In D.N.L. Levy and D.F. Beal, editors, *Heuristic Programming in Artificial Intelligence 2 – The Second Computer Olympiad*. Ellis Horwood, 1991.

Exploratory Learning in the Game of *GO*

Barney Pell[1]
University of Cambridge
Cambridge UK
E-mail: bdp@cl.cam.ac.uk

Abstract

This paper considers the importance of exploration to game-playing programs which learn by playing against opponents. The central question is whether a learning program should play the move which offers the best chance of winning the present game, or if it should play the move which has the best chance of providing useful information for future games. An approach to addressing this question is developed using probability theory, and then implemented in two different learning methods. Initial experiments in the game of **Go** suggest that a program which takes exploration into account can learn better against a knowledgeable opponent than a program which does not.

1 Introduction

One of the earliest aspirations of Artificial Intelligence was to develop computer game playing programs which could improve their play through experience, adapt their strategy to compete against a variety of opponents, and ultimately outplay their programmers. As in most learning problems, a program

[1]Parts of this work have been supported by RIACS, NASA Ames Research Center [FIA], and a British Marshall Scholarship.

attempting to learn to play a game can be trained by a variety of methods, each of which varies in the amount of reinforcement provided to the program.

At one extreme, we give the program a set of features we consider relevant to good play, pit the program against a copy of itself or another opponent, and finally tell the program only the outcome of the game ([Sam59, MC68, Sut84]). In this situation, the program has to decide for itself which moves contributed, and to what extent, to the ultimate outcome (a win or loss) of the game. Although this *credit assignment* problem can be very difficult, the advantage is that the program can in theory accommodate itself to drastic changes in the strategy of is opponents, and we don't need a large body of expert games as training material. This method, which we shall call the *learning by playing* approach, is particularly necessary for new games, for which no such data exists [2].

At the other extreme, if we do have large amounts of training data available, we can spoon feed this to the program before it ever plays a game, and let the program learn a classification function which distinguishes winning from losing positions in expert games. With enough data, and provided that the experts playing the training games are better than the opponents one is likely to encounter, this approach can be very successful ([Sam67, LM88, TS89]). The use of expert games can effectively overcome the credit assignment problem discussed above [3]. However, after completing its training, the final classification function is fixed. The resulting program thus becomes inflexible, suggesting that its evaluation function might be exploited by adaptable future opponents. So while this is a good method of initially *tuning parameters* for a thereafter non-adaptive program, this approach does not provide a reasonable architecture for a sustained learning system.

Thus, if we want a program which can continue learning, even as its opponents adjust to its strategies, we cannot use the *pattern classification* approach described above. Instead, in spite of the credit assignment problems, we must have a program which *learns by playing*. With this in mind, there are several problems which immediately present themselves:

Credit Assignment: How does the program determine which moves contributed to the ultimate outcome of the games, when the only real information it is given is that the entire sequence of moves played by the program and its opponent led to a particular outcome. How does lookahead help this problem? Given the game-playing context, what information should the program actually take into account during learning? See

[2] For games in which no expert exists, there is an important question of where the features come from. The present paper does not address this question here, although it is the subject of current research by the author ([Pel90]). For work on learning new features in games, see ([Fla89, Fla90, YSUB90, Hee85, Mug87])

[3] As argued in ([LM88]), experts playing against other experts tend to win from good positions, so a position which tends to show up on the losing side in expert games is probably a losing position.

([LM88, Sam59, Sam67, CF82]) for discussions of credit assignment in game-playing programs.

Opponent/Self Modeling: As the program learns and its strategies change, much of its knowledge becomes outdated, since a type of move which didn't work well for it with an old strategy may suddenly become a winning play if other aspects of its strategy change. Similar problems arise when the program is playing against different opponents. The key issue here is that all of the program's beliefs about the goodness of its moves should be somehow dependent on a model of its own and its opponents' strategies, both of which may change from game to game. See ([Sel89]) for a discussion of simple game-playing programs which adapt to changes in the strategies of their opponents.

