# Learning the game of Go with Internal Symmetry Networks

Alan Blair, School of Computer Science and Engineering, University of New South Wales

***Abstract*—We develop a cellular neural network architecture consisting of a large number of identical neural networks organised in a cellular array, and introduce a novel weight sharing scheme based on the principle of internal symmetry from particle physics. This Internal Symmetry Network is then trained by self-play and temporal difference learning to perform position evaluation for the game of Go. Lookahead search is achieved by parallelizing the network on a video card, and training an auxiliary network for heuristic pruning.**

## I. Introduction

There has been a growing interest in computation systems which enable global behavior to emerge from the interaction of local rules. Cellular automata are one example of this, but they are limited in having only a finite number of states available at each cell.

A Cellular Neural Network (CNN) is a collection of identical neural networks arranged in a cellular array [1]. CNNs are similar to cellular automata except that (1) the state space at each cell is continuous rather than discrete, and (2) the update rule is given by a neural network rather than a lookup table or other discrete mapping. Recent years have seen a growing interest in CNNs, particularly for image processing [2]. Generally, the weights are determined by hand-crafted design, or by global random search.

One task which seems very appropriate for this kind of architecture is the ancient game of Go. Standard search techniques generally run into trouble in the Go domain due to the very large branching factor [3]. For this reason, Go programs for a long time relied on symbolic reasoning rather than search. However, with the steady increase in desktop computing power a range of other approaches have recently become feasible – most notably, a new breed of strong programs based on UCT search [4].

Human players do make use of search methods in deciding their moves, but they prune the search tree very heavily. Their pruning mechanisms seem to rely on some kind of distributed computation — perhaps making use of low-level processing within the visual system. By mimicking this process with a cellular neural network (or similar architecture) we could potentially develop new Go programs employing heuristic alpha-beta search, but using neural networks for move evaluation and selection.

A number of Go-playing programs have previously been developed which incorporate neural networks in a variety of ways [5] including supervised learning [6] and temporal difference learning [7]. Cellular automata have also been used in this context [8].

The main distinguishing feature of our approach is the use of a cellular neural network architecture, and a novel weight sharing arrangement which we call an *Internal Symmetry Network*, inspired by the phenomenon of internal symmetry in particle physics.

Sections II, III and IV outline the rules of Go, the structure of Internal Symmetry Networks, and the input and output encoding for the network. Section V summarizes our initial experiments without tree search (previously reported in [9]). Sections VI and VII explain how tree search is made computationally feasible in this context, by parallelizing the network on a video card and training an auxiliary network for heuristic pruning. Sections VIII, IX and X describe preliminary experiments, modifications, evaluation and suggestions for future work.

## II. The game of Go

The rules of Go are relatively simple to state but the game is notoriously difficult to master. Two players take turns placing white and black stones on the vertices of a rectangular grid, each attempting to surround as much territory as possible without being captured. The standard size for a Go board is $19 \times 19$ but games are also sometimes played on boards of size $9 \times 9$ or $13 \times 13$.

A contiguous set of stones of the same color (i.e. connected along neighboring edges) is called a *group*. Empty vertices next to a group of stones are called *liberties* of that group. If the number of liberties of a group is reduced to zero at any point during the game (because the group has been surrounded by enemy stones), that group is *captured*. This means that all the stones of that group are removed from the board, leaving empty spaces where new stones can later be played. You are not allowed to play into a position which reduces the liberties of one of your own groups to zero (suicide) unless this move at the same time reduces the liberties of an enemy group to zero. In the latter case the enemy group is captured (thus creating at least one new liberty for your group). There is also a rule – called "ko" – which prevents the board from being returned to a position previously encountered in the same game. It is possible at any time to *pass* instead of making a move. When both players decide to pass one after the other, the game is over.

There are two popular scoring systems for Go: Chinese and Japanese. Under Chinese rules, you score one point for each of your own stones remaining on the board at the end, and one point for each vertex surrounded by your own stones.

Note: this paper was submitted in 2009 to IEEE-TCIAIG, but was not accepted for publication.

Vertices surrounded by a combination of white and black stones do not score a point for either player. The Japanese system is somewhat more complicated because you do not score a point for stones remaining on the board, but only for captured stones and for "territory", where "territory" can be loosely defined as "vertices which both players realize would be surrounded by your stones if the game were to continue".

Chinese rules are easier to implement computationally, and have therefore become the standard for many "computer-only" Go servers like CGOS.

## III. Internal Symmetry Networks

One of the interesting features of Go is its high degree of symmetry. Go has an approximate shift invariance, in the sense that the same arrangement of stones occurring in different places on the board is likely to lead to the next stone being played in the same position within this formation. (The invariance is only approximate because the strategy may be affected by the edges and corners of the board.)

To capitalise on this property, we use an architecture consisting of a large number of identical neural networks organised on a cellular array. Each cell in the array corresponds to a vertex on the Go board at which a stone may be played. If the size of the board is $n$-by-$n$, with $n = 2k + 1$, then the board can be considered as a lattice $\Lambda$ of vertices $\lambda = [a, b]$ with $-k \leq a, b \leq +k$. It will be convenient to denote by $\overline{\Lambda}$ the "extended" lattice which includes an additional row of vertices around the edges of the board, i.e. $\overline{\Lambda} = \{[a, b]\}_{-(k+1)\leq a,b\leq(k+1)}$.

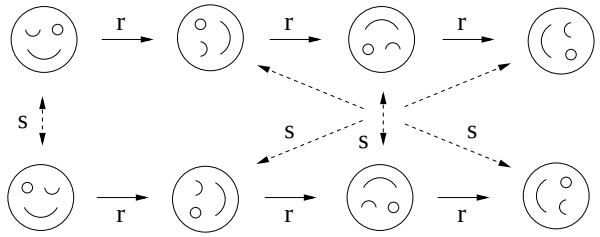


Fig. 1. The Dihedral group $\mathrm{D}_4$ of symmetries of the Go board

In addition to shift invariance, the Go board can be rotated or turned upside down in 8 different ways without affecting the rules. We therefore design our system in such a way that the network updates are invariant under this group of symmetries. As noted in [7], this can be accomplished by appropriate use of weight sharing [10], [11]. Here, we employ a novel weight sharing arrangement, which we call an Internal Symmetry Network.

The group $\mathcal{G}$ of symmetries of the Go board is the dihedral group $\mathrm{D}_4$ of order 8. This group is generated by two elements $r$ and $s$ — where $r$ represents a (counter-clockwise) rotation of $90^\circ$ and $s$ is a reflection in the vertical axis (see Fig. 1). The action of $\mathrm{D}_4$ on $\Lambda$ (or $\overline{\Lambda}$) is given by

$$\begin{aligned} r[a, b] &= [-b, a] \\ s[a, b] &= [-a, b] \end{aligned} \tag{1}$$

We will use $\mathcal{M}$ and $\mathcal{N}$ to denote neighborhood structures in the form of offset values:

$$\begin{aligned} \mathcal{M} &= \{[0, 0], [1, 0], [0, 1], [-1, 0], [0, -1]\}, \\ \mathcal{N} &= \mathcal{M} \cup \{[1, 1], [-1, 1], [-1, -1], [1, -1]\}. \end{aligned}$$

