Final Report

By James Camacho and Joseph Camacho

Week 1

Our first idea was to try deep learning. As an initial test we trained a model to learn hand strengths, and it failed pretty miserably, so we abandoned the idea. A couple weeks later reading over the Deep Counterfactual Regret Minimzation paper, one line stuck out to us, "However, most popular RL algorithms do not converge to good policies (equilibria) in imperfect-information games in theory or in practice." Turns out our model had no chance of working, so it's good we quit without sinking the whole week into it.

Instead, we computed an approximate hand strength by fixing our hand and shuffling the remaining cards 100 times. Our probability of winning was just how often we won in this Monte Carlo search, however using only 100 iterations has a high variance. The exact variance is kind of hard to compute, but it's certainly less than that of a coin toss, or $\frac{0.5^2}{100}$. We subtracted off the approximate standard deviation (0.05) to make our model less prone to errors, as before the flop most hands have win rates close to 50%.

Then, once we have our probability of winning (p from now on), we used the Kelly Criterion to determine how much to bet. The break even point is when

$$\log(\text{bankroll}) = p\log(\text{bankroll} + \text{pot}) + (1-p)\log(\text{bankroll} - \text{bet}).$$

Taking a derivative and setting to zero gives us

$$\text{bet} = \text{bankroll} - \exp\left[\log(\text{bankroll}) - \frac{p}{1-p}\cdot\log(\text{bankroll} + \text{pot})\right].$$

If the minimum bet or continue cost is more than this, it doesn't make sense to raise or call. However, if we bet one *less* than this break-even point, we are expected to make just a little bit more than we put in. Or two less. Etc. (Ideally we could put zero in and win the pot.) If you consider the pot "free money" being auctioned off, then you can exploit the rules of the auction by betting just more than your opponent.

We naively assume our opponent has a chance $q = 1 - p$ to win with their hand, and then apply a similar formula to determine at what point they would *not* continue betting against us. This is a nonlinear equation (when we bet we increase their pot, which is nonlinear with their bet), so we use Newton's method. You can see this implemented here:

```
def kelly_opp(n, q, d):
    # Uses Newton's method.
    if q >= 0.5:
        return np.inf
    def f(x):
        return (1-q)*np.log(d-x)+q*np.log(d+n+x)-np.log(d)
    def fp(x):
        return (q-1)/(d-x)+q/(d+n+x)
    x = d / 2
    for i in range(10):
        x -= f(x) / fp(x)
    return x
```

Obviously when they have more than a 50% chance of winning there is no amount of money we can put in to make them fold. This function provides us with a lower bound on our betting. Put together with the previously computed upper bound gives us some decent rules on how to bet.

Week 2

In week two we mostly focused on making a better estimator for our hand's strength. Even after reducing the number of iterations in our Monte Carlo algorithm down to 100, we still had problems with running out. Furthermore, the standard deviation was unacceptably high (up to 0.05). We tried out a few things during this week to do this:

- Using a neural network to estimate hand strength (again, but with more data and a different teammate). The basic idea was we planned to generate a large number of possible hands, estimate their strengths with a Monte Carlo algorithm run many times, and then train a neural network on the data. It ended up being too much work to port the data generation to C++ (it ran too slowly on Python), so we gave up on this.
- Speeding up our Monte Carlo algorithm. This was much more successful. In Python, list comprehension is much faster than for loops are. By replacing for loops with list comprehension (and using a couple other tricks, such as only creating the deck of visible cards once), we were able to speed up our Monte Carlo algorithm by 20x. This gave us a much better estimate of the true hand strength.

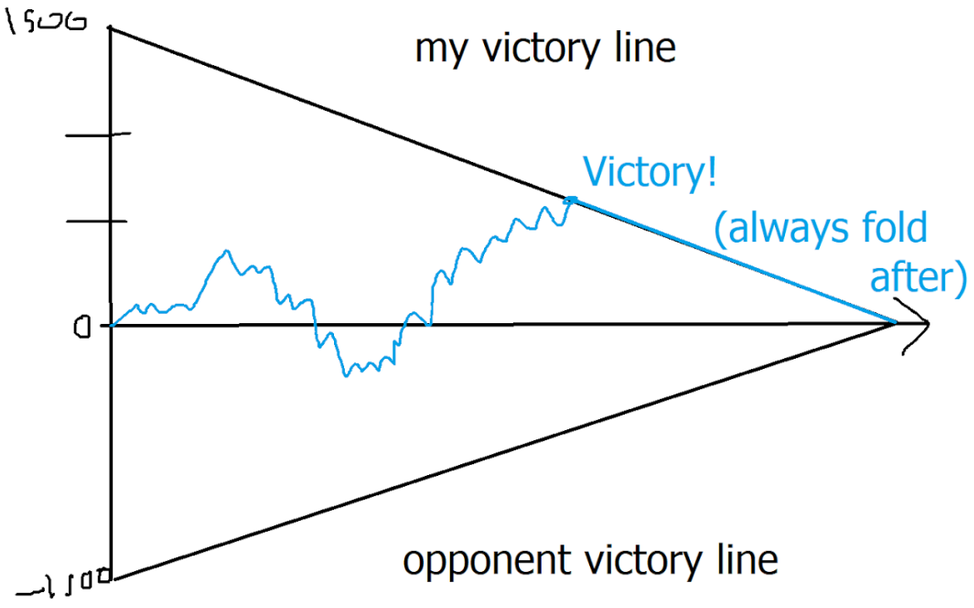
Another minor change we made was being more cautious if our opponent bet a large number of chips. If the opponent tripled the pot size in a given round, we divided our estimate of our chance of winning by two.

