Using Reinforcement (Deep) Learning to play Bluff Poker

Aims: Educational – learn about reinforcement learning, get into ai. Targets – get a bot good at bluff poker, initially 1v1 with 1 card each (solved via game theory so good to see how close the bots can get) and then 1v1 with multiple cards (state space expands quickly). Stretch goal – train bot to play when > 2 players.

Why Reinforcement Learning?

My first thought

It’s cool

Want to learn reinforcement learning

It is possible other methods are better (Counter-factual regret minimization?)

Diary

Attempt 1: Black Box the Deep Learning

(pycharm project bluff\_poker)

19th Jan 2025

---

Got the chatgpt sub. Explained the rules of bluff poker to it in chat “Bluff Poker Program Framework”. Cooked a code implementation with (much) use of chatgpt. Idea was basically to have fixed strategy (tell truth strategy), and then see if could treat as much of the reinforcement learning as a black box as possible. As such, I don’t really understand how the reinforcement deep learning works \*at all\* lol.

I focused on how best to represent the game in vector form. Game state in reality consists of:

1. Net card(s)
2. History of moves
3. Strat card(s)

Notice that, technically we have a Partially Observed Markov Decision Process, since net doesn’t get access to strat’s cards. But since strat deterministic and unchanged, we can consider it part of the environment and use the history to logically determine which card strat must hold when in the middle of a training episode.

On the first move, strat has exactly 1/13 of each card. After strat has made at least one move, we know strat’s card precisely. E.g. strat: 2-high, net:7-high => strat must call since they have a 2 and 7 > 2.

---

Basic idea: Have network (2/3 layer neural network with lstm; chatgpt suggested all the architecture) and bluff poker environment. Train network through episodes; treat training, hyperparameters as black box.

For each episode:

Reset environment, get state = environment.start\_state // randomly deals cards

Till game done:

Action = Net(state)

Next\_state, reward, done = env.one\_step(action)

Record information in trajectory

Train\_net(trajectory)

This took a while to set up logistically, but conceptually is very easy.

---

Me actually thinking (non-black box):

Input: Net must receive (a) and (b) specified above. I decided to use a 1-hot 52-length vector for telling it which card it had. Note there are at most 13\*2=26 possible moves (any card high, any card pair) and since games were always at most length 4 long, decided to use 26\*4=104 more spaces to tell history (1 => that claim was made at some point, 0 => not, and order is implicit since claims must outdo each other).

I (much) later realized that increasing nature of claims means that in fact only need one set of 26, but didn’t implement this as I had moved on to later attempts. I used this idea going forwards, though.

State system: have a class called state. dynamically add new states to state, everything stored in one big dictionary. Did this over say a more traditional array approach since almost all states of the 2-player game are unreachable due to TellTruth acting so inflexibly. In hindsight, should not have worried about memory and focused on simplicity of code, since undoubtedly this added code bloat.

Rewards: 1 if net wins, -1 if it loses and 0 if game continues. Later tried “looking backwards” so every decision would get +- 1 depending on if net won or not. Also tried giving asymmetric rewards to encourage network to find better strategies. Nothing worked too well, except (10,-1,-3) was pretty effective at getting partial learning (described below in Results).

---

Results: I literally put the program in debug mode and would see how it fared with specific starting hands. I realized it was learning nothing so decided instead to always guarantee net got better hand (reward signal clearer since it can always win). It did eventually find the (almost-optimal) strategy of calling its own hand, but it struggled to match everything to everything. E.g. would only relate the jack of clubs to calling jack high but no other jacks, and would completely fail on some cards. Decided to make it easier by reducing input to just 13 for cards (i.e. enforcing the fact that suits don’t matter); this improved policy but not by much, still only had around a 50% success rate. Note this was also against TellTruth that never said pairs, making the task even easier.

Discouraged by lack of full learning, but happy that at least some learning occurred, decided to restart and try again (code bloated).

---

Attempt 2: Black Box the Deep Learning again

(pycharm project bluff\_poker\_2)

21st Jan

With attempt 1, code was half the time written in strings (“JH” = jack high, etc) and half the time with integers (10 = jack high, etc). Maybe this was causing bugs somewhere / slowing down training? I wasn’t too optimistic here to be honest, since how much of a speedup could I realistically get from this process, but hoped to at least spot bugs more easily henceforth.