When viewed as offsets from a particular vertex, $\mathcal{M}$ represents the vertex itself plus the neighboring vertices to its EAST, NORTH, WEST and SOUTH; $\mathcal{N}$ includes these but adds also the diagonal vertices to the NORTH-EAST, NORTH-WEST, SOUTH-WEST and SOUTH-EAST. Assuming the action of $\mathcal{G}$ on $\mathcal{N}$ (or $\mathcal{M}$) is also given by Eqn(1), it is clear that for $g \in \mathcal{G}$, $\lambda \in \Lambda$ and $\nu \in \mathcal{N}$,

$$g(\lambda + \nu) = g(\lambda) + g(\nu).$$

Each cell $\lambda = [a, b] \in \Lambda$ has its own set of input, hidden and output units denoted by $\mathrm{I}^{[a,b]}$, $\mathrm{H}^{[a,b]}$ and $\mathrm{O}^{[a,b]}$. Each edge cell $\lambda = [a, b] \in \overline{\Lambda} \setminus \Lambda$ also has input and hidden units, but no output. The entire collection of input, hidden and output units $\mathcal{I}$, $\mathcal{H}$ and $\mathcal{O}$ for the whole network can thus be written as

$$\begin{aligned} \mathcal{I} &= \{\mathrm{I}^{[a,b]}\}_{[a,b]\in\overline{\Lambda}} \\ \mathcal{H} &= \{\mathrm{H}^{[a,b]}\}_{[a,b]\in\overline{\Lambda}} \\ \mathcal{O} &= \{\mathrm{O}^{[a,b]}\}_{[a,b]\in\Lambda} \end{aligned}$$

For an individual cell $\lambda \in \Lambda$, the neural network update equations are given by:

$$\mathrm{H}^\lambda \leftarrow \mathrm{H}(\mathcal{I})^\lambda = \tanh\big(\mathrm{B}_\mathrm{H} + \sum_{\nu\in\mathcal{N}} \mathrm{V}^\nu_\mathrm{HI}\mathrm{I}^{\lambda+\nu}\big)$$

$$\mathrm{O}^\lambda \leftarrow \mathrm{O}(\mathcal{I}, \mathcal{H})^\lambda = \phi\big(\mathrm{B}_\mathrm{O} + \sum_{\nu\in\mathcal{N}} \mathrm{V}^\nu_\mathrm{OI}\mathrm{I}^{\lambda+\nu} + \mathrm{V}^\nu_\mathrm{OH}\mathrm{H}^{\lambda+\nu}\big)$$

where $\phi$ is the sigmoid function $\phi(z) = 1/(1 + e^{-z})$.
In other words, each cell is connected to its nine neighboring cells (including diagonal neighbors) by input-to-hidden connections $\mathrm{V}_\mathrm{HI}$, hidden-to-output connections $\mathrm{V}_\mathrm{OH}$ and input-to-output "shortcut" connections $\mathrm{V}_\mathrm{OI}$. $\mathrm{B}_\mathrm{H}$ and $\mathrm{B}_\mathrm{O}$ represent the "bias" at the hidden and output units. We assume that for the edge cells ($\lambda \in \overline{\Lambda} \setminus \Lambda$), the hidden units $\mathrm{H}^\lambda$ remain identically zero, while the inputs $\mathrm{I}^\lambda$ take on special values to indicate that they are off the edge of the board.

Any element $g \in \mathcal{G}$ acts on the inputs $\mathcal{I}$ and output units $\mathcal{O}$ by simply permuting the cells:

$$\begin{aligned} g(\mathcal{I}) &= \{\mathrm{I}^{g([a,b])}\}_{[a,b]\in\overline{\Lambda}} \\ g(\mathcal{O}) &= \{\mathrm{O}^{g([a,b])}\}_{[a,b]\in\Lambda} \end{aligned}$$

In addition to permuting the cells, it is possible for $\mathcal{G}$ to act on some or all of the hidden unit activations within each cell, in a manner analogous to the phenomenon of *internal symmetry* in quantum physics. The group $\mathrm{D}_4$ has five irreducible representations, which we will label as *Trivial* (T), *Symmetrical* (S), *Diagonal* (D), *Chiral* (C) and *Faithful* (F). They are depicted visually in Fig. 2, and presented algebraically via these equations:

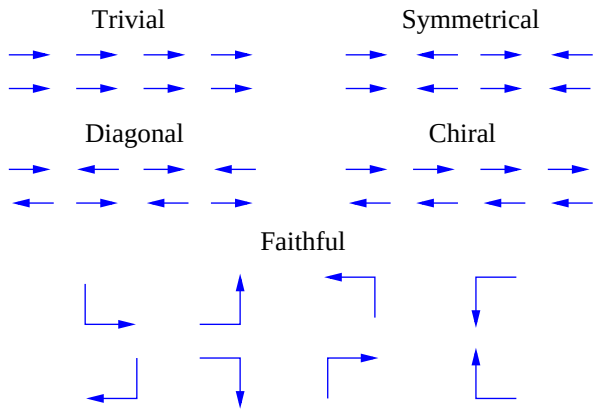


Fig. 2. The five irreducible representations of $D_4$

$$
\begin{array}{ll}
r(\mathrm{T}) = \mathrm{T}, & s(\mathrm{T}) = \mathrm{T} \\
r(\mathrm{S}) = \text{-S}, & s(\mathrm{S}) = \mathrm{S} \\
r(\mathrm{D}) = \text{-D}, & s(\mathrm{D}) = \text{-D} \\
r(\mathrm{C}) = \mathrm{C}, & s(\mathrm{C}) = \text{-C} \\
r(\mathrm{F})_1 = \text{-}\mathrm{F}_2, & s(\mathrm{F})_1 = \text{-}\mathrm{F}_1 \\
r(\mathrm{F})_2 = \mathrm{F}_1, & s(\mathrm{F})_2 = \mathrm{F}_2
\end{array}
$$

We consider, then, five types of hidden units, each with its own group action determined by the above equations. In general, an ISN can be characterized by a 5-tuple specifying the number of each type of hidden node at each cell $(i_T, i_S, i_D, i_C, i_F)$. Because it is 2-dimensional, hidden units corresponding to the Faithful representation will occur in pairs ($\mathrm{F}_1$, $\mathrm{F}_2$) with the group action "mixing" the activations of $\mathrm{F}_1$ and $\mathrm{F}_2$. The composite hidden unit activation for a single cell then becomes a cross product

$$\mathrm{H} = \mathrm{T}^{i_\mathrm{T}} \times \mathrm{S}^{i_\mathrm{S}} \times \mathrm{D}^{i_\mathrm{D}} \times \mathrm{C}^{i_\mathrm{C}} \times (\mathrm{F}_1 \times \mathrm{F}_2)^{i_\mathrm{F}}$$

with the action of $\mathcal{G}$ on $\mathcal{H}$ given by

$$g(\mathcal{H}) = \{\, g\,(\,\mathrm{H}^{\,g([a,b])})\}_{[a,b]\in\overline{\Lambda}}$$

We want the network to be invariant to the action of $\mathcal{G}$ in the sense that for all $g \in \mathcal{G}$,

$$
\begin{aligned}
g\big(\mathcal{H}(\mathcal{I})\big) &= \mathcal{H}(g(\mathcal{I})) \\
g\big(\mathcal{O}(\mathcal{I},\mathcal{H})\big) &= \mathcal{O}(g(\mathcal{I}), g(\mathcal{H}))
\end{aligned}
$$

This invariance imposes certain constraints on the weights of the network, which are outlined in the Appendix. In related work [12] we have conducted experiments to compare the effectiveness of different numbers and types of hidden units for various image processing tasks, and also explored the possibility of connecting each cell to itself and its four immediate neighbors with recurrent hidden-to-hidden connections. In the Go domain, we avoid recurrent connections because a strictly feed-forward architecture allows the network to be parallelized more effectively. Since the experiments are computationally intensive, it is not possible to test all combinations of hidden units. For the experiments reported here, we choose $i_\mathrm{T} = 4$, $i_\mathrm{S} = 2$, $i_\mathrm{D} = 2$, $i_\mathrm{C} = 0$, and $i_\mathrm{F} = 2$, making a total of 12 hidden units, 2310 connections per cell and 714 free parameters in the overall system.

