Yahtzee: Reinforcement Learning Techniques for Stochastic Combinatorial Games

Nicholas Pape
nap626
nickpape@utexas.edu

Abstract

Yahtzee is a classic dice game with a stochastic, combinatorial structure and delayed rewards, making it an interesting mid-scale RL benchmark. While an optimal policy for solitaire Yahtzee can be computed using dynamic programming methods, multiplayer is intractable, motivating approximation methods. We formulate Yahtzee as a Markov Decision Process (MDP), and train self-play agents using various policy gradient methods: REINFORCE, Advantage Actor-Critic (A2C), and Proximal Policy Optimization (PPO), all using a multi-headed network with a shared trunk. We ablate feature and action encodings, architecture, return estimators, and entropy regularization to understand their impact on learning.

Under a fixed training budget, REINFORCE and PPO prove sensitive to hyperparameters and fail to reach near-optimal performance, whereas A2C trains robustly across a range of settings. Our agent attains a median score of 241.78 points over 100,000 evaluation games, within 5.0% of the optimal DP score of 254.59, achieving the upper section bonus and Yahtzee at rates of 24.9% and 34.1%, respectively. All models struggle to learn the upper bonus strategy, overindexing on four-of-a-kind's, highlighting persistent long-horizon credit-assignment and exploration challenges.

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, 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).

There are multiple methods for assigning credit to actions taken by a policy. 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 window; 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. 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. Learning the value function over shorter horizons is a simpler problem than estimating the entire sequence of rewards, so in some environments, TD methods are more sample efficient than REINFORCE. They also provide the benefit of being able to learn online rather than waiting until the end of an episode.

Additionally, 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. 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; Ahmed et al., 2019; 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 carefully (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). RL methods can also be used to reach high levels of performance on adversarial games, 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). 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. 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

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, re-rolling the remaining ones. This process can be repeated two more times, for a total of three total 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:

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

There is an additional "Joker" bonus, detailed in Appendix E.

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).

The action is $\mathbf{a} = (\mathbf{k}, \mathbf{s})$, where $\mathbf{k}$ is the keep vector and s is the score category choice. This can be restated 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.

The reward is the change in total score between steps $R_t = \text{score}(c_{t+1}) - \text{score}(c_t)$.

Since we desire to maximize total score at the end of the game, $\gamma = 1$.

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, this allows us to quickly iterate on architecture and training choices with shorter training times and without the complications of long-term credit assignment.

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. To mitigate this, our environment always sorts the dice in ascending order before passing them to the agent.

We attempted 3 different dice representations:

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

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:

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

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.

$$\phi_{\text{progress}}(t) = \frac{t}{12}$$

$$\phi_{\text{rolls}}(r) \in \{0,1\}^3, \quad \|\phi_{\text{rolls}}(r)\|_1 = 1$$

$$\phi_{\text{joker}}(\mathbf{c}) \in \{0,1\}, \quad \text{(Joker rule active, see Appendix E)}$$

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

$$\phi_{\text{lockin}}(\mathbf{d},\mathbf{c}) \in \{0,1\}^6,$$

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

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 combinations 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.

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

During training, we mask out invalid scoring actions by setting their logits to $-\infty$ before applying the softmax function. To help with exploration (Tijsma et al., 2016), during training we sample from these distributions; during inference we take the argmax action.

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. We created the network using PyTorch (Paszke et al., 2019), and the training loop is implemented using `pytorch-lightning` (Falcon, 2019).

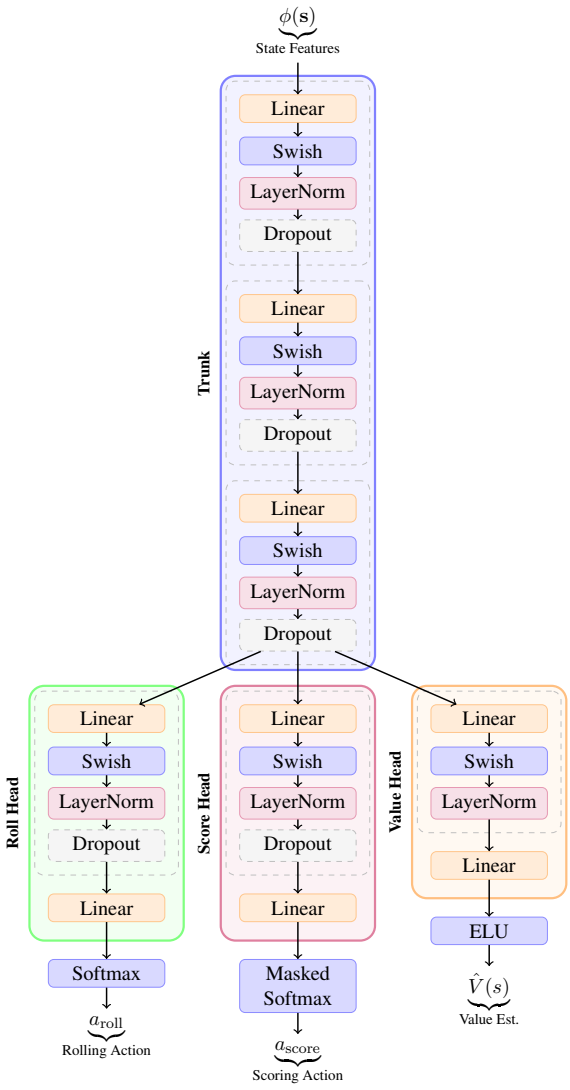

Figure 1: Overall network architecture with shared trunk and three specialized heads

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 typically 600 neurons, found through empirical hyperparameter tuning (and ablated in Section 5.2.3). We utilize layer normalization for improved training stability (Ba et al., 2016; Bjorck et al., 2022), dropout with rate p_d for regularization (Srivastava et al., 2014), 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 which outputs a scalar baseline for REINFORCE or the value estimate for actor-critic methods. 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*.

