**Comp Science NEA**

Contents

[Analysis 3](#_Toc510955628)

[Introduction to Organisation and Client 3](#_Toc510955629)

[Outline of the Problem 3](#_Toc510955630)

[Investigation 3](#_Toc510955631)

[Analysis of Investigation 6](#_Toc510955632)

[Objectives 8](#_Toc510955633)

[Data Flow Diagram of Game 12](#_Toc510955634)

[Data Flow Diagram of Current System 13](#_Toc510955635)

[Research 14](#_Toc510955636)

[Potential Solutions 18](#_Toc510955637)

[Proposed Solution 22](#_Toc510955638)

[Proposed Solution DFD 23](#_Toc510955639)

[DFD for Game System 25](#_Toc510955640)

[ERD for Proposed Solution 26](#_Toc510955641)

[Data Dictionary 27](#_Toc510955642)

[GUI Structure 30](#_Toc510955643)

[Proposed System Neural Network Architecture 31](#_Toc510955644)

[Prototyping and Critical Path 33](#_Toc510955645)

[Documented Design 35](#_Toc510955646)

[BlackJack Prototype 35](#_Toc510955647)

[Introduction 35](#_Toc510955648)

[Design – The Blackjack Game and Interface 36](#_Toc510955649)

[Design – Card Counting AI 46](#_Toc510955650)

[Design – Neural Network Based AI 60](#_Toc510955651)

[Design – Simple AI 66](#_Toc510955652)

[Design – Comparison Tool 66](#_Toc510955653)

[Software Development 67](#_Toc510955654)

[Main Directory 67](#_Toc510955655)

[Agent.py 67](#_Toc510955656)

[Blackjack.py 69](#_Toc510955657)

[Card\_Counter.py 75](#_Toc510955658)

[CC\_Agent.py 80](#_Toc510955659)

[CC\_AI.py 81](#_Toc510955660)

[Comparison\_Tool.py 83](#_Toc510955661)

[CT\_GUI.py 100](#_Toc510955662)

[Deck.py 107](#_Toc510955663)

[GUI.py 109](#_Toc510955664)

[Moves.py 110](#_Toc510955665)

[Rand\_AI.py 111](#_Toc510955666)

[Simple\_AI.py 112](#_Toc510955667)

[/DB 114](#_Toc510955668)

[Create\_Agents\_Table.sql 114](#_Toc510955669)

[Create\_Card\_Counter\_Record.sql 114](#_Toc510955670)

[Create\_Games\_Record.sql 114](#_Toc510955671)

[Create\_Users\_Table.sql 114](#_Toc510955672)

[Populate\_Agents.sql 115](#_Toc510955673)

[CT\_Wrapper.py 116](#_Toc510955674)

[DB\_Wrapper.py 122](#_Toc510955675)

[Users\_DB.py 124](#_Toc510955676)

[/NN\_AI 127](#_Toc510955677)

[Experience\_buffer.py 127](#_Toc510955678)

[NN\_Move.py 128](#_Toc510955679)

[NN.py 129](#_Toc510955680)

[Trainer.py 137](#_Toc510955681)

[/Structs 143](#_Toc510955682)

[Binary\_Tree.py 143](#_Toc510955683)

[Card\_Binary\_Tree.py 150](#_Toc510955684)

[Circular\_Queue.py 152](#_Toc510955685)

[Stack.py 154](#_Toc510955686)

[Testing 156](#_Toc510955687)

[Evaluation 157](#_Toc510955688)

# Analysis

## Introduction to Organisation and Client

Mr McMurray is a freelance graphics designer who wants to begin developing games. His background in graphic design mean he specialises in developing smooth and aesthetically pleasing interfaces. However, whilst he has been learning how to program and develop games on a fundamental level, his technical ability is still growing, and his mathematical ability is not as proficient as he would like it to be for some parts of this development.

Knowing me from secondary school, from being in the same IT class, he remembers our contrasting interests – his in the more design focused aspects, mine in the more technical side – and has contacted me to see if we can collaborate on this project of his.

## Outline of the Problem

His first game is going to be a casual Standard deck card games game. The most technical aspect of this project is the development of a competitive AI, which the user can effectively practice against, with varying levels of difficulty. The developer is looking to delegate this objective to someone who has a deep understanding to the mathematical aspects and concepts underlying. Specifically, the developer wants the AI to have different general personalities (in a game context: aggressive, defensive), as well at higher difficulties to have varying personalities based on the situation.

For example, one of the standard deck card games which will be a part of the game will be poker – in this context, the client would like the AI to have a personality where it is aggressive - raises a lot to attempt to intimidate the opponent for example - and another where it plays more passively – folds more often, and raises less often. In addition, when at the higher levels, the AI would play more aggressively when it has a higher bank, or it thinks the opponent is on tilt, and in other scenarios it would play more passively. Another example is Blackjack, where a more aggressive AI would be more likely to hit at a higher hand value in order to obtain a larger lead, and on the flipside a more passive AI would be more likely to stand with a smaller lead to not risk going bust.

The current system, if it were to be developed by the client, would consist of simple logic, and transparent decisions, where the AI would eventually become predictable and the interest from the end users in the game would drop very fast. Under the current system, the client would have to spend a lot of time explicitly programming the AI, in order to accommodate for specific scenarios, to emulate the nuanced decision making of a human. As a result, the client would like the AI to be competitive and a challenge for the end user, as well as this the game development deadline is 6 months, consequently, the client requires the development time to be kept reasonable to this deadline.

## Investigation

**Interview with Primary Client – Mr McMurray**

**Specifically, what is the game you are thinking of developing and which demographic are you aiming it towards?**

It’s good to talk to you again after all this time! I hope you are doing well.

Well, I have an interest in Standard Deck Games I just find them so fun! However, a side me of dislikes the potential gambling aspect in the sense that someone could destroy themselves over a game, and sometimes people just have not had enough practice to start playing for real – my goal is to develop a game for these sort of people, more of a casual base. I just want people to have fun, and I hope to keep them safe whilst they do it, there’s a bit of a vocational aspect for me personally.

**What are the central focuses of the game you are trying to develop?**

You know graphical design is where my strengths lie, so, I want to practice where the skills overlap, so I am thinking more of a focus on the UI. But, I also realise how technical this field can get, so I want to start learning the basics of the logical process, and some parts of web development. The catch is, with a game like this I do not want some of the more casual users to be put off by pressure of having to play against other real players, if they do not want it, so, I want to develop an AI which they can play against, but I am currently struggling with this more than anything else in the project.

**From your statement, it sounds like you have attempted to implement this AI, which suggests that you have design elements in mind. What, at the moment, are the central design elements or features you would like for this AI?**

I think I have a pretty good vision of the AI. I want this AI to have dynamic difficulty levels – I want it to be competitive, but not exceedingly difficult, to the casual user especially, but I also want it to provide a challenge for the more experienced users. After all, in my humble opinion, there is no fun if there is no challenge.

Also, an exciting part for me is designing this game with a compelling caricature aesthetic, with characters who have striking personalities. As a result, I would like the AI system of this game to have differing personalities to reflect these characters. For example, I am designing a half-man, half-bull character to be in the game, the AI for this character should have a more aggressive personality than usual, to reflect these characteristics.

**What is the current system you have, or are in the process of implementing?**

Ahh, it’s not very good, but so far, I have a simple pattern matching system, where the AI checks the state of the game, and then it checks against conditions I have programmed in, it then follows the corresponding action. Just a long if-elif-else chain, really.

**What are the benefits of the current system?**

Well, although it is not as good as I would like it to be, it does work on some level, I suppose. It can play a simple game, and sometimes it is kind of fun to win, however, it can get really boring, really fast. The only real benefit from a developing standpoint is that I can easily add more conditions for it to check .

**What are the drawbacks of the current system?**

The big problem, really, is that it is very easy to pick up on the patterns the AI follows, I’m not sure if I am a bit biased, because I know the conditions it looks for, as I am the one who made them, regardless, it is too predictable at the moment. I’ll give you a quick example, it’s very similar to the way the dealer plays in that it will always stand when getting to a value. It is so easy to exploit! Whenever you’re getting to those late stages in the game, you know exactly what it’s going to do and it’s hard not to use that to your advantage, after a while it makes the game kind of boring.

Although, one good thing about this is it is easy to change the aggressiveness of an AI like this, because you would think a more aggressive AI would, in general, stand at higher values, so you can turn up the aggression by making it stand on higher values only. Obviously, I would like it to work more subtly than this, but it’s something. Also, the AI will only ever be as good at games as I am, because I cannot program to check scenarios I am not aware of.

Lastly, although a secondary issue, the AI currently does not exhibit the personality types in the way I desire. Like it has a very superficial aggression system as explained above, but this cannot be the system to use, because it has the massive downside of behaving the same way without variation, no matter the condition of the game or the opponent.

Moreover, to program this in would require a lot of time, because, I would have to program a different reaction to each scenario – so, say I want to have 3 personality types, for each pattern the program recognises I would have to program a different decision or reaction for each of these types. Furthermore, the AI would require a lot of maintenance, due to the nature of having to account for a lot of different scenarios – even if one common scenario is not accounted for, the AI is exploitable and would require constant patching, lowering the overall quality of my product. It’s a lot of hassle at the moment.

**Are there any final points you would like to add?**

The development of this AI is stunting the progress of the development of the final product a lot more than I would like. Preferably I would love to collaborate with someone, so that I can focus on developing the skills I want to focus on. In addition, ideally I would like the AI to not have the current maintenance aspect which it currently has, the best case scenario would be for it to be self-sufficient – by learning from its mistakes, for example – so that it is of an acceptable quality, even on the launch date.

## Analysis of Investigation

Overall, the system needs to be a dynamic implementation of a dynamic artificial opponent, appropriate for players of a wide skill range. It needs to be dynamic in the sense that it can adapt to the player’s ability, and playstyle, as well as having a low maintenance requirement – so that it is possible for a small team to maintain. In addition, to fit with the quirkiness of the game’s narrative, the system needs to have different playing “personalities” to fit with the personalities of the in-game NPC personas.

From this interview it appears that the client would primarily like the AI component of the development to be delegated. Firstly, a quality of the AI which the client currently likes is the aspect of its extendibility – “I can easily add more conditions for it to check against.” whatever the new system will entail should keep this aspect of extendibility.

On the flipside, the main issues with the current system appears to be its effectiveness, specifically, the AI is “it is too predictable at the moment” and “It is so easy to exploit!”, from this, we can reason that the new system needs to be more nuanced so that it cannot be predicted quite as easily. The example which is given is that the “it’s very similar to the way the dealer plays in that it will always stand when getting to a value”, so one specific improvement could be to add a calculation element where the AI puts considers the probability of going bust, based on the cards already played, then proceeds to hit, based on whether or not the chances of going bust exceed a threshold, and this threshold could be adjusted (rather than just the raw hand value) to create a more subtle aggression system. Specifically, this issue of nuance is exhibited in the fact that the current system will always make the same decision in the same explicit scenario.

The current system is programmed using a “a simple pattern matching system” where the developer has preprogramed the AI to recognise certain states of the board – supposedly the data is taken straight from the game, since the product is of his own development – and then execute the preprogramed action in reaction to this pattern.

The current system works like this:

The big aspect here is the checking of the different edge cases/patterns – all the nuance in the program relies on having a robust case checking system full of a lot of different patterns. As the user states, this system is difficult, because it requires explicit maintenance of each different scenario the game could be in, and if one case is not accounted for and it occurs often enough, then the entire game is exploitable, which ruins the quality of the overall product. From this, we can infer that a better system would to develop an AI which does not need explicit programming to recognise different features, so that it can implicitly adapt to each new scenario, and develop its own nuance.

State\_of\_hand <- get\_hand()

…

Get state of all aspects of game

…

IF Current\_Hand\_Value is High THEN

STAND

ENDIF

…

Check more edge cases/patterns

…

ELSE   
 HIT

On the flipside it is important to make sure that the AI has differing difficulty levels, so that the difficulty of the game matches the ability of the current user, so that the user and the bot can remain in a competitive game, so that the users of any skill can enjoy the final product. A subtle aspect of this will be that it is more important for the final AI to be able to play less skilled users than users of high skill, because the client states that he wants to orient the product to “more of a casual base” – in addition, users of higher skill would most likely be playing on professional online gambling websites.

Another big aspect of the AI which the user would like implemented, is the feature of “personality”. The client states that his game has a style of “compelling caricature aesthetic”. The example he gives is of a half-man, half-bull character. The consequence of this for the AI is that the AI should reflect these personalities in game – for example, the half-man, half-bull character would be more aggressive than some of the other characters.

**Main Points:**

* The current system is too predictable, there is no challenge for the user once they figure out the very linear behaviour patterns
* Whilst it is easy to extend the current system by providing more nuanced scenarios to check against, this would increase the maintenance requirement of the system by too much.
* The client would like an aspect of playing personalities in the final system, which would synergise with the caricature NPCs being developed for the more story side of the game.
* The user states that they would like to develop a game which includes a variety of Standard Deck games, but from the interview it appears that he has started his focus on Blackjack – perhaps as a prototype – so I will also continue with Blackjack in the same manner.

## Objectives

1. First and foremost, the biggest issue the user has with the current system is that it is too liner, and therefore, easily predictable and not a challenge. Consequently, the AI solution I build must be more dynamic and a reasonable challenge for the average projected user. This will be measured by
   1. Not following a superficial pattern in critical stages of the game. Elucidating this further, this means (in the context of blackjack)
      1. Not always standing at the same hand value, regardless of chances to go bust, or the user trend. Instead it should analyse the current game state and other trends it has access to, to make the move it calculates as the best for the given scenario.
      2. But also, not giving up the strictly dominant strategy in trade for unpredictability – this means, for example, that the solution agent should still always hit when it is impossible to go bust, even if it makes the behaviour of the agent predictable, because this is the strictly dominant strategy for this scenario.
   2. The system should have access to previous information about each user, and be able to generate trends and measurements about each user and their playstyle.
      1. It should be able to determine the user’s skill level, based on their contextualised winrate (for example a user who plays games with more players has a lower chance of winning in general, because there are more people who would win, and there is only one winner per game), and the skill ratings of the users they typically play against
      2. Each user should be classified into an aggression category.
      3. The system should then utilise this analysis about each user and dynamically change their parameters to suit each user. For example, a lower winrate permutation of the AI may be more suitable to a user which has a lower winrate, so that the end user can still enjoy the game and vice versa for a higher winrate user.
2. Externally to playing the game, the AI needs to be configured to play with differing playing personalities:
3. The pre-set personalities which the user has requested are: basic aggressive, and basic passive. (For more detail read the research section)
4. Aggressive personalities should exhibit behaviour which shows that the AI is more likely to bet more, raise more often, and fold less often, hit more, etc. compared to the base level version of the AI.
5. Passive personalities, in general, should be more inclined to perform slow plays (aka Sandbagging or trapping[[1]](#footnote-1)), this is where the AI would be more inclined to bet weakly, or stand despite having a strong hand, in order to deceive the opponent. This slow playing style should be more likely to be exhibited compared to the baseline AI.
6. The mechanics of the personalities should vary based on the difficulty level.
7. At the lower difficulties, these personalities should reflect emotions – for example, after losing a big hand, the program should emulate a level of “tilt” –meaning the system would not perform as much of an optimal play, after losing a few rounds - contrasting this to if the AI had won a big hand, after which it would “be more confident” and be more likely execute the optimal play.
8. In addition to this, at higher difficulties, the AI should not have a preset personality, but a dynamic style which changes based on the state of the game. Also, the program would be less inclined to exhibit an “emotional” play style – for example, it would be less inclined to go on “tilt” after losing a big hand.
9. There should be an interface to configure the personality of the AI outside of the game, however, this should be restrictive to only external to any game.
10. There is potential for the AI to be trained, if this is applicable, then the program needs to fulfil the following requirements in this regard:
11. The program should be provided to the user pre-trained, and ready to be implemented into the game that they are creating.
12. The AI should use each new game it plays to add to its training, meaning that each game it plays will add to the proficiency of the bot. However, it should only train itself after a batch of 50 games, for example, as to not be influenced too much by anomalous games.
13. An interface should be provided, to allow for the client to add their own store of training games, if they wish to train the AI further.
14. The training data, as well as its algorithm should not be stored with the AI, in order to prevent potential malicious users from attempting to change its behaviour.
15. If the AI plays enough games against a significant user, which dominates the AI, the AI should attempt to identify the playstyle of this player, and retrain itself to play against this style in particular, then apply this new training either to this player, or players with a similar style, if the training it already has is not adequate.
16. To fulfil the other requirements of adapting to the user, the program needs to have a method/interface for identifying each user. This can either be built into the program, or be provided by the client. This involves, at some point, the user being able to log in and be provided with a unique account, in order to identify them. This includes:
17. A sign up system, where the user can provide a username and password, to create an account.
18. A log in system, so that previously existing users can sign into the program, with a record of their previous games.
19. The username should be unique to each user, and this should be checked and sanitized during the sign up process.
20. The password will have minimum requirements of at least one capital letter and a number, to ensure the user does not allow themselves to have their account compromised.
21. The username should be sanitised, to prevent attacks on the system. However, the password should be hashed and the hash value should be stored, rather than the password itself, as a result this does not need to be sanitized.
22. Relating to the last objective, keeping a record of past games is a good way for the end user to reflect on previous decisions and improve, however, there is potential for the AI to use this history of games to improve its own performance. As a result of this double need, there should be a method of storing data about each game played.
23. A record of each game should be stored in a database – with each record being an account of each game played.
    1. This record should store a unique game id, a series of winner ids, a series of winning hands with corresponding winning values, the number of turns each game was, and a list of players who played in each game.
24. There should be a relationship between three tables – a table of users, a table of moves; which will contain information about each move in the game, and lastly a table of games, connecting multiple users to a single game.
25. This database should only be writable by a central server which is hosting the game, every user and the AI should only be able to read from it. Additionally, the users should automatically have access to each game they are involved in, and then any other games which they download externally. This access should be locked whilst a game is in play.
26. The end user would like to gain a deeper understanding of the dynamics of different playstyles, and agent types in a game setting. As a result, he would like a tool which can be used to get different data about different games, and then parse this to output easily understandable information about how different players or agent parameters lead to different winrates or behaviours.
    1. Firstly, this comparison tool should be able to parse data provided to it from a database. In the design section, the design of this database, will show the format of data this tool should be able to parse.
    2. The tool needs to be able to output information about players or agents both in isolation, and in relation to other players
       1. In isolation, the tool should provide data about the general winrate of the agent, as well as common statistics: for example, for blackjack information should be calculated about what value an agent or player likes to stand, on average.
       2. In relation to other users, the tool should output data about winrates, in both a specific and general way.
          1. Specifically, if there are standout users that the user being analysed has a high play-rate against, or a high win-rate/loss-rate against.
          2. In general, if there is a particular classification of playstyle which the user is particularly good against, or particularly bad against, then this information should be calculated.
          3. On the other hand, if the player is particularly balanced in terms of their win rates against different playstyles, then this should also be displayed.
          4. The change in data over time should be calculated, so that any trends in the user behaviour can be output.
       3. In relation to other users, a hidden MMR (Matchmaking rank) value should be calculated so that users can be matched up with people of a similar skill level. The different difficulty of agents should also get their own MMR value, so that it can be easily predictable what skill of user they will be matched up against
          1. The user should have a required number of placement matches before they are given a viewable MMR
          2. This MMR value should be updated after every game, and should increase with a win, and decrease with a loss.
          3. The relative change of a user’s MMR should be scaled with the difference in MMR between players – for example, if a low ranked player defeats a high ranked player, they should gain more MMR than they would playing against people of a similar skill level.
          4. Users should be matched with other players of a similar skill level.
    3. This all the information calculated by this comparison tool should be output in a visual way so that the user can easily understand the information being shown.
       1. For isolated information about a user, a line graph, or similar chart, should be output, so that the change in the user’s behaviour over time can be plotted.
       2. In relation to other players, a graph could be output, which would show the relationships of playrate and winrate between players, where thicker lines would show a stronger connection.
    4. This tool should be intuitive for the user to user, as a result a UI should be produced which would allow the client to navigate the tool and choose different parameters. This tool should allow the client to enter a user’s id, and choose which different aspect of the user the client would like to view – such as winrate, or best/worst playstyles to play against.

## Data Flow Diagram of Game

Regardless of the nature of the final system, the nature of the game with stay consistent. It is important to lay out the data flow for the game.



## Data Flow Diagram of Current System

**Level 0 DFD:**



As far as the current system goes, this as detailed as it will be. Within the process of “Calculate System Position Value and Generate Move”, a series of if statements are utilised to generate the calculation and move.

## Research

A lot of Standard deck games, such as blackjack, are games of incomplete information – the complete state of the game is unknown at any given time – unlike a game such as Chess or Checkers. Consequently, this makes designing an AI with concrete, non-flexible calculations and pure mathematical models a less optimal solution. This holds especially since the opponent for the AI will be a human player who will be able to spot these patterns and adapt their own playstyle, and exploit the AI, if it remains linear – this problem is exaggerated especially in the current system.

A decent method would be to find a database of hands for different games, whilst there are not many databases for Blackjack, there are ones for games like Poker: such as the UCI Poker Hand Data Set[[2]](#footnote-2)(dataset of hands for 5 card draw, over 1 million instances) or Michael Maurer's IRC Poker Database[[3]](#footnote-3) (University of Alberta), then use statistical analysis to determine the common patterns or behaviours. One way to get around the issue of not having a statistical store of Blackjack hands is to generate my own, however, this would require a bit of time post-launch to become effective against human players, and this product needs to be launch ready. Another way to get around this is to generate a store with a variety of Ai systems, which may provide enough variety in data samples and behaviours to be effective.

The biggest issue with this is that the data I have found so far is not contextualised, and the dataset does not contain the moves made by each player. For example, the data could have come from a high rolling club in Las Vegas, or a million instances of different kitchen table games, as a result, it is hard to add any context to the analysis of the data. In addition to this, the most the data could be used for is to analyse which hands are most likely to win – it does not provide any information for general player behaviours; this issue which stems from this is that using solely data analysis to calculate the behaviour of the system is that it may become too linear and play in one playstyle – for example, only betting when playing with a big hand. In addition, these datasets will generate a weakness in the system in the sense that it will not be able to reason about other player's behaviours, and adapt to them, whereas, some human players will be able to reason about the system's behaviour of using purely statistical information to inform its play, and perhaps outplay the system too easily.

Regardless, the fact that there are datasets already available makes it easier to provide some sort of baseline to the system I will develop, and still may be useful. Some concepts to apply to this information may be Bayes Theorem[[4]](#footnote-4), and Nash equilibrium[[5]](#footnote-5); these are game theory ideas which could be applied to my system to make it more effective at playing the game. Beyond that, within the game, decision trees and minimax trees to use within the game, despite incomplete information, these can be used to model possible future scenarios and decide which move may lead to the highest value position.

Another aspect which would be explored is the possibility of using random states to explore the trends of different games, as an incomplete information set from any one game makes it hard to provide concrete and reliable analysis. One such method may involve using the Monte Carlo Simulation[[6]](#footnote-6), and generating my own dataset via an AI simulating many games against itself. Whilst this does, to a degree, amend some aspects of the system not being able to analyse aspects of opponent’s behaviour as the AI could be preprogramed to play in a certain way (such as tight passive, etc. Discussed later), and then the actions taken by each playstyle can be analysed, and extrapolated to assume that a human opponent would behave in a similar way, if they were to follow a similar playstyle; the problem is that a human opponent may not play exactly the same way, or any of the ways that the pre-programmed AI may play, in addition to this, this adds an extra dimension of initial opponent behaviour analysis, in order to determine which style category, they may fit into.

In addition to this, another possibility could be explored fitting with the idea of exploring random initial conditions. Neural networks have been used in conjunction with an evolutionary algorithm, or reinforcement algorithm and made to play against itself to develop a unique playstyle and metagame[[7]](#footnote-7), this microcosmic metagame which the AI derives from itself may throw human opponents off, making it more effective[[8]](#footnote-8). Utilising neural networks may be a useful way to find an optimal behaviour to this problem, with the case of incomplete information, without the need for unreliable calculations based on speculation on the part of the system (as the system in general would not be effective at analysing behaviours and extrapolating this to value of future hands). This could be implemented in different ways – for example, a baseline weight for the system could be developed, and then used within the game, but then developed further against new opponents, in order to generate more effective weightings for their particular playstyle.

There are a few different options to implement a system like this. One possibility is to use a matrix library, such as NumPy[[9]](#footnote-9), and then design and implement my own neural network. This is a viable solution, as long as the matrix library in question is efficient in its operations. This would allow me more freedom in my implementation of the network and applied algorithms. However, this would vastly increase development time, and is more likely to have bugs and performance issues, compared to the higher level libraries, as my implementation would not be as optimal as other libraries due to my inexperience. Another option would be to use a higher level library, such as TensorFlow[[10]](#footnote-10), to build a neural network from a higher level, and then build the more specific aspects of my solution around this – for example I could use Tensorflow to implement a neural network and the learning algorithms (such as gradient descent) and then use my own algorithms and manipulations of the data for the personality aspects.

According to Pokerology.com[[11]](#footnote-11), in terms of "personalities" or playstyles, there are generally two spectrum which a player may fit on: Tight vs Loose, and Passive vs Aggressive. Tight players are defined as players who, in general, only play when they have a strong starting hand, and loose players, in contrast, will play more with weaker hands. In addition to this, Passive players tend to fold more often, and call/check rather than raise when they have a strong hand; following this, aggressive players tend to raise more often and risk more chips.

These can then be combined into more specific general playstyles: for example, a Tight aggressive player (colloquially labelled as a "Shark") tend to not play for many pots, but when they do they try to maximise their opportunities when they have a stronger hand. Whilst this model for different playstyles was made from a Poker website, it is still applicable to other games like Blackjack, where moves like betting and folding are analogous to hitting and standing.

Moreover, as humans tend not to be linear entities, their playstyle may change in accordance to their emotion (for example, a player who plays worse after losing many pots in a row may be said to be on "Tilt"), or they may purposely change their playstyle to fit the table they are playing at – consequently, there is an extra dimension of Tricky vs Straightforward, where a straightforward player is more likely to fit better into the theoretical description of their general playstyle, whereas a Tricky player may change it up a lot more. In general, according to this source, aggressive playstyles tend to generate more revenue than passive playstyles, where Tight Passive players is tagged as the worst playstyle, putting their playstyle down to "being scared". Whether this may or may not be the case, all these playstyles are relevant to this problem, because the final product may be suitable for a user who is new to the game and may be inclined to play a more "Tight Passive" style, as they are still learning the game, as a result, this needs to be taken into consideration. Moreover, these concrete examples of different playstyles could be mapped to different personalities, which is one of the objectives of the client.

**Main Points:**

* A lot of standard deck games are games of incomplete information, which means that the traditional AI method of calculating possible moves using minimax trees is not theoretically necessarily the most optimal method.
* UCI Poker Hand Data Set has a large collection of poker hands, along with wins, which could be used for a statistical analysis of the value of each hand. However, this dataset is only relevant for 5 card draw. Other games do not have a datasets, to work around this I could generate my own dataset, from AI games pre-launch, and human games post-launch.
* The Monte Carlo method may be a promising way to explore the trends in behaviour of a system pre-programmed in a particular way (eg. Tight aggressive).
* Neural Networks may be an effective system to explore, because a neural network does not require a huge amount of maintenance after the initial training. Moreover, a different set of weightings could be easily mapped to different playing personalities, based on different reward incentives during training.
* Loosely players can be seen through two different spectrums: Tight vs Loose and Passive vs Aggressive, in terms of play style.

## Potential Solutions

**Extension of the Current System – Pseudo-Manual System**

One potential solution would be to extend the current system. One of the main issues with the current system is that the system depends on the knowledge and understanding of the game of the developer. Consequently, the current system could be improved with deeper patterns to check and a more nuanced patterns to check for.

For example, the current system has the drawback of being too predictable, or its behaviour is too linear, to amend this deeper calculations could be used. Using the example the client gave me, the system, at the moment, will always stand at a particular value, as it determines that hitting would be too risky, regardless of what cards have already been played – ie it does not take into account the absolute chance of going bust. To amend this the system could be extended to add a calculation to determine what the actual chance of going bust is, by including what cards have already been played. Extending the system like this would add a dimension behaviour, and the system would be less predictable – the benefit of this is that it is quite easy to extend the system in such a way (just add more patterns to be checked) as well as this it is a more straight forward design and implementation, as long as the developer understands the underlying theory of the game behind it.

On the other hand, the system still retains of its biggest drawbacks – it will take a lot of time and resources to develop a system like this, because, there are a lot of different possible states for a game like poker and Blackjack, and whilst it is possible to generalise these states, to a degree, if one common scenario is missed out and the system defaults to a linear playstyle, then the system becomes very easily exploitable and has failed its requirements. It is quite hard to depend on the robustness of a system designed like this. In addition to this, it will require a lot of maintenance, because, it is unlikely that all the scenarios will be accounted for on with its launch, as a result the system would require constant updating to account for these scenarios, moreover, until it is all amended, the system will not have fulfilled its goal.

Lastly, to fulfil the requirement of personalities as a part of the system, this would involve even more work as the developer would have to design and implement several behaviours for the same system, based on one scenario – one for a passive personality, one for an aggressive personality etc. This makes this solution less viable, based on the time scope and resources of the project.

Main Points:

* Easily extendable
* Complicated design process.
* Simple implementation, but high implementation time.
* High risk design – hard to reason about robustness
* High Maintenance

**Combination of Current System and Statistical Analysis - Semi-Manual System**

An alternative solution would be similar to the current system, but with some of the fundamental concepts altered. For example, rather than hard-coding patterns for the system to check against, the system could instead use a combination of hard calculations, combined with statistical analysis to determine its behaviour.

For example, when determining the value of a hand, the system could still use a subroutine to determine the theoretical value of its own hand, and the probability of the value of the opponents hand relative to their actions, however, when determining its next move, it could use statistical analysis to determine the probability of the current state of the game leading to a win, and then basing its next action upon that, rather than a simple comparison, and pattern matching system as proposed by the current system.

This could be implemented in two different ways, for example, the system could use decision trees to predict future states of the game, and then based on the average value of those outcomes, and it would make its move. This is similar to the current system, however, rather than hard coding these states in, it would be more self-sufficient, by determining the value of its current position by itself. However, this has the potential to be a very inefficient and slow method, as there are so many states that the game could go into, it would take a long time for the system to analyse all of them and determine their value, especially early on in the game. Moreover, if the system took this approach every time, the decision of the system may stagnate as the value of future outcomes may be quite similar to each other, regardless of the move of the system in certain scenarios.

Another implementation of this analysis would be to have a record of previous games, either played by the system, a separate data store, or a combination, then use this data to determine the likelihood of the current value of the position of the system, based on data from previous games. The benefit of an implementation like this is that the system would be make a move more quickly, as long as the data is stored and can be queried in an efficient way. The drawback of this however, is that this system would require a large data store of previous games in order for this to be effective, or else anomalies in the data would have a large impact in the effectiveness of the systems behaviour. This implementation would only be viable if a large store of poker games and results could be found.

Main Points:

* Combination of current system and statistical analysis, in order to reduce linear behaviour of the system and to make the system more generally effective. Similar to last current system, with less maintenance required.
* Implementation of statistical side can be with either decision trees, with values calculated by the system, or from large data store about previous games.
* This system is more self-sufficient as it could improve with more games, making the system less predictable.
* Implementation for personalities is easier as threshold for behaviour (based on calculated value of current position) can be adjusted to emulate different personality behaviour.

**Neural Network / Machine Learning System (Pseudo-automatic)**

Another potential solution would be to model the problem as a neural network and use this model in conjunction with a learning algorithm to generate a system with learned behaviour; this could also be enhanced with use of hard programmed edge cases, or some extra analysis. I could implement this in two ways: either I could use a linear algebra library or as along as it is efficient in its operations I could design my own network and implement algorithms with more freedom to adjust it, alternatively I could use a library with prebuilt neural networks and then program it to fit the problem, which may be more reliable but the trade-off is that I have less freedom in the design and implementation.

This solution would involve modelling Standard Deck Game AI as a neural network, where the state of the agent’s hand are one set of features and the other known states of the game as other features (such as known cards which have been played, and the visible cards in opponent’s hands). Standard Deck games tend to be games of incomplete information, which makes it hard to calculate accurately the value of the current position of the system, as well as using hard calculations to map patterns with the state of the game to linear behaviours, as a result, a neural network model with well-learned weights could be a better solution for producing an effective AI. Other machine learning options include a classification or polynomial regression system, however, I have chosen a neural network for the time being, because I have found more resources and support to work with online – such as the Tensorflow library, which is quite popular, therefore, has a lot of support behind it.

In conjunction with this would require a learning algorithm. There are a few options in this case, for example, if I could find a large data store of games, I could alter the system to optimise a supervised learning problem, and use an algorithm such as gradient descent, to try an learn weights which are most effective for a certain input state. This has the issue for potentially being a bit predictable, and is susceptible to the data (this entire solution is susceptible to the data, but this one more than others) if it is a record of games which are played extremely non-optimally, then the system will try to emulate this and play equally as bad.

An alternative would be an unsupervised problem, where the system would use something such as reinforcement learning or neuro-evolution algorithm to train the system based on games it has played. The issue with this is that, before the system is viable, it would have to play a large number of games to learn from, this would entail having to play a lot of games against a human user (unviable) or another AI. This to some degree solves the issue, of the requirement of a large data store of poker games, because the system, in this case, could be set up to play games against itself, and then adjust the weights of its network, based on which moves worked and which ones did not – if the number of games played is high enough, this should eliminate anomalous moves, or “cheese” strategies which results in a short term win against an AI, but which would be spotted with a human user – such as maybe all-in’ing every round, to intimidate the opponent into folding. Another issue arises in this case, of being stuck in local optima, to amend this the system can be initialised with many random weights, and then pick the highest effective weighting to apply to the system, or pre-programming some initial behaviour to bring the system up to a given level, rather than just a random level, and letting it train from there.

In relation to the issue of personalities, it would be simpler to implement that in this system, as the system could be retrained with either a random weight, or its most effective weighting from the default training, and change the reward calculation (for example, place a higher emphasis on short term winnings for an aggressive personality), then retrain the system, for each personality.

In addition, when training there are aspects of different exploration strategies, which fundamentally change the AI’s behaviour, because it will have learnt in a different way. Different exploration/learning strategies can lead to different playstyles – for example, e-greedy [[12]](#footnote-12)strategy with a low epsilon value may lead to a more aggressive play style than a more probabilistically weighted approach, such as the Boltzmann approach[[13]](#footnote-13).

Lastly, this system would be the most self-sufficient out of all the proposed systems, because it each game it plays it would be able to add this new game to its training, resulting in a non-linear behaviour, as each batch of games would be used to adjust its behaviour. This would result in a less predictable system which requires less maintenance.

Main Points:

* Model game as a machine learning system with a few features – the low number of features means efficient processing and learning.
* Weights can be learned either supervised (if database of poker games can be found), or unsupervised (reinforcement learning / neuro-evolution algorithm).
* Personalities somewhat simpler to implement – as reward for the system can be adjusted in relation to the desired output, based on personality types (eg. Higher emphasis on short term winnings for more aggressive personalities)
* Self-sufficient as each game it plays can be used to change its behaviour – this also makes the system less linear.

**Other Potential Solutions:**

* The developer changes their design of the final product, to just being a multiplayer game – thus eliminating the need for an AI system entirely. Not viable, because this is the clients choice, not mine
* Buying a retail AI made for playing poker, and adapt it to fit the current system. This is unviable, because prices of Poker bots can be upwards of a thousand dollars – this is not a reasonable price range for the scope and requirements of this project.

## Proposed Solution

My proposed solution is going to be the Neural Network based solution. I have picked this one, because I feel that it would be the best for the incomplete nature of Standard Deck games, for the reasons outlined above. Consequently, this means that the system would be effective against higher skill players, and it is easier to purposely limit a system one it has shown that it can perform better, than picking a less effective method and then trying to push past its optimal ceiling.

In addition to this is potentially the best solution to implement the most difficult, arguably, aspect of the system – the personalities. This can be done by changing the reward system, or exploration strategies (if I use a reinforcement learning algorithm, for example) to correspond to each of the personalities (detailed above). For the other solutions, I would have to adjust with their calculated output manually, and this increases the risk of the system being more linear than desired.

As well as this, for a development team this size, minimal maintenance is paramount, which is another reason why this solution is the most optimal. Due to the nature of the system being able to adapt with beyond the release. Moreover, this quality of utilising a datastore of games to adapt, means that I can use some of the aspects of the statistical analysis solution in conjunction with this solution; I can go further and even hard code some behaviours, if I feel they are fundamental enough, and that the system is not exhibiting them.

I will implement this system using the tensorflow library, because this is one of the most popular machine learning libraries, which means there will be a multitude of support for the library, which makes it very sustainable. Whilst this means there is less sophistication required in the development of the lower level applications of the neural network (such as complex matrix operations and manual implementations of gradient descent) there will still be complexity in setting up the training environment, and reward system. As well as sophistication in implementing different exploration strategies. This is more suitable for a project with a timespan of around 6 months.

Overall, this solution is the most versatile, and adaptable solution, which makes it an optimal choice.

### Proposed Solution DFD

**Level 0 DFD**

The level 0 DFD remains the same, as it is performing the same high level process, however, it will be doing it in a more optimal way.



****

### DFD for Game System



### ERD for Proposed Solution



### Data Dictionary

|  |  |  |
| --- | --- | --- |
| **Users** | | |
| **Field Name** | **Data Type** | **Example** |
| Username (Primary) | Varchar(32) | “mr\_aqa” |
| Password | Varhchar(256) | 333c9fbfa920a452d5a3af8d5cb65ee4e23a024859f077e69  dea4f504b532c16:db794813d55746348418355509bd71bb |
| Games\_won | Integer | 0 |
| Games\_played | Integer | 100 |
| Type | Varchar(32) discrete values acceptable | “user” or “admin” -> Only acceptable values |

This table is what will be used to store login information for human users. There will be a user’s class which will handle all the behaviours required for a user to login, for example, there will be an algorithm to check if the entered password is correct, as the stored password is hashed and has a salt. The type of user is important, because for the comparison tool admins will have all access, including data involving the agents, whereas a normal user will only have access to trends of themselves and other users, this is important because it increases the maintainability of the agents, as users will have a harder time deciphering the trends of the agent’s actions.

|  |  |  |
| --- | --- | --- |
| **Agents** | | |
| **Field Name** | **Data Type** | **Example** |
| Agent\_id (Primary) | Varchar(255) | “nn\_ai” |
| Description | TEXT | “Neural network based agent” |
| Games\_won | Integer | 0 |
| Games\_played | Integer | 100 |

Here is the table for storing all the information about an agent, its primary purpose is to keep a record of all the different types of agents currently implemented, and their win rates, as well as a description describing how the agent is implemented.

|  |  |  |
| --- | --- | --- |
| **Game Record** | | |
| **Field Name** | **Data Type** | **Example** |
| Game\_id (Primary) | Integer | 1 |
| Winner\_ids | TEXT | “mr\_aqa;nn\_ai;” |
| Winning\_hands | TEXT | “(5 of HEARTS), (10 of DIAMONDS); (5 OF DIAMONDS), (10 of SPADES)” |
| Winning\_values | Integer | 15 |
| Num\_of\_turns | Integer | 3 |
| Players | TEXT | “mr\_aqa;nn\_ai;simple” |

This table is the record for each game, it stores information about the game, such as the people/agents player, who won, with what hands and what hand value.

|  |  |  |
| --- | --- | --- |
| **Moves** | | |
| **Field Name** | **Data Type** | **Example** |
| Player\_id (Primary) | Varchar(255) | “nn\_ai” |
| Game\_id (Primary, Foreign -> Game\_Record) | Integer | 1 |
| Turn\_num (Primary, Foreign -> Game\_Record) | Integer | 2 |
| Next\_best\_val | Integer | 15 |
| Hand\_val\_before | Integer | 10 |
| Move | BIT | 1 or 0 (1 => hit, 0 => stand) |
| Hand\_val\_after | Integer | 15 |

This table is linked to the Game Record table and stores information about each move made in the game. It stores information about who made what move, and in what context. This table will be the most important when it comes to the analysis of each player, because it gives specific details about what move a person made, and what the context of that move was – this allows us to analyse properties of a player such as the aggressiveness of that move.

|  |  |  |
| --- | --- | --- |
| **Card\_Counter\_Record** | | |
| **Field Name** | **Data Type** | **Example** |
| Game\_id (Primary, Foreign -> Game\_Record) | Integer | 1 |
| Turn\_num (Primary, Foreign -> Game\_Record) | Integer | 2 |
| Bust | Float | 0.50 |
| Blackjack | Float | 0.05 |
| exceedWinningPlayer | Float | 0.60 |
| alreadyExceedingWinningPlayer | Bit | 1 or 0 (1 => True, 0 => False) |
| move | Bit | 1 or 0 (1 => hit, 0 => stand) |
| trained | Bit | 1 or 0 (1 => already been used for training, 0 => has not been used in training) |

Finally, this table provides more context about the state of different AI in the game which utilise a card counting tree – this provides more insight into the state that these AI were basing their moves on, and allows for deeper analysis. It is also used to track the state of the features of the neural network at different points in each game, so that it can be fetched and used to train the neural network, beyond its initial training.

