Neural Networks for State Evaluation in General Game Playing

Daniel Michulke and Michael Thielscher

Department of Computer Science
Dresden University of Technology
{daniel.michulke,mit}@inf.tu-dresden.de

Abstract. Unlike traditional game playing, General Game Playing is concerned with agents capable of playing classes of games. Given the rules of an unknown game, the agent is supposed to play well without human intervention. For this purpose, agent systems that use deterministic game tree search need to automatically construct a state value function to guide search. Successful systems of this type use evaluation functions derived solely from the game rules, thus neglecting further improvements by experience. In addition, these functions are fixed in their form and do not necessarily capture the game's real state value function. In this work we present an approach for obtaining evaluation functions on the basis of neural networks that overcomes the aforementioned problems. A network initialization extracted from the game rules ensures reasonable behavior without the need for prior training. Later training, however, can lead to significant improvements in evaluation quality, as our results indicate.

1 Introduction

Developing an agent for a specific game allows to include game-specific knowledge as data structures (e.g. as in [15]) or manually designed features (like in today's chess programs [10]). These specializations, however, restrict the agent in that it cannot adapt to game modifications, let alone play completely different games.

In contrast, General Game Playing (GGP) is concerned with the development of systems that understand the rules of previously unknown games and learn to play these games well without human intervention. As such systems cannot benefit from prior knowledge, they have to be endowed with high-level cognitive abilities such as general strategic thinking and abstract reasoning. This makes GGP a good example of a challenging problem which encompasses a variety of AI research areas, including knowledge representation and reasoning, heuristic search, planning and learning.

To allow for comparison of GGP agent systems, the Game Description Language (GDL) has become standard. It allows to describe arbitrary deterministic n-player games with complete information by giving a formal axiomatization of their rules. Progress can be seen at the GGP competition held annually at the

AAAI Conference [8], where the following two different approaches have been recently established in the development of general game playing agents: *simulation-based* approaches [6, 7] employ probabilistic look-ahead search in Monte-Carlo fashion, while *knowledge-based* systems refine domain knowledge (the game rules) in order to extract an evaluation function (or state value function) for assessing the leaf nodes in a depth-limited search tree [11, 2, 13]. Evaluation functions used in the latter type of systems share, however, two major disadvantages:

– They are fixed in their form and do not necessarily capture the real state value function of the game.
– They are determined solely on the basis of the game rules; later experience is ignored.

While the former problem can be solved by using any general function approximator, the latter one can be addressed by employing learning algorithms. Though neural networks represent a solution to the two issues, their training is known to be both time consuming and not converging in general. However, an initialization as done in [16, 3] allows for a sound state evaluation without prior training. In this paper we present an algorithm that, on the basis of $C\text{-}IL^2P$ [3], transforms the game rules to a set of neural networks that can be employed for state evaluation and need not be trained. Learning abilities, however, are retained, and experiments indicate a significant increase in evaluation quality in this case.

The rest paper is organized as follows. In section 2, we give a brief introduction on the fields encompassed by our approach. This is followed, in section 3, by a presentation of the transformation process of domain knowledge to a state value function. In section 4, we present experimental results. These are discussed in section 5. We conclude in section 6.

2 Background

2.1 Running Example: Pentago

For illustration purposes we use the game Pentago[1] as an example throughout this paper. Pentago is a two-player game played on a 6x6 board. The players take turn in first marking a cell that is blank, followed by rotating (either clockwise or counterclockwise) one of the four 3x3 quadrants of the board. The player wins who first achieves a line (horizontal, vertical, or diagonal) with five of his marks. If no blank cell is left on the board and nobody has won, the game ends in a draw. Figure 1 shows a final board position which is won for the white player. The four quadrants can be distinguished by their background color.

2.2 GDL

The Game Description Language (GDL) [12] has become the standard language for GGP, allowing for the definition of deterministic n-player games with

[1] The complete set of rules for this game, and for all others mentioned in this paper, can be found at `http://www.general-game-playing.de/game_db/doku.php`

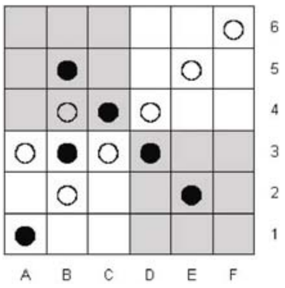

Fig. 1. White wins a match of Pentago

complete information. Examples are popular games like Tic-Tac-Toe, Chess, or Checkers.

