Neural Networks and their Ability to Improve Artificial Intelligent Go Players

Author: Iain Majer (c1335600)
Supervisor: Frank Langbein
Moderator: Martin Caminada

Cardiff University
School of Computer Science and Informatics

Contents

1 Introduction

Through the years people have made computers capable of challenging humans at games with the use of Artificial Intelligence (AI). From simple games such as Tic-Tac-Toe, to more complex games such as poker and even used in video games to accompany human players throughout the game.

Conventional Artificial Intelligence approaches such as Monte Carlo Tree Search (MCTS) are used for creating computational players for the board game Go, in this project I set out to analyse the ability of one of these AIs when augmented with a Neural Network. I plan on testing this by creating two AI players one with the neural network, and make them play each other while analysing their ability to play the game, and their computational efficiency. I hope to see an improvement with the AI that has the neural network helping make it's decisions.

2 Background

2.1 Go[1]

Go is a board game, created in China more than 2.500 years ago, commonly played on a 19 by 19 grid. It is a two player game with full information, meaning there is no hidden elements like in poker, and nothing is left to chance, like games with dice. There are a few specific terms used in the game of Go such as:

- Counter - What each player places on the board, either black or white.
- Chain - A set of counters of the same colour adjacent to each other (vertically and/or horizontally).
- Liberty - This is the number of empty spaces surrounding a counter or chain.
- Territory - Space that has only one counters around it.

Each player takes it in turns to place a counter of their colour on the board in an empty space. The aim of this is to create as much territory as you can. A player can only place a counter if there is an empty space to place it, and when placed it will have a free liberty. When a chain loses all its liberties it is removed from the board.

There are many ranks for players of Go, 30k - 1k (kyu), which is a beginner to an intermediate player; 1d - 7d (Amateur dan), for Advanced players; 1p - 9p, for professional players. 9p is comparable to a Grandmaster in Chess. With these ratings players are able to play others of different ranks with an handicap, so if a 5k player wanted to play a 1k player they would be presented with 4 stones to place, with Chinese rules[2] these stones can be placed anywhere, though other versions of Go have fixed positions for these handicap stones. Also since the player who plays second is also at a slight disadvantage there is sometimes a komi, a set number of points, that are presented to the second player at the end of the game to equalise this advantage.

2.2 Artificial Intelligence[3]

Artificial Intelligence (AI) was coined back in 1955 as "the science and engineering of making intelligent machines", these machines are usually designed to be good at a single (or small range) of tasks and has since been used in a wide range of fields from Technology, used to process and understand natural language[4]; Medicine, to help find patterns in research and patients symptoms[5]; Entertainment, in video games as a computer player to help or challenge the player[6]; and more recently travel, with the appearance of self driving cars[7].

2.3 Minimax Search[8]

One of the commonly used techniques in AI for games is Minimax search, this uses a game tree. Game trees are data structures where many nodes are attached together through branches. The nodes represent separate possible game states, where the branches between the nodes represent the action taken to get from one game state to the other. Minimax search is an AI technique that is used to search through these trees and pick the most promising move from them to increase the chance of a victory. Each node is given a score, using a predefined heuristic which shows how "promising" the game state is for the AI at that moment in time. The algorithm then goes through the tree searching for the best score it can while assuming the opponent will play the best possible move it can at any of it's turns, the move chosen is the highest score it can guarantee, not the highest possible score. However, this is a very intensive search since it needs to search through every node at each layer, and go down as many layers as it possibly can. This can be troublesome if the game (like Chess or Go) which have many different possible moves at any one point (called the branching factor). This causes the amount of memory needed to store the game tree to expand exponentially and be troublesome.

