Yahtzee: Reinforcement Learning Techniques for Stochastic Combinatorial Games

Nick Pape
nap626
nickpape@utexas.edu

Abstract

¡ABSTRACT GOES HERE¿

1 Introduction

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 RL-based 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. These factors make 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, 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 an agent reach optimal performance under a fixed training budget, using only self-play? (iii) Which design choices (state and action encodings, credit assignment, variance controls, baselines, entropy, reward shaping, etc) most affect final performance? (iv) What failure modes exist in learned policies and how could they be addressed?

2 Related Work

2.1 Policy Gradient Methods and Variance Reduction

2.1.1 Return Estimation

In this paper, we follow notation from Sutton and Barto (2018) and the policy gradient theorem (Sutton et al., 2000).

In long episodic games, the choice of return calculation affects sample efficiency, bias, and variance. 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 "bootstrapped" 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 win-

dow; this is called the TD estimate (Sutton and Barto, 2018).

This time window can also be adjusted, depending on the task. For example, n-step returns $G_t^{TD(n)}$ 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. A related method is $TD(\lambda)$, which uses an exponentially weighted average of n-step returns, effectively blending multiple time horizons into a single estimate controlled by λ (Sutton and Barto, 2018).

While TD estimates are biased (since they rely on future value estimates to be accurate), they have much lower variance than full-episode returns. 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. 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.

Pure TD methods can also be viewed as a form of approximate dynamic programming, making them a natural fit for domains where dynamic-programming solutions exist (Bertsekas and Tsitsiklis, 1996).

2.1.2 Policy Gradient Methods

Policy-gradient methods are a family of algorithms which directly optimize a parameterized policy π_θ to follow an estimate of the performance gradient. 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. One trick for reducing variance in REINFORCE is to subtract a baseline (often just an average return, but potentially a learned estimate) from an episode's MC return. This yields an advantage estimate that reduces variance without changing its expectation (Weaver and Tao, 2013; Greensmith et al., 2004).

Actor-critic methods (Konda and Tsitsiklis, 1999) such as Advantage Actor-Critic (A2C) and Asynchronous Advantage Actor-Critic (A3C) (Mnih et al., 2016) typically use a TD-style return estimate to update the policy. These methods learn a separate value function: the critic V_ϕ. This critic is used directly in the TD return estimate as the bootstrap value estimate for a state. For these methods, we can define the TD error δ_t as the difference between the TD estimate and the value estimate for the current state $V(S_t)$. This δ_t error is then used as the advantage estimate for a normal policy gradient update (Konda and Tsitsiklis, 1999).

Another widely used algorithm, proximal policy optimization (PPO), utilizes a clipped objective $L^{CLIP}(\theta)$ and explicit Kullback-Leibler (KL) divergence control to dramatically reduce variance and ensure stable updates (Schulman et al., 2017). PPO uses the Generalized Advantage Estimate (GAE), which is closely related to $TD(\lambda)$, applying a λ-weighted mixture at the level of advantages (Schulman et al., 2016).

2.1.3 Other Variance Reduction Techniques

Aside from return estimation, there is a host of other variance reduction techniques which can be employed for policy gradient methods.

Normalizing advantages across a batch improves gradient conditioning and is common practice (Schulman et al., 2015). 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; Mnih et al., 2016; Schulman et al., 2017). 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.

2.1.4 Reward Shaping

For games that have sparse, delayed, or hard-to-reach rewards, reward shaping can be used to improve learning speed and stability. Conceptually, reward shaping involves defining a potential function: $\Phi(s)$. Environmental rewards are then augmented with the weighted difference in potential between states in a trajectory. This has been shown to give practitioners the ability to change learning patterns while keeping the underlying optimal policy invariant (Ng et al., 1999). The potential function can be hand-designed or learned, although a learned potential function could inadvertently change the optimal policy if not done care-

fully (Devlin et al., 2014).

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. Achieving a near DP optimal score in solitaire *Yahtzee* is a necessary first step towards solving this setting.

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, with moderate success.

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 not a true reinforcement learning approach, Dutschke reports a statistical agent achieving a score of 241.6 ± 40.7 after just 8,000 games, using a combination of heuristics.

3 Problem Formulation

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:

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

and the current state of the dice is represented as:

$$\mathbf{d} \in \mathcal{D} \tag{1}$$

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

$$\mathbf{c} = (c_1, c_2, \ldots, c_{13}) \text{ where } c_i \in \mathcal{D}_i \cup \{\varnothing\} \tag{2}$$

where $\varnothing$ indicates an unused category.

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

$$\begin{aligned} n_v(\mathbf{d}) &= \sum_{i=1}^{5} \mathbb{I}(d_i = v), \quad v \in \{1, \ldots, 6\} \\ \mathbf{n}(\mathbf{d}) &= \big(n_1(\mathbf{d}), \ldots, n_6(\mathbf{d})\big) \end{aligned} \tag{3}$$

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

$$\mathbf{f}(\mathbf{d}) = \big(f_1(\mathbf{d}), f_2(\mathbf{d}), \ldots, f_{13}(\mathbf{d})\big) \tag{4}$$

The current turn number can be represented as:

$$t \in \{1, 2, \ldots, 13\}, \quad t = \sum_{i=1}^{13} \mathbb{I}(c_i \neq \varnothing) \tag{5}$$

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:

$$\mathbf{k} \in \{0, 1\}^5 \tag{6}$$

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:

$$\mathbf{s} \in \{0, 1\}^{13}, \quad \|\mathbf{s}\|_1 = 1 \tag{7}$$

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:

$$\begin{aligned} &\mathbf{u}(\mathbf{c}) \in \{0, 1\}^{13} \\ &\mathbf{u}(\mathbf{c})_i = \mathbb{I}\big(c_i \neq \varnothing\big), \quad \forall i = \{1, \ldots 13\} \end{aligned} \tag{8}$$

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:

$$\text{score}(\mathbf{c}) = B(\mathbf{c}) + \big\langle \mathbf{u}(\mathbf{c}), \mathbf{c} \big\rangle \tag{9}$$

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 F. 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 (totalling 39 individual steps) 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 and increased variance. Additionally, the network must learn to balance optimal single-turn play with long-term strategies, such as planning for the upper bonus.