GDL allows to give modular, logic-based specifications of games which are interpreted as finite state machines subject to certain restrictions. Each player has a specific role in a game. An actual match corresponds to traversing the finite state machine until a terminal state is reached. A goal function is defined on the set of terminal states that determines the outcome for each player. For the formal specification of the goal, the language uses the keyword `goal` in axioms of the form

```
(<= (goal ?role ?value) goalcondition)
```

where `?role` is the role the player occupies and `?value` gives the goal value (in the interval $[0, 100]$ by convention) for this player if `goalcondition` is fulfilled. `(<= (goal white 100) (line white))` in Pentago thus exemplifies a goal condition where “white” (the player with the role “white”) wins 100 points in case there is a line of white marks in the current state. While the roles in a game are defined as ground facts, the auxiliary relation `line` can be resolved down to facts holding in a state. These facts, called fluents, have the form `(fluent arg_1 ... arg_n)` and can be queried with the keyword `true`. We refer to [12] for the complete specification of GDL.

2.3 Knowledge-Based State Value Functions

Existing knowledge-based GGP agents, such as [11, 2, 13], employ different approaches to construct a state value function that gives a degree of preference of non-terminal states to guide search in an instance of the game (a match). For this purpose they analyze the game description and extract position features to be used in the value function. The automatic construction of an evaluation function takes place in three phases as follows.

In the first phase, game-specific features are identified. To begin with, all of the above mentioned systems attempt to detect static structures (that is, which do not depend on the current state of a game) such as successor relations. E.g., `(succ 2 3)` may define the constant `3` as the successor of `2` in a specific ordering, which then may be used to define an ordering over possible values for a variable which appears somewhere in the game rules. Note that this is a syntactic rather than a lexical property.[2] Successor relations are typically used to represent quantities, counters, or coordinates. On the other hand, constants without any underlying order often represent markers (like in Pentago) or pieces (unique markers as the king in Chess).

These identified static structures can then be used to detect high-level structures. For example, a fluent with arguments from a successor relation may be interpreted as a board, where the other arguments of this fluent either describe the contents of the individual cells (e.g., markers or quantities) or identify a specific instance of the board. The Pentago board, for example, is represented by `(cellholds ?q ?x ?y ?player)`, where `?x` and `?y` appear in a successor relation and where an instance of this fluent holds for every pair (`?x`, `?y`) in every state. Thus, `?x` and `?y` can be identified as coordinates. Because specific instances `?x` and `?y` can occur together with several instances of `?q`, the latter must be part of the coordinates as well. But `?q` does not occur in an underlying order, so that this argument is identified as marking different instances of the board. The remaining argument `?player`, then, must describe the contents of the board cell. Thus, with the coordinates `(?x ?y)`, with the four possible values for `?q`, and with `?player` describing the board content, as a high-level structure of Pentago one can identify four 3x3 boards, one for each quadrant.

In a similar fashion, quantities and other structural elements can be detected. The identified structures can be combined arithmetically in order to construct features out of them. In this way, a knowledge-based GGP system can measure, for example, the distance of a pawn to the promotion rank in chess-like games, or the difference in the number of pieces each player has still on the board in the game of checkers, etc.

Other, game-independent features may be added, such as "mobility," which is a measure of how much control a player has in a given game state by counting the number of available moves. Another feature may be "material," characterizing an advantage in the number of markers or pieces on a board (if applicable).

In the second phase of the automatic construction of an evaluation function, useful features are selected. Common conditions for selecting a feature are stability (e.g., the absence of wild oscillation over its domain) [2], an estimation for the effort it takes to compute a high-level feature, correlation with winning or losing terminal states, and the appearance in the goal formulas of a game [13].

In the third and last phase, the selected features are combined in a single state value function. Either the features are combined with fixed or estimated weights according to some predefined scheme, or the goal function of the game is fuzzified [13]. In the latter case, comparison relations that occur in the goal function of the game can be substituted by the features detected in the first phase. In this way, a boolean winning condition of having more markers than

[2] This can be identified by checking whether the relation in question is a singleton for each constant in any argument position, and whether its graph representation is acyclic.

the opponent, for example, can be mapped onto an arithmetic condition. This allows the player to prefer a lead with a higher margin to one with a smaller margin.

2.4 Neuro-Symbolic Integration

Symbolic knowledge offers the advantage to draw precise conclusions, provided there is a sound and sufficiently large knowledge base. Often, however, the knowledge base is insufficient, incorrect, or simply unavailable. Learning systems, on the other hand, can adapt to various domains, but require prior training and are prone to errors due to indiscriminate or too few learning examples as well as bad initialization. Neuro-Symbolic Integration attempts to unify both views to maintain their advantages while eliminating drawbacks by exploiting synergies. KBANN (Knowledge-Based Artificial Neural Network [16]) was the first approach to convincingly utilize both paradigms. Initialized with an “approximately correct” propositional domain theory, the network was automatically generated and trained and eventually led to a correction of the initial theory. Furthermore, KBANN learned much faster than randomly initialized nets and was able to qualitatively outperform any other algorithm during training as well as afterwards. While KBANN allowed the application of a standard learning algorithm, the algorithm still had some weaknesses that made it more complex than necessary and limited its application to domain theories with only a small number of rules and antecedents per rule.

