Marcus Bluestone

Reinforcement Learning
6.7920, Fall 2024
Interpretable RL Policies for Playing Blackjack

Due: December 11, 2024

1 Abstract

We explore the trade-offs between qualitative interpretability and quantitative performance measures in the context of Reinforcement Learning agents for Blackjack. We construct interpretable RL policies by obfuscating the observations for the agent and by simplifying their underlying policy network architectures. These models are then compared to a variety of baselines (Dynamic Programming, Basic Blackjack Strategies, and Random Policies). Our results indicate high performance from a range of different models, and also provide some insights into Blackjack strategy that can be used in real-world settings.

Code Link: https://github.com/MarcusBluestone/Blackjack

2 Introduction

2.1 Deep RL and Interpretable Strategies

Deep Reinforcement Learning methods have transformed the landscape of RL research. By parametrizing value and/or policy functions as deep Neural Network, models can achieve astoundingly high performances on a variety of tasks, ranging from Atari games to robotic motion planning to schedule routing.

However, one significant downside for creating RL agents whose decisions are parametrized by deep neural networks is that they are extremely difficult to interpret. In the context of RL, interpretability refers to the extent to which a human can understand the rationale behind an agent's decisions and behaviors. In the context of this paper, we define the interpretability of a policy to be more specifically the qualitative measure of a human's ability to implement the strategy in a real world game. This paper, in part, hopes to demonstrate the ability of RL algorithms to actually influence and change human strategy.

2.2 Performance vs. Interpretability

In this paper, we analyze the performance of a variety of RL-based agents in the context of the game Blackjack. This game was chosen particularly because it has already been "solved," i.e. the optimal decision at any stage in the game can be efficiently calculated via Dynamic Programming. This gives us an upper bound on the expected performance of any RL agent and allows us to rank the performance of RL agents in comparison to the DP solution.

More specifically, we explore the balance between quantitative measures of performance (as Average Expected Returns) and qualitative measures of interpretability for a wide array of Blackjack strategies. For instance, a Dynamic Programming solution to Blackjack will achieve an upper bound on performance, but is highly uninterpretable, as human players will struggle to implement this easily in the real world. On the other

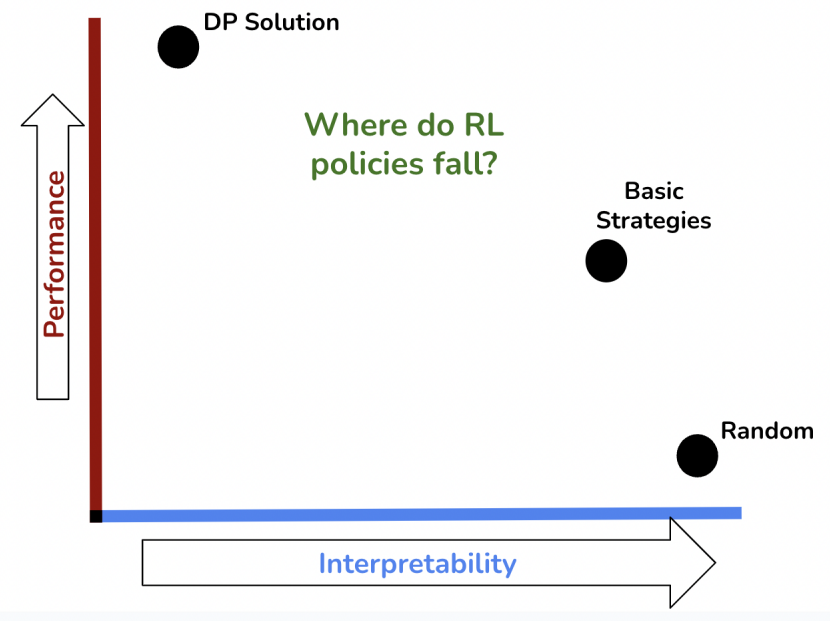


Figure 1: The trade-off between performance and interpretability for Blackjack strategies. For instance, a Dynamic Programming solution will have the highest performance, but will be extremely difficult to implement in-game. On the extreme, randomly choosing a move on each turn is easy to implement, but achieves low results.

extreme, randomly choosing an action at each turn is easy to implement but will achieve low results. As a middle ground, basic Blackjack strategies achieve solid performances and can range in their complexity to be implemented. See Figure 1.

