Population-based Poker AI

by

Jens Kaminsky
Alexander Liebendörfer

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Bachelor of Computer Science

Institute for Data Science
University of Applied Sciences and Arts
Northwestern Switzerland
2020

Jens Kaminsky

jens.kaminsky@students.fhnw.ch

Alexander Liebendörfer

alexander.liebendoerfer@students.fhnw.ch

2020

TABLE OF CONTENTS

ABSTRACT

Poker is a game requiring both handling imperfect information and risk management, and variants with more than 2 players are not optimally solved by approximating the Nash equilibrium. We train poker-playing artificial intelligence with a custom high-performance training environment by simulating large numbers of competing agents in a sports-like league, and investigate the performance and behavior of simple deep reinforcement learning architectures on different metrics. We find that deep Q-Learning outperforms Soft Actor-Critic, and that agent architecture choices matter more for broad behavior than the population of agents trained against. We also investigate the reliability of Microsoft TrueSkill as a metric of agent performance, and find its reliability depends strongly on agent behavior.

SECTION 1

Introduction

1.1 Poker Competition

The institute for Data Science of FHNW[1] ran a poker AI competition during the spring semester of 2020. As one of the three teams our goal was to write a poker-playing bot in a specified framework, which would be evaluated twice a week against those of the other teams at 6-player No-Limit Texas Hold'em Poker. This dissertation describes our approach, methods, internal evaluations, and insights during this competition.

1.2 Approaches

There exist many previous works on autonomously playing poker. Some authors model opponents statistically by assuming optimal play and inferring game state from opponent actions (e.g. [1] for a frequentist approach or [2] for a Bayesian approach), and others emulate players using supervised learning on databases of past games (e.g. [3]). However, the most successful approaches are based on counter-factual regret (e.g. DeepStack [4], Libratus [5], or Pluribus [6]). Pluribus is unique in that it is the first implementation (to our knowledge) that achieved superhuman performance specifically in 6-player Texas Hold'em, instead of only 2-player games.

Deep reinforcement learning has emerged as one of the strongest approaches for game AI in general, reaching superhuman performance in Go [7], Starcraft 2[8], Dota 2 [9], and a number of other games [10]. We chose to apply deep reinforcement learning to poker, inspired by AlphaStar's approach of emulating an entire league of hundreds of competing agents [8].

[1]University of Applied Sciences and Arts Northwestern Switzerland

SECTION 2

Theoretical Methods

2.1 Poker

No-Limit Texas Hold'em Poker is a turn-based game, where each player starts out with a fixed amount of chips they can bet in each round (called a hand in this thesis), and the winner of the hand collects all bets. The total winnings and loss of each player are summed over all hands, and whichever player ends up with the highest total wins.

2.1.1 Cards

The game is played is played with a standard 52-card french deck without jokers. At the start of the game each player receives two cards called hole cards that should not be revealed to the other players. Over the course of the game five more cards are revealed to all players, called community cards. If there is a contest, the winner is determined by hand strength, which depends on patterns of cards between all the cards a player can see [11].

2.1.2 Positions

Before the players are dealt their cards, one player receives the dealer button. This sets the betting order. Two other important roles are the small and big blind, who are forced to place an initial bet before the game starts. The seating positions are as follows:

- Dealer Button (BTN)
- Small Blind (SB)
- Big Blind (BB)
- Early Position (EP), or also called Under the Gun (UTG) because this player is the first to act during the first phase.
- Middle Position (MP)
- Cut Off (CO)

2.1.3 Actions

A player can take one of three actions: fold, call or raise.

Folding means that the player drops out of the hand, loses their current investment, is not eligible to win the pot, cannot take any actions until the next hand, but does not have to reveal their hole cards.

Calling is when a player matches the current highest bet, also called a check if no other player has raised during this phase.

Raising is when a player increases the current highest bet by raising their bet over the previous highest bet. There is a minimum amount of money that the player can raise by, to prevent long games where one player after another raises by a minuscule amount. A raise at the start of a betting round is also called a bet.

2.1.4 Phases

Each hand is divided into five phases. During the first four, all players that have not folded have to either call, raise or fold until either everyone but one player has folded, all remaining players check, or every player after the last raise has either called or folded.

- Preflop (Each player receives their hole cards, the EP starts with betting)
- Flop (The first three community cards are revealed, the SB starts with betting)
- Turn (The fourth community card is revealed, the SB starts with betting)
- River (The fifth and last community card is revealed, the SB starts with betting)
- Showdown (This phase starts when either the river phase finishes or when a player goes all-in and not all other players fold. During this phase all players that have not folded yet reveal their cards and the player with the strongest hand receives the pot. If multiple players have equal hand strengths, the pot is divided equally between them.)

2.1.5 Competition Rules

The competition specified the rules more in depth, so that every team is able to train their agents accordingly. The python framework used is called PyPokerEngine [12] and has been modified for the challenge. There is no guarantee that hands are played in order, because some parts might be parallelized to speed up evaluation. The SB is set to 1 chip and the BB to 2 chips, each player starts every hand with 200 chips. Each team can submit their agent twice per week, which is then directly evaluated against some very simple agents called baselines. This baseline evaluation is run over 10'000 hands and the team receives the result as soon as the evaluation is completed. If the evaluation fails for any reason (software bugs, timeouts, etc.) it does not count against the number of weekly evaluations and the last successful submission remains the current entry. Twice every week the current agents of the competing teams are pitted against each other and the three agents submitted by the supervisor. They are evaluated over 100'000 hands and the results are visible on a leaderboard to which all teams have access.

2.2 Reinforcement Learning

2.2.1 Basic Concepts

Reinforcement learning is a subfield of machine learning where an agent optimizes its actions in a sequential task to maximize some reward signal. It does not require a dataset nor labels, but instead an explorable simulation. This seperates it from both supervised learning, which requires labels, and unsupervised learning, which typically does not have a reward signal.

An agent is a black-box system which receives observations of the current state (usually in the form of a numeric vector), processes this according to an internal policy function (which can have many representations), and outputs selected actions (as a numeric vector).

In a discrete environment, these actions along with external influences lead to a new state. A sequence of states is called a trajectory. Some states are associated with a reward; a real value that represents how desirable it is for the agent to be in that state. The agent seeks to maximize overall reward over its trajectories.

Tasks are divided into episodic and continuous tasks [13, chapters 3.3 & 3.4]. An episodic task is a finite task with a well-defined end, e.g. playing a chess game. Rewards of episodic tasks are often only known at the end of a single episode, and an agent iterates through many such episodes to map out the state space. In contrast, a continuous task may run forever (e.g. an exploration robot), and rewards trickle in at any time.

A concept often used in continuous tasks is time discounting, where rewards in the far future are considered less important than rewards in the near future. Formally, let $r(s)$ denote the reward of particular state s, t denote a particular time-step, S a trajectory of states from time-steps t_0 to t_{end}, and γ the time discount factor with $0 \leq \gamma \leq 1$ (commonly 0.99). Then the total reward $R(S)$ can be defined as

$$R(S) \coloneqq \sum_{t=t_0}^{t_{end}} r\big(S(t)\big)\gamma^{t-t_0} \tag{2.1}$$

which the agent tries to maximize.

Another important concept in reinforcement learning is the tradeoff between exploration and exploitation [13, chapter 2.1]. An agent must both act in the way it believes will lead to greatest reward, but also explore actions with unknown outcomes, and reattempt actions that previously led to bad outcomes. Different algorithms represent this tradeoff in different ways, but it is always present in some form.

2.2.2 Q-Learning

A policy π describes some mapping of states to actions, and captures the strategy of an agent. It is useful to define a value function $V_\pi(s)$ of a state and policy that describes the total reward from following that policy starting from that state[2].

Denoting the action recommended by policy π given state s by $\pi(s)$, and the resulting next state by $Env\big(s, \pi(s)\big)$, we can define a recursive equation the value function must satisfy:

$$V_\pi(s) = r(s) + \gamma V_\pi\Big(Env\big(s, \pi(s)\big)\Big) \tag{2.2}$$

(2.2) and its variants is known as the Bellman equation [14]. Intuitively speaking, it states that the total reward following a policy from a specific state is the reward of that state, plus the discounted total reward from the next state after having acted. One can define a similar equation in probabilistic terms if the next state after acting isn't deterministic.

The value function describes total reward starting from a state following a policy. However, it is helpful to consider the first action separately, and construct a so-called Q-function describing total reward following a policy after an arbitrary action a. We define it as:

$$Q_\pi(s, a) := r(s) + \gamma V_\pi\big(Env(s, a)\big) \tag{2.3}$$

It follows that:

$$Q_\pi(s, a) = r(s) + \gamma Q_\pi\Big(Env(s, a), \pi\big(Env(s, a)\big)\Big) \tag{2.4}$$

We additionally define $V^*(s)$ and $Q^*(s, a)$, respectively, as the value and Q-functions of the optimal policy.

The core concept behind (deep) Q-Learning is to construct a policy using a function approximator (e.g. a neural network), training this approximator as a supervised learning problem to satisfy (2.4), and always select the action with the highest predicted reward [15]. Over time, the policy will converge to $Q^*(s, a)$.[16]

To train an approximator based on itself recursively is highly unstable, so it is common

[2]Some authors define the value function as the total reward from following the optimal policy, and do not condition it on π. We keep it general here to reuse the same equations when looking at approximations of the optimal policy, and to keep notation consistent

to use a frozen copy of the Q-function on the right-hand side of the equation. This copy is updated less frequently, or updated with smoothing applied through so-called polyak averaging ($0 \leq \psi \leq 1$):

$$Q_\pi^{copy} := \psi Q_\pi + (1 - \psi) Q_\pi^{copy} \tag{2.5}$$

Though (2.4) is the general equation for Q-Learning, one can simplify it by introducing certain restrictions. In particular, in episodic tasks like poker, rewards can be set to 0 for all non-terminal states, and time-discounting becomes no longer reasonable unless one wishes to punish longer games, so γ is set to 1. Let $Term_\pi(s, a)$ indicate the terminal state of following policy π starting from $Env(s, a)$:

$$\begin{aligned} Q_\pi(s,a) &= r(s) + \gamma Q_\pi\Big(Env(s,a), \pi\big(Env(s,a)\big)\Big) \\ &= \begin{cases} r(s) & \text{if s is terminal} \\ Q_\pi\Big(Env(s,a), \pi\big(Env(s,a)\big)\Big) & \text{otherwise} \end{cases} \\ &= r\big(Term(s,a)\big) \end{aligned} \tag{2.6}$$

which is no longer recursive, and thus easier to train.

Q-Learning, as presented so far, is maximally exploitative; it never acts in what it believes are suboptimal paths to explore what they are actually like. In isolation, this means that Q-Learning is susceptible to honing immediately on a suboptimal policy and never improving. To remedy this, we add a certain level of noise to the actions, such that an agent selects a random action with probability ϵ [13, chapters 2.2 & 2.3]. The size of ϵ controls the exploration-exploitation trade off for Q-Learning, and ϵ is set to 0 in finished/non-training agents.

A significant limitation of Q-Learning in practice is that the Q-function must be evaluated for each action to compare expected total reward. This does not scale performance-wise to large action spaces, and cannot be applied on continuous action spaces, restricting domains on which it can be applied.

2.2.3 Soft Actor-Critic (SAC)

Soft Actor-Critic is a recent evolution of several deep reinforcement learning algorithms improving on Q-Learning. It can handle both discrete and continuous action spaces, it can handle the exploration/exploitation trade off in a principled way, and it boasts more stable convergence properties than many of its competitors [17].

Like Q-Learning, SAC trains a Q-function that attempts to predict the value of an action given a state and policy. Unlike Q-Learning, it also trains a policy network, that given a state outputs an action, and is trained to maximize the current Q-function. This removes the scaling issues that plagued Q-Learning, and allows continuous action spaces.