In [9] it was shown that three-layer recurrent neural networks can represent propositional logic programs without any restrictions regarding the number of rules or antecedents. Learning via backpropagation, however, was not intended. Though it was possible to enhance this result to some extent to first-order logic (see [1] for an overview), it remains an open question whether this purely theoretical algorithm can be put into practice for a problem domain that is as challenging as GGP.

A combined approach was finally presented in [4], allowing an intuitive one-to-one transformation of propositional logic to neural nets while maintaining learning capabilities. We will use a more general version [3] by the same author.

3 From Domain Knowledge to State Values

Using the standard approaches explained above to identify and select features for state evaluation, we pay particular attention to the construction of the value function. As described, existing GGP systems use evaluation functions which are mathematical combinations of individual features. Though weights or similar measures of influence are determined during the analysis of the game rules, the functions are structurally fixed prior to the analysis and only cover a small part of the space of possible state value functions, which may not include useful (let alone optimal) value functions for the game at hand. Moreover, these functions remain static and do not adapt to playing experience. This is partly due to the one-time challenge situation enforced by the GGP championship rules that propose games as unique and non-repeating within the contest [8]. As a consequence, learning over matches is not applicable within this context. Nevertheless, the ability to adapt would allow for constant refinement of the evaluation function if the underlying formulas for the heuristics were designed in a dynamic way.

For this purpose, we will use neural networks as an established method to approximate functions. The idea is that by applying a neuro-symbolic translation algorithm, we can transform a ground-instantiated goal condition encoded in GDL to a bipolar neural net with a sigmoidal activation function that exactly captures its behavior and need not be trained. In this way, we would obtain a baseline state value function that is correct for all terminal states and provides a measure of preference for all non-terminal states by calculating fluent-wise their similarity to goal states.

3.1 Goal Conditions as Propositional Proof Trees

Consider the set of rules of a specific game defined in GDL. The given goal formulas define for every terminal state and for every role `?p` in the game a goal value `?gv`. For any specific rule of the form `(<= (goal ?p ?gv) conditions)`, the given conditions can be evaluated for any current (terminal or non-terminal) state. We can ground-instantiate a goal clause by iteratively substituting every free variable by the ground terms obtained from the signature of the game description. Algorithm *prop*(*argument*) (see Table 1) transforms a ground-instantiated goal condition to a propositional proof tree.

As some of the facts are static (that is, do not depend on the current state such as successor relations), they can instantly be proved to be false or true and subsequently be replaced by a propositional *false* or *true*. Naturally, this may lead to other simplifications, which can easily be implemented and hence shall not be explained further.

By calling *prop* on a ground-instantiated goal condition, we can thus obtain a propositional proof tree whose edges are either identities or negations, whose leafs are queries of facts in the current state, and whose non-leaf nodes are either conjunctions or disjunctions.

Table 1. Transforming a single goal condition to a propositional proof tree

if argument is		then
a fact	`(true fact)`	return $fact$
a negated argument	`(not argument)`	return $\neg prop(argument)$
the head of a clause	`(<= h b_1)` ... `(<= h b_n)`	return $\bigvee_{1 \leq i \leq n} prop(b_i)$
the body of a clause	`(c_1, ..., c_n)`	return $\bigwedge_{1 \leq i \leq n} prop(c_i)$
an inequality	`(distinct x1 x2)`	return $false$ if $x1 = x2$, else $true$

3.2 Propositional Proof Trees as Neural Networks

As has been shown in [16, 9], it is possible to encode propositional logic as a neural network. The two algorithms, however, were developed under different considerations. While [16] was mainly concerned with eliminating disadvantages of neural nets by the use of logic (namely, indifferent initialization, convergence problems, and long training time), [9] translated logic programs to neural nets so as to benefit from advantages of the different representation. An algorithm that got rid of the problems of each of the two approaches is described in [3] and will be used here with some modifications. The original algorithm was used for logic programs where literals in the body of a clause were interpreted as their conjunctions, while clauses with the same head were interpreted as disjunctions of their bodies. Therefore, we use the rules for encoding multiple clauses with the same head for disjunctions, and literals within a clause as rules for conjunctions. Furthermore, the net was built up as a recurrent net to implement the immediate consequent operator of the logic program. As this stepwise fashion is not needed, we build up a simpler feed-forward architecture that is similar to the propositional proof tree.