The rolling and scoring heads implement the distributions from Section 4.2.1 with a single hidden layer, each.

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 1×10^{-3}, tuned empirically. To improve training stability (Liu et al., 2025; Kalra and Barkeshli, 2024), we utilize a warmup schedule over the first 5% of training, plateau for 70% of training, and then linearly decay over the final 25% of training steps to a minimum ratio r_α (typically 5%) of the maximum (Defazio et al., 2023; Lyle et al., 2024).

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 log explained variance (Schulman, 2016; Schulman et al., 2016). To check 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 ensure learning stability, we track gradient norms and clip rate (Pascanu et al., 2013; Engstrom et al., 2020). Gradient clipping is applied with threshold τ_{clip} to prevent destabilizing updates. To ensure advantages are well-conditioned, we calculate advantage mean and standard deviation (Achiam, 2018). We also monitor standard training metrics such as average reward and loss values. All metrics were logged to Weights & Biases (Biewald, 2020).

4.4 Reinforcement Learning

4.4.1 Reward Shaping

We also implemented a learned potential reward shaping mechanism to assess its impact on the model's final performance and ability to learn the bonus.

First, we implement a new head which predicts the normalized final upper section score at the end of the episode. This head's architecture is similar to the value head, with a fully connected hidden layer followed by a linear output, with no activation, described in Figure 2. The target upper score is normalized to the range $[-1, \frac{5}{3}]$ using the formula:

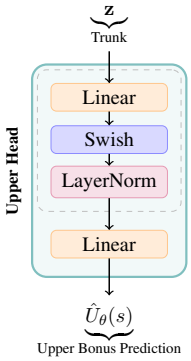

Figure 2: Reward Shaping: Upper bonus prediction head architecture

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

This head 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.

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.

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, $V_\phi(\mathbf{s})$. The loss function is:

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

4.5.3 Advantage Actor-Critic (A2C)

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

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

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

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

4.5.4 PPO

Lastly, we implement Proximal Policy Optimization (PPO) (Schulman et al., 2017); we tried this with TD(0) and GAE advantages. The loss function is:

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

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

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 100,000 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 260,000

games, 10 million examples, using a batch size of 1,014 examples, for 10,000 total gradient updates.

As shown in Figure 3, 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. Figure 4 demonstrates the full-game performance when evaluating the single-turn trained agent. However, we suspected target leakage (selecting parameters and architectures based on full-game performance) could also play a role. This is analyzed in Section 5.1.2.

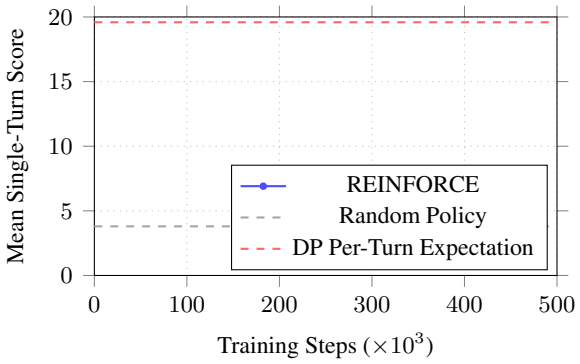

Figure 3: Single-turn agent performance during training (placeholder data)

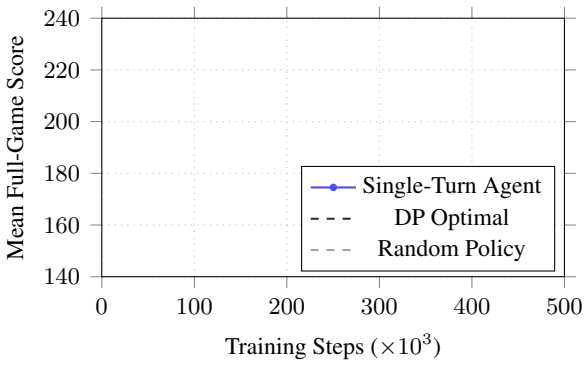

Figure 4: Full-game performance of single-turn agent (placeholder data)

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.

As we suspected, there is a Pareto frontier between these two objectives, as illustrated in Figure 5. 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 imperfectly optimizing their target objective. This indicates that selecting hyperparameters for a single-turn model based on full-game performance could indeed be a form of target leakage.

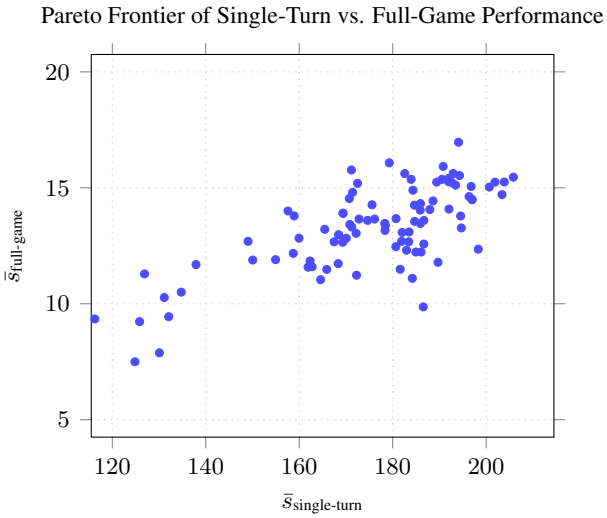

Figure 5: Single-turn vs full-game performance (placeholder data)

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. 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 improve stability, especially with REINFORCE.