TABLE I
REWARD TO BE GAINED FOR EACH VERTEX, BASED ON THE VALUE OF A CAPTURED STONE ($c$) AND A FINAL STONE ($s$)

| Ownership | State | $\mathrm{R}^s_+$ | $\mathrm{R}^s_\circ$ | $\mathrm{R}^s_\bullet$ |
|---|---|---|---|---|
| white | + (empty) | 1 | . | $1+c$ |
| white | ○ (filled) | $s$ | $s$ | $s+c$ |
| atari | + (empty) | 0 | $-c$ | $c$ |
| black | ● (filled) | $-s$ | $-(s+c)$ | $-s$ |
| black | + (empty) | -1 | $-(1+c)$ | . |

## IV. NETWORK INPUT AND OUTPUT

The architecture we have described so far is of a general nature and could be applied to other tasks such as image processing as well as board games like Go. The input and output encoding will depend on the task.

In the case of Go, we assign 14 inputs at each cell with a discrete encoding to indicate the color of the stone occupying that cell, and to provide some aggregate information about the liberties of the group to which that stone belongs (described in later Sections).

### A. Output Encoding for Evaluation Network

Our initial experiments involved one output unit per cell, trained to predict an appropriately scaled estimate of the expected reward associated with that cell. However, we eventually settled on a network with 7 outputs per cell, which together try to predict the expected reward under two different scoring systems.

Different scoring systems for Go can generally be characterized by two parameters $c$ and $s$, where $c$ is the reward for each enemy stone captured and $s$ is the reward for each live stone remaining on the board at the end of the game. (We assume a score of 1 for each vertex of territory that is owned but empty at the end of the game). In this framework, the Chinese scoring system corresponds to $c = 0, s = 1$ while the Japanese system roughly corresponds to $c = 1, s = 0$, but with special rules for ending the game early (discussed below).

The reward to be gained at each board location is shown in TABLE I. The location's current state is indicated by the subscripts at the top of each column, while the rows indicate its ownership and final state at the end of the game. Two of the table entries are blank, because we do not consider the possibility of a white stone becoming a white liberty, or a black stone becoming a black liberty.

We want our network to predict the reward for the two special cases $s$=1 and $s$=0, which are shown in Table II. We first consider the task of predicting $\mathrm{R}^1_+$, $\mathrm{R}^1_\circ$ and $\mathrm{R}^1_\bullet$. In theory, these three values could all be predicted with one output (since only one of them is applicable in any given situation). However, we choose instead to use three separate outputs $\mathrm{Z}_+$, $\mathrm{Z}_\circ$ and $\mathrm{Z}_\bullet$, in order to allow the network more

TABLE II
REWARD TO BE GAINED FOR EACH VERTEX, FOR THE CASES $s = 1$ AND $s = 0$

| | | $R^1_+$ | $R^1_\circ$ | $R^1_\bullet$ | $R^0_+$ | $R^0_\circ$ | $R^0_\bullet$ |
|---|---|---|---|---|---|---|---|
| white | + | 1 | . | $1+c$ | 1 | . | $1+c$ |
| white | ○ | 1 | 1 | $1+c$ | 0 | 0 | $c$ |
| atari | + | 0 | $-c$ | $c$ | 0 | $-c$ | $c$ |
| black | ● | -1 | $-(1+c)$ | -1 | 0 | $-c$ | 0 |
| black | + | -1 | $-(1+c)$ | . | -1 | $-(1+c)$ | . |

TABLE III
RELATIONSHIP BETWEEN REWARDS AND NETWORK OUTPUTS

| | | | |
|---|---|---|---|
| $R^1_+ =$ | $2\,Z_+ - 1$ | $R^0_+ =$ | $A^\circ_+ - A^\bullet_+$ |
| $R^1_\circ =$ | $(2+c)Z_\circ - (1+c)$ | $R^0_\circ =$ | $c(Z_\circ\text{-}1) - A_\circ$ |
| $R^1_\bullet =$ | $(2+c)Z_\bullet - 1$ | $R^0_\bullet =$ | $c\,Z_\bullet + A_\bullet$ |
| $Z_+ =$ | $(1 + R^1_+)/\,2$ | $A^\circ_+ =$ | $\max(\,R^0_+, 0)$ |
| | | $A^\bullet_+ =$ | $\max(\text{-}R^0_+, 0)$ |
| $Z_\circ =$ | $(1+c+R^1_\circ)/(2+c)$ | $A_\circ =$ | $c(Z_\circ\text{-}1) - R^0_\circ$ |
| $Z_\bullet =$ | $(1 + R^1_\bullet)/(2+c)$ | $A_\bullet =$ | $\text{-}c\,Z_\bullet + R^0_\bullet$ |

flexibility in computing these disparate values. The future status of a (currently) empty location is generally determined by the influence of the surrounding stones, while that of a filled location is determined by the likelihood of effecting or avoiding caputure.

It is convenient to linearly re-scale the network outputs $Z_+$, $Z_\circ$ and $Z_\bullet$ from [0,1] to the new ranges [-1,1], [$-(1+c)$,1] and [-1,$1+c$], respectively – since these are the natural ranges for the values of $R^1_+$, $R^1_\circ$ and $R^1_\bullet$ (top left of Table III). During training, the target values can be recovered by the inverse scaling (lower left of Table III).

In order to predict $R^0_+$, $R^0_\circ$ and $R^0_\bullet$, we add four additional outputs $A^\circ_+$, $A^\bullet_+$, $A_\circ$ and $A_\bullet$, and employ the transformations shown in the right column of Table III.
The target values for these seven outputs will then be as shown in Table IV. The current state of the vertex is indicated by the subscripts at the top of each column, while the rows indicate its ownership and final state at the end of the game.

Each of the seven outputs can informally be interpreted as a likelihood:

| output | interpreted as likelihood of ... |
|---|---|
| $Z_+$ | white gaining territory |
| $Z_\circ$ | white avoiding capture |
| $Z_\bullet$ | white effecting capture |
| $A^\circ_+$ | white making an eye |
| $A^\bullet_+$ | black making an eye |
| $A_\circ$ | white stone captured, leading to black eye |
| $A_\bullet$ | black stone captured, leading to white eye |

Note that, although there are 7 outputs, in practice only two or three of them need to be computed for each location ($Z_+$, $A^\circ_+$ and $A^\bullet_+$ if the location is empty, $Z_\circ$ and $A_\circ$ if it contains a white stone, $Z_\bullet$ and $A_\bullet$ if it contains a black stone).

