STAT 4830: Poker Zero Final Report

Problem Definition

The goal of this project is to develop a reasoning model for No-Limit Hold'em poker capable of decision making under incomplete information and adversarial conditions. The goal is to maximize expected value (EV) by refining strategic and adaptive play based on opponents' behavior. Success in poker extends beyond mere monetary gain: it highlights progress in reasoning under incomplete information, adversarial settings, and strategic adaptation, with applications in game theory, economic modeling, and negotiation. We use the following as success criteria: win rate (hands won, stack size, profit over time) and performance against Game Theory Optimal (GTO) strategies.

Traditional poker solvers aim to play GTO strategies, which are theoretically unexploitable. However, in practice, computing Nash equilibria in multi-player zero-sum games is very difficult. It is also often limited to simplified scenarios, such as reduced bet sizes, since poker has a very large game tree. Moreover, GTO-based bots cannot adapt when opponents are playing suboptimally. These limitations open the door for ML based approaches. The state of the art poker bot is Plurbius, which uses a combination of self-play and Monte Carlo Counterfactual Regret Minimization (CFR) to converge close to Nash Equilibrium. However, CFR-based methods require significant computation. Thus, LLMs can provide an alternative of less resource consumption and be able to play exploitatively.

Poker is stochastic, with incomplete information and unpredictable outcomes. We model the problem as follows: $EV = \sum_{h \epsilon H} P(h) \times R(h)$, where P(h) is the probability of a hand outcome and R(h) is the corresponding reward. The goal is to maximize total expected winnings over all rounds r, over the following actions of bet, raise, fold, and call. Since poker is characterized as sequential decision making problems, it is well suited for RL. States are defined by the current game configuration, including the player's hole cards, the community cards on the board, prior betting history, and stack sizes. The reward is determined based on how effective the action is in the context of the current game state.

Optimization Methods

Reinforcement learning is a branch of machine learning in which an agent learns optimal behavior by continually interacting with an environment, choosing actions, receiving feedback as rewards or penalties, and refining its strategy to maximize cumulative reward over time. Poker is an ideal fit because it poses a sequential decision-making problem with long-term payoffs: our RL poker model treats every game configuration—hole cards, community cards, betting history, and stack sizes—as the state, considers moves such as betting, raising, folding, or calling (plus specific bet sizes) as the actions, and assigns rewards according to how advantageous those

actions prove. As the agent plays, these rewards drive ongoing learning, enabling it to improve hand-by-hand and session-by-session.

We incorporated reinforcement learning through policy gradient methods, which optimize policies by maximizing rewards. One of the most popular policy gradient methods is Proximal Policy Optimization (PPO) which tackles the exploration-exploitation trade-off by explicitly constraining how much the policy shifts at each update. For every action taken, it first computes the estimated advantage—the difference between the action's expected return Q(s,a) and the baseline value of the state V(s)—which tells us how much better or worse that move performs relative to average play from the same situation. Simultaneously, it forms a probability ratio $r(\theta) = \pi_{new}(a|s)/\pi_{old}(a|s)$, quantifying how the new policy changes the likelihood of selecting that action compared with the old one. The objective is as follows: $L^{CLIP}(\theta) = E_t[\min(\rho_t(\theta), \hat{A}_t; clip(\rho_t(\theta), 1-\epsilon, 1+\epsilon), \hat{A}_t)]$. PPO's objective multiplies this ratio by the advantage to capture the gain or loss from boosting or shrinking the action's probability, but critically it clips $r(\theta)$ to the interval [1−ε, 1+ε]. If the update would push the ratio outside this band, the clipped value is used instead, preventing overly large policy jumps while still rewarding advantageous actions. By weighting each advantage with a ratio kept close to 1, PPO ensures steady, conservative improvements that neither ignore promising explorations nor abandon the stability of the current strategy.