5.2.1 Algorithm Comparison: REINFORCE, A2C, PPO

During development, we compared algorithms using a fixed training budget of 250,000 full games played. Later, we attempt a longer, 1 million full-game training run for our best algorithm.

REINFORCE proved challenging to optimize to high performance levels given our fixed training budget. It was sensitive to hyperparameters such as the critic coefficient, the entropy bonus, and

batch size. We also found that REINFORCE required more games to converge. After optimization we were able to achieve reasonable performance; the million game training run scored a mean of ¡X¿ points.

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 the ablation studies. With a training budget of 1 million full-games, A2C was able to approach DP-optimal performance: scoring 241.8 points average.

We also attempted to use Proximal Policy Optimization (PPO) with TD(0), but found it difficult to tune. Each PPO rollout requires k epochs of minibatch updates, which significantly increases training time compared to A2C and REINFORCE. For fair comparison to the other algorithms, reduced the total number of games seen during training by a factor of k. PPO was able to outperform REINFORCE, but was not able to reach A2C performance within our training budget. However, it is possible PPO could reach or surpass A2C performance with more extensive hyperparameter tuning.

Figure 6 shows the learning curves for all three algorithms during training. The final performance comparison is summarized in Figure 7, with detailed bonus and Yahtzee achievement rates shown in Figure 8.

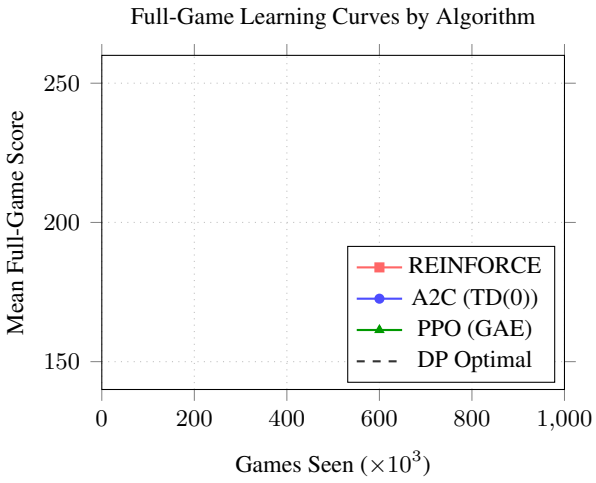

Figure 6: Algorithm comparison learning curves (placeholder data)

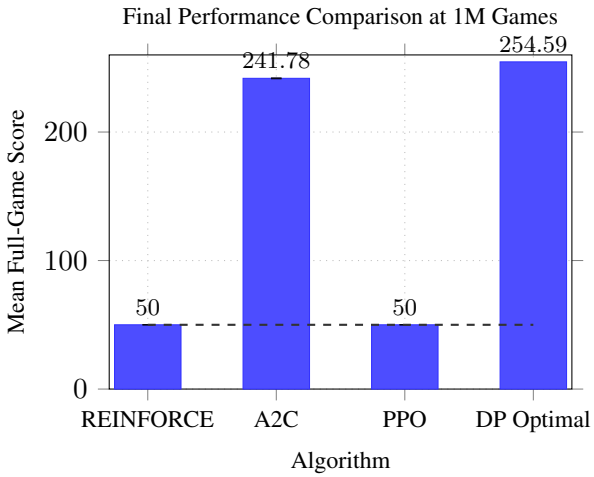

Figure 7: Final performance, mean score (placeholder data)

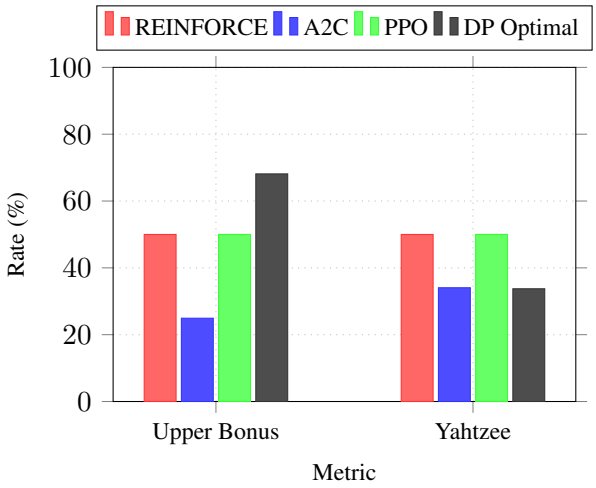

Figure 8: Bonus and Yahtzee achievement rates (placeholder data)

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.

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, as demonstrated in Figure 9.

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, with results shown in Figure 10.