TABLE IV
TARGET VALUES FOR THE SEVEN NETWORK OUTPUTS

| | | $Z_+$ | $Z_\circ$ | $Z_\bullet$ | $A^\circ_+$ | $A^\bullet_+$ | $A_\circ$ | $A_\bullet$ |
|---|---|---|---|---|---|---|---|---|
| White | + | 1 | . | 1 | 1 | 0 | . | 1 |
| White | ○ | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| Atari | + | $\frac{1}{2}$ | $\frac{1}{2+c}$ | $\frac{1+c}{2+c}$ | 0 | 0 | $\frac{c}{2+c}$ | $\frac{c}{2+c}$ |
| Black | ● | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Black | + | 0 | 0 | . | 0 | 1 | 1 | . |

### *B. Clarifying the Final Status*

The ending of a Go game has traditionally been by mutual agreement between the two players. In the case of Japanese rules, this "early" ending of the game has an impact on the final score – because it allows each player to claim the reward for capturing "dead" stones, without sacrificing the territory that would theoretically be lost in the process of capturing them. In the case of Chinese rules, ending the game early has no effect on the final score, but still makes it difficult to predict whether a given location will be filled or empty at the end of the game. In order to train our networks, we need to have a well-defined outcome so that the final status of each location can be sensibly predicted – not only in terms of territory, but also in terms of whether it is filled or empty. We achieve this by adopting a novel scoring system, for training purposes, which is somewhere between the Chinese and Japanese systems, by awarding 0.4 points for each captured stone, and 0.4 points for each stone remaining on the board at the end of the game (i.e. setting the above scoring parameters to $c = s = 0.4$ ). This scoring system encourages each player to chip away at the opponent's liberties during the endgame, without filling in any of their own liberties unnecessarily. Thus, all remaining blank areas will be carved up into isolated eyes, with each player trying to maximise their own eyes while minimizing those of the opponent.

### *C. Input Encoding*

We allocate 14 input units to each board location. Exactly one of these inputs will be "active" for any given location and time step. The active unit will be set to 1, while the other 13 units will be set to 0. This kind of "1-in-$n$" encoding facilitates rapid computation.

If a white stone is present, one of the inputs in the range 1-6 will be active. If a black stone is present, an input in the range 7-12 will be active. Input 13 indicates that this location is empty (no stone), while input 14 indicates that this location is off the edge of the board.

When a white or black stone is present, the choice of input within the range 1-6 or 7-12 is intended to provide the

network with some aggregate information about the liberties of the group to which that stone belongs.

In our early experiments, each stone was classified into one of 6 classes, depending on the total number of liberties of its group. This led to poor network performance, because all liberties were treated equally. We realised it would be advantageous to modify the classification by weighting each liberty according to (a) the likelihood of it being retained as territory, and (b) the likelihood of it remaining a liberty until the end of the game, thus becoming an eye. Since these likelihoods have already been estimated by the network at the previous timestep, we can use this information to classify groups at the current timestep. Specifically, for each white (resp. black) group, we define the *eye-likelihood* $\Sigma$Z to be the sum of $Z_+$ (resp. (1-$Z_+$)), and the *expansion-likelihood* $\Sigma$A be the sum of $A^{\circ}_{+}$ (resp. $A^{\bullet}_{+}$) for all liberties of that group. The group can then be classified into one of 6 classes, as shown in Fig. 3.

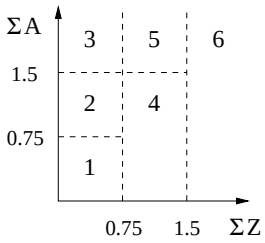


Fig. 3. Categorization of groups into classes, based on eye-likelihood $\Sigma$Z and expansion-likelihood $\Sigma$A.

Roughly speaking, $\Sigma$Z estimates the number of eyes that are likely to be made from current liberties of the group, while $\Sigma$A estimates the number of "openings", i.e. potential avenues for expansion, or connection to other groups. In this context, the six categories can roughly be characterized as:

Class 1: no eyes and no openings
Class 2: no eyes, and only one opening
Class 3: no eyes, but at least two openings
Class 4: one eye, but no opening
Class 5: one eye, plus at least one opening
Class 6: at least two eyes

### D. Implicit Recurrence and Adjustment

Although the network itself is feed-forward, the use of outputs from the previous time step for categorization effectively adds a kind of "implicit recurrence" to the system. Thus, even though each output cell is *directly* dependent only on the stones in a local neighborhood, the categories 1 to 6 (above) implicitly give it access to non-local information about the number (and type) of liberties for the groups to which these stones belong.

On account of being computed at the previous timestep, the likelihoods Z and A are always slightly out of date, because two new stones have been placed on the board, and captures may have taken place, creating new liberties for the capturing group(s). To adjust for this, we slightly increment or decrement the likelihoods Z and A, for locations adjacent to the newly-placed stones. For each group, $\Sigma$Z and $\Sigma$A are also augmented to include the likelihood of capturing a neighboring enemy group (estimated conservatively as the *minimum* value of Z, among the stones in the enemy group).

## V. Network Training

The Evaluation Network was trained by self-play and temporal difference learning [13], [14], [7] in the form of TD($\lambda$) with $\lambda = 0.9$. Each output was trained using cross entropy minimization, with a learning rate of 0.000005. Although this learning rate may appear small, the massive weight sharing in the Internal Symmetry Network causes differentials to accumulate at every single vertex, therefore adding up to a substantial weight adjustment by the end of the game.

The overall board evaluation $R$ is the sum of the expected rewards for all the individual board locations. Moves were chosen according to a Boltzman distribution – meaning that the probability of each (legal) move is proportional to $e^{\beta R}$, where $R$ is the evaluation of the board resulting from that move. The Boltzman constant $\beta$ was set to 4 during the training.

The shortcut connections (i.e. direct from input to output) were trained in a preliminary phase, to provide a linear player with a basic level of functionality (and to ensure that the games would eventually terminate). All the weights of the network were then opened up for 860,000 games of training on a $9 \times 9$ board. The training time was approximately half a second for each game, or five days in total, on a 2.66 GHz Mac Pro.

For evaluation, networks at intervals of 20K were extracted and played 10 games against each other pairwise in a round-robin tournament. For the tournament, moves were again selected from a Boltzman distribution but with $\beta = 20$. Standard Chinese rules were used, with a komi of 3.5. The results are shown in Fig. 4 where we see a noisy but generally upward trend in performance.

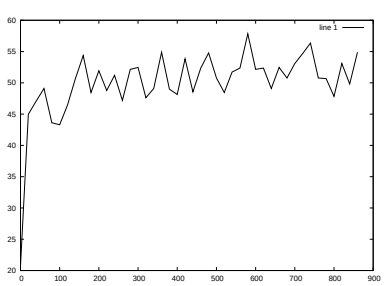


Fig. 4. Percentage of wins in round robin tournament, for networks from 0 to 860K

Our Internal Symmetry Network architecture has the advantage that, even when trained only on the $9 \times 9$ board size, the network can then be made to play on any sized board without changing the actual weights.

We extracted the best network (at epoch 580K) and played several games against it on boards of size $9 \times 9$ and $19 \times 19$. Generally, the network can be observed to perform captures, threats, blocking moves, etc. and its choice of moves seems quite reasonable considering that its decisions are based only on a single board evaluation, with no lookahead.

The next two sections describe our attempts to implement alpha-beta search with heuristic pruning. The necessary speed is achieved by parallelizing the network on a video card.

## VI. Parallelization

We have parallelized our neural network code on an NVIDIA GeForce 8800 graphics card, using the CUDA programming framework.

The CUDA framework requires an overall computation to be divided into a large number of parallel *threads*, with the threads organized into *blocks*. An exact multiple of 64 threads per block is recommended, in order to gain the advantages of "coalesced" memory access. Memory on the device is divided into *shared* memory – accessible only to threads within a particular block – and *global* memory – accessible to all threads. Global memory is slower to access than shared memory, but still faster than accessing the memory of the host machine.