Consider a layered feed-forward neural net with the following properties: if neuron i is a successor of neuron j (e.g. there is a directed connection from neuron j to neuron i), then the output of neuron i is defined as $o_i = h(\sum_j w_{ij} o_j)$, where $h(x)$ is the bipolar activation function $h(x) = \frac{2}{1+e^{-\beta x}} - 1$. We define the interval $(A_{min}, 1]$ to denote truth and $[-1, A_{max})$ to denote falsity. Without loss of generality we set $A_{min} = -A_{max}$. Then we can translate every node of the propositional proof tree to a neuron by calling the following algorithm on the root of the tree and an "empty" neural net:

```
 1 prop_to_net(node, net) {
 2   if (count_children(node) > 0) {
 3     for each (c in children(node)) {
 4       if (is_positive(c))
 5         add_conn(net, node, c, W);
 6       else
 7         add_conn(net, node, c, -W);
 8       prop_to_net(c, net);
 9     }
10     add_conn(net, node, bias, threshold_weight(node));
11   }
12   else
13     mark_as_input_node(net, node);
14 }
```

Basically, the algorithm does the following: for each non-leaf node (line 2) in the propositional tree an edge is added from the node to each of its children with weight W if the child is positive (that is, not negated; line 5) and $-W$ otherwise (line 7). The algorithm is called recursively for each child of the current node (line 8), and a connection to the bias unit (whose output is always 1) is added so as to function as a threshold. Its weight is determined by the type of the node (conjunction or disjunction) and the number n of its children:

$$threshold_{disj}(n) = \tfrac{(1+A_{min})*(n-1)}{2} * W \tag{1}$$

$$threshold_{conj}(n) = \tfrac{(1+A_{min})*(1-n)}{2} * W \tag{2}$$

Leafs of the tree are marked as input nodes (line 13) and used to evaluate the current state s in the match: if $fact$ holds in s, the node returns 1, else -1.

A_{min} / A_{max} and W are subject to some restrictions to ensure logically sound behavior:

$$A_{min} > \frac{\max(disj, conj) - 1}{\max(disj, conj) + 1} \tag{3}$$

$$W \geq \frac{2}{\beta} * \frac{ln(1 + A_{min}) - ln(1 - A_{min})}{\max(disj, conj)(A_{min} - 1) + A_{min} + 1} \tag{4}$$

Here, $disj$ is the largest number of children that a disjunction node in the tree has, while $conj$ is the largest number of children a conjunction node has. Parameter β controls the steepness of the activation function $h(x)$ and is usually set to 1. Note that the absolute weight w_{ij} of each connection has to be increased by a small random float to avoid symmetry effects when learning.

Encoding the goal conditions as neural network thus maintains the logical sound behavior of an automated proof procedure (for a proof see again [3]) while enabling it to perform "soft computing" and to adapt to experience via a standard backpropagation algorithm.

Practical Aspects of C-IL^2P. Though the algorithm is logically sound, its application in the GGP domain is not as straightforward. Recall that the interval $[-1, A_{max})$ denotes falsity and $(A_{min}, 1]$ truth of a fact. A neuron's output representing a conjunction is thus higher than A_{min} iff the output of all positive preceding neurons is higher than A_{min} and of its negative preceding neurons is lower than A_{max}. Due to the monotonicity of the activation function, we know the output of a neuron to be lower if fewer of its antecedents are fulfilled. A problem, however, arises if no or few antecedents are fulfilled: in these cases the absolute value of the neuronal activation is high and, because of the squashing of the activation function, the output of the neuron stays approximately the same. In other words, for conjunction neurons with few fulfilled antecedents the derivative $h'(x) = 1 - h(x)^2$ is near zero and small changes in the input (e.g. one preceding neuron changing from -1 to $+1$) therefore have only little impact on the output of the neuron. While these differences of the output are small but still recognizable in the case of just one neuron, they eventually become smaller than machine precision after passing through several neurons. As a consequence, the neural network loses its ability to effectively guide search as it erroneously evaluates several states to the same value although there is a difference in their state value.

Regarding the network parameters, there are two reasons for this behavior. The first is the restriction of A_{min} (equation 3): while for a small number of rule antecedents k, A_{min} may be set to a small value as well, higher k fix it near 1 with the result that the intervals for truth and falsity become too small to, e.g., distinguish one neuronal state representing *false* from another. In the same way one can argue that with a high A_{min} the "forbidden zone" $[A_{max}, A_{min}]$ (the interval of output values "between *false* and *true*") becomes very large, resulting in a loss of output space, i.e., output resolution of the neuron.

