A Review of Scrabble AI and a Look Toward Improvement

Nathan Comer, Alex Mahrer, Thomas Juszczyk, Luis Chavez
May 16, 2017
CSCI 5521

Abstract

We review advancements in AI as they pertain to the game of Scrabble. We review the history of Scrabble AI research including iterative improvements to the data structures developed for the problem as well as the Scrabble bots that utilize them - Quackle and Maven have been the prominent Scrabble bots for over a decade. We examine modern advancements in AI, specifically in reinforcement learning and deep neural networks, and we propose a direction for future research in Scrabble AI.

Introduction

Scrabble is a word game in which players compete to achieve the highest score by placing tiles onto the board (figure 1). Each tile bears a letter and a set of tiles can only be played if they form a word from the specified dictionary. Each tile also bears a score in the bottom right, which is used to calculate the score gained by a player when a word is placed. Each player has a rack of tiles containing the tiles they have available to play (figure 2). Each player takes a turn placing a word onto the scrabble board, and each word placed on the board must join an existing cluster of tiles. The possible plays that can be made and how various plays are scored is outlined below [9].

Plays:

- Adding one or more letters to an existing word, e.g. (JACK)S, HI(JACK), HI(JACK)ING.
- "Hooking" a word and playing perpendicular to that word, e.g. playing IONIZES with the S hooked on (JACK) to make (JACK)S.
- Playing perpendicular to a word, e.g. (JACK), then YEU(K)Y through the K.
- Playing parallel to a word(s) forming several short words, e.g. CON played under (JACK) that to make (J)O and (A)N.

Scoring:

- Each new word formed in a play is scored separately, and then those scores are added up. The value of each tile is indicated on the tile, and blank tiles are worth zero points.

- The main word (defined as the word containing every played letter) is scored. The letter values of the tiles are added up, and tiles placed on DLS and TLS are doubled and tripled in value, respectively. Tiles placed on DWS or TWS squares double or triple the value of the word(s) that include those tiles.
- If any "hook" words are played (e.g. playing ANEROID while "hooking" the A to BETTING to make ABETTING), the scores for each word are added separately. This is common for "parallel" plays that make up to eight words in one turn.
- Premium squares apply only when newly placed tiles cover them. Any subsequent plays do not count those premium squares.
- If a player makes a play where the main word covers two DWS squares, the value of that word is doubled, then redoubled (i.e. 4× the word value). Similarly, if the main word covers two TWS squares, the value of that word is tripled, then retripled (9× the word value). Such plays are often referred to as "double-doubles" and "triple-triples" respectively. It is theoretically possible to achieve a play covering three TWS squares (a 27× word score), although this is extremely improbable without constructive setup and collaboration. Plays covering a DWS and a TWS simultaneously (6× the word value, or 18× if a DWS and two TWS squares are covered) are only possible if a player misses the center star on the first turn, and the play goes unchallenged (this is valid under North American tournament rules).
- Finally, if seven tiles have been laid on the board in one turn, known as a "bingo", after all of the words formed have been scored, 50 bonus points are added.

A large part of Scrabble's difficulty, from an AI perspective, comes from the uncertainty of the tile drawing and the other player's tile rack and moves. Adversarial search algorithms are often used in competitive games such as Chess and Checkers but are less effective at playing games with a large amount of uncertainty such as scrabble, given that the bot must work with incomplete information. New algorithms and data structures were developed for scrabble to more efficiently handle the uncertainty.

Another difficulty that arising when implementing a scrabble bot is the huge state space. The state space is largely determined by the size of the dictionary being used in the game, thus an obvious method of reducing the state space is to create a new dictionary containing a subset of the original dictionary. The approximate size of the state space would be:

$$|S| = 26^{d^2+r} \qquad (1)$$

With r being the number of tiles on the current players rack, usually 7, and d being the board dimensions. Calculating the exact size of the state space is difficult since word length would also determine where a word can be legally played.

AI game playing bots differ greatly in the strategies they implement compared to their human counterparts. The largest difference between the two is that a bot has the ability to search

through the entire action space, since it will know the entire dictionary, whereas it's unlikely that a human player will have the entire dictionary memorized.