Building upon PPO is Group Relative Policy Optimization, a policy gradient method that is used by Deepseek. To calculate the estimated advantage, PPO computes an entire value model as the baseline. GRPO avoids this, and thus gets rid of many computations, by gathering a batch of "completions"—either futures of the current trajectory or shorter-horizon continuations such as the next few actions—and computing a simple group reward average over those samples. The group reward average is simply the average of all the completions' rewards. For each sampled completion, the algorithm then forms a group advantage by subtracting that average reward from the completion's own reward. The group advantage calculates how much better or worse the outcome is compared with its peers. These per-completion advantages replace the classic Q(s,a)−V(s) term in the loss, so there is no need to estimate V(s) with a dedicated network. Like PPO, GRPO still multiplies each advantage by the probability ratio $r(\theta) = \pi_{new}(a|s)/\pi_{old}(a|s)$, and it clips that ratio whenever it drifts outside [1−ε, 1+ε]. The clipping again caps the influence of any single update, ensuring that policy changes remain incremental even though the baseline is now the quickly computed group mean rather than a learned value estimate.

We selected GRPO for our poker agent because its design works with poker's multi-agent, long-horizon nature while slashing training costs. Poker is inherently non-stationary since each action shifts the other players' strategies in real time. GRPO's group-based baseline captures this interactive setting better than PPO's single-agent value network. By comparing each outcome to the *average* reward of its peers, GRPO supplies a relative advantage signal that already reflects what peers might do, stabilizing learning when several policies co-evolve. GRPO is also light on compute. Because the group average replaces a learned critic, we skip

an entire forward-and-backward pass, cut memory traffic roughly in half, and open the door to low-precision tricks like LoRA adapters in the Unsloth implementation. Finally, poker's payoff arrives only at showdown or tournament end. This is a delayed and sparse reward that doesn't work well with algorithms that use short-horizon value estimates. GRPO works on complete hand "completions," so it can also wait until the hand is over to judge a move.

Finally, we created reward functions for our RL model. The scoring system has many layers. The first layer judges the poker move itself: it gives full points when the reply is precisely "fold," "call," or an exact "raise <number>" that matches ground truth; half points when the model at least picks the right action word ("raise") and another half if its bet size is perfect or within 20% of the target; and zero when the reply lists multiple actions or is wrong. The second layer checks that the answer is written in either the single word "fold," "call," or "raise <number>." Anything with extra text or a malformed number earns nothing. Two additional layers focus on presentation. A small score also encourages perfect XML hygiene by rewarding cases where each tag appears exactly once and no stray text leaks beyond the closing </answer> tag.

Implementation

Because of our compute limitations, we relied heavily on Unsloth to fine tune our models. Unsloth is an open source python framework that allows for fast finetuning. In typical fine-tuning frameworks, backpropagation and forward passes are handled by general-purpose libraries (like PyTorch) that are optimized for flexibility rather than maximum speed. However, Unsloth rewrites key portions of the modeling code into specialized, highly optimized kernels using Triton. By manually deriving the backpropagation steps and implementing them in these low-level kernels, it cuts out much of the overhead that general-purpose implementations carry, allowing for faster computation.

Additionally, in terms of memory, many systems duplicate memory usage (for example, storing additional copies for gradient calculations or keeping separate caches for inference and training), which slows down processing and increases VRAM usage. On the other hand, Unsloth smartly removes this duplication by sharing memory spaces between the inference engine (vLLM) and the training process. It also employs efficient gradient checkpointing—offloading intermediate activations to system RAM asynchronously. This not only slashes the required VRAM but also minimizes latency by reducing data transfers, meaning the model can process larger batches or longer sequences without the overhead that slows down traditional approaches.

Finally, with standard approaches, the training and inference pipelines are distinct, which can lead to inefficiencies when switching contexts or reloading models. Unsloth mitigates this inefficiency by integrating fast inference directly into the training pipeline (using tools like vLLM), Unsloth allows for simultaneous fine-tuning and efficient generation. This avoids the additional overhead of moving data between separate processes or systems.

