Go, SVM, Go

Thore Graepel, Mike Goutrié, Marco Krüger, and Ralf Herbrich
Computer Science Department
Technical University of Berlin
Berlin, Germany

Abstract

We consider the game of Go from the point of view of machine learning. We discuss the representation of both board positions and candidate moves and introduce the common fate graph (CFG) as an adequate representation of board positions for learning. Single candidate moves are represented as feature vectors with features given by subgraphs relative to the given move in the CFG. Using this representation we train a support vector machine and a kernel perceptron to discriminate good moves from bad moves on a collection of life-and-death problems and on 9×9 game records. We thus obtain kernel machines that solve Go problems and play 9×9 Go.

1 Introduction

Go (Chinese: Wei-Qi, Korean: Baduk) is an ancient oriental game that originated in China over 4000 years ago and that was developed to its modern perfection over the last centuries in Japan. The game has elegantly simple rules that lead to intricate tactical and strategic challenges. Its complexity by far exceeds that of chess, an observation well supported by the fact that the human world champion of chess found a worthy challenger in the computer program `Deep Blue`, while numerous attempts at reproducing such a result for the game of Go have been unsuccessful: so far no computer program is a serious challenger even for the average Go amateur. As a consequence, Go appears to be an interesting testing ground for machine learning [2]. In particular, we consider the problem of *representation* because an adequate representation of the board position is an essential prerequisite for the application of machine learning to the game of Go.

A particularly elegant statement of the rules of Go is due to Trump and Taylor[1] and is only slightly paraphrased for our purpose:

1. Go is played on an $N \times N$ square grid of points, by two players called Black and White.
2. Each point on the grid may be coloured *black*, *white* or *empty*. A point P is said to reach a colour C, if there exists a path of (vertically or horizontally) adjacent points of P's colour from P to a point of colour C. Clearing a

[1] See `http://www.cwi.nl/~tromp/go.html` for a detailed description.

colour is the process of emptying all points of that colour that do not reach *empty*.

3. Starting with an empty grid, the players alternate turns, starting with Black. A turn is either a pass; or a move that does not repeat an earlier grid colouring. A move consists of colouring an *empty* point one's own colour; then clearing the opponent colour, and then clearing one's own colour.
4. The game ends after two consecutive passes. A player's score is the number of points of her colour, plus the number of empty points that reach only her colour. The player with the higher score at the end of the game is the winner. Equal scores result in a tie.

While the majority of computer Go programs work in the fashion of rule-based expert systems [4], several attempts have been made to apply machine learning techniques to Go. Two basic learning tasks can be identified:

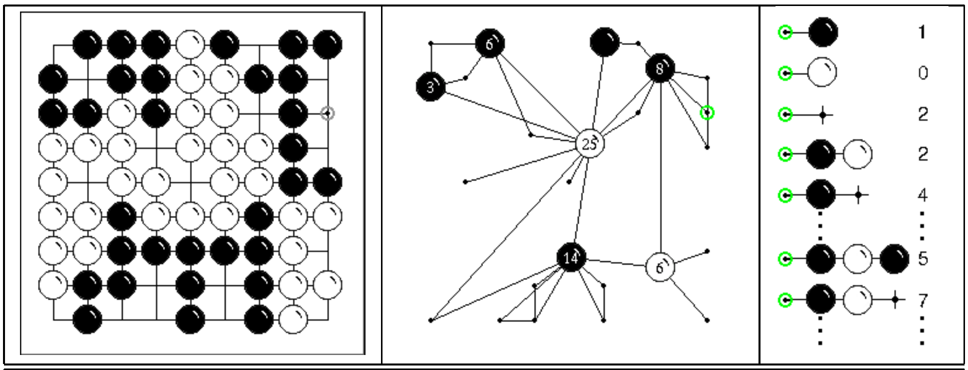
1. Learning an evaluation function for board positions
2. Learning an evaluation function for moves in given positions

The first task was tackled in the framework of reinforcement learning by Schraudolph and Sejnowski [7] who learned a pointwise evaluation function by the application of $TD(\lambda)$ to a multi-layer perceptron. The second task found an application to *tsume* Go in [6] who used a multi-layer perceptron to find problem-solving moves. Eventually, both learning tasks should be combined into a composite system that integrates classical search techniques based on a value function as learned in the first task with an "intelligent" move generator based on the second task.