Lastly, we tested our hypothesis that a 32-way

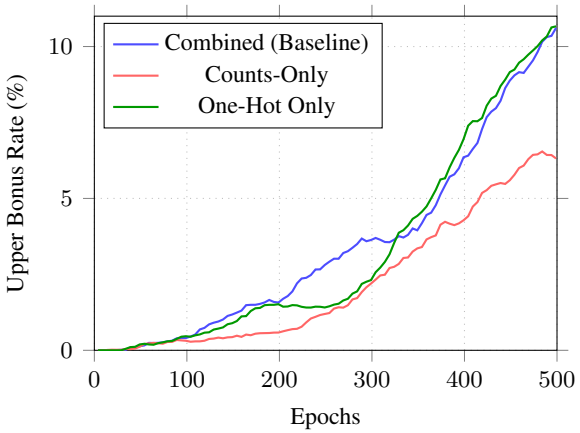

Figure 9: Exponential moving average (EMA) of upper-bonus achievement by dice representation

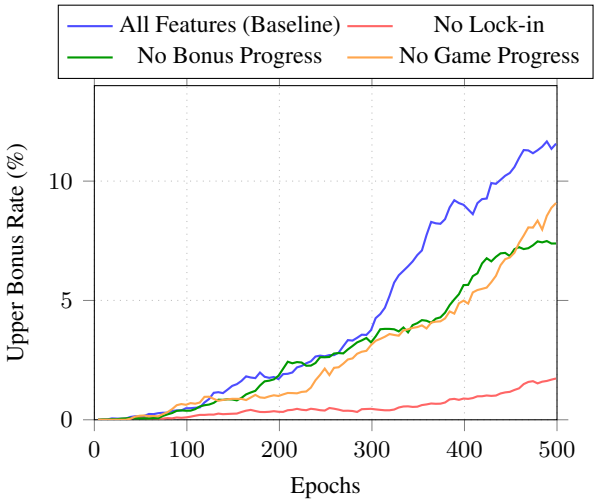

Figure 10: Upper bonus achievement (EMA) by feature ablation

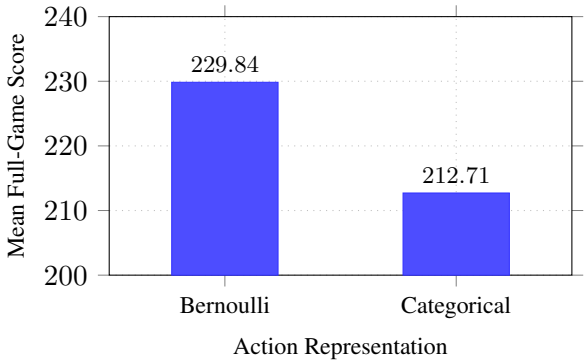

Figure 11: Performance comparison: Bernoulli vs Categorical action representation

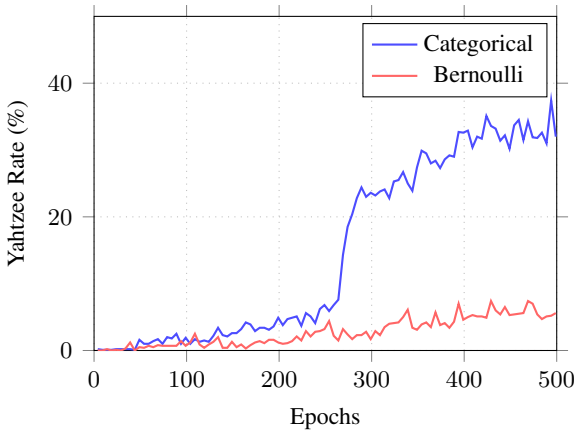

Figure 12: Learning Yahtzee with different action representations

categorical would prove beneficial to complex actions that required specific combinations of dice to be held (see Section 4.2.1). Figure 11 shows the performance comparison, while Figure 12 illustrates how each representation learns to achieve Yahtzee over training.

5.2.3 Architectural Ablations

We performed a simple grid search ablation to understand if our chosen architecture of 3 hidden layers of 600 units each was optimal. Yahtzee is a fairly complex game, so we expected shorter, but wider networks to perform best. Note that each of these has a different number of total parameters, so this is not a pure ablation of depth vs. width. Results are shown in Figure 13.

Based on (Bjorck et al., 2022), we hypothesized that layer normalization (Ba et al., 2016) would improve training stability and performance and used it in all of our main experiments. This was ablated to understand its true impact, with learning curves compared in Figure 14.

Lastly, we used Swish activation functions in our network for all of our main experiments, as it had been reported to work well in RL settings (Elfwing et al., 2017). We ablated this to ReLU to understand its true impact, as shown in Figure 15.

5.2.4 Credit Assignment: TD(0) vs GAE

Later in this research, we noticed the main issue with our network was that it was struggling to earn the bonus, learning it very slowly. We first hypothesized that this was due to high variance in REINFORCE, so we switched to A2C with TD(0) targets. However, the issue persisted. We then hypothesized that the TD(0) targets were not providing sufficient credit assignment for the long-term bonus reward, so we switched to GAE with various λ values to understand if this would help.