To this end, we experiment along two dimensions – the observation space representations that the RL agents are exposed to and the policy network architecture that determines the RL agents' actions. By obfuscating parts of the state and simplifying the underlying actor-network parametrization, we arrive at interpretable, but highly effective, policies that outperform traditional basic Blackjack strategies.

We explore three main questions:

1. Can a general RL agent learn how to play Blackjack optimally?
2. How does changing the observation space affect the performance and interpretability of RL policies?
3. How does simplifying the parametrization of the policy network affect the performance and interpretability of RL policies?

2.3 Blackjack Rules

Blackjack is the most popular casino card game in the world. In this section, we present a simplified version of the game and note where appropriate other variants of the game.

First, a 52-card deck is shuffled (note many versions have up to 8 shuffled decks). To play, the house/dealer deals 2 cards to the player, along with 2 cards for themselves (note many versions have more than one player). Only one of the house's cards is visible to the player and the other is not.

The numerical cards are valued at their corresponding number, face cards are all valued at 10, and aces can count as either 1 or 11 (whichever benefits the player more). On each turn, players can decide to hit or stand. If they hit, another card is drawn from the deck and added to their running hand. If they stand, then the player does nothing and the dealer draws cards for themselves until they reach at least 17, at which point

they stop drawing (note that traditional Blackjack includes other actions like surrender, split, and double down).

If the player begins with a Blackjack, then they win 1.5 times their original bet. If a score of 21 is reached subsequently by either player or dealer, the game is over and that person wins. If either person goes over 21, they lose. Otherwise, the winner is the one with the score closest to 21, and they win the full bet.

We also assume that the deck is shuffled after each round (note: this reduces a player's ability to "count cards" and gives the casino the largest possible advantage. See 2.4).

2.4 Traditional Blackjack Strategies

Blackjack is referred to as a "solved" game. In this section, we explore a few basic blackjack strategies to act as baselines for our experiments and to provide some insights into card counting strategies.

1. **Dynamic Programming**: Because the valid state space of the game is relatively small (see Section 3.2), it can actually be solved efficiently via Dynamic Programming. This strategy achieves an upper bound on performance, but is extremely difficult for human's to implement in real-world settings.
2. **Random**: At each time step, the player chooses between HIT and STAND randomly. This strategy achieves extremely low performance, and is tested only as a worst-case baseline.
3. **Strat17**: In this strategy, a player hits until they get to the score of 17 or more, at which point they stand. This essentially mimics the dealer's strategy and achieves very solid performance metrics. This strategy is highly interpretable and very simple, and we hope that our RL policies can surpass this strategy in performance.

The most popular and effective blackjack strategies involve some form of card counting. Card counting involves keeping track of all cards that have been played so far, and analyzing the subsequent probabilities of drawing certain cards from the deck. See Figure 2 for example strategies. Players assign values to each type of card (e.g. in the first row, 2-6 have value 1, 7-9 have value 0, and face cards have value -1) and keep track of the total running total of these values for all values that have been played so far. Higher total scores favor the player and imply that he should bet more, while lower scores favor the dealer. Intuitively, being dealt face cards significantly increases the likelihood of busting and are thus deemed bad for the player.

When interpreting our results in Section 6.1, we will compare the groupings in these traditional card counting strategies to our experiments and discuss similarities and differences.