In our case, the network activations are computed in two separate kernel invocations – one for the hidden layer, and one for the output layer. We always use exactly 64 threads per block.

For the hidden layer invocation, the number of blocks is equal to the number of hidden units multiplied by $\lceil \frac{n^2}{64} \rceil$, where $n$ is the board size. Each block is assigned to compute one of the 12 hidden units, for a set of 64 contiguous board locations. The weights associated with the relevant hidden unit are loaded into shared memory, as well as the inputs for the 64 assigned board locations, together with the 32 locations above and below them (to ensure that all neighboring board locations are included). Because of the 1-in-$n$ input encoding, the hidden unit computation for each thread consists of a series of array lookups and floating-point additions, followed by a single application of the hyperbolic tangent function. The hidden unit activations are then stored to global memory.

The output layer invocation is similar. Although there are 7 outputs, in practice only 3 of them need to be computed for any particular board location, so the number of threads is $3 \times \lceil \frac{n^2}{64} \rceil$. Each thread first combines the input-to-output weights with a series of array lookups and floating point additions; it then reads the relevant hidden unit activations from global memory, multiplies each of them by an appropriate weight, and applies the sigmoid function to the grand total.

This parallel implementation increases the speed of evaluation to 7000 positions per second for the $19 \times 19$ board size (compared to 1600 positions per second for the non-parallel implementation). This is fast enough to allow a full alpha-beta search to depth 3, or a heuristically pruned search to depth 5.

## VII. Selection Network and Search

The evaluation speed of 7000 positions per second is fast enough to allow alpha-beta search. But, if we want to search more than 2 or 3 ply, we need to prune the tree heuristically in order to tame the very large branching factor. For this purpose, we train an auxiliary *Selection Network* (SelNet), to be used in conjunction with the *Evaluation Network* (EvalNet) described in Sections IV and V.

The inputs to the SelNet are the same as those for the EvalNet, but the SelNet has only one output unit for each board location. While the EvalNet is intended to give an absolute evaluation for a single board position, the SelNet is indended to give a *relative* estimate, for all board locations simultaneously, of the incremental value of playing a stone into that location. In other words, the task of the SelNet is to estimate how much the score would increase, for each possible move, compared to the "benchmark" score of choosing to PASS (thereby leaving the board as it is). Each output of the SelNet is taken as a raw value, with no sigmoid or other transfer function applied.

Note also that the SelNet is applied *before* making the current move, while the EvalNet is applied *afterwards*. This means that the adjustments to Z and A described in Section IV-D only need to take account of one new stone when applying the SelNet, rather than two new stones when applying the EvalNet.

In order to generate training data for the SelNet, the EvalNet is made to play 10,000 games against itself on a $13 \times 13$ board, with Boltzman constant $\beta = 8$. As each move is played, the EvalNet's evaluation of all possible (alternative) moves are dumped to a text file.

Let $S_\lambda$ be the output of the SelNet at location $\lambda$, and let $R_\lambda$ be the (aggregated) output of the EvalNet, which estimates the value of the board position resulting from a stone played at location $\lambda$. The EvalNet and SelNet thereby determine Boltzman distributions $\{p_\lambda\}$ and $\{q_\lambda\}$ given by

$$p_\lambda = \frac{e^{\beta R_\lambda}}{\sum_\mu e^{\beta R_\mu}}, \qquad q_\lambda = \frac{e^{\beta S_\lambda}}{\sum_\mu e^{\beta S_\mu}}.$$

The sum is over all $\lambda \in \text{PASS} \cup \Lambda$, where $R_{\text{PASS}}$ is the evaluation of the board in its current state, and $S_{\text{PASS}} = 0$ by convention.

The SelNet is then trained by supervised learning. The cost function is the Külback-Leibler divergence between the two Bolzman distributions, given by

$$\begin{aligned} \mathrm{D}_{\mathrm{KL}} &= \sum_\lambda p_\lambda \log(\frac{p_\lambda}{q_\lambda}) \\ &= \sum_\lambda p_\lambda \log p_\lambda - \sum_\lambda p_\lambda \log(\frac{e^{\beta S_\lambda}}{\sum_\mu e^{\beta S_\mu}}) \\ &= \sum_\lambda p_\lambda \log p_\lambda - (\sum_\lambda p_\lambda \beta S_\lambda) + \log(\sum_\mu e^{\beta S_\mu}) \end{aligned}$$

We then have

$$\frac{\partial}{\partial S_\lambda} \mathrm{D}_{\mathrm{KL}} = -\beta p_\lambda + \frac{\beta e^{\beta S_\lambda}}{\sum_\mu e^{\beta S_\mu}} = \beta(q_\lambda - p_\lambda)$$

This makes sense intuitively, because it concentrates the effort on moves for which either $p_\lambda$ or $q_\lambda$ is large (i.e. moves that are ranked highly by either the EvalNet or SelNet).

## VIII. Preliminary Results and Modifications

Our network played on the $19 \times 19$ CGOS server for several days, under the name `FishNet1`. Unfortunately, it performed very poorly, with a rating of about 850 (see Table V). There are only about a dozen programs which play regularly on the $19 \times 19$ CGOS server. Two of the low-ranking players among them are `AverageLib` – which uses a simple heuristic function – and `AmigoGtp` – an updated version of a program originally written for the Commodore Amiga. `FishNet1` only won about 20% of its games against `AverageLib` and 15% of its games against `AmigoGtp`. A careful study of these games gave us clues as to how the network's performance might be improved. Fig. 5 shows a game between `FishNet1` and `AmigoGtp`. We also analysed the self-play games (without lookahead) by generating graphs in the ($\Sigma$Z,$\Sigma$A) plane for groups that ultimately survived, and for those which were later captured. From this, we made two important observations:

### A. Changing the way Z-score is computed

We noticed that, in the self-play games, many groups ultimately died, despite satisfying the condition that the eye-likelihood $\Sigma\mathrm{Z} > 1.5$. On closer inspection, it turned out that these groups had a large number of "low-quality" liberties with little or no chance of becoming eyes. Groups of this kind were also observed in games between `FishNet1` and `AmigoGtp` (see Fig. 5). Although it is obvious (to humans!) that the likelihood of making an eye from these liberties is effectively zero, the network was giving estimated likelihoods in the range of 0.05 to 0.15 for each of them. A large number of these small likelihoods can then aggregate to a combined likelihood $\Sigma$Z greater than 1.5. To remedy this, we define a modified Z-score $\Sigma'$Z, by including in the sum only those liberties whose likelihood of becoming an eye is greater than 0.2. This modiified Z-score, with our original thresholds of 0.75 and 1.5, turned out to give a much better prediction of which groups would ultimately survive.

### B. Modification to A-score

We now turn our attention to the classification of groups for which $\Sigma'\mathrm{Z} < 0.75$ . In our original scheme, these groups were classified according to their "expansion-likelihood" or A-score $\Sigma$A, with thresholds of 0.75 and 1.5 .

Fig. 6 plots the likelihood of eventual capture as a function of A-score, in the self-play games, for groups with $\Sigma'\mathrm{Z} <$ $0.75$ (categorized according to the number of stones in the group).