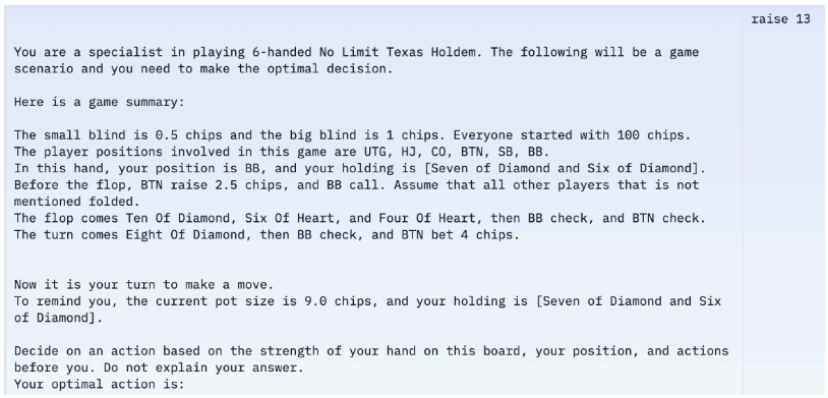
Overall, these key techniques Unsloth employs results in a smoother, faster, and more memory efficient fine tuning process.

In order to actually do the fine tuning, Unsloth relies on a method called LoRA (Low-Rank Adaptation). LoRA is a parameter efficient fine-tuning method, which means that it seeks to adapt pre-trained LLMs for downstream tasks by only updating a fraction of the model weights. In order to understand how LoRA works, we can examine a weight matrix $W_0 \in R^{d \times k}$ in a transformer layer. In full fine tuning, the entire weight matrix would be updated, leading to a total of $O(dk)$ parameters, which, with large models, can be computationally prohibitive due to the high memory and parameter overhead. LoRA mitigates this by freezing W_0 and introducing trainable low-rank decomposition matrices, significantly reducing the number of parameters to optimize. Thus, LoRA would model its adaptation as: $W = W_0 + \Delta W$ where $\Delta W = BA$ where $B \in R^{d \times r}$, $A \in R^{r \times k}$, with $r << min(d, k)$. Then, during fine tuning, the forward pass its modified as $h = W\_0\, x + BAx$, where BAx is the low rank adaptation. The loss function is modified as $f(B, A) = L(W_0 + BA)$, and is optimized by taking gradient steps only with respect to B and A. Thus, the number of trainable parameters is decreased from $O(dk)$ to $O(r(d + k))$. In practice, r is a hyperparameter chosen by the user, and for our training, we choose $r = 32$ on a 3 billion parameter model, and we end up with a parameter count of roughly 2% of the original model. When combined with Unsloth, we're able to run most of our experiments in just a Colab Notebook.

Fine-tuning and self play

For initial fine-tuning, we used GRPO to train our model on a dataset of 500,000 poker hands from the **PokerBench** dataset. Features were the bot's hole cards, any community cards, and any betting action that occurred, while labels were the optimal action for the bot. After training our model's initial weights, we implemented iterative self-play training for the bot to learn directly from real hands.

Example:

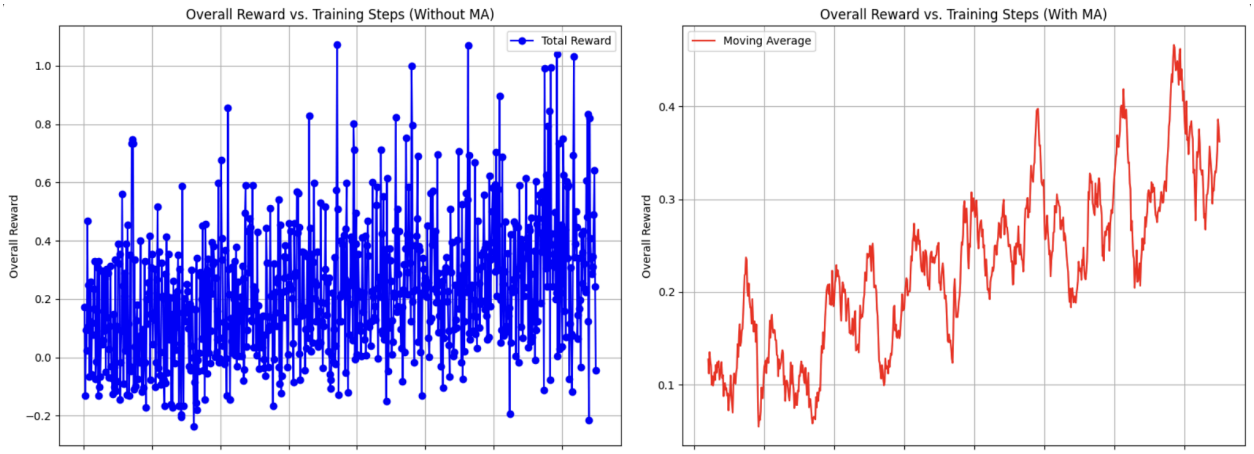


Reinforcement Learning/Self-Play using PyPokerEngine

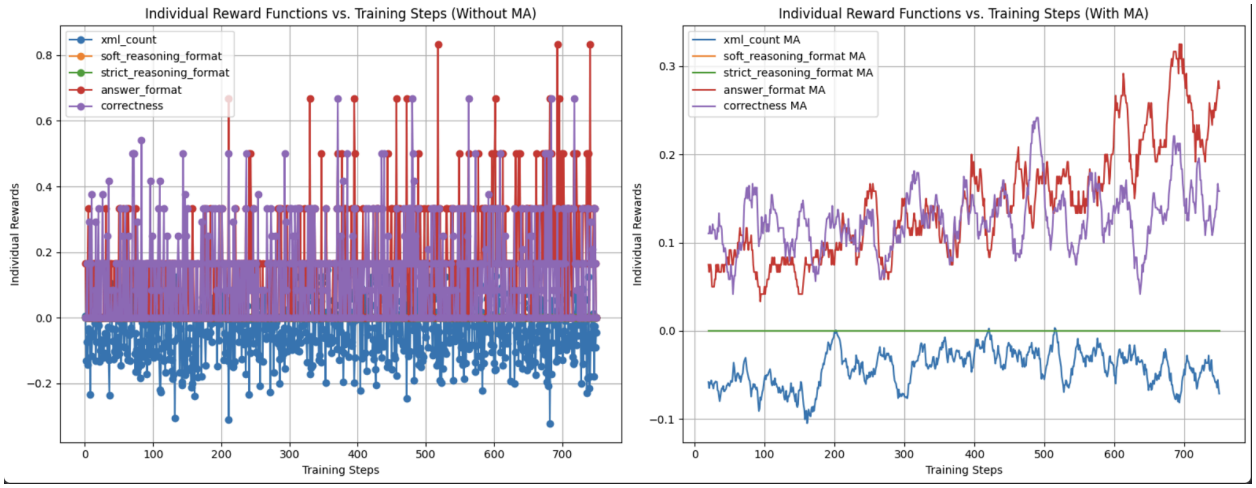
Our project used PyPokerEngine (from [pypokerengine.md]) to provide an easy to use poker environment to test our bots. PyPokerEngine allows us to create player instances that exist in a game environment, receiving game data and output moves. Our model (after some input parsing/cleaning) will read in the info and output a valid move. We can restore and manipulate game states, enabling exploration of different scenarios. We place our model in hands against earlier iterations of itself. After many simulated games, we used the output to generate additional training data for GRPO. We repeated this iterative process many times, recording performance metrics (profit and expected value) to evaluate the latest model against its predecessors.

The disadvantages of this approach were the incredibly long training times, which exacerbated the compute bottleneck we already had. It required 6 instances of our model to read inputs, conduct inferences, and provide outputs for each action, which meant we couldn't simulate as many games or perform as many iterations we wanted to. Furthermore, the randomness of poker meant we needed way more simulations to observe long-term results, since even positive EV actions could result in negative results in the short run.