4 Methodology

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

$$\mathbf{x} = \phi(\mathbf{s}) \tag{10}$$

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 3 different dice representations:

$$\begin{aligned}
\phi_{\text{dice}}^{\text{onehot}}(\mathbf{d}) &= \big[\text{onehot}(d_1), \ldots, \text{onehot}(d_5)\big] \\
\phi_{\text{dice}}^{\text{bin}}(\mathbf{d}) &= \mathbf{n}(\mathbf{d}) \\
\phi_{\text{dice}}^{\text{combined}}(\mathbf{d}) &= \big[\phi_{\text{dice}}^{\text{onehot}}(\mathbf{d}), \phi_{\text{dice}}^{\text{bin}}(\mathbf{d})\big]
\end{aligned}$$

A simple linear representation using the face values of the dice was also tested, but found to perform poorly and was abandoned early in experimentation.

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:

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

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.

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

We also defined a lock-in feature to indicate whether scoring in a given upper category would secure the upper bonus:

$$\begin{aligned}
&\phi_{\text{lockin}}(\mathbf{d}, \mathbf{c}) \in \{0,1\}^6, \\
&\phi_{\text{lockin},k}(\mathbf{d}, \mathbf{c}) = \mathbb{I}\Big\{\sum_{i=1}^{6} \mathbf{u}(\mathbf{c})_i \cdot c_i + f_k(\mathbf{d}) \geq 63\Big\}
\end{aligned}$$

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.

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

4.3 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 L (typically 2 or 3) fully connected hidden layers. The width of each layer (hidden size d_h) is 600 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), dropout with rate p_d for regularization, 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 Optimization & Schedules

We utilize the Adam optimizer (Kingma and Ba, 2014) with maximum learning rate α, typically 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 to a minimum ratio r_α of the maximum (Defazio et al., 2023; Lyle et al., 2024).

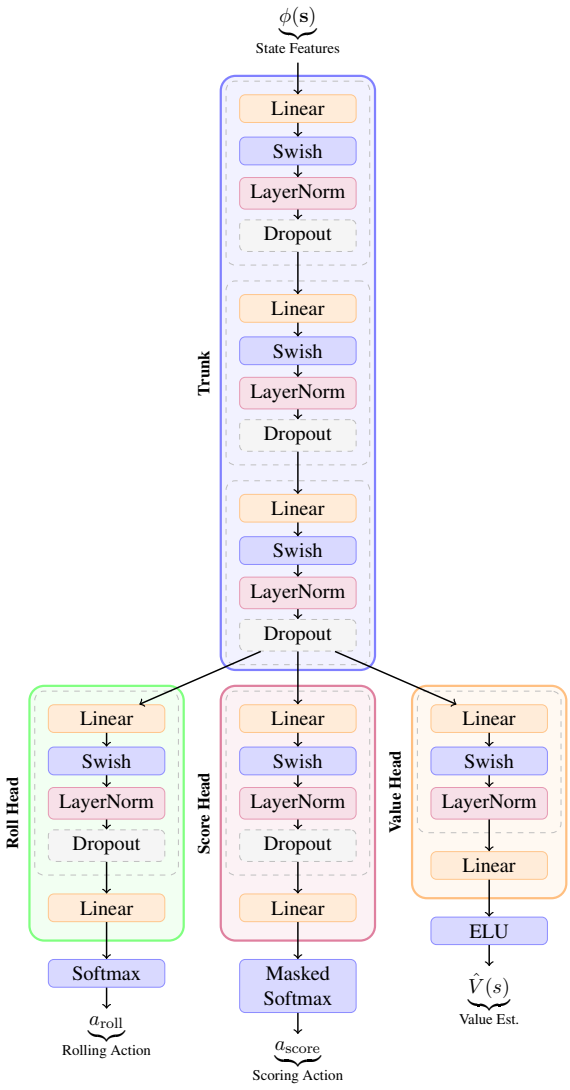

Figure 1: Overall network architecture with shared trunk and three specialized heads

4.3.4 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; En-

gstrom et al., 2020). Gradient clipping is applied with threshold τ_{clip} to prevent destabilizing updates. 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.

4.4 Reinforcement Learning

4.4.1 Reward Shaping

We also performed an experiment with reward shaping, to assess its impact on the model's final performance and ability to learn the bonus.

One of the heads of our model is an upper bonus regression head, which predicts the expected upper section score at the end of the game: $\hat{U}_\theta(s)$. The target is the normalized final upper section score at the end of the episode:

$$U_{\text{norm}} = \frac{U_{\text{final}}}{63} - 1 \in [-1, \frac{5}{3}]$$

This is trained using L_2 loss:

$$\mathcal{L}_{\text{upper}}(\theta) = \overbrace{\beta_{\text{regression}}}^{\text{weight}} \left\| \hat{U}_\theta(s) - U_{\text{norm}} \right\|_2^2$$

We can convert the normalized score back to a predicted upper score and use it in a potential-based reward shaping function:

$$\Phi(s) = 35 \cdot \text{clamp}\big(63 \cdot (\hat{U}_\theta(s) + 1), 0, 63\big) \tag{11}$$

We then modify the rewards using the potential-based shaping formula (Ng et al., 1999):

$$R'(s, a, s') = R(s, a, s') + \beta_{\text{shape}} \cdot (\gamma\Phi(s') - \Phi(s)) \tag{12}$$

Since the potential function Φ is changing during training, this may violate Ng's conditions for policy invariance. However, we wanted to see if it could help the model learn to go for the upper bonus more effectively.