This bootstrapping of two neural networks on each other is often unstable, so SAC uses a few tricks to stabilize it. Two separate Q-functions are trained, each to their own polyak-averaged target as in Q-Learning. When the policy function is trained, both are evaluated, and the minimum is taken as optimization target. This prevents spurious optimism in the Q-functions misleading the policy network.

To understand SAC's approach to exploration, it is necessary to introduce the concept of policy entropy. The entropy of a policy is a measure of how difficult it is to predict what action it will select given a state. A deterministic policy that always returns the same action given the same state has entropy zero. The entropy of a stochastic policy in a specific state can be calculated by:

$$H(\pi, s) \coloneqq \sum_{a \in \pi(s)} -P_{\pi,s}(a) log\big(P_{\pi,s}(a)\big) \tag{2.7}$$

where $P_{\pi,s}(a)$ is the probability of policy π choosing action a in state s. This can be understood as "The entropy of policy π in state s is the sum of $-log\big(P_{\pi,s}(a)\big)$ for every possible action, weighed by the probability of π selecting that action".

SAC implements so-called entropy regularization. Not only does it attempt to maximize long-term reward, but it also attempts to maximize long-term entropy, as high entropy means highly random action selection and thus a high amount of exploration. It does this by adding an entropy bonus to the update equations for both the Q-functions and the policy function, and by making the policy function stochastic and returning a distribution to sample from, not simply a value.

To train the Q-functions, equation (2.4) is used with the added entropy bonus:

$$Q_\pi(s, a) = r(s) + \gamma\Bigg(Q_\pi\Big(Env(s, a), \pi\big(Env(s, a)\big)\Big) + \alpha H(\pi, s)\Bigg) \tag{2.8}$$

α is called the temperature and is a parameter to control the exploration-exploitation trade off, as it regulates how strongly weighed the entropy bonus should be.

The policy function does not return an arbitrary distribution. For simplicity, the neural network returns a normal distribution by outputting the mean and the standard deviation. To squash this in a finite range given by the action space limits, the normal distribution is squashed by a $tanh(x)$ function, leading to the overall policy distribution being a tanh-squashed normal distribution. This restriction allows computing the entropy in closed form as a function of the given standard deviation [18, explained in detail in the code comments of the SpinningUp SAC implementation], so the entropy bonus can be returned by the model alongside the sampled action.

2.3 TrueSkill

TrueSkill is a skill rating system developed by Microsoft in 2007 that attempts to measure relative skill of players over the course of many games played [19] [20].

It supports flexible team setups, draws, and more than 2 teams. It also boasts a solid theoretical foundation. This makes it more suitable for Texas Hold'em (a 6-player free-for-all) than ELO [21] or other common skill ranking systems that are designed for two competing teams.

TrueSkill assumes each player has a numeric skill rating. The delta between two players' skill ratings represents the likelihood of one player winning. The scale involved is arbitrary and defined through the choice of a parameter β, where a player being β points above another implies that player has an approximately 76% chance[3] of winning if matched against each other.

A player's true skill rating is unknown. Instead, TrueSkill models each player with normal distribution of where it believes their skill rating should be. The variance is then a direct measure of how certain TrueSkill is of a player's skill. First-time players start with a wide distribution, and as they accrue games over time the system learns more information about their skill and their distribution narrows.

When players match up against each other, TrueSkill performs a bayesian update, interpreting their previous skill rating distribution as a prior and the game results as an observation, returning new posterior belief distributions for players' skill rating [19] [20]. Upsets where a good player is beaten by a bad player cause larger updates, and the wider the prior distribution is, the larger the update.

For judging victories, TrueSkill only takes the player ranking during a match into account, whether they won first place, won second place, and so on. It does not take into account by how much they won, a narrow victory is equivalent to an overwhelming one, and the skill rankings only reflect how likely it is for a player to beat another, not by how much.

This has important consequences regarding poker, as a bad player occasionally wins against a better player thanks to lucky cards, and these occasional victories would have great impact on the ratings due to it being an upset. To remedy this, we group hands into

[3]Specifically, $\Phi\left(\frac{1}{\sqrt{2}}\right)$, where $\Phi(x)$ is the cumulative density function of the standard normal distribution.

a round of 10’000 hands, tally the total winnings of the round, and consider the full round a single match in TrueSkill. The larger the round, the lower the probability that a bad player beats a good player out of luck, and thus the more stable TrueSkill becomes.

SECTION 3

Implementation

3.1 Training Environment

We set up a modular training environment to compare different agent populations. The different divisions can be used to observe how agents influence each other, which matchup method produces the best results, and allows comparing agents that have not been trained against each other.

3.1.1 Agent Manager

The agent manager is the single source of truth for all agents, and is responsible for instantiating agents. It tags every agent with a unique id so other components only need to store agent id. This removes redundancy and also makes it easy to change agent information without side effects.

The agent manager keeps all important information about an agent: type, paths to model files, whether the agent is trainable or not, origin division and its name.

3.1.2 Leaderboard

A leaderboard ranks agents from best to worst according to one of the metrics. There are several different types of leaderboards using different metrics (e.g. TrueSkill, winnings, etc.). When a match in a division concludes, all leaderboards attached to the division are updated with the results. Further detail on the different metrics in section 3.2.

3.1.3 Divisions

A division is responsible for generating match ups between agents, running games, and updating leaderboards with the results. There are several types of divisions with different match up rules.

Divisions differentiate between two types of agents: teachers and students. A student is trainable, using one of the algorithms discussed in section 2.2 and adapts its behavior during play. A teacher does not change its behavior and is either a baseline with hard-coded behavior or a clone of a student with fixed weights. Divisions usually have their own set of agents and do not influence outside agents. However, there are some exceptions, as will be explained in sections 3.1.3.3 and 3.1.3.4.

3.1.3.1 Random Division

The random division requires at least one student and one teacher, and keeps a list of its own agents. Matchups are generated by randomly selecting one student and five teachers. These teachers do not have to be distinct; it is possible that the same agent is chosen five times.

3.1.3.2 Climbing Division

The climbing division also requires at least one student and one teacher, and keeps a list of its own agents. Unlike the random division, this division requires a TrueSkill-based leaderboard to function. The goal of this division is to match students more frequently against better teachers.

This division selects teachers weighted by their TrueSkill rating. Let μ_i denote the TrueSkill rating of the i'th agent, then that agent's probability $p(\mathit{Selected})$ among the n other agents is:

$$p(\mathit{Selected}) = \frac{{\mu_i}^2}{\sum_{j=0}^{n} {\mu_j}^2} \tag{3.1}$$

This is repeated five times, making it likely that the best agent is chosen multiple times.

This has two problems: first, a new agent is assumed to have an average rating, but is likely one of the best agents. Because of this, the agent might only rarely play and therefore take a long time to converge to its actual TrueSkill rating. Second, with a large number of agents there is an inequality between the number of games played per agent which also causes the TrueSkill rating of rarely appearing agents to be unreliable.

To counteract these problems, there is a fixed probability that instead of regular matchups with one student and five teachers, a matchup is generated with six teachers. This has the sole purpose of improving the reliability of the TrueSkill ratings and therefore improving regular matchups. For this six teacher matchup a random agent is selected and five other teachers with similar ranking are chosen as opponents, if enough agents are available no agent is chosen multiple times as to maximise the number of agent rankings being updated.

3.1.3.3 Permanent Evaluation Similar Division

Short "PermEvalSimilar", this division is intended to evaluate all agents across all divisions, not agents inside itself. The goal is to evaluate agents between divisions to not only test how agents react to previously unknown opponents but also to evaluate which divisions produce better agents.

This division does not have an internal agent list, and uses the agent manager directly. It looks at all available agents, except for students to not influence the learning process. It selects one agent at random and chooses five other agents with similar rankings, all unique if enough are available.

3.1.3.4 Permanent Evaluation Direct Division

Short "PermaEvalDirect", this division is special, as it compares only two agents against each other. It selects two random agents from the entire pool and creates three instances of each agent. The two agents must be different and the division tracks previously run matchups and prioritizes those that were not run before.

This division is specifically designed to compute the winnings metric (section 3.2.2).

3.1.4 Division Manager

The division manager works similarly to the agent manager in that it handles all divisions with their configurations and tags every division with a unique id. This is used to simplify the running of games, as user can simply add divisions with the desired configuration and agent constellation and then specify which divisions should be run.

3.2 Performance Metrics

3.2.1 TrueSkill

We did not reimplement TrueSkill, instead using an existing python implementation [22]. We used a β of $\frac{25}{6}$ and a starting distribution $\mathcal{N}(\mu = 100, \sigma = 30)$ for new agents.

3.2.2 Winnings

The winnings metric compares agents in a direct confrontation against each other by matching up both agents with three copies each. For both the average winning is calculated and added to a matrix of all agents.

Seats are randomized to minimize the advantage gained by a better seat. Ideally all seating combinations would be run for an equal number of hands, but this was not implemented because of increased complexity and computational time needed.

To improve runtime, matchups that have not been played are first in line for the next evaluation. Because of the quadratic growth of matchups needed this metric is not a good choice for a large number of agents.

To generate a leaderboard from the winnings matrix, the following measures are used:

- mean of winnings
- median of winnings, which has the advantage of not being distorted by different performance against terrible agents (e.g. baseline agents)
- percentiles (e.g. 20pctl) of winnings, which has the advantage of showing how the agent performs against its worst matchups, giving a measure of robustness

3.3 Strategy Analysis

We use several metrics to try to figure out how different agents behave, and how they differ in their behavior. Some of these are primarily for debugging, others for insight-generation.

3.3.1 Cashflow

Some agents play worse than others in all situations. Others might usually win, but lose hard against certain exploiters. Yet others prefer to "avoid losing" and make small gains but never really exploit others.

To distinguish which agents were exploiting which how much, we measure cashflow; how much money was each agent giving over to each agent on average.

Cashflow is computed individually per round, though conceptually it would not be impossible to compute it on average throughout. However, it requires extensive logs containing the individual results of every single hand, to track who wins what, which slows training considerably.

While useful for debugging specific early agents, cashflow was both too slow and requires too many games until convergence to be a useful metric in larger runs, so we did not use it in section 4.

3.3.2 Strategy Vector

To compare two different player strategies, one concept is to consider strategies as black boxes and compare how they behave in a number of situations. This was the approach we chose for our so-called strategy vector.

Each agent to be analyzed is instantiated and fed fake game states, generated by iterating over the number of visible cards, seating position, number of players who haven't folded yet, and investment into the pot (which also controls pot size). For every combination of these variables, 10'000 hands were generated, sorted into equally-sized card rank bins, and the agent's behavior was logged for every combination until sufficient samples were gathered for each bin.

The action space is divided into 53 bins, corresponding to folding, calling, checking, and raising in intervals of 4 (to a maximum of 200). Each of these bins contains the number of hands the agent performs that action in that situation.

The resulting 4'320-dimensional array of situations with action frequencies (in 53 bins) is considered a characterization of that agent's play-style, called that agent's strategy vector, and saved to disk (a 228'960-dimensional array in total, approximately 1.8 MB per agent).

Computing this vector takes a few minutes per agent, and represents the bulk of computation required for strategy plotting, allowing quick generation of different plots.

Card ranks are calculated differently based on the number of cards. For pre-flop, we use monte-carlo simulation until convergence to generate a table of the victory likelihood of each possible pair, which lets us rank them from best to worse, and divide these into deciles.

For flop, turns, and rivers, this is not feasible. Instead, we computed the card rank as evaluated by Treys [23] directly, and compared that with a precomputed table of the number of possible hands that have higher rank. This is not perfectly equal to the likelihood of a hand winning, but much easier to compute, so we used this approach as an approximation.