All the known approaches suffer from a rather naive representation of board positions and candidate moves. The board is commonly represented as a simple two-dimensional array and single points or candidate moves are characterised by the pattern of stones in their surroundings. These representations obviously do not take into account the specific structure imposed by the rules of the game. In this paper we introduce a new representation for both board positions and candidate moves based on what we call a *common fate graph* (CFG). While our discussion is focused on the game of Go, our considerations about representation should be of interest with regard to all those domains were the standard feature vector representation does not adequately capture the structure of the learning problem. The CFG representation is introduced in Section 2 and complemented with the relative subgraph feature (RSF) representation. After briefly introducing the kernel perceptron and support vector machine we demonstrate the effectiveness of the new representation in Section 3. We train both a kernel perceptron and a support vector machine on a data base of life-and-death problems and on 9×9 game records: The resulting classifier is able to discriminate between good and bad local moves.

2 Representation

An adequate representation of the learning problem at hand is an essential prerequisite for successful learning. One could even go as far as saying that an adequate representation should render the actual learning task trivial. If the objects to be classified are represented such that the intra-class distances are zero, while the inter-class distances are strictly greater than zero a simple nearest neighbour classifier would be able to solve the learning problem perfectly. More realistically, we aim at finding a representation that captures the structure of board positions by mapping



(a) (b) (c)

Figure 1: Illustration of the feature extraction process: **(a)** board position (FGR), **(b)** corresponding common fate graph (CFG), and **(c)** a selection of extracted relative subgraph features (RSF) w.r.t. the node marked by a gray circle in (a) and (b).

similar positions (in the sense of the learning problem) to similar representations. Another desirable feature of a representation is a reduction in complexity: only those features relevant to the learning problem at hand should be retained.

2.1 Common Fate Graph

The value of a given Go position is invariant under rotation and mirroring of the board. Also the rules of the game refer essentially only to the local neighbourhood structure of the game. A board position is thus adequately represented by its *full graph representation* (FGR), a graph with the structure of an $N \times N$ square grid. In fact, the visual structure of a classical Go board supports this view (see Figure 1 (a)). More formally, let us define the FGR $G_{\mathrm{FGR}} = (P, E)$ as an undirected connected graph $G_{\mathrm{FGR}} \in \mathcal{G}_{\mathrm{uc}}$. The set $P = \{p_1, \ldots, p_{N_P}\}$ of nodes p_i represents the points on the board. Also, each node $p \in P$ has any of three given labels $l : P \to \{black, white, empty\}$. The symmetric binary "edge" relation $E = \{e_1, \ldots, e_{N_E}\}$ with $e_i \in \{\{p, p'\} : p, p' \in P\}$ represents vertical or horizontal neighbourhood between points. Note that this representation is suitable for arbitrary topologies of the Go board beyond the standard square grids.

However, the rules of the game provide us with more structural information. In particular, we observe that *black* or *white* points that belong to the same chain (i.e. points of the same colour that see each other) will always have a *common fate*: either all of them remain on the board or all of them are cleared. In any case we can represent them in a single node. We also reduce the number of edges by requiring that any two nodes may be connected by only a single edge representing their neighbourhood relation. The resulting *reduced graph representation* will be called a *common fate graph* (CFG) and will serve as the basic representation for our experiments. More formally, we define the graph transformation $T : \mathcal{G}_{\mathrm{uc}} \to \mathcal{G}_{\mathrm{uc}}$ by the following rule: given two nodes $p, p' \in P$ that are neighbours $\{p, p'\} \in E$ and that have the same non-empty label $l(p) = l(p') \neq empty$, perform the following transformation

1. $P \mapsto P \setminus \{p'\}$ to melt the node p' into p.
2. $E \mapsto (E \setminus \{\{p', p''\} \in E\}) \cup \{\{p, p''\} : \{p', p''\} \in E\}$ to connect the remaining node p to those nodes p'' formerly connected to p'.

Repeated application of the transformation T to G_{FGR} until no two neighbouring nodes have the same colour leads to the common fate graph G_{CFG}. The result of such a transformation is shown in Figure 1 (b). Clearly, the complexity of the representation has been greatly reduced while retaining essential structural information in the representation.