### GUI Structure

Update NN

(admin only)

Stand Value Distribution

Hit Value Distribution

Bust Rate Against Hit Value

Win Rate against Stand Value

Aggression Rating vs Win Rate

Aggression Output Graph of User

Winrate Output Graph of User

Generate Data  
(admin Only)

Log in / Sign up Window

Generate Data options

General Statistics

Isolation Comparison

### Proposed System Neural Network Architecture



On each of the connections of the neural network, there will be a weighting applied, which will correspond to how much “weight” that neural connection has in the output of the network, it means that the network applies a number to how important it believes a given feature is. Here the neural network will output a number between 0 and 1 for both the hit and stand output nodes. A max function will be used to determine which of these is higher, as that will correspond to which move the AI calculates as the better move. This is better than just one output node, because based on the difference between these output nodes, a confidence in the move the AI is about to make can be determined to make the system more sophisticated, and will also provide deeper insight into analysis of the agent’s moves.

An activation function is the function which takes in vector Z - all the inputs of previous connected nodes and weightings and then multiplies them to get the vector Z – these are then fed into the activation function. Although it may sound redundant, the activation function maps vector Z to how activated the node is, these activations are then fed into the next layer, and so on until it reaches the output. Activations can be intuitively thought of as how important a node is considered in the action of the agent. For example, say the hit output node had high weightings for the first and second node of the last hidden layer, but then lower weightings to the other nodes, the activation function will take this into account and then cause the hit node to have a higher value if those with a higher weighting have higher activation outputs.

The activation function for each neuron will be a rectified linear function. The activation function in this starting phase of the project is not as important as other parts, such as the features chosen and the architecture – either the sigmoid activation function or the rectified linear function would have been acceptable.

However, I have chosen the rectified linear function, because the sigmoid function has a property of a vanishing gradient (you can see at the extreme ends of the function the gradient tends to 0), this is a problem if the architecture ever grows to include more layers, because these gradients will be multiplied together, so the output gradient will very quickly drop to 0. If this occurs, the network will be stuck in a local minimum, as backpropagation and gradient descent converge to a gradient of 0, as it is usually the optimal weighting, however, this would not be the case in a network with many layers. In effect, ReLU makes the system more extendable for the future.



The output may then be fed to another function to adjust it for difficulty levels and personality types. For example, at lower difficulties there may be a probability that the lower activated output is chosen over the higher one – ie the AI takes the less move which it calculates as less optimal.

The structure of the hidden layers being the same as the input layers, and the number of hidden layers is arbitrary and different architectures will be tested. In general, on hidden layer with the same number of nodes as the input layer is not a bad option[[14]](#footnote-14), but with a larger time span, other methods could be used to determine the best neural network structure[[15]](#footnote-15).

## Prototyping and Critical Path

The most difficult part of this project will be developing a competitive base AI, which can play the game to a level which is suitable to the user. As a result, this is the first part of the program which I will prototype. Although the system eventually ideally would have variations for many different Standard Deck games, a good game to start with would be Blackjack, because this is the game the client has already started with, and it has a healthy amount of nuance, but not an overwhelming amount of psychological aspects or nuance.

To do this I will first develop the neural network, and train it using a reinforcement learning algorithm, by having it play against another iteration of itself with different randomly generated weights which will also be improved with the same algorithm. After a designated number of iterations, I will play 5 games against it and determine if I can find any exploitable or predictable patterns. If this is not effective, I will attempt to develop a new set of weights, either my more iterations, or by using a different learning algorithm, in this case I would use a neural evolution algorithm to develop new weights, and then use the same process to test how effective the AI is.

After this the next most difficult part of the development will be either, to make the AI difficulty adjustable, or to develop the personality types for the AI. I will develop the system for the personality types first, and then use the same testing method to determine if the AI is still a viable opponent, I will also use the same testers for each game to see if its behaviour is different compared to previous games played against it – to see if its behaviour has changed. In addition to this, I will also have the client play same games against the AI to see if the different personality types I have put in play fit in with what he wants from a design perspective.

One this has been completed, I will develop the system to adjust the difficulty of the AI. I will develop this part, and then just use myself or another new player to Poker to determine if the AI is within a competitive difficulty. As it has already been tested from a baseline perspective against a more experienced poker player before this, it should already be competitive at the highest level this AI has been designed for, therefore I would only need to test it against lower skilled opponents. Essentially this is like maximising the playing level of the AI and then limiting it for more casual users, as this is easier than developing it for casual users and then attempting to increase its ceiling. In addition to this I will implement a system for the AI to detect if the difficulty is skewed for its opponent, and to adjust its difficulty automatically if it is. To test this I will set it to its maximum level, and then play against it and either make purposely bad moves, or just play normally, and see if it adjusts its difficulty level after several losses in a row.

The next most challenging part to develop will be the adaptation against particular users, although this is arguably more challenging to develop compared to than the difficulty adjustment, this is a less important aspect compared to that more fundamental objective. To develop this, I will have to develop the database which will store the games played, and the users played against. Once this has been set up, I will design a log in system for the user, and then play against the AI with a particular style in mind, to see if its behaviour changes, if it does then I can have confidence that it fills out this objective, to some degree.

After this, the bulk of the system should be prototyped and tested. Therefore, after this the full interface should be developed, tested and polished, so that the AI is ready to be deployed, without opportunity of being potentially exploited, due to an interface which provides too much access.

# Documented Design

## BlackJack Prototype

### Introduction

The client has designed his game around Standard Deck card games in general, however, in the interview he made it clear that he started with Blackjack as the intial game. As a result, I have started out with designing and prototyping an AI which can play against a dealer, to a competitive level. Here, competitive level is evaluated on the winrate of the AI over 1000 games, with 45% and higher considered a good win rate for blackjack[[16]](#footnote-16). Blackjack is a good game to prototype the AI design, because it has a lot of overlapping elements with other games like Poker:

* They are both games of incomplete information – it is theoretically impossible to calculate the perfect move, as you never have full information about your opponent’s position, you can only infer it from their behaviour
* Both are games of progressive stages – the game final position of each player is not known until later stages of the game, when each hand has been fully dealt.
* Both have betting aspects, however, blackjack is slightly simpler with less rounds of betting, mostly one round, an optional second round if the player decides to double down, whereas poker has several rounds of betting.

In addition to this, it is a good game as a starting point to being prototyping, because it is also simpler, because:

* In blackjack, there is only one opponent: the dealer.
* The dealer behaves in a predictive way: they will hit until they reach a hand value of 17.
* There are less total cards in play, and to keep track of (If this is a strategy used by the AI)
* The way the value of a hand is determined by each card’s numerical value, whereas in poker a hand’s value is determined by what combination of cards the hand and the community cards match.

As a result, I have begun with designing and prototyping a game of poker, where I will utilise similar elements from the chosen design of the final system, and adapt it to blackjack.

### Design – The Blackjack Game and Interface

I have developed my own environment for the blackjack game. The details of the methods have been provided to me by the client:

As of January 2018, this is the current composition chart for the blackjack game.



As well as the current Inheritence Chart:



#### Blackjack Class Design

|  |
| --- |
| Blackjack |
| - Deck : Deck Structure  - Royal Values : Hashing Structure (Such as Dictionary or HashMap)  - Blackjack : Int (Constant)  - Player Hand : Hand Structure  - Dealer Hand : Hand Structure  - Bust : Boolean  + Continue Game : Boolean |
| + Constructor  + Reset Game  - Evaluate Hand Value (Hand : Hand) : Int  - Determine Ace Value (Hand Total : Int, Number of Aces : Int): Int  - Compare Hands (Hands\* : Hand) : Boolean or Int  - Deal(Hand : Hand) : Hand  - Deal Dealers Cards at End : Hand  + Hit : Hand  + Stand  + Output Current Game State to Console  + End Game |

Below are descriptions of the attributes and the methods described above.

**Methods:**

|  |  |
| --- | --- |
| **Method Identifier** | **Description of Method** |
| Constructor | Declare and Initialise all attributes and constants. |
| Reset Game | Restore Attributes to starting values |
| Evaluate Hand Value | Return the Value of the Hand |
| Determine Ace Value | Return whether the Value of the Ace would be better as 1 or 11 |
| Compare Hands | Compare hands passed in and determine a winner or a draw. |
| Deal | Deal the initial two cards to each player |
| Deal Dealers Cards at End | Deal Cards to Dealer until their hand value is greater than 17 or bust. |
| Hit | Deal a card to the player. |
| Stand | End the Game |
| Output State of Game to Console | Display the state of each player’s hand, and value. |
| End Game | Call the methods which determine a winner at the end of the game: Deal Dealer’s Cards at end, and Compare Hands. |

**Attributes:**

|  |  |
| --- | --- |
| **Identifier** | **Description of Purpose** |
| Deck | Stores the Deck of Cards to Draw from |
| Royal Values | Dictionary, storing the values of each of the royal cards. |
| Blackjack | Constant holding the value for a blackjack (21) |
| Player Hand | Hand structure containing the cards currently in the player’s hand. |
| Dealer Hand | Hand structure containing the cards currently in the dealer’s hand. |
| Bust | Boolean holding the state of whether the player is bust or not. |
| Continue Game | Boolean holding the state of whether the game has ended or not. |

The behaviour of this prototype will be discussed later. First let me present the data structure design used in this prototype, and potentially for the poker prototype also.

Updated January 2018:

Due to the fact that I updated this specification to include more encapsulation for the hand class and the behaviours associated with the hand, some of the functionality has been removed from here and added to the hand class. In addition to this, I have modified the behaviour of the blackjack class so that it keeps the deck between games, until the deck runs out of cards, at which point the deck is reinitialised and shuffled.

Below is the updated class diagram:

|  |
| --- |
| Blackjack |
| - Deck : Deck Structure  - Royal Values : Hashing Structure (Such as Dictionary or HashMap)  - Blackjack : Int (Constant)  - Players Queue : Circular Queue of Hands / Players  - Winners : Hand[]  + Continue Game : Boolean  + Deck Iteration : Int |
| + Constructor  + Reset Game  - Create Player Queue(Structure : Hands/Players)  - Compare Hands (Hands\* : Hand) : Boolean or Int  - Deal(Hand : Hand) : Hand  + Hit  + Stand  + Output Current Game State to Console  + Check if Game is Over  + End Game |

Below are descriptions of the new methods or attributes.

**Methods:**

|  |  |
| --- | --- |
| **Method Identifier** | **Description of Method** |
| Constructor | Declare and Initialise all attributes and constants. |
| Reset Game | Restore Attributes to starting values |
| Compare Hands | Compare hands passed in and determine a winner or a draw. |
| Deal | Deal the initial two cards to each player passed in |
| Hit | Deal a card to the current player, determine if they have gone bust or not, and if they have do not repush them back to the player queue. |
| Stand | Update the current hand to stood, and do not repush them to the queue |
| Output State of Game to Console | Display the state of each player’s hand, and value. |
| End Game | Call the methods which determine a winner at the end of the game: Deal Dealer’s Cards at end, and Compare Hands. |
| Create Player Queue | Create a circular queue and push all the starting players to the queue. A structure of starting players should be passed to the method. |

**Attributes:**

|  |  |
| --- | --- |
| **Identifier** | **Description of Purpose** |
| Deck | Stores the Deck of Cards to Draw from |
| Royal Values | Dictionary, storing the values of each of the royal cards. |
| Blackjack | Constant holding the value for a blackjack (21) |
| Continue Game | Boolean holding the state of whether the game has ended or not. |
| Players Queue | Circular Queue holding each player still in the game |
| Winners | Array of Hand IDs holding the winner of the last game played. |
| Deck Iteration | Hold the value of the current deck iteration – if a new deck has been initialised then increment this. |

#### Structures

##### Card

Firstly, the fundamental object of blackjack (and all the games in this project) is the Standard Deck Card. This card structure has a very simple design:

|  |
| --- |
| Card |
| # Value : Integer or Royal (Enum : Int)  # Suit : Suit (Enum : String) |
| + Constructor  + Get Value : Int or Royal (Enum : Int)  + Get Suit : Suit (Enum : String)  + String Casting Override : String  + Equivalence Override (Other Instance in Comparison : Card) : Boolean |

**Methods:**

|  |  |
| --- | --- |
| **Method Identifier** | **Description of Method** |
| Constructor | Declare and Initialise all attributes and constants. |
| Get Suit | Returns the suit value |
| Get Value | Returns the value associated with the Card |
| String Casting Override | Changes the way an instance of card is converted to string – returns the suit and the value of the card. |
| Equivalence Override | Change the way instance equivalence is evaluated to comparing each instance’s value and suit, returning True if those attributes are equal. |

**Attributes:**

|  |  |
| --- | --- |
| **Identifier** | **Description of Purpose** |
| Value | Holds the Value of the Card Object |
| Suit | Holds the Suit of the Object |

For the purposes of blackjack, only the card value is relevant, however, this class can be reused later for the poker prototype. In addition, it provides an easy method for other structures (such as the deck or the blackjack game) to be able to abstract away more detail into this class, making them easier to maintain. In addition, it provides an easy interface for different classes to determine the value of the royals or the Ace, as for every game this will appear as an Enum of Royals, in which case a hashing structure can be used to map each of these cards to a concrete value. This is better than having to hard code different values of the royals to different settings – for example, the alternative would be to hard code methods into the card or deck structure which changes the value of the royal from within these structure, rather than from within the game structure. This would mean having to store unnecessary data and behaviours within the card and deck structure, making them harder to maintain and use.

##### Stack

The stack structure is what I have used to implement the deck structure. I have used an implementation we made in class for this purpose. Its design is shown below. It is implemented backed by an array.

|  |
| --- |
| Stack |
| - Size : Int  - Top of Stack Pointer : Int / Address  - Stack Array : <T> (Data type of stack) |
| + Constructor (Stack Size : Int, Type of Stack : <T>)  + Size Getter : Int  + Push (Element to be pushed : Element of Stack) : Boolean  + Pop : <T>  - Is Full : Bool  - Is Empty: Bool |

**Methods:**

|  |  |
| --- | --- |
| **Method Identifier** | **Description of Method** |
| Constructor | Declare and Initialise all attributes and constants. |
| Size Getter | Returns the value of the attribute holding the value of the size of the stack. |
| Push | Pushes a new item to the top of the stack, returns true if the push was successful. |
| Pop | Pops the element from the top of the stack |
| Peek | Outputs the element at the top of the stack. |
| isEmpty | Returns a Boolean determining if the stack is empty or not. |

**Attributes:**

|  |  |
| --- | --- |
| **Identifier** | **Description of Purpose** |
| Size | Holds the maximum size of the stack |
| Top of Stack Pointer | Pointer pointing to the top of the stack. |
| Stack Array | The fundamental array holding the data of the stack. |

##### Deck

The deck structure is what I will be using the store the cards, and then deal them from within the blackjack game. I implemented the deck as a stack, and utilised inheritance to specialise the Stack behaviour for the Deck context. A stack was most appropriate because the first in, last out behaviour was suitable for this context, as you only need access to the top card of the deck at any instant of time. As the deck inherits the Stack structure, it has all the methods and attributes, which are either public or protected, which are shown above.

|  |
| --- |
| Stack |
| - Size : Int  - Top of Stack Pointer : Int / Address  - Stack Array : <T> (Data type of stack) |
| + Constructor (Stack Size : Int, Type of Stack : <T>)  + Size Getter : Int  + Push (Element to be pushed : Element of Stack) : Boolean  + Pop : <T>  - Is Full : Bool  - Is Empty: Bool |

|  |
| --- |
| Deck |
| - Values : Int Array  - Suits : Suits Array (Suits -> Enum: String) |
| + Constructor  - Initialise Deck |

**Overridden / Deck Methods:**

|  |  |
| --- | --- |
| **Method Identifier** | **Description of Method** |
| Constructor | Declare and Initialise all attributes and constants. |
| Initialise Deck | Generates a queue of all the cards to be pushed to the deck, shuffles them and pushes them to the deck |

**Deck Attributes:**

|  |  |
| --- | --- |
| **Identifier** | **Description of Purpose** |
| Values | Holds all the different values of each card to be pushed to the deck |
| Suits | Holds all the different Suits of each card to be pushed to the deck |

##### Circular Queue

As the game has a variable number of players, a data structure is required to store the players, so that they can be accessed. I have used a circular queue for this, as it has the double purpose of also providing FIFO behaviour, which will provide me with the next player who needs to have their go. This works by popping from the queue at the start of each turn and then, as long as they are still in the game, pushing them back into the queue. Moreover, if they are out of the game (either by going bust or standing) then they are not pushed back onto the queue, so the end of the game can be determined by whether or not the queue is empty.

Moreover, a circular queue was used, because it is more time efficient than a normal static queue. This is because, with a static queue, after each pop all the data stored in the queue has to be moved down the list; this has a big O complexity of O(n), whereas the cost of popping for a circular queue is O(1), as the front and rear pointers have to just be adjusted. The only downside is that an extra pointer has to be stored, but the memory cost of one extra pointer is not significant compared to the efficiency cost of a normal static queue.

A static queue was utilised, rather than a dynamic one, because the number of players in each game is known at the start of the game, more players cannot join, and if a player leaves it does not have an impact on the queue.

|  |
| --- |
| Circular Queue |
| - Size : Int  - Front Pointer : Int / Address  - Rear Pointer: Int/Address  - Circular Queue Array : <T> (Data type of stack) |
| + Constructor (Queue Size : Int, Type of Stack : <T>)  + Size Getter : Int  + Push (Element to be pushed : Element of Stack) : Boolean  + Pop : <T>  + Peek: <T>  - Is Full : Boolean  - Is Empty: Boolean |

##### Hand

The hand structure is what will be used to store the hand of each player. In this case the hand is only ever added to, using the hit behaviour. In this context, an array or linked list are just as appropriate as each other, the direct access of an array is not as significant in this context, however, the maximum number of cards a player can have before bust or blackjack is known (11 – four 1’s, four 2’s and three 3’s), so the dynamic advantages of a linked list are not quite as significant either. However, for the sake of versatility for other standard deck games, a linked list is the best structure, because it will support hands of any size, efficiently; unless you passed in the maximum hand size to the constructor of the structure with the use of an array, however, this makes the structure harder to maintain. Either way the implementation of the hand structure is not as clear-cut as the use of a stack to implement the deck, however, I have chosen a linked-list based structure.

For now, there are no extra behaviours required for the Hand structure. The only attributes would be the cards that the structure holds.

Updated January 2018 –

Upon updating the blackjack class to accommodate for more than one player, I realised it would be a better design to take some of the behaviours built into the blackjack class regarding the hands, and move them into the hand class. Fundamentally the hand class is still backed by an array of cards, however, I have updated the class to include built in behaviours such as getting the current value of the hand and choosing the ace.

In addition to this, I have also added a new class for the dealer’s hand, which holds all the functionality for the dealer, so that it does not have to be in the blackjack class. This class inherits from the hand class above and has some additional behaviours.

The new class design is as follows:

|  |
| --- |
| Hand |
| - ID : String or Hash Value  - Hand : Card[]  - Stood : Bool  - Bust : Bool  - Royals : Hash Structure (Card -> Integer) |
| + Constructor (Id: String)  + Get Hand Value : Int  - Choose Ace (Hand\_Total : Int, noAces : Int)  + Hit (Card\_To\_Add : Card) : Void |

|  |
| --- |
| Dealer Hand |
| (No additional attributes) |
| + Dealer Play (Deck : Deck) : Void |

**Methods Hand and Dealer Hand:**

|  |  |
| --- | --- |
| **Method Identifier** | **Description of Method** |
| Constructor | Declare and Initialise all attributes and constants. |
| Get Hand Value | Calculates and returns the current value of the hand which the instance holds. |
| Choose Ace | Private method which assists in calculating the value of the hand by picking the value for the ace. |
| Hit | Appends the passed card to the array of cards attribute. |
| Play Dealer | Whilst the value of the instance of the dealer’s hand is below 17: pop cards from the passed deck and hit them to the hand. |

**Attributes:**

|  |  |
| --- | --- |
| **Identifier** | **Description of Purpose** |
| ID | Identifier variable used to identify the instance of the class |
| Hand | Array of cards which is the literal contents of the current instance of the Hand. |
| Stood | Tracker variable keeping track of if this player has stood. |
| Bust | Tracker Variable keeping track of if this player has gone bust. |
| Royals | Hashing structure which maps Instances of Royal Cards to their values within the current game. |

#### Blackjack Class Usage

From a high-level perspective, the blackjack interface provides a hand to each player, and the game mainloop continues until each player stands, reaches a blackjack or bust via hitting. Once the game mainloop ends, the end game method is called to calculate the value of each hand and determine a winner. The game can then be reset using the reset method.

Updated January 2018:

Upon adding the player queue, allowing for multiple players in one game of blackjack, the new functionality of the class is as follows.

During the class mainloop, the current player is output, as well as the value of their hand. Then the player or the AI can decide to hit or stand. This hit or stand functionality is done through one function for each of the behaviours, this method has the same behaviour, however, it performs this behaviour on the hand at the front of the queue which will be popped. If the player stands or is bust, they are out of the game and are not pushed back to the queue. The game ends when there is no more players in the player queue.

After this an end game method is called, dealing cards to the dealer until the value of their hand is above 17, and then another method is called comparing the value of each hand, and then outputting the winner.

### Design – Simple AI, Rand Ai and Agents

For the comparison tool to be tested and to work, a source of data needs to be collected. For this to be operational pre-launch, there needs to be a data source pre-launch. To generate data, I have built a blackjack environment, and I have built several different type of AI to play against each other. With this I hope to cover a wide range of playing styles.

Firstly, an abstract agent class needs to be implemented. This defines methods which every agent needs to implement, to be compatible with the comparison tool. These are mostly methods which are called in the game environment, whilst collecting data, so that there is a guarantee that the method being called is implemented for all AI, preventing run-time errors.

|  |
| --- |
| Agent |
| # ID : String  # type : String[]  + Hand : Hand |
| + Constructor(ID : String, type : String[], Hand : Hand) : Void  + ID\_Getter : String  + update\_end\_game() : Void  + get\_move(all\_players: Hand[]) : MOVE / bool / bit  + set\_parameters(setting : String)  - get\_agent\_hand(all\_hands : Hand[])  - get\_best\_hand(all\_hands : Hand[]) |

**Method Descriptions**

|  |  |
| --- | --- |
| **Method Identifier** | **Description of Method** |
| Constructor | Declare and Initialise all agent ID and type |
| Update\_end\_game | Called at the end of every game by the comparison tool, if there needs to be any updates to the state of the agent, it takes place here. |
| Get\_move | Returns the move the agent is going to make for that turn, pass in all the players, so that the agent can get the required game state. |
| Set\_parameters | Pass in the setting (string of discrete aggression values: “passive”, “default” or “aggressive”), sets the setting for the agent. |
| Get\_agent\_hand | Private helper method to get the agent’s hand, when all the hands are passed in |
| Get\_best\_hand | Private helper method to get the next best hand currently in play. |

**Attribute Descriptions**

|  |  |
| --- | --- |
| **Identifier** | **Description of Purpose** |
| ID | Identifies the Agent |
| Hand | Hand structure associated with this agent |
| Type | Array of agent string descriptors – eg. “NN”, “Card Counter” |

**Simple Agent**

The simple agent is a threshold based agent, where simple components of the game state are calculated based on all the hands passed in. These thresholds define the behaviour of the agent, some examples of the thesholds are: bust threshold; how far off the agent is off being bust before the agent will stand, minimum hand threshold; the minimum hand value the agent needs to have before standing; win margin threshold; how much of a win margin must have before standing.

Apart from this, there are no extra features from the abstract agent class for the simple agent.

**Random Agent**

As a control, I have implemented another agent which will implement this agent abstract class, however, for the get\_move() method, it will always a random move between HIT and STAND. I have implemented this class a control agent when generating the statistics later on.

### Design – Card Counting AI and Card Counter

The blackjack dealer AI behaves in a very linear fashion – it will always hit until it reaches at least above 17. Reaching winrate of at least 45% (as provided by my research) is one metric which could be used to assess the performance of the final product. However, another metric would be useful in determining the aptitude of the automatic machine learning system. As a result, a Card Counting AI will be designed and developed alongside it, to compare how well they perform contrasted against each other. This is useful because the card counting AI’s behaviour is more consistent, predictable and easier to dissect, compared to the automatic system.

This AI will utilise a card counting tactic to predict whether it is optimal to hit or stand, based on what cards have already been revealed. The agent will store a record of all the cards in one standard deck and a counter next to each one, which will be decremented as each card is played. From this it will determine the probabilities of the next card it draws resulting in a bust, blackjack or the chance of exceeding the winning player.

Parameters will have to be tested and chosen which will correspond to the threshold that the agent will operate under before making a move – for example, I will test if the AI should hit up to it predicting a 50% bust chance or less or higher.

Update: January 2018

To increase the extendability and reusability of the whole system, I have separated the card counter and the threshold based Card Counting AI, which uses the statistics generated by the card counter. This allows me to reuse the features provided by the card counter – for example, I have used the card counter in the neural network agent, as well as the threshold based agent CC\_AI.

This has lead to the following class designs:

|  |  |
| --- | --- |
| Card Counter | Agent |
| - Card\_Record : Card\_Binary\_Tree() |
| + Constructor() : Void  + Populate\_Tree() : Void  + decrement\_cards (Cards: Card[]) : Void  + calc\_chances(Agent\_Hand\_Val : Int, Second\_Best\_Hand\_Val : Int) : Hashmap<String : Float>  - calcBustChance(Agent\_Hand\_Val : Int) : float  - calcBlJaChance(Agent\_Hand\_Val : Int) : float  - calcExceedWinningPlayer(Agent\_Hand\_Val, Second\_Best\_Hand\_Val : Int) : Float |

|  |
| --- |
| CC\_AI |
| Does implement any new methods or attributes – just implements methods defined by the abstract classes it inherets from. |

**Method Descriptions**

|  |  |
| --- | --- |
| **Method Identifier** | **Description of Method** |
| Constructor | Initialises the binary counting tree |
| Populate\_Tree | Pushes all the different card values to the binary counting tree |
| Decrement\_cards | Pass in all the cards to be decremented from the internal binary counting tree, and decrement these cards from the tree |
| Calc\_chances | Call this and the internal binary tree will be traverse and a hashmap mapping different chance names (eg “blackjack”, “bust” etc.) to their probabilities. |
| calcBustChance | Private helper method which gets the probability of going bust, depending on the internal binary card counting tree. |
| calcBlJaChance | Private helper method to get the probability of getting blackjack, depending on the internal binary card counting tree. |
| calcExceedWinningPlayer | Private helper method to get the probability of exceeding the highest player in the game (not including the associated agent) |

**Attribute Descriptions**

|  |  |
| --- | --- |
| **Identifier** | **Description of Purpose** |
| Card\_Record | Instnace of Card Counting tree – structure used to track the number of each card left in the game. |

#### Structures

##### Complete Binary Search Tree and Node

Storing the cards and the values associated with them are best stored in a Binary Search Tree, this is because as long as I store all the cards in their value order, then searching for each card has a Big O complexity of O(log n), compared to if I used an array, which would result in a Big O complexity of O(n). Although the data set in this scenario is fixed for a small n, it makes the system more extendable.

There are a few ways I could have implemented the binary tree, however, the way I have chosen to do it, is by creating a Node class which essentially stores a value and can point to a left child and a right child, the node will also have auxiliary behaviours (such as get number of children). The nodes of a tree are linked via pointing of the node objects. The Binary Tree class has one attribute – the root node, through which all the other nodes in the tree can be accessed via tree traversals. Most of the methods utilise tree traversals to operate on the tree, or to define a behaviour.

The advantage of this is that it makes some behaviours – such as searching and inserting very quick and easy to program, using tree traversals; at the same time, it makes some other behaviours harder to program, for example, the maintenance of the binary tree. In general, traversing the tree and operating via traversals is much easier to write code for, however, using this system I am limited to traversals, and there is not direct access to nodes, which is why the maintenance of the tree is harder to write, as the tree can be only be manipulated by their pointers – as a solution to this I have written my own algorithm to handle this.

As well as this, I have created a static class of traversals within which lexically closed functions for the node processing and the base case can be passed to one of the three traversal methods (pre, in, or post order traversal). The use of this class makes writing the functions for this class even easier and faster, as most of the boiler plate code for the recursion is abstracted away, in addition to this, most operations required two methods – handling input data, and then the actual recursion function, the use of this static class allows me to keep the code compact by abstracting away the recursion methods.

The node class and the Binary Tree class have been extended from general purpose to a more specific purpose – card counting. I have achieved this by extending the classes via inheritance to add an extra attribute of “card count” to the node – this means along with the value of the card, a Card Node will also have an attribute which will track how many of the corresponding card is left in the deck. This is the main fundamental change, the extensions to the binary tree to the binary card tree extend the tree around this by providing additional behaviours around the card count attribute. One such example is counting the number of cards in a tree equal to or greater than a given value, which utilises traverses returning the count value rather than the node value.

##### Binary Search Tree and Card Counting Tree Class Diagram

|  |
| --- |
| Binary Search Tree |
| + Root: Node object |
| + Constructor(root : node) : Void  + Get Root() : Node  + Insert(toInsert : Node) : Void  + get\_tree\_size() : Int  + maintainTree() : Void  - swap\_max\_LST(swapRoot: Node) : Void  - swap\_minRST(swapRoot: Node) : Void  + get\_max\_LST() : Node  -get\_min\_RST(): Node  + Delete(node) : Void  - Delete\_oneChild(node, nodeParent, nodeIsLeft): Void  - Delete\_twoChildren(node, nodeParent, nodeIsLeft) : Void  - Delete\_noChildren(node, nodeParent, nodeIsLeft) : Void  - Delete\_root() : Void |

|  |
| --- |
| Card Counting Tree |
| (No additional Attributes) |
| + decrement(nodeValue: Int) : Bool  + cardCountGTET(baseNode: CardNode) : Int  + totalCardCount(baseNode: CardNode) : Int |

|  |
| --- |
| Node |
| + nodeValue : Int  + Left : Node  + Right : Node |
| + hasLeft() : Bool  + hasRight() : Bool  + numOfChildren(): Int  + equality override(other : Node) : Bool  + String Cast Override() : String  + Greater Than / Greater Than Or Equal To Override(other : Node ) : Bool  + Less Than / Less Than Or Equal To Override(other: Node) : Bool |

|  |
| --- |
| Card Node |
| + cardCountValue : Int |
| (No additional Methods) |

#### Algorithms

##### Insertion and Deletion of Nodes within the Binary Tree

In this particular scenario insertion does not occur very often – it will only occur during the initialisation of the tree, when all the cards are to be inserted, this is because cards cannot be pushed to a deck, unless the deck is reinitialised. On the other hand, deletion will occur often, whenever the number of cards the node is associated with reaches 0.

Both of these algorithms are essential to the concept of a binary tree. As the binary tree will be balanced and maintained, the big-O complexity of insertion will be O(log n), because insertion is essentially the same operation as searching for a node, except with an insertion operation at the end. Whilst the explanation and the pseudocode could be considered one and the same for both the insertion and searching, I have implemented these operations in two different ways – the search algorithm (discussed later, function name = getNode) has been implemented as a pre-order traversal, it could also be implemented as a breadth first search; whereas for the insertion operation, I have implemented it utilising a while loop:

###### Insertion Pseudocode

FUNCTION insert(node)  
 IF getNode(node) <> Null THEN # If the node already exists in the tree

RETURN False

ENDIF

IF currentRoot = Null THEN # No root at the moment

Root <- node

RETURN True

ENDIF

nextNode <- Root

lastParentLeft <- True

WHILE nextNode <> Null

IF node.value < nextNode.value THEN

nextNode <- nextNode.left

lastParentLeft <- True

ELSE IF node.value > nextNode.value THEN

nextNode <- nextNode.right

lastParentRight <- False

ELSE

RETURN False # should never reach this point

ENDIF

ENDWHILE

IF lastParentLeft = True THEN

nextNode.parent.left <- node

ELSE

nextNode.parent.right <- node

ENDIF

MaintainTree()

RETURN True  
ENDFUNCTION

The deletion of nodes is a bit more involved in that there are a few edge cases in which deletion can become more or less complex. The different scenarios relate to the number of children the node to be deleted have. Consequently, cost of this operation could vary between constant and one recursion.

If the node to be deleted has no children, then it can simply be deleted from the tree – this is the simplest case. If the node to be deleted has one child, then the node’s parent should be connected to the node’s child, and then all pointers to the node can be removed. If the node to be deleted has two children, then the largest node in the node’s left subtree should replace that node – and then the maximum node in the left subtree should be deleted from its old position – this is guaranteed to recur once, because it can only have one child, as it is the maximum node in the node’s left subtree (it cannot have a right subtree).

The last edge case is when the root of the tree is to be deleted, this has to have a slightly different operation, because the principle of everything on the left of a binary tree must be smaller than the root, and everything in the right subtree must be larger has to be maintained. If the root has no children, then this can be handled the same as normal. If the root has one child, then get either the minimum node in the right subtree, or the maximum node in the left subtree, use this as the new root and delete its old position – based on what subtree exists currently within the tree. Lastly, if the root has two children then the same operation can be used as with one child, however, it should be a consistent node used, for example, if there are two children I will always use the minimum node in the right subtree as the node to use as the new root.

###### Deletion Pseudocode

FUNCTION Delete(node)

NumChildrent <- node.numOfChildren()

IF node = root THEN

Delete\_Root()

ELSE

# executes pre order traversal – similar to getNode method

nodeParent <- getParent(node)

nodeIsLeft <- nodeParent.left = node

IF numChildren = 0 THEN

Delete\_noChildren(node, nodeParent, nodeIsLeft)

ELSE IF numChildren = 1 THEN

Delete\_oneChild(node, nodeParent, nodeIsLeft)

ELSE IF numChildren = 2 THEN

Delete\_twoChildnre(node, nodeParent, nodeIsLeft)

ENDIF

ENDIF  
ENDFUNCTION

FUNCTION Delete\_noChildren(node, nodeParent, nodeIsLeft)

IF nodeIsLeft THEN

nodeParent.left <- node

ELSE  
 nodeParent.right <- node

ENDIF

ENDFUNCTION

FUNCTION Delete\_oneChild(node, nodeParent, nodeIsLeft)

childIsLeft <- node.hasLeft()

swapDestination <- Null

child <- Null

IF nodeIsLeft THEN

swapDestination <- nodeParent.left

ELSE

swapDestination <- nodeParent.right

ENDIF

IF childIsLeft THEN

Child <- node.left

ELSE

Child <- node.right

ENDIF

swapDestination <- child

ENDFUNCTION

FUNCTION delete\_twoChildren(node, nodeParent, nodeIsLeft)  
 max\_LST <- get\_max\_LST(node)

Delete(max\_LST)

max\_LST.right <- node.right

max\_LST.left <- node.left

IF nodeIsLeft THEN

nodeParent.left <- max\_LST

ELSE

nodeParent.right <- max\_LST

ENDIF  
ENDFUNCTION

FUNCTION Delete\_Root()

IF Root has no children THEN

Root <- Null

ELSE

IF Root has left child THEN

swapNode <- get\_max\_LST(root)

ELSE

swapNode <- get\_min\_RST(root)

ENDIF

swapNode.left <- root.left

swapNode.right <- root.right

root <- swapNode

ENDIF

ENDFUNCTION

##### Maintenance of Binary Search Tree

The main benefit of using a binary search tree is the fast access time of O(log n). However, this does benefit is not gained, unless the BST is balanced. An example of an unbalanced binary tree is shown below:

This is the worst case scenario, where the data is inserted into the tree in ascending order. At the worst case, the binary search tree effectively becomes a linked list, and the search time complexity becomes O(n) rather than O(log n), and we lose the benefit of using the BST at all. Although this is the extreme case, there are likely to be other cases where parts of the tree are balanced and other parts are unbalanced, which would result in a search complexity between O(n) and O(log n). Consequently, an algorithm made to maintain the structure of the BST would be useful.

There are a few ways this could be done. One of the ways is to order the data as it comes in and store it in an array. Then the data could be split up into an upper and lower part, where the middle of the upper part of the list and the middle value of the lower part of the list is inserted after the middle value of the data is inserted as the root. This guarantees a balanced binary tree, however, this would require either the full data set to be passed to the tree, as it built, or to have to rebuild the entire tree from the beginning every time a new value is inserted. This also means that the data would have to be tracked in an array as well as the binary tree, which increases the space complexity of the tree. Whilst this could be suitable for the scenario, as we know the full data set before hand, a range of values between 1 and 11, however, using this algorithm would make the BST less extendable for the reasons stated above.

Another way of maintaining the structure of the tree would be to automatically update the structure of the tree as new nodes are added. Since I am storing the tree as Node objects pointing to other node objects, I can simply update the pointers of the nodes to adjust the structure of the tree. The complexity of this algorithm appears, firstly from detecting when the tree is unbalanced, and updating the pointers.

I have designed and implemented an algorithm which does this. From a high level, the algorithm works like this:

1. A Post order traversal is used to count from the bottom up how many children each node has – each recursion of this traversal returns the value returned by each of its children + 1. If a node is None then 0 is returned.
2. Before returning a value, the number of nodes in the current nodes left subtree (returned by the left subtree call) and the number of nodes in the right subtree (returned by the right subtree call) are compared, and if the modulus of the difference between them is greater than or equal to 2, then the structure of this subtree (where the root is the current node) needs adjusting. This is because when there is a difference of two children in either subtree, there is an imbalance which can be rebalanced by changing the root.
3. Swap if the LST has more nodes, swap the root with the max node in the LST. If the RST has more nodes, then swap the root with the minimum value in the RST (more detail in the pseudocode)
4. Return False, to flag that a node has been swapped and the checking should start again

This algorithm would have time complexity of O(n2). Whilst this is a high time complexity (although it is still polynomial), it is worth it in this context, because this only impacts on the program during the population of the tree. As a result of this the searching time complexity becomes O(log n). As the tree will be searched very often during the game, the overall time complexity makes this algorithm worth it.

