Focused Depth-first Proof Number Search using Convolutional Neural Networks for the Game of Hex

Chao Gao, Martin Müller, Ryan Hayward
{cgao3, mmueller, hayward}@ualberta.ca
University of Alberta

Abstract

Proof Number search (PNS) is an effective algorithm for searching theoretical values on games with non-uniform branching factors. Focused depth-first proof number search (FDFPN) with dynamic widening was proposed for Hex where the branching factor is nearly uniform. However, FDFPN is fragile to its heuristic move ordering function. The recent advances of Convolutional Neural Networks (CNNs) have led to considerable progress in game playing. We investigate how to incorporate the strength of CNNs into solving, with application to the game of Hex. We describe FDFPN-CNN, a new focused DFPN search that uses convolutional neural networks. FDFPN-CNN integrates two CNNs trained from games played by expert players. The value approximation CNN provides reliable information for defining the widening size by estimating the value of the node to expand, while the policy CNN selects promising children nodes to the search. On 8x8 Hex, experimental results show FDFPN-CNN performs notably better than FDFPN, suggesting a promising direction for better solving Hex positions where learning from strong players is possible.

1 Introduction

Much algorithm development for two-player zero sum perfect information games is directed towards creating strong players; however, solvers are also of interest, not only because they are typically more challenging, but also they produce flawless perfect play. Proof Number Search (PNS) [Allis *et al.*, 1994; Kishimoto *et al.*, 2012] is a *best-first* search paradigm designed for solving AND/OR game trees. Since its invention, many variants have been proposed for finding theoretical values in various games [van den Herik and Winands, 2008]. Depth-first proof number search (DFPN) [Nagai, 2002] is a depth-first reformulated PNS variant that adopts thresholds. It has the same behavior as PNS in AND/OR trees, but has lower memory footprint, at the expense of re-expansion. DFPN has been successfully applied to Checkers [Schaeffer *et al.*, 2007], Tsume-Go [Kishimoto and Müller, 2005; Kishimoto, 2005], Tsume-Shogi [Kishimoto, 2010] and capturing problems in Go [Yoshizoe *et al.*, 2007].

PNS algorithms employ proof and disproof numbers to guide node expansion, which allows the search to effectively exploit narrow and deep branches that seem to be promising. They are particular effective in games where the branching factor is non-uniform [Allis *et al.*, 1994]; e.g., in Tsume-Shogi, DFPN would search more than 900 plies before finding checkmate [Kishimoto, 2010].

Applying PNS to games with near-uniform branching factors without dedicated heuristics is problematic, since the calculation of proof and disproof number always takes branching factor into account. In worst case, PNS would behaves much the same as inefficient breadth-first search if the branching factor is exactly uniform. To address such a problem, in Go, inspired by the widening technique in *threat* search [Cazenave, 2004], Yoshizoe [Yoshizoe, 2008] proposed a dynamic widening method for DFPN to solve the capturing problems in Go, which works by sorting the nodes by proof or disproof numbers and then only considering a subset of children nodes. Their new algorithm performs much faster than normal DFPN, but is four times slower than aggressive forward pruning. Still, it has the advantage that correctness is guaranteed, as results obtained by forward pruning were occasionally wrong [Yoshizoe, 2008]. In Hex, FDFPN was proposed by Henderson [Henderson, 2010]. FDFPN shares similar dynamic widening idea, but is with a stronger sorting provided by an augmented circuit resistance based evaluation function [Anshelevich, 2002; Henderson, 2010]. Along with heavy knowledge computation, FDFPN is the best serial algorithm for solving Hex.

However, the FDFPN described by Henderson still has deficiencies. First, the resistance-based evaluation function lacks consistent accuracy; i.e., it tends to prefer fillin, dominated cells [Henderson, 2010]. Such a problem is alleviated but not eliminated by the inferior cells engine and connections strategy computation. Second, the definition of widening size is not well-informed: it neglects the inherent differences between expanding nodes other than their original branching factors.

In this paper, we try to address the above deficiencies using convolutional neural networks. Inspired by the recent progress in computer Go [Maddison *et al.*, 2014; Clark and Storkey, 2014; Tian and Zhu, 2015; Silver *et al.*, 2016], we train a policy neural network for prioritizing move selection and a value neural network for deciding widening size, then propose to integrate those two CNNs into focused proof number search, thereby realizing a more robust focused DFPN search. Experiment on 8x8 Hex shows that the resulting new algorithm performs remarkably better than the original FDFPN, solving all 32 openings with less than 46.7% node expansion.