Card Strategy	2	3	4	5	6	7	8	9	10, J, Q, K	A	Level of count
Hi-Lo	+1	+1	+1	+1	+1	0	0	0	−1	−1	1
Hi-Opt I	0	+1	+1	+1	+1	0	0	0	−1	0	1
Hi-Opt II	+1	+1	+2	+2	+1	+1	0	0	−2	0	2
KO	+1	+1	+1	+1	+1	+1	0	0	−1	−1	1
Omega II	+1	+1	+2	+2	+2	+1	0	−1	−2	0	2
Red 7	+1	+1	+1	+1	+1	0 or +1	0	0	−1	−1	1
Halves	+0.5	+1	+1	+1.5	+1	+0.5	0	-0.5	−1	−1	3
Zen Count	+1	+1	+2	+2	+2	+1	0	0	−2	−1	2
10 Count	+1	+1	+1	+1	+1	+1	+1	+1	−2	+1	2

Figure 2: A table of different card counting strategies, where each row is a different strategy. Players keep track of the sums of card scores; higher scores tell the player that they are in a better position.

3 POMPD Formulation

We formulate the game of Blackjack as a finite Partially-Observable Markov Decision Process (POMDP), represented by the 6-tuple (S, A, T, R, Ω, O) where: S is the state space, A is the action space, T is the conditional probabilities between states, $R : S \times A \to \mathbb{R}$ is the rewards function, Ω is the observation space, and $O(o|s', a)$ is the conditional observation probabilities.

3.1 POMDP Basics

We encode the player and dealer hands as the current state of the game. The action space is {HIT, STAND}. Transition probabilities are defined according to the Blackjack rules and the remaining cards in the deck. Rewards are always 0, except for a terminal reward of $1.5, 1, 0$, or -1 depending on who wins/loses (see 2.3).

We define three different observation spaces – True, Values, and High-Low. These observations are what the RL agent are actually exposed to.

3.2 Observation Spaces

We begin by modeling the entire true state of the game, and then we obfuscate certain aspects of the game (suits, some of the values) to create observation state spaces. We argue whether or not each of these representations possesses the Markov Property, discuss the feasibility of the state space size, and also whether they are fully representative of the game.

These formulations intentionally strike a balance between highly descriptive, larger state spaces and less descriptive, smaller state spaces. The former benefits from the amount of information the state carries, though its detailed representation may lead to over-fitting, slow inference speeds, and possibly slower learning times if the representations are too complex too learn. On the other hand, the latter benefits from efficient inferences and quicker learning times, though it may lack enough information about the true state for the agent to learn a good policy. See Figure 3 and Table 1 for overviews of each representation.

3.2.1 True State

The most obvious approach to represent blackjack as an MDP is to represent the state as a length 52 array corresponding to each card in the deck, where each element can be either 0 (no one has picked it yet), 1 (the player has picked it), or 2 (the dealer has picked).

If we one-hot encode these 3 options, we get a vector size of length 156. Although this formulation has the Markov Property (i.e. adding information from previous states won't affect current state transitions or rewards probabilities), this formulation has one tragic flaw. Because we take into account the suits of the cards, the state becomes combinatorially large (especially with more than one deck) and efficient Dynamic Programming approaches become infeasible.

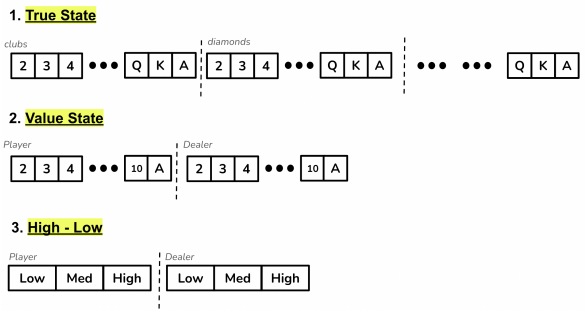


Figure 3: The encodings of each observation type

3.2.2 Values State

Instead, we can model Blackjack as a length 20 array corresponding to the possible **values** that could be picked throughout the game. The first 10 locations in the array correspond to number of cards of a specific values that the player picked, while the last 10 locations correspond to the dealer's values. The possible values are 1...10. (note: the 1 value corresponds to aces, and the case where we want to count aces as 11's is handled explicitly in the rewards/transitions).

We can encode the counts of each value either as one-hots (what we refer to as 'values' representation and would have vector size $20 * 5 = 1000$) or as a numerical value (what we refer to as 'values2' and would have vector size 20).