2.4 Deep Blue[9]

In 1996 a computer program called Deep Blue used an algorithm similar to Minimax to where it would search forward up to 12 moves (layers) and then use an additional heuristic to evaluate those game states. The AI was put to test against a grandmaster of chess, Garry Kasparov[10], it won and was therefore was the first computer player to beat a grandmaster at Chess. This was a demonstration of pure computing power, showing it's ability to evaluate over a 100 million moves each second. The ability of the AI was still needed since pure computational speed was still not enough since chess has approximately 10^{120} possible games[11].

2.5 Monte Carlo Tree Search[12][13]

Due to the memory intensity of Minimax Search other searched such as Monte Carlo Tree Search were designed that use a fraction of the memory space, and also not have the need for a scoring heuristic to score the game states it relies on probability stored in the nodes themselves with the relationship of how often the node has been played in a simulation and how many of those simulations were victories. However, the downfall to this search algorithm is the move it gives back is likely less optimal than that given by Minimax. The longer the algorithm runs the higher the chance it will give back the same move as Minimax, though the rate which it tends to this move is slow. Firstly, the algorithm selects a node already in the tree using an Upper Confidence Bound, which determines if a node should be investigated because it's not already been searched (Exploration) or if the value of a node that's already been searched should be refined (Exploitation). This balance of exploration vs exploitation is where the algorithm saves on memory space since many possible moves may not be searched if it finds one that is promising. From that selected node one or more children are added, then simulated. The nodes visited in the simulation stage aren't added to the tree to save memory space, but the results of the simulation (if it's a win or not) are appended to the node that was simulated from and iterated up changing the values of that nodes win and plays values. Since there is no heuristic evaluation at each node the AI only needs an implementation of the game's mechanics.

2.6 Neural Network[14]

Where the other AI techniques search game trees Neural Networks come from a different realm of AI. Neural Networks are made to mimic the structure of a human brain, where many neurons connect together to accomplish a complex task. The neurons are simple, they take in a value, either augment it, or pass on a signal if a certain condition is met (like if the input is greater than a certain level), then passes that new output or signal onto another neuron. When many of these are connected together you can get complex behaviour that is capable of things like pattern recognition, and classification of data.

2.7 AI & Go

Since Go is such an expansive game having $10^{1}023$ possible games[15] and the amount of possible moves at any one time are much greater than that found in Chess it has been adopted by the AI community as a great challenge for creating ever more intuitive AI for. It has found interest from enthusiasts all the way to technological giants such as Google and their company DeepMind[16]. Since the game space of Go is so large it is practically impossible to search through all of the possible games and their moves, as you would need more atoms than there are in the universe to store all the possibilities on, it has forced people to create more innovative ways to search through the game space and act on their findings. Many human players of Go have described it as an intuitive game where they rely on feeling, and intuition to play well. This has been the biggest challenge for computers since intuitive programs are not easy to define or create.

Recently (2015-2016) Google's DeepMind has made a massive breakthrough with this area of AI and have achieved the main challenge of artificial Go players, their AI named AlphaGo[16] was able to beat a professional level Go player Fan Hui in October 2015. It won 5-0 and was the first AI to beat a professional player with no handicap. They later on, March 2016, went up against the world's top Go player Lee Sedol and played 5 matches, no handicap and a komi of 7.5. AlphaGo won with a score of 4-1 with many praises. The base of this program is the same that many enthusiasts use, Monte Carlo Tree Search. However, there's was augmented with Neural Networks designed to reduce the depth and width of the tree they needed to search, meaning more time could be spent looking at what the AI thought to be the better moves.

