Confidence Based Poker Agents with Evolution Strategy Training - Group #23

Au Liang Jun, David Goh Zhi Wei, Law Yong Qiang Clinton, Lee Yiyuan, Tan E-Liang
National University of Singapore
{e0203163, e0148854, e0175317, e0175089, e0177355}@u.nus.edu

1 Introduction

Poker is a non-deterministic, imperfect information game as neither the opponent's hole cards nor unrevealed community cards can be observed by a player. Given the pervasiveness of imperfect information in real world problems, from investment strategy to fraud detection [Brown and Sandholm, 2017], solving imperfect information games through Poker can give us insights as to what approaches could be adopted to better solve these problems.

We first identified key components of the game state which form the bases for our heuristics. Next, we developed heuristic-driven players with different decision making frameworks, including minimax trees, linear evaluations and neural nets. Through separate methods of genetic algorithms and stochastic optimisation using CMA-ES, we achieved faster convergence rates for the optimal weights for these heuristics and consequently better performances for our trained agents.

We also engineered a framework that allowed us to achieve tremendous speed ups in the training process. Overall, we were able to produce agents that perform consistently well (top 4) in the online tournaments against the other groups.

2 Agent Design

2.1 Features

Our designed players utilised features to make decisions heuristically. The following features (x_i) were considered and incorporated in the players' evaluation functions.

1. Hand strength: The expected win rate of the player's hole cards, combined with the community cards, if available. This is estimated via the Monte Carlo simulation.
2. Pot amount: As the goal of the agent is to earn the largest amount of money possible, the amount of money in the pot should influence its decision making. This value is normalised by dividing the current pot amount by the maximum possible pot which we determined to be $64 \times small\_blind$.
3. Payout: This feature models the likely payout of the game through $hand\_strength \times pot\_amount$. [Billings *et al.*, 1998].
4. General Raises: Since we are unable to observe the opponent's hand, we incorporated the number of raises the opponent has made as a barometer of his hand strength.
5. Street: Advanced poker strategies exhibit slightly different behaviour for each of the different streets [Liley and Rakow, 2010]. Since the implications of raising differ depending on what street they occur on, instead of using a general weight for all raises, our agents use unique weights for raises on each street.
6. Aggression: It is possible for agents to develop a strong aversion to opponent raises, causing them to fold even if the opponent does not actually have a strong hand. Tracking how often opponents raises on average and injecting a bias to counter the aggression can help cautious agents perform better against aggressive opponents.

2.2 Linear Player

In this approach, decisions are made based on a confidence level calculated via a linear combination of features. In particular, for a current game state with feature values $X = (x_1, \ldots, x_N)$, we have

$$\text{Confidence}(X) = \text{Logistic}\left(\sum_{i=1}^{N} w_i \cdot x_i\right)$$

where w_i are weights assigned to each feature and Logistic is a suitable activation function to scale the result to $(-1, 1)$. A decision is then made against threshold values, in particular

$$\text{Decision}(c) = \begin{cases} \text{RAISE} & \text{if } c > rt, \\ \text{CALL} & \text{if } c > ct \text{ OR unable to Raise}, \\ \text{FOLD} & \text{otherwise}, \end{cases}$$

where c = Confidence(X), rt = Raise Threshold, ct = Call Threshold. Note that ct is not upper bounded by rt to allow for cases where $rt < ct$.

2.3 Minimax Player

We concurrently explored the use of adversarial search for decision making. In our implementation, the agent constructs a minimax tree at every call to `declare_action()`. The general structure of the tree is illustrated in Figure 1.

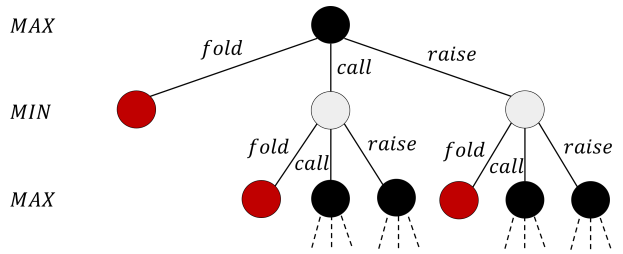

Figure 1: Structure of Minimax Tree

In the figure, the black nodes represent the max player, our agent, while the grey nodes represent the min player, our opponent. Each node in the tree holds information about the game state, used to generate child nodes and filtered to obtain features for evaluation. To reign in the branching factor, the different possible game states generated by the hidden cards are not incorporated in the form of chance layers. Instead, the opponent's hand and corresponding strength are only considered during the calculation of utility at terminal nodes.