The rest of the paper is organized as follows: Section 2 surveys related work. Section 3 describes our method. Section 4 presents detailed experiments, and Section 5 concludes this paper.

2 Related Work

Searching game-theoretical values by proof and disproof numbers [Allis *et al.*, 1994] was originally derived from conspiracy number search (CNS) [McAllester, 1988]. The difference is that proof number search is specialized for AND/OR trees with binary outcomes, and such a specialization leads to significant memory reduction compared to CNS.

The ultimate pursuit in AND/OR tree search is a *solution tree* that is able to decide the outcome of root node (either true or false) by back propagating the values of the terminal nodes, according to the AND/OR structure of the tree. The tree is usually implicit, hence heuristic information is needed to guide tree growth, in quest of a solution tree with as few node expansion as possible. Proof number search uses proof and disproof numbers to guide node expansion, it defines proof and disproof numbers as a measure of quality of this node assuming it appears in the solution tree. The root node is an OR node that represents the game state to solve, any node with *opponent* to play is AND node. Terminal *proven* (win for the first player) node is with proof number 0 and disproof number ∞, while *disproven* node is with proof number ∞ and disproof number 0. If no heuristic initialization is used, leaf nodes are initialized with proof and disproof number both as 1. The proof and disproof number for any given non-terminal and non-leaf node n are calculated from bottom-up:

- If n is OR node, its proof number is the minimum of its children's proof number; its disproof number is the sum of its children's disproof number,
- If n is AND node, its proof number is the sum of its children's proof number; disproof number is the minimum of its children's disproof number.

Therefore, in AND/OR trees, proof (disproof) number of node n can be interpreted as the minimum number of leaf nodes that have to be proven (disproven) in order to prove (disprove) node n. PNS works in an iterative fashion, each iteration, starting from the *root* node, PNS consecutively selects the child node with smallest proof number at OR node, and smallest disproof number at AND node, until a leaf node is selected. Such a leaf node is usually named as Most Proving Node (MPN) and will be expanded subsequently, then the values in the influenced branch will be updated before conducting the next iteration. PNS stops until a solution tree is found or run out of computer memory. Many variants of PNS can be found in the literature, see [Kishimoto *et al.*, 2012] for a survey.

PNS suffers from problems like hunger for memory, the state space in many games are not trees but graphs [Kishimoto *et al.*, 2012]. DFPN [Nagai, 2002] adopts two *thresholds* to avoid unnecessary traverse in the tree, whenever moving to a child node, it passes the *thresholds* and conduct Multiple Iterative Deepening (MID) until thresholds are violated. As a reformulation of PNS, DFPN is usually more applicable than PNS, but far from ideal. Indeed, various techniques have been introduced when applying DFPN in specific domains; e.g., Yoshizoe et al [Yoshizoe *et al.*, 2007] introduced λ search to DFPN to solve the capturing problems in Go, df-pn(r) [Kishimoto, 2005] was proposed to address the repetitions in Tsume-Go, threshold controlling and source node detection [Kishimoto, 2010] were introduced to DFPN to deal with overestimation, underestimation and repetitions.

However, because proof and disproof number always consider branching factor, which makes DFPN (the same for PNS) behaves much like breadth-first search when the branching factor is nearly uniform, as occurs in Go and Hex. As an attempt, Yoshizoe [Yoshizoe, 2008] proposed a *dynamic widening* technique that only considers top-k or a portion of $1/k$ children nodes during the search. However, the new method was not quite satisfying, presumably due to the selection scheme of "promising children nodes", which was realized by sorting proof or disproof numbers of sibling nodes, is too primitive.

Hex has a large and nearly uniform branching factor similar to same board size Go. There is no draw in Hex, and it has been proven that solving arbitrary Hex states is PSPACE-complete [Reisch, 1981]. Early researches focus on H-Search [Anshelevich, 2002] which constructs *connection strategies* recursively. However, H-Search is incomplete; i.e., there exists positions that winning carries cannot be deduced by practical H-Search, hence tree search is still required. All 8x8 Hex openings were first solved by Henderson et al. [Henderson *et al.*, 2009] after largely extending the knowledge abstraction techniques in [Hayward *et al.*, 2004] and implementing them into a depth-first search. Focused Depth-first Proof Number search (FDFPN) was subsequently developed, which is better than DFPN and at least twice faster than the original depth-first search on larger board like 8x8 [Henderson, 2010].