Finally, we noticed that the margin of victory wasn't important in calculating elos. So, we calculated a "winline" where even if our bot folded every remaining turn we would still win. (We actually had an off-by-one error, but we fixed that the next week.) We had lots of games where we won by a margin of less than one hundred chips, but a win is a win!

Double J	1382.0	baby ducks	1569.0	2023-01-20 20:59:00.909537	jj_2_0	Completed	63	-63	Double J	Log	Log
Double J	1349.0	baby ducks	1601.0	2023-01-20 20:58:56.024451	jj_2_0	Completed	13	-13	Double J	Log	Log
Double J	1343.0	baby ducks	1607.0	2023-01-20 20:50:18.059731	jj_2_0	Completed	2	-2	Double J	Log	Log
Double J	1309.0	baby ducks	1642.0	2023-01-20 20:29:20.450538	jj_2_0	Completed	32	-32	Double J	Log	Log

Week 3

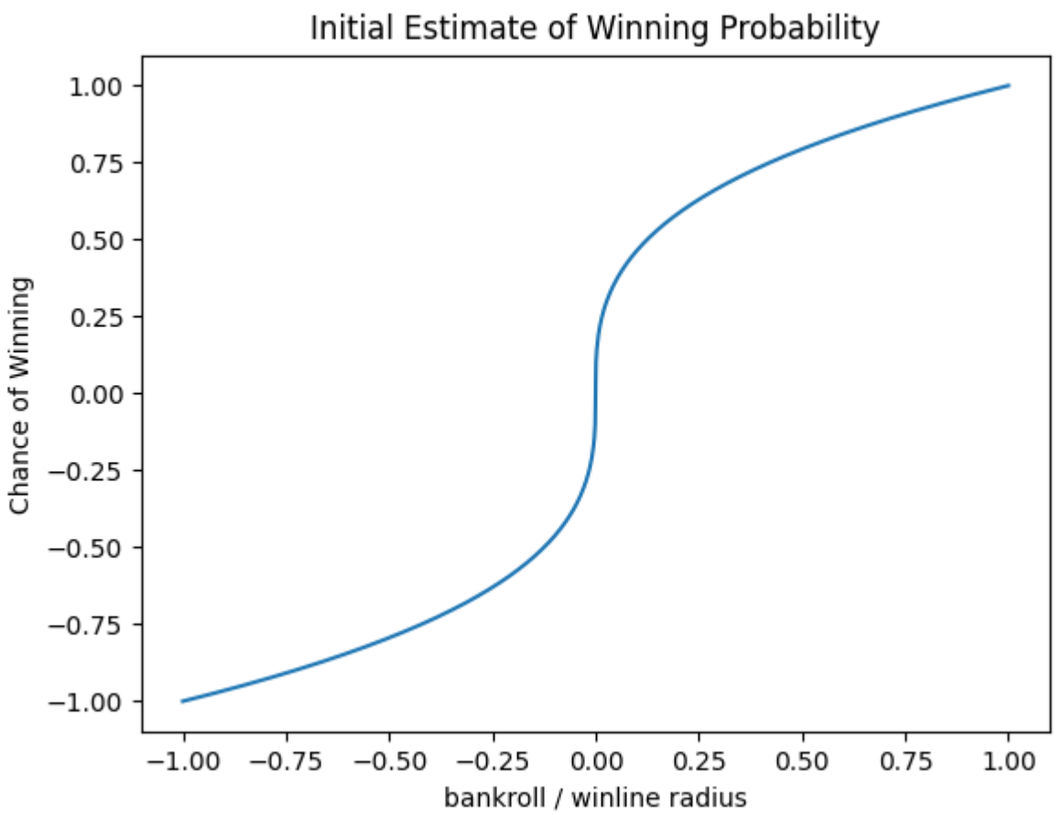
This week we switched away from the Kelly Criterion. It assumes utility works on a log scale, but that's a pretty bad model of this zero-sum game. There's a particular "winline", where once you have that many chips you can make it impossible for the opponent to recover. As the game progresses the winline narrows until just a few chips are worth tremendous utility.