3.3.3 Aggression and Tightness

Two common metrics for playstyle in poker are aggression and tightness [24].

Aggression compares the number of situations in which a player increases the pot vs just calling, and as such measures a player's proactivity. It can be defined as:

$$Aggression := \frac{\#bets + \#raises}{\#calls} \tag{3.2}$$

where # indicates the number of each betting action. Note that checks are not included in the calls.

Because this ratio can go all the way to infinity, it is difficult to plot, and the differences between agents are overstated at small numbers of calls. To remedy this, we use a distortion of aggression in all our plots:

$$\textit{Distorted Aggression} \coloneqq \begin{cases} \frac{\textit{Aggression}}{\textit{Aggression}+1} & \text{if } \textit{Aggression} \text{ is finite} \\ 1 & \text{otherwise} \end{cases} \tag{3.3}$$

This compresses aggression to a $[0, 1]$ range, with $\frac{1}{2}$ being the natural midpoint of the same number of raises/bets and calls[4].

Tightness is defined by how many hands a player folds:

$$\textit{Tightness} \coloneqq \frac{\#\textit{folds}}{\#\textit{calls} + \#\textit{bets} + \#\textit{raises}} \tag{3.4}$$

We decided to ignore checks, as most of our agents have a built-in behavior where they always check if they can and want to call or fold. This makes the number of checks mostly dependent on the situation distribution, and not on the agent behavior.

Both metrics are computed based off of the precomputed strategy vectors, as this both gives a consistent set of situations between agents (unlike game logs), and sufficient information to rapidly compute them. This does affect tightness, as tightness makes more sense computed over full games, but the consistency and lack of dependence on slow-to-produce game logs made the tradeoff worth it.

3.4 Agents

3.4.1 Baselines

We used 2 kinds of baseline agents during training, to give an initial population to train against.

The first was a random agent, who played randomly. Originally we tuned the likelihood of actions to prolong game length, but as the competition baseline used a flat $\frac{1}{3}$ chance for each action, we did the same so our bots would adapt to the same baselines. It chooses from {fold, call/check, raise/bet} with equal probability, and in the case of a raise or bet chooses the amount randomly from what it still has available.

The other frequently used baseline was a call agent, that did nothing but call every hand in every situation.

[4]This is similar to converting odds ratio to probabilities, which uses the same formula and also compresses into a $[0, 1]$ range without losing too much interpretability.

3.4.2 Q-Learning

We used OpenAI's SpinningUp ([18]) as a base implementation of Q-Learning, as it is well-tested, well documented, and coherent with the guides and explanations on the SpinningUp website. This base code was modified to fit into our framework as SpinningUp uses a very specific environment API and does training in a separate loop impractical for our purposes.

In doing so we also made changes to the algorithm. The biggest change was dropping the Bellman equation through the simplification mentioned in equation 2.6, instead training all Q-function outputs of a played trajectory T towards the final reward r with mean squared error loss:

$$Loss \coloneqq \sum_{(s,a)\in T} (Q(s,a) - r)^2 \tag{3.5}$$

Because the final reward of folding is always known ($r = -(current\ stake)$), we do not use Q-Learning to compute the outcome of folding, and the inverse of the current investment is used directly as the Q-function score of the folding action. This allows us to avoid having to explore randomly folding in the middle of games, which dramatically increases skill and prevents the agent from playing too cautiously because it knows that it will throw a fraction of games.

We feed all agents (Q-Learning and SAC) with the same 404-dimensional vector: a one-hot representation of the sorted hole cards, the sorted community cards (with an extra one-hot slot for "hidden"), which players have folded, which players put how much of their capital into the pot so far in the entire hand, which players put how much into the pot in this betting round, which player was the last to raise, which player is the current agent/where it is seated, which betting round it is in, and what is the card rank percentile of the current hand computed in the same way as described in section 3.3.2.

We use two variants of action spaces in the results section, after some explorative experimentation with other variants: The Qlearn-8 agent with 8 different actions (fold, call, raise 4, raise 8, raise 16, raise 32, raise 64, raise 100, raise 200), and a Qlearn-All agent with 201 different actions (fold, call, raise 1, raise 2, . . . , raise 199, raise 200).

Naively implemented, Qlearn-All would perform around 20 times slower than the Qlearn-9 agent. For performance reasons, we added a hack to Qlearn-All: Instead of computing the Q-function score of a state-action pair, it computes the Q-function score of a state for all actions at once, returning a 201-dimensional vector after one traversal. This theoretically "consumes" more of the neural network, but in practice we found this to not matter in initial experiments.

Both variants used a noise ϵ of 0.1 (10% of actions were randomly sampled across all available actions except for folding) and a learning rate of 0.001, using an Adam optimizer.

The network is a 4-layer feed-forward neural network, with each layer containing 256 neurons and a leaky ReLU activation function ($max(x, -0.001x)$), except for the last layer which uses $tanh(x)$.

3.4.3 Soft Actor-Critic

Like Q-Learning, we based our implementation off of [18], for the same reasons. Unlike Q-Learning, we did not remove the Bellman equation, as non-terminal states are relevant for total entropy, which in SAC is part of the Q-function score.

Each SAC agent holds 5 neural networks: two Q-function networks, two polyak-averaged target networks of the respective Q-functions, and the policy network.

We use the same input space for SAC agents as Q-Learning agents, but the action space is different: SAC agents return a 4-dimensional bounded floating-point vector, of which the first three dimensions are interpreted as desire to fold, call or raise (the highest is chosen as action), and the last as how much to raise by.

Our SAC agents use a time discount factor $\gamma = 1$ (no time discounting), Q-function network learning rate of 0.001 (with Adam optimizer), policy network learning rate of 0.1 (also with Adam), polyak averaging factor of 0.999, and differ by temperature: SAC-High uses $\alpha = 0.01$, and SAC-Low uses $\alpha = 0.0005$. The neural networks are identical to the one used by Q-Learning agents, with some added post-processing to the outputs of the policy network for action sampling and entropy calculation.

3.5 Vectorized Poker Engine

The competition itself in AIcrowd uses PyPokerEngine [12] as engine to simulate the poker games. This engine is built in a flexible way, but it is not optimized for speed, even after some modifications to use Treys [23] for card rank calculation. It also computes each hand one after another, making inference mini-batching almost impossible.

To train, we need the ability to simulate millions of games, and ideally also to batch them so that the neural networks can be fed entire mini-batches of games at a time. This led us to develop our own poker simulation engine based primarily on Numpy [25] and Treys.

This engine computes batches of 10'000 hands per so-called round, all of them with the same agents in the same seating positions, but with different cards, bets, and victors. Almost every operation is performed as a numpy array operation, except hand score computation, as Treys does not support our batching method.

For a rough benchmark of performance gains, on one of our machines PyPokerEngine required 39.25 seconds to compute 10'000 hands between 3 call and 3 random bots, and 539.55 seconds for 10'000 hands between 6 Qlearn-All agents. In contrast, our engine on the same machine with the same parameters averaged 0.84 seconds and 8.74 seconds respectively, demonstrating an approximately 50x speedup factor.

One consequence of this is that games are computed "in parallel", and agents cannot form judgements based on the history of previous games. This excludes many potential agent architectures and opponent modeling strategies, but we found it an acceptable limitation for the performance gains, and the competition rules make history unreliable.

To give an example of what this vectorizing looks like, here is some example code that computes showdown winnings. Both `hand_scores` and `total_winnings` are 2-dimensional arrays of size `(BATCH_SIZE, N_PLAYERS)`, with `hand_scores` containing integer card ranks as computed by Treys and `total_winnings` the stack after each game. Side pots do not need to be considered, as all players have the same amount of money available. The somewhat exotic construction `sorted_hands[:, 0][:, None]` tiles the first column of `sorted_hands` to as many columns as is needed to match `hand_scores`.

```
ranks = np.argsort(hand_scores, axis=1)
sorted_hands = np.take_along_axis(hand_scores, indices=ranks, axis=1)
# Get everyone who has the best hand and among which pots will be split
participants = hand_scores == sorted_hands[:, 0][:, None]
# Get the number of times each pot will be split
n_splits_per_game = participants.sum(axis=1)
# Split and distribute the money
gains = pool / n_splits_per_game
total_winnings += participants * gains[:, None]
```

SECTION 4

Results

4.1 Experiment Description

To demonstrate our results, we plotted results from one standardized run of multiple division configurations trained over 2'000 rounds of 10'000 hands each, for a total of 20'000'000 hands.

All divisions included one random and one call agent as baselines, and all divisions cloned a new generation of teachers every 200 rounds, for a total of 90 agents over the whole run.

Name	Matchup Method	Agents
Qln8-Cl	Climbing	Qlearn-8
QlnA-Cl	Climbing	Qlearn-All
SacL-Cl	Climbing	Sac-Low
SacH-Cl	Climbing	Sac-High
AllAg-Cl	Climbing	Qlearn-8, Qlearn-All, Sac-Low, Sac-High
QlnA-Rn	Random	Qlearn-All

Table 4.1: Training division configurations used in the standardized run

We trained on a computer with an Intel 6770k CPU, 16GB Ram, and an Nvidia 1070GTX GPU with 8GB VRAM. The training process took about 44 hours in real-time.

After training, we ran three PermaEval divisions to evaluate all agents according to the various metrics: one for all the winnings-based metrics, one for across-division TrueSkill, and a second TrueSkill to evaluate its consistency.

We computed winnings with 10'000 hands per agent pairing of all 102 agents, for a total of 52'020'000 hands. For fairness of comparison, TrueSkill was evaluated for 5'300 rounds, which is approximately the same number of hands.

4.2 Performance Metrics

First, we compare different metrics in consistency and strategy clustering, and take a closer look at TrueSkill.

4.2.1 Leaderboards

Name	TS	Mean	Med	20-Pctl
exam	151.04	3.26	2.35	0.21
radar	148.97	2.79	2.02	0.14
loss	146.49	3.95	2.76	0.33
tan	143.41	4.47	2.55	0.19
brick	142.55	5.13	2.65	0.35
clam	142.43	4.67	2.81	0.15
seeker	141.14	6.16	2.98	0.33
wisdom	140.39	1.47	0.94	0.06
suitcase	138.85	4.81	2.45	0.33
flight	138.73	4.58	2.53	0.23

(a) Sorted by TrueSkill (TS)

Name	TS	Mean	Med	20-Pctl
orange	129.16	6.79	1.32	-0.07
watt	133.09	6.59	2.24	0.23
seeker	141.14	6.16	2.98	0.33
ecclesia	124.79	6.15	1.42	0.00
config	122.43	6.06	0.89	-1.20
power	132.34	6.04	1.58	0.00
creator	134.82	5.91	2.59	0.03
union	129.96	5.65	1.50	0.01
cowbell	118.01	5.50	0.88	-2.32
yarn	119.79	5.50	1.14	-1.60

(b) Sorted by Mean

Name	TS	Mean	Med	20-Pctl
seeker	141.14	6.16	2.98	0.33
clam	142.43	4.67	2.81	0.15
penguin	135.08	5.22	2.76	0.00
loss	146.49	3.95	2.76	0.33
brick	142.55	5.13	2.65	0.35
creator	134.82	5.91	2.59	0.03
tan	143.41	4.47	2.55	0.19
flight	138.73	4.58	2.53	0.23
suitcase	138.85	4.81	2.45	0.33
exam	151.04	3.26	2.35	0.21

(c) Sorted by Median (Med)

