Michael Gump
MIT Class of 2019

Strategy Report

Pokerbots 2017

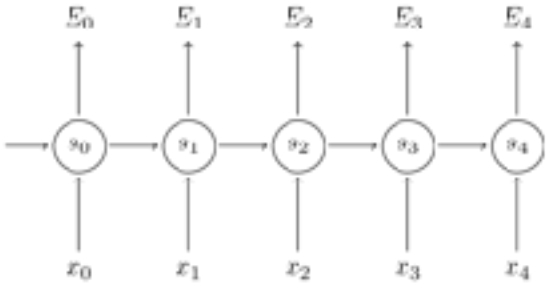
Introduction

My strategy was to use a Recurrent Q Learning Neural Network to train on poker hands in this variation through self-play and games on the scrimmage server. While towards the end of the month I had a system set-up where my bots could successfully converge to strategies that varied intelligently to the opponent's behavior I did not have the time to fully implement my strategy in a way that the bot could reliably act on it's understanding of the poker game.

The Bot

Q-Learning is a type of reinforcement learning in which a neural network learns to predict the maximum reward it can receive at a given game state across its action space. In other words the neural network takes in a state vector, S and transforms it to a vector A that is the length of its available actions. While Q learning in continuous action spaces has been done[1] it was simpler for my purposes to break up the bots actions into chunks of how much the bot had committed into the bot. Committing less than the pot size (or the pot size after calling) would lead the bot to fold, committing the bot size could lead to a check or a small bet if the minimum bet allowed it, and committing more than the pot size would lead to a raise if possible.

The network itself was a recurrent neural network with two layers of 64 hidden gated recurrence units (GRUs) as it's memory. A recurrent neural network is an modification of a normal feedforward neural network that allows it to analyze sequential data. A number of memory nodes store the hidden states of previous timesteps and are used to calculate the output vector[2]:



This allows the model to observe complexes relationships that unfold over the time dimension. This is necessary for games like poker where the decision making agent can not observe the entire state at any given timestep. Recurrent neural networks have been trained to play games as complex as doom[3]. To address the vanishing gradient problem the network also uses GRUs

which have weights not only for propagating information forward in the network but weights to decide how much information to store from a given timestep and how much to forget. This allows the network to find patterns that do not unfold in a simple linear fashion but are loosely connected with many timesteps in between. I had to make many other adjustments for my network to start learning anything about this game, such as changing the gamma (bot's discount factor for reinforcement learning), the learning rate, adjusting how often and when the bot replayed previous formative experiences, and playing with what the bot received at as a reward at any timestep. Just as a quick example I found that giving the bot the reward of how many chips it one or lost at the end of a hand caused to develop extremely risky behavior where it simply tried to play the odds and go all in a large percentage of the time. Giving it the sum of it's winnings over sum number of previous hands turned out to be the most effective method.

I'm kind of unsure how much I should write but I actually used a double Q network which uses separate networks to evaluate the future reward and to evaluate actions at a current timestep[4]. I'm running out of space.

Discarding was handled in an entirely separate process which simply evaluated the chance of each type of poker hand given the board and any subset of the player's hand. From this the bot made a simple decision with a bias towards keeping the same hand to account for inaccuracy in the calculation. The calculation was made by simulating the next of the flop, turn, and river as many times as possible within the timebank. Because of my evaluator's speed this was pretty inaccurate and because I didn't have time to fix it I just added a bias towards keeping the player's existing hand.

Conclusion

Because of other stuff I was doing during IAP I really only had about a week to do all this and most of that time was spent just coding the network and learning how to build a web scraper. I only had about two days to actually optimize the training process and I needed a lot more. By the end my bot could only really beat a random bot and a bot that went allin at every opportunity. And that was challenging enough for it to learn. If I had had more time I would have thought harder about the reward process and coded easier bots for it to play so that it could learn more slowly. As it stands the bot would converge quickly to a suboptimal strategy because its value network from DQN never had time to learn appropriate values of future states. Also I think there is a bug in my parsing code so I mostly had to rely on self-play and playing with hard-coded bots. Yeah so given that it just barely managed to beat a random bot I just submitted a hard coded bot as my final submission. But my code for this bot is on the scrimmage servers

Works Cited:

I'm lazy and I'm just giving links

[1]https://arxiv.org/pdf/1509.02971v5.pdf
[2]http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/
[3]https://arxiv.org/pdf/1507.06527v4.pdf
[4]https://arxiv.org/pdf/1509.06461v3.pdf