The large action space creates a new problem for the bot, so a more effective method is to generate efficient data structures and search algorithms to handle this large action space. In this paper we will analyze the various algorithms and data structures that have been developed to effectively play scrabble and the history and effectiveness of bots that used these used them.

Figure 1: Scrabble Board

Rack

Figure 2: Scrabble Rack

Opponent Modeling [8]

One of the most difficult things involved with implementing a scrabble bot is accounting for the unpredictability of the opponent, along with the uncertainty caused by incomplete information. This includes: which tiles the opponent currently has, which tiles the opponent will draw, and which word the opponent will choose to play. Most competitive games use adversarial search algorithms to account for which move the opponent will make; however, this is difficult with scrabble given the incomplete information.

Richards and Amir, published a paper on opponent modelling. In order to create an accurate opponent model it is essential to create a predictive model which estimates which tiles are on

the opponent's rack. Two players were pitted against each other, one with full knowledge of the opponent's rack, the Full Knowledge Player, and another with no knowledge of the opponent's rack. After having them play 127 games, “the Full Knowledge Player has a great advantage, scoring 37 more points per game on average. The difference is highly statistically significant ($p < 10^{-5}$ using random permutation tests)” [8]. This emphasises the importance of an effective predictive model for the opponent’s rack. One of the most effective ways to estimate what tiles are on the opponent’s rack is to make inferences based on his most recent move. This can be done by considering every possible leave that an opponent can have (the tiles still on the rack after a play). From each leave reconstruct what the full rack would have been, generate the legal moves that could have resulted from the rack, then use Bayes’ theorem to estimate the probability that the opponent having that leave (2).

$$P(leave \mid play) = \frac{P(play \mid leave)\, P(leave)}{P(play)} \qquad (2)$$

This results in:

$$P(play) = \sum_{leave} P(play \mid leave)\, P(leave) \qquad (3)$$

Where $P(play \mid leave)$ is the predictive model being used for the opponent’s decision-making process. Mark Richards summarizes this:

> If we are given to know the letters that comprise the leave, then we can combine those letters with the tiles that we observed our opponent play to reconstruct the full rack that our opponent had when he played that move. After generating all possible legal plays for that rack on the actual board position, we must estimate the probability that our opponent would have chosen to make that particular play.

This is however done with the assumption that the opponent is a computer, so it will select the highest-ranking play. Yet, even against against a human opponent this probabilistic model will capture the opponent’s behavior in many important situations. Mark Richards tested this inference model against the Quackle bot (discussed below). The results showed that on average the Inference Bot scored 5.2 points per game more than the Quackle Bot, and won 18 more games, from a sample of 630 games.

	With Inferences	Quackle
Wins	324	306
Mean Score	427	422
Biggest Win	279	262

Table 1: Summary of results for 630 games between the Inference Bot and the Quackle Bot [8].

Maven and Quackle

History

In 1983, Brian Shepperd developed a scrabble agent which could beat human champions. However, this agent still had room for improvement. One of Shepherd's concerns was that the agent had not been implemented with the full Official Scrabble Player's Dictionary. In 1986, he published an agent named "Maven," which had a much more complete dictionary and that used a more advanced move generator. Maven had a "Perfect Level" which was predicted to have less than 1% error per game. Unfortunately this level was not well suited for computers of its time and would not be able to be used until 1996. When the perfect level first became available, it went on to beat 60% of human champions.

Scrabble Game Phases

Scrabble games can be categorized into three phases which are normal, pre-endgame, and end-game. Depending on which stage the game is in, Maven uses different algorithms accordingly. The component which handles this is the move generator; its design was meant to find all legal plays based on its current environment and calculate the potential point gain by each play. The move generator also has the duty to evaluate the rack and in order to evaluate future scores; and finally analyze the current position.

The Scrabble agent that Shepherd developed in 1983 used an algorithm called "bit-parallel." This algorithm takes the intersection of bit vectors containing constraints. The drawback with this algorithm is that it does not enable a feasible way of searching a state machine restricted by tiles on the rack, tiles on the board, and game rule constraints. A more efficient data structure came to the attention of Shepherd in 1988, where two other developers had published an algorithm which used a directed acyclic graph. This data structure is explained more in depth in a later section.