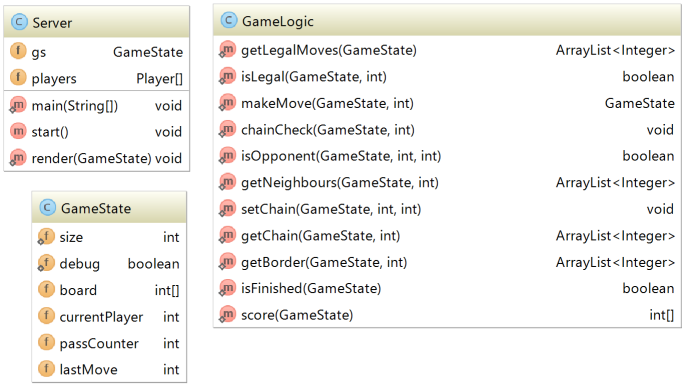
3 Design & Specification

3.1 The Game

3.1.1 Requirements

For this project I needed a platform to play Go on which has the ability to be interfaced with a combination of human or AI players, I needed this to be lightweight as I would be running it on the same machine as the AI and would be sharing the resources. It should handle all the game mechanics, placing counters on the board, calculating a counters and chains liberty, making sure moves are legal, testing if the game has finished, and calculating the scores of a finished game. To reduce memory usage and expand usability the game logic should also be separated from the game state container, letting the game state be stored without extra unneeded methods, and the game logic can be accessed by all classes that need to.

3.1.2 Class Diagram



3.1.3 Overview

My version of the game does not introduce concepts such as handicaps, or komi. This is to reduce the time programming these concepts along with them not being needed for testing the relative ability of my two AI. Along with this my game board will only be 9 by 9 in size to reduce the game space the AI have to search, speed up the time it takes to complete a full game, and it being a common size of board for starting off players and AI.

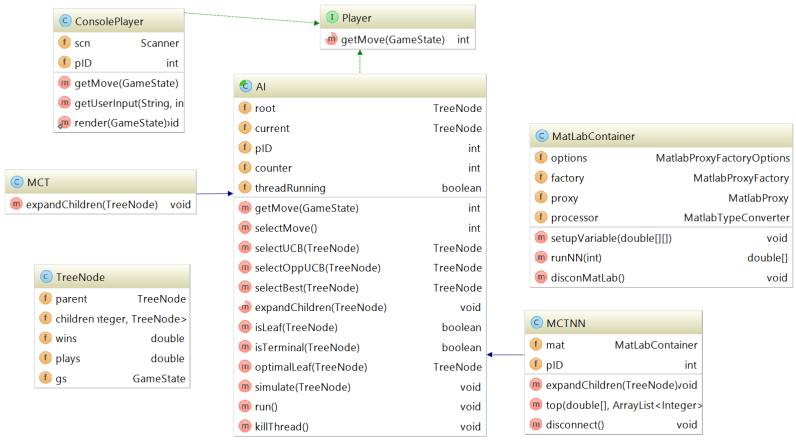
I did originally plan to create a networked version with a GUI though it was later decided that this would not be a wise use of my time and that there was no real benefit from doing this. I ended up using a console interface and having a single computer host the game and the players (AI or Human).

3.2 Monte Carlo Tree Search

3.2.1 Requirements

The first of my two AI is to only use the Monte Carlo Tree Search method to search through the game tree to find its desired move. To do this it needs the ability create a game tree. The nodes store their parent, their children, the amount of times they have been visited, how many of those visits resulted in a win, along with the game state at that time. The main implementation is handled within the AI Class due to the similarities between both of the AI making the code reusable.

3.2.2 Class Diagram



3.2.3 Overview

My implementation of MCTS is one that includes the use of Upper Confidence Bounds, this is how the MCTS scores a node to figure out which node to try more simulations down. It is designed to find a balance between exploration, investigating new moves that haven not been tried yet; and exploitation, simulating moves along already well known paths to create more accurate results. This also improves the speed in which MCTS tends towards the same answer as Minimax search. Though there are many more improvements that can be made to this algorithm competency along with its efficiency I did not feel these were critical parts to concentrate on due to the fact it is being tested against an exact duplicate of itself, so any disadvantages this algorithm has will be mimicked with the other implementation.

$$UCB = V_i + C \times \sqrt{lnN/n_i}\,[17]$$

Where V_i is the probability of the given node, C is a constant that can be tuned to suit the AI and is decided by trial and error, N is the number of times the parent has been visited, and n_i is the amount of times that node has been visited. Changing C will change how much the algorithm balances the exploration and exploitation of the search.