Recently, deep Convolutional Neural Networks (CNNs) [Krizhevsky *et al.*, 2012] have been successfully applied to computer Go [Maddison *et al.*, 2014; Tian and Zhu, 2015; Silver *et al.*, 2016], which have demonstrated the superiority of using CNNs to design well-informed heuristics in search. Motivated by those previous work, but instead of playing, we study how to harness the strength of CNNs for theoretical solving Hex, a challenging task mainly due to its large and near-uniform branching factor.

3 Focused Depth-first Proof Number Search with CNNs

In this section, we first describe our convolutional neural network models, and then show how to incorporate them to proof number search.

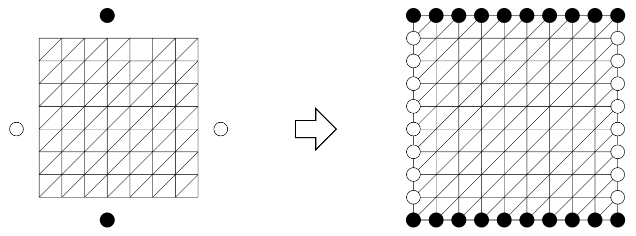

Figure 1: An 8x8 Hex board in Go-style before and after padding.

3.1 Evaluation by Convolutional Neural Networks

Preparation of data

Unlike the situation in 19x19 Go, where tens of thousands of high quality game records are available, no large set of high quality human games is available for 8x8 Hex. However, strong computer players are available. We generated training data for learning by playing matches between two strong computer Hex players under the benzene [1] project: the Monte Carlo Tree Search player Mohex [Huang *et al.*, 2013; Arneson *et al.*, 2010] and an $\alpha\beta$ player Wolve [Henderson *et al.*, 2009] .

To increase the variety of encountered game positions, we varied the time settings per move from 5s to 10s, on a Intel Xeon E5-2650 2.30 GHz CPU machine with 4 gigabytes RAM, iterated over all one and two stone openings, and turned off the endgame solvers in those two players. We collected 62853 games in total, each associated with a game result. We then extract our training and testing data from those games. The first data set $\mathcal{D}_1$ we extracted contains 645249 distinct state-action pairs (s, a), split into $580,000$ as training set and the rest as test set. We prepare dataset $\mathcal{D}_2$ for value regression, this set contains 649373 distinct state-value pairs (s, z). Since the same position s may appear in two or more games but associated with different results, we use the average to label such a state; i.e., let A be the set of games that s have been played, $r(s, g)$ is the playing result (either +1 or -1), $z = \frac{\sum_{g \in A} r(s,g)}{|A|}$. Again, $\mathcal{D}_2$ is split into training set and test set with a portion of about 90% and 10%, respectively. We note that, for better *data efficiency*, instead of selecting only one state-value pair from each game as in [Silver *et al.*, 2016], we collect all the state-value pairs (s, z) played in each game, and rely on the average labelling to reduce correlation.

Input features to CNNs

Like Go, Hex can be viewed as playing on the intersections of a square board, where black tries to connect the *south* and *north* sides, white tries to connect *east* and *west* sides. To represent this goal of the game, we use extra paddings at four boarders with black or white stones, as shown in Figure 1. We adopt one-hot (i.e., binary) encoding to represent the input features, which consist of 5 planes: the first three are respectively black, white occupied stones and empty points, followed by two planes that represent respectively black and white "bridges". Table 1 shows the input features.

[1]https://sourceforge.net/projects/benzene/

Table 1: Input features to CNNs

Features	plane index	description
black stones	0	black stones
white stones	1	white stones
unoccupied	2	empty points
black bridges	3	black bridge endpoints
white bridges	4	white bridge endpoints

Policy neural network

The policy network has 5 hidden layers. The input features have size $10 \times 10 \times 5$ after padding borders. First hidden layer convolves using 48 filters with kernel size 3×3 and stride of 1, then rectified linear unit (ReLU) is applied. The second hidden layer zero pads an image into 10×10 and convolves using same filters of kernel size 3×3 and stride of 1, again with rectified linear unit applied. Hidden layers 3 and 4 repeats the process as layer 2 . Keeping the stride as 1, hidden layer 5 convolves with 1×1 kernel size filters with 64 biases for each position; final layer is a softmax function. The output of policy neural network can be seen as a playing probability for each point in the board.