###### Pseudo Code

FUNCTION Maintain\_TREE()

Completed\_Comparing <- False

While Completed\_Comparing is FALSE

Traversal\_Result <- Maintain\_Traverse(tree\_root)

IF Traversal\_Result <> -1 THEN

Completed\_Comparing <- True

ENDIF

ENDWHILE

ENDFUNCTION

FUNCTION Maintain\_Traverse(current\_node)

IF current\_node is Null THEN

Return 0

ENDIF

Left <- Maintain\_Traverse(current\_node.left)

Right <- Maintain\_Traverse(current\_node.right)

IF (Left = -1) or (Right = -1) THEN

RETURN -1

ELIF abs(left – right) >= 2 //Imbalance in the tree – Therefore Balance

IF Left > Right THEN

Swap Current Node with Max Node in LST

ELSE THEN

Swap Current Node with Min in RST

ENDIF

RETURN -1

ENDIF

RETURN (Left + Right + 1)

ENDFUNCTION

##### Generation of Chances via Card Counting

The premise of the card counting AI is to count the cards which have been played in the game, and then base the next move on the probability of drawing a particular card. The behaviour of the CCAI can then be parameterised, based on the thresholds for action on the probabilities generated.

For the blackjack prototype, I have identified three fundamental probabilities which need to be assessed by the AI when generating its next move, the probability that the next card will bring the AI into the following states: bust, blackjack, winning/exceed winning player. I will also include a Boolean parameter called “winning” which will be true when the current AI is winning, the chance that the next hit will result in a winning state will be 100%, as distinct from the next card being certain to bring the AI from a losing state into a winning state.

The way these probabilities are fundamentally generated are the same: find the critical card value which will bring the AI into the state being checked for, add up all the cards which will lead to this state transition, if the AI receives it, then divide this value by the total number of cards left in the deck.

The way this is implemented is that the Binary Tree used to store the card values and the number of cards associated with that value in the deck will be traversed, and any card which is valid for the state transition will be added to the return value.

For going into a blackjack state, there can only be at most one card which will generate this result. Consequently, when counting the cards for the probability of going into blackjack, only the number of cards for the single value needs to be counted, giving a Big-O complexity of O(log n), as the structure used to count the cards is a binary tree. On the other hand, for exceeding the winning player (given that the AI currently losing) or going bust, there are a number of cards which could result in this state transition – any card which is equal to or greater than a critical value. The operation of counting the nodes for a value in the right subtree of the binary tree would have a big-O complexity much less than O(n), because only the number of cards in the right subtree and that turning node would have to be counted. However, if the turning node is in the left subtree of the binary tree, then the process becomes a bit more complicated as the number of cards in its right subtree have to be counted, as all the cards in the nodes above it in the tree hierarchy, as well as all the nodes in the right subtree of the binary tree.

One issue I discovered later on is that some cards may have been deleted from the card counter tree, as all the cards of that value have been played. As a result, sometimes the card I am looking for may no longer be in play. For blackjack, if this is the case, then transitioning to a blackjack state becomes impossible, however, for the other state transitions, this is not as much of an issue, because, there is more than one card which can lead to this state transition. To amend this, if the turning node is not in the tree, the method will iteration up until it finds the next node up which exists, and then use this as the turning node. If the method iterates up and exceeds the maximum value of the tree, then no cards will lead to this state transition and the function will return 0.

The efficiency of this counting algorithm could be maximised by implementing two different methods for these two scenarios, however, this increases the programming complexity massively, and due to the time constraints, I have decided to use a single umbrella method which checks all the nodes in the subtree. Whilst this does have a big-O complexity of O(n), it is much faster to write, and this time can be guaranteed for all nodes. This method simply executes a post order traversal and counts all the cards values of the nodes which have a card value bigger than or equal to the value of a passed turning node. This method was much simpler to write, as it did not require a node position detection algorithm (detecting if a node is on the left or right side of the subtree) and it only requires a single counting algorithm for all nodes.

The last part of the card counting algorithm is to decrement the nodes in the tree, after each card has been played, and then deleting them when the number of cards left in the deck of that value of card reaches 0. This is achieved by passing in the cards dealt to the CCAI, at the end of a game of blackjack, then decrementing this node, and deleting it if its count reaches 0. This part of the algorithm is quite straight forward, however, there are some edge cases which complicate it a bit. One such case is that every card has been played at the end of a game, in which case the binary tree is reinitialised. The way I have chosen to store the Ace and the Royals in the tree, is to store Jack, King and Queen as 10, and the ace as both 11 and 1; consequently, whenever I decrement the ace I have to decrement both the nodes associated with 1 and 11. Another case is that a deck resets half way through a game, in which case the CCAI has to decrement the binary tree, until it is depleted and then keep the new cards to decrement the new tree after it has been reinitialised. Lastly, when a node has been deleted, the method to maintain the tree to keep a balanced structure is called to keep node searching time at a big-O complexity of O(log n).

From a high level, these algorithms operate in these steps:

Card Counting:

1. Find the current value of the hand of the AI, and then calculate the values of the turning nodes which would result in the transition state of: blackjack, bust, winning. If the turning nodes do not exist, iterate up until you find the next highest turning node, or until you exceed the maximum value of the tree. If the maximum value of the tree is exceeded, return 0.
2. Count the value of the nodes which would result in a state transition – for blackjack, this is just the turning node, for the other states, this would involve counting the cards equal to or greater than the turning node
3. Calculate the probability of the next card resulting in the state transition by counted cards / total number of cards in tree

Decrement and Deletion:

1. Count the number of times a particular value card has been played in the most recent game.
2. Decrement each card value by how many times it has been played in the most recent game. If the value is an ace, decrement both the nodes corresponding to the value 1 and 11.
3. If the count value for a card value has reached 0, after being decremented, delete the corresponding node from the tree.
4. If all the nodes have been deleted from the tree, then reinitialise the tree – the deck will have been reset.

###### Pseudocode

Eg. Calculating bust chance

FUNCTION calcBustChance(hand\_value)

TurningNodeValue <- 22 – hand\_value

IF (TurningNodeValue > MaxCardValue) THEN  
 RETURN 0 # no chance to go bust – card needed is too large

ELSE IF (TurningNodeValue <= 0) THEN

RETURN 1 # Already bust

ENDIF

minExceedValue <- TurningNodeValue

TurningNode <- BinaryCardTree.getNodeByValue(TurningNodeValue, BinaryCardTree.root)

WHILE (TurningNode = Null) and (minExceedVaue <= maximum card value available) THEN minExceedValue <- minExceedValue + 1

TurningNode <- BinaryCardTree.getNodeByValue(minExceedValue, BinaryCardTree.root)

ENDWHILE

IF TurningNode = Null THEN

RETURN 0

ENDIF

numOfBustCards <- BinaryCardTree.cardCountGTET(TurningNode)

totalCards <- BinaryCardTree.totalCardCount()

RETURN numOfBustCards / totalCards

END FUNCTION

**Binary Card Tree**

FUNCTION getNodeByValue(node\_value\_to\_search, current\_node)

IF current\_node is NULL THEN

Pass

ELSE IF current\_node.value = node\_value\_to\_search  
 RETURN current\_node

ENDIF

Left <- getNodeByValue(node\_value\_to\_search, current\_node.left)

Right <- getNodeByValue(node\_value\_to\_search, current\_node.right)

IF Left <> Null THEN  
 Return Left

ELSE IF Right <> Null THEN

Return RIGHT

ENDIF

END FUNCTION

FUNCTION cardCountGTET(turningNode, current\_node)

IF current\_node = NULL THEN  
 RETURN 0

ENDIF

Left <- cardCountGTET(turningNode, current\_node.left)

Right <- carCountGTET(turningNode, current\_node.right)

toAdd <- 0

IF current\_node.value > turningNode.value THEN

toAdd <- current\_node.value

ENDIF

RETURN Left + Right + toAdd

ENDFUNCTION

FUNCTION totalCardCount()

totalValue <- cardCountGTET(self.root)

ace\_node <- getNodeByValue(11)

IF ace\_node <> NULL THEN

totalValue <- totalValue – ace\_node.value

ENDIF

RETURN totalValue

ENDFUNCTION

FUNCTION decrement(card\_value)

Node\_to\_decrement <- getNode(card\_value)

If Node\_to\_decrement = Null THEN

RETURN False

ELSE IF Node\_to\_decrement.value = card\_value THEN

Node\_to\_decrement.value <- Node\_to\_decrement.value – 1

If Node\_to\_decrement.value = 0 THEN

Delete(node\_to\_decrement)

Maintain\_Tree

RETURN True

ENDIF

ENDIF

ENDFUNCTION

**Decrement**

FUNCTION Decrement\_Cards\_From\_Hands(\*args) # \*args will be an arbitrary number of hands

deckUpdated <- False

newCards <- []

FOR hand in args

FOR card in hand

Result <- decrement\_card(card)

IF result = False THEN # this means the card could not be decremented,

because there are no more nodes in the tree

deckUpdated <- True

append(newCards, card)

ENDIF

ENDFOR

ENDFOR

# If deck has updated, reinitialise tree, and decrement the new cards

IF deckUpdated = True THEN

Initialise\_tree()

Decrement\_Cards\_From\_Hands(newCards)

# edge case where the last card dealt is the very last card in the deck

ELSE IF BinaryCardTree.root = Null THEN

Initialise\_tree()

ENDIF

ENDFUNCTION

FUNCTION Decrement\_Card(card)

Result <- Null

IF card.isRoyal() THEN

Result <- Royal\_decrement()

ELSE IF card.isAce() THEN

Result <- Ace\_decrement()

ELSE THEN

Result <- BinaryCardTree.decrement(card.value)

ENDIF

RETURN Result

ENDFUNCTION

FUNCTION Royal\_decrement()

RETURN BinaryCardTree.decrement(10)

ENDFUNCTION

FUNCTION Ace\_decrement()  
 result1 <- BinaryCardTree.decrement(1)

result2 <- BinaryCardTree.decrement(11)

RETURN (result1 OR result2)  
ENDFUNCTION

### Design – Neural Network Based AI

#### Prototype and Progression of Architecture

##### Initial Prototype Design

As decided in the analysis, an automated neural network based AI would be an optimal solution to the requirements, due to its low maintenance cost and versatility. However, the complexity of the neural network based solution comes in the design stage. This is because a neural network is hard to debug, as it is essentially a stochastic process of updating many weights, based on feedback from an environment. As a result, having a well-designed network with a strong fundamental understanding of its architecture will make it easier to make in the long run.

The first prototype of the neural network will have a simple architecture as shown in the analysis. It will have an input layer, a single hidden layer with the same number of nodes and the input layer, and an output layer. It is better to begin simply and prototype the performance and add different features as it needs it.

For the initial prototype for the blackjack prototype, I will simplify it even further by reducing the number of features down to four features: the current value of the AI’s hand, the value of the dealer’s hand and the number of cards in the AI’s and the dealer’s hand. In addition, it will have one hidden layer of the same size as the input layer, and an output layer of two binary outputs: hit and stand.

The network architecture for this prototype is shown below:



In Conjunction with this, I will train this network with the following rewards and the following exploration strategy.

Before that, formally this environment poses as Markov Decision Process to the agent, which means that it provides a set of states, rewards, and trasitions between states which provide rewards. Also, the transitions and rewards are temporal, meaning that the reward may be delayed over time, so the agent needs to strike a balance between short-term and long-term rewards. Below are the formally defined transition function and reward function which are characteristic of a Markov decision process.

s’ = T(s, a)

T is the transition function, where s is the state vector, and a is the action (s ∈ S, where S is all possible sates), (a ∈ where A is all possible actions) and s’ is the new state vector.

An example of this is the agent hitting when the agent has a hand of 5 of diamonds and 5 of hearts, and the dealer has a hand of 6 of spades and 5 of clubs. After hitting the agent gains the 5 of spades. In this scenario

s’ = (15, 2, 11, 2)  
s = (10, 2, 11, 2)  
a = HIT

In addition, the general reward function:

r = R(s, a)

Where r is the reward, R is the reward function, s is the state, and a is the action, where s and a are elements of S and A, same as before.

**Rewards:**

* Hit -> Hand value of agent \* normalisation constant
* Stand -> If the AI is leading after a stand, then Hand value of agent \* normalisation constant
* Bust -> (-hand value of agent – 1) \* normalisation constant
* Absolute Winner -> (hand value of agent + 1) \* normalisation constant

The normalisation of these values assists in the optimisation of the gradient descent algorithms, because all the values for the features will be in a similar range, which means that gradient descent will occur at the same rate for all weights. In this example, I have used 1/30 for the normalisation constant, because no hand will typically exceed a hand value of 30.

These decisions for the rewards is not balanced in the sense that there is often going to be a higher cost for going bust or losing, compared to the reward for winning. This is because the rewards are relative to the hand values, as a result, when the agent goes bust the cost will always be lower than -21, whereas the reward for winning will only be around 21 max. This is somewhat countered by the fact that the agent receives a reward for hitting, and increasing the value of the hand, however this could be optimised. Although this is a good starting point for a prototype.

In addition to this, I will use a discount parameter γ (γ ≤ 1), which is a multiplier which reduces the reward of an agent. The reason for this is to ensure that the sum of the rewards converges to an optimum, rather than diverge. I will leave this fixed at γ = 0.99 so that the rewards are not discounted to too small a value to not have an impact.

Based on the rewards, the weightings of the network will be updated using a loss policy formula:

Loss = -log(π)\*A

Where π is the policy (in this case the weightings associated with a particular action in a particular state of the game), and A is the advantage, which is a value compared to the baseline – in this case the rewards are normalised on a scale between -1 and 1, making the baseline 0. By using this loss formula, the network will update its weights to minimize the loss of it’s future actions, based on its previous actions – hopefully converging at the optimal weighting. The way the weights will be updated using the loss function is by using algorithms called gradient descent and backpropagation.

**Optimisation Algorithms: Gradient Descent and Backpropagation**

There are two main algorithms asociataed with the optimization of weights within a neural network. Grandient descent is the actual optimization algorithm, and backpropagation is the algorithm which uses the output loss to calculate loss for all the other layers in the network, so that gradient descent can update all the nodes in the network.

Intuitively, loss is the square differences between the output value and the target output value. This definition is more suitable for supervised learning, in this context, loss can be thougt of as how far off the agent’s target action’s reward was from the optimal reward.

In this documented design I will document the low-level operations of both algorithms, in the sense of the matrix backed operations which they have been implemented, however, in my implementation I have implemented these algorithms using the Tensorflow library, because it allows me to focus on other requirements of the project such as the generation of statistical analysis to improve the AI, which the client desires. However, these are all documented and expalined, so that a third party can understand the intuition behind the process, but I will not provide pseudocode, because I have not implemented these algorithms myself.

Before I can explain either of these algorithms, I need to establish some notation:

|  |  |
| --- | --- |
| **Symbol** | **Meaning** |
|  | The input layer vector (x(1) is the first input node, for example) |
|  | The output layer vector (y(1) is the first node in the output, for example) |
|  | The number of layers in the network. |
|  | The weight matrix where x signifies the layer, and y siginifies the node a cell is associated with, and is the vector for all the weights in one layer. |
|  | The vector of activations for layer x. = x (input layer) |
|  | The activation function, a function of z. |
|  | The derivative of the activation function, for a particular input z. |
|  | The input vector for the activation function at layer x. Defined by the vector multiplication of (T signifies the transpose of the vector) |
|  | The loss matrix. Where x signifies the layer, and the y signifies the associated node within that layer. is the loss vector for layer x. |
|  | Cost function, a function of . These map the set of weights to the associated cost/loss. |
|  | The matrix of partial derivative of the cost function. |

**The Activation Function: Sigmoid and Rectified Linear Activation**

The activation function was discussed in the analysis, however, I will reiterate its purpose and application context here, as well as fundamentals which I did not introduce before. I

n essence, the activation function takes a in a vector , which is the vector multiplication , and maps it to a normalised range, so that the inputs to each consecutive layer remains within a consistent range, in this case, between 0 and 1. In addition to this, there is often an additional bias unit, which is a constant term subtracted from Z, which makes it so that only larger activations map to a large output.

The output of the activation function is used as the activation for the next layer, ie:

The most common function is the sigmoid function, which is useful because it asymptotes at high and low inputs to 1 and 0, and in between it has a constant gradient, which means it has a consistent mapping of values.

However, I have used the rectified linear activation function for this project (ReLU), because the sigmoid function has a common property of vanishing gradients, meaning that at the asymptotes the gradient becomes almost 0, this is an issue, because these gradients are multiplied together and used in gradient descent to converge to an optimum. However, with sigmoid, over lots of layers, small gradients multiplied with other small gradients means that gradient descent will be more likely to converge at a local minimum, rather than the global optimum (this will make more sense in after gradient descent has been explained). The ReLU function does not have the property of vanishing gradients, which makes the system more extendable. However, both functions are acceptable for this stage in the prototyping process, I have chosen the ReLU activation function, just to increase the extendability of the system.



**Backpropagation – Generating Loss For all Nodes in the network**

As explained before, gradient descent makes use of loss to change the weights so that the system more often decides to take the more rewarding action. The issue at hand is that we only have access to the input layer X and the output layer Y, the rest of the nodes are hidden (hence the name “hidden layer”), however, we want to optimize all the nodes in the system, so to do that we have to generate loss for all the weights in the system. The solution to this problem is an algorithm called Backpropagation.

The first step of backpropation is to perform forward propation, this essentially means multiply each activation by it’s corresponding weight vector and feed it to the next layer, continuing the process until the final value is ouput:

… continue for all layers…

From this first step we have output values – these will be calculated beforehand in sampling the agent’s actions, so this step is not strictly part of this implementation, but it’s needed.

The next step is to generate δ for each layer. is generated using the loss policy equation stated before (and later on using the target network).

After this, the loss for the previous layers are generated by propagating back the loss from the output layers:

Then continue propagating back the loss for all the other layers, replacing L with (L-1) and (L-1) with (L-2) and so on, until you have a full matrix of δ for all the weights in the system. For reference later, after regularization

**Gradient Descent – Optimizing the Weights to Minimize Loss**

Gradient descent is where the magic happens – it takes the gradient of the cost function, and then updates the weights, which after a number of iterations, should converge to a optimum. Fudamentally, the cost function is the squared difference between a desired output and the actual output. Conseuquently, the graph of the cost function (ideally) should look like this:

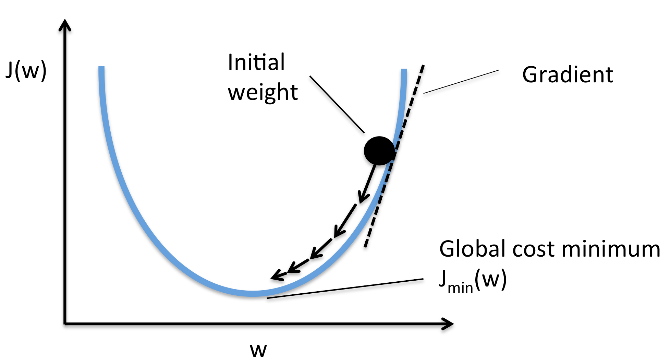


Ignoring the other part of the graph for now, fundamentally you can see that it is a curve with one optima. This is due to the nature of the cost function being a square difference, without the squaring the curve would be linear.

The idea behind gradient descent is that you take the gradient the cost function, of wherever you are on that graph with the current input (the current weighting of the network). Then you subtract the gradient of where you are on the graph from the current weighting (multiplied by learning rate explained later), and this will bring you closer to the optimum.

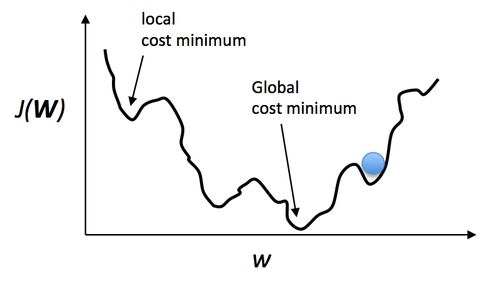
This works because, take for example where the input is on the graph above, the derivative at that point is at that point is a negative, therefore subtractive the gradient at that point will increase the input, moving it to the right of the graph. Now consider the opposite case – on the other side of the miniumum, the graph has a positive gradient, therefore subtracting the gradient at this point will decrease the input, bringing it close to the mimum. The beauty of this is that the closer you get to the mimum, the change is going to be smaller, because at the minimum the gradient is 0, but the further you are from the minimum, the larger the gradient, and so the change will be much larger.

Formally the formula for each iteration of gradient descent is:

 α is the learning rate. This is a parameter which determines how fast gradient descent converges. A low learning rate will mean that the algorithm will converge slowly, but surely. On the other hand, a large learning rate means that you the algorithm will converge more quickly, but if you pick too high a leraning rate, you will overshoot the minimum and diverge from the optimum, rather than converge:



Fundamentally that is gradient descent, however, one large issue with this algorithm to consider is the actual graph of the cost function, as the above graph is the ideal cost function, however, it does not always look like that with one nice global minimum. Often it could look more like this:



As you can see here, there many points of inflexion on this graph, which is a huge issue, because gradient descent will conver to any minimum, or even maximum points – wherever the gradient of the curve is at a minimum. We are also at the disadvantage that the I have no idea what the cost function for the Blackjack environment is, so I cannot easily pick an initial weighting which will converge to the global optimum, and it would be very computationally expensive to compute the cost graph.

To solve this problem, I will train the network with many different random initial weightings. By doing this, I may not guarantee that the network converges to a global optimum, however, it will give a good estimate of an optimum which may be good enough. Another way I will amend this is by using momentum based stochastic gradient descent (explained further down) via the Adam algorithm.

Finally, I will give a quick run down of the variations of gradient descent, so I can then explain why I chose the variant I chose. Firstly, there variations revolve around how often you update the weightings. Above is an example of batch gradient descent, where you calculate the costs/loss of all the weightings all at once, and then update them all at the same time. With very many samples, batch gradient descent will converge straight to the optimum every iteration, however, it will take a long time per iteration. This is because calculating the partial derivative of the cost function involves summing the costs of all the data samples.

Some of the other options include stochastic and mini-batch gradient descent. Stochastic gradient descent involves using one data example from the training set to update the weights. Mini-batch gradient descent strikes a line in the middle, where it takes b training examples, and updates the weights using b samples.

Considering the scenario with many samples, on one end of the spectrum, batch gradient descent takes a long time per iteration, but will always converge straight for the minimum. Whereas stochastic gradient descent only considers one sample at a time, but so each step will be incredibly fast, but will take more iterations to converge to the optimum, because anomalies are not smoothed out with only one sample. Lastly, mini-batch gradient descent takes a strike in the middle. Below is a diagram illustrating these variations, where a 2d graph is represented using contour lines, like on a map, imagine these lines representing a bowl like shape, with the minimum in the middle. This diagram illustrates the path each algorithm takes per iteration.



Overall, batch gradient descent is best for small data samples, and is unreasonable for larger data samples. Not only is it expensive to sum over a large sample, but also every training example has to then be loaded in to the main memory stream at once, which is not space efficient either and could lead to thrashing. Beyond that, at large samples of data, the choice between Mini-batch and stochastic gradient descent does not make as large of a difference.

Within this project, since I have built the blackjack environment locally, and a game of blackjack does not take long to play, it makes it quite quick to be able to generate a sample game, so I can potentially generate a large dataset, making stochastic or minibatch gradient descent optimal.

Specifically, within this implementation have used the Adam optimization algorithm, which is a variant of stochastic gradient descent. In regular gradient descent (bath, mini-batch or stochastic) the learning rate remains fixed; however, Adam makes use of adaptive learning rates, which means that it takes the average first and seconds moments (momentum of change in the weights) and uses this to adapt the learning rates. Intuitively, think of this as if the weights are updated a lot of times in one direction, it will “accelerate” and keep updating in that direction, regardless of smaller changes in direction on the cost curve – this is analogous to how a ball rolling down a hill will gain in momentum and will not be stopped by smaller bumps on the hill. Consequently, the descent will not jitter around as much as in the diagram above, and it amends some of the problem of convergine at local optima, by momentum carrying the descent past small local minima.

**Exploration Strategy**

As an initial exploration strategy, I have used the e-greedy policy. During the training of the neural network, there are different behaviours the AI can take when exploring the problem space. In the parameterisation of the neural network, there is a designated number of games for which is dedicated for exploring, after which the neural network will behave in a more predictable manner and act on what the weighting shows is the optimal move, for the highest reward.

The e-greedy policy has a parameter epsilon, which is a probability that the agent will take a random action. Whilst the agent is still exploring the environment, a random action is always taken, so that the agent can explore the environment and use an optimisation algorithm to change the weightings based on the features and rewards. After this exploration phase of the training, the value for epsilon will start high, and with each action the agent takes, epsilon will be decremented to a final parameterised value – the behaviour of the agent will change based on what final value of epsilon is passed.

The final value of epsilon is above 0, so that the agent will not always act in a greedy way, as there is no guarantee the problem space has been explored fully during the exploration phase. This strategy is good because it combines the random and greedy exploration strategies, and it is a fast implementation for a prototyping phase.

**Episode Buffer**

##### Prototype Results:



This initial prototype has a high winrate of 45.6%, which is higher than expected[[17]](#footnote-17) which means on the surface this initial prototype performs well, in a pure sense. However, keep in mind that this prototype only tests against one other player – the dealer – who behaves in a predictable way, which could be exploited by a more sophisticated agent, this winrate may drop if the agent were 1v1 against a different opponent, or if used in group play.

The big issue here is that the average value for the agent to stand is 14.6, which is lower than desired, considering the dealer will only stand at a value above 17. Since the agent still has a relatively high winrate, perhaps 17 is not the optimal value to stand on, however, the agent’s winrate is lower than 50% which suggests to me that 14 or 15 may not be the optimal stand value either. However, this could also be down to the rewards being skewed towards a higher loss and bust cost, than the typical win reward.

All this and more neural network components will be addressed in the second prototype.

##### Prototype 2 Architecture and Design

One aspect which can be improved in this second prototype is the architecture. Firstly, there were arguably only two important features in the first prototype: the hand values for each of the players. The length of the hand for each player is not relevant, and does not have much relation to the win condition for any of the players. There is some correlation between the length of the hand and the likelihood of going bust from hitting next, however, there is a more direct relationship with this same result and the feature of the value of the hand for each player. In addition to this, the architecture will only work for a 1v1 scenario.

Fortunately, for a game like Blackjack, in a 1vMany game scenario, the hand value of the current winning player is the only relevant hand, because there in order to beat everyone you only have to beat the winning player, therefore it can be simplified down to a 1v1 game almost.

Moreover, the hand value of the AI is not very much to go off information to go off when deciding if it is a good move to hit or not. To increase the amount of information the neural network agent has, I have included the statistics calculated by the card counter as features for the neural network, this brings more nuance to the table for the agent when making decisions. Another secondary benefit is that the probabilities calculated by the card counter are between 0 and 1, so they are already normalised making it easy to include these values as features.

This changes the network architecture to look like this:



###### Recurrent Cells

One of the drawbacks of a normal neural network is that it cannot make temporal based decisions. For example, say the neural network knew that an opponent is nearly more likely to stand on the 3rd turn then, the agent may want to take this into account when deciding whether it is better to stand or hit. In essence, the use of a recurrent layer allows the agent to make better decisions within a given game, rather than each move being considered in isolation. Although, the impact of a recurrent neural network may not be as large for a game like blackjack, where each move could be considered in isolation, in theory, and the effective reward would be similar – for example, it does not matter if it is the first turn or the third, it would still be a bad move to hit if you have a hand value of 21. However, in other games, for example poker where the previous bets of each person, and the progress of each game matters more, the recurrent neural network design is more effective. Therefore, by implementing this now, it makes the system more extendable.

The recurrent node operates by feeding the output of previous calculations back into the hidden layers of the neural network, essentially providing more than one input layer. An alternative option to this would have been to stack the sampling frames and then fed all the frames into the network at once during training, however, whilst that is applicable for the simple nature of blackjack, using a recurrent cell makes it easier to extend the neural network for other games.



###### Experience Sampling

For a similar purpose to the recurrent cell, rather than feeding every game to the network after a enough games has filled a batch size in a linear manner, by storing the experiences and sampling experiences from them randomly, the network can learn more robustly. Each experience will be stored in a simple array structure of [state, action, reward, new state]. The random sampling of experience allows the network to learn from a wider variety of experiences, preventing overfitting and allowing the network to learn and generalise faster, rather than just learning from the immediate past.

I will implement a simple class which will handle the experience sampling:

###### Dropout Layers

One issue which may occur with any sort of learning algorithm is overfitting – this is where a learning algorithm adjusts to weights to be able to perform well on a given dataset, but then lose its ability to perform in the same way generally, because the network’s weights have been adapted to fit the nuances of the given dataset too much. An example of this is when a neural network is being trained in image classification, and it has a 95% accuracy rate for the training set, however, for a test set it may have only 60% accuracy.

Dropout layers are layers within a neural network, in which the layer has a chance to be deactivated, and then the sum is scaled up and carried through the network, so that there is a convergent sum at the end. These improve generalisation, hence reduce the risk of overfitting because the dependency between neurons, in a given layer, is reduced with dropout which, in turn, increases the potency of each individual node. One of the drawbacks of dropout is that it increases the number of iterations required for convergence, but reduces the training time.

For this project, this is not such a large issue for the initial training of the neural network, because the training dataset can be increased to a large amount with a small time cost, because the game is built in and it is quite quick to complete a game. Since so many games will be played, and they will be randomly sampled for training, the risk of overfitting is not incredibly high, as any anomalies can get smoothed out with more data. However, one aspect of the objectives of this project is the ability for the AI to adapt with the behaviours of different user’s. Whilst this could include the feature of classification of users based on their playstyle, and then changing the playstyle of AI to what is effective against these playstyles, one other option could be to use the dataset of a player and then train the AI against this dataset. In this scenario the given dataset will be limited, although there is also a possibility of grouping similar users together in the same dataset) and it will be much smaller than the dataset which can be generated using the other agents and the dealer, consequently, the chance of overfitting is much higher. Therefore, dropout layers will be an important feature.



###### Separate Target Network and Primary Network

In the first prototype, the loss or the cost was generated using a cost policy:

Loss = -log(π)\*A

As explained in the first prototype. In this second prototype, rather than use this cost policy to generate the costs, instead I will use a second target network which will generate the costs for the agent’s actions. The reason for this is that using one network to generate an action and the Q value of an action (how valuable that action is given the current state and the action – essentially the output of the network) often results in overestimated Q-values, where each Q-value is not overestimated equally. Consequently, by separating the concerns of the networks, a primary network to generate the action, and another one to generate the target Q-values, the over estimation is reduced, resulting in a network which will train more reliably and faster.

###### Advantage and Value Streams

Keeping with the theme of serparation of concerns, normally the Q-value of a given state and action, as well as the reward for a given state and action, is given by a function of s and a, which considers both the combined effect of the state and the action. In this new design, the new reward function can be considered as a composite function:

V(s) is a function which returns the value of just being in a given state. For example, in this context, the state which corresponds to the agent of having a hand value of 21, regardless of any of the other features of the state, would return a high value for V.

A(a) is another function which returns the advantage of particular actions, in general, reagardless of the state. For example, hitting may be considered a high value action, in general, because there is a much bigger reward for hitting, compared to standing (not considering the state of whether the agent is winning or on a high hand value), and the agent will want to hit more often to build up to the higher hand values. So, for hitting, the A(a) would return a high value, in theory.

These are then recombined into a single reward function R(), which gives the overall reward. In this context, R() may take discount the value given from A() much more then the value returned from V() because in blackjack the state is much more important than the action – for example, in general having a high hand value is much more important than hitting or standing, in isolation. The principle here is that these values are no longer necessarily bound.

The result of this is that it makes it much easier to consider the value of just being in a state, without having to consider an action.

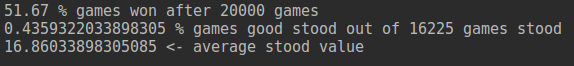
For example, the agent would now have a much higher value associated with having a hand value of just 21, whereas before it may have had mixed evaluations of being at 21, because it had to consider actions with the state and not just the state – 21 has the highest reward if the agent considers 21 with standing, however, it also has the highest chance to give a negative reward if the agent considers 21 with hitting. This scenario is not ideal, because intuitively we know that being at 21 is the best position you can be at, in Blackjack, but the agent before would not have considered it this way. However, not with these separate value streams the agent now be in line with this intuition.

###### Convolutional Layers

One additional neural network feature I considered including was the use of convolutional layers. I will not go into detail about the mechanics of convolutional layers, like I have above, because I have not utilised them. Convolutional layers are used for a neural network to better perceive regions of space, and relationships between objects[[18]](#footnote-18); it does this by changing the way the network perceives images from a series of individual pixels, to a system of shapes. As standard deck games do not require a perception of physics space or image recognition, I felt that these were not a feature worth including, as they would decrease the efficiency of the system, for no real gain.

###### Exploration of Exploration Strategies

##### Prototype 2 Results



Here the improvement of the agent is much better than I had even hoped. As referenced before, the average winrate in Blackjack is about 45% against the dealer, so a 51.67% winrate is fantastic. Here, the average stand value is 16.87, almost 17, which is the value the dealer typically tends to stand at, which makes sense why the agent would have a nearly 50% winrate, as both the players are standing at a similar value. However, it is interesting to note that the average stand value of the agent is a little lower than 17, but its winrate is a little over 50%, this is down to the fact that it has a more nuanced understanding of how likely it is to go bust, due to the card counting features. This signifies that despite the high average stand value, it does not go bust often.

A conclusion which can be made about the neural network based AI is that, in this iteration, it is very similar to the Card Counting AI, in that it utilises the information from the card counter to determine what move it should make, however, it is interesting to note that this system is much more extendable, because the thresholds do not have to be manually determined or modified, as they are trained into the neural network. In addition to this, it is possible to save a different model set for each individual user, in theory, which means that the neural network could have this sort of performance against any player, in theory.

The downside of the agent is that a large dataset is required, in the training, tens of thousands of games were played against the dealer to train it up to this level, which is simply not possible against all users . As a result, being able to generalise users is important, so that many more games can be pooled together to train the system at that classification of user.

#### Algorithms – Training Algorithm

Despite using a high-level library to build the architecture of the neural network and perform all the low-level optimization algorithms, I built a training environment for the network. This training environement was to have the agent playing in lots of different games of blackjack, assigning rewards to each of the actions the agent made at states in the game, then use these rewards to generate a loss. This loss is then backpropagated and the weights of the system is optimized using gradient descent. The usage of the blackjack code has been explained in Design – The Blackjack Game and Environment.

I also included the options for group training, so that the neural network does not overfit playing against the dealer.