3.3 Monte Carlo Tree Search & Neural Network

3.3.1 Requirements

This AI also inherits the majority of it's methods from the AI class, only overriding the expandChildren method since this is where the Neural Network acts. For the Neural Network I decided to use Matlab's Neural Network Toolbox[18]. This gave me a easy way to create and train the structure of the network through graphical tools rather than spending extra time implementing the network in Java. To use this network I also used a utility called matlabControl[19] which let me access and execute Matlab commands and scripts from within my Java code.

3.3.2 Overview

For the structure of the Neural Network I chose to use the pattern recognition toolkit from Matlab. I used an augmented version of the game to create a random version of the game board, checked to see if it's a valid game state, then had a player object simulate 500 games for each legal position on the board. This gave me varying set of game states and the probabilities for a given player (black or white). The board state and probabilities were written to files and used as the dataset for training the neural network. This was done for both the black and white player, since the probabilities for either player differ.

The Neural Network's feedback is used to select the most promising half of the legal moves. This hopefully reduces the branches that the AI needs to investigate by half, while keeping the chance of finding the most promising moves high. This also means a lot more time can be spent by the AI simulating for these moves refining the probabilities these moves have giving more accurate results.

4 Implementation

For my project I decided to use Java[20] as my language of choice, this gave me the advantages of saving time since I have used Java a lot over the past few years and meant I did not need to take time to learn how to implement certain concepts in another language. It also did not disadvantage me with the speed and power of the language since I am comparing relative abilities of AI and since both of them are written in Java they are both hindered by the language in the exact same way.

4.1 Server

The Server class in the entry point of the entire program and is what hosts both the game and the players. It controls the flow of the game and implements the game logic (contained in GameLogic) to play Go.

During the creation of the Server object the GameState and Player objects are also created. This GameState object is used to store the current state of the game. The Player objects can be changed so to switch between a Human or Computer player. The Player objects are created when the game is loaded since some players need time to setup components like the MCTNN Player.

When the server is started up after creation a game loop is started that repeats until the game has finished. The current player is gotten and a request made for a move is made using the Player Interface, if the move is determined to be an illegal move the player is requested to make another. If the move is legal, it is sent with the current game state to the game logic which makes the move. After this, the new game state is stored and it's tested to see if the game has finished. When the game is finished the scores are shown and the program ends.

4.2 GameState

This object is used as a container for all the information about the game at any given point in time. Which players go it is, the state and size of the board and the counters, and the previous player's move.

The game board is an interesting data type. Even though it is stored in an single dimensional integer array it is used as a two dimensional array and stores a few pieces of information. Where the counters are on the board, represented with any number that is not zero. Who the counters belong to, determined with positive (black) or negative (white) numbers. Along with the liberty of chain that counter is apart of, which is the absolute value of the number stored. Storing data in this fashion improved memory usage greatly (as previously I had 3 different objects to

store the same information), along with computational benefits since keeping track of which counters were part of which chain was becoming a complicated task.

4.3 GameLogic

The mechanics and rules of the games are implemented in this class, the methods are static since there only needs to be 1 instance of these methods in the entire program that can be accessed by the other objects in the program. All methods in this class act off a game state that is passed in and any game state given back is a new object so not to interfere with previously stored game's state, this had to be done because or memory management to handle how Java passes objects.

