LEARNING TO EVALUATE GO POSITIONS VIA TEMPORAL DIFFERENCE METHODS

Nicol N. Schraudolph
IDSIA
Galleria 2
CH-6928 Manno
Switzerland
(nic@idsia.ch)

Peter Dayan
Gatsby Unit
17 Queen Square
London, WC1N 3AR
United Kingdom
(dayan@gatsby.ucl.ac.uk)

Terrence J. Sejnowski
Comput. Neurobiology
The Salk Institute
La Jolla, CA 92037
USA
(terry@salk.edu)

Technical Report IDSIA-05-00*

February 21, 2000
revised May 24, 2000

The game of Go has a high branching factor that defeats the tree search approach used in computer chess, and long-range spatiotemporal interactions that make position evaluation extremely difficult. Development of conventional Go programs is hampered by their knowledge-intensive nature. We demonstrate a viable alternative by training neural networks to evaluate Go positions via temporal difference (TD) learning.

Our approach is based on neural network architectures that reflect the spatial organization of both input and reinforcement signals on the Go board, and training protocols that provide exposure to competent (though unlabelled) play. These techniques yield far better performance than undifferentiated networks trained by self-play alone. A network with less than 500 weights learned within 3 000 games of 9x9 Go a position evaluation function superior to that of a commercial Go program.

*To appear in *Soft Computing Techniques in Game Playing*, Jain, L. C., and Baba, N., Eds., Springer Verlag, Berlin, 2000.

1 Introduction

1.1 The Game of Go

Go was developed four millennia ago in China; it is one of the oldest and most popular board games in the world. Like chess, it is a deterministic, perfect information, zero-sum game of strategy between two players. They alternate in placing black and white stones on the intersections of a 19x19 grid (smaller for beginners) with the objective of surrounding more board area *(territory)* with their stones than the opponent. Adjacent stones of the same color form *groups*; an empty intersection adjacent to a group is called a *liberty* of that group. A group is captured and removed from the board when its last liberty is occupied by the opponent. To prevent loops, it is illegal to make certain moves which recreate a prior board position. A player may pass at any time; the game ends when both players pass in succession. Each player's *score* is then calculated as the territory (number of empty intersections) they have surrounded plus the number of enemy stones they have captured; the player with the higher score wins the game.

1.2 Computer Go

Unlike most other games of strategy, Go has remained an elusive skill for computers to acquire — indeed it is increasingly recognized as a "grand challenge" of Artificial Intelligence [1], [2], [3]. The game tree search approach used extensively in computer chess is infeasible: the game tree of Go has an average branching factor of around 200, but even beginners may routinely look ahead up to 60 plies in some situations. Humans appear to rely mostly on static evaluation of board positions, aided by highly selective yet deep local lookahead. Conventional Go programs are carefully (and protractedly) tuned expert systems [4]. They are fundamentally limited by their need for human assistance in compiling and integrating domain knowledge, and still play barely above the level of a human beginner — a machine learning approach may thus offer considerable advantages. Brügmann [5] has shown that a knowledge-free optimization approach to Go can work in principle: he obtained respectable (though inefficient) play by selecting moves through simulated annealing [6] over possible continuations of the game.

Creating (respectively preventing) *live groups* — particular patterns on the board which are stable against opposing play — is one of the basic elements of strategy, so there is a large component of pattern recognition inherent in Go. It is this component which is amenable to connectionist methods. Supervised backpropagation networks have been applied to the game [7], [8], but face a bottleneck in the scarcity of hand-labelled training data. By contrast, a vast number of *unlabelled* Go game records is readily available on the Internet; we therefore propose an approach which can utilize this rich supply of data.

2 Temporal Difference Learning

We use an algorithm called temporal difference (TD) learning [9] to acquire a function that *evaluates* board positions. It can be seen as an adaptation of Samuel's famous program that learned to play checkers [10]. The idea is to take a representation of the board position $\mathbf{x}_t$ at time t in the game and from it produce a number $f(\mathbf{x}_t;\mathbf{w})$ which specifies how good this board position is. Here $\mathbf{w}$ is a set of *parameters* which are tuned during learning to turn $f(\mathbf{x}_t;\mathbf{w})$ into a good evaluation function — *i.e.,* one that accurately reflects the chances of winning the game (against a strong opponent) from the given position $\mathbf{x}_t$. In our case, $f(\mathbf{x}_t;\mathbf{w})$ will be implemented by a neural network, and $\mathbf{w}$ are its weights.

Given an accurate position evaluation function, the computer can choose good moves by conventional search methods. Since our focus here is on learning the evaluation function, we use only the most primitive search strategy: our program typically tries out every legal move, then picks the one leading to the board position with the highest value of $f(\mathbf{x}_t;\mathbf{w})$.