The "activation function" we chose was

$$\sqrt[3]{\frac{\text{bankroll}}{\text{winline radius}}}.$$

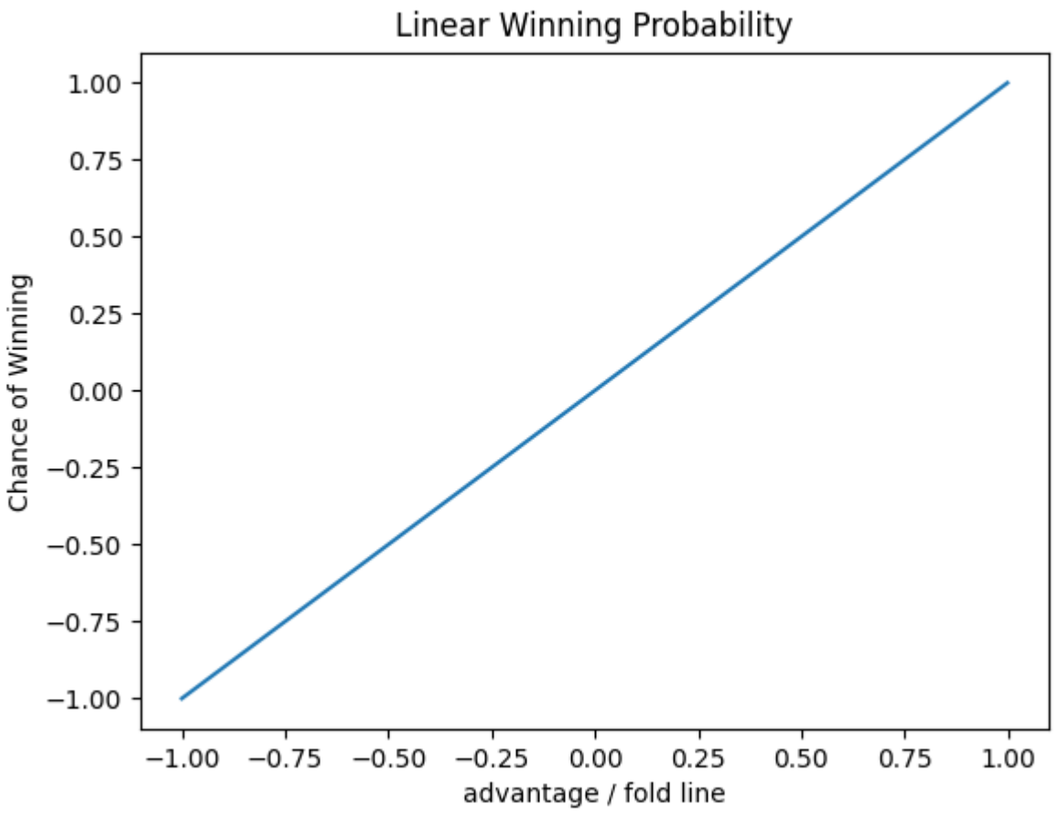
Here's a graph of it:



A convex function encourages riskier bets while a concave function encourages more conservative bets (by Jensen's inequality), so we wanted a function that was convex when we were losing and concave when we were winning, and this was basically the simplest function out there that satisfied this desire.

We then make the decision that maximizes our expected probability of winning given our estimated hand strength. There are hundreds of possible raise actions, so we bucketed them into a "raise minimum, raise maximum, and raise medium" cases, where the medium raise was the geometric mean of the first two. At first we naively assumed that the opponent would always match the number of chips we bet, though later we implemented a minimax algorithm.

While implementing the minimax algorithm, we ran into a problem. It would consistently go all in on the first couple turns, even when it had just a 45% chance of winning the hand. Eventually we figured out the problem: The activation function had such a steep slope at $\text{bankroll} = 0$ that even losing two chips decreased the estimated chance of winning to 43%. On the other hand, going all in basically guaranteed success or failure, so it would give closer to a 45% chance of winning. We switched to a linear model which fixed this issue:



This final minimax bot did end up beating the previous bot 75% of the time (in a trial of 40 games), but it ended up performing much worse on the scrimmage server. Our guess is many teams assumed going all in meant you had a really strong hand, when really in our case it meant you didn't have a terrible hand.

Conclusion

We were also doing the Battlecode competition this IAP, which we spent much more time on. But looking back, we had more clever ideas than it felt like each week. I guess necessity breeds innovation. It's much easier to come up with new models than explain why you failed to improve your bot :D. We had fun with PokerBots this year! It was nice to be able to come up with mathematical models and immediately see the results with only a few dozen lines of code.

Also, sorry for submitting this an hour late, we were talking about some interesting stuff and figured we only needed thirty minutes to write the report (apparently an hour would've been more accurate).

Work Review

James Camacho did a majority of the work in week one, and Joseph Camacho did a majority of the work in week two. In week three Joseph came up and implemented the idea to switch from the Kelly Criterion to a better model, and James implemented the minimax algorithm.