Rack Evaluation

One issue that potentially arises from maximizing an agent's points locally in Scrabble is that they may end up with a rack of undesirable tiles. In order to address this issue, during each turn, the agent must evaluate its own rack. In order to evaluate the rack, the author created a list of combinations of tiles; these combinations were made up by any possible combination that would affect the game. It is also important to mention that the rack evaluation component handles parameter fitting. It does this in order to predict future scores; the most recent method for parameter fitting is using temporal difference learning, describe in depth later in the paper.

Endgame

The phase at the end of the game is one in which every player has all the information at their disposal: every player would have the ability to figure out which game pieces are left based on their own and what's on the board. Based on this given information Maven was designed to work differently at the Endgame phase. The first priority that Maven takes into consideration is to not go out first before its opponent. The reasoning behind this priority is because at the end of each Scrabble game, each player gets a deduction of points from the tiles they have in their rack; furthermore, if a player has no tiles in their rack by the end of the game, they receive all of the deducted points from the other players. In the case of a 2 player match, the player which goes out first would be succeeding those points to their opponent, losing points, and handing an extra turn to their opponent.

Since the endgame phase is a fully observable environment, the first immediate suggestion would be to use the alpha-beta search algorithm. Although it is a viable option, it is ill-suited for Scrabble due to its branching factor and for a couple of other reasons. Due to this, Maven was implemented with the search algorithm B*.

B* Search Algorithm

While testing Maven, one of the desirable traits that Shepherd found was need to find the difference between the highest and the second highest scoring moves for the opponent. Shepherd believed that to find the true evaluation of each node, he need a pessimistic and an optimistic evaluation of each turn; which could be obtained from the evaluation function's analysis. Using those 2 evaluations as bounds, B* search could provide the best move.

In the case that the B* search fails, it rebuilds the search by expanding the interval by a small amount. This decreases the amount of evaluation errors produced by the evaluation. This strategy has been used during championships, whenever Maven is done with searching but there's still more time for its turn.

Data Structures [3][4]

Move generation has always been an active topic in Scrabble-playing agent in literature. Methods to generate all possible moves from a given a board state had historically been too large of a problem to conquer in a reasonable amount of time. This all changed in 1988 when Appel and Jacobson introduced a new heuristic based algorithm and lexicon representation that had the ability to outpace previous methods by several orders of magnitude. This new lexicon representation, referred to as a *DAWG*, laid the framework for a new generation of move generation algorithms that are still in use today.

Before Appel and Jacobson published The World's Fastest Scrabble Program in 1988 [3], introducing the dawg data structure, typical state-of-the-art scrabble algorithms parsed a tree like structure called a *trie* for move generation. This structure represents the search space as a tree whose edges are letters and marked terminal nodes as valid words in the lexicon stemming from the root. When two words begin the same way they share the initial parts of their path.

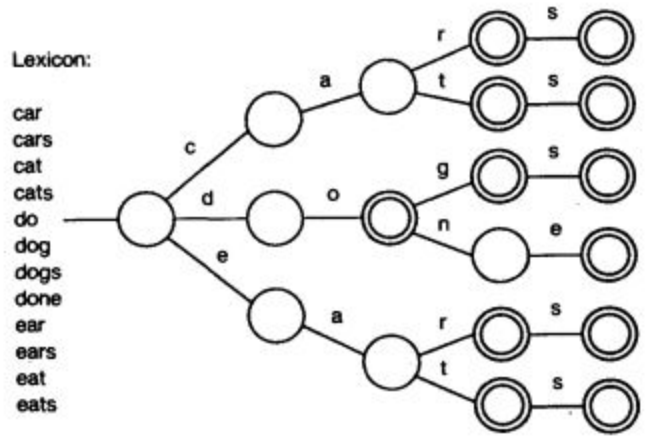

Figure 3. A lexicon and the Corresponding Trie with terminal nodes circled [3]

The DAWG data structure can be thought of as an extension to the idea that a trie lexicon, when reasoned about as the language of a finite-state machine, is a non-optimal representation in its space complexity. Even though there are many different finite-state recognizers for a language, there must exist one with a minimum number of states that is easy to find for a finite language [6]. Using this idea, a trie of the finite language shown in figure 3 can be reduced to a DAWG (Directed Acyclic Word-Graph) representation as shown in figure 2. This structure can basically be thought of as a trie where all equivalent sub-tries (corresponding to identical patterns of acceptable word-endings) have been merged. [3]. Reducing a 94,240-word lexicon from a trie to dawg representation, the authors were able to reduce the number of nodes from 117,150 to 19,853, an astounding 83% decrease in graph nodes required to represent the language. This reduction also drastically reduces memory space requirements by taking the 780 Kbyte raw world-list down to just 175 Kbytes, a size that was able to be reasonably held in memory by the VAX 11-780 "minicomputer" used in benchmarking.