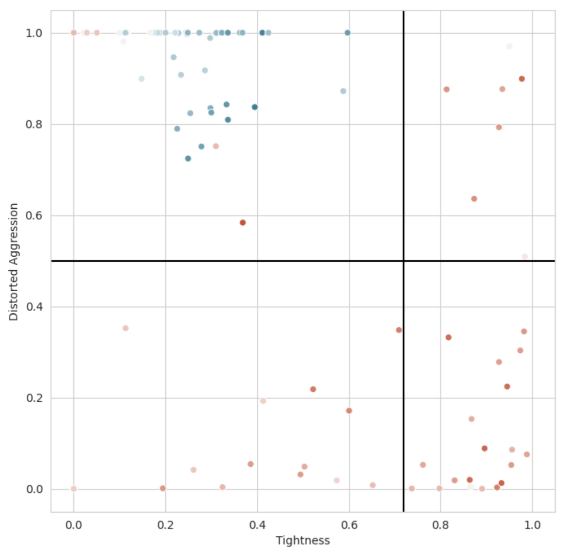
Name	TS	Mean	Med	20-Pctl
brick	142.55	5.13	2.65	0.35
suitcase	138.85	4.81	2.45	0.33
loss	146.49	3.95	2.76	0.33
seeker	141.14	6.16	2.98	0.33
watt	133.09	6.59	2.24	0.23
flight	138.73	4.58	2.53	0.23
exam	151.04	3.26	2.35	0.21
tan	143.41	4.47	2.55	0.19
rain	135.02	4.20	1.01	0.17
clam	142.43	4.67	2.81	0.15

(d) Sorted by 20-Percentile (20-Pctl)

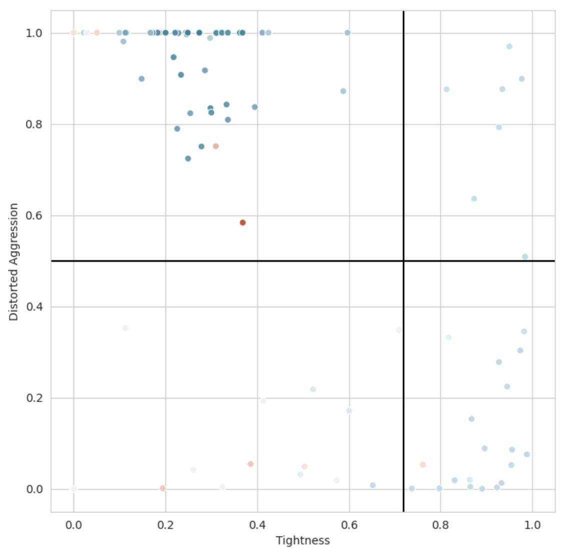
Table 4.2: Top 10 agents on leaderboard sorted by each metric

We can see that the four metrics we used disagree on which agents were best. There are a handful of agents (e.g. *seeker*, *exam*, *brick*, or *suitcase*) that show up in most leaderboards near the top, and it is from these agents that we selected candidates for the AIcrowd competition (specifically, *seeker* was our final agent), but overall there is substantial disagreement.

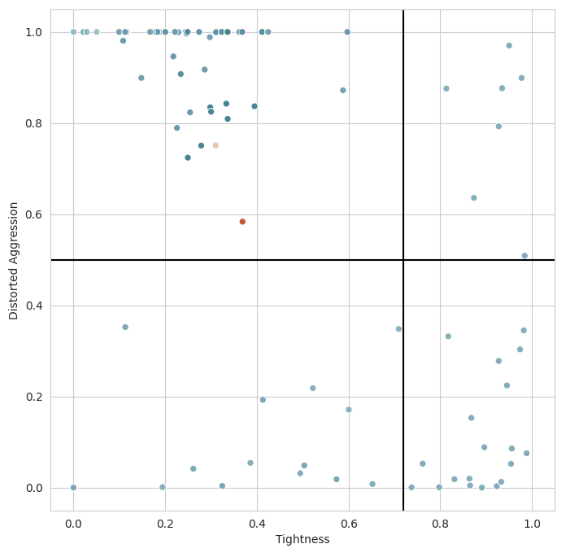
As one might expect, mean diverges from the other metrics most. We believe this is because mean does not reward consistency, e.g. losing a bit often and winning big sometimes is a good strategy for mean optimization. All three other metrics discourage it, and reward worst-case consistency.



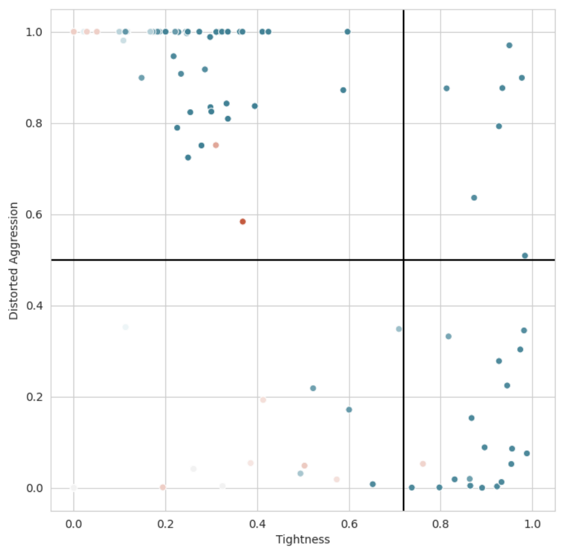
(a) Colored by TrueSkill



(b) Colored by Mean



(c) Colored by Median



(d) Colored by 20-Percentile

Figure 4.1: Aggression and Tightness colored by rank for each metric, blue is better (normalized)

However, this difference is not visible in the strategy plots. Instead here it is TrueSkill that is an outlier, as it ranks tight and passive agents as worse overall than all others, though mean also considers them worse than the other two.

This would lead to the hypothesis that tight and passive players, which we will see later (section 4.3) are mostly SAC agents, play consistently ok in their worst games (decent 20pctl and median), but rarely take advantage of good games (low mean), and also rarely win matches (low TrueSkill). This is consistent with the definition of passivity and tightness, though one would expect the aggressive and loose players to have lower 20pctl scores, which we do not observe.

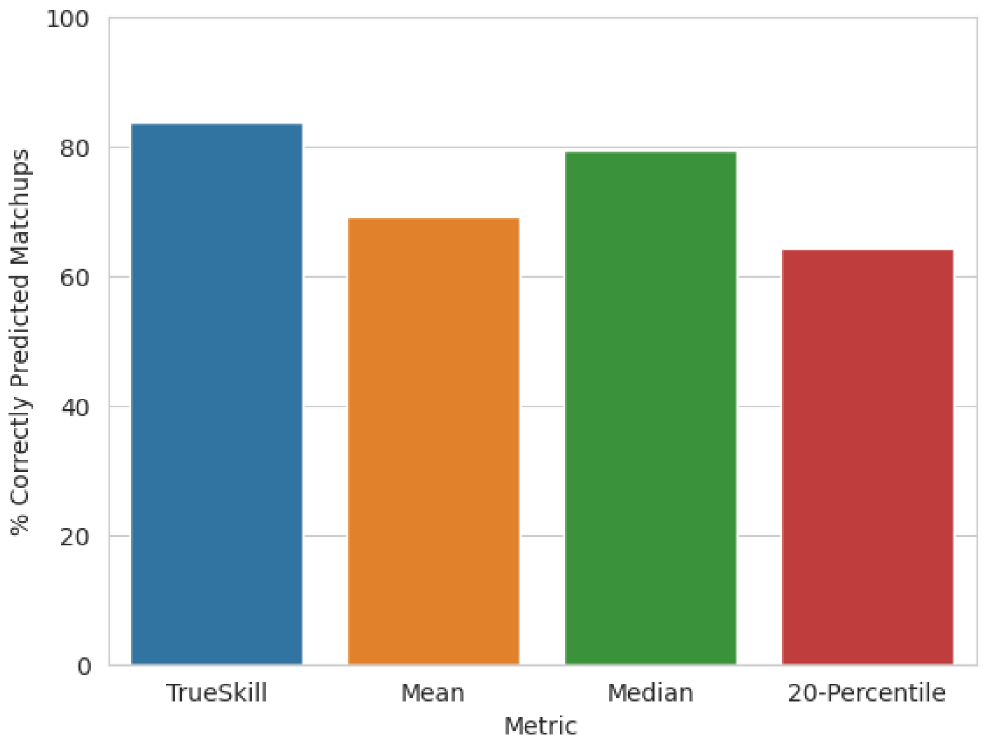
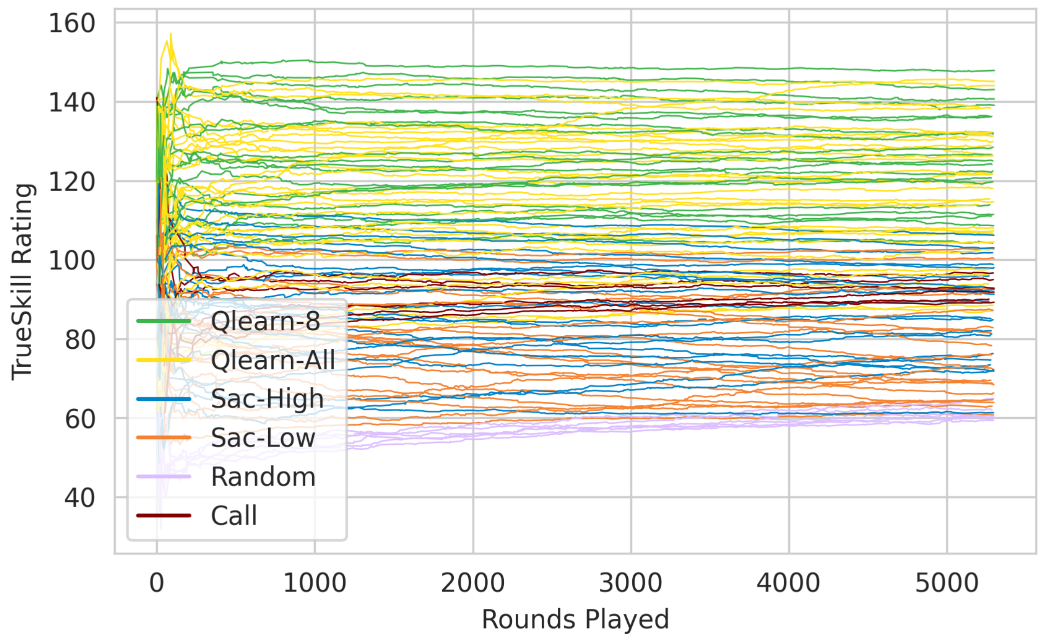


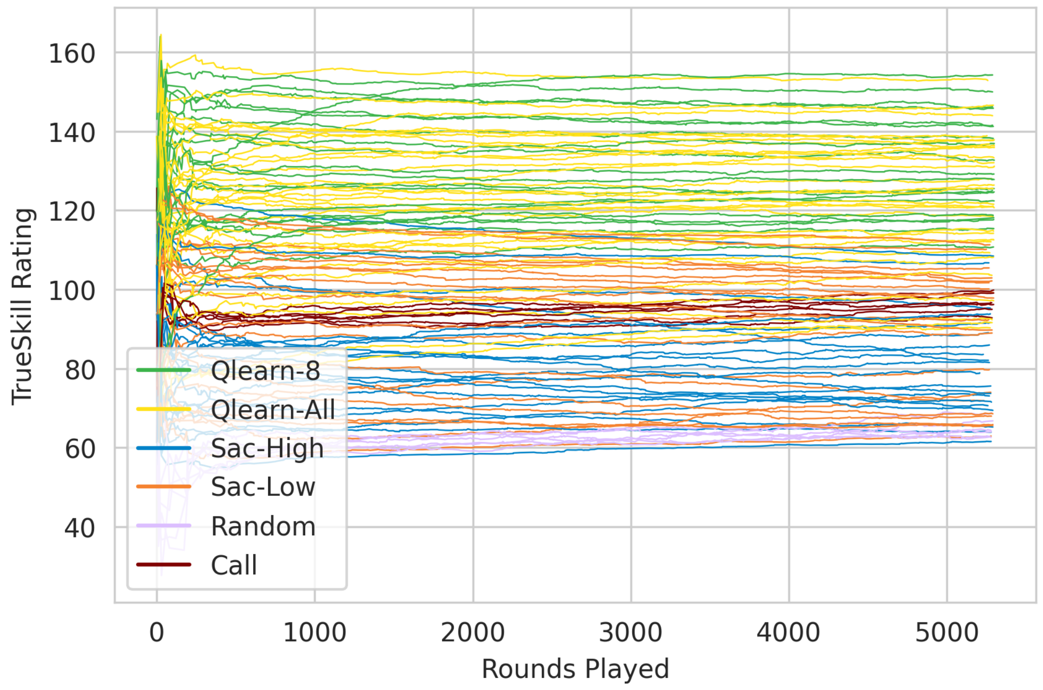
Figure 4.2: Percent 1v1 evaluations whose outcome matches what the metric difference predicts

This plot demonstrates how good these metrics are at predicting actual win/loss in the 3v3 evaluation. We can see that TrueSkill as the only metric specifically optimizing for win probability performs best, closely followed by median.