Winning vs. Exploring: Should the program play those moves which give it the best chance of winning the present game, those which have the best chance of providing new information which may help it play better in future games, or some combination of the two? If we neglect this consideration, and have the program always play moves which seem to give the best chance of winning this game (given its present knowledge), this may dramatically slow down its learning rate. Worse, it may steer the program into a locally optimal strategy from which it may never improve.

This last issue, that of the relative value of playing moves which increase our chance of winning *this* game, versus those which increase our chance of winning *future* games, is the main topic of the present paper. Although the issue of exploration in learning has been discussed in the context of robotics ([Moo90, Kae90]) and problem-solving ([SS89]), this issue does not appear to have received much attention in the context game-learning. This is surprising, because it may be more critical here than these other domains, as the presence of an active opponent makes game-learning a particularly difficult problem ([Mic86]).

In what follows, we shall assume that the program is playing against a fixed opponent, it receives no reinforcement except the ultimate outcome of the game, and is only capable of looking 1-ply ahead, to choose among all moves available in a current position.

Although this last consideration is certainly a gross restriction on a learning system, it simplifies the learning task: the only feedback the program receives is the outcome of its moves at the end of the game, and we want to see if the evaluation function that the program learns can use this feedback to improve play. Using deeper search as additional feedback, as was done in ([Sam59]), is an interesting idea which should be incorporated into the current approach. For an argument that restricting programs to 1-ply search provides a more accurate test of evaluation functions, see ([Abr90]). For a discussion of programs using the value of information in deeper search contexts, see ([HM89, RW89, Goo68]).

In the next section, we outline the learning task considered here. Then we propose a learning method which may be better suited to this task than those which ignore the value of information. Then we present initial results of experiments using this new learning method in the game of **Go**. We finish with a discussion of these results and directions for future research.

2 The Learning Task

The simplified form of learning we consider here is where we provide the learning program with a set of discrete *features* that it can use to describe *moves* in the game [4]. We assume that those moves which take on the same values for all features are thus indistinguishable. The resulting space created by a particular feature set is called a *feature space*, and is essentially a single-level version of the *signature tables* data structure used in Samuel's later experiments ([Sam67]). With each point p in feature-space (describing a set of moves) we can associate two numbers: w and l, corresponding to the number of times that a move in p has been part of a winning or losing game, respectively [5].

To make a move in a given position, the program considers each legal move it can make in this position. Each such move is mapped into feature space, to retrieve the values w and l, and the program assigns some evaluation to this considered move, which is solely a function of these two values (in particular, we assume that the value of this move is independent of the w and l values of alternative moves). Finally, the move which receives the highest evaluation is played, and the game continues. At the end of the game, the w counts of the points corresponding to moves played by the winner are incremented, as are the l counts for those of the loser.

Given just this data, the goal of this learning task is for the program to win as many games as possible out of a pre-specified match length, playing against a specific opponent.

2.1 Effects of Learning Methods on Match Trajectories

Now, considering that the program is not allowed to search beyond the set of currently available moves, and that its information about each available move is restricted to the empirical success counts w and l associated with each move, it is clear that all of its success will be attributable to the function which assigns values to points in feature space.

[4] Earlier evaluation-function learning systems ([LM88, Sam59, Sam67]) learn to evaluate *board positions*. However, many **Go** programs ([Ryd71, Shi89]) evaluate local features of the *moves*, or changes between positions. Although these state-based and operator-based approaches are in some sense equivalent, we have chosen to follow the operator-based convention in **Go** programs, and thus learn to evaluate *moves*.

[5] Assuming a discrete feature space is clearly simplistic. I am currently considering possibilities to eliminate this assumption. An incremental version of the Bayesian classification scheme developed in ([LM88]) may be appropriate for this purpose.

The obvious effect of different evaluation functions is their relative impact on the outcome of the current game. Two programs which are provided with different functions will likely choose different moves from the same position, even when they have had exactly the same experiences. But of even more importance is the less obvious effect of different evaluation functions: the moves a program plays during *this* game will change its total knowledge (the w and l counts for the moves played this game), which will influence the choices it will even be able to consider in *future* games.