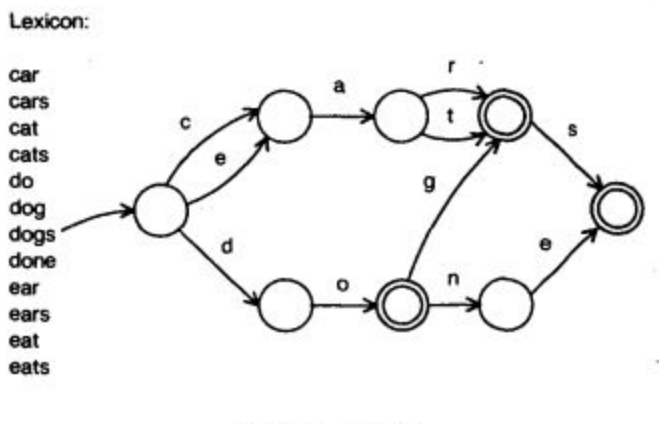

Figure 4. A Dawg structure [3]

Along with the DAWG representation, an algorithm was also presented by Appel and Jacobson which took advantage of the space-optimization properties of the structure. Using a backtracking traversal strategy along with the notion of an anchor piece, a newly placed tile adjacent to an existing tile in play which to root the search, the algorithm could exhaustively search the lexicon space with much higher performance than had previously been possible. The most popular commercial Scrabble agent at the time with a lexicon of 44,000 words, MONTY, took about two minutes per move, and it was said that while MONTY was a challenging opponent, most Scrabble experts could still beat it consistently. Another Scrabble-playing program from AT&T Bell Laboratories and headed by Peter Weinberger was able to traverse a 94,000 word lexicon in a minute or two. The new backtracking algorithm and DAWG structure presented by Appel and Jacobson could generate all legal moves from its 94,240 word lexicon in a mere one or two seconds.

The DAWG structure was revolutionary for lexicon representation because of its minimization of space requirements. However in 1994 a new data structure was proposed by Steven A. Gordon in his paper A Faster Scrabble Move Generation Algorithm. In this paper Gordon presents the GADDAG data structure along with a complementing algorithm that could generate moves twice as fast as the DAWG structure proposed by Appel and Jacobson just five years earlier.

Like DAWG, GADDAG is a specialization of a trie structure that reduces nodes into an optimized directed graph. However unlike the uni-directed graph structure of the DAWG, Gordon used the space/time tradeoff and an opportunistic view of decreasing memory costs to present GADDAG, a representation that contains bidirectional paths between nodes. GADDAG also differs in structure in that it stores every reversed prefix of every word in its lexicon. While this method requires a distinct representation of each word for each letter in the word, it allows for a much more efficient search during move generation since traversal of any word can be rooted from any letter that occurs in it.

	DAWG		GADDAG		Ratio (G/D)
	Unminimized	Minimized	Semi-minimized	Minimized	minimized)
States	55,503	17,856	250,924	89,031	4·99
Arcs	91,901	49,341	413,887	244,117	4·95
Letter sets	908	908	2575	2575	2·84
Expanded					
Bytes	5,775,944	1,860,656	27,110,092	9,625,648	5·17
Bits/char	91·6	29·5	429·8	152·6	
Compressed					
Bytes	593,248	272,420	2,669,544	1,342,892	4·93
Bits/char	9·4	4·3	42·3	21·3	

Table 2. Relative sizes of DAWG and GADDAG structures [4]

	DAWG		GADDAG		Ratio
	overall	per move	overall	per move	(D/G)
CPU time					
Expanded	9:32:44	1·344s	3:38:59	0·518s	2·60
Compressed	8:11:47	1·154s	3:26:51	0·489s	2·36
Page faults					
Expanded	6063		32,305		0·19
Compressed	1011		3120		0·32
Arcs Traversed	668,214,539	26,134	265,070,715	10,451	2·50
Per sec (compressed)	22,646		21,372		
Anchors used	3,222,746	126·04	1,946,163	76·73	1·64
Number of moves	25,569		25,363		
Average score	389·58		388·75		

