# Playing a toy poker game with Reinforcement Learning

Jun 6, 2017

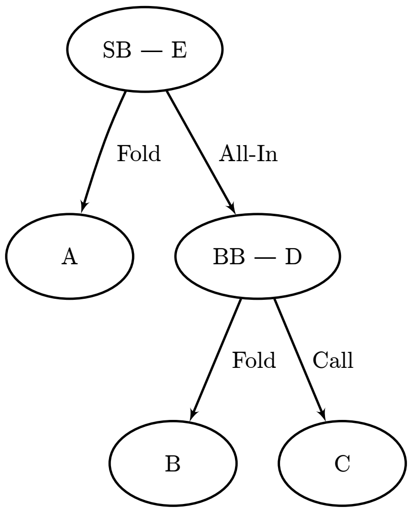
Reinforcement learning (RL) has had some high-profile successes lately, e.g. AlphaGo, but the basic ideas are fairly straightforward. Let's try RL on our favorite toy problem: the heads-up no limit shove/fold game. This is a pedagogical post rather than a research write-up, so we'll develop all of the ideas (and code!) more or less from scratch. Follow along in a Python3 Jupyter notebook!

## Problem setup

A quick reminder – shove/fold is a 2-player no limit hold'em game where:

1. Both players start with stacks of and a randomly-dealt 2 card hand.
2. The BB player posts 1.0 blind, and the SB player posts 0.5 blind.
3. The SB can go all-in or fold.
4. Facing an all-in, the BB can call or fold.

We might visualize this as a decision tree like the one shown here. A hand beings in where the SB can shove or fold. If she folds, we transition to , and the hand is over. If she shoves, we end up in where the BB must decide between calling and folding. If one player folds, the other captures the blinds, and if both players get all-in, a 5 card board is dealt and payouts are given according to the normal rules of poker.



The solution for this game is well known, and we've looked at other approaches elsewhere, e.g. fictitious play and direct optimization. Here, we'll estimate the solution using RL.

Now, there are distinct starting 2-card hand combinations in hold'em. We can thus order all the hands and number them from 0 to 1325. The specific order won't matter as long as we're consistent. The following function implicitly defines such an ordering and creates a map from hand number back to the strategically-relevant info: card ranks and suitedness.

```
numHoldemHands = 1326  # nchoosek(52,2)
ranks = ['2','3','4','5','6','7','8','9','T','J','Q','K','A']
suits=['c','s','d','h']
numRanks = len(ranks)
numSuits = len(suits)

# Input: N/A
# Output:
#   A map from int hand representation in [0,1235] to tuple of form
#   (rank1, rank2, isSuited).
def makeIntToHandMap():
    result = [0 for i in range(numHoldemHands)]
    c = 0
    for r1 in range(numRanks):
        for r2 in range(r1, numRanks):
            for s1 in range(numSuits):
                for s2 in range(numSuits):
                    if r1 == r2 and s1 >= s2:
                        continue
                    # hand number c corresponds to holding
                    # ranks[r2], suits[s2], ranks[r1], suits[s1]
                    result[c] = (r2, r1, s1 == s2)
                    c += 1
    return result

intToHandMap = makeIntToHandMap()
```

Notice that the first entry in the output tuples (`r2` in the code) is always the higher rank, if there is one. For example, hand number 57 happens to be 6♦2♣, and we have:

```
>>> r2, r1, suited = intToHandMap[57]
>>> print(ranks[r2], ranks[r1], 's' if suited else 'o')

6 2 o
```

When the players get all-in, the amount of the pot they capture on average (their "equity") is given by the rules of the game. The file pf_eqs.dat holds a numpy matrix `pfeqs` (see numpy.savetxt) where `pfeqs[i,j]` is the equity of hand when the opponent holds hand .