Here is an overview of what each method does:

getLegalMoves: Goes through every empty position on the board if it is a legal move (with use of the isLegal method) for the given player it adds it to a list and passes that list back when it is evaluated all empty positions.

ifLegal: Tests a given move and player to see if that move is legal within the rules of Go. If the position is filled then it is outright illegal and returns false, if the move is to pass it is returned true straight away since these are easy to compute. For the move to be legal it must, after evaluation of neighbouring chains, have a liberty. This is tested by making sure only of the following are true:

- One of the counters neighbours is empty.
- One of the neighbouring chains is owned by the same player and that chain has a liberty of two or more, the liberty can by 1 only if one of the neighbours is empty which is covered by the previous condition.
- One of the neighbouring chains is owned by the opponent and has a liberty of equal to one. This means that chain would be removed thus creating an empty place around the counter thus making a liberty.

Though if this move would put the board in the same state it was in 2 moves ago that is also illegal.

makeMove: Takes in the game state and the move that is to be applied to that game state, assumes that it is a legal move, and acts on that game state accordingly. Placing the current players counter at the given position, it then iterates around that counter recalculating the chains of the neighbouring counters.

chainCheck: This is what is used to recalculate the liberty of a given chain, given the game state and the position of any counter in a chain it calculates what the liberty is and sets the value of it for each counter in the chain.

isOpponent: Simply returns if the counter at a given location is owned by the given player or their opponent.

getNeighbours: Returns an iterable list of all the neighbouring positions around a given position, handles whether the given position is at the edge of the board or not.

getChain: Returns a list of all the positions of counters in a chain that the given counter belongs to. It does this by expanding out from the given position looking for counters with the same sign (positive/negative) keeping track of which counters it has visited. Once no more counters are being added to the chain it returns the discovered chain.

setChain: Uses the getChain method to set the chain to a given value.

getBorder: This is used for getting the surrounding positions of a chain, and is mostly used within the chainCheck to count the liberty of a chain. It simply uses getChain to get a chain, then expands all the counters out in all directions, if a position which is not in the chain is found it must be on the border of the chain. These are all stored and returned.

ifFinished: Checks to see if the game is at it is end state, which is simply if both players have passed consecutively then the game is over.

score: This is used at the end of the game to evaluate who won the game. I used Chinese scoring since it was one of the simplest to implement. A player's score is equal to the number of counters they have on the board, plus the number of spaces in territory they own. A player owns territory if only their counters surround the empty spaces.

4.4 Player

Player is an interface used by the server to interact with any type of player be it Human or Computer without special exceptions. The only method that is required is getMove which is what is called when the server is requesting a move from a certain player.

4.5 ConsolePlayer

ConsolePlayer is a simplistic console based interface for humans to interact and play the game. It implements the Player interface so that the server knows how to request a move from them. This is done by rendering the board in the console (through use of a text representation of the board and counters that occupy it). Then requests an X and Y coordinate to be entered into the console. This is turned into a single number, since the board is stored as a single dimensional array and returned to the server.

4.6 Treenode

Treenode is what the game trees are made out of. Each tree node holds a reference to its parent, children, the amount of times it has been visited, and the amount of those visits which resulted in a win. They also hold the game state for the game at that given node. Using these the AI is able to create and search through a game tree.

4.7 AI

The AI class is an abstract class that implements many methods that are shared by all the other AI and lays out the methods that should be implemented by the AI that inherit it. The methods are as follows:

getMove: Implemented from the Player interface, this method is was starts up the AI and makes it decide which move it should make dependant on the game state it's given. Starts a thread which expands the game tree while it counts down a predefined amount of time then kills the thread.

selectMove: Picks the best move from the children of the current game state node. This is what's called after the AI has expanded it's tree to get the most promising move.

selectUCB: Used for the selection part of Monte Carlo Tree Search, where it picks the node with the best Upper Confidence Bound to iterate down and simulate.

selectBest: Does the same as selectMove but a more general implementation for any given node.

expandChildren: Abstract method not implemented in this class, view description of this method in the AI implementations. Handles the expansion part of MCTS

isLeaf: Returns a boolean of if the given node has any children.

isTerminal: Returns a boolean of if the given node is a terminal node and represents the end of the game.

optimalLeaf: Uses *selectUCB* to iterate down the tree from a given node for the selection part of MCTS

simulate: Handles the simulate part of MCTS, plays a random game from a given tree node and checks to see if that random game wins, the results of the win iterate back up the nodes, changing their win/plays values.

run: Executes the thread which expands the game tree using the stated methods above. Going through each step, selection, expansion, and simulation, until the thread is killed by the timer.

killThread: This is what's called to kill the thread in a safe manner.

