

Learning Position Evaluation for Go with Internal Symmetry Networks

Alan Blair, *Member, IEEE*

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.

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. CNNs are similar to a 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.

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 [1]. 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 [2].

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 [3] including supervised learning [4] and temporal difference learning [5]. Cellular automata have also been used in this context [6].

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, because it is inspired by the phenomenon of internal symmetry from particle physics.

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 19×19 but games are also sometimes played on boards of size 9×9 or 13×13 .

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. 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 Λ of vertices $\lambda = [a, b]$ with $-k \leq a, b \leq +k$. It will be convenient to denote by $\bar{\Lambda}$ the “extended” lattice which includes an additional row of vertices around the edges of the board, i.e. $\bar{\Lambda} = \{[a, b]\}_{-(k+1) \leq a, b \leq (k+1)}$.

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 [5], this can be accomplished by appropriate use of weight sharing [7]. 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 D_8 of order 8. This group is generated by two elements r and s — where r represents a rotation of 90° and s is a reflection in the vertical axis. The action of D_8 on Λ (or $\bar{\Lambda}$) is given by

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

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 $I^{[a, b]}$, $H^{[a, b]}$ and $O^{[a, b]}$. Each edge cell $\lambda = [a, b] \in \bar{\Lambda} \setminus \Lambda$ 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

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

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

$$\begin{aligned} H^\lambda &\leftarrow H(\mathcal{I})^\lambda = \tanh\left(B_H + \sum_{\nu \in \mathcal{N}} V_{HI}^\nu I^{\lambda+\nu}\right) \\ O^\lambda &\leftarrow O(\mathcal{I}, \mathcal{H})^\lambda = \phi\left(B_O + \sum_{\nu \in \mathcal{N}} V_{OI}^\nu I^{\lambda+\nu} + V_{OH}^\nu H^{\lambda+\nu}\right) \end{aligned}$$

where ϕ 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 V_{HI} , hidden-to-output connections V_{OH} and input-to-output “shortcut” connections V_{OI} . B_H and B_O represent the “bias” at the hidden and output units. We assume that for the edge cells ($\lambda \in \bar{\Lambda} \setminus \Lambda$), the hidden units H^λ remain identically zero, while the inputs I^λ 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:

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

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 D_8 has five irreducible representations, which we will label as *Trivial* (T), *Symmetrical* (S), *Diagonal* (D), *Chiral* (C) and *Faithful* (F). All of them are 1-dimensional except the Faithful representation which is 2-dimensional. We consider, then, five types of hidden node, each with its own group action determined by the following equations:

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

In general, the hidden unit activations for a single cell will consist of some number of each type of hidden node:

$$H = T^{i_T} \times S^{i_S} \times D^{i_D} \times C^{i_C} \times (F_1 \times F_2)^{i_F}$$

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

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

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

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

This invariance imposes certain constraints on the weights of the network, which are outlined in the Appendix. For the experiments reported here,

$i_T = 4$, $i_S = 2$, $i_D = 2$, $i_C = 0$, and $i_F = 2$, making a total

of 12 hidden nodes, 2310 connections per cell and 714 free parameters in the overall system.

Internal Symmetry Networks can also be made recurrent, by additionally connecting each cell to itself and its four immediate neighbors (i.e. excluding diagonals) with recurrent hidden-to-hidden connections. Group representation theory can be used to derive the appropriate constraints for these recurrent connections. Although we have made some preliminary experiments with recurrent networks, the current paper will focus only on feed-forward networks.

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. For example, networks trained for image processing tasks would naturally use one real-valued input per cell for black-and-white images and three real-valued inputs per cell for color images.

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 Section 4.2).

A. Output Encoding

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 R_+^1, R_\circ^1 and R_\bullet^1 . In theory, these three values could all be predicted with one network output (since only one of them is applicable in any given situation). However, we choose instead to use three separate outputs Z_+, Z_\circ and Z_\bullet , in order to allow the network more flexibility in computing these disparate values.

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	R_+^s	R_\circ^s	R_\bullet^s
white	+ (empty)	1	.	$1+c$
white	o (filled)	s	s	$s+c$
atari	+ (empty)	0	$-c$	c
black	• (filled)	$-s$	$-(s+c)$	$-s$
black	+ (empty)	-1	$-(1+c)$.

TABLE II

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

		R_+^1	R_\circ^1	R_\bullet^1	R_+^0	R_\circ^0	R_\bullet^0
white	+	1	.	$1+c$	1	.	$1+c$
white	o	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	$-(1+c)$.

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 capture.