Red nodes correspond to terminal nodes, which occur when either (i) the action taken to reach it is a fold, (ii) the player has no further valid actions, or (iii) an imposed depth cut-off is reached. At terminal nodes, utility is calculated as such:

$$\text{Utility}(s) = \begin{cases} \text{Eval}(s) \cdot \textit{pot\_amount}/2 & \text{if } \text{In\_Edge}(s) \neq \text{FOLD}, \\ -\textit{pot\_amount}/2 & \text{if } \text{Parent}(s) = \text{MAX}, \\ \textit{pot\_amount}/2 & \text{otherwise}, \end{cases}$$

where $\text{Eval}()$ is a modular function that can be modified to take on linear or neural functions, such as the Confidence() function of the Linear Player.

The edges in the tree represent actions taken by the player represented by the parent node. In our implementation, we utilise the *PyPokerEngine's* `Emulator` class to simulate how a particular game state changes when a specific action is applied.

3 Agent Training Methodology

3.1 Genetic Algorithm Based Training

Since our final assessment would involve a single agent playing against a series of opponents in 1-versus-1 matches, we decided to model our training to maximise performance in this setting. We felt that a genetic algorithm best suited our training needs of swift convergence in a large, complex search space and good performance against a population of similarly competent agents [De Jong, 1988].

Our implementation involved generating a large number of n bots and arranging every bot to play a 1-versus-1 match against every other bot, resulting in a total of $n \times (n-1)$ matches. The performance of a bot is determined by its overall win/loss ratio. Once all matches have been played, the weights with win/loss ratios > 1 are allowed to remain in training. All weights with win/loss ratios < 1 (i.e. more losses than wins) are removed and replaced with new weights. This technique allows us to optimise weights for multiple agents simultaneously.

Agent weight generation

Training a population of players with purely randomly generated weights raises a problem of efficiency as a majority of these weights result in weak players that are defeated by the most basic strategies. To accelerate convergence, we adopted a genetic method of weight generation. At the end of every training generation, the top performing weights are cloned and added into the training. Simultaneously, pairs of bots with high win rates are selected as parents to create a "child" with the mean of their weights. We reason that this genetic approach of weight production will result in a faster convergence than generating completely random weights while continuing to explore the solution space.

To increase training efficiency and consistency, we seed the training with standardised players such as *RaisePlayer*. This allows us to quickly cull weak weights as they would lose their matches against the seeded players. Also, it guarantees that the dominant players of the population will be able to beat standardised players. One potential disadvantage of this method is that we may be unable to escape a local minimum if the seeded players are too strong compared to the population. To mitigate this, we only introduce the seeded players after an arbitrary number of rounds to allow the randomly generated player population an opportunity to refine and evolve.

Evaluation of trained agents

The training cycle ends if the training exceeds an arbitrary number of generations in the interests of time. Once a cycle is over, we extract the best bots of the final generation and observe their performance against standardised players such as the *RaisePlayer* and *RandomPlayer* as a measure of objective performance. As we trained stronger agents, we began to measure newly trained agents against the best of the previously trained agents as a benchmark test.

Training evaluation

The progress of this training method is observed through the average standard deviation across each weights in the agents with a positive win rate, referred to as WinStdDev. A low WinStdDev indicates that the weights of winning agents are more homogeneous and is a sign of convergence. When WinStdDev increases sharply after a period of consistent decrease, we read it as a sign of overfitting. We reason that if

a population has begun to overfit, it will perform very well against other evolved agents, but lose badly against yet unseen randomly generated players. The only instance where $\mathrm{WinStdDev}$ increases suddenly is when a weight set that is very different from the current dominant weights begins to win suddenly, $\mathrm{WinStdDev}$, which causes the weights of winning players to be more varied. Thus, if $\mathrm{WinStdDev}$ begins to diverge after a period of convergence, we interpret it as a sign of overfitting.

3.2 CMA-ES Training

In CMA-ES [Hansen and Ostermeier, 2001], a single hypothesis of the optimal player is tracked in terms of a multivariate normal distribution in the weight space. At each generation, samples are generated according to the current mean and variance, and based on the fitness of each sample, the mean and variance of the tracked hypothesis are modified according maximum likelihood updates.

An interesting property of CMA-ES is that it only requires the *rank* of the samples rather than the absolute fitness values. In fact, the fitness values are discarded after determine the fitness ranking of the samples. As such, we only need to determine the relative performance among the sampled players in each generation.

Round robin matches are played among the samples, and the samples are ranked according to their win rate. In principle, we can build some suitable directed acyclic graph for the samples and determine a ranking via topological sort. However, this approach is unsuitable as cycles are typically produced due to the general lack of match transitivity.

4 Agent Training Supporting Infrastructure

We developed several tools and optimisation tweaks that allowed us to achieve extraordinary speed ups during training, as summarised in Table 1.

Measure	Speed Up
Parallelisation on SoC Compute Cluster	Ability to run up to 2300 games at once
Perform Monte Carlo simulation for Win Rate Estimation in C/C++	Reduced time taken by a factor of ~ 180.
Cython	Increases Python execution speed by about 22%

Table 1: Speed ups achieved from taken measures.