In how far is the CFG a suitable representation for learning in the game of Go? While this question will partly be answered in the experiments in Section 3 we would like to consider the CFG from the point of view of Go tactics. Go players' intuition is often driven by a certain kind of aesthetics that refers to the local structure of positions and is called *good or bad shape*. Certain types of moves or resulting positions are considered *good shape* in the local context and have potential for future tactical gain. As an example consider the two white groups in Figure 1 (a) and (b). Although they look quite distinct to the layman in Figure 1 (a) they share the property of *being alive,* because they both have *two eyes* (i.e. two isolated internal empty points called *liberties*). This essential property of stability is preserved in the CFG: living groups are represented by a coloured node with two dangling empty points (see Figure 1 (b)), irrespective of the particular bulk structure of the group. The abstraction implemented by the CFG lies in the fact that only chains of stones are collapsed into joint nodes and that the structure of *liberties* is preserved. Of course, some information like, e.g. the number of points and their structure within a node is lost.

2.2 Relative Subgraph Features

Unfortunately, almost all algorithms that aim at learning on graph representations directly suffer from severe scalability problems (see [5]). Most practically applicable learning algorithms operate on object representations known as feature vectors $\mathbf{x} \in \mathbb{R}^d$. We would thus like to extract feature vectors $\mathbf{x}$ from G_{CFG} for learning. Both learning tasks mentioned in the introduction can be formulated in terms of mappings from single points to real values: the position evaluation function can be (approximately) decomposed into a sum of pointwise evaluation functions [7] and the move evaluation function is naturally formulated as a function from (empty) points to real numbers. In both cases we would like to find a mapping $\boldsymbol{\phi} : \mathcal{G}_{\mathrm{uc}} \times P \to \mathbb{R}^d$ that maps a particular node $p \in P$ to a feature vector $\mathbf{x} \in \mathbb{R}^d$ given the context provided by the graph $G = (P, E) \in \mathcal{G}_{\mathrm{uc}}$, an idea inspired by [5], who apply context dependent classification to the mutagenicity of chemical compounds. We enumerate d possible connected subgraphs $\tilde{G}_i = (\tilde{P}_i, \tilde{E}_i) \in \mathcal{G}_{\mathrm{uc}}$, $i = \{1, \ldots, d\}$ of G such that $p \in \tilde{P}_i$. The relative subgraph feature $x_i = \phi_i(p)$ is then taken proportional to the number of times n_i the subgraph $\tilde{G}_i$ is present in G and normalised to $\|\mathbf{x}\| = 1$.

Clearly, finding and counting subgraphs $\tilde{G}_i$ of G becomes quickly infeasible with increasing subgraph complexity as measured, e.g. by $|\tilde{P}_i|$and $|\tilde{E}_i|$. We therefore restrict ourselves to connected subgraphs $\tilde{G}_i$ with the shape of chains without branches or loops. In practice we limit the features to a local context $|\tilde{P}_i| \leq s$, which — given the other two constraints — also limits the number d of distinguishable features. It should be noted, however, that even with a relatively small number of s, e.g. $s = 6$, a considerable range on the board is achieved due to the compact CFG representation. Also, the relative subgraph features extracted from the CFG can be partly interpreted in terms of Go terminology. As an example, consider the feature in Figure 1 (c) row 3. This feature effectively counts the *liberties* of the point relative to which it is extracted (marked by gray ring), a number that constitutes an important feature for human Go players and is a measure of the tactical safety of a chain of stones.

test \ train	white	black	w&b	#probs	#moves	%good
white	65.8/65.3	48.0/48.8	63.8/62.6	1455	10.4	13.4
black	57.7/57.3	66.4/64.5	65.9/65.9	1711	9.8	23.0
w&b	61.5/61.0	58.0/57.3	64.9/64.4	3166	10.0	18.0
# pts	2718	2682	5400			

Table 1: Results of the selection of *tsume* Go moves by an SVM (left) and a kernel perceptron (right). Shown is the percentage of successful determination of a problem-solving move. The numbers should be compared to the last column that indicates the average percentage of moves that solve the problem.

3 Experimental Result

Learning Algorithms In our experiments we focused on learning the distinction between "good" and "bad" moves. While the training data are labelled as "good" and "bad" moves, we are really interested in the real-valued output of the classifier which enables us to order and select candidate moves according to their quality. Given the RSFs extracted from the CFG any learning machine that takes feature vectors as inputs and provides a real-valued output is suitable for the task. In order to take into account the relatively high number of features we choose the class of binary kernel classifiers for learning. The predictions of these classifiers are given by

$$\hat{y}(\mathbf{x}) = \operatorname{sign}(f(\mathbf{x})) = \operatorname{sign}\left(\sum_{i=1}^{m} \alpha_i k(\mathbf{x}_i, \mathbf{x})\right) .$$