Another reason is the restriction of the weight W (equation 4): for high values of W the space of activation values is larger, the absolute neuronal input is more likely far away from zero and thus its derivative close to zero. High absolute weights therefore contribute to the problem.

We are, however, not interested in a strict propositional evaluation of a goal condition, but in a degree of preference between states. Specifically, the following two conditions have to be satisfied:

– A terminal state fulfilling the underlying goal function will yield the greatest possible network output.
– Any state will yield a smaller value than another one if it constitutes a worse matching of the corresponding goal function, that is, it is fluent-wise less similar to the state pattern that corresponds to the goal function.

Therefore, we calculate the lower bound $l_A = \frac{\max(disj, conj)-1}{\max(disj, conj)+1}$ for A_{min} and set it such that it enables the smallest possible weight. With $A_{min} \in (l, 1)$, obviously the following condition is satisfied:

$$A_{min} = (l_A - 1) * \alpha + 1 \tag{5}$$
$$\alpha \in (0, 1) \tag{6}$$

We numerically determined W to be minimal for $0.15 \leq \alpha \leq 0.3$ and set it to $\alpha = 0.2$. As this is not enough, we additionally ignore the weight condition (equation 4) and set it to a fixed value $W = 1$.

3.3 From Neural Networks to State Values

To finally obtain a state value, we map the output of the neural networks $o_{player,gv} \in [-1, 1]$ to values in $[0, 1]$ and multiply these with the corresponding goal value. In this way, we obtain the share $v_{player,gv}(s)$ that each pair of network and goal value contributes to the state value. The normalized sum of these shares then forms our value function in the interval $[0, 100]$.

$$v_{player,gv}(s) = \frac{o_{player,gv}(s)+1}{2} * gv^p \tag{7}$$
$$V_{player}(s) = \frac{\sum_{gv \in GV} v_{player,gv}(s)}{\sum_{gv \in GV} gv^p} * 100 \tag{8}$$

With the selection of $p \in [1, \infty)$ we can choose the degree to which higher goal values are more influential on the state value function. For $p = 1$ every goal value is considered linearly and for $p \to \infty$ only the highest goal value is considered, resulting in a riskier behavior.

As we have to ensure that any calculated state value is better than a real loss (where $gv = 0$) and worse than a real win ($gv = 100$), the final state value is set to $V(s)' = V(s) * 0.98 + 1$.

3.4 Increasing Flexibility

The resulting set of neural networks can be enhanced in several ways. Any feature obtained by domain analysis or designed manually can be introduced in the networks by adding an input node for it, normalizing the output of the node to $[-1, 1]$, and connecting it to the output nodes of all networks for a player. In much the same way, spatial or quantitative comparisons in the goal conditions can be substituted by a more detailed feature to increase expressiveness.

3.5 Learning

As result of the transformation we get a set of neural networks to which back-propagation learning can be applied. To train these neural networks, we use terminal states s_{term} from past matches which can be obtained by self-play or are a by-product of probabilistic look-ahead searches as applied in Monte-Carlo search [6, 7]. The goal value of the terminal state indicates the corresponding network which then can be trained by presenting value 1 as signal for the output neuron. All other networks are trained with -1.

We can further increase the amount of available examples by applying the TD(λ) algorithm as used, e.g., in TD-Gammon [15]. By assuming the predecessors of a terminal state to likely leading to the goal value of the terminal state, we can use these predecessors as weaker but still valid training examples. We therefore discount the training signal relative to the distance to the terminal state in order to weaken the impact of the predecessor state on the neural network.

The training signal $t(s)$ for the k-th predecessor of the terminal state s_{term-k} is calculated thus:

$$t(s_{term-k}) = t(s_{term}) * \lambda^k \tag{9}$$

The parameter $\lambda \in [0, 1]$ controls the speed of decay of the training signal. The network corresponding to the terminal state is thus trained with the signals $1, \lambda, \lambda^2, \ldots$ for the terminal state, its predecessor, the predecessor of the predecessor, and so on.

We further enhance TD(λ) to include domain knowledge by checking the predecessors of the terminal state subject to learning whether they inevitably lead to a terminal state with the same goal value. This is done by employing a full-depth game tree search and, in case the search yields a positive answer, substituting a potentially flawed training signal by a provably correct and more expressive one. Note that the required tree search was already performed during the match and can be reused here.

Assuming s_{term} leads to a goal value $gv(s_{term})$ we can thus train the corresponding neural network with the following signal:

$$t(s_{term-k}) = \left\{ \begin{array}{l} 1: \quad minimax(s_{term-k}) = gv(s_{term}) \\ \lambda^k : else \end{array} \right\} \tag{10}$$

Altogether, the network training over several states of one match has two major benefits:

- The network prefers stable features over wildly oscillating ones. The importance of this concept has been discussed in [2] and is implicitly included here.
- The learned patterns match states preceding the terminal states. Along with feedforward search during matches the algorithm thus exhibits a behavior comparable to bidirectional search.