Of course, sometimes two starting hands have a card in common, in which case they can't both be dealt simultaneously, and it doesn't make sense to ask for their equities. The file pf_confl.dat holds another matrix where every entry is either 0 or 1. A 0 indicates that the hands conflict, and a 1 indicates that they don't.

```
import numpy as np
pfeqs = np.loadtxt("pf_eqs.dat")
pfconfl = np.loadtxt("pf_confl.dat")
```

For example, since hand 56 is 6♦2♣, 57 is 6♥2♣, and 58 is 6♣2♠, we have:

```
>>> pfconfl[56,57]
0.0

>>> pfconfl[57,58]
1.0

>>> pfeqs[57,58]
0.49655199999999999
```

Why isn't that last result precisely 0.5, by the way?

## Reinforcement learning

Now – a crash course on RL. There are three important components to an RL problem: state, action, reward. They fit together as follows

1. We are in some **state** (i.e. the state of the world, which we observe).
2. We use that info to take some **action**.
3. We get some **reward**.
4. Repeat.

We do this over and over again: observe state, take action, get reward, observe new state, take another action, get another reward, etc. The RL problem is simply to figure out how to choose actions to get as much reward as possible.

This turns out to be a pretty general framework. Lots of problems can be thought of in this way, and there are lots of different approaches to solving these problems. Generally, solutions involve wandering around, choosing various actions in various states, remembering which rewards were obtained, and then trying to do something smart with that info to make better choices in the future.

How does this apply to the shove/fold game? At any decision point, the player knows her hole cards and the position she's in. This is the state. She can then take an action: either FOLD or GII. (GII is "get it in"! For the SB, GII means shove, and for the BB, GII is a call). And then rewards – this is money won, and we'll use the players' total stack sizes at the end of the hand. For example, if the initial stack size is , the SB shoves and the BB folds, then the players' rewards are 11 and 9, respectively.

We'll find strategies for playing this game by simulating a bunch of hands. We'll deal both players some random cards, let them make decisions about how to play, and then observe how much money they end up with at the end each time. We'll use this info to learn (estimate) a function . takes in a description of the state and an action and outputs the value of taking that action in that state. Once we have (or some estimate thereof) , making strategy choices is easy: we can just evaluate each of our options and see which one is better.

So, our job here is to estimate , and we'll use (pronounced "Q hat") to refer to this estimate. We'll start with some random initial guess for . Then, we'll simulate a bunch of hands where both players make decisions according to . After each hand, we'll adjust the estimate to reflect the actual values the players got after taking particular actions in particular states. Eventually, we should end up with a pretty good estimate, which, again, is all we need to determine the players' strategies.

There's one wrinkle here – we need to make sure to take all the actions in all the states at least occasionally if we want to end up with good estimates of each possibility's value. So, we'll have the players act randomly some small fraction of the time but otherwise use their (currently-estimated) best options. At first, we should explore our options a lot, frequently making a random choice. As time goes on, we'll opt to exploit the knowledge we've gained more often. That is to say, will shrink over time. There are lots of ways to do this. Here's one:

```
# Input:
#   nHands: total number of simulations we plan to run
#   i: current simulation number
# Output:
#   Fraction of the time we should choose our action randomly.
def epsilon(nHands, i):
    return (nHands - i) / nHands
```

happens to be called *the action-value function*, since it gives the value of taking any particular action (from any state). It plays a central place in most RL methods. How exactly might we represent ? Evaluate it? Update it after every hand?

## Features: inputs to

First, 's inputs: the state and action. We could pass this info to our function for straightaway as the position (say, 1 for SB and 0 for BB), the hand number (from 0 to 1325), and the action (say, 1 for GII and 0 for FOLD). However, as we'll see, we'll get better results if we do a bit more legwork. Here, we'll describe the state and action together with a vector of 7 numbers:

```
nParams = 7

# Input:
#   hand: int hand between 0 and 1325
#   isSB: boolean indicating whether position is SB, else BB
#   isGII: boolean indicating whether action is GII, else FOLD
# Output:
#   numpy array containing features describing a state and action
def phi(hand, isSB, isGII):
    rank2, rank1, isSuited = intToHandMap[hand]
    return np.array([1,
                     rank2/numRanks if isGII else 0,
                     rank1/numRanks if isGII else 0,
                     abs(rank2-rank1)**0.25 if isGII else 0,
                     1 if (isSuited and isGII) else 0,
                     1 if isSB else 0,
                     1 if isSB and isGII else 0,
                    ], dtype=np.float64)
```

The vector returned by `phi` here will be the inputs to and is known as the *feature vector* and the individual entries are all *features* ( is pronounced "fee" – get it?). As we'll see, the features we choose can make a big difference in the quality of the result. Choosing features (known as "feature engineering") is one place where we can leverage domain knowledge about our problem. It's as much art as science. Here, we've encoded our knowledge about what info is relevant in this situation in several ways. Let's take a look.