For simplicity, we utilize r to denote the shaped reward R' for the remainder of this paper.

This head's architecture is similar to the value head, with a fully connected hidden layer followed by a linear output, with no activation.

4.5 Entropy

To encourage exploration, we also add an entropy bonus to the loss function (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 and won't explore other strategies as they often trade off short-term rewards for long-term gains.

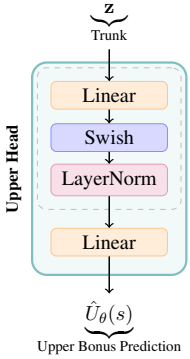

Figure 2: Upper bonus regression head architecture for reward shaping

We can define the entropy bonus as:

$$\mathcal{L}_{\text{entropy}}(\theta) = \overbrace{\beta_{\text{roll}}}^{\text{weight}} \underbrace{\mathcal{H}\big[\pi_{\theta,\text{roll}}(\cdot \mid s_t)\big]}_{\text{rolling action entropy}} + \overbrace{\beta_{\text{score}}}^{\text{weight}} \underbrace{\mathcal{H}\big[\pi_{\theta,\text{score}}(\cdot \mid s_t)\big]}_{\text{scoring action entropy}} \tag{13}$$

4.5.1 Auxilliary Losses

For all algorithms, we have auxilliary losses for both the shaping head and for entropy:

$$\mathcal{L}_{\text{aux}}(\theta) = \mathcal{L}_{\text{upper}}(\theta) + \mathcal{L}_{\text{entropy}}(\theta) \tag{14}$$

4.5.2 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, $V_\phi(\mathbf{s})$. We collect trajectories in batches of B games before computing gradient updates. Thus, the loss function is:

$$\mathcal{L}(\theta, \phi) = \underbrace{\overbrace{-\log\left(\pi_\theta(a_t \mid s_t)\right)}^{\text{negative log likelihood}} \overbrace{\left(\hat{R}_t - V_\phi(s_t)\right)}^{\text{advantage}}}_{\text{policy loss}} + \underbrace{\overbrace{\lambda_V}^{\text{weight}} \left\| V_\phi(s_t) - \hat{R}_t \right\|_2}_{\text{value loss}} + \mathcal{L}_{\text{entropy}(\theta)} \tag{15}$$

4.5.3 Advantage Actor-Critic (A2C)

Second, we utilize an episodic, one-step TD(0) Advantage Actor-Critic (A2C) method. Its loss is:

$$\delta_t = \overbrace{r_t}^{\text{reward}} + \overbrace{\gamma V_\phi(s_{t+1})}^{\text{bootstrap}} - \overbrace{V_\phi(s_t)}^{\text{current estimate}} \tag{16}$$

$$\begin{aligned}\mathcal{L}_{\text{TD-AC}}(\theta,\phi) = &\underbrace{\overbrace{-\log\left(\pi_\theta(a_t \mid s_t)\right)}^{\text{negative log likelihood}} \overbrace{\delta_t}^{\text{TD-error}}}_{\text{policy loss}} \\ &+ \underbrace{\overbrace{\lambda_V}^{\text{weight}} \left\|\delta_t\right\|_2^2}_{\text{value loss}} \\ &+ \mathcal{L}_{\text{aux}}(\theta)\end{aligned} \tag{17}$$

As this turned out to be the most successfully tuned algorithm, this is the only one for which we attemped reward shaping.

4.5.4 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). The loss we use is:

$$r_t(\theta) = \frac{\overbrace{\pi_\theta(a_t \mid s_t)}^{\text{current policy}}}{\underbrace{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}_{\text{behavior policy}}} \tag{18}$$

$$\begin{aligned}\mathcal{L}(\theta,\phi) = &\underbrace{-\min\left\{\begin{matrix} r_t(\theta)\,\hat{A}_t, \\ \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t \end{matrix}\right\}}_{\text{policy loss}} \\ &+ \underbrace{\overbrace{\lambda_V}^{\text{weight}} \left\|V_\phi(s_t) - \hat{R}_t\right\|_2^2}_{\text{value loss}} \\ &+ \mathcal{L}_{\text{entropy}(\theta)}\end{aligned} \tag{19}$$

4.5.5 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.

During training, we run 1,000 game episodes every 5 epochs (1% of training) to monitor progress. These are run using deterministic actions (i.e., taking the action with highest probability) to get a clear picture of the learned policy's performance. At the end of training, we run a final evaluation of 10,000 games to get a robust estimate of the agent's performance.

5 Results

5.1 Single-Turn Results

5.1.1 Baseline Single-Turn Performance

For state representation, the baseline model utilizes:

$$\phi(\mathbf{s}) = [\phi_{\text{dice}}^{\text{combined}}(\mathbf{d}), \phi_{\text{cat}}(\mathbf{c}), \phi_{\text{bonus}}(\mathbf{c}), \phi_{\text{rolls}}(r)]$$

For outputs, it uses Bernoulli rolling actions and categorical scoring actions. The single turn model has a short horizon (3 steps); REINFORCE was the natural choice here. We trained on ¡X¿ single-turn episodes, 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 5.1.2.