Unfortunately, precisely quantifying this insight, about the relative value of winning this game versus future games, is extremely difficult. For a complete analysis, a program would need to consider the possible impacts of its current decisions on many possible future states of knowledge. It would also have to decide how likely it is to be in a better position to win future games provided that it played one move instead of another, and weigh this by some utility function describing the number of games left to play. If this is an early game in the match, the program knows very little anyway, so it might as well try to gain as much information as possible. If this is the last game, the program should just try to win, as after this game it will not matter anymore. But in the middle lies a difficult decision problem whose solution may be vital if we want to build really promising game-learning programs.

Having established the difficulty of the general problem, we now discuss an initial approach to making this tradeoff, where we derive an evaluator for points in feature space using probability theory.

2.2 Deriving the Incomplete Beta Function

The current problem, then, is to define a function which takes as input a set of moves m_i, each with a pair of corresponding statistics which indicate that this class of move has been been part of w_i winning games, and l_i losing games. The function assigns a value to each move, and then returns the one with the highest value. How should we define this function, in order to optimize our chances of winning as many as possible of the next n games?

As discussed in ([LM88]), the optimal choice if we are considering only the outcome of the *current* game and are also provided with highly accurate data, is to play the move with the highest *mean probability of winning*. In the current [6] decision problem, this would translate to choosing the move m_i which maximizes:

$$\frac{w_i + 1}{w_i + l_i + 2} \tag{1}$$

This is essentially the decision procedure used by Samuel for choosing moves using signature tables based on correlation with book moves ([Sam67]). However, as discussed in the Introduction, we are here considering a learning

[6] This assumes a prior in which all moves are assumed to have a 0.5 chance of winning, in the absence of any experience with them (when w and l are zero). If we have reason to assign different priors, these can be incorporated into this formula.

system which obtains its experience through its own play, in which case its statistics are likely to be less accurate than those obtained through book learning. Thus, there is also more room for exploration, in the hopes of improving chances for success in future games.

In this case, we may not actually want the move with the highest *mean probability of winning*. Instead, it might be beneficial to choose the move which has the best chance of really being a winning move. I shall now formalize this intuition.

Given m, w, and l (subscripts omitted for clarity), we imagine that there is some *true* probability of winning by playing m, call it θ_m (or θ), which we wish to estimate using w and l. From Bayes's theorem, we have:

$$p(\theta\,|\,wl) = \frac{p(\theta)\,p(wl\,|\,\theta)}{p(wl)}. \tag{2}$$

In words, the posterior probability that the true mean probability of m being part of a winning game is really θ, is equal to the prior probability of any true mean being θ, times the probability that we would have seen w wins and l losses from a move whose true mean was really θ, divided by the probability of having seen w wins and l losses from any kind of move at all [7].

Since we have assumed that all moves we will be comparing will have the same prior probability of winning, or $p(\theta)$, we can eliminate this term from the above equation. Now, given just θ, w, and l, and the assumption that the θ_i for each move class is independent, it follows that the posterior mean lies on a binomial distribution:

$$p(wl\,|\,\theta) = \mathrm{B}(\theta, w, l) = (1-\theta)^{l+1}\theta^{w+1} \tag{3}$$

In this case, the probability of observing w and l is just the integral of the above expression for all possible means between 0 and 1, which is the definition of the *Beta distribution*:

$$p(wl) = \mathrm{Beta}(w, l) = \int_0^1 dt\,(1-t)^{1+l}t^{1+w} \tag{4}$$

Substituting back into the original equation gives us the final probability distribution of the true mean winning chances, given just w and l:

$$p(\theta\,|\,wl) = \frac{(1-\theta)^{l+1}\theta^{w+1}}{\int_0^1 dt\,(1-t)^{1+l}t^{1+w}} \tag{5}$$

For a visual representation of this formula, Fig. 1 plots this posterior distribution on the true mean for two moves. The darker curve corresponds to a class of moves which has been played in 3 games, all of which have ended up losing for the side who played it. The lighter curve corresponds to a move which won 3 times and lost 11 times.