4.2.2 TrueSkill Coherence



(a) First PermaEval division TrueSkill



(b) Second PermaEval division TrueSkill

Figure 4.3: TrueSkill over time in both PermaEval divisions

TrueSkill did not converge quickly or well in preliminary experiments, so to establish that it is at least consistent we computed the full ranking twice.

While the broad tendencies stabilize quickly, we still observe agents swapping or even completely changing their skill rating by as much as 40 points after the first 500 rounds, despite all agents being static and always playing at the same skill level. This does not bode well for convergence.

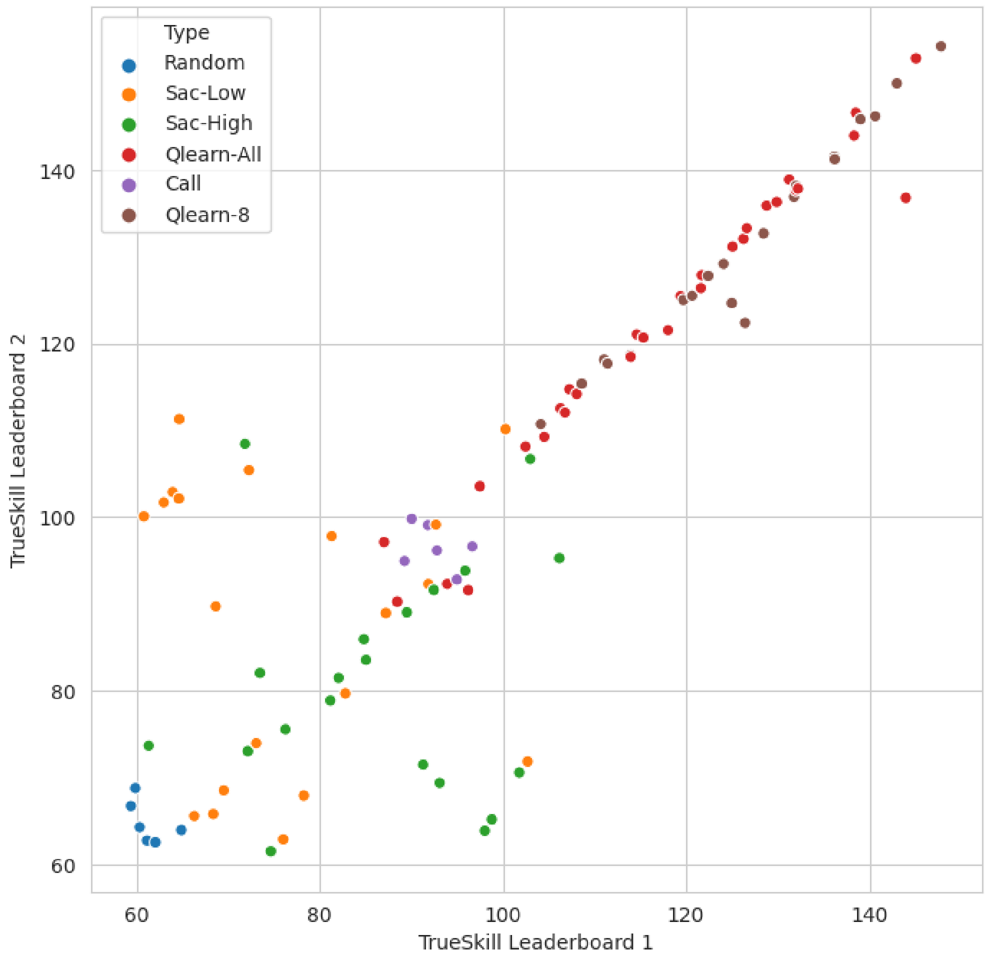


Figure 4.4: Agent rank in both TrueSkill PermaEval divisions, colored by type

Here we see the rank of all agents as a scatterplot. Were TrueSkill perfectly consistent, this would be a straight line. Instead we see a mystifying result of it being dependent on agent types; SAC agents appear to play inconsistently when judged by TrueSkill (i.e. ranking after matches), but other agents are consistent, if not perfectly so. This seems less an artifact of TrueSkill and more an issue with SAC agents themselves.

4.3 Agents

How do the different agent architectures compare?

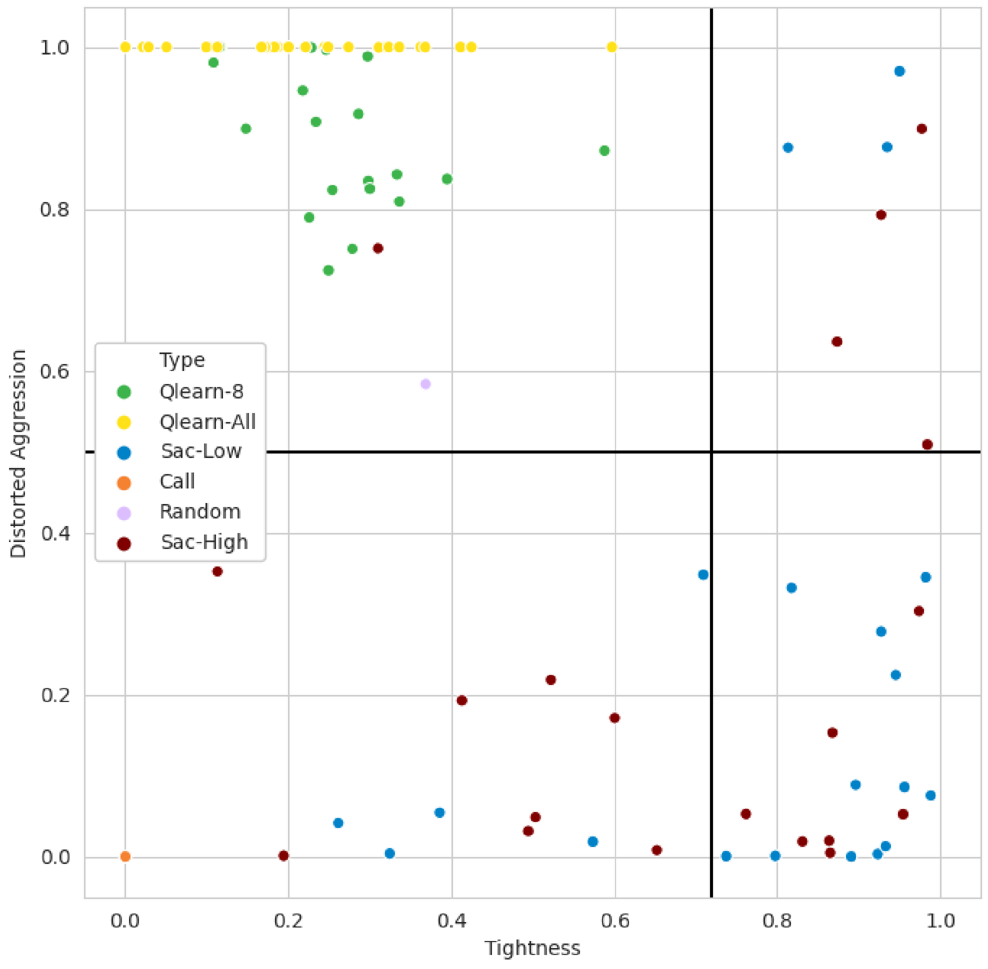


Figure 4.5: Aggression and Tightness, colored by agent type

Agent architectures appear paramount for their strategy. This plot mixes results from multiple divisions (i.e. different seeds and different opponents), which does not prevent a remarkable convergence by agent type.

We see that Qlearn-All almost never calls, and either folds or raises, leaning towards more raises. Qlearn-8 shares a similar tendency but not nearly as strong (the scale of *Distorted Aggression* is hyperbolic towards the top). We believe this is because of implementation details of exploration, more in section 5.3.

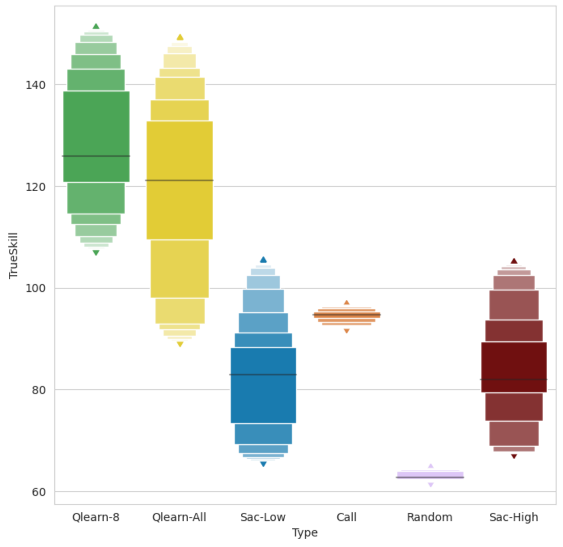
In contrast, SAC agents play aggressive only when tight, or loose when passive. This is more "sensible" by common sense (if you call less, you fold more), but appears to perform worse in our leagues.

