_t = \overbrace{r_t}^{\text{reward}} + \overbrace{\gamma V_\phi(s_{t+1})}^{\text{discounted future}} - \overbrace{V_\phi(s_t)}^{\text{current estimate}}$$

$$\begin{aligned}\mathcal{L}_{\text{TD-AC}}(\theta,\phi) = & \underbrace{\overbrace{-\log\big(\pi_\theta(a_t \mid s_t)\big)}^{\text{negative log likelihood}}\ \overbrace{\delta_t}^{\text{TD-error}}}_{\text{policy loss}} \\ & + \underbrace{\overbrace{\lambda_V}^{\text{weight}}\ \big\|\delta_t\big\|_2^2}_{\text{value loss}} \\ & - \underbrace{\overbrace{\beta}^{\text{weight}}\ \mathcal{H}\big[\pi_\theta(\cdot \mid s_t)\big]}_{\text{entropy bonus}}\end{aligned}$$

In TD(0), the value function effectively learns using a single timestep; this is a much simpler problem than estimating the entire sequence of rewards. Pure TD methods can be considered a form of estimated DP, which makes them potentially well suited for *Yahtzee*, for which a DP solution is known to exist.

Other forms of TD returns, such as n-step TD and TD(λ) can be used by replacing δ_t with the appropriate return estimate.

4.5.3 PPO

PPO improves on TD by utilizing a ”surrogate” objective which clips large policy updates, improving training stability. This allows for substantially larger batch sizes and learning rates (Schulman et al., 2017).

$$r_t(\theta) = \frac{\overbrace{\pi_\theta(a_t \mid s_t)}^{\text{current policy}}}{\underbrace{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}_{\text{behavior policy}}}$$

$$\begin{aligned}\mathcal{L}(\theta,\phi) = & \underbrace{-\min\begin{Bmatrix} r_t(\theta)\,\hat{A}_t, \\ \text{clip}\,\big(r_t(\theta), 1-\epsilon, 1+\epsilon\big)\,\hat{A}_t \end{Bmatrix}}_{\text{policy loss}} \\ & + \underbrace{\overbrace{\lambda_V}^{\text{weight}}\ \big\|V_\phi(s_t) - \hat{R}_t\big\|_2^2}_{\text{value loss}} \\ & - \underbrace{\overbrace{\beta}^{\text{weight}}\ \mathcal{H}\big[\pi_\theta(\cdot \mid s_t)\big]}_{\text{entropy bonus}}\end{aligned}$$

4.5.4 Training Regimes

We analyze several distinct training regimes for *Yahtzee* agents: (i) REINFORCE directly on the single-turn optimization task and evaluating full-game performance (ii) REINFORCE, TD, and PPO directly on the full-game optimization task.

4.6 Evaluation Protocol

During training, we run 1,000 game episodes every 5 epochs (1% of training) to monitor progress.

For a model's final evaluation, we run 10,000 games, reporting the mean score and standard deviation. For a typical variance $\sigma^2 \approx 50$, this generally gives us a standard error $\sigma/\sqrt{10000} = \pm 0.5$. Likewise, we compute category statistics to analyze strategic preferences of different agents.

5 Results

5.1 Generalization of Single-Turn Agent to Full Game

¡200-400 words¿ ¡Talk about the single-turn agent's performance when directly applied to full-game Yahtzee without further training.¿ ¡Tradeoff with full-game performance, etc.¿

5.2 Single-Turn Results

¡600-700 words¿

5.2.1 Baseline Model Performance

For state representation, the baseline model utilizes a combined dice representation (ordered one-hot + bin), the category usage mask for the scorecard, and computed features for bonus progress, and roll index. For outputs, it uses Bernoulli rolling actions and categorical scoring actions. We utilized REINFORCE algorithm to train on ¡X¿ single-turn episodes (consisting of 3 steps), using a batch size of ¡Y¿ episodes, for ¡Z¿ total gradient updates.

Although it does not nearly reach optimal performance, it performs surprisingly well over the full game; this is likely due to the high correlation between single-turn and full-game optimal actions. However, we suspected target leakage (selecting parameters and architectures based on full-game performance) could also play a role and analyze the full-game vs. single-turn tradeoff in Section 5.2.4.