[7] All this talk about probabilities of probabilities may be a bit confusing. Basically, we have some idea that the move m in question will tend to win some fraction θ_m of the time. This fraction θ_m is the variable whose distribution we are calculating, given w_m and l_m.

Figure 1: The distributions of the true mean probability of winning, for two moves with with $wins{:}losses$ = 0:3 and 3:11, respectively.

Now, careful examination of the curves for these two candidate moves reveals the tradeoff with which our evaluation function is faced. The darker curve, corresponding to the move with 0 wins and 3 losses, has a mean probability of winning 0.2, while the lighter curve, corresponding to the move with 3 wins and 11 losses, has a mean of 0.25. If we are faced with a position in which we must choose between these two moves, we are evidently unlikely to win the game under either choice. However, as we have had less experience with the first move, we see from the graph that it has a *much* better chance of being a truly good move than does the first curve, if we consider a *good* move as one which, when we have finished the match, will have had a better than 0.5 percent success rate. Another way of expressing this is as follows: we are reasonably certain that the true mean of the move in the lighter curve, which we have seen 14 times, is close to 0.25. However, although its empirical mean is lower, the move we have seen 3 times could conceivably have a true mean much greater than the observed mean of 0.2.

To use this insight as the basis for a new evaluation function, let X be a number between 0 and 1, which we shall call a *fickleness parameter* [8]. Now, instead of choosing the move with the highest mean, we can define a function which chooses a move which has the greatest probability that the *true mean*, θ, is greater than X. Analytically, this will just be the incomplete integral of

[8] Alternatively, X can be considered a *conservatism parameter*, according to the degree of one's optimism.

Figure 2: The probability that the true mean probability of winning (θ) is greater than X, for the two moves whose distributions were plotted above. Note that this graph shows the area under the curves of Fig. 1 to the right of X.

the Beta function, to the right of X:

$$p(\theta > \mathrm{X} \,|\, wl) = 1 - \mathrm{I}_x(w, l) = 1 - \int_0^x dt \, \frac{(1-t)^{1+l} t^{1+w}}{\mathrm{Beta}(1+w, 1+l)} \qquad (6)$$

Fig. 2 displays graphs of the *Incomplete Beta* functions corresponding to the two moves considered above. As the graphs clearly show, the move which is less favored according to the *highest-mean* criterion is preferred by the *Incomplete Beta* criterion for $X > 0.38$. This explains why we can consider X here to be a fickleness parameter: as X increases, our learning program will be increasingly picky about the moves it plays, favoring those which have the potential to reveal themselves as truly good moves in later games.

3 Exploratory Learning in Go

So far, we have motivated the intuitive appeal of using an alternative to choosing the highest mean, and have derived a suitable alternative function, the Incomplete Beta function, directly from probability theory. However, as discussed earlier, it is not yet clear how to precisely formulate the decision problem of maximizing expected wins over a certain number of games. We have described a *good move* as one with a high value on the incomplete beta distribution given a particular choice of the fickleness parameter, X. This number at present seems fairly arbitrary, and a goal of further research is to tie the variation in the fickleness parameter directly to the ideal decision procedure for the learning problem. In the learning experiments reported here, X was set at 0.5.

Meanwhile, it is important to test whether the notions of exploratoriness and utility of information for future games can actually improve upon the default, *highest-mean* decision rule. To this end, we conducted experiments in which we played both learning methods, henceforth referred to as *BETA* and *HIGHEST-MEAN*, against a fixed, knowledge-engineered program. To emphasize the role of the evaluation function in determining playing strength, the knowledgeable player was only provided with the same set of features as the learning systems, and no system could look ahead beyond the set of legal moves in each position. The two questions of interest were:

1. Which learning method would win more games in a match of pre-specified length (100 games)?
2. How would the performance of the learning methods compare to that of the fixed opponent, in terms of games won?