For instance, imagine that at some point, we have: Player - Jack (Hearts), Ace (Clubs), Queen (Spades). Dealer - 3 (Hearts). This would be encoded using values2 as:

$$[1, 0, 0, 0, 0, 0, 0, 0, 0, 2; 0, 0, 1, 0, 0, 0, 0, 0, 0, 0]$$

This state representation still possesses the Markov Property even though suits are ignored since only the *values* of past cards will affect future transitions probabilities. In addition, the state space of all possible valid hands is relatively small under this representation. In fact, this state space representation works well even if we increase the number of decks (as is common practice in many casinos) or choose a different "blackjack" target value instead of 21 (See Figure 4). This small state space size allows us to efficiently recurse through all possibles games via Dynamic Programming to achieve an optimal solution.

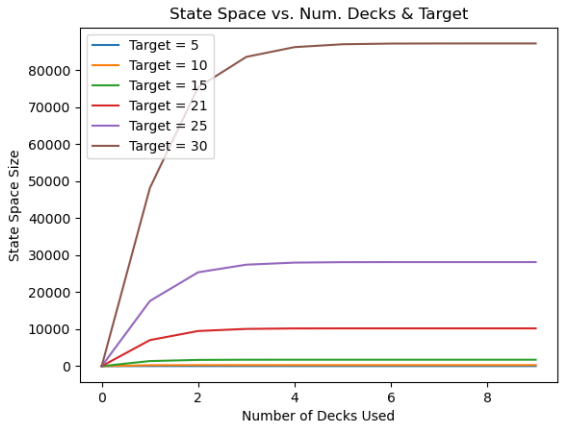


Figure 4: Example representation of Blackjack states using the Values State representation. Values calculated by enumerating all possibilities via recursion

3.2.3 High-Low State

As seen in section 2.4, even basic counting strategies – such as hitting until you have a sum of 17 or keeping track of the number high/low cards – can have highly successful results. We thus propose an *observational space* represented by a length-6 array where the first 3 elements correspond to the total number of player's low, medium, and high cards; and the last 3 elements corresponds to the total number of dealer low, medium, and high cards. Low, medium, and high are defined here by the Hi-Lo strategy in Figure 2. The counts are encoded as numerics, not one-hots.

This observation space representation still possesses the Markov Property, as the state at each time step includes all of the low/medium/high cards that have been played up until this point. The order of when these cards were played has no impact on a single player's strategy. However, this space does not fully capture the Blackjack game since we obfuscate some of the values, and thus we don't expect to achieve optimal performance using this approach.

Table 1: Comparison of the 3 different observation representations

	True	**Values**	**High-Low**
Vector Size	156	20 or 100	6
State Space Size	Massive	Reasonable	Small
Markovian?	Yes	Yes	Yes
Fully Representative?	Yes	Yes	No

3.3 Objective

Given the above representations, we seek an optimal policy such that:

$$V^*(o) = \max_{a \in A}[r(o,a) + \sum_{o' \in \Omega} Pr(o'|o,a)V^*(o')]$$

$$\pi^*(o) = \arg\max_{a \in A}[r(o,a) + \sum_{o' \in \Omega} Pr(o'|o,a)V^*(o')]$$

4 RL Policy Networks

We experimented with two types of policy networks. One (that we term "Deep") is the default in the StabeBaselines3 library and is composed of two fully connected layers each with hidden size = 64 and with *tanh* activation functions[1].

The other, is a custom policy network (that we term "linear") that combines a linear layer with a ReLU to create a linear threshold that decides whether to HIT or STAND at each time step. See Figure 5 for visual of the custom network.

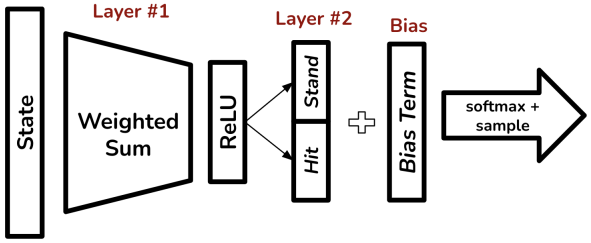


Figure 5: A visual depiction of the linear policy network. It combines a linear sum with a ReLU activation, followed by a small linear layer and bias term. Note that the output of Layer #1 is a scalar, while the output of Layer #2 is a 2-dimensional vector corresponding to hitting or standing

5 Training & Evaluations

To evaluate any given policy, we ran the policy on 300 randomly created games of blackjack and took the average payout of these games. We repeated this $1,000$ times and took the median score. These value were chosen through some experimentation and resulted in a ranking of the policies, where p-values were all well below the .05 threshold.