The first entry is always a 1, for convenience. Consider the next four entries. These represent the player's hand. We've converted from hand number to `rank1`, `rank2`, and `isSuited`. These three variables technically give the same info as the hand number (neglecting the particular suits), but the model will make better use of the information in this format. In addition to the raw ranks, we've also included `abs(rank1-rank2)**0.25`. We happen to know that connectedness is an important property of hold'em hands, and that's what this represents. Also, the model will learn better if all the features have about the same magnitudes. Here, all the features are approximately between 0 and 1, and we've divided the ranks by `numRanks` to make this so.

Finally, if `not isGII` (i.e. if the action is FOLD) we actually set each of these numbers to 0. We know that the particular holding doesn't have any effect on the result when the player is folding (neglecting minor card removal effects), so we just remove the extraneous info in this case.

Now consider the final two entries. The first of these straightforwardly encodes the player's position, but the second depends on both `isSB and isGII`. Why might this be? We'll show the need for this "cross term" later.

## A linear model for

We're going to learn a linear function for our estimate . This means we'll actually learn a vector of parameters, usually called , of the same length (7) as the feature vector. Then, we'll evaluate our estimator for a particular with

Here, the subscripts refer to the particular elements of the vectors, and writing the argument list as indicates that the value of depends on both and but that we're mostly thinking of it as a function of with fixed. It's simple in code:

```
# Inputs:
#   theta: vector of parameters of our model
#   phi: vector of features
# Output:
#   Qhat(phi; theta), an estimate of the action-value
def evalModel(theta, phi):
    return np.sum(theta * phi)
```

Though it's commonly used, there's nothing particularly fundamental about this scheme that makes it the right choice for this problem. It's just one way (among many) to combine some learned parameters with some features to get an output, and it's entirely up to us to find a vector that makes this produce the output we want. However, with the right choice of , this will give us a pretty good estimate of the value of taking a particular action in a particular position with a particular hand.

## Simulating poker

We're going to "play" a bunch of hands. We'll put everything together to do this in the next couple of sections, but for now, let's build some three important pieces. These relate to the three important components of an RL problem: state, action, and reward. First, the state – each hand, we'll start by dealing each player a random holding.

```
# Input: N/A
# Output: tuple of two random hand numbers representing hands
#         that don't conflict.
def dealCards():
    hand1 = hand2 = 0
    while not pfconfl[hand1, hand2]:
        hand1 = np.random.randint(0, numHoldemHands)
        hand2 = np.random.randint(0, numHoldemHands)
    return hand1, hand2
```

Second, we need actions. Each player will use the current model (given by `theta`) and knowledge of his hand (`hand`) and position (`isSB`) to choose an action. In the following function, we estimate the values of GII and FOLD (`qGII` and `qFOLD`, resp.). We then choose the best option of the time and otherwise choose randomly. We return the action taken, as well as the corresponding value estimate and feature vector which we'll need later.

```
# Input:
#   theta: parameter for current model Qhat
#   hand: hand number
#   isSB: boolean position
#   epsilon: chance of making a random move
# Output:
#   A tuple of form (isGII, qhat, phi) describing the action
#   taken, its value, and its feature vector.
def act(theta, hand, isSB, epsilon):
    phiGII = phi(hand, isSB, True)
    phiFOLD = phi(hand, isSB, False)
    qGII = evalModel(theta, phiGII)
    qFOLD = evalModel(theta, phiFOLD)
    isGII = qGII > qFOLD
    if np.random.rand() < epsilon/2:
        isGII = not isGII
    if isGII:
        return isGII, qGII, phiGII
    else:
        return isGII, qFOLD, phiFOLD
```