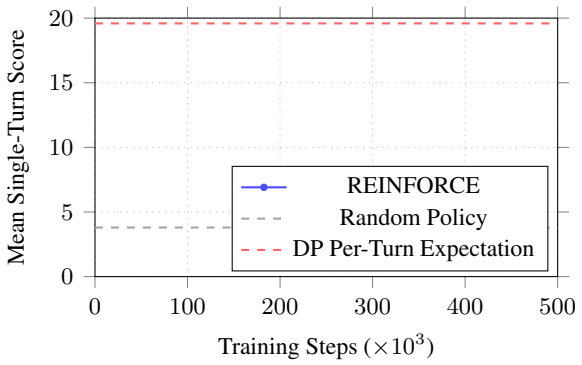

Figure 3: Single-turn agent performance during training (placeholder data)

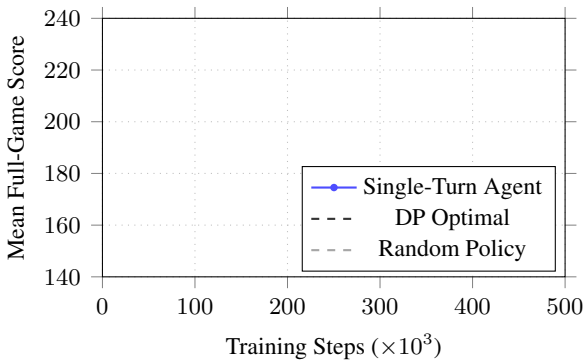

Figure 4: Full-game performance of single-turn agent (placeholder data)

5.1.2 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.

¡Pick one:¿

¡As expected, we can see that full game performance generally increases linearly with single-turn performance. However, at very high levels of full-game performance, single-turn performance begins to plateau, and even decline slightly. Since the single-turn model does not have access to the full game context, these are actually imperfectly optimizing their purported objective. This indicates that selecting hyperparameters for a single-turn model based on full-game performance is indeed a form of target leakage.¿

¡Contrary to our expectations, we do not see a clear tradeoff between single-turn and full-game performance at high performance levels. This suggests that single-turn optimization is a good proxy for full-game performance, or that the model is able to reconstruct sufficient full-game context from the single-turn state representation to make effective long-term decisions.¿

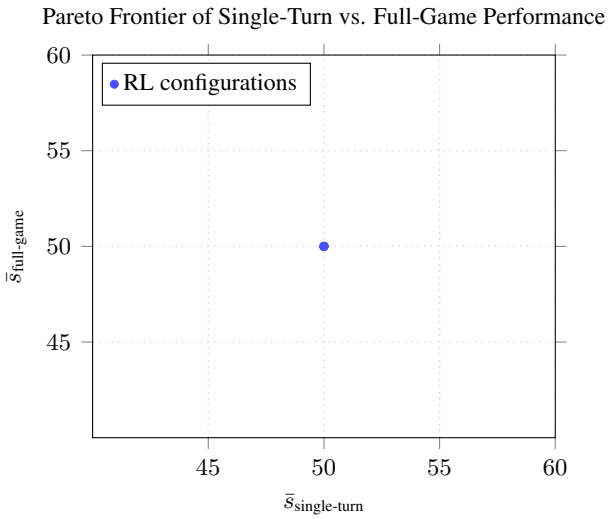

Figure 5: Single-turn vs full-game performance (placeholder data)

5.2 Full-Game Results

For the full-game model, we added several additional features to the state representation: $\phi_{\text{progress}}(t)$ and $\phi_{\text{potential}}(\mathbf{d}, \mathbf{c})$ while reusing the same underlying neural network architecture as the single-turn model. These additions were necessary to provide the model with sufficient context to make long-term strategic decisions. While the $\phi_{\text{progress}}(t)$ feature could be inferred from the scorecard, the model struggled to do so reliably. We intentionally omitted the $\phi_{\text{potential}}(\mathbf{d}, \mathbf{c})$ feature in single turn, as we wanted to ensure the model was capable of learning to reason about category potential on its own, but found it to be necessary, especially with REINFORCE.

5.2.1 Algorithm Comparison: REINFORCE, A2C, PPO

REINFORCE proved challenging to optimize to high performance levels given our fixed training budget of 1 million full-game episodes (39 million steps). It was highly sensitive to hyperparameters such as the critic coefficient, the entropy bonus, and batch size. We also found that REINFORCE simply required more training data to converge at a reliable performance level across seeds; our implementation was trained on 1,000,000 games. However, after optimization we were able to achieve reasonable performance, scoring a mean of ¡X¿ points on average over 10,000 full games.

Our most successful algorithm was TD(0)-style Actor-Critic (A2C). We found is easiest to tune and with an immediate performance boost over REINFORCE. This was the algorithm we use for ablation studies in Section **??**. With a fixed training budget of 1 million full-games, A2C was able to approach DP-optimal performance.

We also attempted to use Proximal Policy Optimization (PPO) with TD(0), but found it difficult to tune effectively within our computational budget. Each PPO rollout requires k epochs of minibatch updates, which significantly increases training time compared to A2C and REINFORCE. Sample efficiency wasn't a huge factor for us, since Yahtzee is so easy to simulate. For fair comparison to the other algorithms, we had to reduce the total number of games seen during training by a factor of k. However, it is possible PPO could reach or surpass A2C performance with more extensive hyperparameter tuning.

