DAT410 - Assignment 6

Eric Johansson & Max Sonnelid

March 1, 2021

Group number: 70	
Module number: 6	
Eric Johansson	**Max Sonnelid**
970429-2854	960528-5379
MPDSC	MPDSC
johaeric@student.chalmers.se	sonnelid@student.chalmers.se
25 hours spent	25 hours spent

We hereby declare that we have both actively participated in solving every exercise. All solutions are entirely our own work, without having taken part of other solutions.

Introduction

In this assignment, we will be working with game playing systems. In particular, we will work with tree search (Monte-Carlo or otherwise) in order to learn to win (or at least not lose) at the game Tic-Tac-Toe on the classic 3x3 grid. Firstly, an article will be summarized and reflected around. Secondly, the implementation of the game playing system will be described. Thirdly, summaries of lectures will be presented. Fourthly, reflections on the previous module will be done.

1 Summary of "Mastering the game of Go with neural networks and tree search"

1.1 Eric Johansson

In October 2015, AlphaGo became the first computer to beat a proffesional player in the game of Go. In March 2016, managed to win against the best player in the world, Lee So-dol, something that just before the occasion was seen upon as an impossible task. In the article *Mastering the game of Go with deep neural networks and tree search* (Silver et al., 2016) the creators behind this machine talks about how the built and train the model and in what way it differs from previous computer generated players.

One key contributor to the high performance of the model was Monte Carlo Tree Search, which estimates a value on each possible state in order to decide which move to make. Doing this a lot of time will eventually converge into a value function $v(s) \approx v^*(s)$. A proper value function $v^*(s)$ would be the optimal function, but is yet unreachable due to immense amount of different play-outs of a game of go (around 250^{150}). On top of the Monte Carlo Tree Search, additional policies that stems from expert plays are added to make the model more efficient. The result of this model is impressive, but when combineed with reinforcement learning of value networks, the performance increases even more. To do this, the AlphaGo team uses multiplte CPUs and GPUs (48 reps. 8). The next step in development was when multiple machine were used. In total 1202 CPUs and 176 GPUs made the model practically unbeatable. Out of 495 games, it won all but one. The best performance was achieved when the model included an equal proportion of the value network and roll-outs.

1.2 Max Sonnelid

The article *Mastering the game of Go with deep neural networks and tree search* describes how the computer program AlphaGo eventually succeeded at defeating Fan Hui, one of the world's champions,

in 5 different games of Go in 2016. Compared to chess, Go has a much bigger game complexity and traditional value function methods, which has lead to superhuman performance in e.g. chess, have not succeeded in Go. Making a computer defeat a professional Go player was therefore for many years considered as one of the great challenges for AI. Consequently, the eventual success of AlphaGo was considered as a major breakthrough for AI research.

What made AlphaGo successful was a novel combination of both supervised and reinforcement learning for training *deep neural networks*, which are then used for evaluating moves in a *Monte Carlo Tree Search* from which eventually the most promising move can be selected. The training of the neural networks was done in several steps. Firstly, a *supervised learning (SL) policy network* was given training data consisting of previous Go games played by experts and the network then learned to predict these expert moves with an accuracy of 57 %. Secondly, a much faster, but less accurate (24 % accuracy) *roll-out policy network* was trained on the same training data. Thirdly, a *reinforcement learning (RL) policy network* was trained by playing games against randomly selected previous iterations of the policy network. The RL policy network was eventually able to win more than 80 % of the games played against the SL policy network. Fourthly, a *value network* was trained on previous games played by instances of the RL policy network and the value network was then able to estimate a value function that predicts a binary outcome from a certain position, compared to the probabilistic outputs from the policy networks.

Finally, the policy and value networks were combined in a *Monte Carlo Search Tree* algorithm, where the nodes in the search tree are visited in a way that balances between maximizing the action value and exploring nodes with few visits. Explored leaf nodes are first processed by the SL policy network and then evaluated by a combination of the value network and a random roll-out played out by the fast roll-out policy network until a terminal node is reached. After traversing the search tree through many simulations, all nodes have accumulated visit counts. As the nodes are primarily selected for visits based on maximizing the action value, the most visited node is considered to be the most promising one and therefore selected as AlphaGo's draw. Enabling this algorithm to run demanded a large amount of parallel computing power, which resulted in the final version of AlphaGo using 40 search threads, 48 CPUs and 8 GPUs.