Table 1: Category statistics for the optimal strategy

Category	$\bar{s}(c)$	$\sigma^2(c)$
Ones	¡X¿	¡Y¿
Twos	¡X¿	¡Y¿
Threes	¡X¿	¡Y¿
Fours	¡X¿	¡Y¿
Fives	¡X¿	¡Y¿
Sixes	¡X¿	¡Y¿
Three of a Kind	¡X¿	¡Y¿
Four of a Kind	¡X¿	¡Y¿
Full House	¡X¿	¡Y¿
Small Straight	¡X¿	¡Y¿
Large Straight	¡X¿	¡Y¿
Chance	¡X¿	¡Y¿
Yahtzee	¡X¿	¡Y¿
Upper Bonus	¡X¿	¡Y¿
Yahtzee Bonus	¡X¿	¡Y¿

Some of our training metrics are shown in Figure 1.

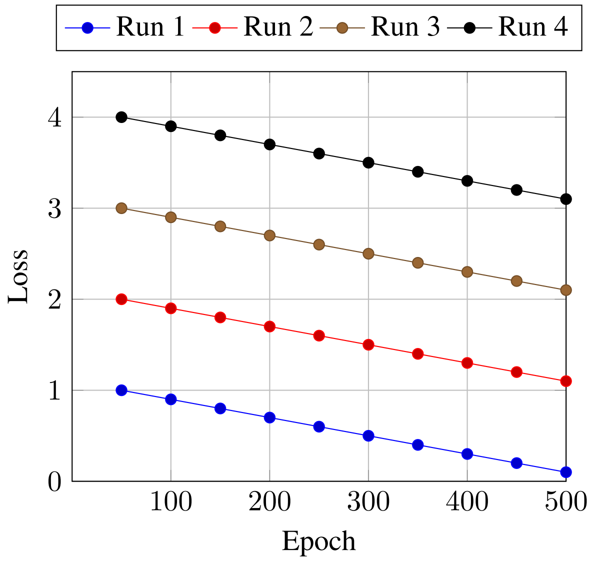

Figure 1: Training loss over epochs (placeholder data).

¡train entropy graph¿

¡train top k freq graph¿

¡train policy KL graph¿

¡table of per category scores and stddev¿

5.2.2 Representational Ablations

¡Show how adding some representations pushes performance up or down.¿ ¡Graph average score vs representation choice¿

5.2.3 Architectural Ablations

¡Show how changing architecture (layer norm, activation fn, width, depth) affects performance.¿ ¡Graph average score vs architecture choice¿

5.2.4 Single vs Full-game Tradeoff Curve

To understand the tradeoff between single-turn and full-game performance, we ablated our model using small changes to various hyperparameters and captured the resulting performance on both the primary single-turn score, as well as the auxiliary full-game score. We expect that there is a Pareto frontier between these two objectives, and that some hyperparameter choices push performance towards one or the other.

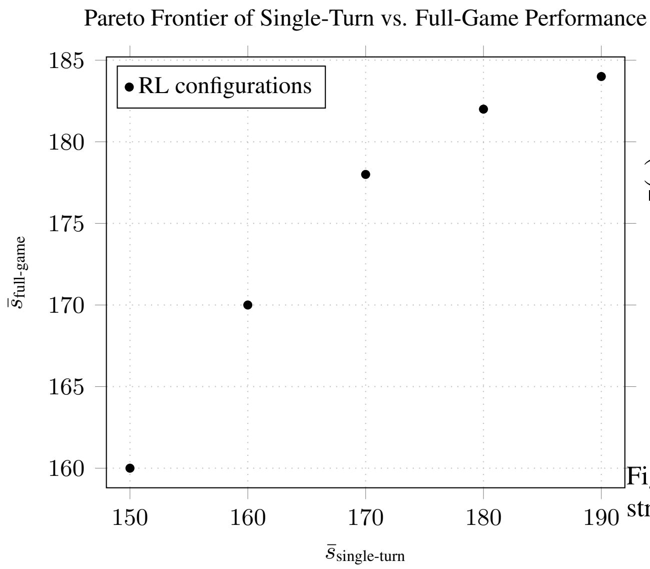

Figure 2: Empirical Pareto frontier between single-turn and full-game average scores.