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_\circ^1 and R_\bullet^1 (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_\circ^0 and R_\bullet^0 , we add four additional outputs $A_+^0, A_\circ^0, A_\bullet^0$ and A_\bullet^0 , and employ the transformations shown in the right column of Table III.

TABLE III

RELATIONSHIP BETWEEN REWARDS AND NETWORK OUTPUTS

$$\begin{aligned}
 R_+^1 &= 2Z_+ - 1 & R_+^0 &= A_+^0 - A_\bullet^0 \\
 R_\circ^1 &= (2+c)Z_\circ - (1+c) & R_\circ^0 &= c(Z_\circ - 1) - A_\bullet^0 \\
 R_\bullet^1 &= (2+c)Z_\bullet - 1 & R_\bullet^0 &= cZ_\bullet + A_\bullet^0 \\
 \\
 Z_+ &= (1 + R_+^1) / 2 & A_+^0 &= \max(R_+^0, 0) \\
 & & A_\bullet^0 &= \max(-R_+^0, 0) \\
 Z_\circ &= (1+c + R_\circ^1) / (2+c) & A_\circ^0 &= c(Z_\circ - 1) - R_\circ^0 \\
 Z_\bullet &= (1 + R_\bullet^1) / (2+c) & A_\bullet^0 &= -cZ_\bullet + R_\bullet^0
 \end{aligned}$$

TABLE IV
TARGET VALUES FOR THE SEVEN NETWORK OUTPUTS

		Z_+	Z_o	Z_\bullet	A_+°	A_+^\bullet	A_o	A_\bullet
White	+	1	.	1	1	0	.	1
White	o	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	.

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_o	white avoiding capture
Z_\bullet	white effecting capture
A_+°	white making an eye
A_+^\bullet	black making an eye
A_o	white stone captured, leading to black eye
A_\bullet	black stone captured, leading to white eye

B. 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 categories, 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, let ΣZ be the sum of Z_+ (resp. $(1-Z_+)$) and let ΣA be the sum of A_+° (resp. A_+^\bullet) for all liberties of that group (as computed at the previous time step). The group can then be classified into one of 6 classes, as follows:

- C_1 , if $\Sigma A < 0.75$, $\Sigma Z < 0.75$,
- C_2 , if $\Sigma A < 0.75$, $0.75 \leq \Sigma Z < 1.5$,
- C_3 , if $\Sigma A < 0.75$, $1.5 \leq \Sigma Z$,
- C_4 , if $0.75 \leq \Sigma A < 1.5$, $\Sigma Z < 1.5$,
- C_5 , if $0.75 \leq \Sigma A < 1.5$, $1.5 \leq \Sigma Z$,
- C_6 , if $1.5 \leq \Sigma A$.

Roughly speaking, ΣA estimates the number of eyes that are likely to be made from current liberties of the group, while $(\Sigma A - \Sigma Z)$ 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:

- C_1 : no eyes and no openings
- C_2 : no eyes, and only one opening
- C_3 : no eyes, but at least two openings
- C_4 : one eye, but no opening
- C_5 : one eye, plus at least one opening
- C_6 : at least two eyes

Although the network itself is feed-forward, this 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 C_1 to C_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.

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.

V. TRAINING, RESULTS AND DISCUSSION

Our network was trained by self-play and temporal difference learning [8], [9], [5] in the form of TD(λ) with $\lambda = 0.9$.

Each output was trained using cross entropy minimization, with a learning rate of 0.000005. Weight decay of 0.99999 was applied when the weights were updated at the end of each game. 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 non-trivial 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 β 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 9×9 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 Figure 1 where we see a noisy but generally upward trend in performance.

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

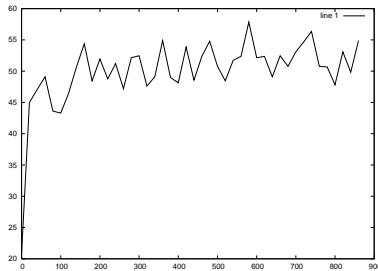


Fig. 1. Percentage of wins in round robin tournament, for networks from 0 to 860K

We extracted the best network (at epoch 580K) and played several games against it on boards of size 9×9 and 19×19 . Generally, the network's evaluation and choice of moves seem quite reasonable, and it can be observed to perform captures, threats, blocking moves, etc. We also let it play for two days on CGOS, where it achieved a rating of 500 on the 9×9 server and 1000 on the 19×19 server. This performance is not strong in absolute terms, but is quite respectable

considering that the network was basing its moves only on immediate evaluation, with no lookahead.

We have recently parallelized our neural network code on an NVIDIA GeForce 8800 graphics card, thus increasing the speed of evaluation to 7000 positions per second for the 19×19 board size. Based on this speedup, we hope to implement a heuristic alpha-beta search to depth 5, using the network itself – or a similar auxiliary network – for heuristic pruning.