Table 3. Relative performance of DAWG and GADDAG algorithms playing both sides of 1000 random games on a VAX4300 [4]

As mentioned, GADDAG stores every reversed prefix of every word in the dictionary resulting in every word having as many representations in the structure as it does letters. As an expected consequence of redundant representations, table 1 shows that the GADDAG structure introduces a significant cost in space requirements when compared to a DAWG structure of the same lexicon. Given the regulation Scrabble dictionary having an average word size of around five letters, it is not surprising that the space requirements of a GADDAG when compared to a DAWG of the same lexicon is about five times higher. Table 3 highlights the performance increase gained with a GADDAG representation, an over 50% decrease in both time per move and arc traversal.

Both DAWG and GADDAG data structures improve performance for Scrabble word generation algorithms when compared to traversing raw word lists or tree-like structures. However, performance in the game of Scrabble is not all about finding the maximum point value each turn for the tiles in player's hand. Blank tile pieces and drawing random replacement tiles after each turn introduces an aspect of imperfect information into the game. Letter and word multiplier tiles on the board also create an environment with inconsistent utility from a positioning standpoint. It is because of this that most of the recent work done on the game of Scrabble has not been

finding more performant lexicon representations, but finding better heuristic functions to aid in finding a more accurate utility function for a given game state.

	DAWG			GADDAG			Ratio (D/G)		
	Secs/ move	Arcs/ move	Page faults	Secs/ move	Arcs/ move	Page faults	Secs/ move	Arcs/ move	Page faults
Greedy vs. Greedy	1·154	26,134	1011	0·489	10,451	3120	2·36	2·50	0·32
RackH1 vs. Greedy	1·390	32,213	1013	0·606	12,612	3136	2·29	2·55	0·32
RackH2 vs. Greedy	1·448	33,573	1016	0·630	13,060	3139	2·30	2·57	0·32
RackH3 vs. Greedy	1·507	35,281	1017	0·655	13,783	3141	2·30	2·56	0·32

Table 4. Relative performance of DAWG and GADDAG algorithms aided by rack evaluation heuristics [4]

Table 4 shows the relative performance of both data structures when used in tangent with a heuristic based algorithm. Notice that a ratio of ~2.3 remains near static when each structure is used with different heuristic algorithms simulated against a greedy agent. This table helps support the argument that GADDAG is an all-around more performant structure given a choice between the two and enough memory to support the increase in size.

Advancements in Other Games

Advancements in AI in the area of deep reinforcement learning have been successful in games other than Scrabble. Play at or exceeding human levels was achieved early on in Backgammon with TD-Gammon and much more recently in Go with AlphaGo. A look into the methodology behind these successes may suggest new directions for improving Scrabble AI.

TD-Gammon [2]

Overview

TD-Gammon was published in 1995. It employs temporal difference learning on a multi-layer perceptron to learn a game state evaluation function for the game of backgammon. Backgammon is a game of perfect information, but there is uncertainty in the form of non-deterministic dice rolls. The dice rolls increase the size of the search tree for the game, as the legal moves at each turn are dictated by the dice.

A notable aspect of TD-Gammon is that it is able to perform well without knowledge from human backgammon expert play - in its first iteration, without any additional hand-crafted input features at all - instead, learning from an initially random evaluation function at the start of training. TD-Gammon performs at the level of its predecessor Neurogammon which is a MLP trained via supervised learning from games between human backgammon experts. That said, TD-Gammon is not an entirely zero-knowledge algorithm. Specifically, an ad-hoc algorithm to handle the

"doubling cube" aspect of the game is added in addition to the MLP. Furthermore, TD-Gammon ignores the rare 3-point "backgammon" game outcome for which the game is named.

Offline Training

The neural network leveraged by TD-Gammon is a 3-layer MLP with sigmoidal activation functions. The MLP accepts a game state as its input and predicts the outcome of the game based on that game state. Game outcomes are slightly more complex than win/lose, with two different types of win possible (1-point win or 2-point "gammon") for a total of four possible game outcomes, each a separate probability in the output vector.