2 Implementing a Game Playing System

The final implementation resulted in a so called *Human vs Bot* Tic-Tac-Toe game, where the human player is able to play against a bot, which automatically makes its draws based on a Monte-Carlo search tree. We also made a method where two computer generated players plays against each other.

This final implementation consists of two main parts, which are both described below: *the representation of the Tic-Tac-Toe game* (found in the class `State`) and *the Monte-Carlo tree search* (found in the classes `MonteCarloTreeSearchNode` and `MonteCarloTreeSearch`).

In order to connect these parts into a functioning *Human vs Bot* Tic-Tac-Toe game, a method called `play_game()` was created, which initializes an empty state using the State-class. This empty state is represented through a 3x3 matrix that initially consists of zeroes. `play_game()` then lets the user make a move, and creates a new board based on that decision. Then, the bot takes this board as an input by first making a `MonteCarloTreeSearchNode`, which is an object based on the class with the same name. After that, a new object is created by the class `MonteCarloTreeSearch` which takes the above-mentioned object as input. Finally, the best move that the bot decides to make is made through the method `best_action`. This procedure is then repeated until either the user or the bot wins. The logic behind these classes and methods will be described below.

Lastly, it was also interesting to see how a *Bot vs Bot* Tic-Tac-Toe game would play out and therefore the method called `play_game_two_bots()` was created. It works in the same manner as `play_game()`, but with the difference that the human input is replaced by an additional `MonteCarloTreeSearch`, which takes the most recent board state as input and gives the new board state with the best move as output to the other `MonteCarloTreeSearch`.

2.1 Representation of Tic-Tac-Toe

The `State` class makes up the representation of the Tic-Tac-Toe game. When a `State` is first created, it is assigned an empty 3x3 matrix only filled with zeroes and its current player is assigned as -1. As it was easy to implement the periodic change of current player by only multiplying the current

player with -1, it was decided to set the two possible markers to -1 and 1 (compared to the standard X and O). The `draw` method then lets a player place its marker at a free spot of its own choice in the board. After each draw, the same method also checks whether someone has won the game. This win check is done by the `check_win_3` method, which returns the number of the player by first checking for a win in the rows, then checking for a win in the columns and eventually checking for a win in the two diagonals. If no player has won, zero is returned. The last method in this class is `possible_child_states`, which for a current board state returns an array with all possible next board states (i.e. all possible draws for the current player).

2.2 Monte Carlo Tree Search

The Monte Carlo Tree Search is split up into two separate classes. The first one, `MonteCarloTreeSearchNode` takes a state (also known as a board) and a parent as input. The class consists of multiple methods, and the easiest way to describe them is probably in the order of which are utilized by other methods. Therefore, we move forward to the other class `MonteCarloTreeSearch`. It is the method called `best_action` that is ultimately called in the `play_game()` method when the bot is making a move. What `best_action` does is that it proposes a node to go to (a move to make). This is done using the tree policy (which will be discussed further down). After that, a reward (win or lose) is determined using the `rollout` method, which basically simulates the progress of a match that starts in the specific state that was initialized earlier. The simulation follows some rules, called `rollout policy` which also will be discussed further down. When the simulation reaches a final state (a player win or the board is full) the method returns a 1, -1 or 0 depending on who won the game. When the rollout function returns a result, the reward is then backpropagated up in the tree. This means that all visited nodes get counted as visited one extra time, and the total score from the games that started from that node is updated in a defaultdict. At this stage, all possible children are expanded and and thus there is a possibility to explore them. Each of them have their respective number of visits and wins. Finally, the method `best_child` calculates the child with the highest parameter. To conclude, the algorithm explores different moves a predetermined number of times, and then chooses the move that yields the highest score (i.e, the highest probability to result in a win).