4.8 MCT

This is where the first AI using a simplistic Monte Carlo Tree Search is implemented. It extends the class AI since that handles the majority of the programming, the only method in this class is the implementation of *expandChildren*.

expandChildren: Given a tree node it gets the list of legal moves using the Game Logic, then expands the tree out with all the legal moves at that are possible from that node.

4.9 MatLabContainer

Since I created the Neural Network using Matlab I needed some way to interact and execute it. To do this I created this class which implements the abilities of a Java MATLAB Interface (JMI) called matlabcontrol to the extend needed in this project. When created this object sets up the interface between Java and Matlab and configures it with predefined settings, such as whether to create a new session or connect to a previous session.

setupVariable: Variables have to be defined and set before executing the Neural Network script so this is what handles sending Matlab the board state that is being tested and creates a variable for the returned data.

runNN: This method is what adds the location of the Neural Networks script, requests that Matlab run it, and waits for the response.

disconMatLab: Closes the JMI properly so extra connections aren't laying around, usually executed at the end of a game.

4.10 MCTNN

This is the second implementation of the AI class, it also uses the MatLabContainer since this is where the Neural Network is used. Like MCT this class doesn't have many methods since they are handled by the AI class.

expandChildren: The implementation for this method differs quite a bit from the one in MCT. It takes the game board and formats it into a 81x1 matrix, passes it to Matlab and has the MatLabContainer execute the Neural Network. The returned results are passed through the top method which decides which of the positions should be added to the game tree and investigated.

top: This method decides from the Neural Network data which positions should be added to the game tree. It does this by looking at the returned values for the legal moves. The method makes a list of all the legal moves greater than the median value of the legal moves. If the amount of legal moves is lower than a certain percentage, the list of legal moves is just used. This helps the AI in late game scenarios where there are very few positions to choose from, but helps greatly with speed at the start of the game since the branching factor at the start is greatly reduced.

The reason I go back to using the entire set of legal moves is due to the percentage any single good move makes of the entire legal set. When at the beginning of the game a

single good move is 1/81 of the entire set, meaning the amount you gain from simulating one more move (if it's good or not) is very low. Where as when you reach the late game you have single moves representing 1/5 to 1/3 of the legal set, giving a lot more intensive to investigate and simulate these moves.

5 Testing

The aim of this project is to determine the relative ability of each of the aforementioned AI to play Go. To do this I have decided to test these AI by comparing their outright skill to play Go by running multiple matched between them, since their running on the same computer this means their resources will be exactly the same, and also the time they get to expand the game trees will also be restricted to the same length. This will ensure a fair test and will purely show how well they are at playing against each other. In the matches the AI that goes first will be switched so to remove any advantage the first player may obtain by going first. After each game the program will be restarted so that the game trees are destroyed. This gives both AI a fair starting point. All tests are run over 30 games of Go.

5.0.1 Results

For a control I tested MCT vs MCT, since this will show the disparity and advantage the black player has for going first.

Table 5.1: MCT vs MCT

Black	Draw	White
14	0	16

Table 5.2: MCTNN vs MCT

Black	Draw	White
19	0	11

Table 5.3: MCT vs MCTNN

Black	Draw	White
13	0	17

5.0.2 Analysis

The results of the control test seem to show that the AI are reasonably evenly matched with the game going either way, with the results looking to hover around half each. This seems to suggest there is not an advantage for going first between the two players.