Unfortunately, we found that GAE did not im-

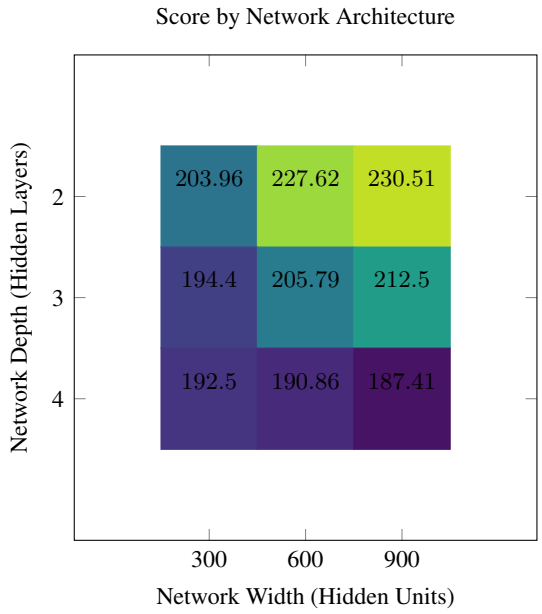

Figure 13: Network architecture ablation

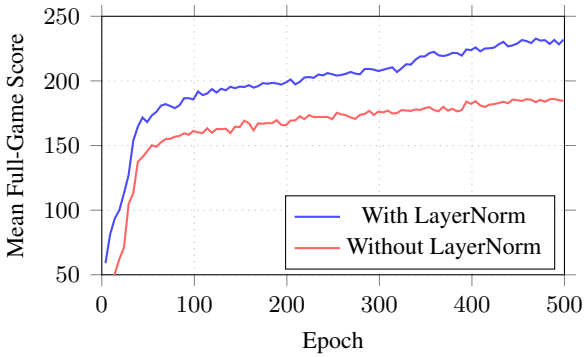

Figure 14: Learning curves with and without LayerNorm

prove performance over TD(0), and values that were too high ($\lambda \geq 0.8$) significantly degraded performance, as shown in Figures 16 and 17.

5.2.5 Entropy Sensitivity

During experimentation, we noticed that the entropy regularization coefficients had a significant impact on training stability and final performance, as described in Section 4.5. To better understand this sensitivity, we trained models under three different entropy regimes: Low Entropy, Baseline, and High Entropy, as defined in Table 1. The learning curves are shown in Figure 18, with rolling entropy values tracked in Figure 19.

The overall results for each entropy regime are summarized in Table 2.

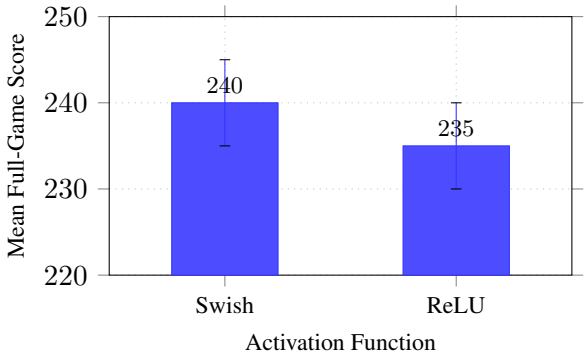

Figure 15: Performance comparison: Swish vs ReLU activation (placeholder data)

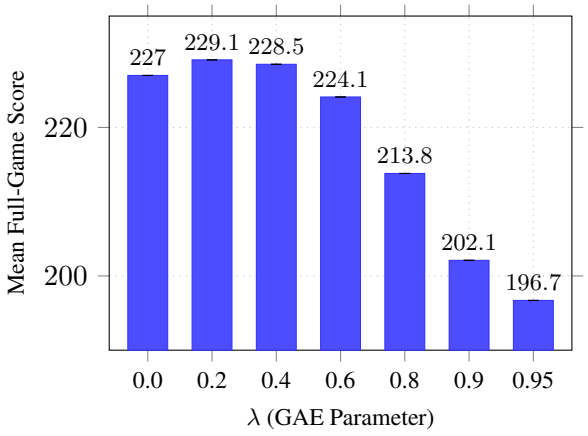

Figure 16: Final performance by GAE lambda

Table 1: Entropy regime definitions

Regime	β_{roll}	β_{score}	Hold / Anneal
None	0.0	0.0	0 / 0
Low	$0.05 \rightarrow 0.01$	$0.01 \rightarrow 0.005$	0.2 / 0.4
Baseline	$0.1 \rightarrow 0.02$	$0.03 \rightarrow 0.01$	0.3 / 0.6
High	$0.2 \rightarrow 0.04$	$0.06 \rightarrow 0.02$	0.35 / 0.65

Table 2: Entropy regime performance (placeholder data)

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

5.2.6 Summary

In summary, we found that A2C with TD(0) targets, a combined dice representation, Layer Normalization, SILU activations, and carefully tuned entropy regularization produced the best results. Table 3 presents a comprehensive comparison of all algorithms tested.

For our best configuration, A2C trained over 1

Table 3: Full-game performance summary

Algorithm	Training Budget	Mean Score	Std Dev	Bonus Rate (%)	Yahtzee Rate (%)	≥250 (%)
DP Optimal	–	254.59	–	68.12%	33.74%	48.37%
A2C	250K games	230.38	2503.83	11.37%	31.08%	27.82%
A2C	1M games	241.78	3230.86	24.93%	34.05%	36.87%
PPO ($\lambda = 0.3, k = 5$)	50k games	204.54	860.02	2.49%	6.54%	6.08%
PPO ($\lambda = 0.3, k = 4$)	250K games	$< X >$	$< Y >$	$< X >$	$< Y >$	$< Z >$
REINFORCE (full-game)	250K games	189.84	812.24	2.83%	1.70%	2.06%
REINFORCE (full-game)	1M games	$< X >$	$< Y >$	$< X >$	$< Y >$	$< Z >$
REINFORCE (single-turn)	500K games	201.48	1366.00	0.89%	19.63%	9.34%