4.1 Parallelisation on SoC Compute Cluster

As the games between separate pairs of players are entirely independent, we were able to parallelise our training games across the *SoC Compute Cluster*. We engineered an architecture that could run multiple training algorithms and player designs simultaneously, thus allowing us to experiment widely and iterate quickly.

Games are supplied by a game server and processed by game clients spread across the 174 cluster nodes. The game clients are started with GNU Parallel [Tange, 2011] and communicate with the server over the network using the Eclipse Mosquitto [Light, 2017] implementation of MQTT, a lightweight and efficient machine-to-machine connectivity protocol. To ensure that we do not consume more resources than are known to be free and thus disrupt the tasks of other users, we scale our operations on each node based on their existing 5-minute load averages. Typically, our clients ran on anywhere between 1400-2300 CPU cores.

To illustrate the dramatic performance gains achieved, consider our internal evaluation tournament termed "Botlympics", in which our best agents play a total of 9000 games against each other, for a total of $9,000,000$ rounds of poker. On a single core, every 5000 rounds takes roughly 3 minutes to complete. A standard 4-core machine would require 22.5 hours to complete all the games, but 1400 CPU cores would take only about 20 minutes.

4.2 Native Win Rate Estimation

We re-implemented the win rate estimation calculations in C/C++ with the help of the OMPEval library [A., 2016] for hand rank evaluations. The bridging between Python code and native code was done using Python's `ctypes` library. This approach greatly reduced the time taken for 200 Monte Carlo simulations from ~ 49.5 ms to ~ 0.278 ms on average, providing a ~ 180x speed up compared to the original implementation in `pypokerengine.utils.card_utils`.

4.3 Cython

To further speed up the training process, we compiled our Python code using Cython [Behnel *et al.*, 2011]. In 5 trials of a series of 5 games with a total of 4500 rounds of poker, the regular Python code averaged 23.86s per trial, while the Cythonised code averaged 18.55s per trial, a speed up of approximately 22.3%.

5 Results

5.1 Agent Performance

Top Bots				
Bot Name	**Agent Class**	**Training Method**	**Games Won**	**Games Lost**
WXMG3d	Linear	Genetic	774	216
WXPr	Linear	Genetic	752	238
CMA2TP2	Linear	CMA-ES	337	653
CMA2TP3	Linear	CMA-ES	328	662
Minmax3	Minimax	Genetic	158	832
Minmax4	Minimax	Genetic	153	837

Figure 2: Win rates of selected bots in a round robin matchup. Each bot plays 30 games of 1000 rounds against 33 other bots.

Our experiments revealed that trained linear players performed the best, as it was the only agent design that was able to consistently defeat trained agents of different architectures

as well as generic players such as RaisePlayer. Figure 2 is an excerpt of the results of the internal tournament between all our trained agents.

Minimax Player

The trained Minimax Players did not perform as well as the linear agents. We attribute this to a number of factors:

1. Restricted viable weight space: There was a need to scale the utility produced by evaluation functions to the deterministic utility of folding-induced terminal states. Failure to do so may result in a player with irrational behaviour, such as always choosing to fold. This greatly narrowed down the viable weight space available, and as such a Minimax Player typically needed to train for more generations to achieve comparable performance with a Linear Player with the same evaluation function. Furthermore, each match a Minimax Player takes relatively longer because the agent must perform graph creation and traversal for every decision.

2. Poor time efficiency: Construction of the search tree was time-expensive even with $\alpha\beta$ pruning and action abstraction techniques. Despite the relatively small branching factor, a search cut-off had to be implemented in order to meet the specified agent reaction time. This limited the effectiveness of the minimax search.

3. Invalid assumptions: The minimax search hinges on the assumption that the opponent is playing rationally and evaluating utility identically, a belief that allows the establishment of a Nash equilibrium. If the opponent is not acting rationally, or obtaining different utilities from game states compared to our agent, the minimax search would not produce a sequence of actions that models reality.

5.2 Training

Genetic algorithm method

As mentioned in section 3.1, convergence is measured by the average standard deviation of each weight amongst winning bots or $WinStdDev$. Generally, the genetic training algorithm converged within 200 generations.

A recurring problem was the risk of divergence when the previous generation of trained players were seeded into the training. While an increased $WinStdDev$ is to be expected with the introduction of strong players, this can result in overfitting as bots over-optimise to defeat the strong players and lose against weaker players. This trend can be observed in the Linear Player statistics in Figure 3. In this instance, the trained players were injected in generation 200. Although $WinStdDev$ recovers from the large spike at around generation 240, it never managed to converge to the value of $\sim$0.025 that was observed before generation 200.