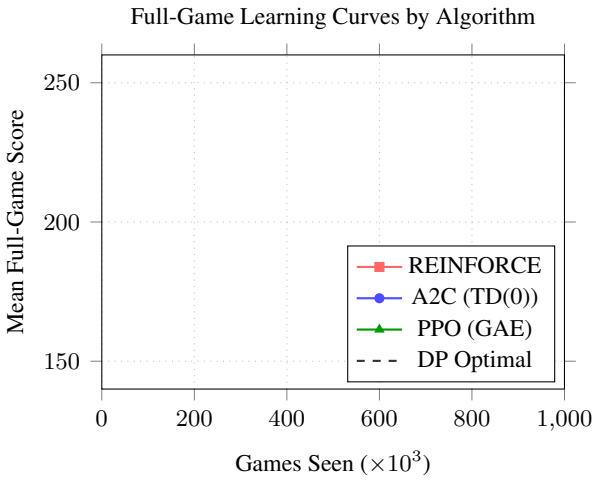

Figure 6: Algorithm comparison learning curves (placeholder data)

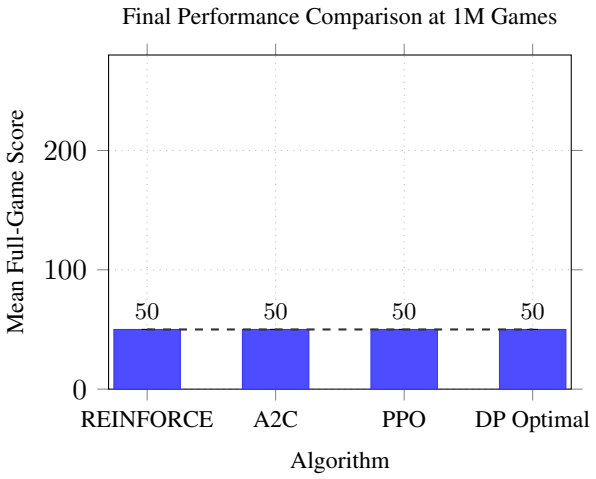

Figure 7: Final performance, mean score (placeholder data)

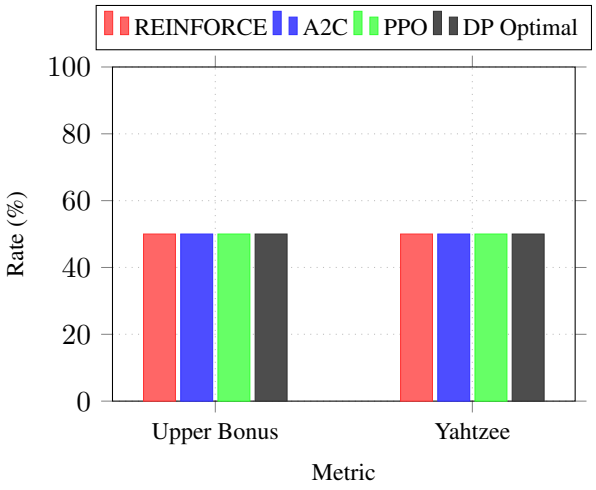

Figure 8: Bonus and Yahtzee achievement rates (placeholder data)

5.2.2 Representational Ablations

While a number of additional representational choices were explored, one of the most important is the state representation of the dice. One important thing to note is that our environment always sorts the dice in ascending order before passing them to the agent. If this was not done, the network would be forced to waste capacity on understanding permutations of the same dice configuration, which we found early on was a significant impediment to learning. ¡Future work could explore permutation-invariant architectures such as Deep Sets (**?**) or embeddings with self-attention to handle unsorted dice.¿

We wanted to highlight the importance of using a combined representation of both one-hot encodings and counts-based encodings of the dice. While the network could theoretically learn to reconstruct either representation from the other, in practice we found that using both improved performance.

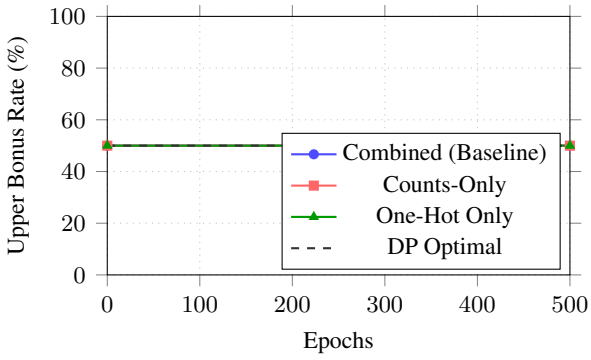

Figure 9: Upper bonus achievement by dice representation (placeholder data)

For the full-game model, we added several additional features beyond the single-turn representation. To understand the importance of each, they were ablated individually.

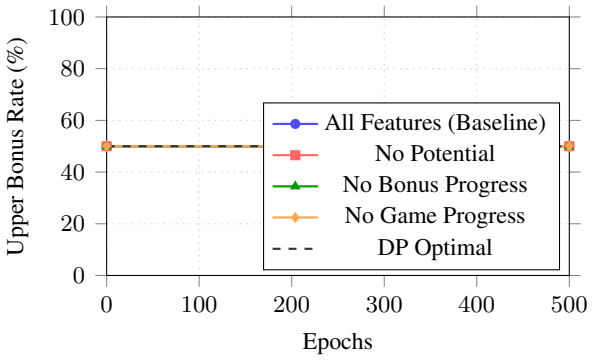

Figure 10: Upper bonus achievement by feature ablation (placeholder data)

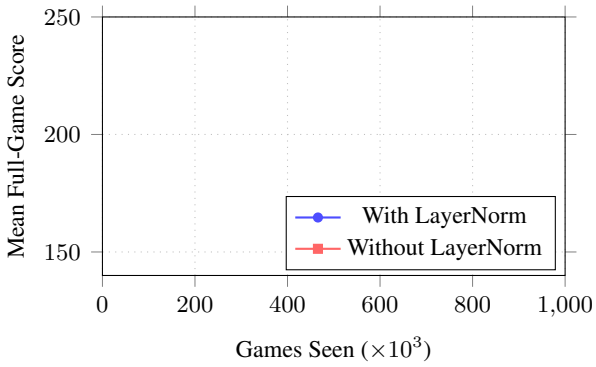

Figure 12: Learning curves with and without LayerNorm (placeholder data)