To train the RL policies, we used 750 rollouts, where 2048 gradient update steps occurred in each rollout. We evaluated the policy every 5 rollouts. We trained each policy three times using PPO and took the best-performing policy of the three trials.

[1] https://stable-baselines3.readthedocs.io/en/master/guide/custom_policy.html

We trained a total of 8 reinforcement learning (RL) agents, combining the four observation space representations (true, values, values2, and high-low) with two types of policy network architectures (linear and deep). We also evaluated three baseline policies – DP, Strat17, and Random. See Section 6.1 for the results.

We built a custom OpenAI Gym environment for Blackjack, and used stablebaselines3 to run PPO.

6 Results and Interpretation

6.1 Training

We present and analyze the results of training the RL agents with the parameters as discussed above in Figure 6. For each of the policies, we saved the model that performed best.

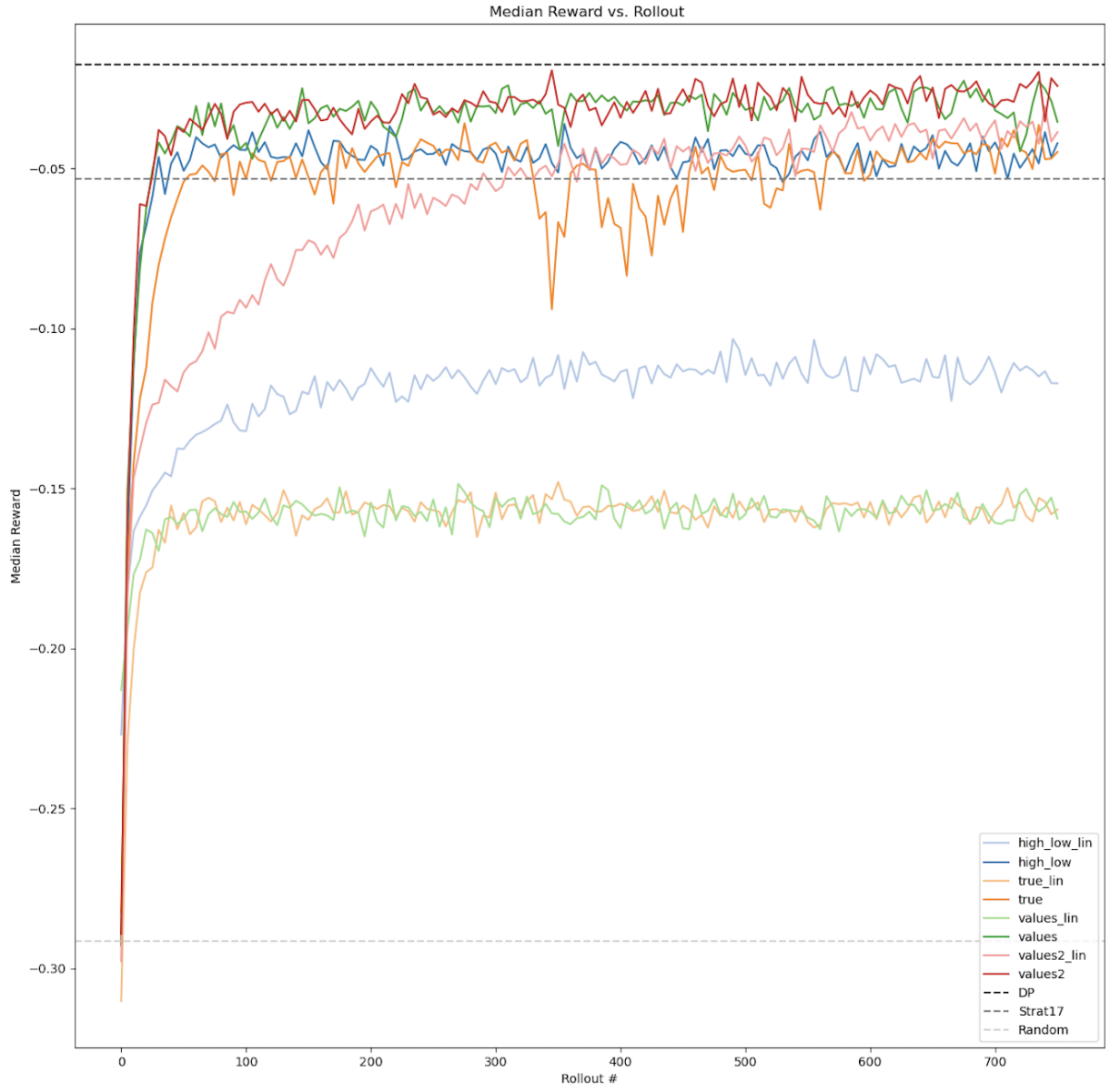


Figure 6: The results of training the 8 RL agents described above. The dotted lines represent the Baseline Policies (DP, Strat17, and Random).

We point out a few key observations:

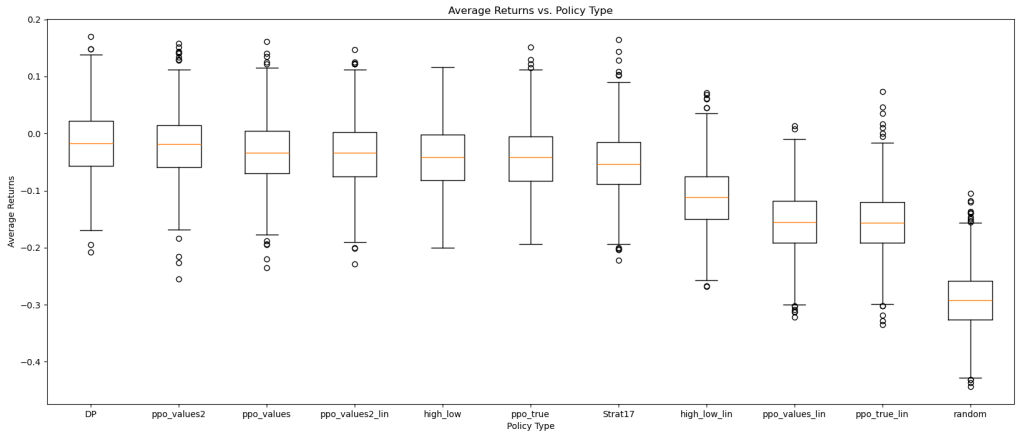
1. The policies are learned extremely quickly (usually within this first 200 rollouts).
2. All of the Deep-Architecture policies outperformed the Strat17 policy but were still upper-bounded by the DP baseline.
3. The greyed-lines represent the linear policies which significantly underperformed compared to Strat17, except for values2_lin which beat out Strat17 and will be analyzed in Section 6.3.

6.2 Evaluations

Next, we present and compare the performance and speeds of all of these models. Although this paper does not factor in evaluation speed as a metric for performance or interpretability, we provide our results for the sake of completion and robustness. See Figures 7 and 8.

Notice that even the DP optimal solution only has a median return of $\approx -1\%$, which is why the Casino always wins in expectation in Blackjack. Also note the values representations of the observation space achieves the best results, which makes sense since it is the most condensed form of information that is still fully representative of the entire game.

Of greatest importance for this paper is the linear policy with the values2 observation representation that achieves near-optimal performance that surpasses both the Strat17 benchmark and other policies with more sophisiticated policy architectures.