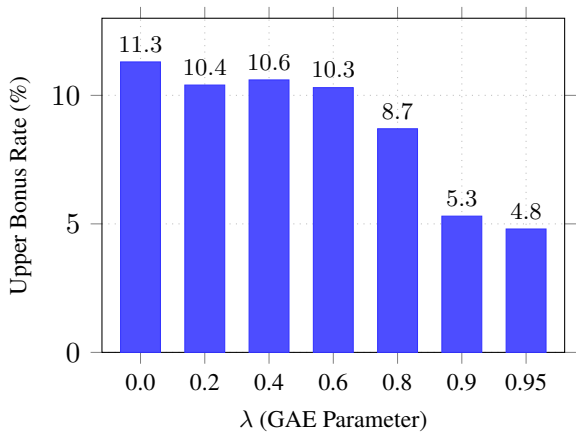

Figure 17: Upper bonus achievement by GAE lambda

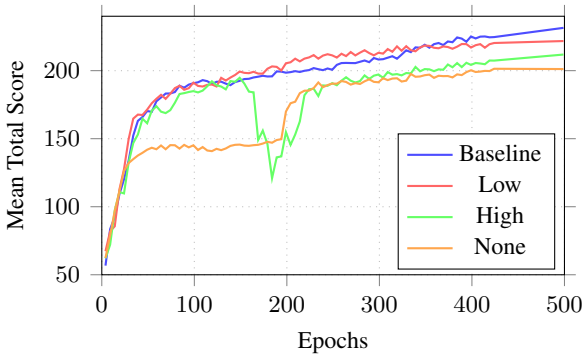

Figure 18: Mean total score across training epochs for different entropy regimes

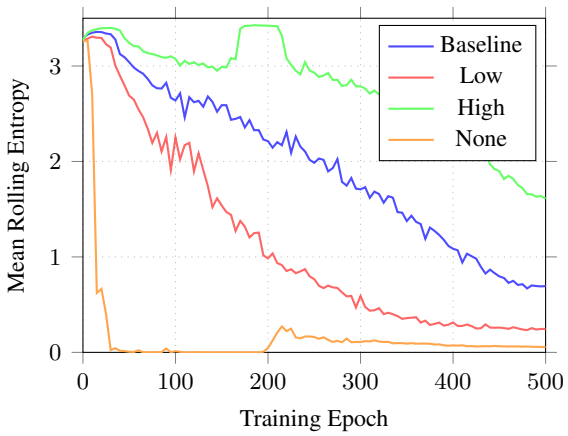

Figure 19: Mean rolling entropy across training epochs for different entropy regularization settings

Table 4: $P(\text{score} \geq n)$, 100,000 games

n	A2C	DP
50	1.000000	1.000000
100	0.999980	0.999998
150	0.989730	0.991230
200	0.820980	0.863584
250	0.368730	0.483683
300	0.109080	0.143265
400	0.025960	0.038351
500	0.004870	0.007192
750	0	$5.11603 \cdot 10^{-6}$
1000	0	$5.57508 \cdot 10^{-9}$
1250	0	$6.49213 \cdot 10^{-13}$
1500	0	$3.93308 \cdot 10^{-19}$

million games, the final score distribution is compared to DP-optimal in Table 4.

5.3 Policy Analysis

5.3.1 Category Usage

To understand the overall performance of the agents, we compare their average scores in each category as a percentage of the DP-optimal score for that category in figures 20 and 21.

The primary differentiators between our RL agents and the DP optimal solution are in the upper section (and bonus), four-of-a-kind and Yahtzee. Performance in the upper section is critical to achieving a high overall score, as it enables the bonus however it appears to be traded off against the four-of-a-kind; it seems the agents prefer to take the immediate points from the 5th die rather than place themselves in a safer position to earn the bonus later.

The high performance for many agents on Yahtzee category was interesting, as it requires agents to be performing at a competent level across multiple turns. The interesting takeaway here is that agents appear to exhibit a "mode shift" in their strategy once they figure out Yahtzee (as shown in Figure 12), whereas the bonus is learned

more gradually over time (as shown in Figure **??**).

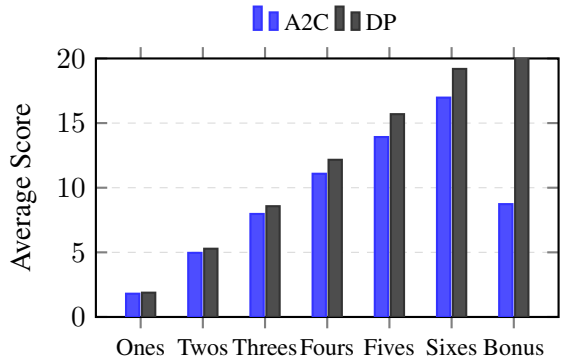

Figure 20: Upper section and bonus scores (placeholder data)

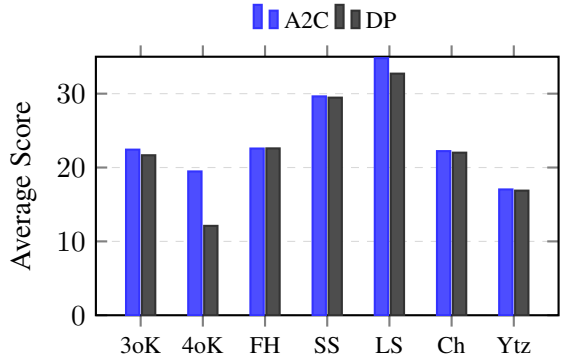

Figure 21: Lower section scores (placeholder data)

5.3.2 Strategy Comparison Across Agents

We also compared some high-level strategy metrics across our different agents to understand how their learned policies differed. First, we analyzed the distribution of the first category choice made by each agent in Figure 22. Additionally, Table 5 shows the top three most frequently used categories at each turn of the game, providing insights into the agent's strategy throughout a game.