Thirdly, once we know each player's cards and action, we simulate the rest of the hand to get the players' rewards. If either player folded, we can immediately return the correct values. Otherwise, we reference the players' equities to choose a random winner the right fraction of the time.

```
# Input:
#   S: stack size at the beginning of the hand
#   sbHand: SB hand number
#   sbIsGII: boolean indicating SB's action
#   bbHand: BB hand number
#   bbIsGII: boolean indicating BB's action
# Output:
#   A tuple of the form (SB value, BB value) indicating each player's
#   stack size at the end of the hand.
def simulateHand(S, sbHand, sbIsGII, bbHand, bbIsGII):
    if not sbIsGII:
        return (S-0.5, S+0.5)
    if not bbIsGII:
        return (S+1, S-1)
    # GII. Note: neglecting chops!
    sbEquity = pfeqs[sbHand, bbHand]
    if np.random.rand() < sbEquity:
        return (2*S, 0)
    return (0, 2*S)
```

We've cheated a little here in the case where the players get all-in. Instead of actually simulating the game by dealing out a 5 card board and evaluating the players' hands to see who won, we instead just choose a random winner according to the pre-calculated probabilities. This is mathematically equivalent (neglecting chops which don't matter here); it's just a bit more convenient and computationally efficient.

Most importantly, our learning process isn't taking advantage of these equities or knowledge about how the game "works". As we'll see shortly, the learning process would proceed exactly the same if we did bother to do the full simulation, or indeed even if our agent were interacting with some external, black-box poker game system which might even behave according to some different rules! Now, speaking of the learning process, how does that work exactly?

## Learning: updating

After a hand is over, we need to update `theta`. For each player, we have the state observed and the action taken. We also have the estimated value of the action as well as the actual rewards obtained from the game. In some sense, the actual reward obtained is the "correct answer", and if the value we estimated is different than this, there's an error in our model. We want to update `theta` to make a little closer to this correct answer.

Let be a particular state seen by one of the players and the actual reward she obtained. Let . here is known as the *loss function*. We've constructed so that the smaller it is, the closer is to , and if is 0, then exactly equals . In other words, we want to find a small adjustment to to make somewhat smaller. (Note that there are many possible loss functions that do what we want: get smaller as gets closer to . Here we've just made a common choice).

So, "updating " means changing to make smaller. There's more than one way to do this as well, but a simple approach is known as *stochastic gradient descent*. Wikipedia has the details, but in short, the rule for updating is:

We get to choose the "hyperparameter" (known as the *learning rate*) which controls the size of the updates we make. If is too small, learning is slow, but if it's too large, the process may not converge. Plugging in to this update rule and doing a couple lines of calculus, we get

The last line gives us the version of the update rule that we'll code. Keep in mind that both and here are vectors of length . This update rule applies to each element individually.

## Putting it all together

Finally, it's time to put everything together. We'll repeatedly:

1. Deal each player a random hand.
2. Let them each choose an action.
3. Get the results.
4. Update the model using the state-action-result tuples we observe.

The following function implements this Monte Carlo algorithm and returns the parameters `theta` of the model we learn.

```
# Input:
#   S: effective stack size in BB
#   nHands: number of random hands to play
#   alpha: learning rate hyperparameter
# Output:
#   An 7-vector of weights parameterizing our linear model
```

```
def mc(S, nHands, alpha):
    # Start with a random guess for theta.
    theta = np.random.rand(nParams)
    for i in range (nHands):
        sbHand, bbHand = dealCards()
        # SB action
        sbIsGII, sbQhat, sbPhi = act(theta, sbHand, True, epsilon(nHands, i))
        # BB action
        bbIsGII, bbQhat, bbPhi = act(theta, bbHand, False, epsilon(nHands, i))
        # get result from environment
        sbReward, bbReward = simulateHand(S, sbHand, sbIsGII, bbHand, bbIsGII)
        # update the model using each player's results
        theta += alpha * (sbReward-sbQhat) * sbPhi
        theta += alpha * (bbReward-bbQhat) * bbPhi
    return theta
```

Notice in particular how the update rule derived in the previous section was expressed in code.