The MLP is trained using temporal difference learning for the gradient update step. Training takes place over the course of many games of self-play.

w

$$w_{t+1} = w_t + \alpha(Y_{t+1} - Y_t) \text{ sum}\{k=1\}\{t\} \lambda^{t-k} \nabla_w Y_k$$

After each turn, t, of the game the weights, w, of the MLP are updated. The difference between the previous predicted outcome and the current predicted outcome is weight by the learning rate, α, and the weighted sum of the gradient of each previously predicted outcome. This is the core of temporal difference learning, where current estimates are updated based on prior estimates. The prior estimates are weighted in such a way that older estimates have less of an impact than more recent estimates. In other words, estimates from more recent turns of the game have more impact on the current update than estimates from the beginning of the game. The amount of impact that each of these updates has is determined by the "heuristic parameter", λ. The λ term controls the rate at which older estimates are discounted, decaying exponentially in the total number of turns.

Online Play

TD-Gammon plays using a standard search tree approach. The MLP is used to predict the action to take at each turn (node of the search tree). It does this by predicting the outcome of each legal move, using the previously trained MLP, and selecting the move with the highest expected outcome. That is, the weighted average of each of the games possible outcomes as there are more than two. In later iterations, the search algorithm searches an extra layer deeper into the search tree before consulting the MLP for move evaluation.

Remarks

TD-Gammon is over two decades old, but the methods it uses are still relevant today. There have since been a number of advances in machine learning that might improve the initial implementation. For example, the overall size and depth of TD-Gammon's neural network might be increased using modern hardware as well as rectified linear units in place of the traditional sigmoidal activation functions. Additionally, convolutional layers might be added to allow for additional automated feature extraction. Improved gradient ascent/descent methods might provide for more efficient optimization of the loss gradient. Since the dice rolls in backgammon

are uniformly random, the approach of focusing the tree search (as accomplished by AlphaGo) might not be as successful, as each turn allows for many uniform possibilities.

Alpha Go [1]

Overview

AlphaGo made its debut in 2015, where it defeated a human professional and various state-of-the-art Go programs at the game of Go. Unlike Scrabble, Go is a game of perfect information. In other words, there is no information hidden from the players about the state of the game, and there is no non-determinism (apart from the choices of the opponent). Go has a huge state space, larger than Scrabble and many orders of magnitude larger than chess. While Monte Carlo tree search (MCTS) can be effective at managing large branching factors, it doesn't provide the requisite predictive power on its own for super-human Go play. On the other hand, super-human Scrabble play has been achieved with MCTS.

AlphaGo supplements MCTS with deep neural networks to focus the MCTS on the most important branches of the search tree and to better evaluate the MCTS stopping nodes. The neural networks, themselves, are trained on games by human experts and games generated from self-play of the networks. AlphaGo is not a "from scratch" solution in the sense that it is trained on, and therefore relies on, human expertise.

There are four deep neural networks employed by AlphaGo:

- An "SL policy" network, p_σ, that performs action prediction, trained via supervised learning on expert human players
- A "fast policy" network, p_π, that is used during MCTS rollouts to quickly generate actions given games states at each step of the search tree
- An "RL policy" network, p_ρ, used to optimize the SL policy through self-play. This extends the SL policy from predicting human expert play to predicting actions that optimize chances of winning. The distinction being that expert human play is suboptimal.
- A "value" network, v_θ, that predicts the value of a given game state. In games of perfect information, the value of the game state determines which player will win. Thus, v_θ, generates estimates that are used directly in MCTS.

Offline training

The neural networks used by AlphaGo are trained offline. The networks are similar in topology, however some of their input and output layers differ.

The SL policy network, p_σ, is trained on games generated by human Go professionals. It represents a general human expert Go policy - which moves human players tend to play. The network is made up of 13 layers, including convolutional layers used to extract spatially-related features of the game board. The input to p_σ is a set of hand-crafted features representing a game state. These features include the positions and colors of the game pieces on the board as

well as pre-computed metrics about different aspects of the game, e.g., "how many opponent stones would be captured" at a given section of the board. The output of p_σ is a probability distribution over all legal actions that may be taken from the given input game state. This distribution is computed by a final softmax layer in the SL policy network. An individual action is predicted by p_σ by selecting the max action over the entire output distribution. This network is trained using stochastic gradient ascent. Notably, p_σ maintains only a 57% prediction accuracy against the training set hold-out.