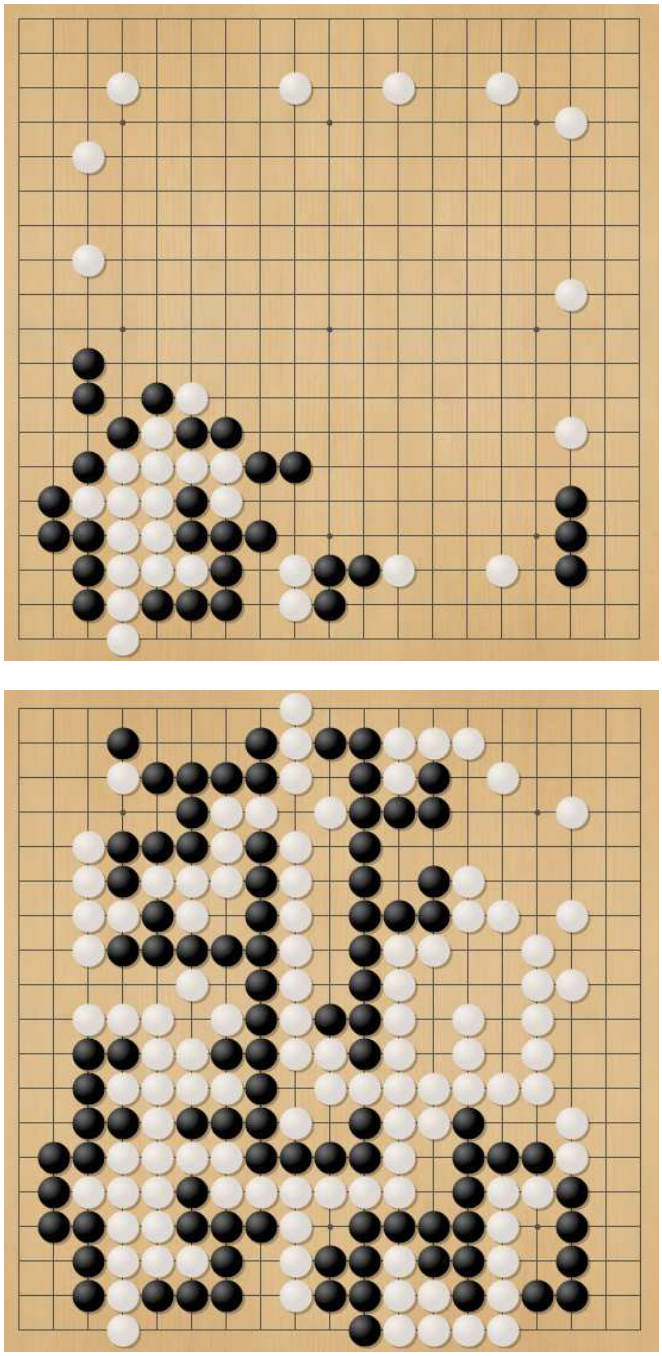

Fig. 5. Game between `FishNet1` (black) and `AmigoGtp` (white) at move 61 (above) and 214 (below).

We see that (1) our original $\Sigma$A thresholds of 0.75 and 1.5 were much too low to give a useful prediction, and (2) the thresholds ought to vary according to the number of stones in the group (with higher thresholds for larger groups).

This also explains certain deleterious behaviors exhibited by the network. Looking at the early stages of the game in Fig. 5, we see that `AmigoGtp` (white) has placed sin-

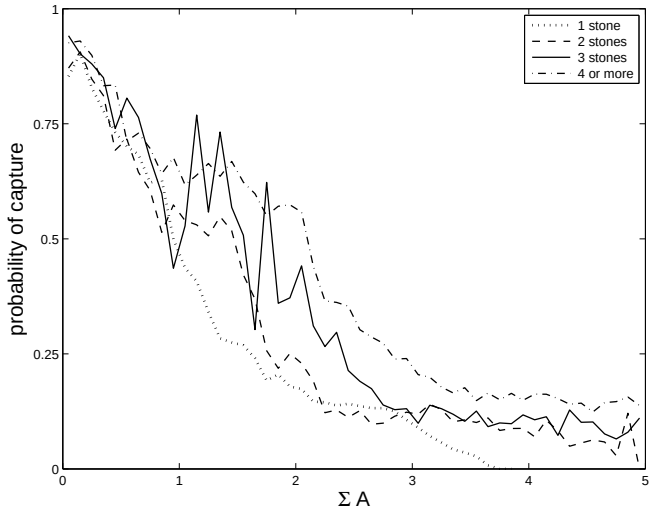


Fig. 6. Likelihood of eventual capture, as a function of ΣA, for groups with $\Sigma'Z < 0.75$ (categorized according to the number of stones in the group).

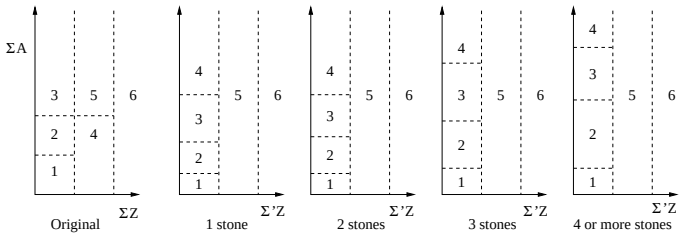


Fig. 7. Categorization of groups, based on the (modified) eye-likelihood $\Sigma^{(\prime)}Z$, the expansion-likelihood ΣA and the number of stones in the group.

gle stones nicely around the edge of the board, whereas `FishNet1` (black) has avoided placing stones in isolation and instead places its stones together in groups of 2 or 3. This is because a 2-stone group in an open area of the board will fall into Class 5, whereas a 1-stone group would fall into Class 3 (with an A-score of approximately 2.6).

In general, a single stone in an open area of the board has a good chance of survival and ought to be rated highly. In contrast, the snake-like (black) groups appearing in Fig. 5 have virtually no chance of survival, but because of the large number of low-quality liberties they may easily accrue an expansion likelihood higher than 1.5. Under our original scheme, these two situations are "confused" into the same class (3), causing the network to have an optimistic view of the snake-like group, and a pessimistic view of the single-stone group. As a result, in the mid-game the network fails to adequately defend some of its groups, thereby allowing them to be captured.

Unfortunately, the performance of the network virtually did not improve at all in moving from no lookahead to 3- or 5-move lookahead (see Table V). In other words, the misconceptions in the network's evaluation (discussed above) are quite fundamental, and are not of the kind that can be "cured" by deeper search.

TABLE V
RATINGS ON $19 \times 19$ CGOS SERVER

| Player | No lookahead | 5-move lookahead |
|---|---|---|
| Fishnet 1 | $850 \pm 50$ | $850 \pm 50$ |
| Fishnet 2 | $850 \pm 50$ | $1050 \pm 50$ |

## IX. MODIFIED NETWORK

A new evaluation network was trained for 400K epochs, using the modified Z-score and revised A-score thresholds outlined in Section VIII. The new thresholds, shown in Fig. 7, were chosen at the places where the graphs in Fig. 6 cross the boundaries of 0.75, 0.5 and 0.25 (probability of capture).

To make the training more stable, we introduced the following *Negatrain* enhancement: in our original experiments, for training purposes, the network only considered the evaluation of every *second* board position, i.e. the position after each move by a particular player. In the new experiments, the position after *every* move was considered; the likelihoods generated by the opponent network were used for the positions following each opponent move. This did indeed seem to make the training more stable, and allowed us to increase the learning rate to 0.0001. The parameter $\lambda$ was also increased from 0.9 to 0.95 so that the new network would "look equally far into the future".

