Xploit Hand History Analyser and Neural Network Training on Poker Databases

Santiago Weight

December 24, 2019

1 Hand History Parsing

1.1 Bovada History Format

An example Bovada text file is as follows. Though the format of the Bovada hand data format is defined, the actual structure of the files output by the website are extremely inconsistent. For example, though the small blind post should always come after the cards are dealt followed by the big blind post, either or both can not appear at all or come earlier. This means that the grammar definition is quite loose, and so a well-defined parser cannot be created. Instead a single Parser Action was created for all cases. Since this is incredibly inefficient, in the future, a more well-defined parser would be better. Other simple issues exist in the files such as spelling inconsistencies and file corruption.

```
Bovada Hand #3781339018  Zone Poker ID#1382
HOLDEMZonePoker No Limit - 2019-06-06 01:56:35
Seat 1: Big Blind [ME] ($25 in chips)
Seat 2: UTG ($96.40 in chips)
Seat 3: UTG+1 ($6.65 in chips)
Seat 4: UTG+2 ($24.80 in chips)
Seat 5: Dealer ($25 in chips)
Seat 6: Small Blind ($57.32 in chips)
Dealer : Set dealer [5]
Small Blind : Small Blind $0.10
Big Blind  [ME] : Big blind $0.25
*** HOLE CARDS ***
Big Blind  [ME] : Card dealt to a spot [As Qh]
```

1.2 Haskell's MegaParsec package

The entire parsing codebase was programming using Haskell's MegaParsec parsing library, which is an industrial strength parsing library and is currently maintained. The MegaParsec library allows for fast speeds for free. When benchmarking the Xploit parsing functionality against a commonly used poker database tool, PokerTracker 4, the performance with the Haskell implementation is at least comparable and likely faster. We have to bear in mind that the PokerTracker 4 implementation is at least uploading the data to a local PostgresSQL database, and likely calculating some cacheable values for hands, suich as whether a 3-bet occurred.

Megaparsec allows us to create simple text parsers by declaring `Parser` monads. Currently the `Parser` is declared as a simple Parser that parses strings with no inbuilt error capability. In the future, as needs arise, the parser will have to include some other functionality in the general monad declaration.

```
handHeaderP :: Parser Int
handHeaderP = do
  lexeme (choice [ Bovada <$ string "Bovada"
                 , PokerStars <$ string "PokerStars" ]) <?> "Network
   "
  string "Hand #"
  handID <- integer <?> "Hand Number"
  tableType <- lexeme . optional $ Zone <$ string "Zone Poker"
  pokerID <- lexeme . optional $ string "ID#" *> integer
  tableID <- lexeme . optional $ string "TBL#" *> integer
  lexeme $ string "HOLDEM" *> optional (string "ZonePoker") <*
   string " No Limit -"
  (year,month,day) <- lexeme $ (,,)
                      <$> (integer <* char '-')
                      <*> (integer <* char '-')
                      <*> integer
  (hour, minute, second) <- lexeme $ (,,)
                            <$> (integer <* char ':')
                            <*> (integer <* char ':')
                            <*> integer
  let time = DateTime year month day hour minute second
  return handID
```

Listing 1: An example Parser that parses the history's header text

Due to the extremely inconsistent ordering of the hand actions, almost all actions have to be parsed in a single large for-loop, since we cannot rely on any ordering for speed purposes. In the future, a better implementation would be a greedy-ordering parser with recovery so that we can take advantage of the partial ordering whilst allowing for recovery.

1.3 Data Structures

The data-structure for Actions in a hand is described in the `data` declaration below. All actions were parsed into this Action datatype to ensure a monomorphic data structure.

```
data Action = PlayerAction { position::Position
                            , action::ActionValue
                            , isHero :: IsHero }
            | DealerAction DealerActionValue
            | TableAction Position TableActionValue
            | UnknownAction
    deriving (Read, Show)
```

Listing 2: Basic `Action` datatype

Since the parsing doesn't currently parse all actions into appropriate data, the `UnknownAction` exists to accomodate actions such as `Showdown` and `Hand Result`.