Rank	TrueSkill	Mean	Median	20-Percentile
1	Qlearn-8	Qlearn-All	Qlearn-All	Qlearn-All
2	Qlearn-All	Qlearn-All	Qlearn-8	Qlearn-All
3	Qlearn-8	Qlearn-All	Qlearn-8	Qlearn-8
4	Qlearn-8	Qlearn-All	Qlearn-8	Qlearn-8
5	Qlearn-All	Qlearn-All	Qlearn-All	Qlearn-All
6	Qlearn-8	Qlearn-All	Qlearn-All	Qlearn-8
7	Qlearn-All	Qlearn-All	Qlearn-8	Qlearn-8
8	Qlearn-All	Qlearn-All	Qlearn-8	Qlearn-8
9	Qlearn-8	Qlearn-All	Qlearn-8	Qlearn-All
10	Qlearn-8	Qlearn-All	Qlearn-8	Qlearn-8
11	Qlearn-8	Qlearn-All	Qlearn-All	Qlearn-All
12	Qlearn-All	Qlearn-All	Qlearn-All	Qlearn-All
13	Qlearn-All	Qlearn-8	Qlearn-All	Qlearn-8
14	Qlearn-All	Qlearn-All	Qlearn-8	Qlearn-All
15	Qlearn-8	Qlearn-8	Qlearn-All	Qlearn-All
16	Qlearn-All	Qlearn-8	Qlearn-8	Qlearn-8
17	Qlearn-All	Qlearn-8	Qlearn-All	Qlearn-All
18	Qlearn-8	Qlearn-8	Qlearn-All	Qlearn-All
19	Qlearn-All	Qlearn-8	Qlearn-All	Qlearn-All
20	Qlearn-All	Qlearn-8	Qlearn-All	Qlearn-All

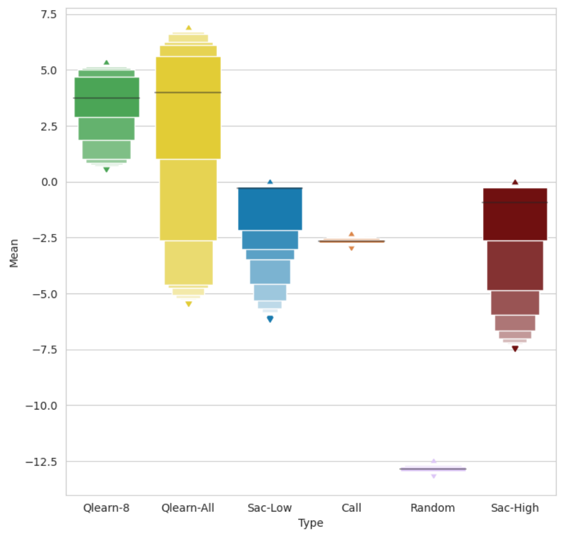
Table 4.3: Agent type of the top 20 agents according to each metric

The first immediate observation is that no architecture other than the two Qlearns was able to compete.

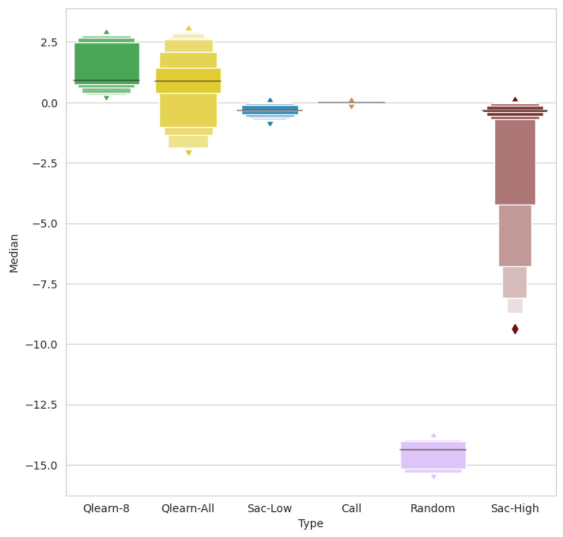
The second is that while Qlearn-8 and Qlearn-All seem approximately evenly matched in most metrics, in mean Qlearn-All is dominant, earning more money on average despite not winning more often. This is consistent with Qlearn-All's higher aggression.



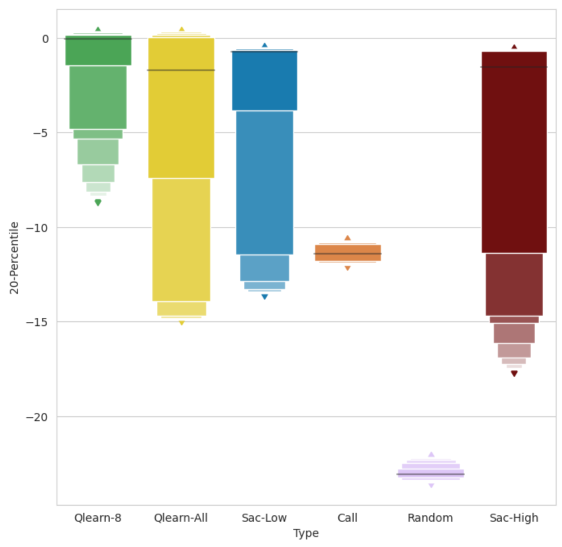
(a) Ranked by TrueSkill



(b) Ranked by Mean



(c) Ranked by Median

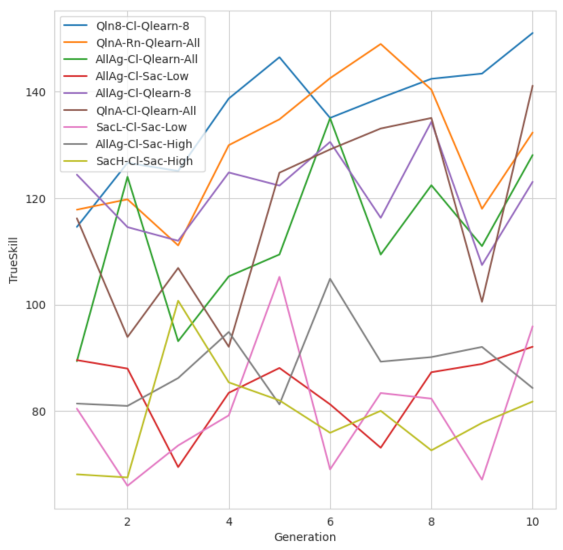


(d) Ranked by 20-Percentile

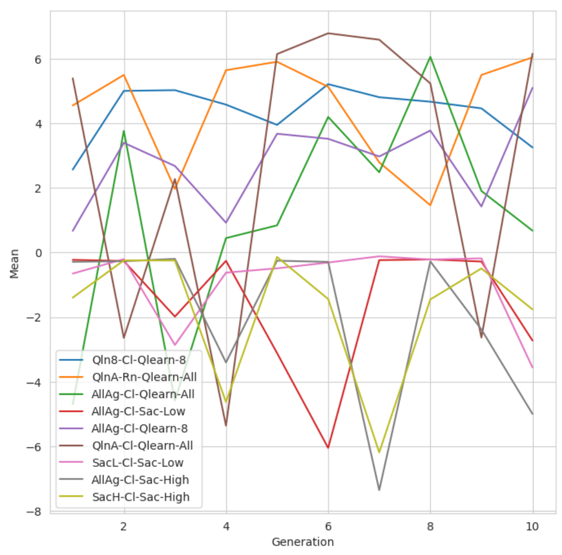
Figure 4.6: Agent distribution by type and rank

We can see that the relative performance of SAC agents differs depending on metrics; ranked by mean or median they outperform call agents, but ranked by TrueSkill only the best of them do.

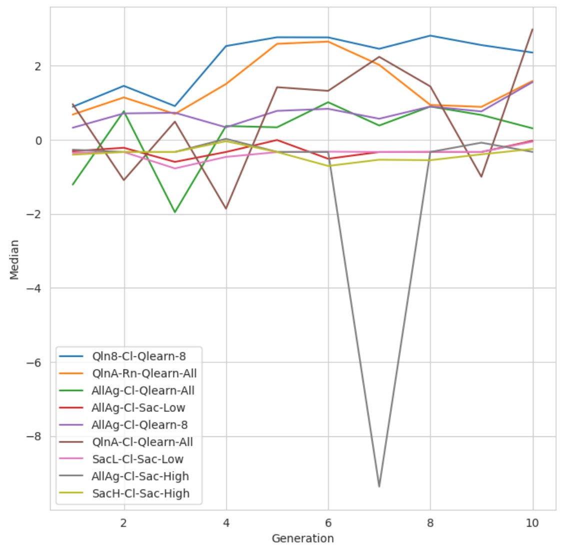
We can also see that 20pctl is left-skewed, with the best agents being closer to the norm than the worst agents. This makes it difficult to compare the best half, but it allows some insight in the worst; we see that Qlearn-All is surprisingly good, even with its worst untrained agents, in its worst 20% of matches, despite extreme aggression.



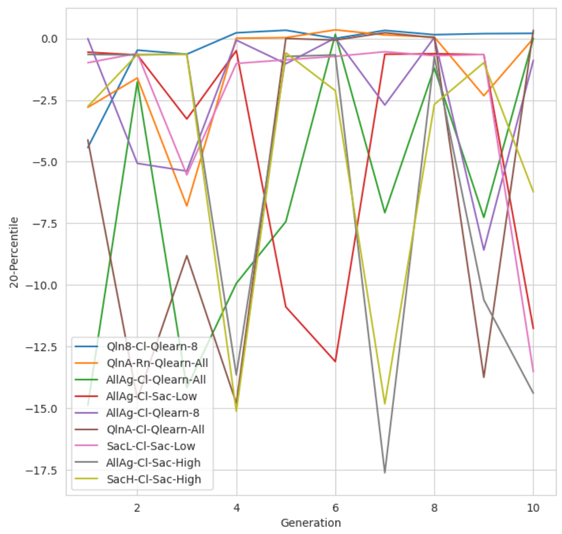
(a) Ranked by TrueSkill



(b) Ranked by Mean



(c) Ranked by Median



(d) Ranked by 20-Percentile

Figure 4.7: Agent evolution over time by type and rank

Each agent left clones of itself every 200 iterations. Comparing these clones by generation allows us to see how agents improved over time.

On first sight, it looks like they haven't. Improvement is visible in TrueSkill for the Qlearn agents, as well as small improvements in the other metrics, but overall this is less improvement than expected.

4.4 Populations

How important are division populations?

Rank	TrueSkill	Mean	Median	20-Percentile
1	Qln8-Cl	QlnA-Cl	QlnA-Cl	QlnA-Rn
2	QlnA-Rn	QlnA-Cl	Qln8-Cl	QlnA-Cl
3	Qln8-Cl	QlnA-Cl	Qln8-Cl	Qln8-Cl
4	Qln8-Cl	QlnA-Cl	Qln8-Cl	Qln8-Cl
5	QlnA-Rn	AllAg-Cl	QlnA-Rn	QlnA-Cl
6	Qln8-Cl	QlnA-Rn	QlnA-Rn	Qln8-Cl
7	QlnA-Cl	QlnA-Rn	Qln8-Cl	Qln8-Cl
8	QlnA-Rn	QlnA-Rn	Qln8-Cl	Qln8-Cl
9	Qln8-Cl	QlnA-Rn	Qln8-Cl	AllAg-Cl
10	Qln8-Cl	QlnA-Rn	Qln8-Cl	Qln8-Cl
11	Qln8-Cl	QlnA-Cl	QlnA-Cl	QlnA-Rn
12	QlnA-Cl	QlnA-Cl	QlnA-Rn	QlnA-Rn
13	AllAg-Cl	Qln8-Cl	QlnA-Rn	AllAg-Cl
14	QlnA-Rn	QlnA-Rn	AllAg-Cl	QlnA-Rn
15	AllAg-Cl	AllAg-Cl	QlnA-Rn	QlnA-Cl
16	QlnA-Cl	Qln8-Cl	Qln8-Cl	AllAg-Cl
17	QlnA-Rn	Qln8-Cl	QlnA-Cl	QlnA-Rn
18	AllAg-Cl	Qln8-Cl	QlnA-Cl	AllAg-Cl
19	QlnA-Rn	Qln8-Cl	QlnA-Cl	QlnA-Cl
20	QlnA-Cl	Qln8-Cl	QlnA-Rn	QlnA-Rn

Table 4.4: Agent division of the top 20 agents according to each metric

Climbing appears only marginally better than Random matching, though enough to be relevant. This accords with smaller experiments we did before this run. It also seems that mixing all agents decreases performance overall, indicating that overfitting is not (yet) an issue. It might be that competition against better agents is important, but its relative underperformance may also be a symptom of its lower training time; because the mixed division has more agents, the training rounds are split between them, leaving each individual agent with less rounds per cloning generation.