As before, 10,000 self-play games by the new network were used to train a selection network, employing the same modified Z-score and revised A-score thresholds.

The new network played on CGOS under the name `Fishnet2`. When playing with no lookahead, it achived approximately the same rating as `Fishnet1`. However, with 5-move lookahead, the rating increased by 200 ELO, from 850 to 1050 (see Table V). `FishNet2` (with lookahead) wins approximately 85% of its games against `AverageLib`, and 45% of its games against `AmigoGtp`.

Fig. 8 shows a game between `FishNet2` and `AmigoGtp`. We see that the level of play represents a substantial improvement over `Fishnet1`, both in the opening and in the later stages of the game.

## X. CONCLUSION

We have demonstrated that Internal Symmetry Networks can be trained to play the game of Go using self-play and temporal difference learning. Lookahead search is achived by parallelizing the network on a video card, and training an auxiliary network for heuristic pruning.

The inclusion of 5-move lookahead search increases the performance by approximately 200 ELO, giving hope that deeper search enabled by faster computation and greater parallelization may lead to continued improvement. We have identified two other main areas for future research:

1. Concerning the implicit recurrence of the system (Section IV-D) we currently compute the likelihoods Z and A from the previous board position, and try to account for the two subsequent moves by making slight ajustments to the values at locations adjacent to the two new stones. This

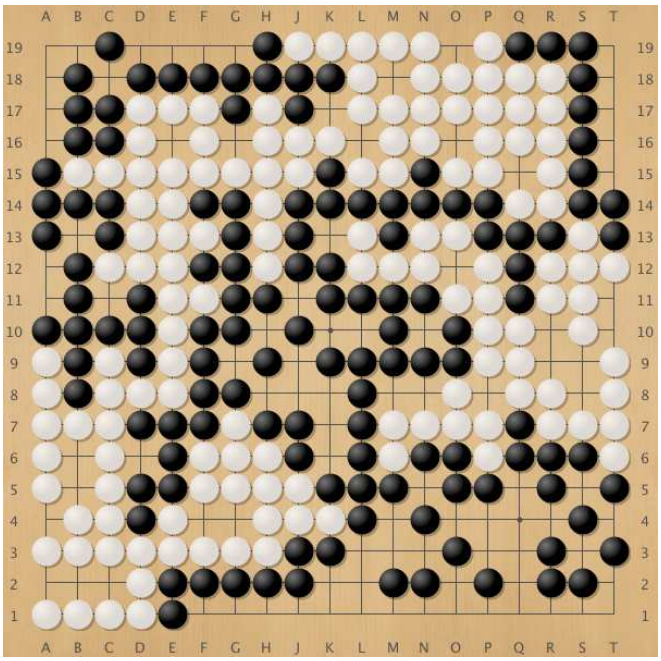

Fig. 8. Game between `FishNet2` (black) and `AmigoGtp` (white).

gives a sensible result in most cases, but seems to cause problems in certain specific situations, where strategic battles are played out on the edge of the board. In future work, we plan to use the network itself to compute the updated likelihoods in a more methodical way.

2. In our existing architecture, each hidden unit is connected only to its neighbors in a $3 \times 3$ window. We have recently found that, for image processing tasks, the performance can be improved by extending this to a $5 \times 5$ window (with appropriate symmetry constraints). We plan to soon apply the same modification in the Go domain.

## APPENDIX: WEIGHT SHARING

The constraints on the various network connections are outlined below – with neighborhood relationships abbreviated to E (EAST), N (NORTH), W (WEST), S (SOUTH), NE (NORTH EAST), NW (NORTH WEST), SW (SOUTH WEST), SE (SOUTH EAST) and O (ORIGINAL).

$$\mathrm{V}^{\nu}_{\mathrm{OH}} = \left[ \begin{array}{cccccc} \mathrm{V}^{\nu}_{\mathrm{OT}} & \mathrm{V}^{\nu}_{\mathrm{OS}} & \mathrm{V}^{\nu}_{\mathrm{OD}} & \mathrm{V}^{\nu}_{\mathrm{OC}} & \mathrm{V}^{\nu}_{\mathrm{OF_1}} & \mathrm{V}^{\nu}_{\mathrm{OF_2}} \end{array} \right]$$

$$\mathrm{V}^{\nu}_{\mathrm{HI}} = \left[ \begin{array}{cccccc} \mathrm{V}^{\nu}_{\mathrm{TI}} & \mathrm{V}^{\nu}_{\mathrm{SI}} & \mathrm{V}^{\nu}_{\mathrm{DI}} & \mathrm{V}^{\nu}_{\mathrm{CI}} & \mathrm{V}^{\nu}_{\mathrm{F_1I}} & \mathrm{V}^{\nu}_{\mathrm{F_2I}} \end{array} \right]^{\mathrm{T}}$$