##### Pseudocode

# train\_type can be ether “group” or “dealer\_only”

# Agents is a hashmap <String, Agent> where the key is the agent id

# get\_game\_state is a function which returns the game state needed by agents to make a move -

# normally this means the agent’s hand and the next best player’s hand  
FUNCTION train(no\_iterations, explore\_steps, update\_frequency, train\_type)

Blackjack\_env <- Blackjack()

experience\_buffer <- []

epsilon <- 1

FOR i <- 0 to no\_iterations DO

Episode\_buffer <- []

Done <- FALSE

Game\_state <- get\_game\_state()

WHILE NOT Done DO

Current\_player <- Blackjack\_env.get\_current\_player() # returns a hand

IF current\_player.id <> NN.ID THEN

Move <- Agents[Current\_player.id].get\_move()

Process\_action(Blackjack\_env, Move)

New\_game\_state <- get\_game\_state()

Done <- blackjack.game\_over()

ELSE

Exploring <- (i ≤ explore\_steps)

Move <- get\_nn\_move(game\_state, exploring, “e-greedy”, epsilon)

Process\_action(Blackjack\_env, Move)

New\_game\_state <- get\_game\_state()

Reward <- gen\_step\_reward(Blackjack\_env, action)

Done <- blackjack.game\_over()

Episode\_buffer.append(game\_state, action, reward, new\_game\_state, Done)

IF i % update\_frequency = 0 THEN

Agents[NN.ID].update(experience\_buffer)

ENDIF

ENDIF

Game\_state <- new\_game\_state

ENDWHILE

Reward <- gen\_end\_reward(Blackjack\_env)

Episode\_buffer.append(game\_state, action, reward, new\_game\_state, Done)

experience\_buffer.append(Episode\_buffer)

Blackjack\_env.reset()

ENDFOR

ENDFUNCTION

FUNCTION get\_nn\_move(all\_hands, exploring, policy, epsilon)

Poss\_moves <- [HIT, STAND]

IF policy=”e-greedy” THEN

IF random() < epsilon OR exploring THEN

Rand\_ind <- randint(0, Poss\_moves.length)

Action <- Poss\_moves[Rand\_ind]

ELSE

action\_ind <- MAX( Agents[NN.ID].feed\_forward() )

Action <- Poss\_moves[action\_ind]

ENDIF

RETURN Action

ENDIF

ENDFUNCTION

FUNCTION gen\_step\_reward(Blackjack\_env, action)

Reward <- NULL # wanting error to be thrown if none of these conditions are met

Current\_winners <- Blackjack\_env.get\_winners()

IF action = HIT THEN

IF Blackjack\_env.Hands[NN.ID].get\_value > 21 THEN

Reward <- LOSS VALUE

ELSE

Reward <- HIT REWARD

ELSEIF action = STAND THEN

IF NN.ID in Current\_winners THEN

Reward <- STAND REWARD

ELSE

Reward <- LOSS VALUE

ENDIF

RETURN Reward

ENDFUNCTION

FUNCTION gen\_end\_reward(Blackjack\_env)  
 winners <- Blackjack\_env.get\_winners()

Reward <- 0

IF NN.ID in winners THEN

Reward <- WIN VALUE

ELSE

Reward <- LOSS VALUE

RETURN Reward

ENDFUNCTION

FUNCTION Neural\_Network.update(Experience\_buffer)

Training\_samples <- Experience\_buffer.get\_sample()

Loss <- Neural\_Network.backprop(Training\_Samples)

Neural\_Network.Grad\_Desc(Loss)

END

FUNCTION Process\_move(Blackjack\_env, action)

IF action = HIT THEN

Blackjack\_env.hit()

ElSEIF action = STAND THEN

Blackjack\_env.stand()

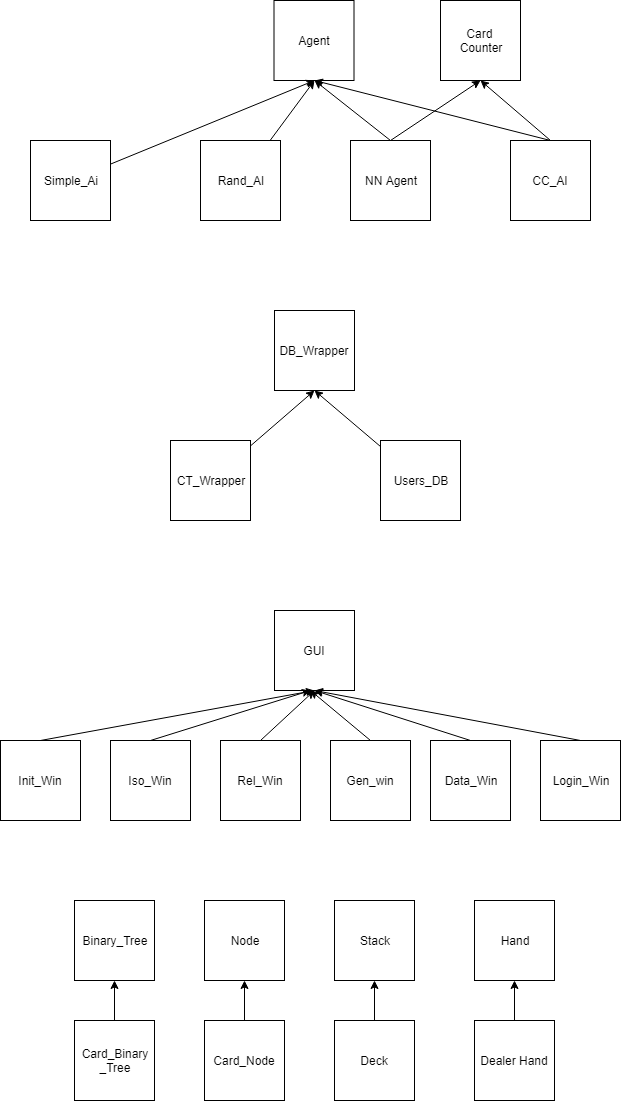
ENDIF

ENDFUNCTION

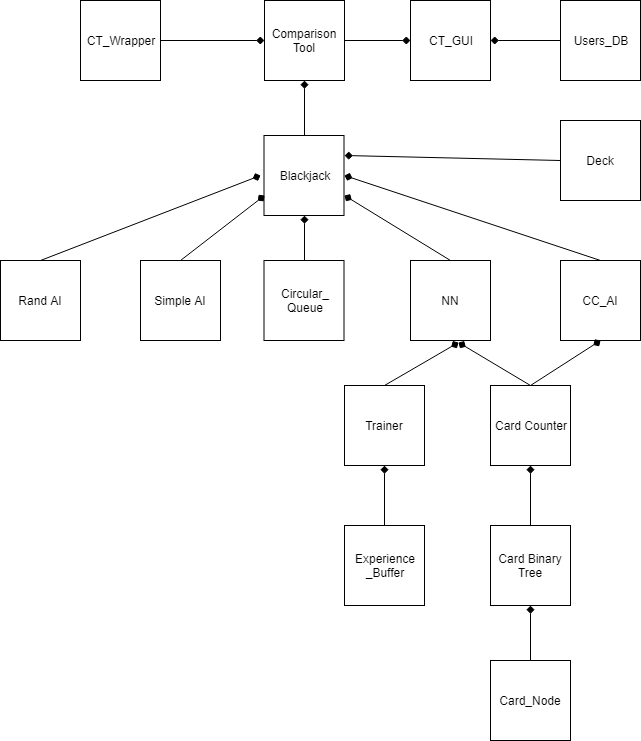
### Design – Comparison Tool

### Final Inheretence and Composition Charts



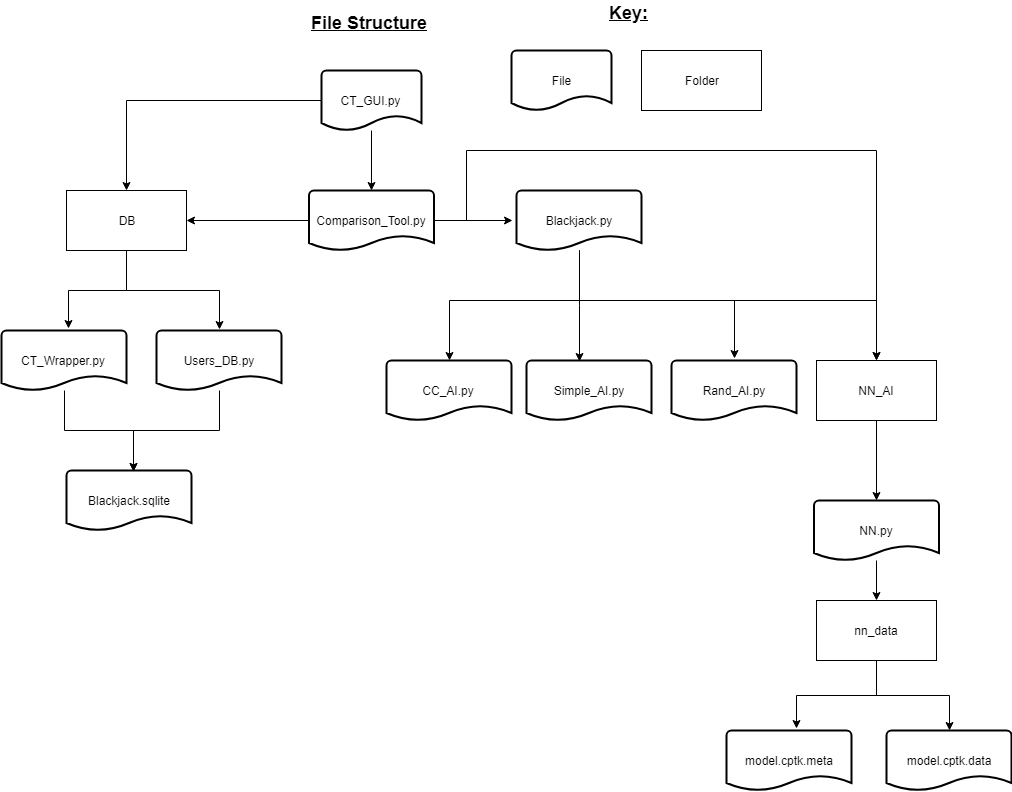




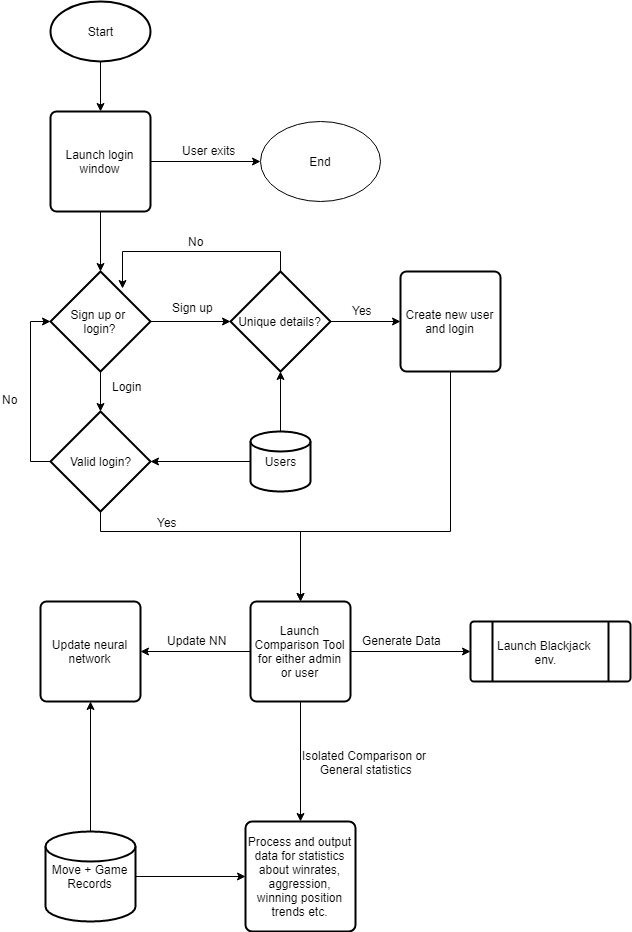


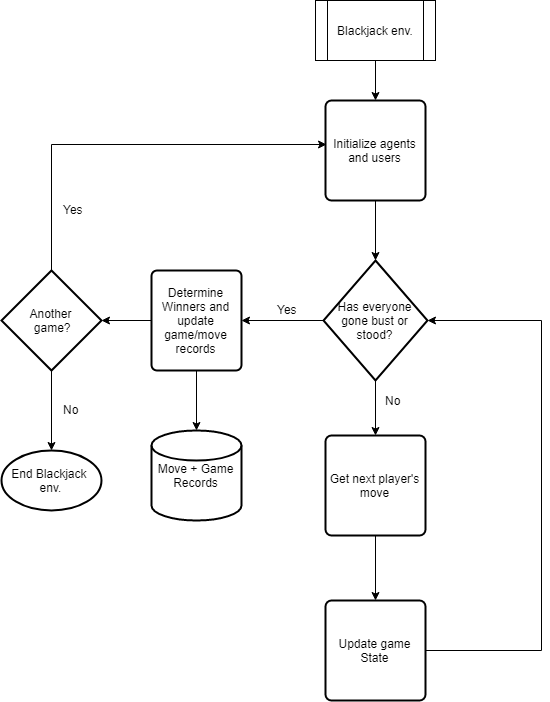
### File Structure

My solution combines multiple different systems, from databases to neural networks, and is intended primarily to be extended and used for another developer in their own program, as a result, an intuitive file structure, with each large system is isolated in their own folder, is imperitive to make the transition of the system as easy as possible.



### System Flowchart





### Interface Design

### Database Design

I have three different purposes for a database in this solution: accounting profiles for different users and agents, recording each move played and the outcome of each game, and the state of the card counter for updating the neural network later.

Some of the queries I have are simply taking the record I have stored in a structure in game, and then pushing them to the database. However, some of the queries are more involved:

For example,

SELECT Card\_Counter\_Record.\*

FROM Card\_Counter\_Record, Moves

WHERE Moves.player\_id='{0}' AND Card\_Counter\_Record.trained=0 AND

Moves.game\_id=Card\_Counter\_Record.game\_id

AND Moves.turn\_num=Card\_Counter\_Record.turn\_num;

SELECT COUNT(Card\_Counter\_Record.\*)

FROM Card\_Counter\_Record, Moves

WHERE Card\_Counter\_Record.trained=0 AND Moves.player\_id='{0}' AND Moves.game\_id=Card\_Counter\_Record.game\_id

AND Moves.turn\_num=Card\_Counter\_Record.turn\_num;

# SoftwareDevelopment

## Main Directory

### Agent.py

1. """
2. - Abstract class implementing and initialise needed properties and methods
3. - every agent should inheret from this and overrdide the behaviours implemented in this class
4. """
5. **from** abc **import** ABC, abstractmethod
7. **class** Agent(ABC):
8. # type is an array of strings signifying the type agent
9. # eg NN, Card Counter
10. **def** \_\_init\_\_(self, ID="", type=None):
11. self.ID = ID
12. self.type = type
13. super().\_\_init\_\_()

16. # this method will be called at the end of every game, the agent is playing in
17. # an example use is for CC agents to decrement their card counter trees
18. @abstractmethod
19. **def** update\_end\_game(self):
20. **pass**
22. # Virtual method which will be called in every game that an agent plays in
23. # this method returns Moves.HIT or Moves.STAND based on what the angent wants to do
24. @abstractmethod
25. **def** get\_move(self):
26. **pass**
28. # Virtual method which sets the agents parameters, on a discrete scale of
29. # "passive", "default", and "aggressive", however will also have a number
30. # scale for more customisability
31. **def** set\_parameters(self, setting="default"):
32. **pass**
34. # mehod which allows the agent to find the next best player in the game
35. # all agents do this in the same way, so it is implemented here
36. # not necessarily going to be anyone but the current agent
37. # pass in list of hands, returns the best hand value, not including tehe agent
38. **def** get\_best\_hand(self, hands):
39. best\_hand\_value = 0
40. best\_hand\_hand = None
41. # find the next best player
42. **for** player **in** hands:
43. **if** player.id == self.ID **or** player.bust:
44. **continue**
45. player\_value = player.get\_value()
46. **if** player\_value > best\_hand\_value:
47. best\_hand\_value = player\_value
48. best\_hand\_hand = player
49. # should never go to this section, because the dealer always plays last and cannot be bust
50. **if** best\_hand\_hand == None:
51. **print**("NO BEST PLAYER")
52. **for** player **in** hands:
53. **if** player.id == "dealer":
54. **for** card **in** player.hand:
55. **print**(card)
56. **print**(player.get\_value())
57. best\_hand\_hand = self.get\_agent\_hand(hands)
58. **return** best\_hand\_hand
60. # pass in list of hands, returns the hand which the agent corresponds to
61. **def** get\_agent\_hand(self, hands):
62. **for** player **in** hands:
63. **if** player.id == self.ID:
64. **return** player

### Blackjack.py

1. **from** Deck **import** Deck
2. **from** Deck **import** Royals
3. **from** random **import** shuffle
4. **from** Structs.Circular\_Queue **import** Circular\_Queue
6. """
7. Class which implemetents a blackjack environment
8. - usage call hit() or stand() based on what action you want to take
9. - winners stored in \_winners
10. - call check\_game\_over() to check if game is over
11. - then call end\_game() to determine the winners
12. - then reset() to reset the env
13. """
15. **class** Blackjack:
16. # playersDict is a dictionary <String => Hand> where the key is the name
17. # of the player and the mapped value is their hand
18. **def** \_\_init\_\_(self, playersDict=None):
19. self.deck = Deck()
20. self.\_blackjack = 21 # The winning value
21. self.\_winners = [] # List of ids of the winners of the previous game
22. self.\_beat\_dealer = []
23. self.turnNumber = 1
24. self.auto\_reset = False
26. # defensive programming - if nothing is passed, default playersDict
27. **if** playersDict **is** None:
28. playersDict = {
29. "dealer": Dealer\_Hand("dealer")
30. }
32. # if dealer has not been added, add the dealer to playersDict
33. **if** bool(playersDict) **is** False **or** "dealer" **not** **in** playersDict.keys():
34. playersDict["dealer"] = Dealer\_Hand("dealer")
36. self.players = playersDict
38. # queue which holds the player hands, keeps track of whose turn it is
39. # pop to get the next player, and push if they are still in the game
40. # game over when nobody left in the queue
41. self.players\_queue = self.create\_player\_queue()
43. self.continue\_game = True
44. self.deckIteration = self.deck.deckIteration # tracks deck resets
45. self.new\_cards = [] # each time a new card is added to the game this it is appended to here
47. # Deals to each player
48. self.init\_deal()
50. @property
51. **def** winners(self):
52. **return** self.\_winners
54. **def** blackjack(self):
55. **return** self.\_blackjack
57. # returns circular queue storing player hands, determining the order of play
58. # random order each time
59. **def** create\_player\_queue(self):
60. playerList = [self.players[key] **for** key **in** self.players.keys() **if** self.players[key].id != "dealer"]
61. shuffle(playerList)
62. cQ = Circular\_Queue(len(playerList))
63. **for** player **in** playerList:
64. cQ.push(player)
65. **return** cQ
67. # returns the player id whose turn it is
68. **def** get\_current\_player(self):
69. **return** self.players\_queue.peek().id
71. # Reset the hands and the tracking variables
72. **def** reset(self):
73. **for** key **in** self.players.keys():
74. self.players[key].reset()
75. self.continue\_game = True
76. self.init\_deal()
77. self.players\_queue = self.create\_player\_queue()
78. self.new\_cards = []
79. self.turnNumber = 1
81. # deals two cards to every player, called at the start of the game
82. **def** init\_deal(self):
83. **for** key **in** self.players.keys():
84. **for** x **in** range(2):
85. self.deal(self.players[key])
87. # compares the hands of the passed players
88. # returns the winners of teh game
89. **def** compare\_hands(self):
90. best\_hand = [] # array of player ids
91. best\_value = 0
92. **for** key **in** self.players.keys():
93. current\_player = self.players[key]
94. current\_player\_name = current\_player.id
95. current\_player\_value = current\_player.get\_value()
96. **if** current\_player.bust:
97. **continue**
98. **elif** best\_hand **is** []:
99. best\_value = current\_player\_value
100. best\_hand.append(current\_player\_name)
101. **elif** current\_player\_value >= best\_value:
102. # if equal to, going to append anyways, so do not need to do anything if equal to.
103. **if** current\_player\_value == best\_value:
104. **pass**
105. **elif** current\_player\_value > best\_value:
106. best\_hand[:] = []
107. best\_value = current\_player\_value
108. best\_hand.append(current\_player\_name)
109. **return** best\_hand
111. # Deals a card to players passed
112. **def** deal(self, \*args):
113. **for** player **in** args:
114. next\_card = self.deck.pop()
115. player.hit(next\_card)
116. # updates the deck iteration, if the deck has reset
117. **if** self.deckIteration != self.deck.deckIteration:
118. self.deckIteration = self.deck.deckIteration
120. # pops the next player and then hits their hand
121. # if they go bust do not push them back onto the queue
122. **def** hit(self):
123. current\_player = self.players\_queue.pop()
124. self.deal(current\_player)
125. self.turnNumber += 1
126. **if** current\_player.bust:
127. self.check\_game\_over()
128. **else**:
129. self.players\_queue.push(current\_player)

132. # pop the next player and stand
133. # do not push them back onto the queue
134. **def** stand(self):
135. current\_player = self.players\_queue.pop()
136. current\_player.stand()
137. self.turnNumber += 1
138. self.check\_game\_over()
140. # Outputs the current state of the game to console
141. # - each hand contents followed by their current value.
142. **def** display\_game(self):
143. **for** key **in** self.players.keys():
144. currentPlayer = self.players[key]
145. **print**(currentPlayer.id, end = " ")
146. **for** card **in** currentPlayer.hand:
147. **print**(card, end = " ")
148. p\_total = currentPlayer.get\_value()
149. **print**(str(p\_total))
151. # Calls all the methods associated with ending the game, and return winner
152. **def** end\_game(self):
153. # dealer deals to itself until it reaches above its threshold hand value
154. self.players["dealer"].dealer\_end(self.deck)
155. self.\_winners = self.compare\_hands()
156. self.update\_new\_cards()
157. **if** self.auto\_reset:
158. self.reset()
159. **return** self.\_winners
161. # appends all cards dealt that game to the new\_cards array
162. # alternative: append each card as it is being dealt
163. **def** update\_new\_cards(self):
164. **for** player\_id, player\_hand **in** self.players.items():
165. **for** card **in** player\_hand.hand:
166. self.new\_cards.append(card)
168. # check if everyone has bust or stood
169. # game is over if the players\_queue is empty
170. **def** check\_game\_over(self):
171. **if** self.players\_queue.isEmpty():
172. self.continue\_game = False
173. #self.end\_game() # end game manually
174. **return** True
175. **return** False
177. # converts player queue to array and returns the Hands of all players currently in play
178. **def** get\_all\_hands\_playing(self):
179. players = []
180. **while** **not** self.players\_queue.isEmpty():
181. #print("gapp", self.players\_queue.peek())
182. current\_player = self.players\_queue.pop()
183. players.append(current\_player)
184. **for** player **in** players:
185. self.players\_queue.push(player)
186. **return** players
188. # returns array of ints
189. # each element is the hand value of a player
190. **def** get\_all\_hand\_values(self):
191. hand\_values = []
192. **for** key **in** self.players.keys():
193. hand\_val = self.players[key].get\_value()
194. hand\_values.append(hand\_val)
195. **return** hand\_values
197. # returns all hands in self.players dictionary as an array, in no particular order
198. **def** get\_all\_hands(self):
199. toReturn = []
200. **for** key, hand **in** self.players.items():
201. toReturn.append(hand)
202. **return** toReturn
204. """
205. - class to handle functionality of each hand
206. - has player id associated with it
207. - handles hit and stand functionality
208. - automatically picks best ace value for the player
209. """
210. **class** Hand:
211. **def** \_\_init\_\_(self, id):
212. self.\_id = id
213. self.\_hand = []
214. self.\_\_has\_stood = False
215. self.\_bust = False
216. self.blackjack = 21
217. self.Royals = {  # Defines the values for the royals
218. Royals.JACK: 10,
219. Royals.QUEEN: 10,
220. Royals.KING: 10,
221. Royals.ACE: 11
222. }
224. @property
225. **def** id(self):
226. **return** self.\_id
228. @property
229. **def** hand(self):
230. **return** self.\_hand
232. @property
233. **def** has\_stood(self):
234. **return** self.\_\_has\_stood
236. @property
237. **def** bust(self):
238. **return** self.\_bust
240. # Only blackjack parent class will have access to the deck, will pass it to cards
241. **def** hit(self, card):
242. self.\_hand.append(card)
243. hand\_value = self.get\_value()
244. **if** hand\_value > self.blackjack:
245. self.\_bust = True
247. **def** stand(self):
248. self.\_\_has\_stood = True
250. # Calculate the best total value of the passed hand (where hand is an array of cards)
251. **def** get\_value(self):
252. total = 0
253. noAces = 0
254. **for** card **in** self.\_hand:
255. cValue = card.value
256. **if** isinstance(cValue, Royals):
257. cValue = self.Royals[cValue]
258. **if** card.value == Royals.ACE:
259. noAces += 1
260. total += cValue
261. **if** noAces > 0:
262. total = self.\_\_choose\_ace(total, noAces)
263. **return** total
265. # If bust, changes the ace from 11 to a 1.
266. **def** \_\_choose\_ace(self, total, noAces):
267. **for** \_ **in** range(noAces):
268. **if** total > self.blackjack:
269. total -= 10
270. **return** total
272. # resets teh hand and tracking variables
273. **def** reset(self):
274. self.\_hand = []
275. self.\_\_has\_stood = False
276. self.\_bust = False
278. # returns true if this hand is out of the game
279. **def** bust\_or\_stood(self):
280. **return** self.bust **or** self.\_\_has\_stood
282. **def** get\_hand\_size(self):
283. **return** len(self.\_hand)
285. **def** \_\_str\_\_(self):
286. **return** self.id
288. # equality comparison to check if two hands are equal
289. **def** \_\_eq\_\_(self, other):
290. **if** **not** isinstance(other, Hand):
291. **return** False
292. **return** self.id == other.id

295. # child class which is to be used for the dealer only
296. # only new method is the dealer\_end() method which
297. # takes in the deck and deals to self until it gets above 17
298. **class** Dealer\_Hand(Hand):
299. **def** \_\_init\_\_(self, id="dealer"):
300. super().\_\_init\_\_(id)
301. self.dealer\_threshold = 17
303. # should you not do this in the blackjack class?
304. **def** dealer\_end(self, deck):
305. **while** self.get\_value() < self.dealer\_threshold **and** **not** self.bust:
306. self.hit(deck.pop())
308. """
309. - test class which allows for manual testing of the blackjack environment
310. - including possibility to play a game from the console
311. """
312. **class** Blackjack\_Tests:
313. @staticmethod
314. **def** setUp\_Blackjack\_Instance():
315. player1 = Hand("mariusz")
316. player2 = Hand("vince")
317. dealer = Dealer\_Hand("dealer")
319. # testhand = Hand("mariusz")
320. # print(testhand == player1)
322. players = {
323. "player1": player1,
324. "player2": player2,
325. "dealer": dealer
326. }
327. bj = Blackjack(players)
328. **return** bj
330. # allows client to play a game of blackjack via the console
331. @staticmethod
332. **def** manual\_test():
333. bj = Blackjack\_Tests.setUp\_Blackjack\_Instance()
334. **for** x **in** range(2):
335. **while** bj.continue\_game:
336. bj.display\_game()
337. currPlayer = bj.whoseTurnIsIt()
338. c = input("\n" + currPlayer + ": would you like to hit (h) or (s) ")
339. **if** c **is** "h":
340. bj.hit()
341. **elif** c **is** "s":
342. bj.stand()
343. **else**:
344. **print**("please input h or s")
345. bj.display\_game()
346. bj.end\_game()
347. **print**(bj.winners, "are the winners!")
348. bj.reset()
350. @staticmethod
351. **def** get\_players\_playing\_test():
352. bj = Blackjack\_Tests.setUp\_Blackjack\_Instance()
353. **for** i **in** bj.get\_all\_players\_playing():
354. **print**(i)
356. **if** \_\_name\_\_ == "\_\_main\_\_":
357. **print**(Royals)
358. #Blackjack\_Tests.manual\_test()

### Card\_Counter.py

1. **import** sys, os
2. sys.path.append(os.path.realpath("./Structs"))
3. **from** Card\_Binary\_Tree **import** Card\_Binary\_Tree
4. **from** Card\_Binary\_Tree **import** Card\_Node
5. **from** Blackjack **import** Blackjack
6. **from** Blackjack **import** Hand
7. **from** Blackjack **import** Dealer\_Hand
9. """
10. - class which implementes the card counter
11. - main functionality is managing the card counter tree
12. - and getting the chances for going bust/blackjack etc.
13. """
14. **class** Card\_Counter:
15. **def** \_\_init\_\_(self, range\_of\_values=None, num\_of\_suits=None):
16. **if** range\_of\_values **is** None:
17. self.rangeOfValues = [1,2,3,4,5,6,7,8,9,10,11]
18. **else**:
19. self.rangeOfValues = range\_of\_values.sort()
20. **if** num\_of\_suits **is** None:
21. self.num\_of\_suits = 4
22. **else**:
23. self.num\_of\_suits = num\_of\_suits
25. self.royalVal = 10
26. self.deckIteration = 1 # tracks deck iteration
27. self.CardRecord = Card\_Binary\_Tree() # instance of binary tree
28. self.populate\_tree\_auto\_maintain(self.rangeOfValues, self.num\_of\_suits)
30. # rangeOfValues is sorted
31. self.maxCard = self.rangeOfValues[-1]
32. self.minCard = self.rangeOfValues[0]
34. # populates the binary tree post-auto maintain
35. # used to have another method which would insert them in a way
36. # that the tree was balanced
37. **def** populate\_tree\_auto\_maintain(self, range\_of\_values, num\_of\_suits):
38. **for** value **in** range\_of\_values:
39. # there are 4 different cards which have a value of 10 in blackjack
40. **if** value == self.royalVal:
41. self.CardRecord.insert(Card\_Node(value, num\_of\_suits \* 4))
42. **else**:
43. self.CardRecord.insert( Card\_Node(value, num\_of\_suits) )
45. # reinitialises the tree - to be called whenever the deck is reset
46. **def** init\_tree(self):
47. self.CardRecord.clearTree()
48. self.populate\_tree\_auto\_maintain(self.rangeOfValues, self.num\_of\_suits)
49. self.deckIteration += 1
51. # returns true if the binary tree is empty
52. **def** Tree\_isEmpty(self):
53. **return** self.CardRecord.root == None
55. # pass all the hands which were in the game just played
56. # decrements the tree by all the cards which were in the game
57. # to be called at the end of each game
58. **def** decrement\_cards(self, \*args):
59. # Unpack the game state and decrement records
60. deckUpdated = False
61. newCards = []
62. **for** hand **in** args:
63. **for** card **in** hand:
64. result = self.decrement\_card(card) # returns false if card cannot be found in Binary Tree
65. # if the deck has reset half way through the game, take these new cards and decrement them later
66. # so that the records are all accurate for which card has been played
67. **if** result == False:
68. deckUpdated = True
69. newCards.append(card)
70. # resets and recurisvely decrements the new cards
71. **if** deckUpdated:
72. self.init\_tree()
73. self.decrement\_cards(newCards)
75. # if tree is empty after each card has been processed
76. **elif** self.Tree\_isEmpty():
77. self.init\_tree()
79. # pass in card
80. # returns true if the card was successfully decremented
81. # returns false otherwise
82. **def** decrement\_card(self, card):
83. **if** card.isRoyal():
84. result = self.royal\_decrement()  # update this so result does not have to be on 3 lines
85. **elif** card.isAce():
86. result = self.ace\_decrement()
87. **else**:
88. result = self.CardRecord.decrement(card.value)
89. **return** result
91. # ace counts as both 1 and 11, so both have to be decremented
92. **def** ace\_decrement(self):
93. result1 = self.CardRecord.decrement(1)
94. result2 = self.CardRecord.decrement(11)
95. **return** result1 **and** result2
97. # royals are defined as a value of 10
98. **def** royal\_decrement(self):
99. **return** self.CardRecord.decrement(self.royalVal)
101. # Next few methods define the CCAI behaviour as well as calcualte the chances.
103. # Calculates the probabilities of different critical scenarios. These are used to determine the next move.
104. # todo update "alreadyExceeding" to "win margin" => more useful value
105. **def** calcChances(self, handValue, winning\_value):
106. bustChance = self.calcBustChance(handValue)
107. blackjackChance = self.calcBlJaChance(handValue)
108. alreadyExceeding = self.getExceedingWinningPlayer(handValue, winning\_value)
109. # if the ai tied to this card counter is winning
110. **if** alreadyExceeding:
111. exceedWinningPlayer = 1
112. **else**:
113. exceedWinningPlayer = self.calcExceedWinningPlayer(handValue, winning\_value)
115. chances = {
116. "bust": bustChance,
117. "blackjack": blackjackChance,
118. "exceedWinningPlayer": exceedWinningPlayer,
119. "alreadyExceedingWinningPlayer": alreadyExceeding
120. }
121. **return** chances
123. # maybe convert this to a margin
124. **def** getExceedingWinningPlayer(self, handValue, wnrValue):
125. **return** handValue > wnrValue
127. # Calc chance next hit will result in bust.
128. **def** calcBustChance(self, handValue):
129. bustValue = (22 - handValue)
130. **if** bustValue > self.maxCard: # If cannot go bust
131. **return** 0
132. **elif** bustValue <= 0:
133. **return** 1 # already bust
135. minExceedValue = int(bustValue)
136. turningNode = self.CardRecord.getNode(bustValue) # returns None if node cannot be found
137. # if turning node is not available, will look for the next card up, until it finds one, or not possible
138. **while** turningNode **is** None **and** minExceedValue <= self.maxCard:
139. minExceedValue += 1
140. turningNode = self.CardRecord.getNode(minExceedValue)
141. **if** turningNode **is** None:
142. **return** 0 # cannot go bust, because card needed is higher than highest card in play
144. # Get total number of cards in right subtree of turning node
145. numOfBustCards = self.CardRecord.cardCountGTET(turningNode)
146. totalNumofCards = self.CardRecord.totalCardCount()
147. **return** numOfBustCards / totalNumofCards
149. # Calculate chance next hit will result in blackjack
150. **def** calcBlJaChance(self, handValue):
151. BlJaValue = (21 - handValue)
152. turningNode = self.CardRecord.getNode(BlJaValue) # only 1 card can bring blackjack
153. **if** BlJaValue > self.maxCard **or** turningNode **is** None:
154. **return** 0 # Cannot get blackjack - either card needed is too large, or card needed is not in deck
155. **elif** BlJaValue == 0:
156. **return** 1 #already blackjack'd
157. # get total number of cards which will result in a blackjack
158. numOfBlJaCards = turningNode.countValue
159. totalNumofCards = self.CardRecord.totalCardCount()
160. **return** numOfBlJaCards / totalNumofCards
162. **def** calcExceedWinningPlayer(self, handValue, wnrValue):
163. # Calc Chance to exceed dealer
164. exceedValue = (wnrValue + 1) - handValue  # Value needed to exceed the dealer's current hand/
165. **if** (wnrValue - handValue) <= 0:  # hand already exceeds best player's, or is equal to dealer's
166. **return** 1
167. **elif** exceedValue > self.maxCard **or** wnrValue == 21:
168. # not possible to exceed the best player, either because it will result in bust or the card needed does not exist
169. **return** 0
171. minExceedValue = int(exceedValue)
172. turningNode = self.CardRecord.getNode(exceedValue)
173. # if turning node is not available, will look for the next card up, until it finds one, or not possible
174. **while** turningNode **is** None **and** minExceedValue <= self.maxCard:
175. minExceedValue += 1
176. turningNode = self.CardRecord.getNode(minExceedValue)
177. **if** turningNode **is** None:  # no cards available to provide the value needed
178. **return** 0
180. numOfExceed = self.CardRecord.cardCountGTET(turningNode)
181. totalCards = self.CardRecord.totalCardCount()
182. exceedChance = numOfExceed / totalCards
183. **return** exceedChance
185. **def** displayCardRecord(self):
186. self.CardRecord.in\_order\_traversal(self.CardRecord.root)
188. # Interface between the game and the counting card AI.
189. """
190. - NO LONGER NEEDED
191. - DECIDE TO LEAVE IT IN AS EVIDENCE OF PROTOTYPE OR WHAT TO DO WITH IT
192. """
194. **class** Counting\_Interface:
195. **def** \_\_init\_\_(self, blackjackInstance, countInstance, CCAI\_Hand):
196. self.blackjack = blackjackInstance
197. self.CCAI = countInstance
198. self.CCAI\_Hand = CCAI\_Hand
200. **def** getGameState(self):
201. playerHand = self.CCAI\_Hand.hand
202. dealerHand = self.blackjack.players["dealer"].hand
203. playerValue = self.CCAI\_Hand.get\_value()
204. dealerValue = self.blackjack.players["dealer"].get\_value()
205. **return** (playerHand, playerValue, dealerHand, dealerValue)
207. **def** takeMove(self, chances = None):
208. **if** chances == None:
209. self.CCAI.calcChances(self.getGameState())

212. # Test Functions - might as well just do unit testing???
213. **class** Testing\_Class:
214. @staticmethod
215. **def** leftDecrementTest( CI):
216. **print**(CI.CardRecord.cardCountGTET(CI.CardRecord.root.left.left))
217. **print**(CI.CardRecord.cardCountGTET(CI.CardRecord.root.left.left))
219. @staticmethod
220. **def** rightDecrementTest(CI):
221. **print**(CI.CardRecord.cardCountGTET(CI.CardRecord.root.right.right))
222. CI.CardRecord.decrement(CI.CardRecord.root.right.right)
223. **print**(CI.CardRecord.cardCountGTET(CI.CardRecord.root.right.right))
225. @staticmethod
226. **def** decUntilDeleteTest(CI):
227. **print**(CI.CardRecord.cardCountGTET(CI.CardRecord.root.right.right))
228. **for** x **in** range(4):
229. CI.CardRecord.decrement(CI.CardRecord.root.right.right)
230. **print**(CI.CardRecord.cardCountGTET(CI.CardRecord.root.right.right))
231. CI.CardRecord.in\_order\_traversal(CI.CardRecord.root)
233. @staticmethod
234. **def** blackjackChanceTesting(CI, testIters):
235. CCAI\_Hand = Hand("CC\_AI")
236. players = {
237. "CC\_AI" : CCAI\_Hand,
238. "dealer" : Dealer\_Hand("dealer")
239. }
240. blackjack = Blackjack(players)
241. CCAI\_Interface = Counting\_Interface(blackjack, CI, CCAI\_Hand)
242. # Get the game state then calc chances
243. **for** x **in** range(testIters):
244. **print**()
246. blackjack.display\_game()
247. gameState = CCAI\_Interface.getGameState()
249. hand = gameState[0]
250. handValue = gameState[1]
251. dealer\_hand = gameState[2]
252. dealerValue = gameState[3]
253. AI\_Winning = hand == dealer\_hand
255. CI.decrement\_cards(hand, dealer\_hand)
256. CI.displayCardRecord()
257. chances = CI.calcChances(hand, handValue, dealer\_hand, dealerValue, AI\_Winning)
258. **for** key **in** chances.keys():
259. **print**(key, chances[key])
260. blackjack.reset()
262. **if** \_\_name\_\_ == "\_\_main\_\_":
263. range\_of\_values = [1,2,3,4,5,6,7,8,9,10,11]
264. number\_of\_suits = 4
265. CI = Card\_Counter(range\_of\_values, number\_of\_suits)
267. **print**("6:", CI.CardRecord.root)
268. **print**("3:", CI.CardRecord.root.left)
269. **print**("9:", CI.CardRecord.root.right)
271. Testing\_Class.blackjackChanceTesting(CI, 20)
273. """
274. range\_of\_values = [1,2,3,4,5,6,7,8,9,10,11]
275. number\_of\_suits = 4
276. CI = Card\_Counter(range\_of\_values, number\_of\_suits)
277. test = Testing\_Class()
278. totalNumofCards = CI.CardRecord.totalCardCount()
279. #print(totalNumofCards)
281. #test.decUntilDeleteTest(CI)
282. test.blackjackChanceTesting(CI, 5)
283. """

### CC\_Agent.py

1. """
2. - abstract class which all card counter agents will inheret
3. - provides basic functionality and interface for the card counter class
4. """
5. **from** Card\_Counter **import** Card\_Counter
6. **from** Agent **import** Agent
7. **from** abc **import** abstractmethod

10. **class** CC\_Agent(Agent):
11. **def** \_\_init\_\_(self, ID, extra\_type=None):
12. **if** extra\_type **is** None:
13. extra\_type = []
14. super().\_\_init\_\_(ID=ID, type=["Card Counter"] + extra\_type)
15. self.CC = Card\_Counter() # instance of card counter
17. # pass in the game state and generate the chances to win
18. # state = [AIHAND, Best hand (not including CCAI)]
19. # what if state[1] is None: No people left playing - handle this (defensive programming)
20. # TODO: Implement a win margin chance/feature
21. **def** get\_chances(self, state):
22. # unpack the state
23. AI\_hand = state[0]
24. AI\_hand\_val = AI\_hand.get\_value()
25. best\_hand = state[1]
26. best\_hand\_val = best\_hand.get\_value()
27. chances = self.CC.calcChances(AI\_hand\_val, best\_hand\_val)
28. **return** chances
30. # gets the agent's next move
31. # returns either Moves.HIT or Moves.STAND
32. **def** get\_move(self, all\_players):
33. game\_state = self.get\_state(all\_players)
34. chances = self.get\_chances(game\_state)
35. move\_next = self.getNextAction(chances, game\_state)
36. **return** move\_next
38. # this is the meat of where each CC agent is different
39. # within this method is the ways that the agent utilises the chances to
40. # generate action. Returns either Moves.HIT or Moves.STAND
41. @abstractmethod
42. **def** getNextAction(self, chances, game\_state):
43. **pass**
45. # pass in hands at end of game
46. # decrements the card counter
47. **def** decrement\_CC(self, new\_cards):
48. self.CC.decrement\_cards(new\_cards)
50. # game state used for generating the chances
51. **def** get\_state(self, hands):
52. agent\_hand = self.get\_agent\_hand(hands)
53. best\_player\_hand = self.get\_best\_hand(hands)
54. **return** [agent\_hand, best\_player\_hand]
56. # called once at end of game, pass in each hand contents
57. **def** update\_end\_game(self, new\_cards):
58. self.decrement\_CC(new\_cards)