The fluctuation in network performance (Figure 1) is reminiscent of what has previously been observed for recurrent neural networks trained to predict context-free or context-sensitive languages [10]. In those studies, it was found that the learning could be made more stable by the use of Evolutionary Computation. The application of such methods in the current context, as well as the training of recurrent networks, is the subject of ongoing work.

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).

$$\begin{aligned}
 V_{OH}^{\nu} &= \left[V_{OT}^{\nu} \quad V_{OS}^{\nu} \quad V_{OD}^{\nu} \quad V_{OC}^{\nu} \quad V_{OF_1}^{\nu} \quad V_{OF_2}^{\nu} \right] \\
 V_{HI}^{\nu} &= \left[V_{TI}^{\nu} \quad V_{SI}^{\nu} \quad V_{DI}^{\nu} \quad V_{CI}^{\nu} \quad V_{F_1I}^{\nu} \quad V_{F_2I}^{\nu} \right]^T \\
 V_{OI}^E &= V_{OI}^N = V_{OI}^W = V_{OI}^S, \quad V_{OI}^{NE} = V_{OI}^{NW} = V_{OI}^{SW} = V_{OI}^{SE} \\
 V_{OT}^E &= V_{OT}^N = V_{OT}^W = V_{OT}^S, \quad V_{OT}^{NE} = V_{OT}^{NW} = V_{OT}^{SW} = V_{OT}^{SE} \\
 V_{TI}^E &= V_{TI}^N = V_{TI}^W = V_{TI}^S, \quad V_{TI}^{NE} = V_{TI}^{NW} = V_{TI}^{SW} = V_{TI}^{SE} \\
 V_{OF_1}^O &= V_{OF_2}^O = V_{F_1I}^O = V_{F_2I}^O = 0 \\
 V_{OF_1}^E &= V_{OF_2}^N = -V_{OF_1}^W = -V_{OF_2}^S = V_{F_1I}^E = V_{F_2I}^N = -V_{F_1I}^W = -V_{F_2I}^S \\
 V_{OF_2}^E &= V_{OF_1}^N = V_{OF_2}^W = V_{OF_1}^S = V_{F_2I}^E = V_{F_1I}^N = V_{F_2I}^W = V_{F_1I}^S = 0 \\
 V_{OF_1}^{NE} &= -V_{OF_1}^{NW} = -V_{OF_1}^{SW} = V_{OF_1}^{SE} = V_{OF_2}^{NE} = V_{OF_2}^{NW} = -V_{OF_2}^{SW} = -V_{OF_2}^{SE} \\
 V_{F_1I}^{NE} &= -V_{F_1I}^{NW} = -V_{F_1I}^{SW} = V_{F_1I}^{SE} = V_{F_2I}^{NE} = V_{F_2I}^{NW} = -V_{F_2I}^{SW} = -V_{F_2I}^{SE} \\
 V_{OS}^E &= -V_{OS}^N = V_{OS}^W = -V_{OS}^S, \quad V_{OD}^{NE} = -V_{OD}^{NW} = V_{OD}^{SW} = -V_{OD}^{SE} \\
 V_{SI}^E &= -V_{SI}^N = V_{SI}^W = -V_{SI}^S, \quad V_{DI}^{NE} = -V_{DI}^{NW} = V_{DI}^{SW} = -V_{DI}^{SE} \\
 V_{OD}^{\mu} &= V_{DI}^{\mu} = 0, \quad \mu \in \{O, E, N, W, S\} \\
 V_{OS}^{\nu} &= V_{SI}^{\nu} = 0, \quad \nu \in \{O, NE, NW, SW, SE\} \\
 V_{OC}^{\nu} &= V_{CI}^{\nu} = 0, \quad \nu \in \{O, E, N, W, S, NE, NW, SW, SE\}
 \end{aligned}$$

REFERENCES

- [1] 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*.
- [2] S.Gelly & Y.Wang, 2006. Exploration exploitation in Go: UCT for Monte-Carlo Go, *Proceedings of Neural Information Processing Systems Conference*.
- [3] 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
- [4] 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.
- [5] 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.
- [6] 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.
- [7] 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.
- [8] R.Sutton, 1988. Learning to Predict by the Methods of Temporal Differences, *Machine Learning* **3**, 9–44.
- [9] G.Tesauro, 1992. Practical Issues in Temporal Difference Learning, *Machine Learning* **8**, 257–277.
- [10] B.Tonkes, A.Blair & J.Wiles, 1998. Inductive bias in context-free language learning, *Ninth Australian Conference on Neural Networks*