2.1 Mathematical Derivation

Consider first trying to learn how to evaluate board positions based on playing a fixed strategy against randomly selected opponents. If we arrange for a *reward* $r_T = 1$ if the game ends at time T and the program wins, and set $r_t = 0$ if $t \neq T$ or if the program loses, then the probability of winning the game from position

$\mathbf{x}_t$ can be written as

$$V(\mathbf{x}_t) \;=\; E\left[\sum_{s=t}^{\infty} r_s\right] \,, \tag{1}$$

where $E[\cdot]$ takes an expectation, in this case starting from position $\mathbf{x}_t$, over the play of the opponents as well as any randomization in the program's own moves.

The TD approach provides a learning rule for changing the parameters $\mathbf{w}$ to improve the fit of our evaluation function $f(\mathbf{x}_t; \mathbf{w})$ to the true *value function* $V(\mathbf{x}_t)$. Separating the first and subsequent terms of the sum in (1) gives

$$\begin{aligned} V(\mathbf{x}_t) &= E[r_t] + E\left[\sum_{s=t+1}^{\infty} r_s\right] && (2) \\ &= E[r_t] + E[V(\mathbf{x}_{t+1})] && (3) \\ &\sim r_t + V(\mathbf{x}_{t+1}) && (4) \\ &\approx r_t + f(\mathbf{x}_{t+1}; \mathbf{w}) && (5) \end{aligned}$$

where the second expectation in (3) is over the value of $\mathbf{x}_{t+1}$, (4) uses random *samples* of r_t and $\mathbf{x}_{t+1}$ instead of their expectations (making this a Monte Carlo method), and (5) replaces the true value $V(\mathbf{x}_{t+1})$ with its current approximation $f(\mathbf{x}_{t+1}; \mathbf{w})$. Using the right-hand side of (5) as a target for $f(\mathbf{x}_t; \mathbf{w})$, we construct a *prediction error* term

$$\delta_t \;=\; r_t + f(\mathbf{x}_{t+1}; \mathbf{w}) \;-\; f(\mathbf{x}_t; \mathbf{w}) \tag{6}$$

which is used to change the weights $\mathbf{w}$. The simplest TD rule, called TD(0), changes the weights according to

$$\Delta\mathbf{w} \;\propto\; \delta_t \, \nabla_{\mathbf{w}} f(\mathbf{x}_t; \mathbf{w}) \,, \tag{7}$$

where $\nabla_{\mathbf{w}} f(\mathbf{x}_t; \mathbf{w})$ is the vector of partial derivatives of the evaluation function with respect to the parameters. This can be shown under very particular circumstances to make $f(\mathbf{x}; \mathbf{w})$ converge to $V(\mathbf{x})$. (Our model, together with many other applications of TD, uses the learning rule successfully in a regime where these guarantees do not hold.) There is a refinement of TD(0) called TD(λ), which instead uses

$$\begin{aligned} \Delta\mathbf{w} &\;\propto\; \delta_t \, \bar{\mathbf{f}}_t \,, \;\; \text{where} && (8) \\ \bar{\mathbf{f}}_t &\;=\; \nabla_{\mathbf{w}} f(\mathbf{x}_t; \mathbf{w}) \;+\; \lambda \, \bar{\mathbf{f}}_{t-1} && (9) \end{aligned}$$

and whose free parameter $0 \leq \lambda \leq 1$ can be shown to trade off bias against variance in the weight changes [11].

2.2 Use for Game Playing

Once appropriate parameters **w** are learned, the evaluation function can be used to choose moves — naturally favoring those that lead to better board positions. In this case, as **w** changes, the program's strategy will change, leading to different outcomes of the game, which in turn will trigger yet other changes in **w**. Done carefully, this process is related to the engineering optimization method of *dynamic programming* [9, 12]. However, even when done in a more heuristic manner, as we are forced to, this can work quite well.

Using the TD approach, a program can thus learn to play a game without ever having been trained on explicitly labelled examples of good *vs.* bad play. In fact, any generator of legal moves — game records, other computer programs, random move generators, and so forth — can in principle be used for TD training. At the most extreme, the TD learner itself can be used to play *both* sides of the game according to its current evaluation function. Learning from such *self-play* is intriguing in that in this setup the program has no access at all to even implicit knowledge about the game it is to learn, beyond its bare rules (which must necessarily be hardcoded into the program).

TD learning has been successfully applied to the game of backgammon by Tesauro [13]. His TD-Gammon program used a backpropagation network to map preselected features of the board position to an output reflecting the probability that the player to move would win. It was trained by TD(0) while playing only itself, yet learned an evaluation function that — coupled with a full two-ply lookahead to pick the estimated best move — made it competitive with the best human players in the world [14], [15].