6 Discussion

6.1 Summary

In this work, we attempted to use policy-gradient reinforcement learning methods to teach agents to play *Yahtzee* through self-play. We found that with appropriate algorithmic and architectural choices, it is possible to approach near-optimal performance. Advantage Actor-Critic (A2C) with TD(0) was consistently stable and efficient; REINFORCE and PPO were more fragile and underperformed at equal training budgets. Our best A2C agent, trained over 1 million games, achieved a

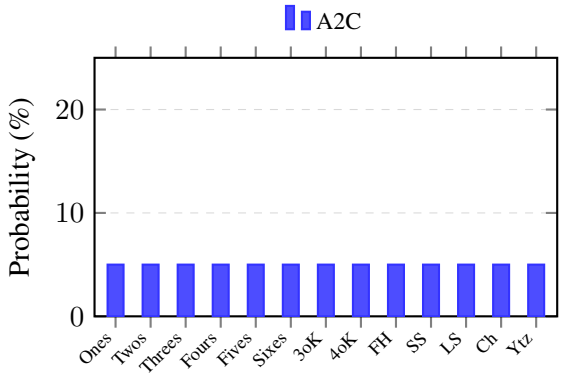

Figure 22: First category chosen distribution (placeholder data)

median score of 241.78 points over 100,000 evaluation games, which is within 5.0% of the DP-optimal score of 254.59. In games played, it achieved Yahtzee at an optimal rate (34.1%), and had a good upper bonus achievement rate (24.9%). These results are best reported in Table **??**.

We found that single-turn REINFORCE gets surprisingly high scores (even with a smaller model), often outperforming full-game REINFORCE agents, but fails to learn a coherent bonus strategy. We observed a tradeoff between single-turn and full-game performance, especially at higher performance levels, as seen in Figure **??**, and found that with certain hyperparameters, we actually improved full-game performance while decreasing single-turn performance, indicating an interesting Pareto frontier and risk for target leakage.

Our ablation studies highlighted several design choices that significantly impacted final performance. The choice of RL algorithm and credit assignment was crucial; A2C with TD(0) outperformed both PPO and REINFORCE in terms of stability and final score, as shown in Table **??**. As seen in Table **??**, the state and action encodings played a significant role; specifically providing (rather than forcing the network to learn) some easily calculable features improved learning. Categorical action distributions outperformed Gaussian ones, as seen in Table **??**, allowing the network to better model compound actions. The choice of activation function also mattered; with ¡BLAH outperforming BLAH¿, likely due to better gradient flow. Adding LayerNorm layers improved stability and final performance as well, as shown in Table **??**. Lastly, we found diminishing returns for model size; larger models improved

Table 5: Top 3 most frequently used categories by turn (A2C, 1M games, 100K evaluation games)

Turn	Category	Usage %	Median Score
1	Small Straight	17.0%	30.0
	Full House	12.3%	25.0
	Large Straight	11.8%	40.0
2	Small Straight	15.1%	30.0
	Full House	11.2%	25.0
	Large Straight	10.5%	40.0
3	Small Straight	12.8%	30.0
	Full House	10.4%	25.0
	Large Straight	9.3%	40.0
4	Small Straight	10.8%	30.0
	Full House	9.9%	25.0
	Fours	8.9%	12.0
5	Small Straight	9.1%	30.0
	Full House	9.0%	25.0
	Fours	8.8%	12.0
6	Twos	9.3%	5.0
	Fours	9.1%	12.0
	Threes	9.0%	9.0
7	Twos	10.3%	4.0
	Ones	9.3%	1.0
	Threes	9.2%	9.0
8	Ones	11.2%	1.0
	Twos	11.0%	4.0
	Three of a Kind	9.7%	23.0
9	Ones	12.8%	1.0
	Twos	11.0%	4.0
	Three of a Kind	9.4%	23.0
10	Ones	13.9%	1.0
	Twos	10.2%	4.0
	Chance	9.4%	22.0
11	Ones	13.8%	1.0
	Yahtzee	13.0%	0.0
	Twos	9.3%	4.0
12	Yahtzee	20.5%	0.0
	Ones	10.9%	2.0
	Four of a Kind	8.0%	6.0
13	Yahtzee	26.6%	0.0
	Large Straight	13.4%	0.0
	Four of a Kind	11.7%	0.0

performance up to a point, but after a certain size, gains were minimal, as shown in Figure **??**.

When moved from single-turn to full-game play, REINFORCE immediately began struggling with the variance and credit assignment challenges inherent in the full-game setting; in fact, it was extremely difficult to even get back to the same level of performance as the single-turn agent, which was a surprise. Performance immediately improved when we switched to TD(0) returns and A2C, after adjusting certain hyperparameters (notably the critic coefficient). When A2C still struggled with the upper bonus strategy, we implemented GAE returns, However, they did not lead to significant improvement, as shown in Table **??**; moderate values of GAE were slightly helpful, but high values led back to instability. The underlying issue here is that most of Yahtzee's reward is local, the bonus is a small, rare, and delayed reward that is difficult to assign credit for without accepting high variance.

Entropy regularization played a huge role in stabilizing training and encouraging exploration, best seen in Figure **??**; striking the right balance of explore vs exploit was critical. There's a narrow sweet spot: too little entropy and the policy collapses, does not explore and we get stuck in local minima; too much entropy and the policy becomes too random to learn strategies from signals that require high-coherence.

There were some limitations to our approach. First, we fixed the training budget to 1 million games for all agents, but could only explore the hyperparameter space using 250,000 games per run. Likely, increased batching, parallelism, or more performant simulation would allow larger

budgets in the same time frame, improving final performance. Second, we typically ran only a single seed per configuration due to compute constraints, so it's possible that some results were affected by random chance. Third, we only explored a limited set of architectures and hyperparameters; more extensive sweeps, especially around PPO, could yield better performance. We also did not explore transfer learning from single-turn to full-game agents, which could improve sample efficiency.