The second round of games MCTNN vs MCT show a that the black player won a great deal more than before where it was nearly 1:1 this round is closer to 2:1. This suggest that the Neural Network has improved the AIs ability to play Go.

This is further shown by the results of the third round where MCTNN (now playing white) won just a few more games than MCT (playng black). This lead could likely have been extended

by spending more time training the Neural Network as white, since the training data for this neural network is slightly less than that taken for the black player.

Overall this shows that the Neural Network has been shown to increase the chances that the AI will win against it's counterpart, which was the target for this project.

6 Improvements and Future Work

There are many things in this project that would have been implemented, improved, and changed if I were to of had more time. One of these things is the language I chose to write the project in, a language like C++ have given me more control over the way my Objects were passed throughout my program and in turn would have given me more control to implement some aspects of the game with greater ease. I also would have used threading more if I had time to improve my code, since this would have benefited the AI greatly. Other improvements would be things such as a GUI for human interaction, Networking so AI can be run on separate machines, and the ability to run multiple games at once for faster data collection.

As for the research I would like to have looked into other ways of using the Neural Network to improve the AI. Since my approach reduced the width of the game tree, there may be uses for Neural Networks to reduce the height of which the AI has to go through the game tree. Maybe more localised Neural Networks could pick out defined patterns that should be noted and which positions to play when those patterns appear.

Other tests could also be run to determine how the Neural Network helps the AI, through testing the amount of simulations it runs you can see if the fidelity of the values used for selection are improved, and by testing how well the AI play consecutive games shows how quickly they learn and expand the game tree.

7 Applications

The ability for AI to play more intuitive games is a huge step forward for AI in many fields of science and life. With the ability to make more intuitive decisions it could improve the ability for physical robots to make last minute actions in situations it was not specifically designed for, which means we could possibly send robots into more dangerous situations like earthquake zones to help find survivors, or deeper seas and caves for research. Computers may see a boost in ability for understanding natural language and speech, making human computer interfaces more user friendly and intuitive to use.

8 Conclusion

Overall I feel that I have achieved my aim of this project. I implemented the game of Go in Java that was capable of being played by AI. I implemented an AI using Monte Carlo Tree

Search then made an augmented version with a Neural Network helping choose which moves it should investigate. Testing these AI against each other I was able to show an increased level of skill and speed at which the AI expands the game tree. Thus proving that Neural Networks are able to be used to improve Monte Carlo Tree Search based AI at playing the game of Go.

Bibliography

[1] https://en.wikipedia.org/wiki/Go_(game)

[2] https://en.wikipedia.org/wiki/Rules_of_go

[3] https://en.wikipedia.org/wiki/Artificial_intelligence

[4] http://nlp.stanford.edu/projects/DeepLearningInNaturalLanguageProcessing.shtml

[5] http://www.nature.com/nm/journal/v7/n6/full/nm0601_673.html

[6] http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.28.5457&rep=rep1&type=pdf

[7] https://arxiv.org/ftp/arxiv/papers/1409/1409.7121.pdf

[8] https://en.wikipedia.org/wiki/Minimax

[9] https://en.wikipedia.org/wiki/Deep_Blue_(chess_computer)

[10] https://en.wikipedia.org/wiki/Deep_Blue_versus_Garry_Kasparov

[11] https://en.wikipedia.org/wiki/Shannon_number

[12] https://en.wikipedia.org/wiki/Monte_Carlo_tree_search

[13] mcts.ai/

[14] https://en.wikipedia.org/wiki/Artificial_neural_network

[15] https://en.wikipedia.org/wiki/Go_and_mathematics#Game_tree_complexity

[16] https://deepmind.com/alpha-go

[17] http://www.cameronius.com/research/mcts/about/index.html

[18] http://uk.mathworks.com/products/neural-network/

[19] https://code.google.com/archive/p/matlabcontrol/

[20] http://www.oracle.com/technetwork/java/javase/overview/index.html