1.4 Parsing Benchmarking

The MegaParsec implementation was tested using GHCi, Haskell's interactive prompt, and setting time duration output with `:set +s`. The program then parsed the `Imported Hands` directory, and counting the number of hands parsed.

Program	Completion Time
MegaParsec	10.90 secs
PokerTracker4	55

Table 1: Runtime Comparisons

A big question mark around the Haskell implementation is unfortunately currently unanswerable. Haskell used lazy evaluation, which means that values will only be evaluated once they are needed. For example `length [2 î000]` will never do any multiplication and will return `1` instantly. Therefore, it can often be difficult to reason about the eager-evaluation ("normal" evaluation) runtime of the program. This could mean that the program is significantly slower when certain parts of the data are queried later down the line. However, when running my database query program, I did not experience any runtime lossses, which would indicate that the above runtime is at least close to eager-evaluation runtime.

A significant runtime optimisation was gained by using `mapConcurrently` from the `Control.Concurrent.Async` package. This allows the list of files to be parsed in parralel. Though the runtime only improved by 25%, the memory useage reduced from 14 megabytes to 3 kilobytes. This second memory useage is also very encouraging, though it may be due to unevaluated parts of the data.

2 Game Emulation

2.1 Purpose

In order to extract data points from each hand, we have to emulate the hand, as it is by no means trivial to calculate stack and pot sizes without iterating through the hand history. Furthermore, game emulation was used since in the future, should I choose to expand the functionality of Xploit, I will be able to reuse the game emulation code for other features such as graphical hand replay and running a simple game server.

2.2 Room for Expansion

I have been working with Tom Chambrier on working to expand his current codebase, that implements a server for recreational poker play. Currently he has an independently working server/client model working using Haskell for the backend and JS REACT for the front end. His primary goal is to expand the functionality of his server to allow for different poker agents to play against each other. One feature we discussed is the potential to put different Neural Networks against each other in order to allow programmers to develop neural networks easily within a single framework, with potentially some kind of leaderboard describing the effectiveness of the neural networks.

The ultimate goal with him is two-fold: to implement a fully functioning poker client with the ability to put players against each other safely; to implement a version of the server that allows users to pitch different poker players against each other, be they human or AI.

I have yet to do any extensive work with him yet as he was having trouble with setting up the server. The primary current goal is to fix up the game logic to ensure that all expected functionality is implemented for Texas Hold 'Em.

For more information you can visit his github repo.

3 Neural Networks

From now own, we will treat the entire database of anonymous players on the Bovada network as a single unit, “the player population”. Different players in the population will make perhaps drastically deifferent decisions; however, this is, at least in principle, no different from players who make different decisions when in different moves. This way we can create a probabilistic model that shows population habits in the long term. Again, we can clearly not deterministically predict a player’s actions in an anonymous database, but instead the likelihood of decisions being taken by a random player in that population.

3.1 Choice of loss function

The goal of our neural network is to predict the probabilities of an action occurring. For example, if the UTG open rate with KTs is 70%, we would like the network to output 0.7 for open when given KTs.

The choice of loss function switches dependent on type of prediction we intend to receive. In the bucket structure, we use categorical cross-entropy, for example. However, when it comes to more unique predictions, such as predicting the specific action, a custom loss function may be needed, as it may be more important to predict the action versus the bet sizing, and so losses for certain vector entries will be higher than others.

3.2 Neural Network Structure

The current structure of the neural network is a simple Sequential neural network composed of multiple dense layers. The use of dense layers allows for simple declaration of neural networks but also speedy runtime of neural networks. The difficulty with simple, dense neural networks is providing them with context-level data. In particular, neural networks will always require that they are provided with same-size input matrices. In poker games, however, the data provided to the neural network must be of variable size, due to varying sizes of poker hands. For example, the a hand that ends pre-flop may typically have around 6-10 actions, whilst a hand that reaches the river may have as many actions as 30 or even more.

Another challenge that arises when feeding poker hand data to a neural network is sequentially sensitive data, especially community cards dealt on the flop, turn and river. In order to address this requirement, the location of the of the community card data seems integral to its interpretability by the neural network. This fixed location also fixes the location of actions taken on each street.