Similar to previous work on Go [Maddison *et al.*, 2014; Clark and Storkey, 2014; Tian and Zhu, 2015; Silver *et al.*, 2016], we train the policy network to maximize the likelihood of the move a that has been played in state s, $\Delta\sigma \propto \frac{\partial \log p_\sigma(a|s)}{\partial \sigma}$. Instead of vanilla stochastic gradient descent, for faster convergence, we use the adaptive learning rate method Adam [Kingma and Ba, 2014] with default parameters $\textit{learning rate} = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$ to train our model. We train the policy network for 1.2×10^5 steps (13 epochs) with mini-batch size 64, this took about 2.5 hours on an Intel i7-6700 CPU machine with 32GB RAM and NVIDIA GTX 1080 GPU. Top one prediction accuracy on the testing data is 60.4% and about 67% on training data – a results slightly higher than the best accuracy in computer Go [Maddison *et al.*, 2014; Tian and Zhu, 2015]. Note that, as in previous work on Go, "ground-truth" labels in the training data are inherent noisy, because they were produced by imperfect players. Therefore, a higher accuracy does not necessarily imply a better neural network. Figure 2 presents the top-k accuracy varying $k = 1 \dots 10$. We implement the policy network using tensorflow [2].

Value neural network

We train a value regression neural network on the training data split from $\mathcal{D}_2$. As a result of average labelling, early game positions were generally associated with a moderate value; e.g., opening $a4$ is labelled as -0.08 in our dataset. Since for arbitrary position, the optimal value is unknown, we believe that by this average labelling, the risk of training the neural network with strong but wrong signals gets mitigated.

The first 6 layers of our value network are exactly the same as the policy network. After layer 6, the next layer convolves with kernel size 1×1 filters of stride 1, followed by a fully connected layers with 48 units, the last layer is a $\tanh$ function that squashes the output to $[-1, +1]$.

[2]https://www.tensorflow.org/

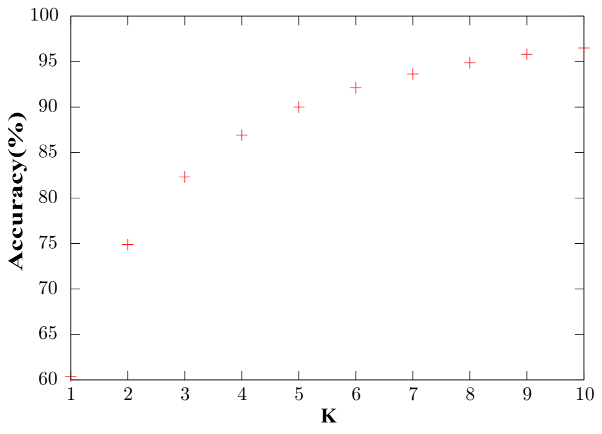

Figure 2: Top k move prediction accuracy on test data.

We train the value network to minimize the Mean Square Error (MSE) between the predicated value $v_\theta(s)$ and corresponding label value z, i.e., $\Delta\theta \propto \frac{\partial v_\theta(s)}{\partial\theta}(z - v_\theta(s))$. With mini-batch of 64, using Adam optimizer [Kingma and Ba, 2014], we train the value network for 1.6×10^5 steps, taking about 3.5 hours. This achieves an MSE of 0.067 on the training dataset, and 0.083 on the test dataset, which indicates the over-fitting is small. Again, the value network is implemented with tensorflow.

3.2 Incorporate CNNs to Proof Number Search

Before describing how to integrate policy and value networks to proof number search, we first review the general idea of focused depth-first proof number search, then explain how to use these two CNNs to redefine FDFPN. Note that when we say a node is *losing (wining)*, it is with regarding to the player to play at this node.