2.3 Naive Application to Go

In an early experiment we investigated a straightforward adaptation of Tesauro's approach to the Go domain. Go is similar to backgammon in that complete expansion of the search tree at even moderate levels is futile. In contrast to backgammon, however, it has no random element to ensure adequate exploration, or to render extended strategies ineffective. We thus had to supply the stochasticity necessary for TD learning by randomizing our network's self-play. This was achieved by picking moves through Gibbs sampling [16], that is, with probability proportional to $e^{\beta f(\mathbf{x}_{t+1})}$, over the board positions $\mathbf{x}_{t+1}$ reached by all legal moves (*i.e.*, a full single-ply search). The pseudo-temperature parameter β was gradually increased from zero (random play) towards infinity (best-predicted play) over the course of training [6].

We trained a fully connected backpropagation network on a 9x9 Go board (a standard didactic size for humans) in this fashion. The network had 82 inputs (one for each point on the board, plus a bias input), 40 hidden units with hyperbolic tangent activation function, and a single output learning to predict the difference between the two players' scores. Inputs were encoded as +1 for a black stone, -1 for a white one, and zero for an empty intersection. Reward r_t was given for captured enemy stones at the time of capture, and for surrounded territory at the end of the game.

This network did learn to squeak past *Wally*, a weak public domain program [17], but it took 659 000 games of training to do so. One reason for this sloth lies in the undifferentiated nature of the fully connected network, which fails to capture any of the spatial structure inherent in the game. Another another contributing factor is the paucity of the scalar reinforcement signal. An important difference to backgammon is that the terminal state of a Go board is quite informative with respect to the play that preceded it: much of the board will be designated as territory belonging to one or the other player, indicating a positive *local* outcome for that player. Finally, while self-play did suffice for TD-Gammon, Go is a game of far higher complexity, and exposure to somewhat more competent play may be required in order to achieve results within reasonable training time.

We have found that the efficiency of learning to play Go by TD methods can indeed be vastly improved through appropriately structured network architectures, use of a richer *local* reinforcement signal, and training strategies that incorporate but do not rely exclusively on self-play. In the remainder of this chapter, we will describe these improvements in detail.

3 Network Architecture

Figure 1 illustrates the neural network architecture we propose for TD learning of a position evaluation function for Go. Its particular features are described

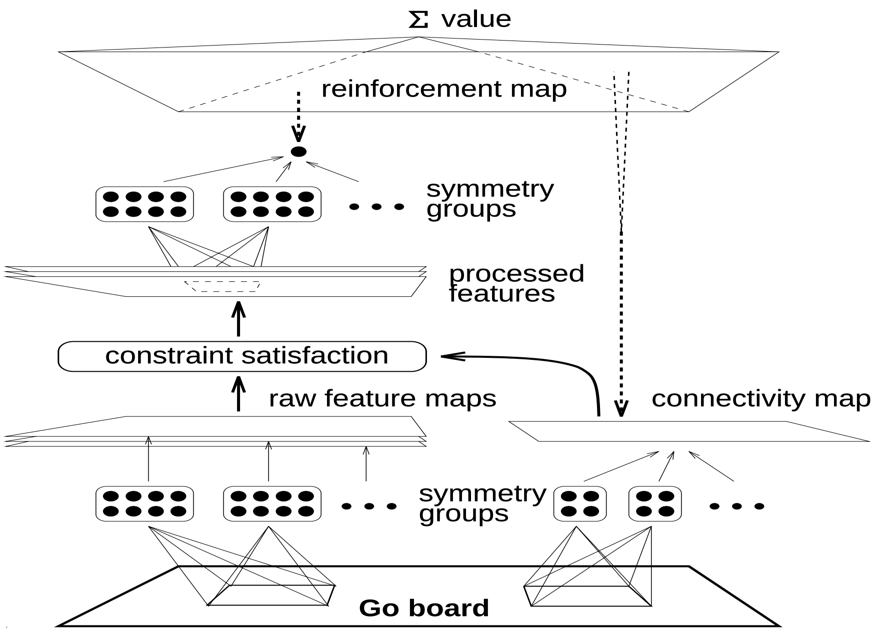


Figure 1. A network architecture that takes advantage of board symmetries, translation invariance and localized reinforcement. Also shown is the proposed connectivity prediction mechanism (Section 3.4).

in detail below. In our experiments we implemented all of them except for the connectivity map and lateral constraint satisfaction, which are the subject of future work.

3.1 Local Reinforcement

One of the particular advantages of Go for predictive learning is that there is much richer information available at the end of the game than just who won. Unlike chess, checkers or backgammon, in which pieces are taken away from the board until there are few or none left, Go stones generally remain where they are placed. This makes the final state of the board richly informative with respect to the course of play; indeed the game is scored by summing contributions from each point on the board.

We make this spatial credit assignment accessible to the network by having it predict the fate of every point on the board rather than just the overall score, and evaluate whole positions accordingly. (This bears some similarity with the Successor Representation [18] which also integrates over vector rather than scalar destinies.) Specifically, the network now has an output for each point on the board, which receives a reward r_t of ± 1 for the capture of a prisoner at that point (during the game), and likewise for that point being black or white territory at the end of the game.