The fast policy network, p_π, is a smaller network that functions similarly to p_σ. However, p_π is faster and less accurate as a result. The fast policy network has around half the accuracy of the SL policy network, but it is around an order of magnitude faster. The set of inputs that p_π deals with are more concise, high-level generalizations of the game state. These inputs end up being more locally-focused, encoding common patterns and arrangements of game pieces.

The RL policy network, p_ρ, shares the same network topology as p_σ, and it starts with initial weights taken directly from p_σ. The purpose of the RL policy network is to improve upon the SL policy network, moving from a policy that approximates human expert play to a policy that approximates optimal play. Training of p_ρ is performed via self-play. The current iteration of p_ρ is matched against a previous iteration (to avoid overfitting). While p_ρ is also trained via SGA, it is trained after each turn of the self-play game so that the policy is updated iteratively. After training, p_ρ, has more predictive power than p_σ.

The value network, v_θ, also has a similar topology to p_ρ and p_σ with the exception that the output layer consists of a single node rather than a node for each legal action. This output node predicts whether or not a player will win given a game state input, as v_θ is used for game state evaluation. Importantly, the value network makes its predictions under the assumption that both players are playing according to a particular policy. In the case of AlphaGo, that policy is p_ρ. That is, the value network is trained on games generated by self-play from p_ρ, and so the value network encodes the value of game states when both players are playing according to this approximate optimal policy. Training of v_θ is done via supervised learning on state-outcome pairs rather than on entire games. This is done to avoid overfitting. The network does not generalize well on full game data due to the high correlation between game states in a given game. SGD is used to train the network using the mean squared error of the predicted outcome.

Online play

AlphaGo chooses its moves using Monte Carlo tree search since the breadth of the Go search tree is prohibitively large (approximately 250 legal moves per turn). The MCTS algorithm works by simulating games, selecting the highest valued action at each step. Once a stopping node is reached, the SL policy network, p_σ, is used to compute predictions for each action at the stopping node (The SL policy is used, counter-intuitively, because it performs better in practice than the RL policy.) The stopping node is then evaluated using the value network, v_θ, which is highly efficient compared to Monte Carlo rollouts using any of the policy networks including p_π.

However, p_π is used in addition to perform a MCTS rollout to the bottom of the search tree. The results of v_θ and p_π are then combined to give an evaluation of the stopping state that is more predictive than either metric on its own. The probabilities of the actions at the stopping state and the value of the stopping state are used to update stored averaged values of the edges (actions) in the search tree which are then used by future simulations. After a number of simulations, the action is chosen that was most valuable across all simulations.

Remarks

AlphaGo's MCTS is much more computationally expensive than traditional MCTS because of the overhead of the neural networks to guide the MCTS and to evaluate the MCTS stopping nodes. However, this combination of using the policy network to focus and guide the MCTS, and using the value and fast policy networks to evaluate the chosen actions, results in more accurate predictions than standard MCTS. While AlphaGo has proven capable of super-human play, it is still a method bootstrapped from human play which leaves a desire for a "from scratch" method in the future. The remarkable success of AlphaGo's pairing of deep neural networks with MCTS suggests an additional approach to try for future Scrabble AI.

References

1. Alpha Go
   https://storage.googleapis.com/deepmind-media/alphago/AlphaGoNaturePaper.pdf
2. TD-Gammon
   http://www.bkgm.com/articles/tesauro/tdl.html
3. DAWG
   https://pdfs.semanticscholar.org/da31/cb24574f7c881a5dbf008e52aac7048c9d9c.pdf
4. GADDAG
   http://ericsink.com/downloads/faster-scrabble-gordon.pdf
5. Position Heaps
   http://www.sciencedirect.com/science/article/pii/S1570866710000535
6. Balancing Binary Trees
   http://dl.acm.org/citation.cfm?id=358509
7. World championship caliber scrabble
   http://www.sciencedirect.com/science/article/pii/S0004370201001667#
8. Opponent modeling in Scrabble
   https://www.aaai.org/Papers/IJCAI/2007/IJCAI07-239.pdf
9. Hasbro Scrabble Rules
   https://scrabble.hasbro.com/en-us/rules