The α_i are the adjustable parameters and $k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ is the kernel function. These classifiers can be viewed as linear classifiers in a space that is spanned by non-linear features and have recently gained a lot of popularity due to the success of the support vector machine (SVM) [8]. We decided to use an RBF kernel with diagonal penalty term of the form

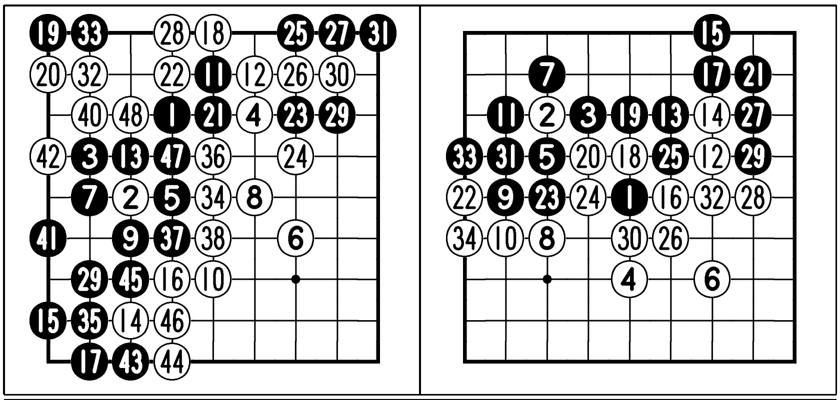
$$k(\mathbf{x}, \mathbf{x}') = \exp\left(-\|\mathbf{x} - \mathbf{x}'\|^2 / \sigma^2\right) + \lambda \mathbf{I}_{\mathbf{x}=\mathbf{x}'} .$$

Two methods of "learning" the expansion coefficients α_i were employed:

1. A simple kernel perceptron [1] that — initialised at $\boldsymbol{\alpha} = \mathbf{0}$ — passes through the training vectors $\mathbf{x}_i$ and increments the coefficient α_i by y_i whenever $\mathbf{x}_i$ is incorrectly classified until convergence.
2. A soft margin support vector machine[2] [3] that finds the solution to the QP problem of minimising $C\sum_{i=1}^{m} \xi_i + \sum_{i,j=1}^{m} \alpha_i \alpha_j k(\mathbf{x}_i, \mathbf{x}_j)$ under the constraints $\forall j \in \{1, \ldots, m\} : \sum_{i=1}^{m} \alpha_j k(\mathbf{x}_i, \mathbf{x}_j) \geq 1 - \xi_i$ and $\boldsymbol{\xi} \geq \mathbf{0}$.

Life and Death An interesting challenge in the game of Go — referred to as *tsume Go* — is to kill opponent groups and to save one's own groups from dying. Our *tsume* Go study was based on a database of 40000 computer-generated Go problems with solution by Thomas Wolf [9]. In order to calibrate the parameters of both the representation and the learning schemes, we created a simple task: The training set consisted of $m = 2700$ moves, where we took the best move provided by Wolf's problem solver GoTools [9] to be "good" and the worst one to be "bad". The length

[2] Publicly available via `http://kernel.anu.edu.au/`



(a) (b)

Figure 2: Two examples of games played by a support vector machine (Black) against Gnu-Go (White). The SVM was trained on 9×9 game records from good amateur games. Despite the fact that the SVM eventually looses both games it finds a number of remarkable moves, creates stable groups, and surrounds territory.

s of the subgraph chains was chosen to be $s = 6$ resulting in $d \approx 400$ features. Using a held-out data set of size $m_{\text{heldout}} \approx 3000$ we systematically scanned parameter space. For the SVM we set $\lambda = 0$ (relying on C for regularisation) and found the optimal parameter values to be $\sigma = 1$ and $C = 10$. For the kernel perceptron we used the same value of σ and found $\lambda = 0.05$ to be optimal. Both solutions had a sparsity level $\|\boldsymbol{\alpha}\|_0 / m$ of approximately 50% and lead to a success rate of up to 85% for discriminating between the best and the worst move in a given problem.

Given these parameters we turned to the more interesting task of picking the correct move among all the possible moves in a problem. Using essentially the same training paradigm as above we applied the resulting classifier to all the possible moves in a problem and used the real-valued output $f(\mathbf{x})$ for each legal move $\mathbf{x}$ for ranking. We counted a success when the move ranked highest by the classifier was indeed one of the winner moves. The results of these experiments are given in Table 1. Focusing on the full training and test sets (*w&b*) the success rate is more than 3 times that of random guessing. Although we were not able to obtain the data-base used in [6] due to copyright problems, our result of a 65% success rate compares favourably with the 50% reported in [6] on similar problems using a naive local context representation and a multi-layer perceptron with backpropagation.