	DP	ppo_values2	ppo_values	ppo_values2_lin	high_low	ppo_true	Strat17	high_low_lin	ppo_values_lin	ppo_true_lin	random
count	1000.000000	1000.000000	1000.000000	1000.000000	1000.000000	1000.000000	1000.000000	1000.000000	1000.000000	1000.000000	1000.000000
mean	-0.016972	-0.021788	-0.032758	-0.036057	-0.041680	-0.043190	-0.052128	-0.112355	-0.155082	-0.155837	-0.290838
std	0.057630	0.058134	0.057066	0.056936	0.054619	0.057999	0.058163	0.058661	0.056046	0.057535	0.055213
min	-0.208333	-0.255000	-0.235000	-0.228333	-0.200000	-0.193333	-0.221667	-0.268333	-0.321667	-0.335000	-0.443333
25%	-0.056667	-0.058750	-0.070000	-0.075000	-0.081667	-0.083333	-0.088333	-0.150000	-0.191667	-0.191667	-0.326667
50%	-0.017500	-0.018333	-0.033333	-0.033333	-0.041667	-0.041667	-0.053333	-0.111667	-0.155000	-0.156667	-0.291667
75%	0.021667	0.015000	0.005000	0.002083	-0.001667	-0.005000	-0.015000	-0.075000	-0.118333	-0.120000	-0.258333
max	0.170000	0.158333	0.161667	0.146667	0.116667	0.151667	0.165000	0.071667	0.013333	0.073333	-0.105000

Figure 7: The results of evaluating all of our strategies ordered from best to worst. Notice that the only linear policy to pass the Strat17 benchmark is values2_lin

6.3 Weight Inspection of Values2_lin

Because the RL strategy values2_lin achieved high results, we suspect that it is learning a high-performance, interpretable strategy. As such, we seek to understand its strategy on an intuitive level. To do so, we analyze the weights in each of it's layers.

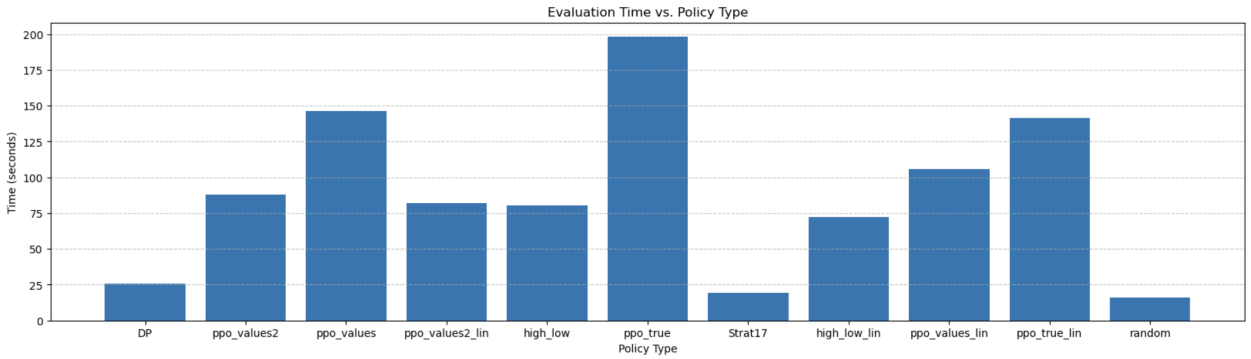


Figure 8: The time it took in seconds to evaluate each of these different strategies.

To review, the linear policy architecture first performs a linear sum of the state values and outputs a single scalar. That value is then passed through a ReLU, which then multiplies a 2×1 weight vector, and then a bias term is added to it (see Figure 5). This means that if the initial weighted sum is negative, it will be zeroed by the ReLU, and the bias term will dominate. Otherwise, the bigger the sum, the more the weights in the second layer will dominate the output. After softmaxing the 2×1 output vector, the first term of the output corresponds to the probability of hitting and the second term corresponds to the probability of standing.

After inspecting the actual weights, we see that the bias term is $[4.803, -3.969]$ and the weights for the second layer are $[-6.2179, 5.997]$. This means that the bias term favors hitting, while the second layer favors standing. For instance, if the weighted sum is negative, the output will only be the bias term, which after softmaxing gives a 99.98% chance of hitting and a .02% chance of standing.