Testing with Dense Layer Network

Testing with a dense layer network, I used UTG opening decisions as the test space to produce a network. Each hand is made into a single data point: the network is provided with the starting stack sizes and the player's opening hand.

The network was able to achieve

3.3 Recurrent Neural Network

The use of a recurrent neural network is a seemingly ideal solution to the problem of providing context to the neural network. This is because we no longer have the issue of providing the neural network with all the valid context at one time. Whereas in dense neural networks, a single data point (or matrix) is provided to the network for each training instance, in RNN's, each data point is a sequence of data items. In particular, we can use an LSTM to enable our recurrent network to retain information over the length of the sequence of data entries within a total data point.

One immediately apparent issue that must be confronted with an RNN is that not every point in a hand should be used to calculate the loss of the RNN for that data point. Though an RNN should certainly receive and predict those points at which a member of the Bovada population made a decision, it is less clear what should occur when I myself was making a decision. Though the RNN should be able to consume the information provided by my actions in the hand, if we take its predictions as to how I behave into account, the RNN will drastically overfit to my play style, and not adequately learn to the population exclusively.

This is an issue that I have not yet solved with my codebase. In order to solve this problem, I will most likely have to implement my own recurrent network layer inheriting from LSTM and then implement a call method for this new layer. Another option might be to artificially set the RNN's predictions for those predictions we are not interested to the correct prediction, so that the loss of the RNN is never affected by the invalid data points, whilst still being able to incorporate those points in the hand into a single training instance.

Encoding the data Encoding the data is an extremely difficult problem that I have not yet solved. When passing the data to a neural network, the shape of the matrices must be unchanging. To that ends, the encodings of bets must also be consistent, and as such, the encoding for a Raise must be equivalent to that of a Fold. The Algebraic Data Type that describes the bets a player can make in a game of Texas Hold 'Em is as follows:

```
data ActionValue = Call Double
            | Raise {amt_from :: Double,
                      amt_to :: Double}
            | AllInRaise {amt_from :: Double,
                          amt_to :: Double}
            | Bet Double
            | AllIn Double
            | Fold | FoldTimeout
            | Check | CheckTimeOut
            | OtherAction
```

Listing 3: Basic `Action` datatype

Since some of the types here are product types, such as (Bet, Double) or (Raise, Double, Double), but others are not, it is not clear how to standardise the size of the data. Currently I have coded two ways in which the data can be produced. For all encodings, the type of the output is consistently a list of doubles, or [Double] in Haskell, since we require float32 or float64 when handing the data off to a Keras neural network.

Simple Encoding - in this case the encoding is a binary encoding followed by the bet size. For example, Raise 2 5 will encode to [1.0, 2.0, 5.0], Call 2 will encode to [0.0, 2.0, 0.0], and finally Fold will encode to [5.0, 0.0, 0.0]. There is some flexibility around how to encode this way. Firstly, the bet enumeration values themselves can encode to one-hot values, where the first 10 bits are binary values indicating the type of the bet (Fold, Call etc.) and the final two values are optional values that indicate optional bet/call/raise sizings. Furthermore, we can place a sign bit in the optional bet sizes to indicate whether a bet has been made. In this encoding, a Fold would become [5.0, -1.0, -1.0] and a Call would become [0.0, 2.0, -1.0]. This is particularly effective when used in conjunction with a non-continuous activation function for x¡0, such as relu, as the network will be able to more astutely parse out the useless bits from the useful bits in the data, and therefore more effectively train on the data. All of these encodings can become one-hot encodings trivially, as we can simply replace any calls to functions with Action -¿ Double like type signatures with equivalent one-hot functions. Bucket Encoding - another potential encoding is bucket encoding, which groups subsets of actions into a single set. For example, we might decide that Bets of 1/2 - 3/4 of the pot are all equivalent, and that Bets of 3/4 - 5/4 of the pot are also equivalent. This reduces the dimensionality of the problem drastically, but also loses a lot of information in the process. This encoding (which can also be one-hot) is also very easy to learn how to play against, since we reduce one level of complexity when observing the playing habits of the population. However, the abstraction level is decided at compile time, as opposed to runtime from the perspective of the program user. In the unsimplified version, the user will decide on the abstraction by selecting buckets for a vast variety of bet sizes; in the bucket encoding, the abstraction happens at compile time, but with the user still having the option to bucket. Some issues that arise with such an encoding are: Loss of information. We are inherently removing information from the bet by reducing the accuracy of the data. Furthermore, it's quite possible that whole number vs non-whole number bets represent very different player behaviour. Such behaviour would likely be lost, though some fraction of it could be maintained, at least in principal with some extra feature engineering. Addition of unintentional information. By placing a bet in a bucket, we naturally add the information from other bets into our bet, since all the bucket's bets will be treated as equal. This is a problem defined largely by the selection of bucket sizes. Selection of buckets. How we choose the buckets to use is a classic example of programmer feature manipulation that can cause significant issues at training time. Though I hold some opinions about what bet sizes are discretely different, I (perhaps no one) am not aware of all intricacies around bet sizes. Some are obvious (minimum raises are a big red flag), but others might be extremely obscure or even un observable by humans. One tactic might be to bucket by some defined rule (say by 0.2bb increments), but this solution will likely cause poor buckets that I can easily observe.