5.2.5 Summary

¡Talk about the biggest levers¿

5.3 Full-Game Results

¡600-700 words¿

5.3.1 REINFORCE

¡Talk about reusing the single-turn config and how it doesn't perform well¿ ¡Talk about tuning it for full-game and how that helps¿ ¡Graph showing single turn vs full game REINFORCE performance¿

5.3.2 TD(λ)

¡Talk about how it performs better than REINFORCE¿

5.3.3 PPO

¡Talk about how it performs best¿

5.3.4 Transfer Learning

¡Show results for transfer learning directly¿ ¡Show results for transfer learning with LoRA¿

5.3.5 Summary

¡Summarize full-game results and key findings¿

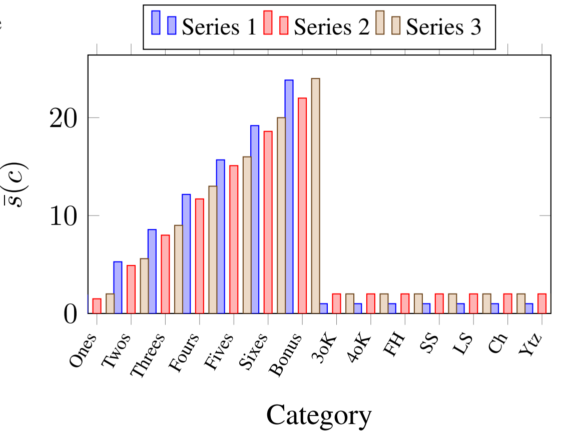

Figure 3: Average score $\bar{s}(c)$ per category for three strategies.

5.4 Policy Analysis

¡300-500 words¿

5.4.1 Category Usage Distribution

¡Bar chart showing average score per category for different agents¿

¡table comparison against DP optimal solutions¿

5.4.2 Score Breakdown

¡Notes on how different agents achieve their scores differently¿

5.4.3 Strategy Comparison

¡Compare strategies learned by different agents¿

5.4.4 Failure Modes

¡Analyze common failure modes observed in learned policies¿ ¡Analyze failures during training such as policy collapse¿

6 Discussion

¡400-600 words¿ ¡Discuss implications of results, limitations, and potential improvements.¿

7 Conclusion and Future Work

¡200-300 words¿ ¡Summarize key findings and suggest directions for future research.¿

References

Joshua Achiam. 2018. Spinning up in deep reinforcement learning. Technical report. OpenAI educational resource.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. 2016. Layer normalization. *arXiv preprint arXiv:1607.06450*.

Alae Belaich. 2024. YAMS: Reinforcement Learning Project. GitHub repository, accessed 2025-11-16.

Dimitri P. Bertsekas and Sergey Ioffe. 1996. Temporal differences-based policy iteration and applications in neuro-dynamic programming. Technical Report LIDS-P-2349, Laboratory for Information and Decision Systems, MIT.

Johan Bjorck, Carla P. Gomes, and Kilian Q. Weinberger. 2022. Is high variance unavoidable in rl? a case study in continuous control. *arXiv preprint arXiv:2110.11222*.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. Openai gym. *arXiv preprint arXiv:1606.01540*.

Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. 2016. Fast and accurate deep network learning by exponential linear units (ELUs). In *Proceedings of the International Conference on Learning Representations*. ArXiv:1511.07289.

Aaron Defazio, Ashok Cutkosky, Harsh Mehta, and Konstantin Mishchenko. 2023. Optimal linear decay learning rate schedules and further refinements. *arXiv preprint arXiv:2310.07831*.

Markus Dutschke. A yahtzee/kniffel simulation making use of machine learning techniques. GitHub repository; accessed: 2025-11-16.

Logan Engstrom, Andrew Ilyas, Shibani Santurkar, Dimitris Tsipras, Federico Janoos, Larry Rudolph, and Aleksander Madry. 2020. Implementation matters in deep policy gradients: A case study on PPO and TRPO. In *International Conference on Learning Representations (ICLR)*.

Victor Gabillon, Mohammad Ghavamzadeh, Alessandro Lazaric, and Bruno Scherrer. 2013. Approximate dynamic programming finally performs well in the game of tetris. In *Advances in Neural Information Processing Systems*, volume 26.

Jeffrey R. Glenn. 2006. An optimal strategy for yahtzee. Technical Report CS-TR-0002, Loyola College in Maryland, Department of Computer Science.

Jeffrey R. Glenn. 2007. Computer strategies for solitaire yahtzee. In *2007 IEEE Symposium on Computational Intelligence and Games (CIG)*, pages 132–139.

Evan Greensmith, Peter L. Bartlett, and Jonathan Baxter. 2004. Variance reduction techniques for gradient estimates in reinforcement learning. *Journal of Machine Learning Research*, 5:1471–1530.

Dion Häfner. 2021. Learning to play yahtzee with advantage actor-critic (a2c). Accessed: 2025-11-16.

Boris Hanin. 2017. Universal function approximation by deep neural nets with bounded width and relu activations.