### CC\_AI.py

1. **from** CC\_Agent **import** CC\_Agent
2. **from** Blackjack **import** Hand
3. **from** Moves **import** Moves
5. """
6. - Threshold based Card Counting AI
7. """
8. **class** CC\_AI(CC\_Agent):
9. **def** \_\_init\_\_(self, parameters=None, hand=None):
10. super().\_\_init\_\_(ID="cc\_ai")
11. self.parameters = parameters
12. self.Hand = hand
13. **if** parameters **is** None:
14. self.set\_parameters(setting="default")
15. **if** hand **is** None:
16. self.Hand = Hand(self.ID)
18. # entire agent bevahiour defined here
19. # will hit if below best player or below bust threshold or below the minimum hand threshold
20. # beyond that, it will hit, in risky situations defined by it's parameters which maps to an aggression rating
21. **def** getNextAction(self, chances, game\_state):
22. # not exceeding the dealer, hit.
23. playerHandValue = game\_state[0].get\_value()
24. bestHandValue = game\_state[1].get\_value()
26. winMargin =  playerHandValue - bestHandValue
27. belowBestPlayer = (winMargin < 0) **and** bestHandValue <= 21
28. belowBustThreshold = chances["bust"] <= self.parameters["bust\_tol"]
29. highBlackjackChance = chances["blackjack"] >= self.parameters["blackjack\_thresh"]
30. belowWinMarginThresh = winMargin < self.parameters["winMarginThresh"]
31. belowMinHandThresh = playerHandValue < self.parameters["minHandThresh"]
33. # BEHAVIOUR: Hit IF:
34. # - losing or below the bust threshold or below the min hand threshold or below min hand threshold/win margin threshold
35. # - or winning, above the bust threshold and below the risky bust threshold
36. **if** (belowBestPlayer **or** belowBustThreshold **or** belowMinHandThresh):
37. **return** Moves.HIT
38. **elif** highBlackjackChance:
39. belowRiskyBustThreshold = chances["bust"] <= self.parameters["bust\_tol"] \* self.parameters["riskTolerance"]
40. **if** belowRiskyBustThreshold:
41. **return** Moves.HIT
42. **elif** belowWinMarginThresh:
43. **return** Moves.HIT
44. **return** Moves.STAND
46. # Sets the  parameters of the CCAI
47. **def** set\_parameters(self, setting="default"):
48. # Change these parameters to change the behaviour of the CCAI
49. # Change these to personality parameters, then calculate these thresholds based on parameters
50. **if** isinstance(setting, dict):
51. self.parameters = setting #todo check to see if all the keys are in here
53. **if** setting == "default":
54. self.parameters = {
55. "bust\_tol" : 0.5,
56. "blackjack\_thresh" : 0.2,
57. "riskTolerance" : 1.3,
58. "winMarginThresh" : 5,
59. "minHandThresh" : 15
60. }
61. **elif** setting == "aggressive":
62. **pass**
63. **elif** setting == "passive":
64. **pass**

### Comparison\_Tool.py

1. **import** Blackjack as BJ
2. **from** Simple\_AI **import** Simple\_AI
3. **from** CC\_AI **import** CC\_AI
4. **from** Rand\_AI **import** Rand\_AI
5. **import** os,sys
6. sys.path.append(os.path.realpath("./NN\_AI"))
7. sys.path.append(os.path.realpath("./NN\_AI/nn\_data"))
8. sys.path.append(os.path.realpath("./DB"))
9. sys.path.append(os.path.realpath("./Structs"))
11. **from** Circular\_Queue **import** Circular\_Queue as cQ
12. **from** NN **import** NN
13. **from** Moves **import** Moves
14. **from** CT\_Wrapper **import** CT\_Wrapper
15. **import** matplotlib.pyplot as plt
16. **import** numpy as np
17. **from** Card\_Counter **import** Card\_Counter
19. """
20. - class which carries out all the functionality of the comparison tool
21. - TODO - MOVE SOME OF THE GRAPHING PARTS TO THE GUI, THIS CLASS IS FAR TOO MEATY
22. """
24. **class** Comparison\_Tool:
25. ID\_NN = "nn"
26. ID\_SIMPLE = "simple"
27. ID\_CC\_AI = "cc\_ai"
28. ID\_RAND\_AI = "rand"
29. param\_types = ["default", "passive", "aggressive"]  # maybe convert this to an aggressive scale?
31. **def** \_\_init\_\_(self):
32. self.blackjack\_val = 21
33. # Dictionary holding all the hands of the agents
34. self.db\_wrapper = CT\_Wrapper("DB/blackjack.sqlite") # change this to the normal one
35. self.agent\_params = {}
36. self.agents = {}
37. self.agents\_hands = {}

40. # instanciates the agents, sets their parameters and hands
41. **def** init\_agents(self):
42. # The instances of all the agents
43. self.agents = {
44. Comparison\_Tool.ID\_NN: NN(Training=False),
45. Comparison\_Tool.ID\_SIMPLE: Simple\_AI(),
46. Comparison\_Tool.ID\_CC\_AI: CC\_AI(),
47. Comparison\_Tool.ID\_RAND\_AI: Rand\_AI()
48. }
50. # set the params of the agents
51. self.set\_params()
52. **for** agent\_id **in** self.agents.keys():
53. # rand does not have parameters - pass
54. self.agents[agent\_id].set\_parameters(self.agent\_params[agent\_id])
56. self.agents\_hands = dict()
57. self.agents\_hands["dealer"] = BJ.Dealer\_Hand()
58. **for** agent\_key **in** self.agents.keys():
59. self.agents\_hands[agent\_key] = BJ.Hand(agent\_key)
61. # sets the hands in the agent class to the same ones in the agent hands dictionary
62. **for** agent\_id, agent **in** self.agents.items():
63. agent.hand = self.agents\_hands[agent\_id]
65. # sets the parametrs, takes in a dictionary as an argument to set the parameters
66. **def** set\_params(self):
67. self.agent\_params = {}
68. **for** ID **in** [Comparison\_Tool.ID\_NN, Comparison\_Tool.ID\_SIMPLE, Comparison\_Tool.ID\_CC\_AI, Comparison\_Tool.ID\_RAND\_AI]:
69. **if** ID **not** **in** self.agent\_params:
70. self.agent\_params[ID] = "default"
71. **return** self.agent\_params
73. # run X games of blackjack, either returns the winrates of each agent or the game\_id of the games played
74. # create and manage a mainloop game of blackjack
75. # pass the id's of the agents who are playing - DEALER IS NOT AUTOMATICALLY INCLUDED
76. # todo make this function not so large - abstract away all the database queue stuff
77. # todo abstract away all the aggression and win rates stuff too
78. **def** get\_data(self, \*args, no\_games=1000, data\_get="win"):
79. # if a list of the players has been passed in
80. **if** isinstance(args[0], list):
81. args = args[0]
83. # Initialise the agent hands and the agents playing
84. self.agents = {}
85. self.agents\_hands = {}
86. self.init\_agents()
88. agent\_hands\_playing = {}
89. agents\_playing = {}
90. **if** data\_get == "win":
91. win\_rates = {}
92. **elif** data\_get == "ids":
93. game\_ids = []
94. **for** id\_agent **in** args:
95. **if** id\_agent **in** self.agents\_hands:
96. agent\_hands\_playing[id\_agent] = self.agents\_hands[id\_agent]
97. agents\_playing[id\_agent] = self.agents[id\_agent]
98. **if** data\_get == "win":
99. win\_rates[id\_agent] = 0
101. cc = Card\_Counter()
102. blackjack = BJ.Blackjack(agent\_hands\_playing) # local instance of blackjack
103. move\_q = cQ(5000)
104. cc\_q = cQ(5000)
105. game\_q = cQ(2500)
106. game\_id = self.db\_wrapper.get\_next\_game\_id()
107. # play the games and get the win rates
108. **for** game\_num **in** range(no\_games):
109. **if** game\_num % 250 == 0:
110. **print**("game\_num:", game\_num)
111. **while** blackjack.continue\_game:
112. turn\_num = blackjack.turnNumber
113. ID\_current\_player = blackjack.get\_current\_player().id
114. all\_hands = blackjack.get\_all\_hands()
115. agent\_current = self.agents[ID\_current\_player]
117. hand\_val\_before = agent\_current.hand.get\_value()
118. next\_best\_hand = self.get\_next\_best\_hand(ID\_current\_player, all\_hands)
119. next\_move = agent\_current.get\_move(all\_hands) # pass in all player's hands
121. **if** next\_move == Moves.HIT:
122. blackjack.hit()
123. **elif** next\_move == Moves.STAND:
124. blackjack.stand()
126. # calculate the required information to the databases and push to the query queues
127. hand\_val\_after = agent\_current.hand.get\_value()
128. move\_info = (ID\_current\_player, game\_id, turn\_num, next\_move,
129. next\_best\_hand, hand\_val\_before, hand\_val\_after)
130. move\_q.push(move\_info)
132. chances = cc.calcChances(handValue=hand\_val\_before, winning\_value=next\_best\_hand)
133. cc\_info = (game\_id, turn\_num, chances["bust"], chances["blackjack"], chances["exceedWinningPlayer"],
134. chances["alreadyExceedingWinningPlayer"], next\_move)
135. cc\_q.push(cc\_info)
137. **if** move\_q.isFull():
138. self.empty\_queue\_push(move\_q, "move")
139. **if** cc\_q.isFull():
140. self.empty\_queue\_push(cc\_q, "cc")
142. # PROCESS END OF GAME
143. # get the winners, increment their wins, update the agents
144. blackjack.end\_game()
145. # push winners to db
146. winners = blackjack.winners
147. winning\_hands = []
148. **for** winner\_id **in** winners:
149. winning\_hands.append(agent\_hands\_playing[winner\_id])
150. **if** data\_get == "win":
151. **if** winner\_id == "dealer":
152. **continue**
153. win\_rates[winner\_id] += 1
154. game\_info = (game\_id, winners, winning\_hands, blackjack.turnNumber, agents\_playing)
155. game\_q.push(game\_info)
157. **if** game\_q.isFull():
158. self.empty\_queue\_push(game\_q, "game")
160. #update agents and card counter then reset and increment game\_id
161. self.update\_agents(agents\_playing, blackjack)
162. cc.decrement\_cards(blackjack.new\_cards)
163. blackjack.reset()
165. **if** data\_get == "ids":
166. game\_ids.append(game\_id)
167. game\_id += 1
169. # stop session when agent action no longer needed
170. **if** Comparison\_Tool.ID\_NN **in** self.agents:
171. self.agents[Comparison\_Tool.ID\_NN].stop\_session()
172. # convert win records to % and return the win rates
173. self.empty\_queue\_push(move\_q, "move")
174. self.empty\_queue\_push(game\_q, "game")
175. self.empty\_queue\_push(cc\_q, "cc")
177. # normalise win\_rates
178. **if** data\_get == "win":
179. **for** key **in** win\_rates.keys():
180. win\_rates[key] /= no\_games
182. **if** data\_get == "win":
183. **return** win\_rates
184. **elif** data\_get == "ids":
185. **return** game\_ids
187. # NOT COMPLETED
188. # returns a dictionary mapping a parameter (string) to an aggression rating (int)
189. **def** map\_params\_aggression(self):
190. ids = {}
191. param\_agg = {} # map between parameter type and aggression
192. **for** param **in** Comparison\_Tool.param\_types:
193. self.agent\_params[Comparison\_Tool.ID\_SIMPLE] = param
194. ids[param] = self.get\_data(Comparison\_Tool.ID\_SIMPLE, no\_games=100, data\_get="ids")
195. **for** key **in** ids.keys():
196. game\_ids = ids[key]
197. total = 0
198. no\_games = len(game\_ids)
199. **for** game\_id **in** game\_ids:
200. aggr\_rating = self.get\_aggressive\_rating\_game(game\_id, player\_ids=Comparison\_Tool.ID\_SIMPLE)[Comparison\_Tool.ID\_SIMPLE]
201. total += aggr\_rating
202. param\_agg[key] = (total/no\_games)
203. **return** param\_agg
205. # method for emptying a db queue and pushing all the queries
206. # pass in the queue and a string showing the type of queue "move" or "game"
207. **def** empty\_queue\_push(self, queue, q\_type):
208. **print**("Emptying q: " + q\_type)
209. **if** q\_type == "move":
210. # push all moves to db
211. **while** **not** queue.isEmpty():
212. move\_info = queue.pop()
213. self.db\_wrapper.push\_move(agent\_id=move\_info[0], game\_id=move\_info[1], turn\_num=move\_info[2],
214. move=move\_info[3], next\_best\_val=move\_info[4],
215. hand\_val\_before=move\_info[5], hand\_val\_after=move\_info[6])
216. **elif** q\_type == "game":
217. **while** **not** queue.isEmpty():
218. game\_info = queue.pop()
219. self.db\_wrapper.push\_game(game\_id=game\_info[0], winners=game\_info[1], winning\_hands=game\_info[2],
220. num\_of\_turns=game\_info[3], agents=game\_info[4])
221. **elif** q\_type == "cc":
222. **while** **not** queue.isEmpty():
223. cc\_info = queue.pop()
224. self.db\_wrapper.push\_cc(game\_id=cc\_info[0], turn\_num=cc\_info[1], bust=cc\_info[2],
225. blackjack=cc\_info[3], exceedWinningPlayer=cc\_info[4],
226. alreadyExceedingWinningPlayer=cc\_info[5], move=cc\_info[6])
228. # pass in agent id
229. # returns the hand value of the next best agent
230. # will return 0 if all other agents are bust
231. **def** get\_next\_best\_hand(self, agent\_id, all\_hands):
232. best\_value = 0
233. **for** hand **in** all\_hands:
234. **if** hand.id == agent\_id:
235. **continue**
236. hand\_val = hand.get\_value()
237. **if** hand\_val > best\_value:
238. best\_value = hand\_val
239. **return** best\_value
241. # pass in the hands of all the agents playing, returns dictionary mapping strings to agent instances
242. **def** get\_agents\_playing(self, agent\_hand\_playing):
243. toReturn = {}
244. **for** key **in** agent\_hand\_playing.keys():
245. **if** key == "dealer":
246. **continue**
247. toReturn[key] = self.agents[key]
248. **return** toReturn
250. **def** update\_agents(self, agents\_playing, blackjack\_instance):
251. new\_cards = blackjack\_instance.new\_cards
252. deck\_iteration = blackjack\_instance.deckIteration
253. # pass new cards to card counters
254. **for** player\_id, player **in** agents\_playing.items():
255. player.update\_end\_game(new\_cards)
257. # pass in agent id, returns values on agent analysis
258. **def** get\_general\_agent\_analysis(self, agent\_id):
259. agent\_data = self.db\_wrapper.get\_agent\_moves(agent\_id)
260. analysis = self.process\_move\_data(agent\_data)
261. analysis["total\_winrate"] = self.db\_wrapper.get\_agent\_win\_rate(agent\_id)
262. **return** analysis
264. # pass in the data from X games and then process the game to show different stats
265. # data should follow this format: turn\_num, next\_best\_val, hand\_val\_before, move, hand\_val\_after
266. # overall winner, winrates of each player,
267. # extend this to compare winrates of each player, based on each parameter setting
268. # 18 march 2018 -> currrently only analyses agents in isolation, not much relational analysis
269. **def** process\_move\_data(self, data):
270. # NOTE THIS IS NOT RELATED TO AGGRESSION RATING
271. aggresive\_threshold = 17 # TODO make this more sophisticated -> maybe make it so that it takes into account win margin?
273. analysis = {
274. "aggressive\_hits" : 0, # hits when winning and larger than 17
275. "total\_stand\_value" : 0, # used to calculate average stand value later on
276. "no\_times\_stood" : 0,
277. "no\_times\_hit" : 0,
278. "sample\_size" : len(data),
279. "no\_times\_bust" : 0,
280. "no\_times\_bust\_after\_aggressive\_hit" : 0,
281. "average\_stand\_value" : 0,
282. }
283. **print**(data[0])
284. # iterate over data and count different cases
285. **for** move **in** data:
286. next\_best = move[1]
287. val\_before = move[2]
288. action = move[3]
289. val\_after = move[4]
290. went\_bust = False
292. # aggressive check
293. # TODO add a win margin!!!
294. **if** action == Moves.STAND:
295. analysis["total\_stand\_value"] += val\_before
296. analysis["no\_times\_stood"] += 1
297. **if** val\_after > self.blackjack\_val: # and move = Moves.HIT ??
298. analysis["no\_times\_bust"] += 1
299. went\_bust = True
300. **if** action == Moves.HIT:
301. analysis["no\_times\_hit"] += 1
302. **if** val\_before > next\_best **and** val\_before >= aggresive\_threshold **and** action == Moves.HIT:
303. analysis["aggressive\_hits"] += 1
304. **if** went\_bust:
305. analysis["no\_times\_bust\_after\_aggressive\_hit"] += 1
306. analysis["average\_stand\_value"] = analysis["total\_stand\_value"] / analysis["no\_times\_stood"]
307. analysis["%\_hit"] = analysis["no\_times\_hit"] / analysis["sample\_size"]
308. analysis["%\_stood"] = analysis["no\_times\_stood"] / analysis["sample\_size"]
309. **return** analysis
311. # outputs graph of player winrate over games played
312. # pass in player id
313. # todo change this to getting data, plot in the gui class??
314. **def** output\_player\_wr(self, id):
315. agent\_id\_as\_text = self.db\_wrapper.convert\_agents\_to\_text([id])
316. # gets all the game record numbers which the user has played in
317. games\_query = """
318. SELECT game\_id
319. FROM Game\_Record
320. WHERE winner\_ids LIKE '%{0}%'
321. ORDER BY game\_id ASC;
322. """.format(id)  # add validation by selecting from users table
324. # get the data from the database
325. games = self.db\_wrapper.execute\_queries(games\_query, get\_result=True)
326. d\_win\_rate = []
327. games\_won = 0
328. win\_rate = 0
329. batch\_count = 0
331. # increases the batch size with number of games, to optimize speed (more queries slow it down a lot)
332. no\_games = len(games)
333. **if** no\_games >= 50000:
334. batch\_size = 1000
335. **elif** no\_games >= 10000:
336. batch\_size = 100
337. **elif** no\_games >= 1000:
338. batch\_size = 10
339. **elif** no\_games >= 100:
340. batch\_size = 3
341. **else**:
342. batch\_size = 1
344. batch\_size = 100
345. **for** record **in** games:
346. batch\_count += 1
347. game\_id = record[0]
348. games\_won += 1
349. **if** batch\_count % batch\_size == 0: # batches of x so that it is not too jagged
350. # count how many games the player has played in up until this game\_id
351. games\_played\_q = """
352. SELECT COUNT(\*)
353. FROM Game\_Record
354. WHERE game\_id <= {0} AND players LIKE '%{1}%'
355. """.format(game\_id, agent\_id\_as\_text)
356. games\_played = self.db\_wrapper.execute\_queries(games\_played\_q, get\_result=True)[0][0]
357. win\_rate = games\_won / games\_played
358. next\_game = [game\_id, win\_rate]
359. d\_win\_rate.append(next\_game)
360. batch\_count = 0
362. x\_vals = [d[0] **for** d **in** d\_win\_rate]
363. y\_vals = [d[1] **for** d **in** d\_win\_rate]
365. avg\_wr = self.db\_wrapper.get\_avg\_wr()
366. avg\_x = list(x\_vals)
367. avg\_y = [avg\_wr **for** d **in** range(len(avg\_x))]
368. self.plot\_2d(x\_vals, y\_vals, x2=avg\_x, y2=avg\_y, label=id+" Winrate", label2="Avg. Winrate",
369. title="Average winrate over time", x\_lbl="no games", y\_lbl="Win rate")
370. plt.show()
372. # returns two zipped objects:
373. # stand\_value => win rate
374. # hit\_value => bust rate
375. **def** get\_zipped\_aggression\_data(self):
376. stand\_vals, win\_rates = self.get\_stand\_vs\_wr\_data()
377. wr\_for\_sv = zip(stand\_vals, win\_rates)
378. hit\_vals, bust\_rates = self.get\_hit\_vs\_br\_data()
379. br\_for\_hv = zip(hit\_vals, bust\_rates)
380. zipped\_rates = [wr\_for\_sv, br\_for\_hv]
381. **return** zipped\_rates
383. # get an average of the chance to go bust if hit and the chance to win if stood at a particular value
384. # this average will be the aggression rating: ie if there is 1 chance to win if stood and a 1 chance to go bust
385. # if hit, then the aggression will be 1 -> stupidly aggressive
386. # hits are always aggressive -> but can have an aggression of 0 ie. hitting with no chance to go bust AND standing would never result in win
387. # TODO ADD A BUTTON IN THE GUI FOR THIS
388. **def** map\_hit\_val\_to\_aggression(self):
389. zipped\_rates = self.get\_zipped\_aggression\_data()
390. # init aggr map
391. aggr\_mapping = {}
392. **for** i **in** range(1, 22):
393. aggr\_mapping[str(i)] = 0
395. # add all the decimal rates to the aggression map
396. **for** z **in** zipped\_rates:
397. **for** pair **in** z:
398. key = str(pair[0])
399. val = pair[1]
400. aggr\_mapping[key] += val
402. # normalise the rates
403. aggr\_mapping = self.normalise\_dict(aggr\_mapping)
404. **return** aggr\_mapping
406. # maps\_stand values to aggression rating
407. **def** map\_stand\_val\_to\_aggression(self):
408. zipped\_rates = self.get\_zipped\_aggression\_data()
409. # init aggr map
410. aggr\_mapping = {}
411. **for** i **in** range(1, 22):
412. aggr\_mapping[str(i)] = 0
413. **for** z **in** zipped\_rates:
414. **for** pair **in** z:
415. key = str(pair[0])
416. val = pair[1]
417. # 100 is really abritary, find a better way of picking this value
418. aggr\_mapping[key] += val
420. **for** key **in** aggr\_mapping.keys():
421. aggr\_mapping[key] /= -2
423. aggr\_mapping = self.normalise\_dict(aggr\_mapping)
424. **for** key **in** aggr\_mapping.keys():
425. aggr\_mapping[key] \*= -1
426. **return** aggr\_mapping
428. # pass in a dictionary and normalise all the values within it
429. **def** normalise\_dict(self, d):
430. d\_vals = d.values()
431. max\_val = max(d\_vals)
432. min\_val = min(d\_vals)
433. norm\_range = (max\_val - min\_val)
434. **for** key **in** d.keys():
435. d[key] = (d[key] - min\_val) / (max\_val - min\_val)
436. **return** d
438. # pass in player id, get back aggresion rating
439. # TODO extend this to include context (such as was losing at time of aggressive hit) and maybe avg / std devs of stand vals
440. # Returns aggression on scale of -1 to 1 where -1 is extremely passive and 1 is extremely aggressive
441. # eg of 1 => hitting when hit has prob of 1 to go bust and prob of 1 to win if stood
442. # eg of -1 => standing when hitting would have 0 chance of going bust and standing has 0 chance of winning
443. # overall aggression rating
444. # RELATIVE SCALE -> not as absolute as shown above, ie there will be a move which is rated 1 and another -1
445. **def** get\_aggression\_rating(self, id):
446. exists = self.db\_wrapper.agent\_exists(id)
447. **if** **not** exists:
448. **return** None
449. aggr\_map\_hit = self.map\_hit\_val\_to\_aggression()
450. aggr\_map\_stand = self.map\_stand\_val\_to\_aggression()
451. get\_all\_moves\_q = """
452. SELECT turn\_num, next\_best\_val, hand\_val\_before, move, hand\_val\_after
453. FROM Moves
454. WHERE player\_id='{0}'
455. """.format(id)
456. all\_moves = self.db\_wrapper.execute\_queries(get\_all\_moves\_q, get\_result=True)
457. no\_moves = len(all\_moves)
458. total\_aggr = 0
459. **for** move **in** all\_moves:
460. move\_rating = self.get\_aggression\_rating\_move(move, hit\_map=aggr\_map\_hit, stand\_map=aggr\_map\_stand)
461. total\_aggr += move\_rating
462. total\_aggr /= no\_moves
463. **return** total\_aggr
465. # returns aggression rating for a move
466. # pass in data -> [turn\_num, next\_best\_val, hand\_val\_before, move, hand\_val\_after]
467. # for a move to be aggressive it must be above a critical threshold
468. # the player must be winning,
469. # depending on the win margin -> hitting even with "high" win margin => aggressive
470. # does not work as stated in the comment -> just checks the mapping
471. **def** get\_aggression\_rating\_move(self, move, hit\_map, stand\_map):
472. turn\_num = move[0]
473. next\_best\_val = move[1]
474. hand\_val\_before = move[2]
475. action = Moves.convert\_to\_move(move[3])
476. hand\_val\_after = move[4]
478. aggr\_rating = None
479. **if** action == Moves.HIT:
480. aggr\_rating = hit\_map[str(hand\_val\_before)]
481. **elif** action == Moves.STAND:
482. aggr\_rating = stand\_map[str(hand\_val\_before)]
483. **return** aggr\_rating
485. # pass in a query which gets single values
486. # returns array of each value, and array of coressponding decimal frequency
487. **def** get\_values\_frequency(self, query):
488. all\_instances = self.db\_wrapper.execute\_queries(query, get\_result=True)
489. frequencies = self.get\_freq(all\_instances)
490. distinct\_values = [int(key) **for** key **in** frequencies.keys()]
491. distinct\_values.sort()
492. no\_instances = len(all\_instances)
494. y\_vals = [(frequencies[str(distinct\_values[i])] / no\_instances) **for** i **in** range(len(distinct\_values))]
495. **return** distinct\_values, y\_vals
497. # outputs the frequencies of wins from standing at a particular win margin
498. # also outputs the win margin stand frequency - NORMAL DISTRIBUTION
499. # TODO ADD A BUTTON FOR THIS
500. **def** output\_win\_margin\_at\_stand\_vs\_winrate(self):
501. get\_win\_margins\_which\_win = """
502. SELECT (Moves.hand\_val\_before - Moves.next\_best\_val)
503. FROM Moves, Game\_Record
504. WHERE Moves.move=0 AND Game\_Record.winner\_ids LIKE '%'||Moves.player\_id||'%'
505. AND Game\_Record.game\_id=Moves.game\_id
506. """
507. x\_vals, y\_vals = self.get\_values\_frequency(get\_win\_margins\_which\_win)
509. get\_win\_margins = """
510. SELECT (hand\_val\_before - next\_best\_val)
511. From Moves
512. WHERE move=0
513. """
514. x2, y2 = self.get\_values\_frequency(get\_win\_margins)
515. self.plot\_2d(x\_vals, y\_vals, x2=x2, y2=y2, label="Win Frequency", label2="Frequency",
516. title="Win Margin Win Dist", x\_lbl="Win Margin", y\_lbl="% Frequency / % Win Frequency")
517. plt.show()
519. # pass in a game\_id and player ids
520. # aggression rating for game is measured with an average of the aggression of moves from each game
521. # returns the aggression rating for that game
522. **def** get\_aggressive\_rating\_game(self, game\_id, player\_ids):
523. **if** isinstance(player\_ids, str):
524. player\_ids = [player\_ids]
525. hit\_map = self.map\_hit\_val\_to\_aggression()
526. stand\_map = self.map\_stand\_val\_to\_aggression()
527. ratings\_for\_game = {}
528. **for** player **in** player\_ids:
529. get\_moves\_q = """
530. SELECT turn\_num, next\_best\_val, hand\_val\_before, move, hand\_val\_after
531. FROM Moves
532. WHERE game\_id={0} AND player\_id='{1}';
533. """.format(game\_id, player)
534. all\_moves = self.db\_wrapper.execute\_queries(get\_moves\_q, get\_result=True)
535. no\_moves = len(all\_moves)
536. total\_aggr = 0
537. **for** move **in** all\_moves:
538. move\_rating = self.get\_aggression\_rating\_move(move, hit\_map=hit\_map, stand\_map=stand\_map)
539. total\_aggr += move\_rating
540. total\_aggr /= no\_moves
541. ratings\_for\_game[player] = total\_aggr
542. **return** ratings\_for\_game
544. # pass in agent id
545. # outputs graph of average stand value against games played
546. **def** output\_avg\_stand\_value(self, id):
547. query = """
548. SELECT hand\_val\_before
549. FROM Moves
550. WHERE player\_id='{0}' AND move=0
551. ORDER BY game\_id ASC;
552. """.format(id)
553. games = self.db\_wrapper.execute\_queries(query, get\_result=True)
554. x\_vals = []
555. y\_vals = []
556. total\_stand\_value = 0
557. batch\_count = 0
558. **for** i **in** range(len(games)):
559. game = games[i]
560. stand\_value = game[0]
561. total\_stand\_value += stand\_value
562. avg\_stand\_value = total\_stand\_value / (i+1)
564. **if** batch\_count % 10 == 0:
565. x\_vals.append(i)
566. y\_vals.append(avg\_stand\_value)
567. batch\_count = 0
569. self.plot\_2d(x\_vals, y\_vals, title=id+"'s Avg Stand Val Over Time", x\_lbl="no games", y\_lbl="avg stand value")
570. plt.show()
572. # outputs the win rate against aggresion values
573. # plots a point for each user / agent
574. **def** output\_aggression\_win\_relation(self):
575. get\_all\_agents\_q = """
576. SELECT agent\_id, games\_won, games\_played
577. FROM agents
578. WHERE games\_played != 0;
579. """
580. get\_all\_users\_q = """
581. SELECT username, games\_won, games\_played
582. FROM users
583. WHERE games\_played != 0;
584. """
585. all\_agents = self.db\_wrapper.execute\_queries(get\_all\_agents\_q, get\_result=True)
586. all\_users = self.db\_wrapper.execute\_queries(get\_all\_users\_q, get\_result=True)
587. all\_players\_data = all\_agents + all\_users
589. # data => [player\_id, games\_won, games\_played]
590. aggression\_ratings = [self.get\_aggression\_rating(data[0]) **for** data **in** all\_players\_data]
591. win\_rates = [(data[1] / data[2]) **for** data **in** all\_players\_data]
593. # sorting the aggression ratings, but keeping the association
594. agg\_wr\_dict = {}
595. **for** i **in** range(len(aggression\_ratings)):
596. rating = aggression\_ratings[i]
597. agg\_wr\_dict[str(rating)] = win\_rates[i]
599. sorted\_ratings = []
600. corresp\_wr = []
601. **for** rating **in** sorted(aggression\_ratings):
602. as\_key = str(rating)
603. sorted\_ratings.append(rating)
604. corresp\_wr.append(agg\_wr\_dict[as\_key])
606. self.plot\_2d(sorted\_ratings, corresp\_wr, title="Aggression ratings vs Winrates",
607. x\_lbl="Aggr. Rating", y\_lbl="Win Rate")
608. plt.show()
610. # outputs a graph displaying hand values when players have stood, and their % frequency
611. **def** output\_stand\_dist(self):
612. get\_all\_stands\_query = """
613. SELECT hand\_val\_before
614. FROM Moves
615. WHERE move=0;
616. """
617. x\_vals, y\_vals = self.get\_values\_frequency(get\_all\_stands\_query)
619. self.plot\_2d(x\_vals, y\_vals, title="Stand Distribution", x\_lbl="Stand Value", y\_lbl="% Frequency")
620. plt.show()
622. # outputs a graph displaying hand values when players have hit, and their % frequency
623. **def** output\_hit\_dist(self):
624. get\_all\_hits\_query = """
625. SELECT hand\_val\_before
626. FROM Moves
627. WHERE move=1;
628. """
629. x\_vals, y\_vals = self.get\_values\_frequency(get\_all\_hits\_query)
630. self.plot\_2d(x\_vals, y\_vals, title="Hit Distribution", x\_lbl="Hit Value", y\_lbl="% Frequency")
631. plt.show()
633. # pass in a 2d list of values
634. # returns a dictionary where key is a value the corresponding value is the frequency of that value from the dataset
635. **def** get\_freq(self, all\_events):
636. frequencies = {}
637. **for** event **in** all\_events:
638. val = event[0]
639. val\_as\_key = str(val)
640. **if** val\_as\_key **not** **in** frequencies:
641. frequencies[val\_as\_key] = 0
642. frequencies[val\_as\_key] += 1
643. **return** frequencies
645. # pass in x and y values and optional labels
646. # outputs new figure and plots
647. # make the arguments for legends and multiple lines better - really messy atm
648. **def** plot\_2d(self, x, y, \*\*kwargs):
649. fig = plt.figure()
650. ax = fig.add\_subplot(111)
652. **if** "title" **in** kwargs:
653. fig.suptitle(kwargs["title"])
654. **if** "x\_lbl" **in** kwargs:
655. plt.xlabel(kwargs["x\_lbl"])
656. **if** "y\_lbl" **in** kwargs:
657. plt.ylabel(kwargs["y\_lbl"])
659. **if** "label" **in** kwargs:
660. ax.plot(x, y, label=kwargs["label"])
661. **else**:
662. ax.plot(x,y)
664. **if** "x2" **in** kwargs **and** "y2" **in** kwargs:
665. **if** "label2" **in** kwargs:
666. ax.plot(kwargs["x2"], kwargs["y2"], label=kwargs["label2"])
667. **else**:
668. ax.plot(kwargs["x2"], kwargs["y2"])
670. **if** "label" **in** kwargs **or** "label2" **in** kwargs:
671. ax.legend()
672. ax.legend()
674. # pass in move.hit or move.stand and get back all the distinct values of each hit/stand
675. **def** get\_distinct\_vals(self, move):
676. **if** isinstance(move, Moves):
677. move = Moves.convert\_to\_bit(move)
678. **elif** **not** (isinstance(move, int) **and** (move == 0 **or** move == 1)):
679. **return** False
680. get\_distinct\_vals = """
681. SELECT DISTINCT hand\_val\_before
682. FROM Moves
683. WHERE move={0};
684. """.format(move)
685. distinct\_vals\_res = self.db\_wrapper.execute\_queries(get\_distinct\_vals, get\_result=True)
686. values = [i[0] **for** i **in** distinct\_vals\_res]  # all distinct hand values
687. values.sort()
688. **return** values
690. # get series of stand values
691. # get games won when standing on said value
692. # get number of games when stood of value
693. # returns the distinct stand values, returns the stand values and the corresponding winrates
694. **def** get\_stand\_vs\_wr\_data(self):
695. stand\_values = self.get\_distinct\_vals(Moves.STAND)
696. total\_games\_stood = [] # total number of games stood with value coressponding to stand\_values
697. games\_won = [] # games won with value coressponding to stand\_values
699. **for** value **in** stand\_values:
700. total\_games\_query = """
701. SELECT COUNT(\*)
702. FROM Moves
703. WHERE move=0 AND hand\_val\_before={0};
704. """.format(value)
705. # cross param sql
706. no\_games\_won\_query = """
707. SELECT COUNT(\*)
708. FROM Moves, Game\_Record
709. WHERE Moves.move=0 AND Game\_Record.winner\_ids LIKE '%'||Moves.player\_id||'%'
710. AND Game\_Record.game\_id=Moves.game\_id AND hand\_val\_before={0};
711. """.format(value)
713. total\_games = self.db\_wrapper.execute\_queries(total\_games\_query, get\_result=True)[0][0]
714. no\_games = self.db\_wrapper.execute\_queries(no\_games\_won\_query, get\_result=True)[0][0]
716. total\_games\_stood.append(total\_games)
717. games\_won.append(no\_games)
719. win\_rates = [games\_won[i]/total\_games\_stood[i] **for** i **in** range(len(stand\_values))]
720. **return** stand\_values, win\_rates
722. # outputs graph of winrates against stand values
723. **def** output\_stand\_vs\_wr(self):
724. stand\_values, y\_vals = self.get\_stand\_vs\_wr\_data()
725. self.plot\_2d(stand\_values, y\_vals, title="Chance to Win Based on Stand Value",
726. x\_lbl="Stand Value", y\_lbl="Win Rate")
728. plt.show()
730. # returns different values that have been hit on, and the decimal rate of going bust when hit on that value
731. **def** get\_hit\_vs\_br\_data(self):
732. hit\_values = self.get\_distinct\_vals(Moves.HIT)
733. total\_times\_hit = []  # total number of games stood with value coressponding to stand\_values
734. games\_bust = []  # games won with value coressponding to stand\_values
735. **for** value **in** hit\_values:
736. total\_times\_query = """
737. SELECT COUNT(\*)
738. FROM Moves
739. WHERE move=1 AND hand\_val\_before={0};
740. """.format(value)
741. # cross param sql
742. no\_games\_bust\_query = """
743. SELECT COUNT(\*)
744. FROM Moves
745. WHERE move=1 AND hand\_val\_before={0} AND hand\_val\_after > 21;
746. """.format(value)
748. total\_games = self.db\_wrapper.execute\_queries(total\_times\_query, get\_result=True)[0][0]
749. no\_games = self.db\_wrapper.execute\_queries(no\_games\_bust\_query, get\_result=True)[0][0]
751. total\_times\_hit.append(total\_games)
752. games\_bust.append(no\_games)
753. bust\_rates = [games\_bust[i] / total\_times\_hit[i] **for** i **in** range(len(hit\_values))]
754. **return** hit\_values, bust\_rates
756. # outputs graph of bust rate against hit value
757. **def** output\_hit\_vs\_br(self):
758. hit\_values, y\_vals = self.get\_hit\_vs\_br\_data()
759. self.plot\_2d(hit\_values, y\_vals, title="Chance to Go Bust Based on Hit Value",
760. x\_lbl="Hit Value", y\_lbl="Bust Rate")
762. plt.show()
764. # updates the neural network with new data from games it has played
765. **def** update\_nn(self):
766. nn = NN()
767. nn.update\_training()
768. nn.stop\_session()
770. # outputs the aggression scaled for different moves
771. # TODO add button for this
772. **def** output\_aggression\_scale(self):
773. hit\_map = self.map\_hit\_val\_to\_aggression()
774. stand\_map = self.map\_stand\_val\_to\_aggression()
776. hit\_map\_keys = hit\_map.keys()
777. x1 = sorted([int(key) **for** key **in** hit\_map\_keys])
778. y1 = []
779. **for** x **in** x1:
780. y1.append(hit\_map[str(x)])
782. stand\_map\_keys = stand\_map.keys()
783. x2 = sorted([int(key) **for** key **in** stand\_map\_keys])
784. y2 = []
785. **for** x **in** x2:
786. y2.append(stand\_map[str(x)])
788. self.plot\_2d(x1, y1, x2=x2, y2=y2, label="Hit", label2="Stand",
789. title="Hit and stand aggression values", x\_lbl="Hand Value", y\_lbl="Aggression Rating")
790. plt.show()
792. # pass in a user id and get a graph of their aggression rating over time
793. # really slow
794. **def** output\_aggression\_over\_time(self, id):
795. agent\_id\_as\_text = self.db\_wrapper.convert\_agents\_to\_text([id])
796. # gets all the game record numbers which the user has played in
797. games\_query = """
798. SELECT game\_id
799. FROM Game\_Record
800. WHERE players LIKE '%{0}%'
801. ORDER BY game\_id ASC;
802. """.format(id)  # add validation by selecting from users table
804. # get the data from the database
805. games = self.db\_wrapper.execute\_queries(games\_query, get\_result=True)
806. batch\_count = 0
807. # increases the batch size with number of games, to make the querying faster
808. no\_games = len(games)
809. **if** no\_games >= 50000:
810. batch\_size = 1000
811. **elif** no\_games >= 10000:
812. batch\_size = 100
813. **elif** no\_games >= 1000:
814. batch\_size = 10
815. **elif** no\_games >= 100:
816. batch\_size = 3
817. **else**:
818. batch\_size = 1
820. d\_aggr = []
821. games\_num = 0
822. total\_aggr = 0
823. **for** record **in** games:
824. batch\_count += 1
825. game\_id = record[0]
826. games\_num += 1
827. **if** games\_num % 100:
828. **print**("(aggression over time) on game num: " + str(games\_num))
830. **if** batch\_count % batch\_size == 0:  # batches of x so that it is not too jagged
831. # count how many games the player has played in up until this game\_id
832. game\_aggr = self.get\_aggressive\_rating\_game(game\_id, id)[id]
833. total\_aggr += game\_aggr
834. d\_aggr.append([games\_num, total\_aggr / games\_num])
835. batch\_count = 0
837. x\_vals = [d[0] **for** d **in** d\_aggr]
838. y\_vals = [d[1] **for** d **in** d\_aggr]
840. self.plot\_2d(x\_vals, y\_vals, label=id + " Aggression",
841. title="Average Aggression over time", x\_lbl="no games", y\_lbl="Aggr. Rating")
842. plt.show()