For reasons of computational efficiency, it is desirable to combine the gradient information with respect to all outputs into a single scalar for each hidden unit of a backpropagation network; this forces us to use $\lambda = 0$ when implementing of TD(λ) with local reinforcement signals (*i.e.,* multiple outputs). Note that although Tesauro did not have this constraint, he nonetheless found TD(0) to be optimal [13]. Our experience has been that the advantages of incorporating the much richer local reinforcement signal by far outweigh the disadvantage of being limited to $\lambda = 0$.

3.2 Symmetries

Given the knowledge-based approach of existing Go programs, there is an embarrassment of input features that one might adopt for Go: *Wally* already uses about 30 of them, stronger programs disproportionately more. In order to demonstrate reinforcement learning as a viable alternative to the conventional approach, however, we require our networks to learn whatever set of features they might need. The complexity of this task can, however, be significantly reduced by exploiting a number of symmetries that hold *a priori* in this domain. Specifically, patterns of Go stones retain their properties under color reversal, reflection and rotation of the board. Each of these invariances should be reflected in the network architecture.

Color reversal invariance implies that changing the color of every stone in a Go position, and the player whose turn it is to move, yields an equivalent position from the other player's perspective. We build this constraint directly into our networks by using antisymmetric input values (+1 for black, -1 for white) and hidden unit activation functions (hyperbolic tangent) throughout, and changing the bias input from $+1$ to -1 when it is white's turn to move. This arrangement obviously guarantees that the network's outputs will have identical magnitude but opposite sign when the input position is color-reversed.

Go positions are also invariant with respect to the eightfold (reflection × rotation) symmetry of the square.[1] We provided a mechanism for constraining the network to obey this invariance by creating *symmetry groups* of eight hidden units, each seeing the same input under a different reflection/rotation, with appropriate weight sharing and summing of derivatives [19] within each symmetry group.

Although this was clearly beneficial during the evaluation of the network against its opponents, it appeared to actually impede the course of learning, for reasons that are not clear at this point. We settled on using symmetry groups only in play, using a network trained without them.

3.3 Translation Invariance

Modulo the considerable influence of the board edges, patterns of Go stones also retain their properties under translation across the board. To implement translation invariance we use convolution with a weight kernel rather than multiplication by a weight matrix as the basic mapping operation in our network, whose layers are thus *feature maps* produced by scanning a fixed receptive field (the weight kernel) across the input [20]. One particular advantage of this technique is the easy transfer of learned weight kernels to different Go board sizes.

It must be noted, however, that strictly speaking, Go is *not* fully translation-invariant: the edge of the board not only affects local play but modulates other aspects of the game, and indeed forms the basis of opening strategy. We currently account for this by allowing each node in our network to have its own bias weight, thus giving it one degree of freedom from its neighbors. This enables the network to encode absolute position at a modest increase in the number of adjustable parameters. Furthermore, we provide additional redundancy around the board edges by selective use of convolution kernels twice as wide as the input. Weights near the edge of such extra-wide kernels are used only for points near the opposite edge in the feature map, and are thus free to specialize in encoding board edge effects.

For future implementations, we suggest that it is possible to augment the input representation of the network in such a way that its task becomes fully

[1] There are human conventions about the propriety of starting the game in a particular corner, which is a concern in teaching the network from recorded human games.

translation-invariant, by adding an extra input layer whose nodes are activated when the corresponding points on the Go board are empty, and zero when they are occupied (regardless of color). Such a scheme represents board edges in precisely the fashion in which they influence the game: through the absence of free board space beyond them. This consistency should make it possible for the network to encode reasonable evaluation functions with fully translation-invariant receptive fields, thus eliminating the need for any special treatment of the board edges. As an additional benefit, the augmented input representation also makes the three possible states of a point on the board (black stone, white stone, or empty) *linearly separable* — hence easier to process — for the network.

3.4 Connectivity

The use of limited receptive field sizes raises the problem of how to account for long-ranging spatial interactions on the board. In Go, the distance at which groups of stones interact is a function of their arrangement in context; an important subproblem of position evaluation is therefore to compute the *connectivity* of groups of stones. We propose to model connectivity explicitly by training the network to predict the *correlation* pattern of local reinforcement from a given position. This information can then be used to inform the lateral propagation of local features in the hidden layer through a constraint satisfaction mechanism. The task is to *segment* the board into groups of stones that are (or will be) effectively connected; image segmentation algorithms from computer vision may prove useful here.

4 Training Strategies

Temporal difference learning teaches the network to predict the consequences of following particular strategies on the basis of the play they produce. The question arises as to which strategies should be used to generate the large number of Go games needed for training. In principle, any generator of legal Go moves could be used to play either side of the game; in practice, a carefully chosen combination of move generation strategies is key to achieving good TD learning performance.