Long story short – new network could barely learn \*anything\*. I tried reducing number of cards; only when there were 3 or 4 cards in the deck (as opposed to 13) could it learn anything. This was extremely surprising, since I used the exact same black box architecture and hyperparameters as for Attempt 1. I initially suspected there was some atrociously bad bug, but the fact that network learned when reducing number of cards suggested there wasn’t an obvious quick fix. Pretty disillusioned with the black-box approach at this point.

Forgave myself a little bit after reading this: <https://www.alexirpan.com/2018/02/14/rl-hard.html>

A close-up of a text

AI-generated content may be incorrect.In particular:

Decided to actually learn the black-box, and understand reinforcement learning.

---

Interlude: David Silver Reinforcement Learning, UCL Youtube Course

Watched first three lectures – since at this point I’ve studied Markov Chains before at university, was pretty easy. Understood various Bellman equations.

Realized probably best to have gamma = 1 with Bluff Poker, at least initially, since episodes so short.

---

Attempt 3: Planning via DP against TellTruth

(main.py, dp\_bluff\_poker pycharm project)

24th Jan

Implemented planning as described in Silver’s lectures to actually fully solve the 1v1, 1 card each version of BP against TellTruth. Used value iteration since this seemed simplest.

Notes:

* Quite hard with current implementation, but not in theory, to do so against arbitrary strategy and not just TellTruth.
* Only 2 iterations required, highlighting simplicity of the game.
* Probably not scalable? I.e. even in 2-player, if multiple cards then things get really big really quick.

---

Attempt 3.5: Planning via DP against fixed arbitrary strategy

Tried redoing this for arbitrary strategy. The issue comes with the fact that state does not encode the information for strat’s cards, and the chance strat has each of the 13 cards has to computed from scratch. In theory possible – what I tried to do was figure out probability that strat has card x given the history that had occurred – but implementation went wrong.

Notes:

* Definitely possible, just coding skill issue lol.
* Also not that deep.

---

Attempt 4: Model-free learning

~25th-30th Jan

(monte\_carlo\_against\_fixed.py, dp\_bluff\_poker pycharm project)

Continued following Silver’s lectures, watching lectures 4 and 5. Thus decided to implement monte carlo at first, since conceptually simplest. Also decided to do essentially all coding by myself rather than with chatgpt since needed to fundamentally understand what was happening. For this, need to have a big lookup table q\_space(state, action) that we update as episodes happen in a training loop, and n\_space that counts how many times we visited each (state, action) pair.

States: [N + 1][action\_space\_size(N, K) + 1][action\_space\_size(N, K) + 1)]

* First index is card (with last spot being reserved for done states and are always 0).
* Second index is last\_move. Is 0 for start of game, then for K=1 there are 2N+1 possible moves (card high, card pair, call) so e.g. has size 28 when N=13.
* 3rd index is action to take. Actually, impossible to reach any state where 3rd index is 0, so could have 3rd index have sized just action\_space\_size(N, K), but for consistency between 2nd and 3rd indices have it as so. Then represented same as last\_move.

The main Monte Carlo loop runs as so:

glie\_monte\_carlo:

for each episode:

net\_cards, strat\_cards = deal\_init\_cards(n)

states, actions, reward = sample\_episode(net\_cards, strat\_cards, epsilon)

for action in actions:

n\_update(state, action, reward)

q\_update(state, action, reward)

Reward: +-1 for win / loss, applied to every action in a given episode.

A math equations on a white background

AI-generated content may be incorrect.Q\_update: With G\_t as described in the Reward section just above:

This worked pretty well, and I could tell learning was occurring by pausing code via debugging mode and seeing what q\_space looked like. Decided needed some visuals, and here employed chatgpt to generate two types of visuals.

---

A graph of multiple loss curves

AI-generated content may be incorrect.Type 1: Loss over time. Since we knew the theoretically best strategy against TellTruth, could just figure out what that should be, test our strategy in every situation by iterating over all pairs of cards, and then plotting.

A screenshot of a graph

AI-generated content may be incorrect.Type 2: Visualising q\_space and n\_space to see whether exploration rate was ok, etc.

---

Now path forward was clear:

* see how tuning hyperparameter affected learning (both for fundamental understanding and with an eye to increasing number of cards)
* improve algorithm: SARSA vs Monte Carlo
* evaluate changes via loss curves and examining q\_space, and thus makes sure changes are not random.

What are we looking for?

Obvious strategy when going first is to say your own card-high. This strategy normally very easy to find. What is hard is realizing that “bluffing” by going lowest-high and thus learning strat’s card is optimal. This takes some trial and error. Note this would not work if strat’s strategy could change, since they would start calling more often on first move, but we have fixed their strategy to be TellTruth.