Now, we inspect the weights in the first layer. See Figure 9 below, which shows the value of the weights corresponding to each value in the state.

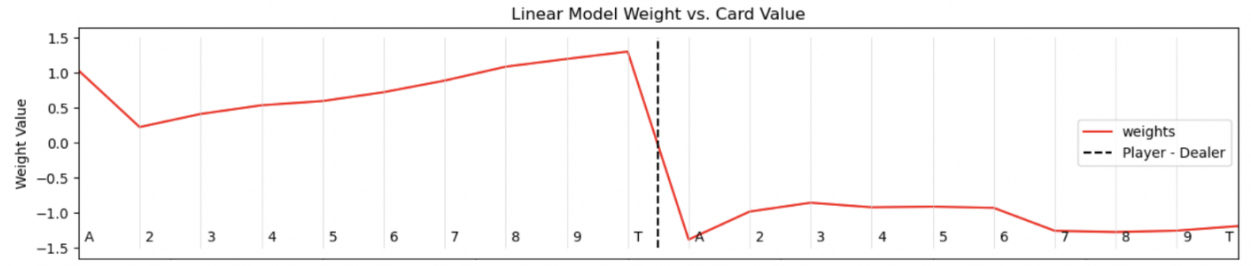


Figure 9: The value of the weights in the first layer of the value2_lin policy network as a function of the index.

There are a few key details to note here:

1. **Linear Trend for Player**: The weights for the player demonstrate an increasing linear trend with respect to the card value. This is not necessarily obvious, as the likelihood to hit may not necessarily be proportional to card value since other factors, like card frequency (many more cards of value Ten than other cards) may play a role.
2. **Player Ace $\approx$ 7**: What is most fascinating is that the weight given to the player having an Ace is approximately equal to the weight associated with a 7. This actually goes against a lot of card counting strategies seen in Figure 2, where an Ace is usually grouped together with the face cards.
3. **Non-Linear Trend for Dealer**: For the dealer's cards, values $2 - 6$ have approximately the same

weight, implying the strategy employed by the player should be about the same in any of those cases. Moreover, the values $7 - T$ are in another group. And the *Ace* has the most negative weight, implying that seeing a dealer's Ace is the greatest indication that the player should stand. These groupings are certainly non-obvious, as one might expect some type of linear trends.

By inspecting the weights of this RL model, we have learned a great deal about real-world, practical, and interpretable Blackjack strategies. The linear trend present in player card values, the fact that a player's Ace is probabilistically equivalent to having a 7, and the grouping patterns present with the dealer's hand, all can be implemented in the real world.

Certainly, further testing in this area is required before the results can truly be turned into an explicit player blackjack strategy. See Section 7 for elaboration on this topic.

7 Conclusion & Future Work

This paper tackles the problem of finding RL policies for Blackjack that have high performance and are interpretable for human players. By experimenting with different RL model parameters – such as the observation space and the underlying policy architecture – we were able to gain insights into new human-playable strategies for playing Blackjack.

However, there are still many future areas for further exploration and improvements.

1. **Increase # Players**: Card-counting strategies because much more effective, as the number of players increase. This is because more insights can be gained about what future cards could be picked. Thus, one would expect RL methods to do much better in these situations as well. It would be extremely interesting to study how the performance and interpretability of these RL methods improve/evolve as the number of players increase.
2. **Experiment with Different Blackjack Versions**: Our model in this paper assumes a simplified version of Blackjack. However, a more robust or sophisticated paper would also tackle Blackjack with more actions (split, double down, surrender), increased number of decks (up to 8), or even additional rules about how rewards or shuffling work. See the Wikipedia page linked in Section 2.3.
3. **Explore Methods of Interpretability**: The only method we use in this paper to interpret our policies is to study the underlying weights following training. Researching and applying what other methods exist for these types of problem could be quite interesting.
4. **Traditional vs. RL Blackjack Strategies**: Much more work needs to be done to explore and understand the relationship between traditional effective strategies in Blackjack and the results found by the RL agents in this study. We mention some interesting inconsistencies between the two in 6.3, but much more research and testing into both traditional Blackjack strategies and our results is required.