## Results

### Interpreting the model

We'll fix for the sake of the example.

```
>>> theta = mc(10, 10000000, 0.0001)
>>> print(theta)

[ 9.63250365,  6.16764962, -1.34843332, -2.72533963,  0.22571655,
 -0.15230302,  0.14547532]
```

We have numbers! Do they make sense? There are actually a couple ways we can sanity check these and gain some intuition for what our model is telling us.

First, let's consider some specific situations. What's the estimated value for the SB when she plays FOLD? This is easy to find because is quite simple in this case. In fact, all its entries are 0 except for the first (fixed to 1) and the sixth (corresponding to isSB): `phi = [1, 0, 0, 0, 0, 1, 0]`. So, evaluating our linear model just amounts to adding the first and sixth entries of `theta`:

Now, we actually know that the value for the SB when she chooses to fold is exactly 9.5 according to the rules of the game. So, cool – pur model is pretty close! This is a nice sanity check and gives us one example of the magnitude of errors our model might be making.

Take another specific situation: BB folding. Only the first entry of phi is nonzero here, and we find an estimated value . It's not as clear what the right answer should be here, except that it should certainly be between 10.5 (as it would be if SB always plays FOLD) and 9 (what it'd be if SB always plays GII). And indeed, it is, and it's somewhat closer to 9 than 10.5, which is consistent with the SB playing GII more than FOLD.

There's a more general way to think about each individual entry of . An entry is the increment to due to an increasing the corresponding feature by 1. For example, having a suited hand when playing GII increases the the fifth entry of by exactly one. Thus, the estimated the benefit of having the suited hand is 0.22571655 – a small, positive benefit. Seems reasonable.

The second entry of (corresponding to the player's higher-ranked card) is 6.16764962.

This corresponds to the feature `rank2/numRanks if isGII else 0`, which corresponds to the player's higher-ranked card when playing GII. We divide here by `numRanks`, so an increment of 1 in the feature is approximately the difference between a 2 and an ace. An extra 6 BB for getting it in with an ace rather than a 2 seems reasonable. (But, why do you think that having a higher second card is apparently negative?!)

Examining the entry of corresponding to the sixth feature (`1 if isSB else 0`), the additional value of being in the SB is apparently -0.15230302, if all other features are equal. We might interpret this as the positional disadvantage: the small penalty that comes from having to act first.

However, all else isn't necessarily equal. If the SB is playing GII, the last feature becomes active as well. So, -0.15230302 is the additional value of being in the SB only when playing FOLD. When playing GII, we include the contribution from the last feature also to find a benefit of . Apparently the positional disadvantage is less when the SB takes the more aggressive option!

As we see here, choosing features that are meaningful in our problem domain can help us to meaningfully interpret our results. Interestingly, there's an old scheme for playing shove/fold scenarios known as SAGE. It was constructed to be easy to remember at the table during live tournament play. The idea is to construct the "power index" for your hand which has contributions for rank, suitedness, and pair and then use that to decide whether to GII or not. How do their set of features compare to ours? How about their results?

Finally, why did we choose the last feature to depend on `isSB and isGII` rather than just `isGII`? Think about it as follows. The estimated value of (BB, FOLD) is simply the first entry of , so this first entry needs to be free to change to whatever it needs to be in order to get the correct value for (BB, FOLD). Then, the sixth entry is the additional contribution for being in the SB, and it needs to be free to vary to get (SB, FOLD) right.

Once we switch from FOLD to GII, entries 2-5 become active and adjust the value for the player's particular, but these contributions apply equally to both the SB and BB. The model needs some way to give a different contribution for the SB getting all-in as opposed to the BB doing so.

Suppose our final feature were just `1 if isGII else 0`. This doesn't depend on the player, so the only difference in estimated value between SB and BB would be due to the `isSB` term. This single number would have to account for both the difference between SB and BB when playing FOLD as well as the difference between SB and BB when playing GII. The model would be forced to pick a single number for both of these differences, and it would probably end up with some poor compromise between them. Instead, we need `1 if isGII and isSB else 0`. With this, the model can differentiate between the incremental value of the SB GII vs the BB GII.