Table 1. Comparison of alternative move generation strategies.

move generator	speed	quality	quantity	coverage	flexibility
game record	fast	high	limited	conventional	none
Go program	slow	medium	unlimited	questionable	some
TD network	slow	low	unlimited	questionable	high
random play	fast	none	unlimited	ergodic	high

We evaluate particular move generators according to five criteria: the **speed** with which they can provide us with moves, the **quality** of the moves provided, the **quantity** of moves we can obtain from the generator, to what extent these moves **cover** the space of plausible Go positions, and finally the **flexibility** of the move generator. We regard a move generator as flexible if it can be used in arbitrary board positions and against arbitrary opponents. Table 1 lists four types of move generators, and how they fare in regard to these criteria. In what follows, we shall discuss each type in greater detail.

4.1 Recorded Games

The growth of the Internet, and the popularity of *Internet Go Servers* — where Go aficionados from all over the globe congregate to play networked games — has led to an explosion in the amount of machine-recorded Go games. We estimate that at present about 500 000 recorded games between Go professionals and serious amateur players are available in machine-readable format. They offer a supply of instantaneous (since prerecorded), high-quality Go moves for TD training. As to their coverage, these games naturally represent conventional human play, which might help a trained network in routine play against humans but exposes it to brittleness in the face of unconventional moves by the opponent.

There are other drawbacks to training from recorded games: there is no flexibility (the game record must be played through from the start), and the supply of *suitable* games can be quite limited. Specifically, most machine learning approaches to Go use the smaller 9x9 board due to computational limitations; among humans, this board size is used only by rank beginners to learn the basics of the game. Thus only a few thousand 9x9 games, and of questionable quality, have been recorded to date.

Another major obstacle is the human practice of abandoning the game once both players agree on the outcome — typically well before a position that could be scored mechanically is reached. While the game record typically contains the final score (sufficient for our naive TD-Go network), the black and white territories (required for local reinforcement) are rarely given explicitly. This issue can be addressed by eliminating early resignations from the training set, and using existing Go programs to continue the remaining games to a point where they can be scored mechanically. For verification, the score thus obtained can then be compared to that given in the game record, with mismatches also eliminated from the training set.

4.2 Other Go Programs

Existing computer Go programs can also be used as a source of data for TD training. Although these programs are not as good as typical human players, they do incorporate a significant body of knowledge about the game, and provide reasonable moves in unlimited quantity, albeit at relatively slow speed. Regarding coverage, these programs typically respond reasonably to conventional human play, but can react in rather bizarre ways to unconventional moves (*e.g.*, those of a computer opponent). The major practical issues in using computer Go programs for TD learning are the tradeoff between their speed and quality of moves, and their flexibility (or lack thereof). We have explored the use of two Go programs in this fashion: *Wally* and *The Many Faces of Go*.

Wally [17] is a rather weak public domain program based on simple pattern matching. It does have the advantages of being quite fast, purely reactive, and available in source code, so that we were able to seamlessly integrate it into our TD-Go system, and use it as a move generator with full flexibility.

The commercial *Many Faces* [4], by contrast, is a self-contained DOS program. To use it, we had to hook a spare PC to our system via serial cable, and pretend to be a modem through which a remote opponent (*i.e.*, our system) was playing. Since it was not possible to set up arbitrary board positions by modem, we always had to play entire games. Parameters such as its skill level and handicap had to be set manually as well, so overall flexibility was low, as was the speed of move generation. These drawbacks are redeemed by the fact that for a computer program, *Many Faces* delivers comparatively high-quality moves.

4.3 TD Network Moves

Tesauro trained TD-Gammon by self-play — *i.e.*, the network's own position evaluation was used (in conjunction with a full search over all legal moves) to pick both players' moves during TD training. This technique is impressive in that does not require any external source of expertise beyond the rules of the game: the network is its own teacher. We already adopted this approach for 9x9 Go in our "naive" TD-Go network (Section 2.3); now we re-examine it as one possible move generation strategy. As a move generator, the TD network is comparable to Go programs like *Wally*, providing (with full flexibility) an unlimited supply of relatively slow and (at least early in training) low-quality moves.

As for coverage, Go (unlike backgammon) is a deterministic game, so we cannot always pick the estimated best move when training by self-play without running the risk of trapping the network in some suboptimal fixed state. Theoretically, this should not happen — the network playing white would be able to predict the idiosyncrasies of the network playing black, take advantage of them thus changing the outcome, and forcing black's predictions to change commensurately — but in practice it is a concern. We therefore pick moves stochastically by Gibbs sampling [16], in which the probability of a given move is exponentially related to the predicted value of the position it leads to, through a pseudo-temperature parameter that controls the degree of randomness. It is an open question, however, just *how much* stochasticity is required for TD learning to proceed most efficiently.