3.1 Go as a Domain for Machine Learning

The experiments were conducted using the game of **Go**, as part of a long-term research project in Machine Learning with the game of **Go** as the testbed ([Pel90]). **Go** is a good domain for these experiments, for at least three reasons. First, as discussed in ([Gre83]), experiments can be conducted first on small boards, and if successful, they can then be tried out on progressively larger boards, without having to change the rules or underlying representation. This provides a range of difficulty going from just harder than naughts and crosses (on a 3x3 board) to possibly the most difficult game actually played by humans (on a 19x19 board). Second, traditional methods of game-tree search, which provide the basis for many studies in machine learning ([Sam59, Sam67, LM88]), appear to be of little use in analyzing the bushy search trees prevalent in **Go** ([Ryd71]). Furthermore, one typically needs to search several plies deep to discover even the simplest tactical features (like captures and connections), and occasionally upwards of 30 plies for other basic features (e.g., ladders) ([KCN90]). So the possibility of programs which learn which moves to consider further could greatly improve the performance of computer go programs. And finally, some successful new go programs have benefitted from a representation of knowledge which expresses priorities for moves on the basis of a set of features ([Shi89]). This is similar to the form of knowledge which is produced by the learning methods considered here.

3.2 A Simple Go Feature Space

The feature space used in these experiments was extreemely small, to increase the chance that particular classes of moves would be encountered more than once during the learning match. Research is now in progress to determine a way of extending the size of the feature space by using multivariate regression. But as the main topic of this paper was exploration, the experiments using a small feature space provide a useful simplification of the learning task (see the section on limitations and future research for more on extending the feature space.

The features used were a minimal set in which it was possible to encode such critical notions as attack, defense, and connections as encountered on small boards. It is important to note that the use of *move-oriented* features instead of *position-oriented* features allowed a concise representation of these critical concepts, although it may be possible to represent them as well using *state-oriented* feature set. The features, with their possible values and their meanings, were as follows:

Capture? True if the move captures an enemy group, False otherwise.

Edge? True if the move is on played on the edge of the board.

Friend-Strength The number of liberties of the weakest friendly group adjacent to the point where the move was played, evaluated *before* the move. Values: 0 (no friendly neighbors), 1, 2, *3 or More.*

Enemy-Strength The number of liberties of the weakest enemy group adjacent to the point where the move was played, evaluated *before* the move. Values: 0, 1, 2, *3 or More.*

Group-Strength The number of liberties of the resulting group that the move creates, joins with, or merges together. Values: 1,2, *3 or More.*

Merge-Count The number of friendly groups which the move merges together. Values: 0 (creates new group) 1 (extends), 2, 3, 4 (merges).

3.3 Go-specific Issues

This section briefly discusses some details relevant to learning experiments in the game of **Go**.

3.3.1 Passing Criteria

Perhaps the major difficulty in designing a program which begins as a random player, as do the programs here, involves representing a termination condition. The rules of **Go** allow a player to pass at any time, or to continue playing until there is no legal move available. If we just let them play until there is no legal move left, the programs wind up filling in their own eyes, even if they have learned that this decreases their chance of winning, since they have nothing else to do and they cannot pass yet. Until I develop a passing criterion (like: continue playing unless the chance of wining is lower than a certain threshold), I have prevented the programs from playing in a space completely surrounded by a single friendly string. As all programs used in these experiments play with this same constraint (even the random player), this simplification does not detract from the results presented here.

3.3.2 Predicting Final Score Difference

The other major issue I encountered concerned the value to be predicted by the evaluation function. In many games, the only outcome of interest is whether a player won, lost, or drew the game. But in **Go**, we are also concerned with the final score difference, and to be more correct, this is what program should really be predicting. In these initial experiments, I have only provided the program with a knowledge of whether it won or lost the game, so it has no notion of how *close* it came to actually winning. In future experiments, I hope to address this problem and have the programs learn to predict score differences.

3.4 A Hand-Tailored Evaluation Function

Although this feature-space is relatively small (fewer than 900 distinct classes of reachable moves), the features provide a language which is descriptive enough to build a very reasonable small-board **Go**-player. The purely local features do not pose much of a problem until the board size exceeds 9x9 (although this is the most difficult case for this feature set). The resulting program is strong enough to play perfectly on 3x3 boards, and to beat a random player consistently on virtually every board size. However, the lack of lookahead and the small feature space also mean that the program cannot play perfectly (even on small boards beyond 3x3), and will occasionally play losing moves which it could not distinguish from other winning moves, because the features map both moves into the same move class.