2.2.1 Roll-out policy of your player

The chosen policy in the roll-out method was to make totally random next moves by generating a random integer that corresponds to one of the available moves. We think that there exist better options, such as following some additional rules. For example, we could have added some basic rules that forces the model to always prevent ourselves from winning if we have two in a row, and in the same manner always choose the winning option when there is a possibility to win in one move. This is something that our model already takes into account since it loops through all possible moves and chooses the best one, but adding these rules in the roll-out policy could speed up the performance and hopefully mitigate the chance of faulty moves. In more advanced games, such as go or chess, we think the roll-out policy could be more useful. In those cases, it would be very inefficient to move totally at random since an average chess game consists of 40 moves, whereas an average Tic-Tac-Toe might consist of six or seven moves. Thus, there would have to be a large amount of random simulations before such a strategy yields any valuable insights. However, in our case, there exists relatively few outcomes, and thus, we chose to stick with the random simulation strategy.

2.2.2 Policy of opponent

When simulating possible outcomes, the computer is playing against itself. Thus, there exists an opponent to our bot. To make these simulations, we used the roll-out policy that was mentioned before, and when a move was made in the simulation, the `current_player` variable was multiplied by -1, meaning that it is now the other player's turn. When it is the other player's turn, it uses the same methods as before. In this way, the same code, and thus the same strategy is used for both our bot and its opponent. Therefore the roll-out-policy is still uniformly random. In the end of our code, we made a method that makes two bots play against each other. Here, player one takes an empty board as input and initializes a Monte Carlo Tree Search and eventually makes a move and returns that state. The second bot takes the new state as input and makes an optimal decision. This procedure is repeated until one of them wins.

2.2.3 Selection policy in search tree

When the simulations have run, an explored tree is then created. All nodes have a value `N` which represents the number of times that node has been visited. There is also a dictionary called `results` which store the wins for each player and the draws. During the simulation, when the algorithm decides which move to try out, it uses something called *Upper Confidence Bound applied to trees* which is the following formula:

$$UCT(v_i, v) = \frac{Q(v_i)}{N(v_i)} + c * \sqrt{\frac{log(N(v))}{N(v_i)}}$$

where Q is the results, N is the number of visits, `v` is the node from where the decision is to be made (the parent/root) and `v_i` is the node that represents a possible move (the child). The first term of the algorithm is known as the exploitation part. This favours the moves that results in win. However, if this were to be used solely in the formula, a lot of great moves would have been missed, since not being included in the first simulation does not necessarily mean that it is a bad move. Thus, the second term is added, known as the exploration part which makes sure that nodes with few visits eventually is tried out again, even if that node has few wins in earlier simulations. This policy was something that we were exposed to in the lectures, as well as in the blog by int8 (2018). We tried to use some different c-values, to test which mix between the exploration and exploitation term that yielded the best result. After some different values, we decided to use 1.0 as c-value.

Eventually, after the simulation has run its course, it is time to chose which move to make. We did this by using a method that returns the node with the highest value of `N`, i.e, the one that has the most visits. This seems reasonable since the algorithm eventually visits the node which yields a lot of wins. And thanks to the exploration term, fewer nodes is ignored. Thus, we felt that using a method that takes the actual number of wins into account would be obsolete.

2.2.4 Updates to search tree ("back-up")

The updating of the search tree occurs every time one simulation for a visited node is finished. This is done using backpropagation. The method `backpropagate` takes a node and a result (-1, 0, or 1) as input and then updates the current node with the new values, and then through recursion calls for backpropagate and uses the same result but its parent as input node. This is repeated until there does not exist any more parents - which means that we have reached the initial root. This method guarantees that each node's results and visits reflect the results from the simulation started in their parents and grand parents and so on...

2.3 Sampling implementation

In order to make the optimal choice, different moves are explored and to evaluate these moves, simulations are made in order to see if a specific move ends up in a win or loss. One way to make sure that one move is the best one is to simulate every possible second move, and out of those simulate every possible third move, and so forth until the game is ended. In the case of Tic-tac-toe, this would have been possible since there is a total of approximately 26 000 possible states in the game. This is something that a regular computer would be able to simulate. However, in the game of chess, the different possible outcomes outgrows the number of all the atoms in the universe, the number of hairs on a person, and the number of grains of sand on earth multiplied together (Wolchover, 2010). Hence, the computing power needed to execute these calculations is not yet accessible. Therefore, we chose to simulate different outcomes of moves a predetermined number of times. In this way, we can control how much computing power that is needed, and can yield a relatively good result with quite few iterations. In our final method, we iterate 20 times.