3.6 Transformation of Pentago Rules

For better understanding we will explain the approach in the domain of the game Pentago by showing the result of the transformation for the winning goal condition for the player with the role "white". The goal condition in this game is given as `(<= (goal white 100) (line white))`. A white line, according to the rules, can be a white row, column, or diagonal. As the three cases are analogous, we will just concentrate on white winning with a row. The definition of auxiliary predicate `(row ?player)`, after substituting `?player` by `white`, is:

```
(<=  (row white)
     (role white)
     (true (cellholds ?q1 ?x1 ?y1 white))
     (globalindex ?q1 ?x1 ?y1 ?x1g ?yg)
     (succ ?x1g ?x2g)
     ...
     (succ ?x4g ?x5g)
     ...
     (true (cellholds ?q5 ?x5 ?y5 white))
     (globalindex ?q5 ?x5 ?y5 ?x5g ?yg))
```

Hence, a white row is implied by a conjunction of the following facts. The expression `cellholds` is a fluent which describes the contents of individual cells in a game state. Relation `globalindex` is essentially a function which translates the local quadrant coordinates `?q1`, `?x1`, and `?y1` to the global coordinates `?x1g` and `?yg`. Predicate `succ` describes a successor relation that connects five instances of `cellholds` such that the global x-coordinate of the second cell is a successor of the global x-coordinate of the first cell, and so on. All `cellholds` fluents need to refer to the same y-coordinate `?yg`. The head of the above clause, `row(white)`, is implied if there exists an instantiation for the 21 free variables such that each of the atoms in the body holds.

To translate this goal condition to a neural network, we have to ground-instantiate the rule. A naive ground instantiation would yield at least 3^{21} instances, as each domain of the variables has a size greater or equal to 3. While instantiations of this size cannot be handled efficiently, the inherent structure of the rule can be easily exploited. Because successor relations are functional, each variable `?xg2, ..., ?xg5` is in fact a function of `?xg1`. The domain of `?xg1` shrinks accordingly, indicating that the starting point of a row must have x-coordinate 1 or 2; if it started at 3, the penultimate cell in the row would have no successor.

The `globalindex` relation is functional and injective as well, allowing to bring down the number of different ground instances of the rule to 12, where all variables are functions of the global coordinates A1, A2, B1, ..., F2. With these twelve possible rows on the board we end up with a proof tree consisting of a disjunction of the rows with each of them being a conjunction of five adjacent cells. The `role` relation can be statically evaluated to *true* and thus immediately omitted in the conjunctions.

The resulting net consists of an input layer with 36 nodes of the form `(true (cellholds ?q ?x ?y white))`, a first hidden layer representing the 12 instances of the `row` rule as a conjunction of five adjacent cells, and a second hidden layer representing the generalized `row` rule as a disjunction of the twelve ground-instantiated ones. A similar structure is added for the possibility of a vertical or a diagonal white line, resulting in a total of three neurons in the third hidden layer. The output node would then again be a disjunction of these three cases.

4 Experiments

For testing purposes we implemented a standard General Game Player according to the approach described above. To evaluate its quality and efficiency, we let it play against "Fluxplayer" [13], the best non-simulation based system, ranked at least third in the three recent AAAI GGP World Championships 2006-2008. We refer to the agents with "Neuro" and "Fluxplayer" respectively throughout the experiments.

4.1 Experimental Setup

The tests were run on a dual-processor system at 3.16 GHz with 768 MB RAM available for each agent. The roles were switched after each match, thus forming pairs of matches in which each agent had the first move once. Both agents used α-β-search. For learning, we used all "solved" states that could be proven to lead to a terminal state with the same goal value. If the number of these states was less than 10% of the states occurring in the match, "unsolved" states were used in addition. In this way, at least 10% of the match states were used for training. To distinguish solved from unsolved states, we reused the values calculated in the match. Unsolved states were discounted by the factor of $\lambda = 0.8$ and the learning rate was set to 0.001.

Table 2. Average Goal Value achieved

		Pentago		3D-Tic-Tac-Toe	
	#Matches	Fluxplayer	Neuro	Fluxplayer	Neuro
Init 1-ply	300	57.33	42.67	50	50
Learning 1-ply	5x500	25.72	74.28	50	50
Init Real-Time	300	48.36	51.64	38.67	61.33
Learning Real-Time	5x500	45.46	54.54	45.5	54.5

The tests were run in two different games: Pentago and a 3D version of Tic-Tac-Toe where the first player to achieve a line of four marks in a 4x4x4 cube wins. The results can be seen in Table 2. For each of the two games we examined the two dimensions *Init vs Learning* and *1-ply vs Real-Time*:

Init 1-ply. The quality of the initialized evaluation function was determined in 300 matches with a search depth of 1 and newly initialized networks in each match.