3.4.1 Examples of Hand-Coded Knowledge

I will here provide a few examples of the knowledge which is encoded in the customized function, from which the reader can get a feel for the representational system I used.

The evaluation function increments or decrements a score total based on the presence or absence of certain combinations of features of the move being evaluated. Some move adjustment routines are:

Free Captures are Very Good: If *capture* **and** *group-strength* $>$ **1, then increment score by 200**.

Figure 3: The output of the knowledgeable player's evaluation for the most recently played move on a 9x9 **Go** board.

Playing on the edge is Bad: If *edge* **then decrement score by 30**.

Opposing Strength is Good: If *group-strength* $=$ **3 and** *enemy-strength* $=$ **3, then increment score by 40**.

Damezumari without Capture is Bad: If *group-strength* $=$ **1 and not** *capture* **then decrement score by 100**.

Threaten weak enemy groups

Defend weak friend groups

Fig. 3 displays explanatory output of the hand-coded evaluation function as it evaluates the most recently played move on a 9x9 **Go** board (output provided by the program).

3.4.2 The Knowledge Bottleneck

An interesting issue which developed while implementing this evaluation function is that this representation, while fully expressive (since it allows higher-order interaction of features), is difficult to utilize fully. It is extremely difficult to decide how features interact on higher levels, as is necessary to answer the question: is it better to play a capturing move on the edge while joining two friendly groups together, or to play an attacking move in the center which also leaves one's own group somewhat weakened? The fact that both learning methods managed to obtain empirical answers to these questions (with respect to their own total strategies, of course) is indicative of the potential advantages of automated exploration in learning systems.

In any case, the fact that the knowledgeable program plays so well means that the results reported below, in which a learning system begins as a random player with no knowledge of how to use its features, and learns to win games against this knowledgeable program approaching half the time, are promising initial results.

4 Experimental Results

Fig. 4 shows the results of representative 100-game matches on a 9x9 **Go** board. The dark curve plots the cumulative wins for the BETA learning method, with fickleness parameter set to 0.5, against the customized program, and the light curve plots cumulative wins for the HIGHEST-MEAN learning method against the customized program. In both matches, the programs played alternate colors each game.

There are several interesting aspects to this graph. Perhaps the most surprising is that the BETA learner actually begins to win games after only 2 losses. Although there was certainly an element of chance in this (since the features don't capture all the valuable distinctions), this result also shows definite learning, since the chance of a *random* player winning a game against the customized program is extremely small [9].

Another interesting point is that both learning methods do improve with the number of games, but that they do not appear to significantly surpass the customized player, in terms of percentages of games won, on 9x9 boards. On smaller boards, the BETA method fares substantially better than the HIGHEST-MEAN method, and on some runs it reaches a point where it sustains a higher than even success-rate against the customized player, indicating that it has exceeded the abilities of its opponent. Unfortunately, there are also runs where it alternates between streaks of excellent victories, followed by periods of equally excellent blunders. This point is pursued in the Discussion section which follows.

The final and most obvious feature of the graph is that even on 9x9 boards (with the highest level of feature interaction of any small boards), the BETA method dominates the HIGHEST-MEAN, over the course of 100 games. Although these experiments need to be replicated and tested for significance, they do provide support for the claim which was the main point of this paper: playing the move most likely to lead to a win *this* game may not be as valuable for a learning system as playing a move which provides information that will be useful in *future* games.

[9] On a 9x9 board, the random player lost all of 100 games against the customized program.

Figure 4: Cumulative win record for the BETA and HIGHEST-MEAN learning methods, in a 100-game match each against the hand-tailored **Go** evaluation function, playing on a 9x9 board.