3.4 Architecture of RNN

The natural choice for RNN training for this problem is to use an LSTM. The natural inputs to this RNN will be to have player actions (and table actions) as inputs to the RNN, with the outputs being a prediction for the next player action. However, a lot of decisions have to be made around what to provide the neural network with and where to give the neural network that information.

Here is a short summary of the information we would like the neural network to have, with the information being either provided to it or learned in the hidden state:

- The to-act player's holding
- The to-act player's position (may not be important)
- Community card
- Pot size
- Current bet faced
- Stack sizes
- Some notion of the game state (inferred by the RNN's hidden state)

- Who to train on.

The player's current holding and position are simple to fulfill. We can simply pass the information to the RNN through xHAT, choosing some appropriate encoding (continuous or one-hot). However, presenting the other features will be a rather challenging problem to solve.

When the community cards are dealt is crucial to the game of poker. We cannot provide the RNN with community cards before or after the net is run, and so it must occur within the sequence of actions at the exact place the dealing occurred. However, we also do not want to treat card-dealing as similar to a player action. In particular, card dealing should not cause RNN predictions that will effect the loss of the RNN. The RNN should either be able to precisely and easily predict the dealing of cards, or be given that information, so that the RNN can let the information pass through and achieve perfect predictions. For example, we might provide a sign bit in xHAT that signals whenever the next action is a deal. In this case, we ask the RNN to predict a simple vector, perhaps all zeroes or negative ones. In this way, we can ensure that the information from dealer actions is available to the network, and can therefore be trained on by lowering loss through altering the hidden state according to the cards dealt. However, at the same time, we prevent the neural network from becoming stuck on any irrelevant information and make this information easily accessible. A final solution to consider is designing a new Keras layer that is custom built to make it such that though the community card deal is passed through the network, the loss remains unaffected, and so we essentially skip over the datapoint whilst parsing the data. This approach certainly seems viable, and will be subject to the complexity of implementation; it may be especially difficult to program this sort of selective loss learning, since we may lose information about the neural network's weights' loss derivatives after consuming the card deal datapoint.

We assume that the neural network will be able to learn or infer pot size. Since at each step the network receives the size of the bet, we do not need to give it this information, unless we find that it requires it.

We similarly assume that an RNN should find a way to incorporate current bet faced into its hidden state, or at least function of current bet faced.

Stack sizes can either be provided in the xHAT, or potentially in the h_0 initial state. Providing the stack sizes at each point in the sequence will have a small effect on runtime, but will ensure that the RNN always has the information available. We could also, at least in theory, instantiated the initial state to contain the information of 6 stacks, with the rest of the initial state continuing to be randomly initialised. We would hope that the neural network would be able to learn to handle these values appropriately. Again, as with feature encoding, this math be over-engineering, as we are explicitly deciding on learnable weights on behalf of the neural network. However, this does appear to be an exceptional circumstance, as the information is not a priori learnable by the network (since it is variable) and since providing the information at each training step appears somewhat redundant and wasteful.