Learning 1-ply. The improvement of evaluation accuracy through learning was determined by running 500 consecutive matches, starting with a newly initialized network and searching with depth 1. The test was run 5 times to minimize possible random effects.

Init Real-Time. The quality of the initialized evaluation function in a real-time scenario was determined in 300 matches with newly initialized networks in each match. Each system had 100 seconds before a match started and 5 seconds for each move.

Learning Real-Time. The improvement of playing strength through learning in a real-time scenario was determined by running 500 consecutive matches starting with a newly initialized network. Each system had 100 seconds before a match started and 5 seconds for each move. The test was run 5 times to minimize possible random effects.

4.2 Results

As can be seen, the 1-ply performance differs in the two games. In Pentago, Neuro wins 43% in the initial scenario, but reaches 74% when learning, showing that the initialized networks are near a local optimum.

In 3D-Tic-Tac-Toe, however, the player who has the first move has such a big advantage that it can enforce a win, resulting in a 50% win rate each. As no examples for how to win as non-starting player are available, learning has no impact on the win rate.

The real-time performance of Neuro in both games is at least equal to that of Fluxplayer. In Pentago, the initial real-time performance lies at roughly 52%. This can be partly attributed to a state evaluation rate approximately twice as high as that of Fluxplayer. The result is improved by another 3% with learning.

In 3D-Tic-Tac-Toe, the initial real-time performance with a 61% win rate is significantly higher than that of Fluxplayer. For this game, Neuro examines

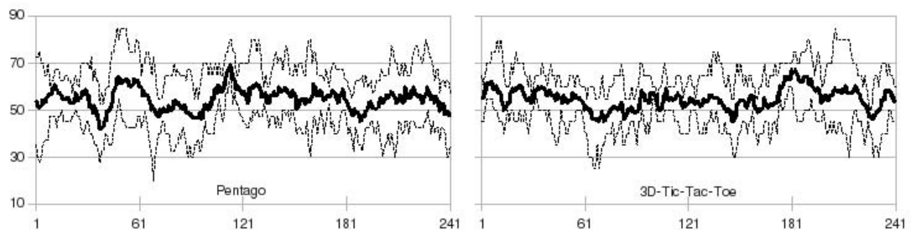

Fig. 2. Evolution of Win Rate in the Learning Real-Time Scenario

about 30% less states than Fluxplayer. The results show that learning decreases the win rate, which might indicate overfitting.

In general, it can be seen that learning has a positive effect if done with caution. Figure 2 depicts the evolution of the win rate in each game by plotting the moving maximum, average and minimum of the recent 10 pairs of matches. Obviously, learning did not converge. However, Neuro achieved in each test run in both games at least a 70% win rate over at least 20 matches at some point throughout the experiments, indicating its potential.

5 Discussion

As can be seen from our initial experiments, the neural approach to General Game Playing can prove advantageous when compared to the currently best non-simulation based system. The relative evaluation quality and, in particular, the real-time performance both depend on the type of the game. Given that the system works in principle while still being in an early development stage regarding possible optimizations, it is likely to outperform the current version of Fluxplayer after thorough parameter tuning and full exploitation of its learning and extension capabilities.

We have seen that learning leads to a significant boost in evaluation quality in the 1-ply scenario in Pentago. On the other hand, it only had a small effect in the real-time scenarios, indicating an overfitting of the network. The drop in win-rate in Pentago when switching from "learning 1-ply" to "learning real-time" as well as the lack of convergence of the win rates support this explanation.

While the initial results look promising and should be put on a broader empirical basis, there are some further theoretical problems related to the ideas presented in this paper. The major disadvantage of the approach is that it requires the use of ground-instantiated fluents. Depending on the complexity of the game, the number of ground instances is subject to combinatorial explosion and can thus easily become too large to be handled by a neural network in an efficient manner. Although there are possibilities to encode first-order logic in neural networks to some degree, current results are not convincing, as they are bound by machine precision for real numbers [14] or have not yet been applied to non-toy examples.

For this reason, we employ several techniques to limit or work around the problem for the time being.

– By exploiting domain constraints, we limit the number or the domain of free variables as demonstrated in the Pentago example.
– Clauses like `(<= c (a ?x) (b ?x))`, where variable bindings hold over more than just one antecedent, or clauses with too many possible ground instances, are redirected to the automated theorem proving system. In this case, we do not transform the clause, but add it as an input node.