6.2 Strategy & Failure Modes

From the per-turn statistics in Figure **??**, we can infer a typical game, which begins with the agent locking in straights or full houses. These are fixed, high-value categories that provide a strong foundation. The agent also prioritizes the 4-of-a-kind category early on, typically on the 6th turn. The agent then turns its attention to the upper section, often scoring in the 4's, 5's, 6's, sometimes taking 3-of-a-kind instead. However, it seems to prefer taking a lower-section score for 5's and 6's rather than using them to build towards the bonus. The agent seems to understand the importance of reaching the 63-point threshold for the bonus, but perhaps does not prioritize it early enough in the game. We also see Ones, Twos, and Chance being used as "dump" categories later in the game when no better options are available. The agent does smartly hold these open.

It could be that the agent is risk-averse, preferring the immediate points from lower-section categories, or it simply doesn't properly realize it needs to take at least one above-average score in each of the 4's, 5's, and 6's to offset lower scores in the 1's, 2's, and 3's later in the game. This is the core issue for the agent; it prefers 4-of-a-kind early in training, and has difficulty learning when to pivot to bonus-seeking behavior later on. We see evidence of this when comparing the category mean scores against DP in Table **??**. This is the best evidence of the long-horizon credit-assignment challenge in Yahtzee; the agent needs to learn to sacrifice the marginal points from the 5th dice in 4-of-a-kind, in order to keep itself in a better position to earn the bonus later.

The agent does surprisingly well in achieving Yahtzee, indicating it has figured out how to lock in on selecting pairs, triples, and quads when the opportunity arises. It would be worth investigating further if the agent is explicitly targeting Yahtzee or if it's a byproduct of its general strategy, and if the agent recognizes that even 1's and 2's can be worth a lot, if they score a Yahtzee.

Reward shaping definitely helped the agent learn the bonus more quickly, but had its limits, as seen in Table **??**. We pushed the potential approach to the limit of it becoming a separate objective, but the linear bonus curve during the learning curve indicated the same underlying challenge remained. At very high levels, the agent completely derailed and failed to prioritize the lower section, most notably ignoring Yahtzees. It's worth restating that, since we are using a learned potential function, we do break the theoretical guarantees of potential-based reward shaping (**?**), this could explained the mixed results, and perhaps a manually designed potential function would work better.

In general our results backed up many known challenges in reinforcement learning, in that deep RL methods struggle with long-horizons, credit assignment, variance, and balanced exploration. Models are typically very quick to learn local strategies that yield immediate rewards; for example, our agents quickly snapped to the DP-optimal scores for full-house, straights, and even Yahtzee. However, they systematically under optimized the only part of Yahtzee that actually requires planning over the entire game: the upper section bonus. Yahtzee exposes this dynamic in a compact setting, showing it's value as a benchmark for RL research.

7 Conclusion and Future Work

Learning a robust policy for *Yahtzee* using reinforcement learning presents several interesting challenges and insights. We showed that with appropriate algorithmic choices, it is possible to approach near-optimal performance using self-play alone. First, we showed that *Yahtzee*'s combinatorial action space and sparse rewards make it suitable as a non-trivial benchmark environment. We found that Yahtzee is trivially broken into several a heirarchy of interesting sub-problems. 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, more efficiently. 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. 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. For the multiplayer setting, future work could explore permutation-invariant architectures such as Deep Sets (Zaheer et al., 2018) or embeddings with self-attention to handle unsorted dice (Vaswani et al., 2017). Additionally, Yahtzee could also be considered as a candidate environment for research into hierarchical reinforcement learning methods (Barto and Mahadevan, 2003).

References

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

Zafarali Ahmed, Nicolas Le Roux, Mohammad Norouzi, and Dale Schuurmans. 2019. Understanding the impact of entropy on policy optimization. In *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 151–160. PMLR.

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

Andrew G. Barto and Sridhar Mahadevan. 2003. Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems*, 13(4):341–379.

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.

Lukas Biewald. 2020. Experiment tracking with weights and biases.

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.

Stefan Elfwing, Eiji Uchibe, and Kenji Doya. 2017. Sigmoid-weighted linear units for neural network function approximation in reinforcement learning.

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)*.

William A. Falcon. 2019. PyTorch Lightning. *GitHub repository*.

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).

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.

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.

Yuxing Liu, Yuze Ge, Rui Pan, An Kang, and Tong Zhang. 2025. Theoretical analysis on how learning rate warmup accelerates convergence. *arXiv preprint arXiv:2509.07972*.

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*.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*.

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.

Nitish Srivastava, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958.

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.

Arryon Tijsma, Madalina M. Drugan, and Marco A. Wiering. 2016. Comparing exploration strategies for Q-learning in random stochastic mazes. In *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–8. IEEE.

Philip Vasseur. 2019. Using deep q-learning to compare strategy ladders of yahtzee.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. In *Advances in Neural Information Processing Systems*, volume 30.

Tom Verhoeff. 1999. Optimal solitaire yahtzee strategies.

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.

Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Ruslan Salakhutdinov, and Alexander Smola. 2018. Deep sets.

A Reproducibility

The custom gym environment, training code, models, and final evaluation statistics for this project are available on GitHub.

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

B Hyperparameters

The following hyperparameters were used for the baseline models:

Table 6: 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 7: 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 8: 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**.

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:

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

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

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

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 $r' = 0,\ t' = t + 1$.