Note that there are still lots of subtle details that this model can't capture. For example, it is entirely built-in to the functional form of our model that the *difference* in estimated values of GII with two particular hands, e.g. A2 and K2, is exactly the same as the SB as for the BB. It's impossbile for our model predict otherwise, regardless of the values of .

We say that such a model has *high bias*. Basically, it's inflexible and has a strong built-in "opinion" about what the result will look like. This is why the feature engineering was so important. If we hadn't made some attempt at providing the algorithm with well-crafted features, it probably wouldn't have even been capable of representing a good solution at all.

We can add more features such as other cross terms to get a lower bias model, but this might come with downsides. We'd lose interpretability very quickly, and we may hit upon more technical problems such as overfitting as well. (Of course, in most modern applications, this ship has sailed. Accuracy is more important than interpretability, and there are ways to deal with overfitting).

## Visualizing the strategies

To find the complete strategies, we'll evaluate the model to see whether GII or FOLD is better for each of the 1326 hand combinations, for each player:

```
sbGIIRange = np.zeros(numHoldemHands)
bbGIIRange = np.zeros(numHoldemHands)

for i in range(numHoldemHands):
    for isSB, r in [(True, sbGIIRange), (False, bbGIIRange)]:
        qHatGII = evalModel(theta, phi(i, isSB, True))
        qHatFOLD = evalModel(theta, phi(i, isSB, False))
        r[i] = 1 if qHatGII > qHatFOLD else 0
```

It looks like the SB is shoving with about 55% of hands, and the BBs calls about 49% of the time:

```
>>> print(np.sum(sbGIIRange)/numHoldemHands)
0.553544494721

>>> print(np.sum(bbGIIRange)/numHoldemHands)
0.485671191554
```

Finally, we can generate some SVGs to draw the GII ranges themselves in a Jupyter environment:

```
from IPython.display import SVG, display
from collections import defaultdict

def drawRange(r):
    weights = defaultdict(int)
    counts = defaultdict(int)
    for i in range(numHoldemHands):
        rank2, rank1, isSuited = intToHandMap[i]
        hand = ranks[rank2] + ranks[rank1] + ('s' if isSuited else 'o')
        weights[hand] += r[i]
        counts[hand] += 1

    svg = '<svg xmlns="http://www.w3.org/2000/svg" version="1.1" width="325" height="325">'
    for i in range(len(ranks)):
        for j in range(len(ranks)):
            if i<j:
                hand = ranks[j]+ranks[i]+'s'
            else:
                hand = ranks[i]+ranks[j]+'o'
            frac = weights[hand] / counts[hand]
            hexcolor = '#%02x%02x%02x' % (int(255*(1-frac)), 255, int(255*(1-frac)))
            svg += '<rect x="' + str((len(ranks)-i-1)*25) + '" y="' + str((len(ranks)-j-1)*25) \
                    + '" width="25" height="25" fill="'+hexcolor+'"></rect>'
            svg += '<text x="' + str((len(ranks)-i-1)*25)+'" y="'+str(((len(ranks)-j))*25-10) \
                    + '" font-size="11" >' + hand + '</text>'
    svg += '</svg>'

    display(SVG(svg))

drawRange(sbGIIRange)
drawRange(bbGIIRange)
```