845. **if** \_\_name\_\_ == "\_\_main\_\_":
846. ct = Comparison\_Tool()
847. **print**(ct.map\_params\_aggression())
849. #ct.get\_data(Comparison\_Tool.ID\_NN, Comparison\_Tool.ID\_CC\_AI,Comparison\_Tool.ID\_SIMPLE, Comparison\_Tool.ID\_RAND\_AI, no\_games=2000)
850. #ct.output\_aggression\_over\_time("nn")
852. #ct.output\_aggression\_scale()
853. #ct.output\_aggression\_win\_relation()
854. #print(ct.get\_aggression\_rating("simple"))
856. #ct.output\_win\_margin\_at\_stand\_vs\_winrate()
857. #Comparison\_Tool.ID\_CC\_AI, Comparison\_Tool.ID\_NN, Comparison\_Tool.ID\_SIMPLE
858. #ct.get\_data(Comparison\_Tool.ID\_NN, no\_games=50)
859. #print(ct.db\_wrapper.execute\_queries(query, get\_result=True))

862. #ct.output\_avg\_stand\_value("cc\_ai")
863. #print(ct.db\_wrapper.get\_avg\_wr())
865. #q = """
866. #   SELECT \*
867. #   FROM Game\_Record
868. #  WHERE game\_id=2;
869. #  """
870. #print(ct.db\_wrapper.execute\_queries(q, get\_result=True))
872. #print(ct.get\_data(Comparison\_Tool.ID\_SIMPLE, Comparison\_Tool.ID\_CC\_AI, Comparison\_Tool.ID\_NN,
873. #                 Comparison\_Tool.ID\_RAND\_AI, no\_games=50000))
874. #connection, cursor = ct.db\_wrapper.execute\_queries(
875. #    "SELECT \* FROM Game\_Record", keep\_open=True)
876. #print(cursor.fetchone())
877. #connection.close()
879. #connection, cursor = ct.db\_wrapper.execute\_queries(
880. #    "SELECT \* FROM Moves WHERE player\_id='simple'", keep\_open=True
881. #)
882. #print(cursor.fetchall())
883. #connection.close()
885. #agent\_to\_analyse = Comparison\_Tool.ID\_SIMPLE
886. #print(agent\_to\_analyse, ct.get\_agent\_analysis(agent\_to\_analyse))
888. #ct.output\_player\_wr(agent\_to\_analyse)
889. #ct.output\_avg\_stand\_value(agent\_to\_analyse)
891. #print(ct.db\_wrapper.get\_stand\_val\_avg())
892. #print(ct.db\_wrapper.get\_stand\_val\_std\_dev())
893. #ct.output\_stand\_dist()
894. #ct.output\_hit\_dist()
896. #ct.output\_stand\_vs\_wr()
897. #ct.output\_hit\_vs\_br()
899. #q = """
900. #    SELECT \*
901. #    FROM Moves
902. #    WHERE hand\_val\_before=21 AND player\_id != 'rand' AND move=1;
903. #    """
905. #res = ct.db\_wrapper.execute\_queries(q, get\_result=True)
906. #total\_games = ct.db\_wrapper.execute\_queries(q1, get\_result=True)[0]

### CT\_GUI.py

1. **from** GUI **import** Window
2. **import** tkinter as tk
3. **from** Comparison\_Tool **import** Comparison\_Tool
4. **import** sys,os
5. sys.path.append(os.path.realpath("../DB"))
6. **from** Users\_DB **import** Users\_DB
8. """
9. - gui for comparison tool
10. - defines the classes for different windows in the GUI
11. """
13. # main menu class - window with buttons to traverse the gui
14. **class** Init\_Win(Window):
15. **def** \_\_init\_\_(self, parent, user\_type="user", root=None):
16. self.ID = "Main\_Menu"
17. self.ct = Comparison\_Tool()
18. self.user\_type = user\_type
19. self.root = root
20. super().\_\_init\_\_(parent)
22. **def** build\_widgets(self, fr):
23. self.title = tk.Label(fr, text="Comparison Tool")
24. self.title.grid(row=0, column=0)
26. self.iso\_comp\_btn = tk.Button(fr, text="Isolated Comparison",
27. command=**lambda**: self.open\_win("iso\_comp"))
28. self.iso\_comp\_btn.grid(row=1, column=0)
30. self.rel\_comp\_btn = tk.Button(fr, text="Relational Comparison",
31. command=**lambda**: self.open\_win("rel\_comp"))
32. self.rel\_comp\_btn.grid(row=2, column=0)
34. **if** self.user\_type == "admin":
35. self.gen\_stat\_btn = tk.Button(fr, text="General Statistics",
36. command=**lambda**: self.open\_win("gen\_stat"))
37. self.gen\_stat\_btn.grid(row=3, column=0)
39. self.data\_win\_btn = tk.Button(fr, text="Gen Data",
40. command=**lambda**: self.open\_win("get\_data"))
41. self.data\_win\_btn.grid(row=4, column=0)
43. self.update\_nn\_btn = tk.Button(fr, text="Update NN", command=self.update\_nn)
44. self.update\_nn\_btn.grid(row=5, column=0)
46. self.nn\_update\_result\_lbl = tk.Label(fr)

49. self.back\_btn = tk.Button(fr, text="Back", command=self.back)
50. self.back\_btn.grid(row=7, column=0)
52. # hides the main menu and runs the next window
53. **def** open\_win(self, win\_to\_open):
54. self.hide()
55. **if** win\_to\_open == "iso\_comp":
56. self.isolated\_comp = Iso\_Win(ct=self.ct, root=self, parent=tk.Toplevel(), user\_type=self.user\_type)
57. **elif** win\_to\_open == "rel\_comp":
58. self.rel\_comp = Rel\_Win(ct=self.ct, root=self, parent=tk.Toplevel(), user\_type=self.user\_type)
59. **elif** win\_to\_open == "gen\_stat":
60. self.gen\_stat = Gen\_Win(ct=self.ct, root=self, parent=tk.Toplevel())
61. **elif** win\_to\_open == "get\_data":
62. self.data\_win = Data\_Win(ct=self.ct, root=self, parent=tk.Toplevel())
64. # command for updating the nn
65. **def** update\_nn(self):
66. self.ct.update\_nn()
67. self.nn\_update\_result\_lbl.grid(row=7, column=0)
68. self.nn\_update\_result\_lbl.config(text="Done")
70. # goes back to login screen
71. **def** back(self):
72. self.root.show()
73. self.destroy()
75. # isolated user comparison
76. # enter user name and get data about them, in isolation
77. **class** Iso\_Win(Window):
78. **def** \_\_init\_\_(self, ct, root, parent, user\_type="user"):
79. super().\_\_init\_\_(parent)
80. self.ID = "iso\_comp"
81. self.ct = ct
82. self.root = root
83. self.default\_text = True
84. self.user\_type = user\_type
86. **def** build\_widgets(self, fr):
87. self.title = tk.Label(fr, text="Isolated Comparison")
88. self.title.grid(row=0, column=0)
90. self.un\_entry = tk.Entry(fr)
91. self.un\_entry.insert(0, "Enter Username here")
92. self.un\_entry.bind("<Button-1>", self.clear\_default)
93. self.un\_entry.grid(row=1, column=0)
95. self.build\_wr = tk.Button(fr, text="Output Winrates",
96. command=**lambda**: self.open\_command(self.un\_entry.get(), "wr"))
97. self.build\_wr.grid(row=2, column=0)
99. self.build\_ar = tk.Button(fr, text="Output Aggression",
100. command=**lambda**: self.open\_command(self.un\_entry.get(), "ar"))
101. self.build\_ar.grid(row=3, column=0)
103. self.back\_btn = tk.Button(fr, text="Back", command=self.back)
104. self.back\_btn.grid(row=4, column=0)
106. self.res\_label = tk.Label(fr, text="")
107. self.res\_label.grid(row=5,column=0)
109. **def** open\_command(self, id, type):
110. agent\_names = [Comparison\_Tool.ID\_NN, Comparison\_Tool.ID\_CC\_AI, Comparison\_Tool.ID\_SIMPLE, Comparison\_Tool.ID\_RAND\_AI]
111. valid\_id = self.ct.db\_wrapper.check\_valid\_id(id)
112. **if** **not** valid\_id **or** (self.user\_type != "admin" **and** id **in** agent\_names):
113. self.res\_label.config(text="User not Found")
114. **return** False
115. self.res\_label.config(text="User Found")
116. **if** type == "wr":
117. self.ct.output\_player\_wr(id)
118. **elif** type == "ar":
119. self.ct.output\_aggression\_over\_time(id)
121. # method for clearing the entry box if default text is inside
122. **def** clear\_default(self, \*args):
123. **if** self.default\_text:
124. self.default\_text = False
125. self.un\_entry.delete(0, "end")
127. **def** back(self):
128. self.root.show()
129. self.destroy()
131. # get realtional data about two users
132. # currently has no implementation
133. **class** Rel\_Win(Window):
134. **def** \_\_init\_\_(self, ct, root, parent, user\_type="user"):
135. super().\_\_init\_\_(parent)
136. self.ID = "rel\_comp"
137. self.ct = ct
138. self.root = root
139. self.user\_type = user\_type
141. self.default\_text\_1 = True
142. self.default\_text\_2 = True
144. **def** build\_widgets(self, fr):
145. self.title = tk.Label(fr, text="Relational Comparison")
146. self.title.grid(row=0, column=0)
148. self.un1\_entry = tk.Entry(fr)
149. self.un1\_entry.insert(0, "Enter Username 1 here")
150. self.un1\_entry.bind("<Button-1>", **lambda** \*args: self.clear\_default(1))
151. self.un1\_entry.grid(row=1, column=0)
153. self.un2\_entry = tk.Entry(fr)
154. self.un2\_entry.insert(0, "Enter Username 2 here")
155. self.un2\_entry.bind("<Button-1>", **lambda** \*args: self.clear\_default(2))
156. self.un2\_entry.grid(row=2, column=0)
158. self.realtions\_btn = tk.Button(fr, text="Relations") # implement the command for this
159. self.realtions\_btn.grid(row=3, column=0)
161. self.back\_btn = tk.Button(fr, text="Back", command=self.back)
162. self.back\_btn.grid(row=4, column=0)
164. self.res\_label = tk.Label(fr, text="")
165. self.res\_label.grid(row=5, column=0)
167. **def** clear\_default(self, entry\_id, \*args):
168. **if** entry\_id == 1 **and** self.default\_text\_1:
169. self.default\_text\_1 = False
170. self.un1\_entry.delete(0, "end")
171. **elif** entry\_id == 2 **and** self.default\_text\_2:
172. self.default\_text\_2 = False
173. self.un2\_entry.delete(0, "end")
175. **def** back(self):
176. self.root.show()
177. self.destroy()
179. # window for general data about all users
180. **class** Gen\_Win(Window):
181. **def** \_\_init\_\_(self, ct, root, parent, geometry="400x400"):
182. super().\_\_init\_\_(parent, geometry)
183. self.ID = "gen\_stat"
184. self.ct = ct
185. self.root = root
187. **def** build\_widgets(self, fr):
188. self.title = tk.Label(fr, text="General Stats")
189. self.title.grid(row=0, column=0)
191. self.stand\_dist\_btn = tk.Button(fr, text="Stand Value Distribution",
192. command=**lambda**: self.display\_dist\_cmd("stand\_dist"))
193. self.stand\_dist\_btn.grid(row=1, column=0)
195. self.hit\_dist\_btn = tk.Button(fr, text="Hit Value Distribution",
196. command=**lambda**: self.display\_dist\_cmd("hit\_dist"))
197. self.hit\_dist\_btn.grid(row=2, column=0)
199. self.stand\_vs\_wr\_btn = tk.Button(fr, text="Win Rate Against Stand Value",
200. command=**lambda**: self.display\_dist\_cmd("stand\_win"))
201. self.stand\_vs\_wr\_btn.grid(row=3, column=0)
203. self.hit\_vs\_bust\_btn = tk.Button(fr, text="Bust Rate Against Hit Value",
204. command=**lambda**: self.display\_dist\_cmd("hit\_bust"))
205. self.hit\_vs\_bust\_btn.grid(row=4, column=0)
207. self.aggr\_vs\_wr\_btn = tk.Button(fr, text="Aggression Rating Vs Win Rate",
208. command=**lambda**: self.display\_dist\_cmd("aggr\_win"))
209. self.aggr\_vs\_wr\_btn.grid(row=5, column=0)
211. self.back\_btn = tk.Button(fr, text="Back", command=self.back)
212. self.back\_btn.grid(row=6, column=0)
214. **def** display\_dist\_cmd(self, dist\_type):
215. **if** dist\_type == "hit\_dist":
216. self.ct.output\_hit\_dist()
217. **elif** dist\_type == "stand\_dist":
218. self.ct.output\_stand\_dist()
219. **elif** dist\_type == "stand\_win":
220. self.ct.output\_stand\_vs\_wr()
221. **elif** dist\_type == "hit\_bust":
222. self.ct.output\_hit\_vs\_br()
223. **elif** dist\_type == "aggr\_win":
224. self.ct.output\_aggression\_win\_relation()
226. **def** back(self):
227. self.root.show()
228. self.destroy()
230. # window for generating new data
231. **class** Data\_Win(Window):
232. **def** \_\_init\_\_(self, ct, root, parent, geometry="400x400"):
233. super().\_\_init\_\_(parent, geometry)
234. self.ID = "get\_data"
235. self.ct = ct
236. self.root = root
238. self.default\_text = True
240. **def** build\_widgets(self, fr):
241. self.title\_lbl = tk.Label(fr, text="Get Data")
242. self.title\_lbl.grid(row=0, column=0)
244. self.no\_games\_ent = tk.Entry(fr)
245. self.no\_games\_ent.insert(0, "Enter num of games to play here")
246. self.no\_games\_ent.bind("<Button-1>", self.clear\_default)
247. self.no\_games\_ent.grid(row=1, column=0)
249. self.instr\_lbl = tk.Label(fr, text="Tick the AI you want to use in your sample")
250. self.instr\_lbl.grid(row=2, column=0)
252. self.nn\_var = tk.IntVar()
253. self.nn\_check = tk.Checkbutton(fr, text="nn", variable=self.nn\_var)
254. self.nn\_check.grid(row=3, column=0)
256. self.cc\_ai\_var = tk.IntVar()
257. self.cc\_ai\_check = tk.Checkbutton(fr, text="cc ai", variable=self.cc\_ai\_var)
258. self.cc\_ai\_check.grid(row=4, column=0)
260. self.simple\_var = tk.IntVar()
261. self.simple\_check = tk.Checkbutton(fr, text="simple", variable=self.simple\_var)
262. self.simple\_check.grid(row=5, column=0)
264. self.rand\_var = tk.IntVar()
265. self.rand\_check = tk.Checkbutton(fr, text="rand ai", variable=self.rand\_var)
266. self.rand\_check.grid(row=6, column=0)
268. self.begin\_btn = tk.Button(fr, text="Begin", command=self.begin\_sample)
269. self.begin\_btn.grid(row=7, column=0)
271. self.back\_btn = tk.Button(fr, text="Back", command=self.back)
272. self.back\_btn.grid(row=8, column=0)
274. self.res\_lbl = tk.Label(fr, text="")
275. self.res\_lbl.grid(row=9, column=0)
277. **def** begin\_sample(self):
278. agents\_playing = []
279. checkboxes = [self.nn\_var, self.cc\_ai\_var, self.simple\_var, self.rand\_var]
280. names = [self.ct.ID\_NN, self.ct.ID\_CC\_AI, self.ct.ID\_SIMPLE, self.ct.ID\_RAND\_AI]
282. # check checkbox state, if checked append to list of agent ids who are playing
283. **for** i **in** range(len(checkboxes)):
284. cbox\_state = checkboxes[i].get()
285. name = names[i]
286. **if** cbox\_state:
287. agents\_playing.append(name)
288. **if** agents\_playing == []:
289. self.res\_lbl.config(text="Please select at least one ai to play")
290. **return** False
292. no\_games = self.no\_games\_ent.get()
293. **try**:
294. no\_games = int(no\_games)
295. **except**:
296. self.res\_lbl.config(text="Please enter a valid number of games")
297. **return** False
299. self.res\_lbl.config(text="Commencing testing for {0} games".format(str(no\_games)))
300. self.ct.get\_data(agents\_playing, no\_games=no\_games)
301. self.res\_lbl.config(text="Data gathered and pushed")
302. **return** True
304. **def** clear\_default(self, \*args):
305. **if** self.default\_text:
306. self.default\_text = False
307. self.no\_games\_ent.delete(0, "end")
309. **def** back(self):
310. self.root.show()
311. self.destroy()
313. # Window for logging in
314. # allows the user to login or signup#
315. # TODO REMOVE THE AUTO FILL FOR ADMIN
316. # sample admin: admin, Pw1
317. # sample user: mr\_aqa, Pw2
318. **class** Login\_Win(Window):
319. **def** \_\_init\_\_(self, parent=None, geometry="400x400"):
320. super().\_\_init\_\_(tk.Tk(), geometry)
321. self.ID = "Login"
322. self.db\_wrapper = Users\_DB("DB/Blackjack.sqlite")
323. self.uname\_default = True
324. self.pword\_default = True

327. **def** build\_widgets(self, fr):
328. self.title\_lbl = tk.Label(fr, text="Login")
329. self.title\_lbl.grid(row=0, column=0)
331. self.un\_ent = tk.Entry(fr)
332. self.un\_ent.insert(0, "admin") # todo CHANGE THIS
333. self.un\_ent.bind("<Button-1>", **lambda** \*args: self.clear\_default("un"))
334. self.un\_ent.grid(row=1, column=0)
336. self.pw\_ent = tk.Entry(fr, show="\*")
337. self.pw\_ent.insert(0, "Pw1") # TODO CHANGE THIS
338. self.pw\_ent.bind("<Button-1>", **lambda** \*args: self.clear\_default("pw"))
339. self.pw\_ent.grid(row=2, column=0)
341. self.login\_btn = tk.Button(fr, text="Login", command=**lambda**: self.login(self.un\_ent.get(), self.pw\_ent.get()))
342. self.login\_btn.grid(row=3, column=0)
344. self.signup\_btn = tk.Button(fr, text="Sign up", command=**lambda**: self.sign\_up(self.un\_ent.get(), self.pw\_ent.get()))
345. self.signup\_btn.grid(row=4, column=0)
347. self.res\_lbl = tk.Label(fr, text="")
348. self.res\_lbl.grid(row=5, column=0)
350. **def** clear\_default(self, type, \*args):
351. **if** self.uname\_default **and** type == "un":
352. self.uname\_default = False
353. self.un\_ent.delete(0, "end")
354. **elif** self.pword\_default **and** type == "pw":
355. self.pword\_default = False
356. self.pw\_ent.delete(0, "end")
358. **def** login(self, username, password):
359. valid\_login = self.db\_wrapper.check\_login(username, password)
360. **if** **not** valid\_login:
361. self.res\_lbl.config(text="Invalid Login")
362. **return** False
363. self.res\_lbl.config(text="Login Successful!")
364. user\_type = self.db\_wrapper.get\_user\_type(username, password)
365. self.open\_win(user\_type)
366. **return** True
368. # creates a new user
369. # todo create specific error message if signup not successful - ie pw or uname error
370. # only creates users - cannot create admins
371. **def** sign\_up(self, username, password):
372. result = self.db\_wrapper.create\_new\_user(username, password, type="user")
373. result\_text = ""
374. **if** result == False:
375. result\_text = "Signup Unsuccessful"
376. **else**:
377. result\_text = "Signup Successful!"
378. self.res\_lbl.config(text=result\_text)

381. **def** open\_win(self, user\_type):
382. self.hide()
383. self.menu = Init\_Win(tk.Toplevel(), user\_type, root=self)

386. **if** \_\_name\_\_ == "\_\_main\_\_":
387. log\_win = Login\_Win()
388. log\_win.db\_wrapper.create\_new\_user("admin", "Pw1", type="admin")
389. **print**(log\_win.db\_wrapper.create\_new\_user("mr\_aqa", "Pw2"))
390. log\_win.run()
392. #g = Init\_Win(tk.Tk())
393. #g.run()
394. #g.ct.update\_nn()

### Deck.py

1. **from** Structs.Stack **import** Stack
2. **from** enum **import** Enum
3. **from** random **import** shuffle
5. """
6. - defines functionality for the stack-backed deck
7. - defines the card class
8. """
10. # defines enums for suits and royals
11. # improves readability of code
12. **class** Suits(Enum):
13. HEARTS = "Hearts"
14. DIAMONDS = "Diamonds"
15. CLUBS = "Clubs"
16. SPADES = "Spades"
18. **class** Royals(Enum):
19. JACK = 11
20. QUEEN = 12
21. KING = 13
22. ACE = 14
24. # Card made up of a suit and a value
25. # and some simple behaviours
26. **class** Card:
27. **def** \_\_init\_\_(self, suit, value):
28. self.\_\_suit = suit
29. self.\_\_value = value
31. @property
32. **def** value(self):
33. **return** self.\_\_value
35. @property
36. **def** suit(self):
37. **return** self.\_\_suit
39. **def** isRoyal(self):
40. **return** isinstance(self.\_\_value, Royals) **and** **not** self.isAce()
42. **def** isAce(self):
43. **return** self.\_\_value == Royals.ACE
45. **def** \_\_str\_\_(self):
46. **return** "{} {}".format(self.\_\_suit, self.\_\_value)
48. **def** \_\_eq\_\_(self, other\_card):
49. **if** isinstance(self, other\_card.\_\_class\_\_):
50. **return** (self.\_\_dict\_\_ == other\_card.\_\_dict\_\_)
51. **return** False
53. # stack-backed deck, auto shuffle, when there is nothign in the deck
54. **class** Deck(Stack):
55. **def** \_\_init\_\_(self):
56. self.\_\_DECK\_SIZE = 52
57. self.\_\_deckIteration = 1
58. self.\_\_autoShuffleWhenEmpty = True
59. self.\_\_suits = [suit **for** suit **in** Suits]
60. self.\_\_values = [num **for** num **in** range(2, 11)] + [royal **for** royal **in** Royals]
61. super().\_\_init\_\_(self.\_\_DECK\_SIZE)
62. self.init\_deck()
64. @property
65. **def** deckIteration(self):
66. **return** self.\_\_deckIteration
68. **def** autoShuffleOff(self):
69. self.\_\_autoShuffleWhenEmpty = False
71. **def** autoShuffleOn(self):
72. self.\_\_autoShuffleWhenEmpty = True
74. # populates the deck randomly
75. **def** init\_deck(self):
76. temp\_deck = []
77. **for** suit **in** self.\_\_suits:
78. **for** value **in** self.\_\_values:
79. temp\_deck.append(Card(suit, value))
80. shuffle(temp\_deck)
81. **for** card **in** temp\_deck:
82. self.push(card)
84. # Removes top object from stack - autoshuffles deck when it is empty
85. **def** pop(self):
86. popped = super().pop()
87. **if** self.isEmpty() **and** self.\_\_autoShuffleWhenEmpty:
88. self.init\_deck()
89. self.\_\_deckIteration += 1
90. **return** popped
92. # test method showing contents of deck
93. **def** display\_and\_empty(deck):
94. **while** **not** deck.isEmpty:
95. **print**(deck.pop)
97. **if** \_\_name\_\_ == "\_\_main\_\_":
98. d = Deck()
99. **print**(isinstance(Card(Suits.SPADES, Royals.QUEEN).value, Royals))

### GUI.py

1. """
2. - abstract parent class for gui
3. - defines required methods
4. - and default implementations
5. """
7. **import** tkinter as tk
8. **from** abc **import** ABC, abstractmethod
10. **class** Window(ABC):
11. **def** \_\_init\_\_(self, parent=None, geometry="400x400"):
12. self.parent = parent
13. **if** parent **is** None:
14. self.parent = tk.Tk()
15. self.hidden = False
16. #self.parent.geometry(geometry)
17. self.body = tk.Frame(self.parent)
18. self.body.grid()
20. self.build\_widgets(self.body)
22. @abstractmethod
23. **def** build\_widgets(self, fr):
24. **pass**
26. **def** run(self):
27. self.parent.mainloop()
29. **def** hide(self):
30. self.parent.withdraw()
31. self.hidden = True
33. **def** show(self):
34. self.parent.deiconify()
35. self.hidden = False
37. **def** destroy(self):
38. self.parent.destroy()

### Moves.py

1. **from** enum **import** Enum
3. """
4. - defines enum for move, with conversion between move and bit
5. - prevents having to write every move as TRUE and FALSE
6. - improves readability of code
7. """
9. **class** Moves(Enum):
10. HIT = True
11. STAND = False
13. @staticmethod
14. **def** convert\_to\_bit(move):
15. **if** isinstance(move, bool):
16. **return** move
17. **elif** isinstance(move, int) **and** move == 0 **or** move == 1:
18. **return** move
19. **elif** move == Moves.HIT:
20. **return** 1
21. **elif** move == Moves.STAND:
22. **return** 0
24. @staticmethod
25. **def** convert\_to\_move(boolean):
26. **if** isinstance(boolean, bool):
27. **return** boolean
28. **elif** boolean == True:
29. **return** Moves.HIT
30. **elif** boolean == False:
31. **return** Moves.STAND
33. **if** \_\_name\_\_ == "\_\_main\_\_":
34. **print**(Moves.convert\_to\_bit(Moves.HIT))
35. **print**(Moves.convert\_to\_move(1))

### Rand\_AI.py

1. **from** Blackjack **import** Hand
2. **from** Agent **import** Agent
3. **from** Moves **import** Moves
4. **from** random **import** randint
6. """
7. - Agent which takes a random action every turn
8. - Control Agent
9. """
11. **class** Rand\_AI(Agent):
12. **def** \_\_init\_\_(self, hand=None):
13. super().\_\_init\_\_(ID="rand", type=["Random"])
14. self.hand = hand
15. **if** hand **is** None:
16. self.hand = Hand(self.ID)
18. **def** get\_move(self, \*args):
19. move\_bit = randint(0, 1)
20. **return** Moves.convert\_to\_move(move\_bit)
22. **def** update\_end\_game(self, new\_cards):
23. **pass**

### Simple\_AI.py

1. **from** Blackjack **import** Blackjack
2. **from** Blackjack **import** Hand
3. **from** Agent **import** Agent
4. **from** Moves **import** Moves
6. """
7. - threshold based agent, which makes it's own calculations
8. - behaviour is defined by thresholds
9. """
11. **class** Simple\_AI(Agent):
12. **def** \_\_init\_\_(self, hand=None, parameters=None):
13. super().\_\_init\_\_(ID="simple", type=["Simple"])
14. self.blackjack\_value = 21
15. self.maxCard = 11
16. self.bust\_value = self.blackjack\_value + 1
17. self.hand = hand
19. **if** parameters **is** None:
20. self.set\_parameters(setting="default")
21. **if** hand **is** None:
22. self.hand = Hand(self.ID)
24. **def** set\_parameters(self, setting="default"):
25. **if** setting == "default":
26. self.bust\_threshold = 5
27. self.win\_margin\_threshold = 5
28. self.min\_hand\_threshold = 15
29. **elif** setting == "aggressive":
30. self.bust\_threshold = 3
31. self.win\_margin\_threshold = 6
32. self.min\_hand\_threshold = 18
33. **elif** setting == "passive":
34. self.bust\_threshold = 7
35. self.win\_margin\_threshold = 3
36. self.min\_hand\_threshold = 13
38. # returns decision to hit or not -> true => hit, false => stand
39. # stands if blackjacked or winning by a sufficient amount
40. # otherwise hits if cannot go bust, or under the bust margin threshold
41. **def** get\_move(self, all\_players):
42. next\_best\_hand = self.get\_best\_hand(all\_players)
43. best\_player\_value = next\_best\_hand.get\_value()
44. best\_player\_stood = next\_best\_hand.has\_stood
46. hand\_value = self.hand.get\_value()
47. win\_margin = hand\_value - best\_player\_value
49. # if blackjack'd or winning by a sufficient amount
50. **if** hand\_value == self.blackjack\_value **or** \
51. (win\_margin > self.win\_margin\_threshold **and** hand\_value > self.min\_hand\_threshold):
52. **return** Moves.STAND
53. # If cannot go bust, or edge case satisfied then hit
54. **elif** (hand\_value < (self.bust\_value - self.maxCard)
55. **or** self.edge\_move\_calc(hand\_value, best\_player\_value)):
56. **return** Moves.HIT
57. **return** Moves.STAND
59. # HAVE A LOOK AT THIS
60. **def** edge\_move\_calc(self, hand\_value, best\_value):
61. bustDiff = abs(hand\_value - self.bust\_value) # how far off being bust SAI is
62. LTBestPlayer = hand\_value < best\_value **and** best\_value <= 21
63. **if** LTBestPlayer **or** bustDiff <= self.bust\_threshold:
64. **return** True
65. **return** False
67. **def** update\_end\_game(self, new\_cards):
68. **pass**

## /DB

### Create\_Agents\_Table.sql

1. **CREATE** **TABLE** IF NOT EXISTS Agents (
2. agent\_id **varchar**(255) **PRIMARY** **KEY**,
3. description TEXT,
4. games\_won **INTEGER** NOT NULL **DEFAULT** 0,
5. games\_played **INTEGER** NOT NULL **DEFAULT** 0
6. );

### Create\_Card\_Counter\_Record.sql

1. **CREATE** **TABLE** IF NOT EXISTS Card\_Counter\_Record (
2. game\_id **INTEGER** NOT NULL,
3. turn\_num **INTEGER** NOT NULL,
4. bust **FLOAT** NOT NULL,
5. blackjack **FLOAT** NOT NULL,
6. exceedWinningPlayer **FLOAT** NOT NULL,
7. alreadyExceedingWinningPlayer **BIT** NOT NULL,
8. **move** **BIT** NOT NULL,
9. trained **BIT** **DEFAULT** 0,
10. **FOREIGN** **KEY** (game\_id) **REFERENCES** Game\_Record(game\_id),
11. **FOREIGN** **KEY** (turn\_num) **REFERENCES** Moves(turn\_num),
12. **PRIMARY** **KEY**(game\_id, turn\_num)
13. );

### Create\_Games\_Record.sql

1. **CREATE** **TABLE** IF NOT EXISTS Game\_Record (
2. game\_id **INTEGER** NOT NULL,
3. winner\_ids TEXT NOT NULL,
4. winning\_hands TEXT NOT NULL,
5. winning\_values TEXT NOT NULL,
6. num\_of\_turns **INTEGER** NOT NULL,
7. players TEXT NOT NULL,
8. **PRIMARY** **KEY**(game\_id)
9. );
11. **CREATE** **TABLE** IF NOT EXISTS Moves (
12. player\_id **varchar**(255) NOT NULL,
13. game\_id **INTEGER** NOT NULL,
14. turn\_num **INTEGER** NOT NULL,
15. next\_best\_val **INTEGER**,
16. hand\_val\_before **INTEGER** NOT NULL,
17. **move** **BIT** NOT Null, -- 1 -> Hit, 0 -> False
18. hand\_val\_after **INTEGER** NOT NULL,
19. **FOREIGN** **KEY** (game\_id) **REFERENCES** Game\_Record(game\_id),
20. **PRIMARY** **KEY**(player\_id, game\_id, turn\_num)
21. );

### Create\_Users\_Table.sql

1. **CREATE** **TABLE** IF NOT EXISTS users (
2. username **varchar**(32) NOT NULL,
3. **password** **varchar**(256) NOT NULL, -- store hashed password
4. games\_won **INTEGER** NOT NULL **DEFAULT** 0,
5. games\_played **INTEGER** NOT NULL **DEFAULT** 0,
6. type **varchar**(32) NOT NULL **DEFAULT** "user",
7. **PRIMARY** **KEY**(username)
8. );

### Populate\_Agents.sql

2. **INSERT** **INTO** "Agents" ("agent\_name", "description") **VALUES**("nn", "neural network based agent. Takes in CC chances and hand values of itself and the best player as features");
3. **INSERT** **INTO** "Agents" ("agent\_name", "description") **VALUES**("cc\_ai", 'Standard Card counting ai, where behaviour depends on chance thresholds.');
4. **INSERT** **INTO** "Agents" ("agent\_name", "description") **VALUES**("simple", "Simple AI, behaviour based on if possible to go bust and residual between agent and best player");

### CT\_Wrapper.py

1. **import** os,sys
2. sys.path.append(os.path.realpath(".."))
3. **from** DB\_Wrapper **import** DB\_Wrapper
4. **from** Moves **import** Moves
5. **from** os **import** remove
6. **from** math **import** sqrt
8. """
9. - wrapper and interface for the database for comparison tool
10. """
12. **class** CT\_Wrapper(DB\_Wrapper):
13. **def** \_\_init\_\_(self, db\_path="blackjack.sqlite"):
14. super().\_\_init\_\_(db\_path)
15. self.\_\_tables\_id = ["Agents", "Moves", "Game\_Record"]
16. self.init\_tables()
17. self.init\_default\_agents()
19. # Creates the required tables - harcoded in -> TODO Change this from hardcoded?
20. **def** init\_tables(self):
21. **global** db\_dir\_path
22. sql\_files = ["Create\_Games\_Record.sql", "Create\_Agents\_Table.sql", "Create\_Users\_Table.sql",
23. "Create\_Card\_Counter\_Record.sql"]
24. **for** sql\_f **in** sql\_files:
25. self.execute\_queries\_from\_file(db\_dir\_path + sql\_f)
27. # pushes the agents into the table, if they do not exist
28. **def** init\_default\_agents(self):
29. agents = [
30. ["nn", "Neural Network based AI, card counter"],
31. ["cc\_ai", "Card Counting, threshold based AI"],
32. ["simple", "Simple AI based on game state thresholds"],
33. ["rand", "Control agent which takes a random move every turn"]
34. ]
35. self.populate\_agents\_table(agents[0], agents[1], agents[2], agents[3])
37. # pass in all the parameters required to push a move to the move table
38. # this method will push the move to the move table
39. # TODO INSERT SOME ERROR HANDLING AND DEFENSIVE PROGRAMMING
40. **def** push\_move(self, agent\_id, game\_id, turn\_num, move, next\_best\_val, hand\_val\_before, hand\_val\_after):
41. move = Moves.convert\_to\_bit(move)
42. query = """INSERT INTO "Moves"
43. (player\_id, game\_id, turn\_num, next\_best\_val, hand\_val\_before, move, hand\_val\_after) \
44. VALUES ("{0}", {1}, {2}, {3}, {4}, {5}, {6});""".format(agent\_id, game\_id, turn\_num,
45. next\_best\_val, hand\_val\_before, move,
46. hand\_val\_after)
47. self.execute\_queries(query)
49. # push the a game to the game record table
50. # agents => array of Agents Instances
51. # winning\_hands => Array of Hand Instances
52. # convert winners to text
53. **def** push\_game(self, game\_id, winners, winning\_hands, num\_of\_turns, agents):
54. wnr\_hands = ""
55. wnr\_vals = ""
56. **for** hand **in** winning\_hands:
57. hand\_as\_text = self.convert\_hand\_to\_text(hand)
58. wnr\_hands += hand\_as\_text + ";"
60. winning\_val = hand.get\_value()
61. wnr\_vals += str(winning\_val) + ";"
62. agents\_as\_text = self.convert\_agents\_to\_text(agents)
63. wnr\_ids = ";".join(winners) + ";"
64. query = """
65. INSERT INTO Game\_Record
66. (game\_id, winner\_ids, winning\_hands, winning\_values, num\_of\_turns, players)
67. VALUES ({0}, '{1}', '{2}', '{3}', {4}, '{5}');
68. """.format(game\_id, wnr\_ids, wnr\_hands, wnr\_vals, num\_of\_turns, agents\_as\_text)
69. self.execute\_queries(query)
71. # increment the winners and the games played in the database
72. **for** agent\_id **in** winners:
73. **if** agent\_id == "dealer":
74. **continue**
75. self.inc\_agent\_win(agent\_id)
76. **for** agent\_id **in** agents:
77. self.inc\_games\_played(agent\_id)
78. **pass**
80. # pushes the state of the cc at a particular move
81. **def** push\_cc(self, game\_id, turn\_num, bust, blackjack, exceedWinningPlayer, alreadyExceedingWinningPlayer, move):
82. move = Moves.convert\_to\_bit(move)
83. alreadyExceedingWinningPlayer = int(alreadyExceedingWinningPlayer)
84. query = """
85. INSERT INTO Card\_Counter\_Record
86. (game\_id, turn\_num, bust, blackjack, exceedWinningPlayer, alreadyExceedingWinningPlayer, move)
87. VALUES ({0}, {1}, {2}, {3}, {4}, {5}, {6})
88. """.format(game\_id, turn\_num, bust, blackjack, exceedWinningPlayer, alreadyExceedingWinningPlayer, move)
89. self.execute\_queries(query)
91. # pass in a hand of cards
92. # returns the hand as a string in the format:
93. # "({value} of {suit}), ({value} of {suit})" .. -> etc
94. **def** convert\_hand\_to\_text(self, hand):
95. hand\_as\_text = ""
96. **for** card **in** hand.hand:
97. **if** hand\_as\_text == "":
98. hand\_as\_text += "({0} of {1})".format(card.value, card.suit)
99. **else**:
100. hand\_as\_text += ", ({0} of {1})".format(card.value, card.suit)
101. **return** hand\_as\_text
103. # pass in agent name, and a game\_id
104. # returns the turn\_num, next\_best\_val, hand\_val\_before, move, hand\_val\_after
105. # pass in a game move if moves from a particular game wanted
106. # or else it will return all the moves the agent has made
107. # returns tuple of move records
108. **def** get\_agent\_moves(self, agent, game\_id=None):
109. # selects all moves
110. **if** game\_id **is** None:
111. query = """SELECT turn\_num, next\_best\_val, hand\_val\_before, move, hand\_val\_after
112. FROM Moves WHERE player\_id='{0}'""".format(agent)
113. **else**:
114. query = """SELECT turn\_num, next\_best\_val, hand\_val\_before, move, hand\_val\_after
115. FROM Moves WHERE player\_id='{0}' AND game\_id={1}""".format(agent, game\_id)
116. results = self.execute\_queries(query, get\_result=True)
118. # convert moves from bit to move
119. toReturn = []
120. **for** result **in** results:
121. record = list(result)
122. move\_as\_move = Moves.convert\_to\_move(record[3])
123. record[3] = move\_as\_move
124. toReturn.append(record)
125. **return** tuple(toReturn)
127. # abstract method for incrementing a field in the agents table
128. **def** inc\_agent(self, field, agent\_id):
129. get\_curr\_query = "SELECT {0} FROM Agents WHERE agent\_id='{1}'".format(field, agent\_id)
130. game\_data = self.execute\_queries(get\_curr\_query, get\_result=True)[0][0]
132. inc\_win\_query = """
133. UPDATE Agents
134. SET {0}={1}
135. WHERE agent\_id='{2}';
136. """.format(field, game\_data+1, agent\_id)
137. self.execute\_queries(inc\_win\_query)
139. # increments the win field of the passed agent
140. **def** inc\_agent\_win(self, agent\_id):
141. self.inc\_agent("games\_won", agent\_id)
143. # increments the games played for each of the agents passed
144. **def** inc\_games\_played(self, \*args):
145. **for** agent\_id **in** args:
146. self.inc\_agent("games\_played", agent\_id)
148. # returns the next available game id
149. # game id has to exist in both the moves table and the game record table
150. **def** get\_next\_game\_id(self):
151. q\_moves = """
152. SELECT MAX(game\_id)
153. FROM Moves
154. """
155. q\_gr = """
156. SELECT MAX(game\_id)
157. FROM Game\_Record
158. """
159. max\_moves = self.execute\_queries(q\_moves, get\_result=True)[0][0]
160. max\_gr = self.execute\_queries(q\_gr, get\_result=True)[0][0]
162. # returns the larger game id of the two
163. # if max\_moves is none and max\_gr is none then there is nothing in the database
164. **if** max\_moves **is** None:
165. **if** max\_gr **is** None:
166. game\_id\_test = 1
167. **else**:
168. game\_id\_test = max\_gr + 1
169. **else**:
170. **if** max\_gr **is** None:
171. game\_id\_test = max\_moves + 1
172. **else**:
173. game\_id\_test = max(max\_moves, max\_gr) + 1
174. **return** game\_id\_test
176. # pass in array of agent instances
177. # returns string which is formatted for database suitability
178. **def** convert\_agents\_to\_text(self, agents):
179. agent\_text = ""
180. **for** agent **in** agents:
181. **if** isinstance(agent, str):
182. agent\_name = agent
183. **else**: # TODO CHANGE THIS IN A WAY SO THAT ITS NOT SHIT
184. agent\_name = agent.id
185. agent\_text += agent\_name + ";"
186. **return** agent\_text
188. # pass in arrays [agents ids, desc], method will populate them in the database
189. **def** populate\_agents\_table(self, \*args):
190. queries = []
191. **for** arg **in** args:
192. agent\_name = arg[0]
193. agent\_desc = arg[1]
195. **if** self.agent\_exists(agent\_name):
196. **continue**
198. queries.append("""
199. INSERT INTO Agents (agent\_id, description, games\_won, games\_played)
200. VALUES ('{0}', '{1}', 0, 0);
201. """.format(agent\_name, agent\_desc))
202. self.execute\_queries(queries)
204. # pass in agent id, returns true or false depending on whether the agent
205. # already exists in the databse or not
206. **def** agent\_exists(self, id):
207. query = """
208. SELECT \*
209. FROM Agents
210. WHERE agent\_id='{0}'
211. """.format(id)
212. result = self.execute\_queries(query, get\_result=True)
213. **if** result == []:
214. **return** False
215. **return** True
217. # pass in an agent id and get the winrate as a decimal
218. **def** get\_agent\_win\_rate(self, agent\_id):
219. # get the number of wins
220. query = """
221. SELECT games\_won, games\_played
222. FROM Agents
223. WHERE agent\_id='{0}'
224. """.format(agent\_id)
225. result = self.execute\_queries(query, get\_result=True)[0]
226. winrate = result[0] / result[1]
227. **return** winrate
229. # todo contextualise this value -> look at opponents hand values before stand also
230. # queries database, returns average stand value players stand on
231. **def** get\_stand\_val\_avg(self):
232. query = """
233. SELECT AVG(hand\_val\_before)
234. FROM Moves
235. WHERE move=0;
236. """
237. result = self.execute\_queries(query, get\_result=True)
238. **return** result[0][0]
240. # gets the standard deviation to the mean of the average value which the players stand on
241. # todo test test test
242. **def** get\_stand\_val\_std\_dev(self):
243. avg\_val = self.get\_stand\_val\_avg()
245. query = """
246. SELECT hand\_val\_before
247. FROM Moves
248. WHERE move=0;
249. """
250. results = self.execute\_queries(query, get\_result=True)
252. s = 0 # sum of (x - xbar)^2
253. n = len(results)
254. **for** result **in** results:
255. s += (result[0] - avg\_val) \*\* 2
257. std\_dev = sqrt(s / (n-1))
258. **return** std\_dev
260. # returns true if id passed is a valid user ID
261. **def** check\_valid\_id(self, id):
262. query\_agents = """
263. SELECT \*
264. FROM Agents
265. WHERE agent\_id='{0}';
266. """.format(id)
268. query\_users = """
269. SELECT \*
270. FROM Users
271. WHERE username='{0}';
272. """.format(id)
274. agents\_result = self.execute\_queries(query\_agents, get\_result=True)
275. users\_result = self.execute\_queries(query\_users, get\_result=True)
277. **return** agents\_result != [] **or** users\_result != []
279. # gets average winrate for the agents, returns as a decimal
280. **def** get\_avg\_wr(self):
281. q = """
282. SELECT games\_won, games\_played, agent\_id
283. FROM Agents
284. """
285. res = self.execute\_queries(q, get\_result=True)
286. **print**(res)
287. win\_rates = [i[0] / i[1] **for** i **in** res]
288. **print**(win\_rates)
289. avg\_winrate = sum(win\_rates) / len(win\_rates)
290. **return** avg\_winrate