Who to train on is a very important aspect of the RNN. We addressed some potential solutions to the problem of predicting card dealing at runtime, namely either producing dummy predictions or custom programming a software solution. A similar issues should be addressed with regards to players who should be ignored at training time. When training the neural network, we would like to avoid overfitting to a single player in the pool, particularly the player from whom data was collected, since they are far more populous than any other player. A similar approach will have to be taken with these such players, where their bet information is crucial, but the treatment of them as a regular datapoint in the sequence is detrimental to the results of the neural network.

3.5 Training by position

The above architecture discussions all confront the problem by treating each hand history as a single datapoint to the RNN. Such an approach causes an array of issues around the ordering of actions and the presentation of only relevant information to the network. It also opens the network to being an extremely general machine, as it has to walk the line of both human predictor and game simulator. A solution to some of these issues is found by splitting a single hand history into 6 distinct data points. Each of the data points is the hand history as told by each of the 6 players, where actions taken by any of the other players are ignored through one of the prediction-dismissing techniques described above.

Configuration State In order to easily modulate the entire network pipeline, we use Haskell's State monad to carry configuration settings through the program. This allows for more modular construction of encodings. Any code with a call to "toEncoding" can be with made where toEncoding is a function Encodable -¿ Config [Double], that takes an Encodable type wrapped in the appropriate wrapper and returns a Config State monadic action containing a [Double], the appropriate encoding of the data passed. When calling the entire program, the Config State is instantiated during the runState call in some parent function. Furthermore, such a convention ensures that configuration is unchanged from begin of execution to the end of execution, allowing for the configuration to be naively passed to the Python subscripts with full safety and assurance.

4 Database Querying and Interface

The current primary functionality of Xploit is the GUI interface that allows the user to specify filtering trees that populate the ranges at nodes in that tree. The game of poker is inherently tree-like in its structure. Most database querying software does allow for tree queries, using “or” or “and” keywords, but the tree is text-based, not visual. The goal of the interface is to provide a tree interface querying system that allows users to visually create and populate trees.

4.1 Implementation

4.1.1 FilterTrees

The underlying implementation of the tree queries is the `FilterTree` datatype.

```
data FilterTree = EmptyNode | FilterNode
                              { _filt :: NamedParser
                              , _filterrange :: ShapedRangeC
                              , _left :: FilterTree
                              , _right :: FilterTree }
```

Listing 4: `FilterTree` type declaration

The tree is currently a binary structure, in which the right hand side represents an “or” and the left represents and “and”. Any tree can also simply be the empty node. Another way to think about the “and” and “or” branches is by “then” and “or else”. In other words, it is a way of describing the structure of a general poker hand and how to traverse it. At each node is an associated range that begins as the empty range, and is populated as hands are pushed through the tree. There is also a `NamedParser` filter, that contains the filter that this node matches on.

The right branch of the node is evaluated on the same action inputs as the node itself receives, since the node does not consume data before handing it off to the right. However, after filtering the input actions, only the remaining actions are handed off to the left branch. In the Xploit GUI, the structure of this binary tree is rotated, such that the diagonals from top to bottom right (the or branches) are on the vertical axes, and the diagonals from top to bottom left are on the horizontal axes.

When a hand is passed through a filter, the root node’s filter attempts to match on the incoming actions, and hands the rest to the left child. The right child receives all inputs as though the node had not existed. At each point, if the filter matches any actions, those actions are put into the node’s range, so that the user can know what actions matched that filter.

4.1.2 Running the trees

The `runTree` function, which takes a `FilterTree`, a hand and its remaining actions (another node may have consumed some of the beginning actions) and returns the new `FilterTree` after populating the range with those actions that matched the filter. A heavily pseudoed version of runTree is below.

```
runTree :: FilterTree -> (Hand, [Action]) -> FilterTree
runTree node (hand, actions) =
    case runParse node.filter actions of
            (rest, accumulated) ->
                -- convert actions to holdings
                let combos = hand.actionsToCombos accumulated
                    newrange = -- add combos to node's range
                    -- run original actions on the right (the ``or
   '')
                    orTree = runTree right (hand, actions)
                    -- run rest actions on the left (the ''and'')
                    andTree = runTree left (hand, rest)
runTree EmptyNode _ = EmptyNode
```

Listing 5: The `runTree` function

4.1.3 Action Filters

The `filter` member of the `FilterTree` is a parser not far from the parsers implemented in MegaParsec. At some point, in order to gain speed optimizations, it is likely I will have to port the parser over to MegaParsec. However, this is quite the challenge and so I won't be doing this in the short term.