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 🟩 AAo | 🟩 AKs | 🟩 AQs | 🟩 AJs | 🟩 ATs | 🟩 A9s | 🟩 A8s | 🟩 A7s | 🟩 A6s | 🟩 A5s | 🟩 A4s | 🟩 A3s | 🟩 A2s |
| 🟩 AKo | 🟩 KKo | 🟩 KQs | 🟩 KJs | 🟩 KTs | 🟩 K9s | 🟩 K8s | 🟩 K7s | 🟩 K6s | 🟩 K5s | 🟩 K4s | 🟩 K3s | 🟩 K2s |
| 🟩 AQo | 🟩 KQo | 🟩 QQo | 🟩 QJs | 🟩 QTs | 🟩 Q9s | 🟩 Q8s | 🟩 Q7s | 🟩 Q6s | 🟩 Q5s | 🟩 Q4s | 🟩 Q3s | 🟩 Q2s |
| 🟩 AJo | 🟩 KJo | 🟩 QJo | 🟩 JJo | 🟩 JTs | 🟩 J9s | 🟩 J8s | 🟩 J7s | 🟩 J6s | 🟩 J5s | 🟩 J4s | J3s | J2s |
| 🟩 ATo | 🟩 KTo | 🟩 QTo | 🟩 JTo | 🟩 TTo | 🟩 T9s | 🟩 T8s | 🟩 T7s | T6s | T5s | T4s | T3s | T2s |
| 🟩 A9o | 🟩 K9o | 🟩 Q9o | 🟩 J9o | 🟩 T9o | 🟩 99o | 🟩 98s | 97s | 96s | 95s | 94s | 93s | 92s |
| 🟩 A8o | 🟩 K8o | 🟩 Q8o | 🟩 J8o | 🟩 T8o | 🟩 98o | 🟩 88o | 87s | 86s | 85s | 84s | 83s | 82s |
| 🟩 A7o | 🟩 K7o | 🟩 Q7o | 🟩 J7o | T7o | 97o | 87o | 🟩 77o | 76s | 75s | 74s | 73s | 72s |
| 🟩 A6o | 🟩 K6o | 🟩 Q6o | J6o | T6o | 96o | 86o | 76o | 🟩 66o | 65s | 64s | 63s | 62s |
| 🟩 A5o | 🟩 K5o | 🟩 Q5o | J5o | T5o | 95o | 85o | 75o | 65o | 🟩 55o | 54s | 53s | 52s |
| 🟩 A4o | 🟩 K4o | 🟩 Q4o | J4o | T4o | 94o | 84o | 74o | 64o | 54o | 🟩 44o | 43s | 42s |
| 🟩 A3o | 🟩 K3o | 🟩 Q3o | J3o | T3o | 93o | 83o | 73o | 63o | 53o | 43o | 🟩 33o | 32s |
| 🟩 A2o | 🟩 K2o | 🟩 Q2o | J2o | T2o | 92o | 82o | 72o | 62o | 52o | 42o | 32o | 🟩 22o |

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 🟩 AAo | 🟩 AKs | 🟩 AQs | 🟩 AJs | 🟩 ATs | 🟩 A9s | 🟩 A8s | 🟩 A7s | 🟩 A6s | 🟩 A5s | 🟩 A4s | 🟩 A3s | 🟩 A2s |
| 🟩 AKo | 🟩 KKo | 🟩 KQs | 🟩 KJs | 🟩 KTs | 🟩 K9s | 🟩 K8s | 🟩 K7s | 🟩 K6s | 🟩 K5s | 🟩 K4s | 🟩 K3s | 🟩 K2s |
| 🟩 AQo | 🟩 KQo | 🟩 QQo | 🟩 QJs | 🟩 QTs | 🟩 Q9s | 🟩 Q8s | 🟩 Q7s | 🟩 Q6s | 🟩 Q5s | 🟩 Q4s | 🟩 Q3s | 🟩 Q2s |
| 🟩 AJo | 🟩 KJo | 🟩 QJo | 🟩 JJo | 🟩 JTs | 🟩 J9s | 🟩 J8s | 🟩 J7s | 🟩 J6s | J5s | J4s | J3s | J2s |
| 🟩 ATo | 🟩 KTo | 🟩 QTo | 🟩 JTo | 🟩 TTo | 🟩 T9s | 🟩 T8s | T7s | T6s | T5s | T4s | T3s | T2s |
| 🟩 A9o | 🟩 K9o | 🟩 Q9o | 🟩 J9o | 🟩 T9o | 🟩 99o | 🟩 98s | 97s | 96s | 95s | 94s | 93s | 92s |
| 🟩 A8o | 🟩 K8o | 🟩 Q8o | 🟩 J8o | T8o | 98o | 🟩 88o | 87s | 86s | 85s | 84s | 83s | 82s |
| 🟩 A7o | 🟩 K7o | 🟩 Q7o | J7o | T7o | 97o | 87o | 🟩 77o | 76s | 75s | 74s | 73s | 72s |
| 🟩 A6o | 🟩 K6o | 🟩 Q6o | J6o | T6o | 96o | 86o | 76o | 🟩 66o | 65s | 64s | 63s | 62s |
| 🟩 A5o | 🟩 K5o | 🟩 Q5o | J5o | T5o | 95o | 85o | 75o | 65o | 🟩 55o | 54s | 53s | 52s |
| 🟩 A4o | 🟩 K4o | Q4o | J4o | T4o | 94o | 84o | 74o | 64o | 54o | 🟩 44o | 43s | 42s |
| 🟩 A3o | 🟩 K3o | Q3o | J3o | T3o | 93o | 83o | 73o | 63o | 53o | 43o | 🟩 33o | 32s |
| 🟩 A2o | 🟩 K2o | Q2o | J2o | T2o | 92o | 82o | 72o | 62o | 52o | 42o | 32o | 22o |