293. **if** \_\_name\_\_ == "\_\_main\_\_":
294. db\_dir\_path = ""
296. ct\_w = CT\_Wrapper()
297. ct\_w.execute\_queries\_from\_file("Create\_Games\_Record.sql")
298. ct\_w.execute\_queries("INSERT INTO 'Moves' (player\_id, game\_id, turn\_num, next\_best\_val, hand\_val\_before, move, hand\_val\_after) VALUES ('asdf', 1, 3, 5, 0, 0, 10);")
299. ct\_w.execute\_queries("INSERT INTO 'Moves' (player\_id, game\_id, turn\_num, next\_best\_val, hand\_val\_before, move, hand\_val\_after) VALUES ('asdf', 2, 3, 5, 0, 0, 10);")
301. ct\_w.execute\_queries("INSERT INTO 'Game\_Record' (game\_id, winner\_id, winning\_hand, winning\_hand\_value, num\_of\_turns, players) VALUES (1, 'asdf', 'agfdsg', 3, 5, 'asdf');")
302. ct\_w.execute\_queries("INSERT INTO 'Game\_Record' (game\_id, winner\_id, winning\_hand, winning\_hand\_value, num\_of\_turns, players) VALUES (2, 'asdf', 'agfdsg', 3, 5, 'asdfsad');")
303. **print**(ct\_w.get\_next\_game\_id())
304. connection, cursor = ct\_w.execute\_queries("SELECT \* FROM Moves", keep\_open=True)
305. **for** i **in** cursor:
306. **print**(i)
307. connection.close()
309. ct\_w.execute\_queries\_from\_file("Create\_Agents\_Table")
310. ct\_w.execute\_queries("INSERT INTO Agents (agent\_id, description, games\_won, games\_played) VALUES ('asdf', 'adsfdf', 0, 0);")
311. **for** x **in** range(2):
312. ct\_w.inc\_agent\_win("asdf")
313. connection, cursor = ct\_w.execute\_queries("SELECT \* FROM Agents", keep\_open=True)
314. **for** i **in** cursor:
315. **print**(i)
316. connection.close()
317. **else**:
318. db\_dir\_path = "DB/"

### DB\_Wrapper.py

1. **import** sqlite3 as sq3
2. **import** os, sys
3. sys.path.append(os.path.realpath(".."))
5. """
6. - parent/abstract class for wrapper to database
7. - provides basic interface for interacting with the database
8. """
10. **class** DB\_Wrapper:
11. **def** \_\_init\_\_(self, db\_path):
12. self.db\_path = db\_path#"DB/"+ db\_path
14. # connects to database
15. # returns connection, cursor
16. **def** connect\_to\_db(self, path=None):
17. **if** path **is** None:
18. path = self.db\_path
19. connection = sq3.connect(path)
20. cursor = connection.cursor()
21. **return** connection, cursor
23. # pass sql file, will execute queries in file
24. # accepted sql format:
25. # newline between query
26. **def** execute\_queries\_from\_file(self, sql\_file\_path):
27. # append file extension if not already included - defensive programming
28. **if** sql\_file\_path[-4:] != ".sql":
29. sql\_file\_path += ".sql"
30. # open sql file, read and execute queries
31. queries = self.read\_queries\_from\_file(sql\_file\_path)
32. self.execute\_queries(queries)
34. # opens file path, returns list of ;-separated queries
35. # TODO add some query sanitation?
36. **def** read\_queries\_from\_file(self, sql\_file\_path):
37. queries = []
38. with open(sql\_file\_path, "r") as sql\_file\_path:
39. query = ""
40. **for** char **in** sql\_file\_path.read():
41. query += char
42. **if** char == ";":
43. queries.append(query)
44. query = ""
45. **return** queries
47. # returns true if query is safe
48. # add checks for illegal symbols
49. **def** sanitize\_query(self, query):
50. **if** "drop" **in** query:
51. **return** False
52. **else**:
53. **return** True
55. # execute passed query or passed array of queries
56. # keep open determines if connection remains open, if so, returns connection and cursor
57. # get result will return all the results of the queries
58. **def** execute\_queries(self, queries, keep\_open=False, get\_result=False):
59. # turns single query into executable form - defensive programming
60. **if** isinstance(queries, str):
61. queries = [queries]
62. connection, cursor = self.connect\_to\_db()  # open connection
63. results = []
64. **for** index, query **in** enumerate(queries):
65. **try**:
66. # print(query)
67. cursor.execute(query)
68. **if** get\_result:
69. results.append(cursor.fetchall())
70. **except** Exception as e:
71. **print**(e)
72. **return** e
73. connection.commit()
75. **if** keep\_open:
76. **return** connection, cursor
77. **else**:
78. connection.close()
79. **if** get\_result:
80. # if there is only one set of results return that result => in the case of a single query passed
81. **if** len(results) == 1:
82. results = results[0]
83. **return** results
84. **return** True
86. # displays all the records in the passed table (pass the name)
87. **def** display\_all\_records(self, table\_name):
88. connection, cursor = self.connect\_to\_db(self.db\_path)  # open connection
89. query = "SELECT \* FROM " + table\_name
90. cursor.execute(query)
91. rows = cursor.fetchall()
92. connection.close()
93. **for** row **in** rows:
94. **print**(row)
96. **if** \_\_name\_\_ == "\_\_main\_\_":
97. db = DB("blackjack.sqlite")
98. db.execute\_queries\_from\_file("Create\_Agents\_Table")
99. db.execute\_queries\_from\_file("Populate\_Agents")
100. db.execute\_queries\_from\_file("Create\_Games\_Record")
101. db.execute\_queries('INSERT INTO "Game\_Record" (winner\_id, winning\_hand, winning\_hand\_value, num\_of\_turns) VALUES (0,"asdf", 10, 2)')
102. db.display\_all\_records("Agents")

### Users\_DB.py

1. **from** DB\_Wrapper **import** DB\_Wrapper
2. **import** hashlib
3. **from** uuid **import** uuid4
4. **from** os **import** remove
6. """
7. - wrapper for users table in the database
8. - password requirement checking
9. - unique username checking
10. - sanitization -> should be built into db parent class
11. """
13. **class** Users\_DB(DB\_Wrapper):
14. **def** \_\_init\_\_(self, db\_path):
15. super().\_\_init\_\_(db\_path)
16. self.init\_table()
18. **def** init\_table(self):
19. self.execute\_queries\_from\_file("DB/Create\_Users\_Table.sql")
21. # checks if passed username is unique
22. # true => unique, false => not unique
23. **def** check\_unique\_username(self, username):
24. query = "SELECT username FROM users WHERE username='{0}'".format(username)
25. query\_result = self.execute\_queries(query, get\_result=True)
27. # if no value is returned from this query, the username is unique
28. **if** query\_result == []:
29. **return** True
30. **else**:
31. **return** False
33. # password must have at least one capital letter and one number
34. # returns true if the passed password is acceptable
35. **def** check\_acceptable\_password(self, password):
36. has\_number = False
37. has\_capital = False
38. ind = 0
39. pword\_len = len(password)  # so it does not have to be recalculated
40. **while** **not** (has\_number **and** has\_capital) **and** (ind < pword\_len):
41. unicode\_num = ord(password[ind])
42. # checks if the character is a capital
43. **if** 65 <= unicode\_num <= 90:
44. has\_capital = True
45. # checks if the character is a number
46. **elif** 48 <= unicode\_num <= 57:
47. has\_number = True
48. ind += 1
49. **return** has\_capital **and** has\_number
51. # hashes a passed password - currently returns the hex version of the hashing
52. # uses sha256 hashing algorithm
53. **def** hash\_password(self, password):
54. salt = uuid4().hex # random data to increase protection on password
55. to\_hash = salt.encode() + password.encode() # encode converts string to bytes, so it can be encoded
56. **return** hashlib.sha256(to\_hash).hexdigest() + ":" + salt
58. # confirms if the passed plaintext password and the hashed password are equivalent
59. **def** verify\_password(self, password, hashed\_password):
60. half\_hashed\_password, salt = hashed\_password.split(":")
61. user\_passed\_hashed = hashlib.sha256(salt.encode() + password.encode()).hexdigest()
62. **return** user\_passed\_hashed == half\_hashed\_password
64. # creates a new user and inserts them into the database
65. # returns false if the password or username is not valid
66. # returns true if successful insertion
67. **def** create\_new\_user(self, username, password, type="user"):
68. unique\_username = self.check\_unique\_username(username)
69. acceptable\_password = self.check\_acceptable\_password(password)
70. **if** **not** unique\_username **or** **not** acceptable\_password:
71. **return** False
72. hashed\_password = str(self.hash\_password(password))
73. query = 'INSERT INTO users (username, password, type) VALUES ("{0}", "{1}", "{2}")'.format(username,
74. hashed\_password,
75. type)
76. self.execute\_queries(query)
77. **return** True
79. # fetch the games won and the games played for a passed id
80. **def** get\_user\_game\_data(self, username):
81. query = "SELECT games\_won, games\_played FROM users WHERE username='{0}'".format(username)
82. result = self.execute\_queries(query, get\_result=True)
83. **return** result
85. # pass a username and password
86. # returns true if the username and password is correct for a login
87. **def** check\_login(self, username, password):
88. # get record from db via username
89. # checkpassword from record,
90. # if username does not exist or password is incorrect, return false
91. query = "SELECT username, password FROM users WHERE username='{0}'".format(username)
92. result = self.execute\_queries(query, get\_result=True)
93. **if** result == []:
94. **return** False
95. result = result[0]
96. hased\_pw = result[1]
97. **return** (self.verify\_password(password, hased\_pw))
99. # pass in username and password and get back the user type
100. **def** get\_user\_type(self, username, password):
101. **if** self.check\_login(username, password) == False:
102. **return** False
103. q = """
104. SELECT type
105. FROM users
106. WHERE username='{0}'
107. """.format(username)
108. res = self.execute\_queries(q, get\_result=True)
109. **return** res[0][0]
111. **if** \_\_name\_\_ == "\_\_main\_\_":
112. u\_name = "SwaggyShaggy999"
113. p\_word = "Adlfkjgklf3"
115. u\_db\_wrapper = Users\_DB("test\_db.sqlite")
116. u\_db\_wrapper.execute\_queries\_from\_file("Create\_Users\_Table")
117. u\_db\_wrapper.execute\_queries("SELECT \* FROM users WHERE username='SwaggyShaggy99'")
118. **print**("acceptable pword", u\_db\_wrapper.check\_acceptable\_password(p\_word))
119. **print**("insertion", u\_db\_wrapper.create\_new\_user(u\_name, p\_word))
120. **print**("unique uname after insertion", u\_db\_wrapper.check\_unique\_username(u\_name))
121. u\_db\_wrapper.display\_all\_records("users")
122. **print**(u\_db\_wrapper.check\_login(u\_name, p\_word))
123. **print**(u\_db\_wrapper.get\_user\_type(u\_name, p\_word))
124. # remove("blackjack.sqlite")

## /NN\_AI

### Experience\_buffer.py

1. **import** numpy as np
2. **import** random
4. # This class allows the network to draw from a batch of experiences, rather than just one at a time
5. # credit - https://medium.com/@awjuliani/simple-reinforcement-learning-with-tensorflow-part-4-deep-q-networks-and-beyond-8438a3e2b8df
6. **class** experience\_buffer():
7. # when buffer reaches max capacity, it drops all old experinces
8. **def** \_\_init\_\_(self, buffer\_size=10000):
9. self.buffer = []
10. self.buffer\_size = buffer\_size
12. **def** add(self, experience):
13. **if** len(self.buffer) + 1 >= self.buffer\_size:
14. self.buffer[0:(1 + len(self.buffer)) - self.buffer\_size] = []
15. self.buffer.append(experience)
17. **def** sample(self, batch\_size, trace\_length):
18. sampled\_episodes = random.sample(self.buffer, batch\_size)
19. sampledTraces = []
20. **for** episode **in** sampled\_episodes:
21. point = np.random.randint(0, len(episode) + 1 - trace\_length)
22. sampledTraces.append(episode[point:point + trace\_length])
23. sampledTraces = np.array(sampledTraces)
24. **return** np.reshape(sampledTraces, [batch\_size \* trace\_length, 5])

### NN\_Move.py

1. **import** numpy as np
2. **import** sys,os
3. sys.path.append(os.path.realpath(".."))
5. """
6. - static class which defines different exploration strategies for the NN
7. """
9. **class** NN\_Move:
10. @staticmethod
11. **def** choose\_action(parameters, Primary\_Network, game\_state, rnn\_state, sess, exploring=False):
12. policy = parameters["policy"]
13. **if** policy == "e-greedy":
14. move = NN\_Move.choose\_action\_e\_greedy(parameters, Primary\_Network, game\_state, rnn\_state, sess, exploring=exploring)
15. **return** move
17. # e-greedy policy => choose highest value action, with chance e of picking a random action
18. # epsilon starts high during training and is decremented over time
19. @staticmethod
20. **def** choose\_action\_e\_greedy(parameters, Primary\_Network, game\_state, rnn\_state, sess, exploring):
21. e = parameters["epsilon"]
23. # random exploration if explore stage or prob is less than epsilon
24. **if** np.random.rand(1) < e **or** exploring:
25. end\_range = parameters["no\_actions"]
26. a = np.random.randint(0, end\_range)
27. **else**:
28. a, new\_rnn\_state = sess.run([Primary\_Network.predict, Primary\_Network.rnn\_state],
29. feed\_dict={
30. Primary\_Network.input\_layer: [game\_state],
31. Primary\_Network.trainLength: 1,
32. Primary\_Network.state\_in: rnn\_state,
33. Primary\_Network.batch\_size: 1}
34. )
35. a = a[0]
37. # decrement epsilon
38. **if** **not** exploring:
39. **if** parameters["epsilon"] > parameters["end\_epsilon"]:
40. parameters["epsilon"] -= parameters["epsilon\_step"]
42. **return** a

### NN.py

1. **import** tensorflow as tf
2. **import** tensorflow.contrib.slim as slim
3. **import** numpy as np
4. **import** matplotlib.pyplot as plt
5. **import** sys,os
6. sys.path.append(os.path.realpath(".."))
7. sys.path.append(os.path.realpath("../Structs"))
8. sys.path.append(os.path.realpath("../DB"))
9. **from** CC\_Agent **import** CC\_Agent
10. **from** Blackjack **import** Hand
11. **from** NN\_Move **import** NN\_Move
12. **from** Moves **import** Moves
13. **from** Trainer **import** Init\_Trainer
14. **from** Trainer **import** Batch\_Trainer
15. **from** datetime **import** datetime
17. """
18. - defines classes for both the primary netowrk and the target network
19. - also defines the behaviour for the NN agent
20. """
22. # class for the primary network
23. **class** Q\_Net():
24. **def** \_\_init\_\_(self, input\_size, hidden\_size, output\_size, rnn\_cell, myScope, training=True):
25. self.init\_feed\_forward(input\_size, hidden\_size, output\_size, myScope)
26. self.rnn\_processing(rnn\_cell, hidden\_size, myScope)
27. self.split\_streams(hidden\_size, output\_size)
28. self.predict()
29. self.gen\_loss(output\_size)
30. self.train\_update()
32. **if** **not** training:
33. self.disable\_dropout()
35. # dropout layers should be disabled when training is complete
36. **def** disable\_dropout(self):
37. **for** dropout **in** self.dropout\_layers:
38. dropout.is\_Training = False
40. **def** rnn\_processing(self, rnn\_cell, hidden\_size, myScope):
41. # Take the output from the final fully connected layer and send it to a recurrent layer.
42. # The input must be reshaped into [batch x trace x units] for rnn processing,
43. # and then returned to [batch x units] when sent through the upper levles.
44. self.trainLength = tf.placeholder(dtype=tf.int32)
45. self.batch\_size = tf.placeholder(dtype=tf.int32, shape=[])
46. output\_flat = tf.reshape(self.final\_hidden, [self.batch\_size, self.trainLength, hidden\_size])
47. self.state\_in = rnn\_cell.zero\_state(self.batch\_size, tf.float32)
48. self.rnn, self.rnn\_state = tf.nn.dynamic\_rnn(
49. inputs=output\_flat, cell=rnn\_cell, dtype=tf.float32, initial\_state=self.state\_in, scope=myScope + '\_rnn')
50. self.rnn = tf.reshape(self.rnn, shape=[-1, hidden\_size])
52. **def** init\_feed\_forward(self, inp\_size, hidden\_size, output\_size, myScope):
53. # Establish feed-forward part of the network
54. self.input\_layer = tf.placeholder(shape=[None, inp\_size], dtype=tf.float32)
56. hidden\_layer1 = slim.fully\_connected(self.input\_layer, hidden\_size,
57. biases\_initializer=None,
58. activation\_fn=tf.nn.relu,
59. scope=(myScope+"\_hidden1"))  # Rectified linear activation func.
61. dropout1 = slim.dropout(hidden\_layer1, scope=myScope+"\_dropout1")
63. hidden\_layer2 = slim.fully\_connected(dropout1, hidden\_size,
64. biases\_initializer=None,
65. activation\_fn=tf.nn.relu,
66. scope=(myScope+"\_hidden2"))
68. dropout2 = slim.dropout(hidden\_layer2, scope=myScope+"\_dropout2")
70. self.final\_hidden = slim.fully\_connected(dropout2, hidden\_size,
71. activation\_fn=tf.nn.relu,
72. biases\_initializer=None,
73. scope=(myScope+"\_final\_hidden"))  # Softmax activation func. -> changed to relu, as no longer output
75. self.dropout\_layers = [dropout1, dropout2]
77. #self.action = tf.argmax(self.output\_layer, 1)
79. **def** split\_streams(self, hidden\_size, output\_size):
80. # The output from the recurrent player is then split into separate Value and Advantage streams
81. self.streamA, self.streamV = tf.split(self.rnn, 2, 1)
82. self.AW = tf.Variable(tf.random\_normal([hidden\_size // 2, output\_size]))
83. self.VW = tf.Variable(tf.random\_normal([hidden\_size // 2, 1]))
84. self.Advantage = tf.matmul(self.streamA, self.AW)
85. self.Value = tf.matmul(self.streamV, self.VW)
86. self.salience = tf.gradients(self.Advantage, self.input\_layer) #self.imageIn
88. **def** predict(self):
89. # Combine streams for final prediction value
90. self.Qout = self.Value + tf.subtract(self.Advantage, tf.reduce\_mean(self.Advantage, axis=1, keep\_dims=True))
91. self.predict = tf.argmax(self.Qout, 1)
93. **def** gen\_loss(self, output\_size):
94. # Loss calculated by taking the sum of squares difference between the target and prediction Q values.
95. self.targetQ = tf.placeholder(shape=[None], dtype=tf.float32)
96. self.actions = tf.placeholder(shape=[None], dtype=tf.int32)
97. self.actions\_onehot = tf.one\_hot(self.actions, output\_size, dtype=tf.float32)
99. self.Q = tf.reduce\_sum(tf.multiply(self.Qout, self.actions\_onehot), axis=1)
101. self.td\_error = tf.square(self.targetQ - self.Q)
103. # In order to only propogate accurate gradients through the network, we will mask the first
104. # half of the losses for each trace as per Lample & Chatlot 2016
105. self.maskA = tf.zeros([self.batch\_size, self.trainLength // 2])
106. self.maskB = tf.ones([self.batch\_size, self.trainLength // 2])
107. self.mask = tf.concat([self.maskA, self.maskB], 1)
108. self.mask = tf.reshape(self.mask, [-1])
109. self.loss = tf.reduce\_mean(self.td\_error \* self.mask)
111. # update the network by using Adam optimizer algorithm
112. **def** train\_update(self):
113. self.trainer = tf.train.AdamOptimizer(learning\_rate=0.0001)
114. self.updateModel = self.trainer.minimize(self.loss)
116. # target network is essentially the same as the primary network, but needs
117. # a few extra behaviours defined to sync it back with the primary network when required
118. **class** Target\_Net(Q\_Net):
119. **def** \_\_init\_\_(self, input\_size, hidden\_size, output\_size, rnn\_cell, myScope, training=True):
120. super().\_\_init\_\_(input\_size, hidden\_size, output\_size, rnn\_cell, myScope, training)
121. self.ops = None
123. #These functions update the parameters of our target network with those of the primary network.
124. **def** updateTargetGraph(self, tfVars, tau):
125. total\_vars = len(tfVars)
126. op\_holder = []
127. **for** idx,var **in** enumerate(tfVars[0:total\_vars//2]):
128. op\_holder.append(tfVars[idx+total\_vars//2].assign((var.value()\*tau) + ((1-tau)\*tfVars[idx+total\_vars//2].value())))
129. self.ops = op\_holder
130. **return** op\_holder
132. **def** updateTarget(self, sess):
133. op\_holder = self.ops
134. **for** op **in** op\_holder:
135. sess.run(op)
136. total\_vars = len(tf.trainable\_variables())
137. a = tf.trainable\_variables()[0].eval(session=sess)
138. b = tf.trainable\_variables()[total\_vars//2].eval(session=sess)
139. **if** a.all() != b.all():
140. **print**("Target Set Failed")

143. # Class for NN agent
144. **class** NN(CC\_Agent):
145. **def** \_\_init\_\_(self, setting="default", hand=None, restore\_type="default", Training=True, auto\_load=True):
146. super().\_\_init\_\_(ID="nn", extra\_type=["nn"])
147. self.rnn\_state = None
148. self.sess = None
150. self.parameters = self.set\_parameters(setting=setting)
151. self.train\_params = self.set\_training\_params(setting=setting)
153. **if** hand **is** None:
154. self.Hand = Hand(self.ID)
156. self.initalise\_NN(Training)
157. self.start\_session()
159. **if** auto\_load:
160. self.load\_model(restore\_type)
162. # sets the bevhiour parameters of the nn
163. **def** set\_parameters(self, setting="default"):
164. **if** setting **is** "default":
165. #Setting the training parameters
166. nn\_params = {
167. "gamma": .99,  # Discount factor on the target Q-values
168. "start\_epsilon" : 1, #Starting chance of random action
169. "epsilon" : 1, # tracking value for epsilon - MAKE THIS AN ATTRIBUTE?
170. "end\_epsilon" : 0.001, #Final chance of random action
171. "path\_default" : "./nn\_data", #The path to save our model to.
172. "hidden\_size" : 32, #The size of the final convolutional layer before splitting it into Advantage and Value streams.
173. "no\_features" : 6, # How many features to input into the network
174. "no\_actions" : 2, # No actions the network can take
175. "summaryLength" : 100, #Number of epidoes to periodically save for analysis -- not applicable?
176. "annealing\_steps": 1000,  # How many steps of training to reduce startE to endE.
177. "tau" : 0.001,
178. "policy" : "e-greedy",
179. "hand\_val\_norm\_const": 1 / 30,  # value to normalise hand values by
180. "batch\_size": 8,  # How many experience traces to use for each training step.
181. "trace\_length": 2,  # How long each experience trace will be when training
182. }
184. start\_epsilon = nn\_params["start\_epsilon"]
185. end\_epsilon = nn\_params["end\_epsilon"]
186. annealing\_steps = nn\_params["annealing\_steps"]

189. # Set the rate of random action decrease.
190. nn\_params["epsilon"] = start\_epsilon
191. nn\_params["epsilon\_step"] = (start\_epsilon - end\_epsilon) / annealing\_steps
193. **return** nn\_params
195. # sets parameters used in intial training
196. **def** set\_training\_params(self, setting="default"):
197. **if** setting == "default":
198. train\_params = {
199. "batch\_size": 8,  # How many experience traces to use for each training step.
200. "trace\_length": 2,  # How long each experience trace will be when training
201. "update\_freq": 2,  # How often to perform a training step.
202. "train\_steps": 20000,  # How many episodes of game environment to train network with.
203. "explore\_steps": 1000,  # how many steps for initial explore, before training
204. "save\_model\_frequency": 50,  # how often to save the model
205. "update\_frequency": 5,  # how often to update the network weights
206. "test\_steps": 20000,  # how many iterations to test
207. "hand\_val\_norm\_const": 1 / 30  # value to normalise hand values by
208. }
210. rewards = {
211. "winReward": 3,
212. "lossCost": -3,
213. "bustCost": 0,
214. "hand\_value\_discount": 1 / 2
215. }
216. **return** train\_params

219. # sets up both the primary and target rnn
220. # initialises tf and sets the target net equal to primary net
221. **def** initalise\_NN(self, Training=True):
222. no\_features = self.parameters["no\_features"]
223. hidden\_size = self.parameters["hidden\_size"]
224. no\_actions = self.parameters["no\_actions"]
225. tau = self.parameters["tau"]
227. tf.reset\_default\_graph()
228. Primary\_rnn\_cell = tf.contrib.rnn.BasicLSTMCell(num\_units=hidden\_size, state\_is\_tuple=True)
229. Target\_rnn\_cell = tf.contrib.rnn.BasicLSTMCell(num\_units=hidden\_size, state\_is\_tuple=True)
230. self.Primary\_Network = Q\_Net(no\_features, hidden\_size, no\_actions, Primary\_rnn\_cell, 'main', Training)
231. self.Target\_Network = Target\_Net(no\_features, hidden\_size, no\_actions, Target\_rnn\_cell, 'target', Training)
232. self.rnn\_state = np.zeros([1, hidden\_size]), np.zeros([1, hidden\_size])
234. self.init = tf.global\_variables\_initializer()
235. trainables = tf.trainable\_variables()
236. self.targetOps = self.Target\_Network.updateTargetGraph(trainables, tau)
237. # saver is used for saving and restoring the nn model (the weights)
238. self.saver = tf.train.Saver() # max\_to\_keep=5
240. # resets the rnn\_state
241. **def** rnn\_state\_reset(self):
242. hidden\_size = self.parameters["hidden\_size"]
243. self.rnn\_state = np.zeros([1, hidden\_size]), np.zeros([1, hidden\_size])
245. # gets the new rnn\_state
246. **def** rnn\_state\_update(self, game\_state):
247. self.rnn\_state = self.sess.run(self.Primary\_Network.rnn\_state,
248. feed\_dict={
249. self.Primary\_Network.input\_layer: [game\_state],
250. self.Primary\_Network.trainLength: 1,
251. self.Primary\_Network.state\_in: self.rnn\_state,
252. self.Primary\_Network.batch\_size: 1}
253. )
254. **return** self.rnn\_state
256. # start the session, run the assigned training type
257. **def** init\_training(self, type="group\_all"):
258. # no context manager so that session does not have to be restarted every time a new move is needed
259. trainer = Init\_Trainer(self, training\_params=self.train\_params, training\_type=type)
260. self.start\_session()
261. self.sess.run(self.init)
262. trainer.train(self.sess)
263. self.stop\_session()
265. # runs trainer for new games NN has played in
266. **def** update\_training(self):
267. self.start\_session()
268. self.sess.run(self.init)
269. trainer = Batch\_Trainer(self, training\_params=self.train\_params)
270. trainer.train\_new\_games()
271. self.stop\_session()
273. # Override from CC\_Agent
274. # exploring relates to the inital part of training when the agent is exploring the environment and takes random actions
275. **def** get\_move(self, all\_players, exploring=False):
276. game\_state = self.get\_state(all\_players)
277. chances = self.get\_chances(game\_state)
278. move\_next = self.getNextAction(chances, game\_state, exploring=exploring)
279. **return** move\_next
281. # returns the next move of the agent
282. # side effect => updated rnn cell state after every move
283. **def** getNextAction(self, chances, game\_state, exploring=False):
284. # pass through NN model, and get the next move
285. game\_state = self.get\_features(chances, game\_state)
286. move = NN\_Move.choose\_action(self.parameters, self.Primary\_Network, game\_state, self.rnn\_state, self.sess,
287. exploring=exploring)
288. self.rnn\_state\_update(game\_state)
289. **if** move == True:
290. move = Moves.HIT
291. **elif** move == False:
292. move = Moves.STAND
293. **return** move
295. # gets the features which wil be used as the parameter for the input layer of the nn
296. # [normalise\_agent\_hand\_val, normalised\_best\_val, chances] TODO update this so that it includes win margin
297. **def** get\_features(self, chances, game\_state):
298. hand\_val\_norm\_const = self.parameters["hand\_val\_norm\_const"]
299. features = []
300. # append the hand values of agent hand and best player hand to the features array
301. # should always be [agent\_hand, best\_player\_hand]
302. **for** hand **in** game\_state:
303. **if** isinstance(hand, int):
304. hand\_val = hand
305. **else**:
306. hand\_val = hand.get\_value()
307. hand\_val\_normalised = hand\_val \* hand\_val\_norm\_const
308. features.append(hand\_val\_normalised)
309. **for** key **in** sorted(chances):
310. features.append(chances[key])
311. **return** features
313. # init the tf session - MUST BE CALLED BEFORE ANY TF ACTION OCCURS
314. **def** start\_session(self):
315. self.sess = tf.Session()
317. # stops the tf session - CALL WHEN TF WORK IS COMPLETED
318. **def** stop\_session(self):
319. self.sess.close()
321. # checkpoints the current model for later use
322. # model based-parameterisation
323. # save a new model for a new paramaterisation
324. **def** save\_model(self):
325. path = self.parameters["path\_default"]
326. # Make a path for our model to be saved in.
327. **if** **not** os.path.exists(path):
328. os.makedirs(path)
329. # eg. 28-02-2018,15-30-10
330. model\_version = datetime.now().strftime("%d-%m-%Y,%H-%M-%S")
331. model\_type = "/model-default-"
332. #(path + model\_type + model\_version + ".cptk")
333. self.saver.save(self.sess, "nn\_data/model.cptk")
335. # loads the last checkpointed model - session must have been started before model loading
336. # TODO IMPLEMENT THE RESTORATION OF DIFFERENT FILES
337. **def** load\_model(self, restore\_type="default"):
338. path = self.parameters["path\_default"]
339. ckpt = tf.train.get\_checkpoint\_state(path) # gets the checkpoint from the last checkpoint file
340. #self.saver.restore(self.sess, ckpt.model\_checkpoint\_path)
341. self.saver.restore(self.sess, "NN\_AI/nn\_data/model.cptk") #NN\_AI/nn\_data/model.cptk
343. # updates the target and the primary network - should be called after game steps reaches its train frequency
344. # exp buffer - experience\_buffer class used to store game samples in training
345. **def** update\_networks(self, exp\_buffer):
346. batch\_size = self.parameters["batch\_size"]
347. hidden\_size = self.parameters["hidden\_size"]
348. trace\_length = self.parameters["trace\_length"]
349. y = self.parameters["gamma"]
351. self.Target\_Network.updateTarget(self.sess)
352. rnn\_state\_update = (np.zeros([batch\_size, hidden\_size]), np.zeros([batch\_size, hidden\_size])) # using intial rnn state for training
353. trainBatch = exp\_buffer.sample(batch\_size, trace\_length)  # Get a random batch of experiences.
355. # Below we perform the Double-DQN update to the target Q-values
356. primary\_out = self.sess.run(self.Primary\_Network.predict, feed\_dict={
357. self.Primary\_Network.input\_layer: np.vstack(trainBatch[:, 3]),
358. self.Primary\_Network.trainLength: trace\_length,
359. self.Primary\_Network.state\_in: rnn\_state\_update,
360. self.Primary\_Network.batch\_size: batch\_size
361. })
363. target\_out = self.sess.run(self.Target\_Network.Qout, feed\_dict={
364. self.Target\_Network.input\_layer: np.vstack(trainBatch[:, 3]),
365. self.Target\_Network.trainLength: trace\_length,
366. self.Target\_Network.state\_in: rnn\_state\_update,
367. self.Target\_Network.batch\_size: batch\_size
368. })
370. end\_multiplier = -(trainBatch[:, 4] - 1)
371. doubleQ = target\_out[range(batch\_size \* trace\_length), primary\_out]
372. targetQ = trainBatch[:, 2] + (y \* doubleQ \* end\_multiplier)
373. # Update the network with our target values.
374. self.sess.run(self.Primary\_Network.updateModel,
375. feed\_dict={
376. self.Primary\_Network.input\_layer: np.vstack(trainBatch[:, 0]),
377. self.Primary\_Network.targetQ: targetQ,
378. self.Primary\_Network.actions: trainBatch[:, 1],
379. self.Primary\_Network.trainLength: trace\_length,
380. self.Primary\_Network.state\_in: rnn\_state\_update,
381. self.Primary\_Network.batch\_size: batch\_size}
382. )
384. # at the end of each game each hand is passed and the card counter is decremented
385. **def** update\_end\_game(self, new\_cards):
386. self.decrement\_CC(new\_cards)
387. self.rnn\_state\_reset()

390. **if** \_\_name\_\_ == "\_\_main\_\_":
391. nn = NN()
392. #nn.init\_training()
393. nn.start\_session()
394. nn.sess.run(nn.init)
395. nn.update\_training()
396. nn.stop\_session()