2.4 Strengths and weaknesses

A major strength with the algorithm is that the bot always makes a feasible draw. This only takes a few seconds, which could be considered a reasonable duration for in a game as Tic-Tac-Toe. Another strength is that the bot knows how to win and most of the time makes a draw, which clearly could lead to a future win. However, the bot struggles when the human player tries to block the bot and therefore is rather deficient at making draws that will prohibit the human player to make good draws. Today, therefore, the human player wins in almost all games.

Another strength with the algorithm is that it is easy to scale it on larger Tic-Tac-Toe grids and also on one kinds of zero-sum games, such as chess. The current implementation of Monte Carlo Tree

Search does not depend at all on the properties of the 3x3 Tic-Tac-Toe game, but rather depends on the more general properties of a zero-sum game, such as either only one or no player wins, the game is played in a sequential manner with alternating player and that there are several possible child nodes originating from the current board state. If a state class for a chess game would be created and follow the same interface as the state class for Tic-Tac-Toe, probably no change at all would have to be made to the Monte Carlo Tree Search. However, the speed of the current algorithm would probably decrease severely because of the many more possible child nodes from each board state in chess. Today all possible child nodes are found in the roll-out method and probably some kind of heuristic would need to be found for chess in order to not have to calculate every single possible move for each node.

Another deficiency with the algorithm is that it is currently not able to save statistics from one draw to another, i.e. every time the algorithm makes its draw it starts its exploration of the game from scratch. This would probably improve both the running time and quality of draws if it could be implemented.

Another strength is that the algorithm can easily be adapted to the powers of the current computer. If the current computer is not rather powerful, the number of simulations in Monte Carlo Tree Search can be decreased and vice versa.

Another strength is that, as long as the algorithm has been gives the rules of the game, it will learn clever tricks about the game by itself by simulating a massive amount of games and getting rewarded for wins. Therefore, as long as the human programmer is able to program the properties of the game and enough computing power is present, the algorithm should be able to gain proficiency in really any kind of zero-sum game.

An alternative algorithm for this case would be the Mini-Max algorithm, which compared to a Monte Carlo Tree Search takes all possible moves a player can take into account when making a draw. Then the algorithm aims to minimize the opponent's chances of winning, while simultaneously maximizing the chances for the bot to win. That the Mini-Max algorithm takes all possible draws into account makes its decision-making more robust compared to Monte Carlo Tree Search, but this is also its drawback. Investigating all possible draws is a very time-consuming task, which scales exponentially with a game with many different board configurations. This task can be simplified somewhat by using heuristics as alpha-beta pruning, but the task still scales exponentially with a larger game. As the game in this assignment is rather small, the Mini-Max algorithm would probably be suitable, simpler and possibly also superior to the Monte Carlo Tree Search. However, as Monte Carlo Tree Search is a more suitable method for the general zero-sum games, as the running time does not necessarily scale exponentially with the game size, it was a more interesting task to implement Monte Carlo Tree Search for this assignment.

3 Summary of lectures

3.1 Eric Johansson

3.1.1 Lecture 6 - Game Playing systems

This is a subject that has been widely written about in the last years. The first automated systems where chess-playing computers that were created almost at the same time as computer were invented. By that time that was considered as being AI, but not in the same extent today. There exists multiple genres of games, one of them being zero-sum games, such as Tic-Tac-Toe. That game can be described as a branching path. The branch consists of mulitple nodes, and we want to build a model that finds the node where they wins (gets three circles in a row or diagonal). This can be done through backtracking. In order to increase the chance of win against any opponent, minimax optimization is used to guearantee syccess agains the best opponent. In tic-tac-toe, all states are observable. However, there exists games where all states are not observable, e.g, Starcraft and DoTA. In the minimax, there exist to policies, π which is our own policy and μ which is the opponents policy. The idea is to optimize using the following formula:

$$min_{\pi} max_{\mu} R(\mu, \pi)$$

A problem that often arises is that the nr of possible state spaces often grows beyond storage or exploration. It works for Tic Tac Toe, but not for chess or Go. A way to approach these problems is to use Monte Carlo tree search. Using this means that we do not need to store all possible actions,

but is rather exploring potiential actions, and then compare these with previous explored paths. In this area, the trade-off between exploration and exploatiation is often discussed. The same trade off is discussed when comparing bandits. Here, upper confidence bounds (UCB) is a commonly used. This idea can also be implmemented in tree (then known as upper confidence trees).