In the file `HandFilter.hs`, filters are implemented as custom monadic parsers, which are commonly used in Haskell. The action parsers are unique however in that they accumulate a result by default. In this sense they are a stateful parser. The parser could equally be implemented such that accumulation must be declared on purpose, which be safer, but less ergonomic. Fundamentally, the parser treats each action as though it were a character, and checks if it matches the requirements of the parser. Each parser, once declared is essentially a function that takes a list of actions and returns a tuple containing those actions that were valid and the actions that remained after parsing.

4.1.4 Serializing Filters

In order to save filters, the monad filters had to be data-serializable in some way. Many of the datatypes in the codebase were made JSON-serializable for free using

Haskell's Aeson package, which includes a convenient `deriveJSON` function. However, since the filter parsers are functions, they cannot be serialized by Aeson. Instead, a custom abstract syntax tree was created to represent the filter monads, as is common with parsers and the languages they parser. The AST declaration is shown below:

```
data NamedParser = NamedParser
                    { _name :: Name
                    , _parserAST :: ParserAST
                    }

data ParserAST = Nop
               | GetAction
               | Or ParserAST ParserAST
               | Do ParserAST ParserAST
               | Union ParserAST ParserAST
               | Satisfy ActionIndex
               | SatisfyOf [ActionIndex]
               | SatisfyNot ActionIndex
               | ManyTillPlus ParserAST ParserAST
               | IgnoreMatch Filter Filter
               | Item
               | RFI [Position]
               | CallAST [Position]
            deriving(Show)
```

Listing 6: Filter parsers AST declaration

When using the filters, we use the function `toParser`, which translates the AST into a monadic parser. The user themselves, though unaware, declares these ASTs as opposed to the actual parsers, in order to allow for the serialization.

4.2 Interface

The interface to these filter trees is simply a reflection of their underlying implementation. There is a range viewer, that displays the currently selected node's range. The side panel is an interface to creating the filters, in which one can declare a parser that can be added to the tree or replace a node currently in the tree. The bottom panel is a tree of buttons that display the filter tree as it currently exists, and where the user can select nodes to delete, replace, or add children to the tree.

5 Type-Safe Poker

One of the primary assurances of online poker play is that of semantic and financial safety. In particular, poker players expect complete safety at the table, and that no corruption or incorrect programming will interfere with the game.

To that end, we can apply type safe programming to the domain of poker play, in order to ensure invariants are maintained at each stage of the game. Furthermore, with the use of dependent types, we can confirm many aspects of the safety expected from online poker statically, at compile time.

The payoff of such an implementation is certainly at question. For one, the implementation time of such a system is by no means trivial, and may be often far beyond justifiable for the gains obtained from static program checking. Some invariant maintaining can be implemented quickly, efficiently, and with high cost-benefit in terms of runtime safety. For example, through the use of the Indexed State monad, poker streets can be emulated in the State index through the use of GADTs, such that a Flop can programmatically only occur after some Pre-Flop street type has been obtained. This constitutes the use of dependent types for the purpose of safe bottlenecking, since we know that a function of the type PreFlop -¿ Flop contains all necessary components of the transitions. Furthermore, it reduces points of breakage, and maintains invariants in a modular way at low cost.

Other such type constructs can prove to be largely useless in the scope of site development. One can, for example, remove the positions of players from the ADT representing game state, at the expense of having to construct type-level lists of player maps. Needless to say the payoff is deep in the red. However, the effectiveness of type safe programming for the use of poker is non-trivial, and parallels the uses that type-level programming has found for such applications as coding on the BlockChain and large-scale financial institutions.

An additional convenience used for the type-safe implementation in Poker/Game.hs is the use of the `Language.Haskell.DoNotation` package. This Haskell package makes the use of the indexed State monad far less taxing and tiresome than can be achieved otherwise (such as the RebindableSyntax extension). It does so by allowing do notation to work with *both* indexed and unindexed monads.