Other problems are directly related to learning and can be seen as a consequence of the induction principle. For example, we have to assume the training states to be distributed among the game tree and opponents to play near optimal to avoid overfitting or biasing of the networks.

6 Conclusions

We have presented a method that overcomes the major restrictions of today's knowledge-based GGP systems. It allows to derive a state value function from the goal definition for a game, provided it can be ground-instantiated with reasonable effort. The goal conditions are used to initialize neural networks such that they need not be trained. The evaluation function can then benefit from training with states extracted from past matches, as our experimental results have shown.

6.1 Future Work

The overall evaluation quality is primarily based on the initial evaluation quality and its increase through training, making both the main areas of possible improvement. As none of the parameters used in the experiments were optimized, an analytical parameter determination of the $C\text{-}IL^2P$ parameters (A_{min}, W) and an autonomous adaptation of the learning parameters (learning rate, discount factor, ...) to specific games could further improve evaluation quality.

Furthermore, we intend to exploit the newly gained flexibility by adding new connections or hidden layer nodes to allow for the emergence of new features or by integrating features generated by other means. In fact, the connection weights of the networks offer a utility feedback for those features that could give rise to feature generation algorithms like in [5].

For the future, we intend to conduct further experiments in other games, address the above issues, and implement a complete GGP system on the basis of the approach.

References

[1] Bader, S., Hitzler, P.: Dimensions of neural-symbolic integration — a structured survey. In: We Will Show Them: Essays in Honour of Dov Gabbay. Kings College Publications, pp. 167–194 (2005)

[2] Clune, J.: Heuristic evaluation functions for general game playing. In: Proceedings of the AAAI National Conference on Artificial Intelligence, Vancouver, pp. 1134–1139. AAAI Press, Menlo Park (2007)
[3] d'Avila Garcez, A., Broda, K., Gabbay, D.: Neural Symbolic Learning Systems. Springer, Heidelberg (2002)
[4] d'Avila Garcez, A., Zaverucha, G., de Carvalho, L.: Logical inference and inductive learning in artificial neural networks. In: Proceedings of the ECAI Workshop on Neural Networks and Structured Knowledge (1996)
[5] Fawcett, T.: Feature Discovery for Problem Solving Systems. PhD thesis, University of Massachusetts, Amherst (1993)
[6] Finnsson, H.: Cadia-player: A general game playing agent. Master's thesis, School of Computer Science, Reykjavík University (2007)
[7] Finnsson, H., Björnsson, Y.: Simulation-based approach to general game playing. In: Proceedings of the AAAI National Conference on Artificial Intelligence, Chicago, pp. 259–264. AAAI Press, Menlo Park (2008)
[8] Genesereth, M., Love, N., Pell, B.: General game playing: Overview of the AAAI competition. AI Magazine 26, 62–72 (2005)
[9] Hölldobler, S., Kalinke, Y.: Towards a massively parallel computational model for logic programming. In: Proceedings of the ECAI Workshop on Combining Symbolic and Connectionist Processing, pp. 68–77 (1994)
[10] Hsu, F.: Behind Deep Blue: Building the Computer that Defeated the World Chess Champion. Princeton University Press, Princeton (2002)
[11] Kuhlmann, G., Dresner, K., Stone, P.: Automatic Heuristic Construction in a Complete General Game Player. In: Proceedings of the AAAI National Conference on Artificial Intelligence, Boston, pp. 1457–1462. AAAI Press, Menlo Park (2008)
[12] Love, N., Hinrichs, T., Haley, D., Schkufza, E., Genesereth, M.: General Game Playing: Game Description Language Specification. Technical Report LG–2006–01, Stanford Logic Group, Computer Science Department, Stanford University, 353 Serra Mall, Stanford, CA 94305 (2006), `http://games.stanford.edu`
[13] Schiffel, S., Thielscher, M.: Fluxplayer: A successful general game player. In: Proceedings of the AAAI National Conference on Artificial Intelligence, Vancouver, pp. 1191–1196. AAAI Press, Menlo Park (2007)
[14] Hitzler, P., Bader, S., Witzel, A.: Towards a massively parallel computational model for logic programming. In: Proceedings of the IJCAI Workshop on Neural-Symbolic Learning and Reasoning (2005)
[15] Tesauro, G.: Temporal difference learning and TD-gammon. Communications of the ACM 38, 58–68 (1995)
[16] Towell, G., Shavlik, J., Noordenier, M.: Refinement of approximate domain theories by knowledge based neural network. In: Proceedings of the AAAI National Conference on Artificial Intelligence, pp. 861–866 (1990)