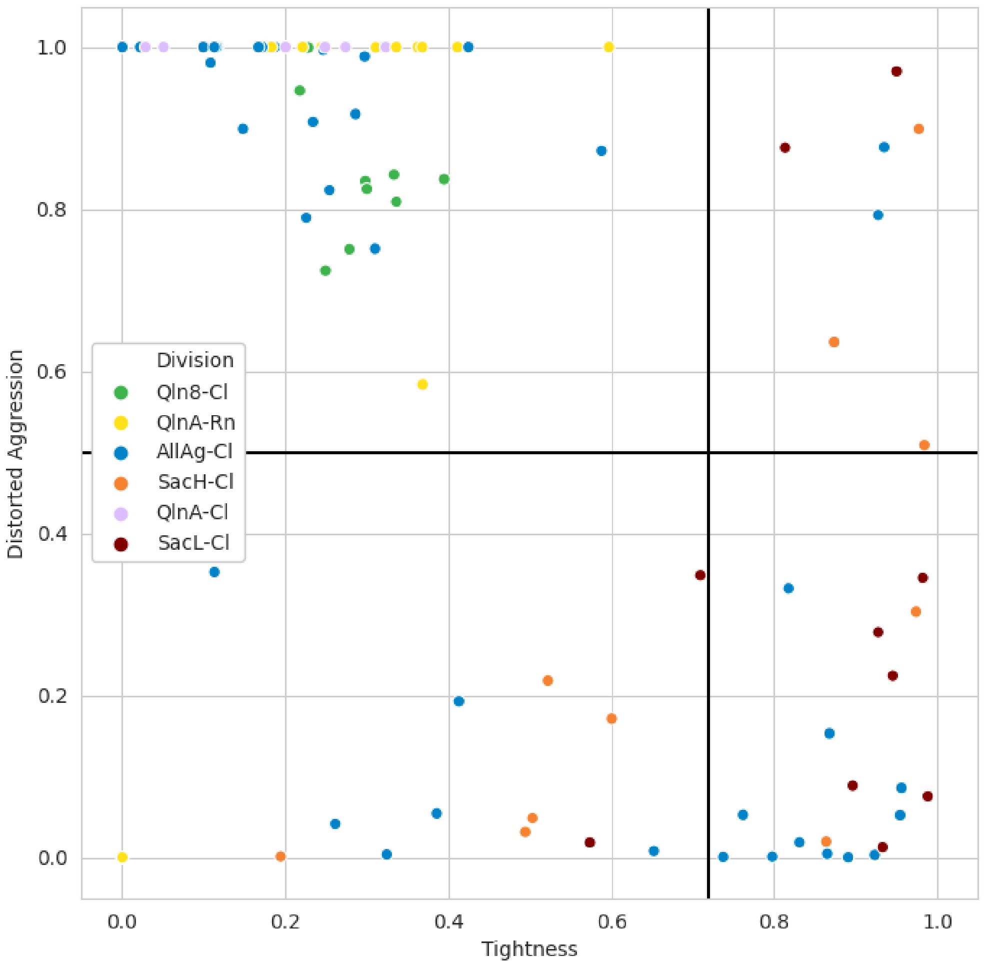
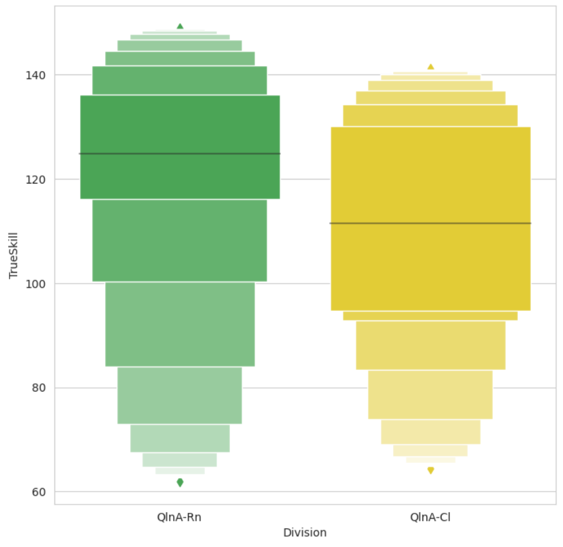
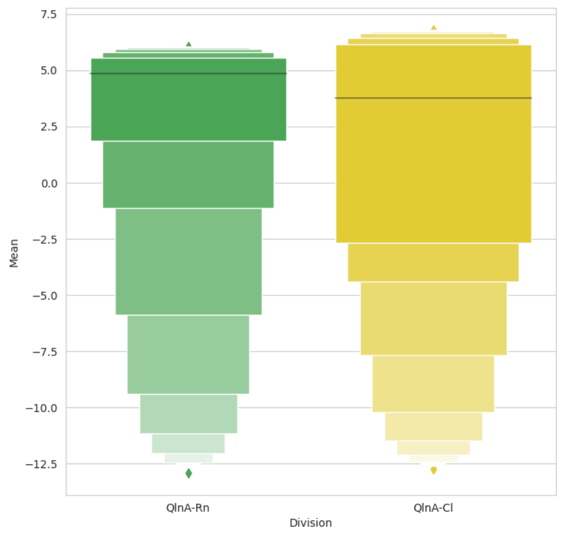


Figure 4.8: Aggression and Tightness, colored by division

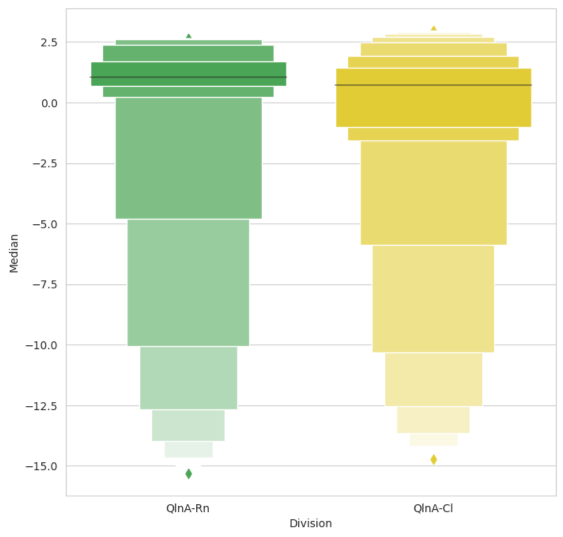
Compared with Figure 4.5, this plot is more mixed. In particular, the mixed division (AllAg-Cl) is spread across the entire plot, and each division specialized to an architecture covers the area of that architecture. This reinforces the concept that strategy space location depends mostly on architecture choice and not on population.



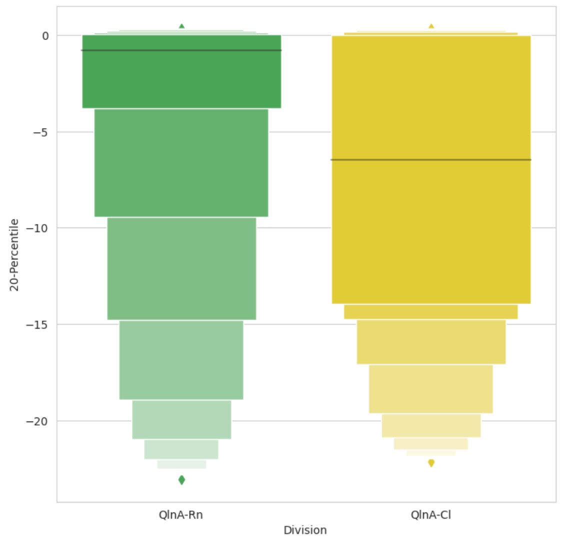
(a) Ranked by TrueSkill



(b) Ranked by Mean



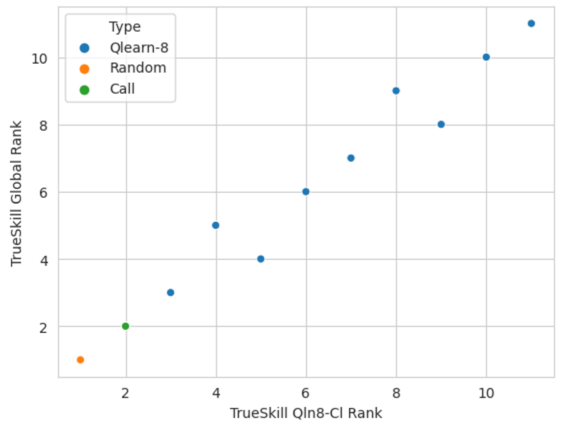
(c) Ranked by Median



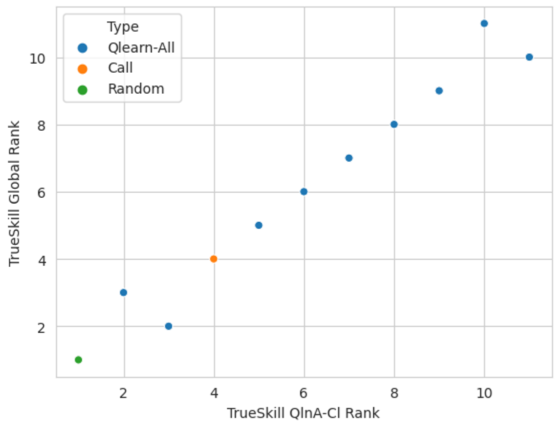
(d) Ranked by 20-Percentile

Figure 4.9: Agent distribution by matchup type

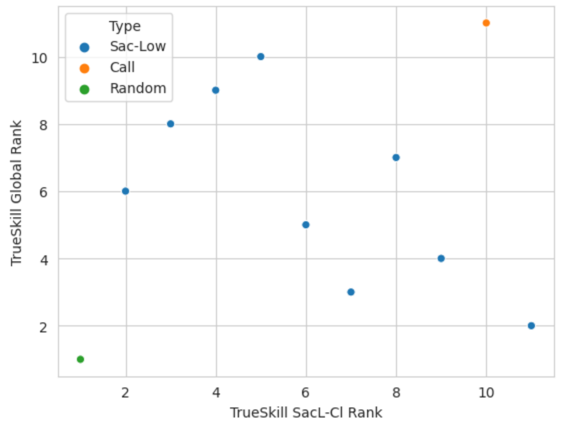
Taking a closer look at Climbing vs Random for the same agent architecture, it looks like Random is more robust in the final evaluation (against all agents), whereas the best Climbing agents earn more money.



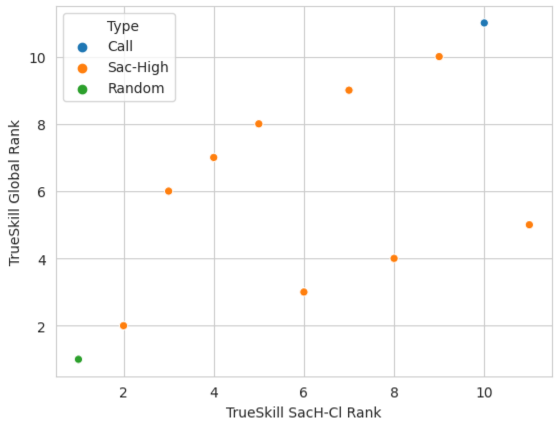
(a) Qln8-Cl



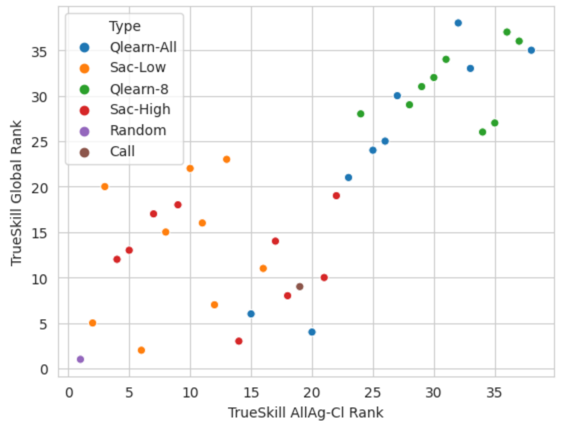
(b) QlnA-Cl



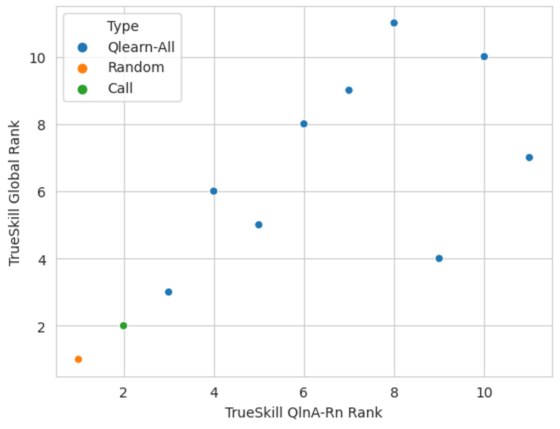
(c) SacL-Cl



(d) SacH-Cl



(e) AllAg-Cl



(f) QlnA-Rn

Figure 4.10: Internal ranking vs PermaEval global ranking by division, colored by agent type

To estimate how much agents in each division overfit, we compared the internal ranking of agents in each division with the global ranking of those agents. Ideally, these would be a straight line from bottom-left to top-right.