How'd we do? A lot of qualitative features we expect are here: big cards are good, pairs are good, suitedness is somewhat better than not suitedness, the SB plays looser than the BB, etc. However, borderline hands are sometimes played differently than in the true equilibrium strategy.

## Conclusions

A fairly introductory use of RL techniques got us some fairly reasonable strategies for playing the shove/fold game. The learning process didn't rely on any knowledge of the structure or rules of the game. It occurred purely by having the agent play itself, observing the results, and using them to make better decisions in the future. On the other hand, significant feature engineering requiring some domain expertise was necessary to learn a good model.

Finally, a bit of context. Many problems fit the can be expressed as RL challenges, and there are many different ways to approach them as well. The solution here might be characterized as model-free, value-based, Monte Carlo, on policy, undiscounted, and using a linear function approximator.

- **Model-free**: Our agent learned simply by taking actions and observing rewards. It didn't require any *a priori* knowledge about how those rewards were generated (e.g. knowledge about things like ranges, equities, or even the rules of the game) nor did it try to learn such things on the fly. In poker, we actually do know a lot about how certain hands and actions lead to particular rewards (and we could have taken advantage of this), but that's not the case in many other applications.
- **Value-based**: We focused on finding the values of each action in each state and then the actual policy (i.e. strategy) was more or less an afterthought. There are also policy-based methods (such a fictitious play), where the focus is on directly learning the action to take in each spot.
- **Monte Carlo**: We sampled entire hands (episodes) and learned based on the values we got at the end of the hand. "Temporal difference" methods make estimates of expected values in all the intermediate states before the hand is over and can learn more efficiently using those. Given that each player only makes a single action in the shove/fold game before it ends, this wasn't important for us, but it can make a big difference in problems with more states.
- **On policy**: We estimated the values of the same strategies that our players were

playing. This is actually somewhat problematic. Because the players sometimes took a random (non-optimal) action, the values we estimated were not quite the values of the optimal strategy, which is what we'd really like. More sophisticated "off policy" methods can actually learn about the optimal policy even while exploring non-optimal choices.

- **Undiscounted**: Most RL problems involve situations where there are many (possibly infinitely many!) states from the beginning to end of an episode. Of course, in that case the agent is looking to maximize the sum of all future rewards rather than its immediate reward. In this case, the agent is assumed to have a small preference for getting a reward now rather than sometime in the future. A hand of shove/fold play is always very short, so we didn't need to worry about this.
- **Linear function approximator**: We learned a linear function to map from our representation of the state-action pair to the value. Alternatives include simple tables which store a separate estimate of the value of every action in every state as well as many other types of function approximators. Neural networks, in particular, have been very successful. To some degree, this is because they don't require much feature engineering to get good results. Neural nets can often learn both a good set of features and how to use them! But that's a topic for another day.

References:

- Textbook by Sutton and Barto
- Lectures by David Silver

comments powered by Disqus

## willtipton.com

- willtipton.com
-

- LinkedIn
- GitHub
- Résumé

Coding and poker, mostly.