### Trainer.py

1. **import** sys,os
2. sys.path.append(os.path.realpath(".."))
3. sys.path.append(os.path.realpath("../DB"))
4. **import** Blackjack
5. **from** DB\_Wrapper **import** DB\_Wrapper
6. **from** CT\_Wrapper **import** CT\_Wrapper
7. **from** Moves **import** Moves
8. **from** CC\_AI **import** CC\_AI
9. **from** experience\_buffer **import** experience\_buffer
10. **from** datetime **import** datetime
11. **import** numpy as np
12. **import** tensorflow as tf
14. """
15. - defines the classes used to train the neural network
16. - one for initial training, with a Blackjack environment
17. - another for updating the neural network based on the games the agent has played
18. """
20. # abstract class implementing the common features of the trainers
21. **class** Trainer:
22. **def** \_\_init\_\_(self, nn\_inst, training\_params=None):
23. self.NN = nn\_inst
24. self.exp\_buffer = experience\_buffer()
25. self.parameters = training\_params
26. # todo insert some default params
27. **if** training\_params **is** None:
28. self.parameters = {}
30. # geneates the reward based on a move and the state it brought the agent into
31. # rewards define the beavhiour of the neural network as it will always attempt to maximise the reward
32. # reward for a move in game
33. **def** gen\_step\_reward(self, nn\_hand\_val\_after, move, nn\_winning):
34. agent\_value = nn\_hand\_val\_after
35. hand\_val\_norm\_const = self.parameters["hand\_val\_norm\_const"]
36. win\_value = (agent\_value + 1) \* hand\_val\_norm\_const # change this so that it is paramaterised
37. loss\_value = (-agent\_value - 1) \* hand\_val\_norm\_const
38. normal\_reward = agent\_value \* hand\_val\_norm\_const
39. scaled\_value = 0
40. **if** move == Moves.HIT:
41. **if** agent\_value > 21:
42. scaled\_value = loss\_value
43. **else**:
44. scaled\_value = normal\_reward
45. # rewards for standing
46. **elif** move == Moves.STAND:
47. **if** nn\_winning:
48. scaled\_value = normal\_reward
49. **else**:
50. scaled\_value = loss\_value
51. **return** scaled\_value
53. # generates reward for the neural network at the end of the game, based on whether or not it won
54. **def** gen\_end\_reward(self, agent\_value, nn\_in\_winners):
55. hand\_val\_norm\_const = self.parameters["hand\_val\_norm\_const"]
56. win\_value = (agent\_value + 1) \* hand\_val\_norm\_const
57. loss\_value = (-agent\_value - 1) \* hand\_val\_norm\_const
58. scaled\_value = 0
59. **if** nn\_in\_winners:
60. scaled\_value = win\_value
61. **else**:
62. scaled\_value = loss\_value
63. **return** scaled\_value
65. # training via internal blackjack environment of class
66. # class defines and runs blackjack game and tehn updates the nn based on its actions
67. **class** Init\_Trainer(Trainer):
68. **def** \_\_init\_\_(self, nn\_inst, training\_params=None, training\_type="dealer\_only"):
69. super().\_\_init\_\_(nn\_inst, training\_params)
70. self.group\_agents = {}
71. self.blackjack = self.init\_blackjack(training\_type)
73. # initialises the blackjack environment and other agents who may be part of the training
74. **def** init\_blackjack(self, training\_type):
75. nn\_hand = Blackjack.Hand(self.NN.ID)
76. self.NN.Hand = nn\_hand
77. dealer\_hand = Blackjack.Dealer\_Hand()
78. hands = {
79. self.NN.ID: nn\_hand,
80. "dealer": dealer\_hand
81. }
82. **if** training\_type == "group\_cc\_ai":
83. self.init\_CC\_AI(hands)
84. **elif** training\_type == "group\_simple":
85. self.init\_Simple\_AI(hands)
86. **elif** training\_type == "group\_all":
87. self.init\_CC\_AI(hands)
88. self.init\_Simple\_AI(hands)
90. **return** Blackjack.Blackjack(hands)
92. # lower level methods used to initialise the other agents who are playing
93. **def** init\_CC\_AI(self, hands):
94. cc\_ai\_hand = Blackjack.Hand("cc\_ai")
95. self.group\_agents["cc\_ai"] = CC\_AI(hand=cc\_ai\_hand)
96. hands["cc\_ai"] = cc\_ai\_hand
98. **def** init\_Simple\_AI(self, hands):
99. cc\_ai\_hand = Blackjack.Hand("simple")
100. self.group\_agents["simple"] = CC\_AI(hand=cc\_ai\_hand)
101. hands["simple"] = cc\_ai\_hand
103. # process agents action for one move
104. **def** process\_NN\_agent\_action(self, game\_num, all\_hands, game\_state, episode\_buffer, exp\_buffer):
105. explore\_steps = self.parameters["explore\_steps"]
106. update\_frequency = self.parameters["update\_frequency"]
107. exploring = (game\_num <= explore\_steps)
108. action = self.NN.get\_move(all\_hands, exploring)
109. self.process\_action(action)
110. new\_game\_state = self.get\_train\_game\_state(all\_hands)
111. reward = self.gen\_reward(action)
112. action = Moves.convert\_to\_bit(action)
113. # push action to buffer, for sampling later
114. episode\_buffer.append(np.reshape(np.array([game\_state, action, reward,
115. new\_game\_state, self.blackjack.continue\_game]), [1, 5]))
116. **if** game\_num % update\_frequency == 0 **and** **not** exploring:
117. self.NN.update\_networks(exp\_buffer)
118. **return** action, new\_game\_state
120. # main training loop, runs the game environment for x num of train\_step
121. # pass in the current tensorflow session
122. **def** train(self, sess):
123. train\_iterations = self.parameters["train\_steps"]
125. self.sess = sess
126. self.NN.Target\_Network.updateTarget(sess)
128. **for** game\_num **in** range(train\_iterations):
129. **print**(game\_num)
130. all\_hands = self.blackjack.get\_all\_hands()
131. episode\_buffer = []
132. game\_state = self.get\_train\_game\_state(all\_hands)
133. action = None
134. new\_game\_state = None
135. # step in game, get reward and new state
136. **while** self.blackjack.continue\_game:
137. current\_agent = self.blackjack.get\_current\_player()
138. # print(current\_agent.id)
139. **if** current\_agent.id != self.NN.ID:
140. move = self.group\_agents[current\_agent.id].get\_move(all\_hands)
141. self.process\_action(move)
142. new\_game\_state = self.get\_train\_game\_state(all\_hands)
143. **else**:  # is the nn agent's turn
144. action, new\_game\_state = self.process\_NN\_agent\_action(game\_num, all\_hands, game\_state,
145. episode\_buffer, self.exp\_buffer)
146. game\_state = new\_game\_state  # update the game state
147. # PROCESS END OF GAME
148. # GET THE END OF GAME REWARD
149. self.end\_game()
150. reward = self.gen\_reward()
151. # should never execute
152. **if** action **is** None:
153. **print**("NO MOVES EXECUTED")
154. # decide if you want to append this
155. action = Moves.convert\_to\_bit(action)
156. episode\_buffer.append(np.reshape(np.array([game\_state, action, reward,
157. new\_game\_state, self.blackjack.continue\_game]), [1, 5]))
158. # Add the episode to the experience buffer
159. bufferArray = np.array(episode\_buffer)
160. episodeBuffer = list(zip(bufferArray))
161. self.exp\_buffer.add(episodeBuffer)
162. self.reset()
163. # saves the new model after training
164. self.NN.save\_model()
166. **def** get\_train\_game\_state(self, all\_hands):
167. # [agent\_hand\_val\_normalised, best\_player\_hand\_val\_normalised, chances (in order of key )]
168. AI\_and\_best\_hand = self.NN.get\_state(all\_hands)  # should return [agent\_hand\_val, best\_player\_val]
169. chances = self.NN.get\_chances(AI\_and\_best\_hand)
170. toReturn = self.NN.get\_features(chances, AI\_and\_best\_hand)
171. **return** toReturn
173. **def** process\_action(self, action):
174. **if** action == Moves.HIT:
175. self.blackjack.hit()
176. **elif** action == Moves.STAND:
177. self.blackjack.stand()
178. **else**:
179. **print**("invalid move")
181. **def** gen\_reward(self, move=None):
182. nn\_hand = self.blackjack.players["nn"]
183. agent\_value = nn\_hand.get\_value()
184. current\_winners = self.blackjack.compare\_hands()
185. scaled\_value = 0
186. # Win and loss rewards regardless of last action - if absolute winner/loser
187. **if** move **is** None:  # end of game rewards
188. **if** self.blackjack.check\_game\_over():
189. nn\_is\_winner = nn\_hand.id **in** current\_winners
190. scaled\_value = super().gen\_end\_reward(agent\_value, nn\_is\_winner)
191. # rewards for hit and cost for bust
192. **else**:
193. nn\_winning = nn\_hand.id **in** current\_winners
194. scaled\_value = super().gen\_step\_reward(agent\_value, move, nn\_winning)
195. **return** scaled\_value
197. # decrement cc's
198. # end the blackjack game
199. **def** end\_game(self):
200. self.blackjack.end\_game()
201. new\_cards = self.blackjack.new\_cards
202. **for** key, agent **in** self.group\_agents.items():
203. **if** "Card Counter" **in** agent.type:
204. agent.decrement\_CC(new\_cards)
206. # reset rnn state
207. # reset blackjack
208. **def** reset(self):
209. self.NN.rnn\_state\_reset()  # Reset the recurrent layer's hidden state
210. self.blackjack.reset()

213. # training via querying the database
214. **class** Batch\_Trainer(Trainer):
215. **def** \_\_init\_\_(self, nn\_inst, training\_params=None):
216. super().\_\_init\_\_(nn\_inst, training\_params)
217. self.db\_wrapper = DB\_Wrapper("DB/blackjack.sqlite")
219. # gets all the games which have not been used for training yet
220. **def** pop\_new\_games(self):
221. # cross table param sql
222. # gets all the cc data for the moves the nn took part in
223. get\_q = """
224. SELECT Card\_Counter\_Record.\*
225. FROM Card\_Counter\_Record, Moves
226. WHERE Moves.player\_id='{0}' AND Card\_Counter\_Record.trained=0 AND Moves.game\_id=Card\_Counter\_Record.game\_id
227. AND Moves.turn\_num=Card\_Counter\_Record.turn\_num;
228. """.format(self.NN.ID)
230. update\_popped\_q = """
231. UPDATE Card\_Counter\_Record
232. SET trained=1
233. WHERE trained=0;
234. """
235. new\_games = self.db\_wrapper.execute\_queries(get\_q, get\_result=True)
236. #self.db\_wrapper.execute\_queries(update\_popped\_q)
237. **return** new\_games
239. # converts a single record to an array of features, train ready
240. # pass in a query result where everything from card counter record has been fetched
241. **def** get\_chances\_and\_data\_from\_rec(self, record):
242. # game\_state => [nn\_hand\_val, best\_hand\_val, ]
243. game\_id = record[0]
244. turn\_num = record[1]
245. hand\_val\_norm\_const = self.parameters["hand\_val\_norm\_const"]
246. # get best hand and best ai hand
247. q = """
248. SELECT hand\_val\_before, hand\_val\_after, next\_best\_val, move
249. FROM Moves
250. WHERE game\_id={0} AND turn\_num={1};
251. """.format(game\_id, turn\_num)
252. hand\_val\_res = self.db\_wrapper.execute\_queries(q, get\_result=True)[0]
254. # convert record to part of the features
255. chances = {
256. "bust" : record[2],
257. "blackjack": record[3],
258. "exceedWinningPlayer": record[4],
259. "alreadyExceedingWinningPlayer": record[5]
260. }
261. **return** chances, hand\_val\_res
263. # returns the number of games in store for the nn to train from
264. **def** get\_num\_games\_to\_train(self):
265. # cross param sql
266. q = """
267. SELECT COUNT(Card\_Counter\_Record.\*)
268. FROM Card\_Counter\_Record, Moves
269. WHERE Card\_Counter\_Record.trained=0 AND Moves.player\_id='{0}' AND Moves.game\_id=Card\_Counter\_Record.game\_id
270. AND Moves.turn\_num=Card\_Counter\_Record.turn\_num;
271. """.format(self.NN.ID)
272. games = self.db\_wrapper.execute\_queries(q, get\_result=True)
273. **if** games == []:
274. **return** 0
275. **else**:
276. **return** games[0][0]
278. # updates the network with the new games in the db
279. **def** train\_new\_games(self):
280. episode\_buffer = []
281. new\_moves = self.pop\_new\_games()
282. last\_game\_id = 0
283. last\_turn\_num = 0
284. nn\_wins = False
286. # iterates over the moves, end game game rewards generated based on the change in game ids
287. **for** move **in** new\_moves:
288. game\_id = move[0]
289. turn\_num = move[1]
290. chances, hand\_val\_res = self.get\_chances\_and\_data\_from\_rec(move)
291. hand\_val\_before = hand\_val\_res[0]
292. hand\_val\_after = hand\_val\_res[1]
293. next\_best\_val = hand\_val\_res[2]
294. action = hand\_val\_res[3]
295. features\_before = self.NN.get\_features(game\_state=[hand\_val\_before, next\_best\_val], chances=chances)
296. features\_after = self.NN.get\_features(game\_state=[hand\_val\_after, next\_best\_val], chances=chances)
298. # should always and only execute whenever processing new game
299. # wraps everything from episode buffer into one structure and then
300. # pushes everything into the experience buffer
301. **if** game\_id != last\_game\_id:
302. **if** last\_game\_id != 0:
303. bufferArray = np.array(episode\_buffer)
304. episodeBuffer = list(zip(bufferArray))
305. self.exp\_buffer.add(episodeBuffer)
306. episode\_buffer = []
307. last\_game\_id = int(game\_id)
308. last\_turn\_num = self.get\_last\_turn\_num(game\_id)
309. nn\_wins = self.get\_nn\_is\_winner(game\_id)
310. reward = self.gen\_end\_reward(hand\_val\_before, nn\_wins)
311. cont\_game = False
312. episode\_buffer.append(np.reshape(np.array([features\_before, action, reward,
313. features\_after, cont\_game]), [1, 5]))
315. nn\_winning = hand\_val\_after >= next\_best\_val
316. reward = self.gen\_step\_reward(hand\_val\_after, move, nn\_winning)
317. cont\_game = turn\_num == last\_turn\_num
319. episode\_buffer.append(np.reshape(np.array([features\_before, action, reward,
320. features\_after, cont\_game]), [1, 5]))
321. self.NN.update\_networks(self.exp\_buffer)
323. **def** get\_last\_turn\_num(self, game\_id):
324. q = """
325. SELECT num\_of\_turns
326. FROM Game\_Record
327. WHERE game\_id={0}
328. """.format(game\_id)
329. **return** self.db\_wrapper.execute\_queries(q, get\_result=True)[0][0]
331. # determines if the nn was part of the winners for a particular game
332. # returns boolean signifying if the nn was a winner
333. **def** get\_nn\_is\_winner(self, game\_id):
334. q = """
335. SELECT \*
336. FROM Game\_Record
337. WHERE game\_id={0} AND winner\_ids LIKE '%{1}%'
338. """.format(game\_id, self.NN.ID)
339. res = self.db\_wrapper.execute\_queries(q, get\_result=True)
340. **return** res != []

## /Structs

### Binary\_Tree.py

1. """
2. - class for binary tree
3. - functions utilising pointers within nodes to point to left and right nodes
4. - structure of the binary tree is automatically maintained so that always remains balanced
5. - binary seach tree has property of every node in left subtree is less than the root
6. - and every node in the right subtree is larger than the root
7. - balanced means that there are the same number of nodes in every subtree in the tree +- 1 node
8. """
10. **class** Binary\_Tree:
11. **def** \_\_init\_\_(self, rootNode=None):
12. self.\_root = rootNode
14. @property
15. **def** root(self):
16. **return** self.\_root
18. # CHange to DFS?
19. # pass in the value of the node you are looking for, or a equivalent to the one you are looking for
20. # returns the node, found via pre order traversals
21. **def** getNode(self, nodeValue):
22. toPass = nodeValue
23. **if** isinstance(nodeValue, Node):
24. toPass = nodeValue.value
26. **def** base\_case(node):
27. **if** node **is** None:
28. **return** None
29. **return** -1
31. **def** node\_processing(node):
32. **if** node.value == toPass:
33. **return** node
34. **return** -1
36. returnNode = Traversals.pre\_order(self.\_root, base\_case=base\_case, node\_processing=node\_processing)
37. **return** returnNode
39. # inserts node using ifs and loops
40. **def** insert(self, node):
41. **if** isinstance(node, int):
42. node = Node(node)
43. **if** self.getNode(node) **is** **not** None:
44. **return** False
45. **if** self.\_root **is** None:
46. self.\_root = node
47. **return** True
49. nextNode = self.\_root
50. nextParent = None
51. lastParentLeft = True # used to choose which side to insert
52. **while** nextNode **is** **not** None:
53. **if** node.value < nextNode.value:
54. nextParent = nextNode
55. nextNode = nextNode.left
56. lastParentLeft = True
57. **elif** node.value > nextNode.value:
58. nextParent = nextNode
59. nextNode = nextNode.right
60. lastParentLeft = False
61. **if** lastParentLeft:
62. nextParent.left = node
63. **else**:
64. nextParent.right = node
65. self.maintainTree()
67. # pass in a node or a node value, and this will return
68. # that node's parent node via pre order traversal
69. **def** getParent(self, nodeToFind):
70. **def** base\_case(node):
71. **if** node **is** None:
72. **return** None
73. **return** -1
74. **def** node\_processing(node):
75. **if** node.left == nodeToFind **or** node.right == nodeToFind:
76. **return** node
77. **return** -1
79. **return** Traversals.pre\_order(self.\_root, base\_case=base\_case, node\_processing=node\_processing)
81. # counts the number of nodes in a tree via post order traversal
82. **def** get\_tree\_size(self):
83. **def** base\_case(node):
84. **if** node **is** None:
85. **return** 0
86. **return** -1
87. **def** node\_processing(node, left, right):
88. **return** (left + right + 1)
89. **return** Traversals.post\_order(self.\_root, base\_case=base\_case, node\_processing=node\_processing)
91. # method which maintains balance within the binary tree
92. # for information on how this works check the documented design section
93. **def** maintainTree(self):
94. **def** base\_case(node):
95. **if** node **is** None:
96. **return** 0
97. **return** -1
99. **def** node\_processing(node, left, right):
100. **if** left == -1 **or** right == -1:
101. **return** -1
102. **elif** abs(left - right) >= 2:
103. **if** left > right:
104. self.swap\_max\_LST(node)
105. **else**:
106. self.swap\_min\_RST(node)
107. **return** -1
108. **return** left + right + 1
110. # unknown number of maintainance actions becuase every time a maintainance occurs
111. # the maintainance must start again from the bast of the tree, because one adjustment
112. # to one subtree may affect other trees
113. completed\_comparing = -1
114. **while** completed\_comparing == -1:
115. completed\_comparing = Traversals.post\_order(self.\_root, base\_case=base\_case, node\_processing=node\_processing)
117. # pass in the root to a subtree, places the maximum node in the left subtree
118. # as the new root, and then puts the old root as the first node in new right subtree
119. **def** swap\_max\_LST(self, swapRoot):
120. """
121. replace the parent with the max in LST
122. - newRoot.right = oldRoot (oldRoot keeps left subtree)
123. - if newRoot is direct parent
124. return False to start again
125. """
126. max\_LST = self.get\_max\_LST(swapRoot)
127. parent = self.getParent(swapRoot)
129. **if** swapRoot.left != max\_LST:
130. self.delete(max\_LST) # one child guarenteed -> one call
131. max\_LST.left = swapRoot.left
132. swapRoot.left = None # Will this break the subtree?
133. max\_LST.right = swapRoot
135. **if** parent == None:
136. self.\_root = max\_LST
137. **elif** parent.left == swapRoot:
138. parent.left = max\_LST
139. **elif** parent.right == swapRoot:
140. parent.right = max\_LST
142. # pass in the root to a subtree, places the minimum node in the right subtree
143. # as the new root, and then puts the old root as the first node in the new left subtree
144. **def** swap\_min\_RST(self, swapRoot):
145. """
146. swap nodes:
147. replace parent with minRST
148. - newRoot.left <- oldRoot
149. - if directChild then keep RST
150. - else then newRoot.right <- OldRoot.Right
151. return False to start again
152. """
153. # get -> delete -> swap
154. min\_RST = self.get\_min\_RST(swapRoot)
155. parent = self.getParent(swapRoot)
157. # if direct child, it keeps its old RST, and is not deleted before swapping
158. **if** swapRoot.right != min\_RST:
159. self.delete(min\_RST)
160. min\_RST.right = swapRoot.right
161. swapRoot.right = None # Will this break the subtree?
162. min\_RST.left = swapRoot
164. **if** parent == None:
165. self.\_root = min\_RST
166. **elif** parent.right == swapRoot:
167. parent.right = min\_RST
168. **elif** parent.left == swapRoot:
169. parent.left = min\_RST
171. # gets the maximum node in the left subtree of the passed node
172. **def** get\_max\_LST(self, root):
173. current\_node = root.left
174. **while** current\_node.right **is** **not** None:
175. current\_node = current\_node.right
176. # current\_node.left, current\_node.right = None, None
177. **return** current\_node
179. # gets the minimum node in the right subtree of the passed node
180. **def** get\_min\_RST(self, root):
181. current\_node = root.right
182. **while** current\_node.left **is** **not** None:
183. current\_node = current\_node.left
184. **return** current\_node
186. # deletes a passed node
187. # DO NOT MAINTAIN WITHIN THIS METHOD
188. # 3 cases - no children, 1 child, 2 children or deleting root
189. **def** delete(self, node):
190. numChildren = node.numOfChildren()
191. **if** node == self.\_root:
192. self.delete\_root()
193. **else**:
194. nodeParent = self.getParent(node)
195. nodeIsLeft = nodeParent.left == node
196. **if** numChildren == 0:
197. self.delete\_noChildren(node, nodeParent, nodeIsLeft)
198. **elif** numChildren == 1:
199. self.delete\_oneChild(node, nodeParent, nodeIsLeft)
200. **elif** numChildren == 2:
201. self.delete\_twoChildren(node, nodeParent, nodeIsLeft)
203. # lower level fuction for deleting with no children
204. **def** delete\_noChildren(self, node, nodeParent, nodeIsLeft=None):
205. **if** nodeIsLeft **is** None:
206. nodeIsLeft = nodeParent.left == node
207. **if** nodeIsLeft:
208. nodeParent.left = None
209. **else**:
210. nodeParent.right = None
212. # lower level fuction for deleting with one child
213. **def** delete\_oneChild(self, node, nodeParent, nodeIsLeft=None):
214. **if** nodeIsLeft **is** None:
215. nodeIsLeft = nodeParent.left == node
216. childIsLeft = node.hasLeft()
218. **if** nodeIsLeft:
219. **if** childIsLeft:
220. nodeParent.left = node.left
221. **else**:
222. nodeParent.left = node.right
223. **else**:
224. **if** childIsLeft:
225. nodeParent.right = node.left
226. **else**:
227. nodeParent.right = node.right
229. # lower level fuction for deleting with two children
230. **def** delete\_twoChildren(self, node, nodeParent, nodeIsLeft=None):
231. **if** nodeIsLeft **is** None:
232. nodeIsLeft = nodeParent.left == node
233. # Find max node in left subtree and replace the node to delete with it
234. max\_LST = self.get\_max\_LST(node)
236. # Recursively delete the old node from its old position, since it is the max node in left tree, it cannot
237. # have a right child, therefore this will be called recursively once at max (one child MAX).
238. self.delete(max\_LST)
239. max\_LST.right = node.right
240. max\_LST.left = node.left
241. **if** nodeIsLeft:
242. nodeParent.left = max\_LST
243. **else**:
244. nodeParent.right = max\_LST
246. # special case when deleting a root, because the principle of everything in lst must be smaller
247. # than root and everything in rst must be bigger than root must be maintained.
248. **def** delete\_root(self):
249. **if** self.\_root.left **is** None **and** self.\_root.right **is** None:
250. self.clearTree()
251. **else**:
252. **if** self.\_root.left **is** **not** None:
253. swapNode = self.get\_max\_LST(self.\_root)
254. **else**:
255. swapNode = self.get\_min\_RST(self.\_root)
256. self.delete(swapNode)
257. swapNode.left = self.\_root.left
258. swapNode.right = self.\_root.right
259. self.\_root = swapNode
261. # For a binary search tree, this should be in ascending order.
262. **def** output\_tree\_console(self, parent): # Always pass in the root node with initial call.
263. **if** parent == None:
264. **return** False
265. self.in\_order\_traversal(parent.left)
266. **print**(parent.value)
267. self.in\_order\_traversal(parent.right)
269. **def** clearTree(self):
270. self.\_root = None
272. # gets the smallest node in the tree
273. **def** get\_min\_node(self, start\_node=False):
274. **if** start\_node == False:
275. start\_node = self.\_root
276. node = start\_node
277. **while** node.left **is** **not** None:
278. node = node.left
279. **return** node
281. # utility static class for abstracting away the traversals
282. # pass in lexically scoped function for the base case (reaching a None node)
283. # and another function for the node processing
284. **class** Traversals:
285. # Static higher level function for pre order traversals
286. @staticmethod
287. **def** pre\_order(root, base\_case, node\_processing):
288. base\_result = base\_case(root)
289. **if** base\_result != -1:  # need another base value (None, and False cannot be used) (maybe a false base object?)
290. **return** base\_result
291. processing\_result = node\_processing(root)
292. **if** processing\_result != -1:
293. **return** processing\_result
295. left = Traversals.pre\_order(root.left, base\_case, node\_processing)
296. right = Traversals.pre\_order(root.right, base\_case, node\_processing)
298. **if** left **is** **not** None:
299. **return** left
300. **elif** right **is** **not** None:
301. **return** right
303. @staticmethod
304. # never used so not implemented
305. **def** in\_order(self):
306. **pass**
308. @staticmethod
309. **def** post\_order(root, base\_case, node\_processing):
310. base\_result = base\_case(root)
311. **if** base\_result != -1:
312. **return** base\_result
314. left = Traversals.post\_order(root.left, base\_case, node\_processing)
315. right = Traversals.post\_order(root.right, base\_case, node\_processing)
317. processing\_result = node\_processing(root, left, right)
318. **return** processing\_result

321. # Class for the node
322. # main properties - pointers to left and right nodes
323. # defines utility behaviours, such as counting children and comparison methods
324. **class** Node: # Association via composition
325. **def** \_\_init\_\_(self, value):
326. self.value = value
327. self.left = None
328. self.right = None
330. **def** hasLeft(self):
331. **return** (**not** self.left == None)
333. **def** hasRight(self):
334. **return** (**not** self.right == None)
336. **def** hasChildren(self):
337. **return** (**not** self.left == None) **or** (**not** self.right == None)
339. **def** numOfChildren(self):
340. **return** self.hasLeft() + self.hasRight()
342. **def** \_\_eq\_\_(self, other):
343. **if** isinstance(other, Node):
344. **return** self.value == other.value
345. **elif** isinstance(other, int):
346. **return** self.value == other
347. **return** False
349. **def** \_\_str\_\_(self):
350. **return** str(self.value)
352. **def** \_\_gt\_\_(self, other):
353. **if** isinstance(other, Node):
354. **return** self.value > other.value
355. **elif** isinstance(other, int):
356. **return** self.value > other
357. **return** None
359. **def** \_\_lt\_\_(self, other):
360. **if** isinstance(other, Node):
361. **return** self.value < other.value
362. **elif** isinstance(other, int):
363. **return** self.value > other
364. **return** None
366. **def** \_\_ge\_\_(self, other):
367. **if** isinstance(other, Node):
368. **return** self.value >= other.value
369. **elif** isinstance(other, int):
370. **return** self.value >= other
371. **return** None
373. **def** \_\_le\_\_(self, other):
374. **if** isinstance(other, Node):
375. **return** self.value <= other.value
376. **elif** isinstance(other, int):
377. **return** self.value >= other
378. **return** None

381. # Testing the functionality
382. **if** \_\_name\_\_ == "\_\_main\_\_":
383. node\_value = 1
384. insertion\_arr = [2,3,4,5,6,7,8,9,10,11]
385. b = Binary\_Tree( Node(node\_value) )
386. **for** num **in** insertion\_arr:
387. **print**(num)
388. b.insert( Node(num) )
389. b.in\_order\_traversal(b.root)
391. #print(b.getNode(3))
393. **print**("6:", b.root)
394. **print**("3:", b.root.left)
395. **print**("9:", b.root.right)
397. #print(b.root.right.left)
398. #b.maintainTree()
399. #print(b.root.left.right)

402. #print(b.get\_max\_LST(b.root))
403. #print(b.get\_min\_RST(b.root))
404. #print(b.get\_max\_LST(b.root))
406. ## print(b.getParent(8))

409. #b.testfunc()
410. #b.display\_tree\_structure()
412. #   b.in\_order\_traversal(b.root)
414. #b.decrement(Card\_Node(3, 0))
415. #print()
416. #b.in\_order\_traversal(b.root)
417. #print(b.root.left.countValue)

### Card\_Binary\_Tree.py

1. **from** Binary\_Tree **import** Binary\_Tree
2. **from** Binary\_Tree **import** Node
3. **from** Binary\_Tree **import** Traversals
5. # Child class for Card Binary tree
6. **class** Card\_Binary\_Tree(Binary\_Tree):
7. **def** \_\_init\_\_(self, rootNode=None):
8. super().\_\_init\_\_(rootNode)
10. # reduces the card count value of a particular card, if it reaches 0 then that node is deleted
11. **def** decrement(self, nodeValue):
12. **if** nodeValue == None:
13. **return** False
14. **elif** isinstance(nodeValue, Node): # defensive programming?
15. nodeValue = nodeValue.value
17. # if node does not exist in the tree
18. node\_to\_dec = self.getNode(nodeValue)
19. **if** node\_to\_dec == None:
20. **return** False
22. **elif** node\_to\_dec.value == nodeValue:
23. node\_to\_dec.countValue -= 1
24. **if** node\_to\_dec.countValue == 0:
25. self.delete(node\_to\_dec)
26. self.maintainTree()
27. **return** True
29. # traverse the entire tree and only add to the node processing if bigger than or greater than the node passed
30. # pass in node, returns the sum of the count values of the nodes which have a value
31. # greater than or equal to the passed node
32. **def** cardCountGTET(self, baseNode=False):
33. **if** baseNode == False:
34. baseNode = self.\_root
35. minNode = self.get\_min\_node()
36. **if** minNode == baseNode:
37. **return** self.totalCardCount()
38. **def** base\_case(node):
39. **if** node **is** None:
40. **return** 0
41. **return** -1
42. **def** node\_processing(node, left, right):
43. toAdd = 0
44. **if** node >= baseNode:
45. toAdd = node.countValue
46. **return** toAdd + left + right
47. **return** Traversals.post\_order(self.\_root, base\_case=base\_case, node\_processing=node\_processing)
49. # COUNT NUM NODES IN TREe -> OTHER METHOD
50. # Post order traversal to count number of CARDS in a tree
51. # sum of each node's count values
52. **def** totalCardCount(self, parent=False):
53. **if** parent == False:
54. parent = self.\_root
55. **def** base\_case(node):
56. **if** node == None:
57. **return** 0
58. **return** -1
59. **def** node\_processing(node, left, right):
60. **return** node.countValue + left + right
61. tree\_count = Traversals.post\_order(parent, base\_case=base\_case, node\_processing=node\_processing)
62. ace\_node = self.getNode(11) # could put this higher up to simplify this logical block, but less efficient as unecessary counting
63. **if** ace\_node **is** **not** None:
64. tree\_count = tree\_count - self.getNode(11).countValue # prevents counting the ace twice
65. **return** tree\_count
67. # prints count values as well as node value
68. **def** output\_tree\_console(self, parent):
69. **if** parent == None:
70. **return** False
71. self.in\_order\_traversal(parent.left)
72. **print**(parent.value, parent.countValue)
73. self.in\_order\_traversal(parent.right)
75. # child class of node.
76. # implements additional attribute of countValue
77. # in this context the corressponding number of cards associated with a card value
78. # left in the tree
79. **class** Card\_Node(Node):
80. **def** \_\_init\_\_(self, value, countValue):
81. super().\_\_init\_\_(value)
82. self.countValue = countValue

### Circular\_Queue.py

1. """
2. - circular queue structure
3. - FIFO logic
4. - additional property tracking size of queue
5. """
7. **class** Circular\_Queue:
8. **def** \_\_init\_\_(self, size):
9. self.\_\_size = size
10. self.\_\_circQ = [None **for** \_ **in** range(self.\_\_size)]
11. self.\_\_front = -1
12. self.\_\_rear = -1
13. self.\_\_num\_elements = 0
15. @property
16. **def** size(self):
17. **return** self.\_\_size
19. @property
20. **def** num\_elements(self):
21. **return** self.\_\_num\_elements
23. **def** push(self, toPush):
24. **if** self.isFull():
25. **return** False
26. **if** self.isEmpty():
27. self.\_\_front += 1
29. # if rear has reached end of queue, circle it to front of array
30. **if** self.\_\_rear == (self.\_\_size - 1):
31. self.\_\_rear = 0
32. **else**:
33. self.\_\_rear += 1
35. self.\_\_circQ[self.\_\_rear] = toPush
36. self.\_\_num\_elements += 1
37. **return** True
39. **def** pop(self):
40. **if** self.isEmpty():
41. **return** False
42. toReturn = self.\_\_circQ[self.\_\_front]
43. self.\_\_circQ[self.\_\_front] = None
45. # if last ele popped
46. **if** self.\_\_front == self.\_\_rear:
47. self.\_\_front, self.\_\_rear = -1, -1
48. # if front pointer has reached front of queue, circle it
49. **elif** self.\_\_front == (self.\_\_size - 1):
50. self.\_\_front = 0
51. **else**:
52. self.\_\_front += 1
53. self.\_\_num\_elements -= 1
54. **return** toReturn
56. **def** peek(self):
57. **return** self.\_\_circQ[self.\_\_front]
59. **def** isFull(self):
60. noPopCond = self.\_\_front == 0 **and** self.\_\_rear == (self.\_\_size -1)
61. generalCond = self.\_\_front == (self.\_\_rear + 1)
62. **return** noPopCond **or** generalCond
64. **def** isEmpty(self):
65. **return** self.\_\_front == -1 **and** self.\_\_rear == -1
67. **if** \_\_name\_\_ == "\_\_main\_\_":
68. q = Circular\_Queue(2)
70. **for** x **in** range(5):
71. **print**()
72. **for** i **in** range(2):
73. q.push(i)
74. **while** **not** q.isEmpty():
75. **print**("peek", q.peek())
76. **print**("pop", q.pop())

### Stack.py

1. """
2. Stack structure
3. - LIFO logic
4. """
6. **class** Stack():
7. # Stack stored using an array, of predefined size
8. **def** \_\_init\_\_(self, size = 15):
9. self.\_\_size = size
10. self.\_\_pointer = -1
11. self.\_\_stack = [None **for** x **in** range(size)]
13. # Getter for stack size
14. @property
15. **def** size(self):
16. **return** self.\_\_size
18. # Pushes new data to stack
19. **def** push(self, toPush):
20. **if** self.\_\_pointer < self.\_\_size-1:
21. self.\_\_pointer += 1
22. self.\_\_stack[self.\_\_pointer] = toPush
23. **else**:
24. **return** None
26. # Removes top object from stack
27. **def** pop(self):
28. **if** self.\_\_pointer >= 0:
29. popped = self.\_\_stack[self.\_\_pointer]
30. self.\_\_stack[self.\_\_pointer] = None
31. self.\_\_pointer -= 1
32. **return** popped
33. **else**:
34. **return** None
36. # Reveals object at top of stack
37. **def** peek(self):
38. **if** **not** self.isEmpty():
39. **return** self.\_\_stack[self.\_\_pointer]
40. **else**:
41. **return** None
43. **def** isEmpty(self):
44. **return** (self.\_\_pointer == -1)
46. **if** \_\_name\_\_ == "\_\_main\_\_":
47. size = 10
48. a = Stack(size)
50. # Testing
51. a.pop()
53. #for x in range(10):
54. #    a.push(x)
55. #print("peek: ", end="")
56. #a.peek()
57. #print("pop: " + str(a.pop()))
59. **for** x **in** range(size):
60. a.push(x)
62. a.peek()
63. a.push(1)
65. **try**:
66. **print**(a.stack)
67. **except** AttributeError as e:
68. **print**("your stuff works")

# Testing

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Test No.** | **Objective Being Tested** | **Test Description** | **Test Type** | **Test Data** | **Expected Output** |
| 1 | 1a | Have user play a move, and time wait duration for AI to generate a move | Normal |  | Less than 30 seconds |
| 2 |  |  | Erroneous |  | Less than 30 seconds |
| 3 |  |  | Extreme |  | Less than 30 seconds |
| 4 | 1b | Have a new user play against the AI, and have the AI maintain a similar winrate as compared to the dealer |  |  |  |
| 5 | 1c | Have a user predict the agent’s actions for 10 random times in a game. |  |  |  |
| 6 |  | Have a user predict what the agent will do at a hand value of 17 or above, in 10 different games. |  |  |  |
| 7 | 1d |  |  |  |  |
| 8 |  |  |  |  |  |
| 9 |  |  |  |  |  |
| 10 |  |  |  |  |  |
| 11 |  |  |  |  |  |
| 12 |  |  |  |  |  |
| 13 |  |  |  |  |  |
| 14 |  |  |  |  |  |
| 15 |  |  |  |  |  |
| 16 |  |  |  |  |  |
| 17 |  |  |  |  |  |
| 18 |  |  |  |  |  |
| 19 |  |  |  |  |  |
| 20 |  |  |  |  |  |
| 21 |  |  |  |  |  |
| 22 |  |  |  |  |  |
| 23 |  |  |  |  |  |

# Evaluation

1. **Sandbagging/Trapping** is purposely betting weaker than your hand would suggest you should, in order to deceive the opponent into thinking that you have a weaker hand than you actually do, causing them to break the fundamental theory of poker. [↑](#footnote-ref-1)
2. <https://archive.ics.uci.edu/ml/datasets/Poker+Hand> (Repository of 5 Card Draw Hands) [↑](#footnote-ref-2)
3. <http://poker.cs.ualberta.ca/irc_poker_database.html> (Repository of Texas Hold ‘em Hands) [↑](#footnote-ref-3)
4. <https://en.wikipedia.org/wiki/Bayes%27_theorem> use of conditional probability to infer the chance of an event occurring – in this context, using known player tendencies to infer the condition of their hand. [↑](#footnote-ref-4)
5. <https://en.wikipedia.org/wiki/Nash_equilibrium> A state of the game where neither player has anything to gain by changing their strategy – not possible in Blackjack, but is in games like Poker. [↑](#footnote-ref-5)
6. <https://en.wikipedia.org/wiki/Monte_Carlo_method> “Monte Carlo methods (or Monte Carlo experiments) are a broad class of computational algorithms that rely on repeated random sampling to obtain numerical results. Their essential idea is using randomness to solve problems that might be deterministic in principle.” [↑](#footnote-ref-6)
7. Metagame – Strategy which transcends a prescribed ruleset. An optimal strategy for the game. [↑](#footnote-ref-7)
8. <http://www2.cs.uregina.ca/~hilder/refereed_conference_proceedings/cig09.pdf> (No-Limit Texas Hold’em Poker Agents Created with Evolutionary Neural Networks – Garrett Nicolai and Robert J. Hilderman). [↑](#footnote-ref-8)
9. <http://www.numpy.org/> (Scientific Computing Library for Python) [↑](#footnote-ref-9)
10. <https://www.tensorflow.org/> (High level library for building graphs in Python, for machine learning) [↑](#footnote-ref-10)
11. [www.pokerology.com/lessons/poker-playing-styles/](http://www.pokerology.com/lessons/poker-playing-styles/) (Article on poker playing styles) [↑](#footnote-ref-11)
12. <http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/XX.pdf> Slide 11 - essentially picking what the AI predicts as the absolute highest value play every time, regardless of confidence or how good other strategies may be. [↑](#footnote-ref-12)
13. <https://medium.com/emergent-future/simple-reinforcement-learning-with-tensorflow-part-7-action-selection-strategies-for-exploration-d3a97b7cceaf> Reinforcement learning with 4 different exploration strategies, which could lead to different behaviours. [↑](#footnote-ref-13)
14. <https://stats.stackexchange.com/questions/181/how-to-choose-the-number-of-hidden-layers-and-nodes-in-a-feedforward-neural-netw> - "So what about size of the hidden layer(s)--how many neurons? There are some empirically-derived rules-of-thumb, of these, the most commonly relied on is 'the optimal size of the hidden layer is usually between the size of the input and size of the output layers'. Jeff Heaton, author of Introduction to Neural Networks in Java offers a few more.” [↑](#footnote-ref-14)
15. <http://dstath.users.uth.gr/papers/IJRS2009_Stathakis.pdf> - Paper showing different ways of determining best NN structure [↑](#footnote-ref-15)
16. <https://wizardofodds.com/games/blackjack/appendix/4/> (Summarised Net Win in Blackjack Table – probability of net win is 42.42%) [↑](#footnote-ref-16)
17. <https://wizardofodds.com/games/blackjack/appendix/4/> (Summarised Net Win in Blackjack Table – probability of net win is 42.42%) [↑](#footnote-ref-17)
18. <https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/> Intuitive explanation behind convolutional layers in neural networks. [↑](#footnote-ref-18)