Further note: network essentially learned each starting card as its own subgame.

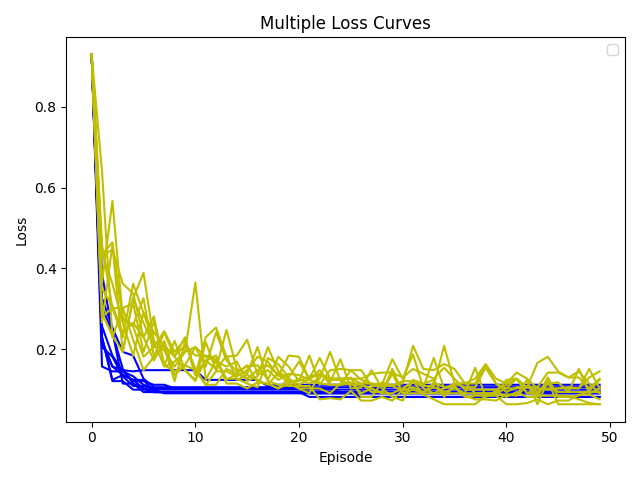
Hyperparameters – Epsilon.

* Started with constant epsilon to see behaviour.
* In general, higher epsilon = faster initial learning, but also long term higher min\_loss.
* Smaller epsilon = slow convergence, but long term is better.
* Therefore, seems sensible to decay epsilon over time.
* Could do something like 1/episode, but decided to keep it simple: epsilon = 0.3 for first 10% of run, then 0.1 for 80% of run, and 0.05 for final 10% of run.
* Will have to revisit as other hyperparameters and underlying update changes.
* In shown graph, 50k episode, mc method, using n for moving average.
* Epsilon values: red=0.5, blue=0.3, green=0.2, orange=0.1, cyan=0.075, purple=0.05, black=0.025. Black would clearly go lower with more episode, whereas red, blue, green have stabilized.

A graph of a graph

AI-generated content may be incorrect.

Hyperparameters – Alpha.

* From looking at some type 2 visuals, it seemed like (for given starting card) with the monte carlo update, either net found the tt-exploit early or not at all. I hypothesized that this was due to early runs being too important in training.
* Thus makes sense to use an alpha to constantly nudge q-values around.
* Key questions: does this work? If so, what value of alpha?
* Answer 1: idea seems sound. Graph below shows 10 runs of monte carlo using average return (n\_space) in blue and 10 runs of monte carlo using alpha=0.1. We notice slower convergence, more instability but better mins than monte carlo. Again, decaying schedule makes sense.
* Answer 2: decaying schedule. Picked alpha 0.3 for a while, then 0.2, and finally 0.1. Could decrease even further, but this seems good.
* In graph below, blue is running average baseline. alpha values: red=decaying as described above, green=0.3, cyan=0.2, magenta=0.15, yellow=0.1, black=0.05. (average of 10 runs each, 50k episodes).

A graph of different colored lines

AI-generated content may be incorrect.

Key insight from Type 2 visualizations: seems like, under monte carlo, network can learn to tt-exploit for lower initial cards much better / more easily than for higher cards. This might be because with low card you lose very often anyway, so maybe network tries other options more i.e. by random chance the good-enough strategy becomes low and then network finds the tt-exploit strategy. This is in opposition to higher cards, where the good-enough strategy is just much better than random, so maybe network “locks in” these decisions early.

30th Jan

SARSA / Q-learning

Decide to move away from monte carlo and try other algorithms, starting with SARSA. Idea: instead of giving +-1 to each action, give +-1 to last action (almost certainly correct) and then update previous q(s,a) based on \*perceived own value\* of next state at some weighting alpha. This ends up having effect of updating early states less, encouraging more exploration there (I think? Not sure, just sure that it empirically works).

Action

Action

Action

A graph of multiple loss curves

AI-generated content may be incorrect.Actually, I ended up using a pure greedy process to figure out perceived value, inadvertently implementing q-learning instead of sarsa. Illustrates how rewards at end of episode only => these two methods quite similar to each other.

Graph: red = mc, alpha=1/n. orange=mc, alpha=decaying (described above). Blue=sarsa, alpha=1/n. cyan = sarsa, alpha=decaying (described above). 5 runs each, 50k episodes.

As can be seen in above graph, SARSA / Q-learning is really good. Sometimes almost hitting theoretical optimum of 0 (did actually literally hit 0 in a few runs), and on average a lot better than either monte carlo. Some room for hyperparameter optimization, as done before, particularly to deal with noise as can be seen.