Hasbro, Inc. 2022. *YAHTZEE Game: Instructions*. Official rules and instructions.

Matthew Hausknecht and Peter Stone. 2016. Deep reinforcement learning in parameterized action space. In *Proceedings of the International Conference on Learning Representations (Workshop Track)*. ArXiv:1511.04143.

Bill G. Horne and Don R. Hush. 1994. Bounds on the complexity of recurrent neural network implementations of finite state machines. In *Advances in Neural Information Processing Systems 6*, pages 359–366, San Francisco, CA. Morgan Kaufmann.

Itay Hubara, Brian Chmiel, Moshe Island, Ron Banner, Seffi Naor, and Daniel Soudry. 2021. Accelerated sparse neural training: A provable and efficient method to find n:m transposable masks. *Advances in Neural Information Processing Systems*, 34:21099–21111. Introduces the mask-diversity metric.

Dayal Singh Kalra and Maissam Barkeshli. 2024. Why warmup the learning rate? underlying mechanisms and improvements. *arXiv preprint arXiv:2406.09405*.

Minhyung Kang and Luca Schroeder. 2018. Reinforcement learning for solving yahtzee. AA228: Decision Making under Uncertainty, Stanford University, class project report.

Diederik P. Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*. Published as a conference paper at ICLR 2015.

Vijay Konda and John Tsitsiklis. 1999. Actor-critic algorithms. In *Advances in Neural Information Processing Systems*, volume 12. MIT Press.

Clare Lyle, Zeyu Zheng, Khimya Khetarpal, James Martens, Hado van Hasselt, Razvan Pascanu, and Will Dabney. 2024. Normalization and effective learning rates in reinforcement learning. In *Advances in Neural Information Processing Systems*.

Volodymyr Mnih, Adria Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous methods for deep reinforcement learning. In *Proceedings of the 33rd International Conference on Machine Learning (ICML)*.

Matej Moravčík, Martin Schmid, Neil Burch, Viliam Lisý, Dustin Morrill, Nolan Bard, Trevor Davis, Kevin Waugh, Michael Johanson, and Michael Bowling. 2017. Deepstack: Expert-level artificial intelligence in heads-up no-limit poker. *Science*, 356(6337):508–513.

Ian Osband, Charles Blundell, Alexander Pritzel, and Benjamin Van Roy. 2016. Deep exploration via bootstrapped DQN. In *Advances in Neural Information Processing Systems*, volume 29.

Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. 2013. On the difficulty of training recurrent neural networks. *arXiv preprint arXiv:1211.5063*.

Jakub Pawlewicz. 2011. Nearly optimal computer play in multi-player yahtzee. In *Computers and Games*, pages 250–262, Berlin, Heidelberg. Springer Berlin Heidelberg.

Martin L. Puterman. 1994. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley Series in Probability and Statistics. Wiley.

Prajit Ramachandran, Barret Zoph, and Quoc V. Le. 2017. Searching for activation functions. *arXiv preprint arXiv:1710.05941*.

John Schulman. 2016. The nuts and bolts of deep RL research. NIPS 2016 Deep Reinforcement Learning Workshop. Slides.

John Schulman, Philipp Moritz, Sergey Levine, Michael I. Jordan, and Pieter Abbeel. 2016. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. 2016. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. 2017. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359.

Haoran Sun, Yekun Chai, Shuohuan Wang, Yu Sun, Hua Wu, and Haifeng Wang. 2025. Curiosity-driven reinforcement learning from human feedback. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 23517–23534, Vienna, Austria. Association for Computational Linguistics.

Richard S. Sutton. 1988. Learning to predict by the methods of temporal differences. *Machine Learning*, 3(1):9–44.

Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction*, 2nd edition. MIT Press, Cambridge, MA.

Richard S. Sutton, David McAllester, Satinder Singh, and Yishay Mansour. 2000. Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems 12 (NIPS 1999)*, pages 1057–1063.

Arash Tavakoli, Fabio Pardo, and Petar Kormushev. 2018. Action branching architectures for deep reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32.

Gerald Tesauro. 1995. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68.

Philip Vasseur. 2019. Using deep q-learning to compare strategy ladders of yahtzee. Source code available at `https://github.com/philvasseur/Yahtzee-DQN-Thesis`.

Tom Verhoeff. 1999. Optimal solitaire yahtzee strategies (slides). `https://www-set.win.tue.nl/~wstomv/misc/yahtzee/slides-2up.pdf`.