In the problems used, *white* was the attacker and *black* the defender. The sections of the table refer to subsets of the whole training set containing only *black*, only *white*, or *black* and *white* moves. It turns out that in accordance with the Go proverb "Your opponent's move is your own move" the quality of certain moves does not depend on whether they are played for the purpose of attack or defence. It should be noted that the performance of the kernel perceptron is absolutely comparable to that of the SVM — despite of the much simpler training paradigm that requires much less computational resources than the SVM. We conjecture that in this special task the representation is so crucial that the difference between the learning algorithms becomes negligible.

Game Play The test of the pudding is in the eating, and accordingly the test of the Go machine is in the playing. We applied the same learning paradigm as used above, but this time to a collection of 9×9 Go game records collected from the International Go Server (IGS) and pre-processed and archived by Nicol Schraudolph. We selected game records that were produced by players with a ranking of at least *amateur shodan*, assuming that their moves could be viewed as optimal from the point of view of the learning machine. For each of the $m \approx 2500$ moves played we randomly generated an arbitrary counterpart as a "bad" move. We would like to provide two samples of the SVMs play against Gnu-Go[3] in Figure 2. Connoisseurs of the game will appreciate that the machine plays amazingly coherent considering that it takes into account only local shape and has no concept of territory and urgency. Surprisingly, the program even creates situations of *atari* (in moves 9 and 45 of game (a) and moves 7, 25, and 29 of game (b)) and is able to capture attacked opponent stones in move 13 of game (a) and in move 11 of game (b). Unfortunately, it misses other opportunities and looses both games.

4 Conclusions and Future Work

We presented Go as a successful application of learning based on feature vectors that are extracted from a graph representation. While the approach appears to be particularly suitable for the game of Go it seems that other domains could benefit from these ideas as well. With regard to the problem of building a competent Go playing machine our approach represents only a first step. In its present form, the system could serve as an intelligent move generator thus cutting down the width of the search tree in an evaluation function based system. In addition, the same representation that serves now to learn move evaluations could be used to learn position evaluation functions decomposed into a sum of the nodes of the CFG. Both the move and position evaluation combined with search would certainly lead to progress in playing strength. Eventually, is seems plausible that the idea of abstraction as represented by the CFG must be carried out over a hierarchy of representations mimicking the way human players think about the game of Go.

Acknowledgements We would like to thank Nic Schraudolph and Thomas Wolf for providing datasets. Finally, we would like to thank Klaus Obermayer for hosting our project in the NI group at the TU Berlin.

References

[1] M. Aizerman, E. Braverman, and L. Rozonoer. Theoretical foundations of the potential function method in pattern recognition learning. *Automation and Remote Control*, 25:821–837, 1964.

[2] J. Burmeister and J. Wiles. Temporal difference learning of position evaluation in the game of go. In *Proceedings of the 3rd Australian and New Zealand Conference on Intelligent Information Systems*, 1994.

[3] C. Cortes and V. Vapnik. Support Vector Networks. *Machine Learning*, 20:273–297, 1995.

[4] D. Fotland. Knowledge representation in The Many Faces of Go, 1993.

[5] P. Geibel and F. Wysotzki. Learning relational concepts with decision trees. In *Machine Learning: Proceedings of the Thirteenth International Conference*, pages 1141–1144. Morgan Kaufmann Publishers, 1998.

[6] N. Sasaki and Y. Sawada. Neural networks for tsume-go problems. In *Proceedings of the Fifth International Conference on Neural Information Processing*, pages 1141–1144, 1998.

[7] N. N. Schraudolph, P. Dayan, and T. J. Sejnowski. Temporal difference learning of position evaluation in the game of go. In J. D. Cowan, G. Tesauro, and J. Alspector, editors, *Advances in Neural Information Processing Systems*, volume 6, pages 817–824. Morgan Kaufmann Publishers, Inc., 1994.

[8] V. Vapnik. *Statistical Learning Theory*. John Wiley and Sons, New York, 1998.

[9] T. Wolf. The program GoTools and its computer-generated Tsume Go database. In *Proceedings of the 1st Game Programming Workshop*, 1994.

[3] Publicly available via `http://www.gnu.org/software/gnugo/`