5.2.3 Architectural Ablations

¡Discuss network architecture choices and their impact¿

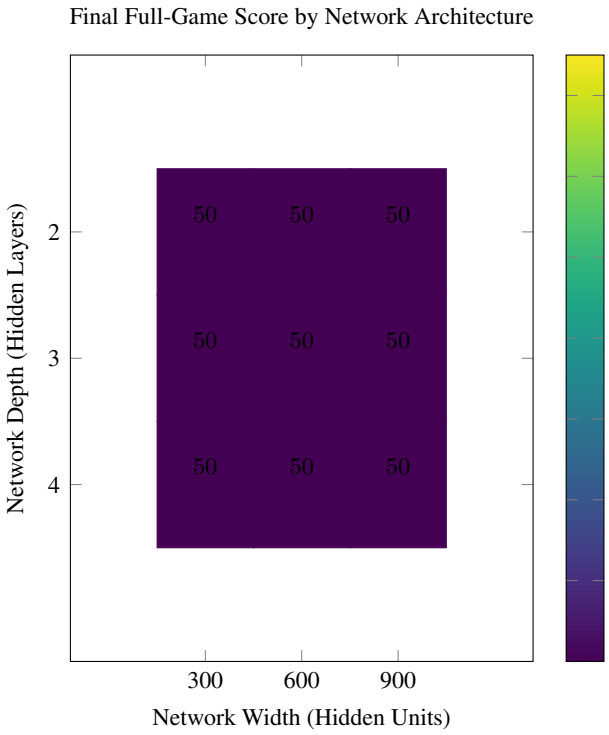

Figure 11: Network architecture ablation (placeholder data)

¡Discuss LayerNorm's impact on training stability and performance¿

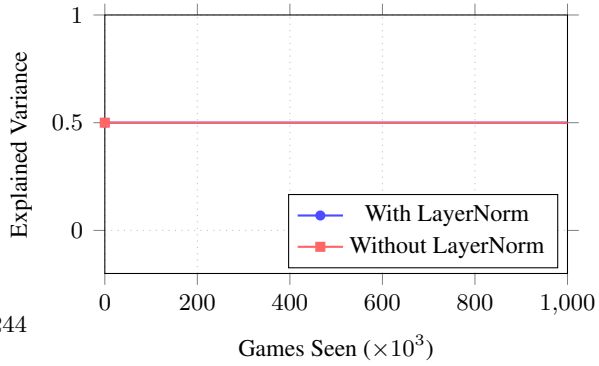

Figure 13: Value function quality with and without LayerNorm (placeholder data)

5.2.4 Credit Assignment: TD(0) vs GAE

¡Discuss the effect of GAE lambda on performance and training dynamics¿

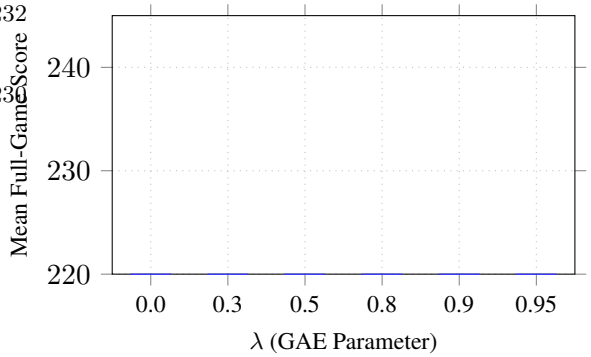

Figure 14: Final performance by GAE lambda (placeholder data)

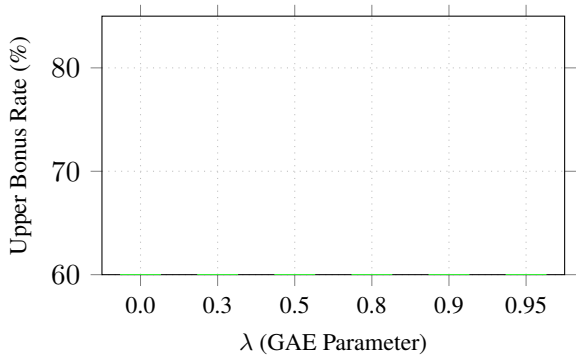

Figure 15: Upper bonus achievement by GAE lambda (placeholder data)

5.2.5 Entropy Sensitivity

¡Discuss the importance of entropy scheduling for exploration and policy collapse prevention¿

Table 1: Entropy regime definitions

Regime	β_{roll}	β_{score}
Low Entropy	$0.005 \rightarrow 0.001$	$0.01 \rightarrow 0.002$
Baseline	$0.01 \rightarrow 0.002$	$0.02 \rightarrow 0.005$
High Entropy	$0.02 \rightarrow 0.005$	$0.04 \rightarrow 0.01$

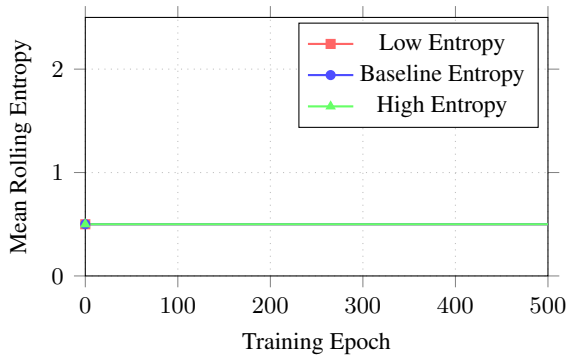

Figure 16: Rolling-head entropy over training (placeholder data)