Although it offers the unique opportunity for the TD network to learn from its own idiosyncrasies, we found self-play alone to be rather cumbersome for two reasons: firstly, the single-ply search used to evaluate all legal moves is computationally intensive — and although we are investigating faster ways to accomplish it, we expect move evaluation to remain a computational burden. Secondly, learning from self-play is sluggish as the network must bootstrap itself out of ignorance without the benefit of exposure to skilled opponents. When we do use the TD network as a move generator for its own training, we thus find it generally preferable to let the TD network play against another Go program, such as *Wally* or *Many Faces*. This also provides a convenient way to monitor the progress of training, and to determine whether the architectures we have chosen provide enough flexibility to represent a useful evaluation function.

4.4 Random Moves

Recorded games aside, the fastest way to generate legal Go moves is to just pick a random one. Although this approach doesn't generate play of any appreciable quality, we found that TD networks can learn a surprising amount of basic Go knowledge by observing a few thousand quick games of random Go; this accords well with Brügmann's results [5]. In particular, this proved an effective way to prime our networks at the start of training.

The random move generator combines the advantages of high speed and ergodicity, *i.e.*, it explores all legally reachable Go positions. In order to provide a minimum of stability and structure to its play, we do prevent it from filling in its own single-point *eyes* — a particular, locally (and easily) recognizable type of suicidal move.

4.5 Matching Opponents

Sufficiently flexible move generators can in principle be arbitrarily combined to play a game between two players. In order to create useful training data, however, the two opponents should be well-matched in their skill level. Otherwise, trivial predictions of the game's outcome (such as "white always wins") become possible, which undermines the network's learning process. Human Go players are matched using a system of ratings and handicaps; our TD-Go framework permits at least three additional ways to ensure that opponents are of equal strength:

- use the same move generator on both sides (self-play),
- have the players trade sides several times during the game, or
- *dilute* the stronger player by interspersing it with an appropriate proportion of random moves.

For move generators that are sufficiently flexible to support it, we favor the dilution approach, since it has a number of advantages: firstly, the proportion of random moves can be changed adaptively, based on the outcome of past games. When one of the players is the TD network, this not only keeps the opponents well-matched while the network improves over time, but also — secondly —

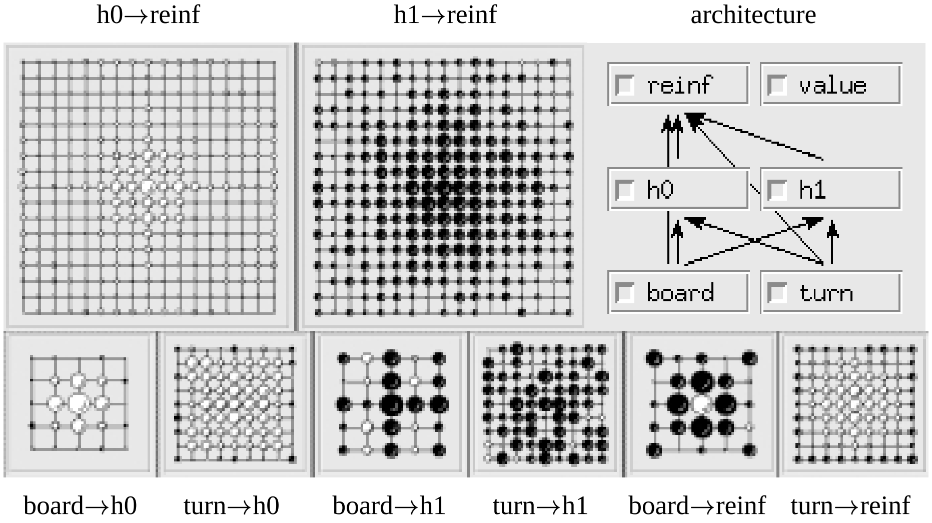
provides us with a convenient on-line performance measure. Finally, the injection of random moves also serves to guarantee sufficient variety of play (*i.e.*, coverage) in cases where this would otherwise be in doubt.

Since, in all cases, the strategies of both players are intimately intertwined in the predictions, one would never expect them to be correct overall when the network is playing a real opponent. This is a particular problem when the strategy for choosing moves during learning is different from the policy adopted for “optimal” network play. Samuel [10] found it inadvisable to let his checker program learn from games which it won against an opponent, since its predictions might otherwise reflect poor as well as good play. This is a particularly pernicious form of over-fitting — the network can learn to predict one strategy in exquisite detail, without being able to play well in general.

5 Empirical Results

In our experiments we trained many networks by a variety of methods. A small sample network that learned to beat *Many Faces* (at low playing level) in 9x9 Go within 3 000 games of training is shown in Figure 2. This network was grown during training by adding hidden layers one at a time; although it was trained without the (reflection × rotation) symmetry constraint, many of the weight kernels learned approximately symmetric features. The direct projection from board to reinforcement layer has an interesting structure: the negative central weight within a positive surround stems from the fact that a placed stone occupies (thus loses) a point of territory even while securing nearby areas. Note that the wide 17x17 projections from the hidden layers have considerable fringes — ostensibly a trick the network uses to incorporate edge effects. (Absolute position is also encoded explicitly in the bias projections from the *turn* unit.)