The problem of storing a possible states can be approach by approximating a function that translate of a specific state into a value. This function can be approximated using deep learning.

3.1.2 Follow-up lecture

A theme of the last assignment was interpretability. It can be defined through multiple criteria, such as trust, casualty, transferability, informativeness and fairness. One of the more important desiderata is casualty. To illustrate what is casualty (or actually, what is not) one can look at a graph where the x-axis is the amount of chocolate a specific country eats annually, and the y-axis represents the number of Nobel Price winners from the same country. At a brief glance, it looks like they depend on each other, but this is rather an example of correlation without causation. However, without any additional information beyond the data samples, it is very hard to draw any conclusions about what phenomena that causes the other. To conclude, regular statistics is in this case not enough to draw any valuable insights. One way to learn more about a system is to perturb different variables to see how the other changes. This can be notated as $do(A = a)$, where do represents some kind of intervention. Then, by comparing P(B—A) and P(B—do(A=a)), one can draw more conclusions about what are the root cause to B. This lecture was not really about explaining how to calculate causality in data used in AI models, but rather about highlighting the importance of thinking in these directions.

3.2 Max Sonnelid

3.2.1 Lecture 6 - Game Playing Systems

- In the late 1900s, *chess computers* started to be more successful than humans players. This was a great achievement for AI systems in general, as chess playing had previously been considered as one as the great challenges for AI. Nowadays, *Open AI Five* is an example of a system able to succeed at playing much more complex game than chess, in this case DoTA.
- *Zero-sum games*, like Tic-Tac-Toe, is defined as one player winning implies the other player losing.
- Games are normally about *deciding which action to take*. The goal is to select the actions that improve the player's chances of winning the most. In most games the actions taken change the possible future actions. Therefore, in order to win, we need to account for both good and bad futures when selecting the actions.
- *Minimax optimization* is a common algorithm for deciding which action to take and aims to minimize the maximum success of your opponent. Assume you know the best possible move for your opponent and that your opponent will play that move. Then count future success rates of each possible action and take the action with the highest future success rate.
- However, an *exhaustive search* of all possible actions is not always feasible, e.g. in Tic-Tac-Toe the number of possible board states is around 20,000, while in chess the number of possible board states is 10^{47}.
- One solution to the problem with exhasutive search is to use *Monte-Carlo Tree Search (MCTS)*, which takes advantage of random sampling of possible actions and calculates the value of each action sampled and thus does not need to explore all possible actions. The statistics collected can then be used for improving the selection policy and the simulation/roll-out policy.
- There is a *trade-off between exploitation and exploration* in in the selection policy. A *greedy policy* only focuses on exploitation (taking the best already known action), while an *epsilon-greedy policy* has a balance between exploitation and exploration (sometimes taking a random action with the aim that it might be better than the already known actions).
- MCTS is repeated until sufficient statistics have been gathered and eventually the action with the highest accumulated value is selected.

3.2.2 Follow-up lecture - Diagnostics Systems

- This lecture revolves around the concept *causality*, which is a necessary concept to define when answering questions such as: Which genes cause which disease? Does smoking cause cancer?
- Measuring which variable that is causing another variable is a rather hard task as a model with the same *probabilistic independencies* can be interpreted as two models with different *causal relationships*. Therefore, it is not possible to identify causal relationships between variables only based on joint distribution of the same variables.
- Causality is often defined in terms of *interventions*, i.e. what happens when we change a single variable in a system and observe what then happens with the other variables. This is also called retrieving the *interventional distribution*, in contrast to the normal *observational distribution*.
- A *causal graphical model* describes the causal interpretations of the structure between variables which are not implied by the statistical distribution.
- In a *mutilated graph*, one single variable is intervened on and is therefore no longer influenced by the other variables. Thus all incoming edges to the intervened variable are removed.
- A *structural causal model (SCM)* links distribution and structure of variables through a set of deterministic relations, which is done by capturing all variance in noise variables. The left-hand side in the structural equations building up the model corresponds to the start of an arrow, while the right-hand side corresponds to the end of an arrow. An intervention on a variable replaces an equation in the SCM and mutilates the graph, i.e. removes incoming edges from the intervened variable.
- In order to prove a causal relationship, *causal identifiability* is needed, which is currently a big research topic. Causal identifiability can be retrieved both by *randomized experiments* and by *controlling for confounders*.