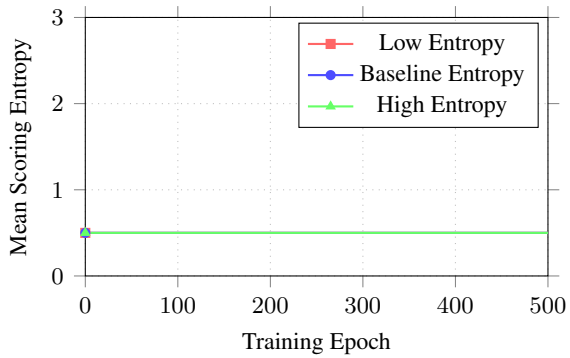

Figure 17: Scoring-head entropy over training (placeholder data)

Table 2: Entropy regime performance (placeholder data)

Regime	**Mean Score**	**Bonus %**	**Yahtzee %**
Low Entropy	$< X > \pm < Y >$	¡X¿	¡Y¿
Baseline	$< X > \pm < Y >$	¡X¿	¡Y¿
High Entropy	$< X > \pm < Y >$	¡X¿	¡Y¿

¡Discuss how low entropy causes early policy collapse (lower scores and bonus rates), baseline achieves best performance, and high entropy maintains exploration but slows learning¿

5.2.6 Summary

Table 4: Score distribution (placeholder data)

n	$P(\textbf{score} \geq n)$
50	$< X >$
100	$< X >$
150	$< X >$
200	$< X >$
250	$< X >$
300	$< X >$
400	$< X >$
500	$< X >$
750	$< X >$
1000	$< X >$
1250	$< X >$
1500	$< X >$

¡Summarize full-game results and key findings¿

5.3 Policy Analysis

5.3.1 Category Usage

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

¡Summarize full-game results and key findings¿

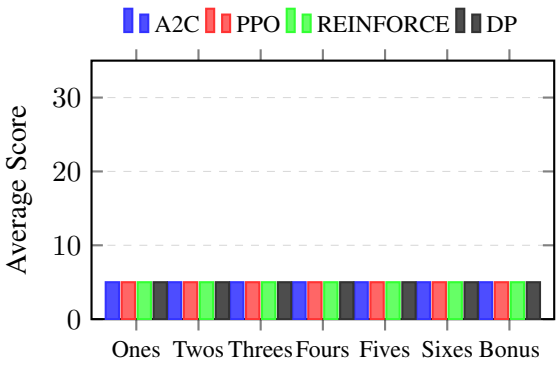

Figure 18: Upper section and bonus scores (placeholder data)

Table 3: Full-game performance summary (placeholder data)

Algorithm	Training Budget	Mean Score	Std Dev	Bonus Rate (%)	Yahtzee Rate (%)	≥250 (%)
DP Optimal	–	$< X >$	–	$< X >$	$< Y >$	$< Z >$
A2C (TD(0), best config)	1M games	$< X >$	$< Y >$	$< X >$	$< Y >$	$< Z >$
PPO ($\lambda = 0.5$, best config)	1M games	$< X >$	$< Y >$	$< X >$	$< Y >$	$< Z >$
REINFORCE (full-game)	1M games	$< X >$	$< Y >$	$< X >$	$< Y >$	$< Z >$
Single-Turn REINFORCE (rollout)	500K games	$< X >$	$< Y >$	$< X >$	$< Y >$	$< Z >$

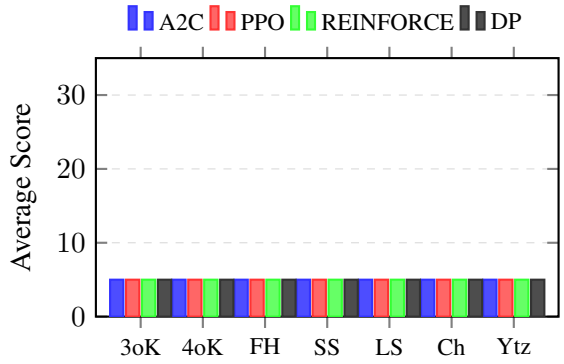

Figure 19: Lower section scores (placeholder data)

¡table comparison against DP optimal solutions¿

5.3.2 Strategy Comparison Across Agents

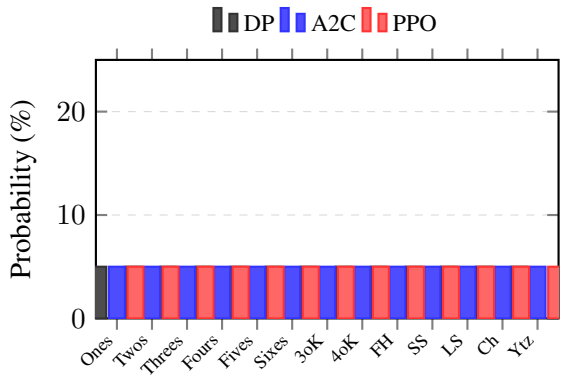

Figure 20: First category chosen distribution (placeholder data)

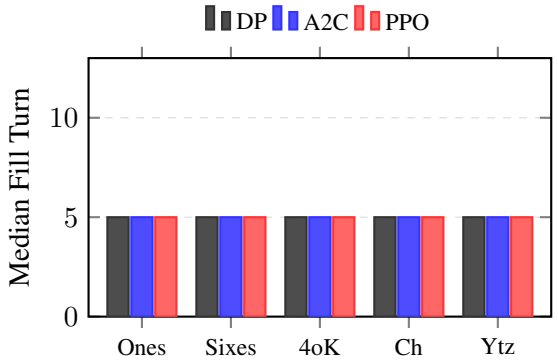

Figure 21: Median fill turn for key categories (placeholder data)

