Modeling a Poker Player with Neural Networks

Walter Peregrim, Luke Sanyour
Department of Electric and Computer Engineering
Virginia Tech
Blacksburg, VA 24060
`peregrim@vt.edu, lukes11@vt.edu`

December 12, 2020

Abstract

Texas Hold'em is one of the more popular variations of poker with game play focused as much on betting as the cards being played. Modeling the behaviours of a poker player present an interesting challenge due to the sheer amount of variability and randomness of the game. Unlike games of chess or checkers, Texas Hold'em presents a complex problem with no clear-cut modelling solution as there are many unique variables at play. We have acquired a data set of about 52,000 poker games, with each game containing a common player. Our neural network is able to learn the given player's playing style and predict the most likely action given game information. In this report, we will describe our approach as well as possible alternatives, our findings and results, and possible future work.

1 General Approach

Deep neural networks are exceptionally good at handling large amounts of data, modelling complex and nonlinear relationships, and operating on incomplete information. These three traits are ideal for tackling the volatile intricacies of Texas Hold'em. The vastness of poker's game state space, the number of all possible game configurations, almost makes a NN a prerequisite for a successful poker agent. The erratic probability swings in each stage of a given hand can also be handled through the exploitation of a net's activation functions. Lastly, the unknown game information with respect to a given opponent's hand strength/potential does not necessarily have to hinder the accuracy of a neural network and can even be estimated with opponent modeling.

We actually implemented a few different models and created our own data, in addition to using a pre-existing data set, to see how changes in our approach would affect the accuracy of the model. The first model was trained on a data set of approximately 52,000 poker hands all involving the same player. Each hand contains information on seat order, blinds/dealer, our player's hole cards, stage of hand, community cards, each player's monetary stack, and each player's actions as the hand progresses. We calculated hand potentials and strengths from the raw data as well as recorded in-game metrics to be fed into a neural network. While this was a pretty basic attempt at modeling a poker agent, it provided a good baseline to compare future results to. Moving forward, we decided to generate our own game data to have more control over the quality of the agents in the game. Thus we also created three types of poker agents in which our model could learn from: an aggressive player, a standard player, and a passive player. Our dataset was primarily comprised of opponent information (play style, aggression, recent actions, hand quality etc.) which allowed our model to identify patterns in playing style and estimate the strength of an opponent's hand.

Given the variability and lack of game information associated with poker, we decided to evaluate how having precise statistical knowledge on winning probabilities affects the outcome of the hand. Players had no knowledge of their opponents' win probability, only their own. Even with this wisdom the volatility of poker swings in a given hand causes win probabilities to constantly fluctuate, and also brought about some intriguing results when training our model. Similar to our neural network approach, We tested and compared different model configurations to find the most optimal hyperparameters.

2 Data Ingestion

The cleaning and extraction of useful data proved to be a challenging part of creating our neural network. The raw data included a lot of unnecessary information in a convoluted format. Our player's hole cards were obtained from each hand and converted into hand potential and hand strength metrics. The latter represents the strength of a hand at a given stage, and the former represents the potential of a given hand to improve in the next stages. We used calculated statistics from both a poker website and a poker python library to represent our player's hand. Some of the player actions were grouped together (Raise, Bet, Call, All In) because they all refer to essentially the same action, a player putting money into the pot. While seat order is an important feature when playing professional poker, we decided to forego this metric to keep the network slightly simpler. Also, opponents' cards are only shown in the rare case of a showdown, which makes opponent modeling more difficult as we cannot see whether or not someone was bluffing.

In the following model we were able to keep player actions separate while maintaining a good accuracy, track seat order and blinds, and estimate opponent hand strength using a neural network. Generating our own poker data provided a lot of freedom and creativity to explore different game options, but creating various agents and simulating an entire poker game presented their own challenges. Obtaining excellent data, whether creating it from scratch or processing an existing dataset, has proven to be a very challenging aspect of machine learning, but implementing a few different options allowed us to produce more successful models.

3 Feature Encoding

There are a number of ways to encode categorical features. The most widely used method is one-hot encoding, where categorical features are represented as binary variables in a vector that has the same length as the number of categories. For example, in our implementation, card information is represented as a 52-length binary vector, one element for each card in a standard deck. Stage information is represented as a 4-length binary vector, for each stage of the hand (preflop, flop, turn, river). Representing categorical features in this manner allows the network to separate categorical information, as opposed to attempting to encode all of the information into a single feature. Ordinal encoding is another method of categorical feature encoding, where the categorical values are encapsulated into a single feature. This type of encoding can be disadvantageous in some cases. Ordinal encoding can create an order relationship between features, which for our model serves no purpose. One-hot encoding can have it's disadvantages too, it expands the feature set greatly, and the extra variables can cause the network to pick up on redundant information that is not relevant to predicting actions accurately. In our tests, one-hot encoding categorical variables gave a 10-20% increase in testing accuracy over ordinal encoding of the same features, showing that one-hot encoding the features enables our network to separate the data into categories effectively.