4 Reflection on the previous module

4.1 Eric Johansson

The purpose of this assignment was learn more about multiple classification system and the importance of interpretabililty. The first part of the module consisted of some reading material. My personal favourite was "The Mythos of Model Interpretability" which discusses the different parts of interpretability and the need for it. The implementation part consisted of three separate model. The first model were based on predefined policies. The rules, however, were set by the group individually. This part was fun, since it involved playing with the data in order to gain insights about proper thresholds. The more thought we put into the thresholds, the better result we got. I also enjoyed the last part where we got to implement a classifier of our own choice. After a lot of discussion about different models and their interpretabililty, we decided to search the web for additional inspiration. Luckily enough, it turned out that there existed a classifier that managed to exceed our previous models in both accuracy as well as interpretability. From that, I realised the importance of understanding different machine learning models in order to make the most out of them. Depending on what task you want to solve, different models are suitable. A final take-away from this weeks assignment was that a relatively simple classifier can yield great result (up to 97% accuracy). Of course, the result might not be as good in real life, due to more variation in real life data sets, but even if they are a bit worse they could still be of great use. I really think that machine learning will continue to grow in health care, as well as in other areas.

4.2 Max Sonnelid

The main objectives in previous week's assignment was to implement a diagnostics system for breast cancer as well as making sure that the decisions made by the system are interpretable for the end user. First of all, it was an interesting task to put less emphasis on a single, complicated implementation of a classifier, and instead put more emphasis on comparing and optimizing several kinds of classifiers. In our case, it turned out that the for us previously unknown classifier "Rule-Fit" was superior to both the rule-based and the Random Forest classifier with respect to accuracy, sensitivity and specificity. For real-life settings, when implementing a bigger project, it is probably important to do

research about the most suitable classifier for the current task, as small improvements in accuracy can have a big impact in larger software projects. This is an important experience I will take with me from this module.

In previous assignments, the interpretation of features has been rather straight-forward. However, in this assignment, it was firstly hard in the beginning to interpret what the suffixes of the features meant and secondly hard to know how the different features were going to be combined to the more abstract features pre-defined for the rule-based classifier. Instead of taking a lot of time to gain more subject knowledge about the features, we went on and tried rather arbitrary combinations of the feature, which eventually in the end gave a rather high accuracy. Looking back, the rule-based classifier had probably benefited from us learning more about the actual subject, as the combinations of features could have been done in a more fact-based manner. It is hard to know whether this approach would have been able to improve the accuracy of the classifier. However, the rule-based classifier would probably appear more trustworthy for external users with a higher degree of fact-based rules.

Finally, it was very interesting to learn more about interpretability and different ways of defining this concept. A big take-away from the article about interpretability was that if humans can not always be considered rather interpretable, is it then reasonable that machines should have much higher requirements on interpretability? Take for example a physician making cancer diagnoses, which has gone through a long education and years of professional experience. He can probably not explain every single step in his diagnostics, but rather relies on his experience. Furthermore, it was interesting to read about that neural networks can be trained to describe selected parts of their complex decision making without describing every single step of the decision making (be so called post hoc explainable), which can be likened to the example about the physician. Having interpretability in mind is important when implementing AI in sensitive settings such as healthcare and is something that would be interesting to work deeper with in future assignments.

References

int8 (2018). Monte carlo tree search – beginners guide.

Silver, D., Huang1, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. (2016). Mastering the game of go with deep neural networks and tree search. *Nature*.

Wolchover, N. (2010). Fyi: How many different ways can a chess game unfold?