Results

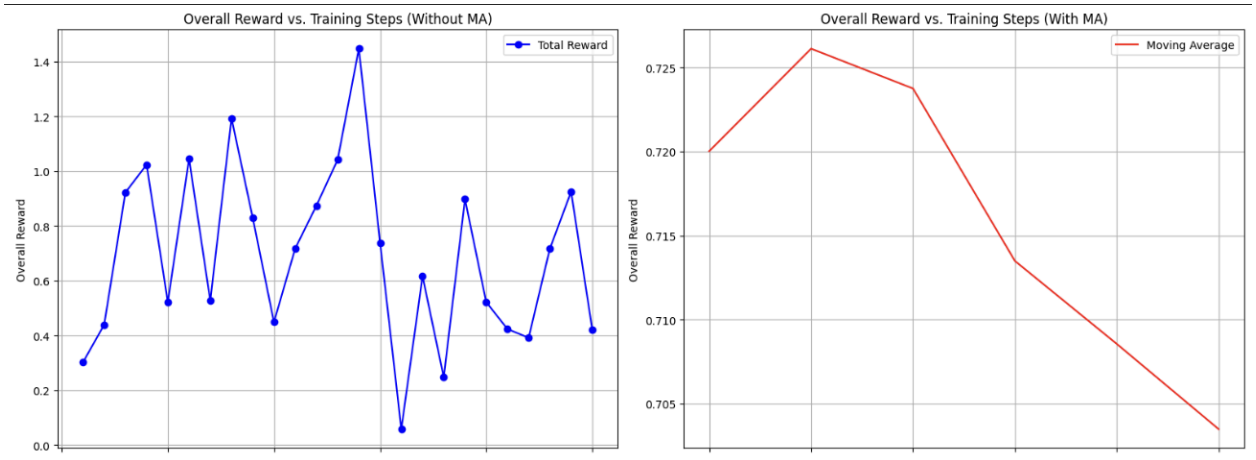


The graph above illustrates the overall reward the model received during training. The graph on the right is a trended version of the graph, with a window size of 20. Evidently, the overall reward that our model receives is steadily increasing, implying that our model is learning to play poker better.

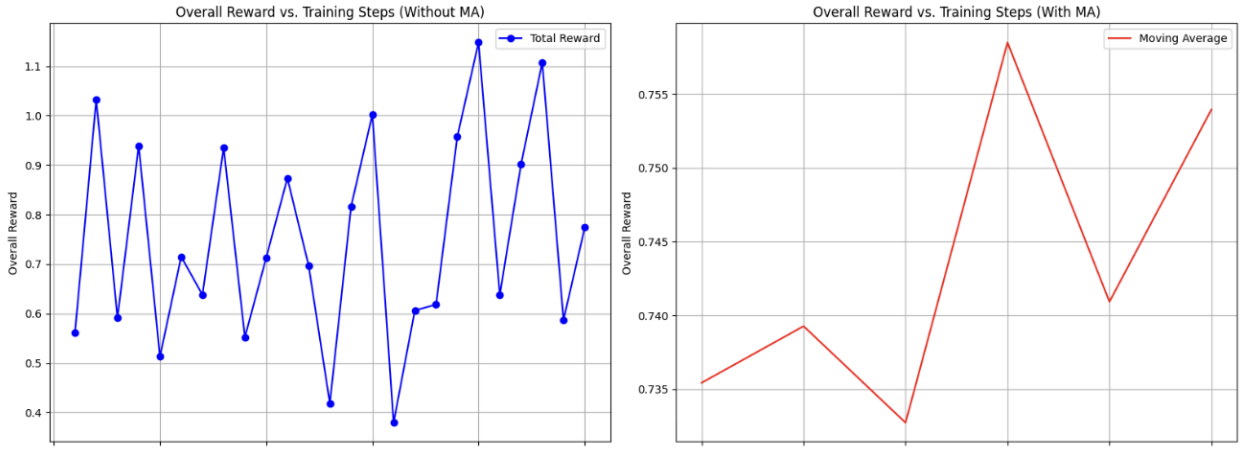


The two graphs above show a more granular breakdown of the different rewards received, each corresponding to a different reward function. From the graphs, it's evident that the model received the most rewards for answer format and correctness, which were two reward functions designed by us. In essence, the former rewards the model for outputting one word answers with a valid action while the latter rewards the model for the correct action and bet sizing. The xml count, soft reasoning format, and strict reasoning format rewards all hovered at or below zero. These were reward functions that came with unsloth's colab notebook and were thus not as relevant for our purposes.