We compared training this network architecture by self-play versus play against *r-Wally*, a version of *Wally* diluted with random play in adaptive proportion. Figure 3 show the network’s performance during training against both *r-Wally* and (to evaluate generalization) *Many Faces*. Although the initial rate of learning is similar in both cases, the network playing *r-Wally* soon starts to outperform the one playing itself; this demonstrates the advantage of having a skilled opponent. After about 2 000 games, however, both start to overfit their opponents, and consequently worsen against *Many Faces*.



h0→reinf h1→reinf architecture

board→h0 turn→h0 board→h1 turn→h1 board→reinf turn→reinf

Figure 2. A small network that learned to play 9x9 Go. Boxes in the architecture panel represent 9x9 layers of units, except for *turn* and *value* which are scalar. Arrows indicate convolutions with the corresponding weight kernels. Black disks represent excitatory, white ones inhibitory weights; within each panel, disk area is proportional to weight magnitude.

Switching training partner to *Many Faces* — set to a skill level of 2-3, out of a maximum of 20 — at this point produced after a further 1 000 games a network that could reliably beat this opponent (dotted line in Figure 3). The low skill setting we used essentially disabled *Many Faces*' deep lookahead mechanisms [21]; since our TD network move generator does not search any deeper than a single ply either, this can be viewed as a fair test of static position evaluation and move selection capabilities.

Although less capable, the self-play network did manage to edge past *Wally* after 3 000 games; this compares very favorably with the undifferentiated network described in Section 2.3. Furthermore, we verified that weights learned from 9x9 Go offer a suitable basis for further training on the full-size (19x19) Go board. Computational limitations did not permit comprehensive training on the full-size board though, where recorded games would offer a rapid source of high-quality play.

Subjectively, our networks appear more competent in the opening than further into the game. This suggests that although reinforcement information is indeed

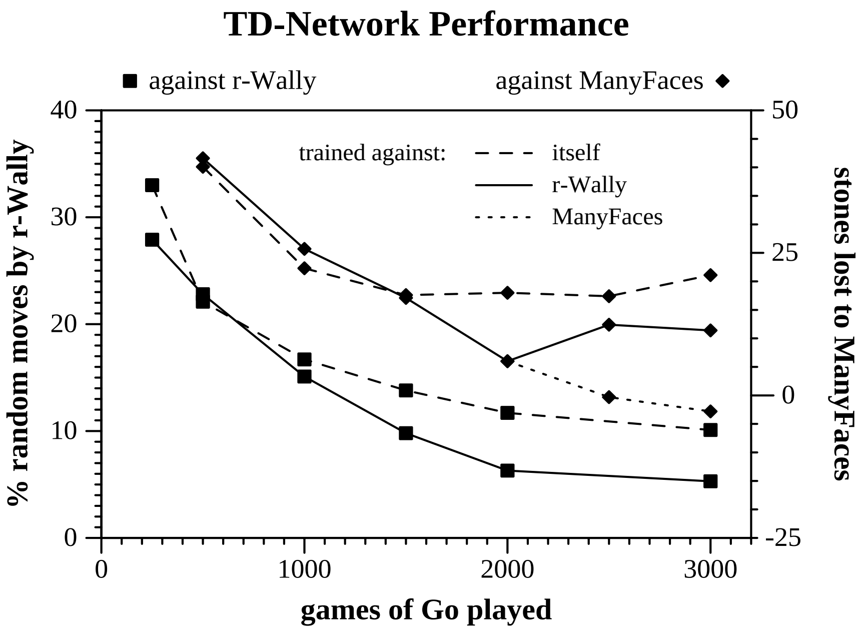


Figure 3. Performance of our 9x9 Go network, measured against two opponents — *Wally* diluted with random moves (boxes, left axis), and *Many Faces* (diamonds, right axis) — when trained by playing against itself (dashed), the randomized *Wally* (solid line), or *Many Faces* (dotted).

propagating all the way back from the final position, it is hard for the network to capture the multiplicity of mid-game situations and the complex combinatorics characteristic of the endgame. These strengths and weaknesses partially complement those of symbolic systems, suggesting that hybrid approaches might be rewarding [22], [23].

6 Summary

We have shown that with sufficient attention to network architecture and training procedures, a neural network trained by temporal difference learning can achieve significant levels of performance in this knowledge-intensive domain. Specifically, we have identified salient characteristics of Go, such as the informative nature of the final state of every game, the mixture of translation variance and invariance, and color reversal symmetry, and have shown how to capture them efficiently in the network architecture. Networks with a relatively small number of weights learn very quickly to beat complicated conventional programs, and, judging from the mistakes they exhibit, would perform substantially better yet if given a small amount of “symbolic” help.