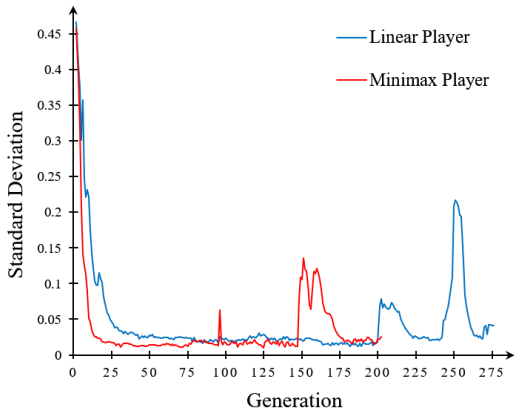

Figure 3: Convergence of Population-Based Training

CMA-ES

We keep track of the value of the step size during CMA-ES, which correlates directly to the variance of the tracked hypothesis for each weight. The algorithm converges at local minima before exploring other areas of the weight space multiple times (Figure 4).

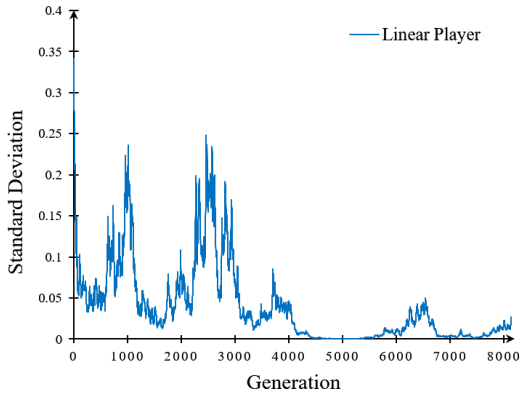

Figure 4: Convergence of CMA-ES Training

6 Conclusion

Our study finds that a trained linear player performs better than players with more sophisticated strategies, and our agent performed favourably in the online tournaments hosted by the CS3243 staff from March to April 2019, consistently ranking in the top 4. Furthermore, a genetic training algorithm is effective in performing reinforcement learning for the task, achieving convergence over a reasonable number of generations. We also attribute our agent's favourable performance to the efficient parallelisation of training, which enabled many variations of our agents to be trained across a large number of generations.

Appendix A Glossary

PyPokerEngine: A Python library that we utilised as a framework for our poker playing AI. See `https://github.com/changhongyan123/mypoker`

RaisePlayer: A PyPokerEngine agent that always raises when it can. See `https://github.com/changhongyan123/mypoker/blob/master/raise_player.py`

RandomPlayer: A PyPokerEngine agent that selects its action at random. See `https://github.com/changhongyan123/mypoker/blob/master/randomplayer.py`

SoC Compute Cluster: A collection of compute nodes provided by the NUS School of Computing. See `https://dochub.comp.nus.edu.sg/cf/services/compute-cluster`

Appendix B Neural Player

We initially attempted to extend linear players by using neural networks to compute $\mathrm{Confidence}(X)$ from feature values as a means of universal approximation to cover a larger class of possible mappings compared to just linear functions.

In the early stages of our project, the produced neural network based players generally performed better than linear players using the same base features. However, as more features were added, the number of weights in the neural network increased rapidly and the training became too infeasible for performant players to be produced.

References

[A., 2016] Timo A. Ompeval. `https://github.com/zekyll/OMPEval`, 2016.

[Behnel *et al.*, 2011] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D.S. Seljebotn, and K. Smith. Cython: The best of both worlds. *Computing in Science Engineering*, 13(2):31 –39, March 2011.

[Billings *et al.*, 1998] Darse Billings, Denis Papp, Jonathan Schaeffer, and Duane Szafron. Opponent modeling in poker. In *Proceedings of the Fifteenth National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence*, AAAI '98/IAAI '98, pages 493–499, Menlo Park, CA, USA, 1998. American Association for Artificial Intelligence.

[Brown and Sandholm, 2017] Noam Brown and Tuomas Sandholm. Libratus: The superhuman ai for no-limit poker. In *IJCAI*, pages 5226–5228, 2017.

[De Jong, 1988] Kenneth De Jong. Learning with genetic algorithms: An overview. *Machine Learning*, 3(2):121–138, Oct 1988.

[Hansen and Ostermeier, 2001] Nikolaus Hansen and Andreas Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary computation*, 9(2):159–195, 2001.

[Light, 2017] Roger A Light. Mosquitto: server and client implementation of the MQTT protocol. *The Journal of Open Source Software*, 2(13):265, may 2017.

[Liley and Rakow, 2010] James Liley and Tim Rakow. Probability estimation in poker: A qualified success for unaided judgment. *Journal of behavioral decision making*, 23(5):496–526, 2010.

[Tange, 2011] O. Tange. Gnu parallel - the command-line power tool. *;login: The USENIX Magazine*, 36(1):42–47, Feb 2011.