Lex Weaver and Nigel Tao. 2013. The optimal reward baseline for gradient-based reinforcement learning. *CoRR*, abs/1301.2315.

Ronald J. Williams. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3-4):229–256.

Ronald J. Williams and Jing Peng. 1991. Function optimization using connectionist reinforcement learning algorithms. *Connection Science*, 3(3):241–268.

Max Yuan. 2023. Using deep q-learning to play two-player yahtzee. Senior essay, Computer Science and Economics.

A AI Usage

This paper utilized artificial intelligence tools in the following ways:

- **GitHub Copilot (Claude Sonnet 4.5)** was used for typesetting assistance with LaTeX/KaTeX and for autocomplete suggestions during coding.
- **ChatGPT (GPT-5)** was used for brainstorming ideas for reinforcement learning applications in games, guidance in hyperparameter tuning, helping to outline the structure of the paper, and suggesting relevant related work.

All other content, including research methodology, analysis, results interpretation, and conclusions, represents original work by the author. The AI tools were used only for initial ideation and to assist research, not for generating substantive content or analysis.

B Yahtzee Scoring Rules

Next we define the indicator functions for each of the scoring categories:

$$
\begin{aligned}
\mathbb{I}_{\text{3k}}(\mathbf{d}) &= \mathbb{I}\{\max_v n_v(\mathbf{d}) \geq 3\} \\
\mathbb{I}_{\text{4k}}(\mathbf{d}) &= \mathbb{I}\{\max_v n_v(\mathbf{d}) \geq 4\} \\
\mathbb{I}_{\text{full}}(\mathbf{d}) &= \mathbb{I}\Big\{\exists i, j \in \{1, \ldots, 6\} \text{ with } n_i(\mathbf{d}) = 3 \wedge n_j(\mathbf{d}) = 2\Big\} \\
\mathbb{I}_{\text{ss}}(\mathbf{d}) &= \mathbb{I}\Big\{\exists k \in \{1, 2, 3\} \text{ with } \sum_{v=k}^{k+3} \mathbb{I}\{n_v(\mathbf{d}) > 0\} = 4\Big\} \\
\mathbb{I}_{\text{ls}}(\mathbf{d}) &= \mathbb{I}\Big\{\exists k \in \{1, 2\} \text{ with } \sum_{v=k}^{k+4} \mathbb{I}\{n_v(\mathbf{d}) > 0\} = 5\Big\} \\
\mathbb{I}_{\text{yahtzee}}(\mathbf{d}) &= \mathbb{I}\{\max_v n_v(\mathbf{d}) = 5\}
\end{aligned}
$$

The potential score for each category can then be defined as:

$$
\begin{aligned}
f_j(\mathbf{d}) &= j \cdot n_j(\mathbf{d}), \qquad j \in \{1, \ldots, 6\} \\
f_7(\mathbf{d}) &= \mathbf{1}^\top \mathbf{d} \cdot \mathbb{I}_{\text{3k}}(\mathbf{d}) \\
f_8(\mathbf{d}) &= \mathbf{1}^\top \mathbf{d} \cdot \mathbb{I}_{\text{4k}}(\mathbf{d}) \\
f_9(\mathbf{d}) &= 25 \cdot \mathbb{I}_{\text{full}}(\mathbf{d}) \\
f_{10}(\mathbf{d}) &= 30 \cdot \mathbb{I}_{\text{ss}}(\mathbf{d}) \\
f_{11}(\mathbf{d}) &= 40 \cdot \mathbb{I}_{\text{ls}}(\mathbf{d}) \\
f_{12}(\mathbf{d}) &= 50 \cdot \mathbb{I}_{\text{yahtzee}}(\mathbf{d}) \\
f_{13}(\mathbf{d}) &= \mathbf{1}^\top \cdot \mathbf{d} \\
\mathbf{f}(\mathbf{d}) &= \big(f_1(\mathbf{d}), f_2(\mathbf{d}), \ldots, f_{13}(\mathbf{d})\big)
\end{aligned}
$$

C State Transition Function

P can be defined by the following generative process.

- If $r < 2$ and $a = k$, for each die i:
  - if $k_i = 1$, keep $d'_i = d_i$;
  - else sample $d'_i \sim \text{Unif}\{1, \ldots, 6\}$ independently.

  Set $c' = c,\ r' = r + 1,\ t' = t$.
- If $r = 2$ and $a = i$, set $d' = d$, update $c' = \text{score}(c, d, i)$, set $r' = 0,\ t' = t + 1$.