31st January

Couple thoughts today:

* Streamlined process for creating graphs, and constructed baselines to compare future improvements to
* “Backwards” q-learning (where you update ont the last action of an episode first) didn’t do anything different; I guess makes sense tbh since most of the data is from previous episodes given short length of episodes.
* Tried messing around with hyperparameters a bit; interestingly, can visually see when various hyperparameters change if using a discrete change:
* A blue line graph with white text

  AI-generated content may be incorrect. Clear boundaries at 2k eps (task gets hard), 6k (=0.3, alpha change), 14k (=0.7, alpha change, and 18k (0.9, alpha and epsilon change).
* Best alpha schedule found is to vary alpha not on episode but actually on how many times we have visited the particular state, action pair (makes sense, if we are visiting node for first time we should aggressively change its value but less so if we have visited it a lot).
* Best epsilon -> 0 which is funny; I think this is because starting state (all 0s) is pretty close to optimal (evidenced by fact that, when initializing with random noise, cannot reach nearly as good results).
* Then tried increasing N, number of cards. All the hyperparameter-optimised algorithms do badly, since epsilon too low, but I think continuous alpha did actually help.
* Strangely, sarsa is slower to converge than mc, but converges to lower limit (just takes more episode as N increases though).
* Also read about some MARL algorithms

1st February

Thoughts

* Tried coming up with principles for hyperparameters that should extend to different situations (looking towards increasing number of cards and/or MARL systems).
* First tried comparing standard epsilon choice (haphazardly picked decreasing piecewise constant) to “epsilon minus one” (continuous decreasing, goes like 1/1+x).
* A graph showing a line of loss curves

  AI-generated content may be incorrect.Found it did not matter:
* Now optimizing for what types of alpha are good.
* Noticed odd bump in graph above. Thought about it a little, realized that I had unintentionally been biasing the system to prefer earlier numbers because in case of tied value functions I always chose the first action that lead to that value.
* This obviously doesn’t generalize, but happens to more easily lead to optimal outcome for this particular setup. Thought about other bias: initial values in q\_space are all 0.
* From now on, pick randomly from all best actions in greedy\_choice() and also initialize weights randomly. This makes it much harder to find any tt-exploits sadly :( but it is more realistic.
* Hump is now gone! Also turns out that standard alpha choice is quite bad, and even alpha minus-1 choices not great, but sqrt(1/visited\_num) works really well.
* Strangely, algorithm sometimes finds tt-exploits, but also sometimes fails to find the proper continuation (varying over starting card). I think this is basically because if net gets unlucky and the response to saying net-card high after saying 0-high starts off with low value, it can be difficult to ever explore this option, and thus it doesn’t do too well.
* Tried varying epsilon again and varying power in 1/visit num and varying cutoff (i.e. sqrt(1/visit\_num)+0.05) but nothing did much.
* BUT --- increasing episode length back up to 100k actually lead to more improvement, so just had to train for longer (consistently hit 0 loss!). This only works as long as we continue to have alpha as we progress, though (obviously).
* A graph of a graph

  AI-generated content may be incorrect.A graph of multiple loss curves

  AI-generated content may be incorrect.Established 3 benchmarks (for both 20k and 200k eps): alpha\_minus\_1 with sarsa, alpha\_sqrt\_visit\_num with sarsa, and average with monte carlo:
* Bit strange that sqrt\_visit does better at 200k but worse at 20k, but realize that this is probably because you want your alpha schedules (and epsilon schedules, I guess) to be independent of episode length; i.e. first 20k steps should look the same in all cases, but currently doesn’t. Could find more params for this case but these 2 benchmarks good enough for now (mc trash, maybe TD-l good but not implemented yet).
* If using constant alpha, looks like 0.1 is best? But obvs should decay with time… same overfitting issue.

2nd Feb – CUAI Hack day

* Basically took whole day to understand fictious self-play paper (<https://www.davidsilver.uk/wp-content/uploads/2020/03/fsp.pdf>)

Around 4th feb – tried a refactor, didn’t really work :(

22nd Feb

* Refactor for real
* Got confused about not being able to replicate eval curves, realized this was due to changing to uniform start from net always starting.
* Revamped visualizations to actually be good.
* Storing past q\_spaces properly (npz file)
* Able to now develop strategies AGAINST npz files, so can set up a loop
* Next: do the randomization mix as described in self-fictitious play paper, rather than pure exploit. Should get an optimal 1-card player.