Focused proof number search

For game trees with large branching factor, the idea of a focused search is to narrow the breadth of search by first focusing effort on a proportion of promising moves. In the case of proof number search, reducing the width to a small but fixed value is problematic, since the strongest point of PNS is to exploit the non-uniformity of a game tree. Indeed, FDFPN [Henderson, 2010] controls the number of child nodes expanded for a search node by the following formula:

$$\textit{child limit} = \textit{base} + \lceil \mu \times |\textit{live children}| \rceil \tag{1}$$

Here, widening factor $0 < \mu < 1$. The parameter $\textit{base}$ is usually set as 1; $\textit{live children}$ contains all children nodes that have not been pruned. For arbitrary node in the search tree, once a winning children node is revealed, it will be pruned, and the above formula either maintains the same $\textit{child limit}$ or introduces a new child. If a losing child node is found, clearly that all other children nodes can be abandoned since the current node is thus solved. The intended strength of Equation (1) is that when the move ordering function is good, a node can be solved to be winning by only exploring a few of its children nodes. However, when the move ordering function is poor, perhaps quite a few children nodes have to be revealed as winning before a losing child node can be included to the search.

We summarize that the idea of FDFPN constitutes two crucial components: (a) a manner to define widening size and (b) an external move ordering function for selecting promising moves. In [Henderson, 2010], (a) was done by manually setting an empirical widening factor, and (b) was realized by a resistance-based heuristic evaluation function.

It is uninformative to adopt a static widening factor for all expanding nodes regardless their likeliness of being win or lose in optimal. The resistance-based evaluation function, on the other hand, though had been improved by the aid of the complicated inferior cell and virtual connection engines, is still lacking of consistent accuracy [Henderson, 2010].

Dynamic FDFPN using CNNs

We now describe how to use CNNs to redefine FDFPN. Through the depiction above, we can see that the move ordering function is merely used for selecting promising moves into the search tree; the strict order of the selected moves does not matter much since proof number search ultimately uses proof or disproof numbers for choosing the most proving node. The high move prediction accuracy of our trained policy neural network has shown its capacity in mimicking expert's playing of good quality moves. Therefore, our first modification of FDFPN is to use the trained policy network to replace the resistance-based evaluation function.

How many "promising moves" should be selected is a question. A too large widening factor would lead to decreased performance, while too small would increase the possibility of mis-selection of existing winning moves. Preferably, the widening size should be a function of the quality of the expanding node, which can be measured by the value neural network. Hence, our second modification is, for any expanding state (or tree node) s, its widening size is dynamically decided by the following revised formula:

$$l(s) = \textit{base} + \lceil f(s) \times |\textit{live children}| \rceil, \tag{2}$$

where $f(s)$ is defined as

$$f(s) = \min\{\mu, 1 + v_\theta(s)\} \tag{3}$$

As in Equation (1), μ is a widening parameter. The difference between Equation (1) and Equation (2)–(3) is that: when $v_\theta(s)$ is close to -1, which indicates s is likely to be losing, then the smaller estimation value $v_\theta(s)$ will be used as the widening factor. Therefore, the smaller $v_\theta(s)$ is, the smaller widening size would be. This may seem counter-intuitive, since one may expect to expand all the children nodes, if a state s has an optimal value of -1 — because eventually all s's children must be solved to certificate s is losing. However, a losing node is a wining move for its parent state. As proof number search prefers nodes with small branching factor, giving a losing node a smaller widening size would increase its chance of being selected by its parent node. This is desirable for the root state and every other parent states in the search tree. Another explanation is that if a state is losing, all its children nodes must be solved, it is perhaps better sequentially solving them one by one in a depth-first fashion rather than frequently jumping around.

4 Experiments

4.1 Setup

We built FDFPN-CNN upon benzene, a c++ open source Hex code base for state-of-the-art solving and playing Hex [Arneson *et al.*, 2010; Henderson, 2010; Huang *et al.*, 2013]. Notable features of benzene include (1) An Inferior Cell (IC) engine for knowledge computation, it eliminates many *inferior* cells from consideration; (2) A reimplemented Virtual Connection (VC) engine [Pawlewicz *et al.*, 2015] for winning strategy computation by H-search. We note that now the best Olympiad automated players (e.g., MoHex, Wolve, DeepHex, Ezo) [Hayward *et al.*,] and the best solver (FDFPN) are all built upon benzene.