5 Discussion of Experimental Results

In connection with the experimental results presented above, we noted the peculiar behavior of the BETA learning method, in which it follows a series of excellent successes with a run of horrible blunders. Although the full reasons for this behavior are the subject of ongoing research, one hypothesis is that this behavior results from locking the fickleness parameter used in these experiments at 0.5. Eventually, all really good moves become equally likely to have true mean probabilities above 0.5, at which point the learning system can no longer distinguish them. If this analysis is correct, then future experiments, in which the fickleness parameter will vary as an integrated component of the decision-theoretic reasoner, may produce a program which can continue to improve without these setbacks.

6 Limitations of the Proposed Learning Methods

In order to extend the utility of the methods discussed here, there are at least four important issues which must be addressed:

Extended Search. The current programs looked only 1-ply ahead. While this simplification is useful for our initial experiments, it is clearly unrealistic for a good game-playing system. The approach taken in ([LM88]) was to learn an evaluation function first, and then use this as a part of a full system performing minimax search. Samuel's early experiments ([Sam59]) used the results of deeper lookahead as feedback to the learning system. In some form, both methods will be necessary for a large-scale game-learning system.

Independence of Alternative Moves. In this paper, all programs have been forced to evaluate particular moves independent of a knowledge of available alternatives. That is, we give the program a set of features corresponding to a move (for example, that the move is a capture, not on the edge, and defends a friendly group), and the program returns a value saying how good that move is. But the actual goodness of this move may depend very much on what other moves are available. For example, captures in **Go** are almost always played more by the winner than the loser, yet this does not mean that one should play the capture as soon as it is available.

Size and discreteness of the present feature space. As the number and precision of our features increases, the discrete feature-space will grow exponentially, causing problems for storage and learning. While we would like to incorporate the Bayesian classification function approach developed in ([LM88]), this is not appropriate for a system which continues to learn as it plays. Some possibly *incremental* version of this approach may be appropriate, provided that it also provides a measure of *variance* necessary for determining the value of exploration, as developed here.

Choosing the Fickleness Parameter. In these initial experiments, the fickleness parameter was fixed at 0.5. Ideally, we would like the value of this parameter to be determined as a component of a decision-theoretic reasoner. The tricky part is that the strategies of the opponent, as well as those of the learning program, can change from game to game, yet the learning system must somehow use its past, possibly outdated experience to make decisions for the present game. The key question for this purpose is: what exactly is it that the learning system is really trying to predict? This paper has taken a step in this direction, but a more complete answer to this question, in the context of a game-playing system which learns by playing, is an important area for future research.

7 Conclusion

As noted throughout the paper, the results presented here represent the early stages of a long-term research project, and thus should be interpreted as a place-holder for later developments, rather than as a general learning method which can improve existing programs (at this stage).

The primary contribution of this work has been to apply the notion of the importance of exploration to the context of programs which learn by playing, to provide an initial framework, using probability theory, for addressing the subtle issues involved in this exploration, and to present initial results which suggest that a program which takes exploration into account can learn better than a program which does not.

8 Acknowledgements

I want to thank Peter Cheeseman, who has been my constant mentor and inspiration, and a co-creator of the Bayesian Go-Playing Project. Andy Moore, Alex Scott, and Mark Torrance influenced this work in too many ways to describe. Will Taylor designed an excellent graphic interface. Finally, thanks to Andy Mayer, Othar Hansson, Wray Buntine, John Stutz, Robin Hanson, Bob Kanefsky, Troy Anderson, and Michael Ginn, and to my supervisor, Steve Pulman.

References

[Abr90] Bruce Abramson. Expected-outcome: A general model of static evaluation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(2), February 1990.

[CF82] P. R. Cohen and E.A. Feigenbaum, editors. *Handbook of Artificial Intelligence*, volume 3. William Kaufman, 1982. Samuel's Checkers Player discussed on pp. 457–464.

[Fla89] Nicholas S. Flann. Learning appropriate abstractions for planning in formation problems. In *Proceedings of the Sixth International Workshop on Machine Learning*, pages 235–239, 1989.

[Fla90] Nicholas S. Flann. Applying abstraction and simplification to learn in intractable domains. In *Proceedings of the International Machine Learning Conference*, pages 235–239, 1990.