$$\mathrm{V}^{\mathrm{E}}_{\mathrm{OI}} = \mathrm{V}^{\mathrm{N}}_{\mathrm{OI}} = \mathrm{V}^{\mathrm{W}}_{\mathrm{OI}} = \mathrm{V}^{\mathrm{S}}_{\mathrm{OI}} \ , \quad \mathrm{V}^{\mathrm{NE}}_{\mathrm{OI}} = \mathrm{V}^{\mathrm{NW}}_{\mathrm{OI}} = \mathrm{V}^{\mathrm{SW}}_{\mathrm{OI}} = \mathrm{V}^{\mathrm{SE}}_{\mathrm{OI}}$$
$$\mathrm{V}^{\mathrm{E}}_{\mathrm{OT}} = \mathrm{V}^{\mathrm{N}}_{\mathrm{OT}} = \mathrm{V}^{\mathrm{W}}_{\mathrm{OT}} = \mathrm{V}^{\mathrm{S}}_{\mathrm{OT}} \ , \quad \mathrm{V}^{\mathrm{NE}}_{\mathrm{OT}} = \mathrm{V}^{\mathrm{NW}}_{\mathrm{OT}} = \mathrm{V}^{\mathrm{SW}}_{\mathrm{OT}} = \mathrm{V}^{\mathrm{SE}}_{\mathrm{OT}}$$
$$\mathrm{V}^{\mathrm{E}}_{\mathrm{TI}} = \mathrm{V}^{\mathrm{N}}_{\mathrm{TI}} = \mathrm{V}^{\mathrm{W}}_{\mathrm{TI}} = \mathrm{V}^{\mathrm{S}}_{\mathrm{TI}} \ , \quad \mathrm{V}^{\mathrm{NE}}_{\mathrm{TI}} = \mathrm{V}^{\mathrm{NW}}_{\mathrm{TI}} = \mathrm{V}^{\mathrm{SW}}_{\mathrm{TI}} = \mathrm{V}^{\mathrm{SE}}_{\mathrm{TI}}$$
$$\mathrm{V}^{\mathrm{O}}_{\mathrm{OF_1}} = \mathrm{V}^{\mathrm{O}}_{\mathrm{OF_2}} = \mathrm{V}^{\mathrm{O}}_{\mathrm{F_1I}} = \mathrm{V}^{\mathrm{O}}_{\mathrm{F_2I}} = 0$$
$$\mathrm{V}^{\mathrm{E}}_{\mathrm{OF_1}} = \mathrm{V}^{\mathrm{N}}_{\mathrm{OF_2}} = -\mathrm{V}^{\mathrm{W}}_{\mathrm{OF_1}} = -\mathrm{V}^{\mathrm{S}}_{\mathrm{OF_2}} = \mathrm{V}^{\mathrm{E}}_{\mathrm{F_1I}} = \mathrm{V}^{\mathrm{N}}_{\mathrm{F_2I}} = -\mathrm{V}^{\mathrm{W}}_{\mathrm{F_1I}} = -\mathrm{V}^{\mathrm{S}}_{\mathrm{F_2I}}$$
$$\mathrm{V}^{\mathrm{E}}_{\mathrm{OF_2}} = \mathrm{V}^{\mathrm{N}}_{\mathrm{OF_1}} = \mathrm{V}^{\mathrm{W}}_{\mathrm{OF_2}} = \mathrm{V}^{\mathrm{S}}_{\mathrm{OF_1}} = \mathrm{V}^{\mathrm{E}}_{\mathrm{F_2I}} = \mathrm{V}^{\mathrm{N}}_{\mathrm{F_1I}} = \mathrm{V}^{\mathrm{W}}_{\mathrm{F_2I}} = \mathrm{V}^{\mathrm{S}}_{\mathrm{F_1I}} = 0$$
$$\mathrm{V}^{\mathrm{NE}}_{\mathrm{OF_1}} = -\mathrm{V}^{\mathrm{NW}}_{\mathrm{OF_1}} = -\mathrm{V}^{\mathrm{SW}}_{\mathrm{OF_1}} = \mathrm{V}^{\mathrm{SE}}_{\mathrm{OF_1}} = \mathrm{V}^{\mathrm{NE}}_{\mathrm{OF_2}} = \mathrm{V}^{\mathrm{NW}}_{\mathrm{OF_2}} = -\mathrm{V}^{\mathrm{SW}}_{\mathrm{OF_2}} = -\mathrm{V}^{\mathrm{SE}}_{\mathrm{OF_2}}$$
$$\mathrm{V}^{\mathrm{NE}}_{\mathrm{F_1I}} = -\mathrm{V}^{\mathrm{NW}}_{\mathrm{F_1I}} = -\mathrm{V}^{\mathrm{SW}}_{\mathrm{F_1I}} = \mathrm{V}^{\mathrm{SE}}_{\mathrm{F_1I}} = \mathrm{V}^{\mathrm{NE}}_{\mathrm{F_2I}} = \mathrm{V}^{\mathrm{NW}}_{\mathrm{F_2I}} = -\mathrm{V}^{\mathrm{SW}}_{\mathrm{F_2I}} = -\mathrm{V}^{\mathrm{SE}}_{\mathrm{F_2I}}$$
$$\mathrm{V}^{\mathrm{E}}_{\mathrm{OS}} = -\mathrm{V}^{\mathrm{N}}_{\mathrm{OS}} = \mathrm{V}^{\mathrm{W}}_{\mathrm{OS}} = -\mathrm{V}^{\mathrm{S}}_{\mathrm{OS}} \ , \quad \mathrm{V}^{\mathrm{NE}}_{\mathrm{OD}} = -\mathrm{V}^{\mathrm{NW}}_{\mathrm{OD}} = \mathrm{V}^{\mathrm{SW}}_{\mathrm{OD}} = -\mathrm{V}^{\mathrm{SE}}_{\mathrm{OD}}$$
$$\mathrm{V}^{\mathrm{E}}_{\mathrm{SI}} = -\mathrm{V}^{\mathrm{N}}_{\mathrm{SI}} = \mathrm{V}^{\mathrm{W}}_{\mathrm{SI}} = -\mathrm{V}^{\mathrm{S}}_{\mathrm{SI}} \ , \quad \mathrm{V}^{\mathrm{NE}}_{\mathrm{DI}} = -\mathrm{V}^{\mathrm{NW}}_{\mathrm{DI}} = \mathrm{V}^{\mathrm{SW}}_{\mathrm{DI}} = -\mathrm{V}^{\mathrm{SE}}_{\mathrm{DI}}$$
$$\mathrm{V}^{\mu}_{\mathrm{OD}} = \mathrm{V}^{\mu}_{\mathrm{DI}} = 0, \quad \mu \in \{\mathrm{O}, \mathrm{E}, \mathrm{N}, \mathrm{W}, \mathrm{S}\}$$
$$\mathrm{V}^{\nu}_{\mathrm{OS}} = \mathrm{V}^{\nu}_{\mathrm{SI}} = 0, \quad \nu \in \{\mathrm{O}, \mathrm{NE}, \mathrm{NW}, \mathrm{SW}, \mathrm{SE}\}$$
$$\mathrm{V}^{\nu}_{\mathrm{OC}} = \mathrm{V}^{\nu}_{\mathrm{CI}} = 0, \quad \nu \in \{\mathrm{O}, \mathrm{E}, \mathrm{N}, \mathrm{W}, \mathrm{S}, \mathrm{NE}, \mathrm{NW}, \mathrm{SW}, \mathrm{SE}\}$$

## REFERENCES


[1] L.Chua & L.Yang, 1988. Cellular Neural Networks: Theory, *IEEE Trans. on Circuits and Systems* **35**(10), pp. 1257–1272.

[2] L.Chua & T.Roska, 2002. *Cellular Neural Networks and Visual Computing*, Cambridge University Press.

[3] J.Burmeister & J.Wiles, 1995. The challenge of Go as a domain for AI research, *Proceedings of the Third Australian and New Zealand Conference on Intelligent Information Systems*.

[4] S.Gelly & Y.Wang, 2006. Exploration exploitation in Go: UCT for Monte-Carlo Go, Proceedings of Neural Information Processing Systems Conference.

[5] M.Enzenberger, 1996. The integration of a priori knowledge into a Go playing neural network, `www.cgl.ucsf.edu/go/Programs/neurogo-html/NeuroGo.html`

[6] F.A.Dahl, 2001. Honte, a Go-playing program using neural networks, in J.Fürnkranz & M.Kubat (Eds.) *Machines that learn to Play Games*, Chapter 10, pp. 205–223. Huntington.

[7] N.Schraudolph, P.Dayan & T.Sejnowski, 1994. Temporal difference learning of position evaluation in the game of Go, In *Advances in Neural Information Processing 6*, Morgan Kaufmann, 817–824.

[8] S.Welsh & T.Bossomaier, 1999. Evolving cellular automata tools for the game of Go, *Proceedings of the Third Australia-Japan Joint Workshop on Intelligent and Evolutionary Systems*, 159–166.

[9] A.Blair, 2008. Learning Position Evaluation for Go with Internal Symmetry Networks, *Proc. 2008 IEEE Symposium on Computational Intelligence and Games*, pp. 199-204.

[10] Y.LeCun, B.Boser, J.Denker, D.Henderson, R.Howard, W.Hubbard & L.Jackel, 1989. Backpropagation applied to handwritten character recognition, *Neural Computation* **5**, 541–551.

[11] T.Schaul & J.Schmidhuber, 2008. A Scalable Neural Network Architecture for Board Games, *Proc. 2008 IEEE Symposium on Computational Intelligence and Games*.

[12] A.Blair & G.Li, 2009. Training of Recurrent Internal Symmetry Networks by Backpropagation, *Proc. 2009 International Joint Conference on Neural Networks* (to appear).

[13] R.Sutton, 1988. Learning to Predict by the Methods of Temporal Differences, *Machine Learning* **3**, 9–44.

[14] G.Tesauro, 1992. Practical Issues in Temporal Difference Learning, *Machine Learnin* **8**, 257–277.