2nd march (CUAI Hack day)

* Spent whole day setting up loop + extra visualizations
* Did end up getting it to work in the end, after lots of effort, but the loop didn’t even get close to converging…
* Will need a refactor to make things cleaner and experiment with loop dynamics.

6th March (and a few days prior)

* Did the proper refactor (folder: setting\_up\_cycle), so file storage is proper now :)
* Set up the loop system which did the exploit + mix cycle properly.
* Figured out what was causing lack of convergence: needed eta (mixing parameter) to go to 0 really quite quickly, e.g. 1/n, while I was using a small constant.
* This WORKED!! Remarkably well to be honest.
* The cycle is just
  + Start with initial strategy; can be random or tell\_truth or previously found probabilistic strategy
  + Find its deterministic best exploiter
  + New\_strat = old\_strat \* (1-eta) + best\_exploiter \*(eta)
  + Repeat
* Obviously, lots of hyperparameter tuning possibilities. Most immediate concern is how to efficiently get exploitability, i.e. how to do step 2 quickly. Currently, training X agents for N steps via RL and then evaluating current strategy against them all, taking best one as the best exploiter and incorporating its probs.
* Typical values are X=5,10 and N=20k, 50k or 100k.
* Unsurprisingly, as strategy gets better, takes longer to exploit it. E.g. below graph shows how with N=5 exploiters do against a relatively mature strategy (lots of cycles of mixing).
* Clearly 20k steps is not enough; the exploiters can barely crack 0.1 (note 1 = current strat wins 100% of time on average, 0 = 50% chance of winning. All best exploiters in theory should lead to values <=0, and at nash equilibrium no exploiter should be able to crack 0. Of course randomness means even theoretically at nash there would be occasional runs with exploitability crossing 0).
* Seems like 100k steps is about optimal\* (dangerous word here lol).
* A graph with colorful lines

  AI-generated content may be incorrect.Have decided to use RL hyperparams that were chosen based on what worked at the very beginning, against tell-truth; this may need to be revisited, but there is a very large hyperparameter space.
* Should also be wary of current paradigm: strategies only get access to just the last move (and their card, of course). This is a sensible thing to expect if game lengths are short, but it is possible that the nash equilibrium in the restricted set of strategies is not a true nash equilibrium!
* TODO:
  + Visualizations! Reuse code from before.
  + Maybe try to adjust hyperparam for RL, not a priority though
  + Mini-goal: Try to get a strategy that is almost certainly at least twice as good as tell\_truth, i.e. operates at 48.5% = loss of -0.03.
  + Then, onwards to more cards + memory. Probably using linear features to start.

~ 10th March

* Implemented visualizations, and can see optimal strategy for n=3:

A yellow and purple squares

AI-generated content may be incorrect.A yellow and blue squares

AI-generated content may be incorrect.A yellow and blue squares

AI-generated content may be incorrect.A scale with a number

AI-generated content may be incorrect.

* Way to read visualization: each graph is what to do with each starting card (1,2,3). Each row is showing what to do when opponent has just done something. E.g. first row of first table says what to do if starting, and having card 1. Rows go start, 1h, 2h, 3h, 1p, 2p, 3p. The columns are probability of doing that action. So e.g. if starting and you have a 3, mix ~50-50 between saying 1h and 3h (interesting?)

17th May

* Mid exams, but final (?) CUAI session
* Starting fresh in new file, using github.
* Much better documentation so I get what’s going on lol.
* Trying linear function approximation, n cards. For now just using high card, pairs and trips for simplicity.

Appendix

Bluff Poker Rules: TBW (can see chatgpt history from when I explained rules to it).

Network and Strategy: We always call the fixed strategy we are playing against (normally TellTruth) the “strat” and the policy we are trying to learn “network” or “net”.

TellTruthStrategy (1v1, 1 card each). If last move by opponent is lower than the card the strat has, then strat says “their-card” high. If last move by opponent is higher than the card strat has, it calls; in particular, calls any pair. If last move is exactly strat’s card, goes “their-card” pair (and subsequently calls anything higher if the game gets there).

Strategies against TellTruth. Net can guarantee win if given strictly higher card by just claiming net-card high (“good-enough” strategy). This is an almost optimal strategy, since if you have the strictly lesser card you haven’t got many options. However with lowest card may as well guess something since 0-high guaranteed to lose. And a super-exploitative strategy (“tt-exploitative”) is to always say 0-high with any card other than a 0 since then TellTruth exposes their own card at which point net can win with an equal or higher card (thus winning all pair situations).

MARL. Multi-agent reinforcement learning. BP is a 2-agent setting.