It seems very little overfit happens, as these results echo those of Figure 4.4, meaning that most variation here is expected simply due to TrueSkill seed.

One must recall Figure 4.3, and consider that internal rankings were computed parallel to training. This means the latest agents had only 200 rounds to establish their ranking, and even the oldest less than two thirds of the time compared to the global ranking.

SECTION 5

Conclusions

5.1 Populations

Contrary to our initial expectation, we find that different populations do not lead to visible differences in strategy coverage, and matter far less than agent structure (Figure 4.8, Figure 4.5).

Matchup generation does matter, but only marginally, and is a tradeoff; while climbing is better for mean gain, random creates more robust agents. This makes sense as agents in random matchups play against all agents equally, whereas climbing agents spend more time practicing against better agents. We also find that matchup generation (climbing vs random) does not influence training strongly (Figure 4.9).

5.2 Metrics

TrueSkill does not converge but gets close (Figure 4.3). Its consistency between runs depends on the agent type (Figure 4.4), being quite high even after few runs for Qlearn agents, and terrible for SAC agents even after 5'000 rounds.

The different metrics do not agree (Table 4.2), though some agents perform well in all of them. They favor different strategies (Figure 4.1), though not as much as expected.

5.3 Agents

SAC agents perform poorly (Figures 4.6 and 4.7), despite significant effort and experimentation in the early stages of this project. They also play inconsistently (Figures 4.4 and 4.10).

The action space of Q-Learning agents influences their behavior heavily, with Qlearn-All playing more aggressively than any other architecture (Figure 4.5). We believe this is because of how exploration is implemented; every action has a $\epsilon = 10\%$ chance of being replaced with a randomly selected non-folding action from the action space. The action space for Qlearn-All is composed of one call action and 199 raise actions, whereas Qlearn-8 uses one call action and only 7 raise actions. This means that Qlearn-All explores aggressive actions more than Qlearn-8, and more than other agents like SAC. A further experiment would be to weigh this exploration likelihood by action type, such that calling is explored as much as raising in total.

5.4 Future Work

Testing how Qlearn-All develops given a more balanced exploration between calling and raising is a simple change and could hold much insight.

Our experiments have only tested 2'000 rounds, with clones every 200 rounds. It is difficult to see whether this is capped out from the generation plots (Figure 4.7), but we suspect not and it would be interesting to see results of a much longer training period with an order of magnitude more agents, with fewer different leagues.

Another open possibility is using the strategy vector to filter a subset of agents that behave in a certain way and overfitting a new agent against them. This might create dedicated exploiter agents, whose behavior could give insight into the game itself.

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

s	A state
a	An action
S	A trajectory, i.e. a set of states that follow each other
π	A policy, i.e. a function that selects an action given a state
t	A time-step, i.e. a specific point in time during a trajectory
t_0	The first time-step of a trajectory
t_{end}	The final time-step of a trajectory
$r(s)$	The reward given for reaching state s
$R(S)$	The total reward (potentially discounted) of trajectory S
γ	The time-discount factor of rewards, i.e. how much a reward one time-step in the future is worth (generally a number like 0.99)
$Env(s, a)$	The next state generated by the environment if action a was undertaken at state s
$Term_\pi(s, a)$	The final state reached if an agent follows policy π after acting a from state s
$V_\pi(s)$	The value, or total reward, of following policy π starting from state s
$V^*(s)$	The value, or total reward, of following the optimal policy starting from state s
$Q^*(s, a)$	The value, or total reward, of following the optimal policy after acting a from state s
ϵ	The chance of acting randomly to allow exploration in Q-Learning

ψ	The polyak averaging constant
$P_{\pi,s}(a)$	The probability of policy π selecting action a in state s
α	The Soft Actor-Critic temperature, the relative weight of the entropy bonus to the reward
β	The "skill chain length" of TrueSkill, skill rating difference representing a 76% win likelihood
μ	The mean of a player's TrueSkill rating distribution, which is also assumed to be the player's actual rating
$p(Selected)$	Probability of an agent being selected in a climbing division

Bibliography

[1] D. Billings, D. Papp, J. Schaeffer, and D. Szafron, “Opponent modeling in poker,” AAAI Press, 1998, pp. 493–499.

[2] F. Southey, M. P. Bowling, B. Larson, C. Piccione, N. Burch, D. Billings, and C. Rayner, *Bayes’ bluff: Opponent modelling in poker*, 2012. arXiv: 1207.1411 [cs.GT].

[3] L. Teófilo and L. Reis, “Building a no limit texas hold’em poker agent based on game logs using supervised learning,” Jun. 2011, pp. 73–82. DOI: 10.1007/978-3-642-21538-4_8.

[4] M. Moravčík, M. Schmid, N. Burch, V. Lisý, D. Morrill, N. Bard, T. Davis, K. Waugh, M. Johanson, and M. Bowling, “Deepstack: Expert-level artificial intelligence in heads-up no-limit poker,” *Science*, vol. 356, no. 6337, pp. 508–513, 2017, ISSN: 0036-8075. DOI: 10.1126/science.aam6960. eprint: https://science.sciencemag.org/content/356/6337/508.full.pdf. [Online]. Available: https://science.sciencemag.org/content/356/6337/508.

[5] N. Brown and T. Sandholm, “Superhuman ai for heads-up no-limit poker: Libratus beats top professionals,” *Science*, vol. 359, no. 6374, pp. 418–424, 2018, ISSN: 0036-8075. DOI: 10.1126/science.aao1733. eprint: https://science.sciencemag.org/content/359/6374/418.full.pdf. [Online]. Available: https://science.sciencemag.org/content/359/6374/418.

[6] ——, “Superhuman ai for multiplayer poker,” *Science*, vol. 365, no. 6456, pp. 885–890, 2019, ISSN: 0036-8075. DOI: 10.1126/science.aay2400. eprint: https://science.sciencemag.org/content/365/6456/885.full.pdf. [Online]. Available: https://science.sciencemag.org/content/365/6456/885.

[7] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, “Mastering the game of go without human knowledge,” *Nature*, vol. 550, no. 7676, pp. 354–359, Oct. 2017, ISSN: 1476-4687. DOI: 10.1038/nature24270. [Online]. Available: https://doi.org/10.1038/nature24270.

[8] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev, J. Oh, D. Horgan, M. Kroiss, I. Danihelka, A. Huang, L. Sifre, T. Cai, J. P. Agapiou, M. Jaderberg, A. S. Vezhnevets, R. Leblond, T. Pohlen, V. Dalibard, D. Budden, Y. Sulsky, J. Molloy, T. L. Paine, C. Gulcehre, Z. Wang, T. Pfaff, Y. Wu, R. Ring, D. Yogatama, D. Wünsch, K. McKinney, O. Smith, T. Schaul, T. Lillicrap, K. Kavukcuoglu, D. Hassabis, C. Apps, and D. Silver,

"Grandmaster level in starcraft ii using multi-agent reinforcement learning," *Nature*, vol. 575, no. 7782, pp. 350–354, Nov. 2019, ISSN: 1476-4687. DOI: 10.1038/s41586-019-1724-z. [Online]. Available: https://doi.org/10.1038/s41586-019-1724-z.

[9] OpenAI, : C. Berner, G. Brockman, B. Chan, V. Cheung, P. Dębiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse, R. Józefowicz, S. Gray, C. Olsson, J. Pachocki, M. Petrov, H. P. de Oliveira Pinto, J. Raiman, T. Salimans, J. Schlatter, J. Schneider, S. Sidor, I. Sutskever, J. Tang, F. Wolski, and S. Zhang, *Dota 2 with large scale deep reinforcement learning*, 2019. arXiv: 1912.06680 [cs.LG].

[10] K. Shao, Z. Tang, Y. Zhu, N. Li, and D. Zhao, *A survey of deep reinforcement learning in video games*, 2019. arXiv: 1912.10944 [cs.MA].

[11] Wikipedia contributors, *Texas hold 'em hand values— Wikipedia, the free encyclopedia*, [Online; accessed 9-August-2020], 2020. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Texas_hold_%27em&oldid=971612259#Hand_values.

[12] ishikota, *Pypokerengine - poker engine for ai development in python*, 2016-2017. [Online]. Available: https://github.com/ishikota/PyPokerEngine.

[13] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction.* Cambridge, MA, USA: A Bradford Book, 2018, ISBN: 0262039249.

[14] A. K. Dixit, *Optimization in economic theory*, 2nd ed. Oxford Univ. Press, 2009, p. 164.

[15] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, *Playing atari with deep reinforcement learning*, 2013. arXiv: 1312.5602 [cs.LG].

[16] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, no. 3, pp. 279–292, May 1992, ISSN: 1573-0565. DOI: 10.1007/BF00992698. [Online]. Available: https://doi.org/10.1007/BF00992698.

[17] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, *Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor*, 2018. arXiv: 1801.01290 [cs.LG].

[18] J. Achiam, *Spinning Up in Deep Reinforcement Learning*, 2018. [Online]. Available: https://github.com/openai/spinningup.

[19] R. Herbrich, T. Minka, and T. Graepel, "Trueskill™: A bayesian skill rating system," in *Advances in Neural Information Processing Systems 19*, B. Schölkopf, J. C. Platt, and T. Hoffman, Eds., MIT Press, 2007, pp. 569–576. [Online]. Available: http://papers.nips.cc/paper/3079-trueskilltm-a-bayesian-skill-rating-system.pdf.

[20] J. Moser, *Computing your skill*, Mar. 2010. [Online]. Available: http://www.moserware.com/2010/03/computing-your-skill.html.

[21] A. E. Elo, *The Rating of Chessplayers, Past and Present.* New York: Arco Pub., 1978, ISBN: 0668047216 9780668047210. [Online]. Available: http://www.amazon.com/Rating-Chess-Players-Past-Present/dp/0668047216.

[22] H. Lee, *Trueskill python library*, Sep. 2018. [Online]. Available: https://trueskill.org/.

[23] W. Drevo, M. Saindon, and I. Hendley, *Treys - a pure python poker hand evaluation library*, 2013-2019. [Online]. Available: https://github.com/ihendley/treys.

[24] L. F. Teófilo and L. P. Reis, *Identifying playerś strategies in no limit texas holdém poker through the analysis of individual moves*, 2013. arXiv: 1301.5943 [cs.AI].

[25] T. Oliphant, *NumPy: A guide to NumPy*, USA: Trelgol Publishing, [Online; accessed <today>], 2006–. [Online]. Available: http://www.numpy.org/.