¡Compare strategies learned by different agents¿

6 Discussion

6.1 Summary

¡Discuss implications of results, limitations, and potential improvements.¿

6.2 Strategy & Failure Modes

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

7 Conclusion and Future Work

Learning a robust policy for *Yahtzee* using reinforcement learning presents several interesting challenges and insights. First, we showed that *Yahtzee*'s combinatorial action space and sparse rewards make it suitable as a non-trivial benchmark environment. Our results back up theoretical results in the literature regarding training stability and sample efficiency of common RL algorithms. Likewise, our ablation studies highlight the importance of finding semantically meaningful state and action representations that align the model architecture with the underlying structure of the problem. Our analysis of learned policies showed that these algorithms often struggle to learn rare, yet high-reward strategies, especially if they require strong coherence over longer time horizons.

Future research could be done to find architectures, samples, and learning methods that allow the model to better approximate optimal play. For example, curriculum learning approaches, where the agent is gradually exposed to more complex scenarios over time, could be used to help the model overcome some challenges outlined in this paper.

We found that Yahtzee is trivially broken into several a heirarchy of interesting sub-problems. It was fairly expensive to train a full-game agent from scratch, but training a single-turn agent was much more efficient. Transfer learning could be explored further to see if knowledge from single-turn optimization could be effectively transferred to full-game, multiplayer Yahtzee, or other variants of the game. Additionally, Yahtzee could also be considered as a candidate environment for research into hierarchical reinforcement learning (HRL) methods.

Perhaps most interestingly, we showed that an architecture that is, in theory, applicable to the multiplayer setting can be effectively trained to play Solitaire *Yahtzee* at a high level. This opens the door to future research into adversarial formulations of the game, which are intractable using analytic methods. Our agent tries only to achieve the best game score, but in a multiplayer setting, it would need to reason about opponents' strategies and scorecards and adapt accordingly to maximize its chances of winning. Likewise, the use of self-attention and other permutation-invariant architectures could be applicable to many games or scenarios involving multiple agents in a shared environment.

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.

D. Bertsekas and J.N. Tsitsiklis. 1996. *Neuro-Dynamic Programming*. Athena Scientific.

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*.

Sam Devlin, Logan Yliniemi, Daniel Kudenko, and Kagan Tumer. 2014. Potential-based difference rewards for multiagent reinforcement learning. In *Proceedings of the 2014 International Conference on Autonomous Agents and Multi-Agent Systems*, AAMAS '14, page 165–172, Richland, SC. International Foundation for Autonomous Agents and Multiagent Systems.

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.

Andrew Y. Ng, Daishi Harada, and Stuart J. Russell. 1999. Policy invariance under reward transformations: Theory and application to reward shaping. In *Proceedings of the Sixteenth International Conference on Machine Learning*, ICML '99, page 278–287, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

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, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. 2015. Trust region policy optimization. In *Proceedings of the 32nd International Conference on Machine Learning*, pages 1889–1897.

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 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 Reproducibility

The code for this project is available on GitHub.

Likewise, data for all experiments used in this graph is available in the Weights & Biases report.

The trained A2C full-game model can be found on HuggingFace.

B Hyperparameters

The following hyperparameters were used for the baseline models:

Table 5: Shared hyperparameters across all algorithms

Hyperparameter	Value
d_h (Hidden Size)	600
L (Hidden Layers)	3
p_d (Dropout Rate)	0.1
r_α (Min LR Ratio)	0.01
B (Games per Batch)	20
Activation Function	Swish
Rolling Action Representation	Categorical

Table 6: Algorithm-specific hyperparameters

Hyperparameter	REINFORCE	A2C	PPO
α	0.001	0.0001	0.0001
γ (min)	0.95	0.99	0.99
γ (max)	1.0	0.99	0.99
τ_{clip}	0.0	1.0	1.0
λ_V	0.025	0.005	0.01
β_{roll} (max)	0.1	0.06	0.02
β_{roll} (min)	0.01	0.02	0.005
β_{score} (max)	0.02	0.03	0.05
β_{score} (min)	0.003	0.008	0.01
Entropy Hold Period	0.1	0.3	0.1
Entropy Anneal Period	0.55	0.6	0.8

Table 7: PPO-specific hyperparameters

Hyperparameter	Value
PPO Clip ϵ	0.2
PPO Games per Minibatch	4
PPO Epochs	3

C Compute Costs

Experiments were collected using a mix of a local RTX 3090 and AWS-hosted Tesla T4 GPUs. The total cost of cloud compute was approximately **$130**. Over **312** training runs were logged in Weights & Biases, totaling approximately **566.59 GPU hours**. The estimated carbon footprint of the compute used is approximately **A kg CO_2e**, based on the methodology from **?**.

D 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, IDE autocomplete suggestions during coding, and to occassionally perform straightforward refactorings, CUDA performance optimizations, and debugging.
- **ChatGPT (GPT-5.1)** was used for brainstorming ideas for reinforcement learning applications in games, guidance in hyperparameter tuning, helping to outline the structure of this paper, assistance in discovering relevant research and citations, and for writing tone and quality feedback.

All other content, including research methodology, analysis, results interpretation, and conclusions, represents original work by the author. The AI tools were not used to generate substantive content or analysis in this document.

E Yahtzee Scoring Rules

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

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

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

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

F 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 $d'_i \sim \mathrm{Unif}\{1, \ldots, 6\}$ independently.

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