Acknowledgements

We are grateful to Patrice Simard and Gerry Tesauro for helpful discussions, to Tim Casey for game records from the Internet Go Server, and to Geoff Hinton for CPU cycles. A condensed description of this work has previously appeared at the NIPS conference [24]. Support was provided by the McDonnell-Pew Center for Cognitive Neuroscience, SERC, NSERC, the Howard Hughes Medical Institute, and the Swiss National Fund.

References

[1] Rivest, R. (1993), invited talk, Conference on Computational Learning Theory and Natural Learning Systems, Provincetown, MA.

[2] Johnson, G. (1997), “To test a powerful computer, play an ancient game”, *The New York Times*, July 29, `http://www.cns.nyu.edu/~mechner/compgo/times/`

[3] Mechner, D. A. (1998), “All systems go”, *The Sciences*, **38**(1):32–37, `http://www.cns.nyu.edu/~mechner/compgo/sciences/`

[4] Fotland, D. (1993), “Knowledge representation in the Many Faces of Go”, `ftp://www.joy.ne.jp/welcome/igs/Go/computer/mfg.Z`

[5] Brügmann, B. (1993), “Monte Carlo Go”, `ftp://www.joy.ne.jp/welcome/igs/Go/computer/mcgo.tex.Z`

[6] Kirkpatrick, S., Gelatt Jr., C., and Vecchi, M. (1983), “Optimization by simulated annealing”, *Science*, **220**:671–680, reprinted in [25].

[7] Stoutamire, D. (1991), “Machine learning applied to Go”, Master’s thesis, Case Western Reserve University, `ftp://www.joy.ne.jp/welcome/igs/Go/computer/report.ps.Z`

[8] Enderton, H. D. (1991), "The Golem Go program", Tech. Rep. CMU-CS-92-101, Carnegie Mellon University, `ftp://www.joy.ne.jp/welcome/igs/Go/computer/golem.sh.Z`

[9] Sutton, R. S. and Barto, A. G. (1998), *Reinforcement Learning: An Introduction*, The MIT Press, Cambridge, MA.

[10] Samuel, A. L. (1959), "Some studies in machine learning using the game of checkers", *IBM Journal of Research and Development*, **3**:211–229.

[11] Watkins, C. (1989), *Learning from Delayed Rewards*, PhD thesis, University of Cambridge, England.

[12] Bertsekas, D. P. and Tsitsiklis, J. N. (1996), *Neuro-Dynamic Programming*, Athena Scientific, Belmont, MA.

[13] Tesauro, G. (1992), "Practical issues in temporal difference learning", *Machine Learning*, **8**:257.

[14] Robertie, B. (1992), "Carbon versus silicon: Matching wits with TD-Gammon", *Inside Backgammon*, **2**(2):14–22.

[15] Tesauro, G. (1994), "TD-gammon, a self-teaching backgammon program, achieves master-level play", *Neural Computation*, **6**(2):215–219.

[16] Geman, S. and Geman, D. (1984), "Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **6**, reprinted in [25].

[17] Newman, W. H. (1988), "Wally, a Go playing program", `ftp://www.joy.ne.jp/welcome/igs/Go/computer/wally.sh.Z`

[18] Dayan, P. (1993), "Improving generalization for temporal difference learning: The successor representation", *Neural Computation*, **5**(4):613–624.

[19] LeCun, Y., Boser, B., Denker, J., Henderson, D., Howard, R., Hubbard, W., and Jackel, L. (1989), "Backpropagation applied to handwritten zip code recognition", *Neural Computation*, **1**:541–551.

[20] Fukushima, K., Miyake, S., and Ito, T. (1983), "Neocognitron: A neural network model for a mechanism of visual pattern recognition", *IEEE Transactions on Systems, Man, and Cybernetics*, **13**, reprinted in [25].

[21] Fotland, D. (1994), personal communication.

[22] Enzensberger, M. (1996), "The integration of a priori knowledge into a Go playing neural network", `http://www.cgl.ucsf.edu/go/Programs/neurogo-html/NeuroGo.html`

[23] Dahl, F. A. (1999), "Honte, a Go-playing program using neural nets", `http://www.ai.univie.ac.at/icml-99-ws-games/papers/dahl.ps.gz`

[24] Schraudolph, N. N., Dayan, P., and Sejnowski, T. J. (1994), "Temporal difference learning of position evaluation in the game of Go", in *Advances in Neural Information Processing Systems*, Cowan, J. D., Tesauro, G., and Alspector, J., Eds. vol. 6, pp. 817–824, Morgan Kaufmann, San Francisco.

[25] Anderson, J. and Rosenfeld, E., Eds. (1988), *Neurocomputing: Foundations of Research*, MIT Press, Cambridge.