[Goo68] I. J. Good. A five-year plan for automatic chess. In E. Dale and D. Michie, editors, *Machine Intelligence 2*, pages 89–118. Oliver and Boyd, 1968.

[Gre83] H. S. Green. Go and artificial intelligence. In M. A. Bramer, editor, *Computer Game-Playing: theory and practice*, chapter 9. Ellis–Horwood Publishers, 1983.

[Hee85] Albrecht Heefer. Automated acquisition of concepts for the description of middle-game positions in chess. Technical Report TIRM-84-005, The Turing Institute, 1985.

[HM89] Othar Hansson and Andrew Mayer. Decision-theoretic control of search in BPS. In *Proceedings of the AAAI Symposium on AI and Limited Rationality*, 1989.

[Kae90] Leslie Pack Kaelbling. Learning functions in k-DNF from reinforcement. In *Proceedings of the Seventh International Workshop on Machine Learning*, pages 162–169. Morgan Kaufman, 1990.

[KCN90] A. Kierulf, K. Chen, and J. Nievergelt. Smart Game Board and Go explorer: A case study in software and knowledge engineering. *Communications of the ACM*, 33(2), February 1990.

[LM88] Kai-Fu Lee and Sanjoy Mahajan. A pattern classification approach to evaluation function learning. *Artificial Intelligence*, 36:1–25, 1988.

[MC68] D. Michie and R.A. Chambers. Boxes: an experiment in adaptive control. In E. Dale and D. Michie, editors, *Machine Intelligence 2*, pages 137–152. Oliver and Boyd, 1968.

[Mic86] Donald Michie. Puzzle-learning vs. game-learning in studies of behavior. In *On Machine Intelligence.* Addison–Wesley Publishers, 1986.

[Moo90] Andrew W. Moore. Acquisition of dynamic control knowledge for a robotic manipulator. In *Proceedings of the Seventh International Workshop on Machine Learning*. Morgan Kaufman, 1990.

[Mug87] Stephen Muggleton. Structuring knowledge by asking questions. In I. Bratko and N. Lavrac, editors, *Progress in Machine Learning*, pages 218–229. Sigma, 1987.

[Pel90] Barney Pell. Machine learning in the game of go. Unpublished Thesis Proposal, University of Cambridge, June 1990.

[RW89] Stuart Russell and Eric Wefald. On optimal game-tree search using rational meta-reasoning. In *11th IJCAI*, pages 334–340, 1989.

[Ryd71] J. Ryder. *Heuristic Analysis of Large Trees as Generated in the Game of Go.* PhD thesis, Stanford University, 1971. Microfilm no. 71-3, 162.

[Sam59] A. L. Samuel. Some studies in machine learning using the game of Checkers. *IBM Journal*, 3:210–229, 1959.

[Sam67] A. L. Samuel. Some studies in machine learning using the game of Checkers. ii. *IBM Journal*, 11:601–617, 1967.

[Sel89] Oliver G. Selfridge. Adaptive strategies of learning: A study of two-person zero-sum competition. In *Proceedings of the Sixth International Workshop on Machine Learning*, pages 412–415, 1989.

[Shi89] K. Shirayanagi. A new approach to programming go: Knowledge representation and its refinement. *Computer GO*, 11:10–24, Summer 1989.

[SS89] Wei-Min Shen and Herbert A. Simon. Rule creation and rule learning through environmental exploration. In *11th IJCAI*, pages 675–680, 1989.

[Sut84] Richard S. Sutton. *Temporal Credit Assignment in Reinforcement Learning*. PhD thesis, Department of Computer and Information Science, University of Massachusetts, 1984.

[TS89] G. Tesauro and T.J. Sejnowski. A parallel network that learns to play Backgammon. *Artificial Intelligence*, 39:357–390, 1989.

[YSUB90] Richard C. Yee, Sharad Saxena, Paul E. Utgoff, and Andrew C. Barto. Explaining temporal-differences to create useful concepts for evaluating states. In *Proceedings of AAAI-90*, 1990.