4 Neural Network and Hyper-Parameter Tuning

Our networks were implemented using the TensorFlow interface API, Keras. We created a configurable network architecture that allows for different amounts of input, hidden, and output nodes along with different activation functions. This allowed us to reuse this network across the different feature sets and data sources that we used. We leveraged the hyper-parameter tuning API, KerasTuner, to try many different combinations of models in order to find the one that could best solve the problem. Using KerasTuner, we set ranges of values for different parameters of the network, including possible activation functions loss functions, learning rates, number of layers, and number of hidden units. We tested different ranges of values for each parameter and choose the best that seemed to fit the model. For example, learning rates below 1e7 and above 0.01 tended to reduce testing accuracy greatly. Having more than 5 hidden layers also reduce accuracy by a large amount, possibly indicating that a network with too many layers may be needlessly complex for our problem. KerasTuner provides a couple of different heuristics for model searching. The most basic heuristic that KerasTuner provides is the Random Search heuristic, where models are randomly created every iteration from a range of possible combinations. The

HyperBand heuristic is a form of adaptive Random Search, which aims to speed up the searching process by adaptive resource allocation and early-stopping when it believes that there is no need to continue searching (for example, if validation accuracy for many models is the same). The final heuristic we tested is the Bayesian Optimization heuristic, which assigns a probability to a given set of hyper-parameters that is an indicator of how well the model may perform. It will then only attempt to test models it believes are a possible improvement over the best model it currently has. Overall, there is no heuristic that is considered the best, and it's very problem dependent. For our model, we achieved the best results by using the Random Search heuristic.

5 Results and Conclusion

Results from training our model on a preexisting data set, and a generated data set as discussed in section 1, are shown in figures 1 2.

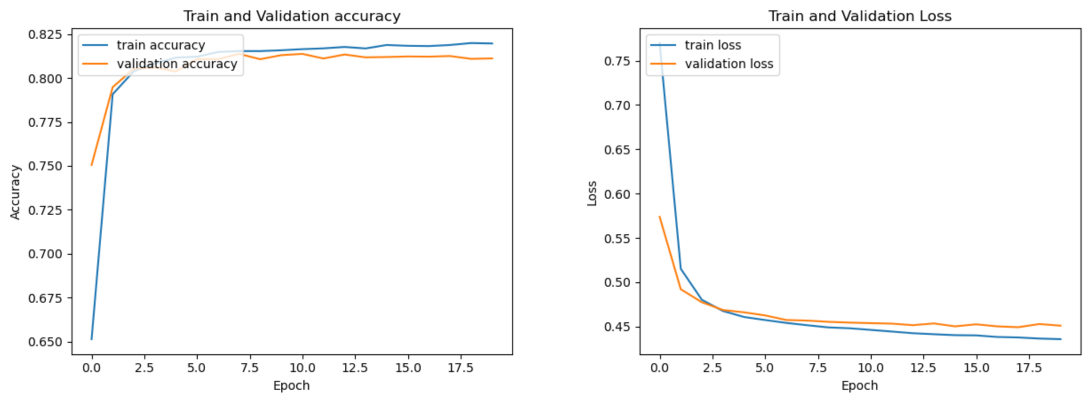

Figure 1: Imported data set accuracy and loss

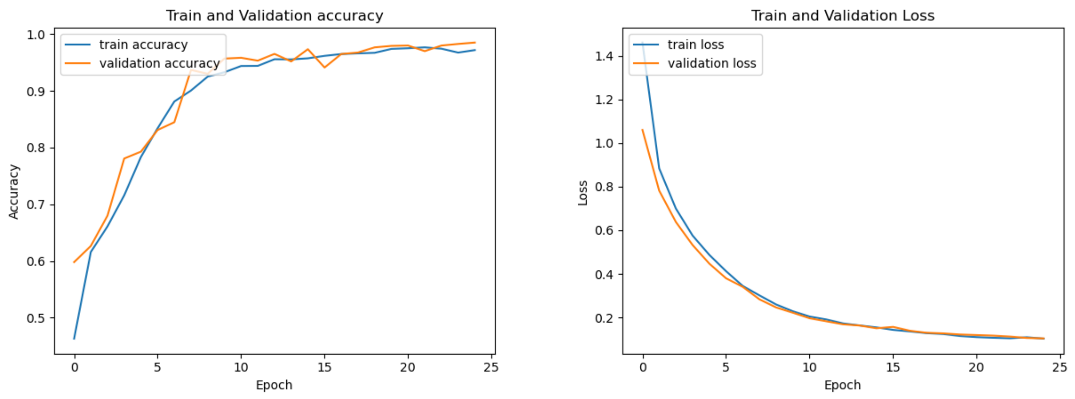

Figure 2: Generated data set accuracy and loss

We were able to achieve a validation accuracy of about 82% when modeling the real-world player, and an accuracy of 98% when modeling our generated data. This disparity may come from the difference in features that each model uses. Due to the limitations of the external data-set, we were only able to include the player's card information, stage of the game, hand potential, and hand strength. It included no information about the opponent's actions because we could not effectively model opponent's decisions without card information, of which there was very little in the external data set. Playing off an opponent's actions is a large part of playing poker, so this may account for the 20% difference. Poker is also a very dynamic game where most often players are, in a sense, playing each other and not specifically the game, and thus it is incredibly difficult to encapsulate the nuances of the human-human interaction aspect of the game into a model. The features included in the generated data set do take into account opponent actions, and intuitively, they seem to be better indicators of a certain action (for example, a win probability that accounts for an opponent's play-style, or including the previous action of the player). The high accuracy of the generated model may because it was generated in a controlled environment with well-defined rationale for a specific action, and as a result might not perform well against real players. All in all, we developed a model that can imitate a real-world player with somewhat reasonable accuracy given our constraints, and created a more optimal model capable of playing against different play styles with a high accuracy of choosing an optimal action. To extend this project, it would be interesting to test our models against real players, either to see if we can play them effectively or model them effectively (a pro player perhaps). We could then use this information to improve our approach.

5.1 Contributions

Luke Sanyour - Neural network, feature encoding, hyper-parameter tuning
Walter Peregrim - Game-level statistical calculation, data ingestion

Code Base: https://github.com/walterperegrim/poker_nn