Round 1 Self-Play

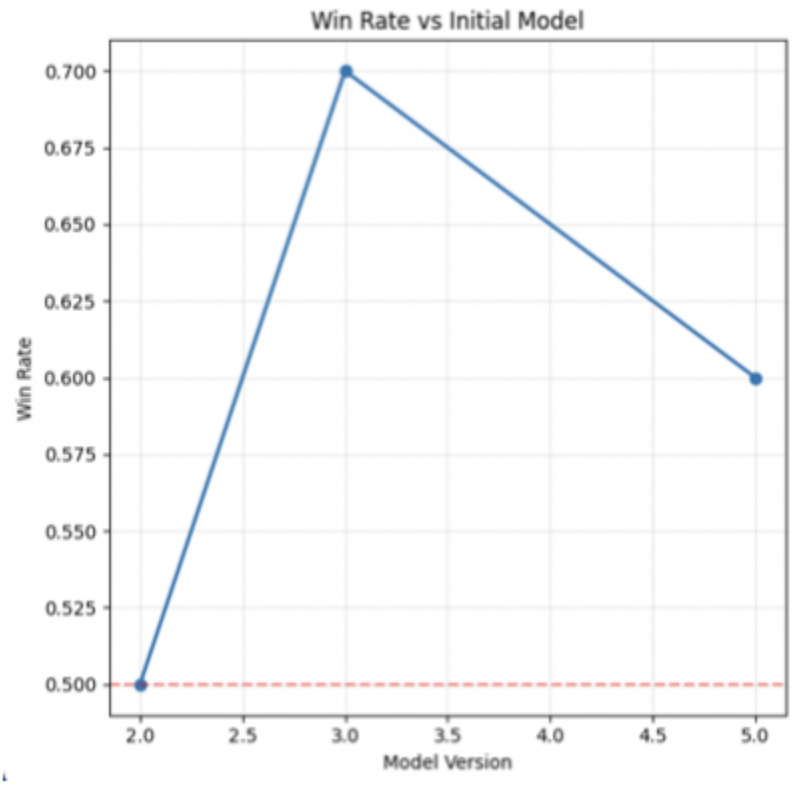


Round 5 Self-Play



The above graphs detail the rewards received by our model during self-play with 5 copies of our own model. Importantly, it's clear that by round 5, the overall reward being received is higher than the rewards being received during round 1, indicating a level of improvement as the game goes on.

We also had different versions of our model play themselves, and we can see that as the model version increases from 2.0, the overall win rate also gets above the baseline of 50%. However, there is still a little bit of fluctuation among the newer versions of the model, likely because we didn't run the model for many steps due to limited compute.



Another key component of our self-play pipeline was data generation, and after each hand, we took the game state and converted it into a datapoint that matched the PokerBench dataset shown above.

This allowed us to generate more data and have more training points the model could use for fine-tuning. Overall, our results indicate that the model is improving and finding rewards during fine-tuning, and further improving during self-play. This illustrates that there is significant potential for continued improvement through this pipeline.

Future Work

In the future, one thing we'd like to further explore is the idea of a good 'fold' during self play. In our current data generation pipeline, we only generate new data based on the winning player's action, which is usually a bet that induces everyone else to fold. This leads to our dataset being biased towards calling or raising during self-play training. However, there are times when folding is truly the right play in the long run, and we'd want to figure out a way to incorporate that into our self play data.

Additionally, another idea to explore is to play our model against humans or other bots as another way to evaluate performance, rather than just with self play.

Finally, with more time and compute, we'd like to train our models for longer, as for this project, we were very limited with the compute credits we had.