We export the trained models for solely forward inferencing after training is finished. FDFPN-CNN was modified upon FDFPN, and we compile it with tensorflow libraries using g++ with O3 optimization. Since those two CNNs are independent from the VC and IC engines in benzene, two *asynchronous* threads were used, overlapping with the VC and IC computation. We note that the current *resistance-based* evaluation does not permit similar asynchronous computation, because it requires VC and IC information. Due to asynchronous evaluation, we found the runtime overhead because of evaluating of CNNs becomes small: evaluating CNN takes less than $1\ ms$, but computing VC and IC could be 10 times slower in early positions.

We conduct comparative experiments on tractable but still challenging 8x8 Hex openings. It should be noted that, by now, with prolonged computation time of several months, all 9x9 Hex openings have been solved by turning the FDFPN solver into parallel [Pawlewicz and Hayward, 2013]. We consider only 8x8 Hex in this paper for the convenience of empirical study of new ideas.

All experimental results in this section were obtained on the same Intel i7-6700 CPU machine with 32 gigabytes memory and NVIDIA GTX 1080 GPU, running a 64bit Linux system.

4.2 Experimental comparison of DFPN, FDFPN and FDFPN-CNN

Table 2 compares the performances of FDFPN and FDFPN-CNN on solving all 32 8x8 Hex openings. As a reference, we also list the results of DFPN, which can be seen as setting FDFPN's widening factor as 1.0 and sorting moves randomly. As suggested in [Henderson, 2010], $base$ is set to 1 and widening factor is set to 0.2 for FDFPN, the transposition table size is set to have 2^{21} entries. To see the performance gaining after applying CNNs, the same $\text{factor} = 0.2, base = 1$ is adopted by FDFPN-CNN.

It is apparent from Table 2 that both FDFPN and FDFPN-CNN are better than DFPN, since they could solve every opening much faster than DFPN: the cumulative time of DFPN is respectively 2.6 and 4.1 larger than that of FDFPN and FDFPN-CNN, indicating the focused search is indeed worthwhile.

FDFPN-CNN also performs remarkably better than FDFPN, as it can solves 27 (out of 32) positions using less computation time, the largest improvement was observed on opening $a7$, where FDFPN-CNN reduces the computation time from 5495s to 2614s. In summary, FDFPN-CNN solves all 32 openings with cumulative time 36% less than that of FDFPN. Since expanding a node is typically expensive, another perspective for performance assessment is node expansion, FDFPN and FDFPN-CNN respectively expands 8509140 and 4534084 nodes in total. Thus, cumulatively, the node expansion of FDFPN-CNN is 46.7% less than that of FDFPN.

FDFPN-CNN does not perform better than FDFPN in every cases. In terms of node expansion, on openings $a6, c2, c8$, FDFPN-CNN's results are worse than FDFPN, we suspect that it is because some nodes were poorly evaluated by v_θ and v_σ, thus the search has to focus on solving unnecessary nodes before including winning moves to the search tree.

Table 2: Experimental comparison of DFPN, FDFPN and FDFPN-CNN, better results marked by boldface. All times are rounded into seconds.

posi.	DFPN		FDFPN		FDFPN-CNN	
	#node	time	#node	time	#node	time
a1	47383	240	30462	71	**12063**	**46**
a2	264718	489	104581	161	**61900**	**116**
a3	370973	1350	212140	486	**103940**	**275**
a4	1418942	4482	570207	1167	**217130**	**477**
a5	3929824	11128	1797393	3377	**1226009**	**2856**
a6	525308	1177	**272614**	**473**	295163	474
a7	3230008	10067	2874465	5496	**1058361**	**2615**
a8	1408664	4024	844403	1799	**572892**	**1300**
b1	49607	265	30317	73	**12490**	**43**
b2	204342	421	123728	140	**48074**	**82**
b3	89920	405	39683	115	**19077**	**82**
b4	541376	2003	270571	565	**186693**	**455**
b5	463360	1974	193799	526	**100489**	**368**
b6	563084	1808	497961	931	**223486**	**576**
b7	19182	53	12146	25	**6601**	**16**
b8	54590	201	15106	47	**12150**	**43**
c1	46926	226	28775	58	**10645**	**38**
c2	128112	287	**43899**	**74**	118384	133
c3	125365	521	56597	154	**25819**	**82**
c4	56951	281	18687	54	**11618**	54
c5	54388	271	28816	85	**17600**	**63**
c6	41018	205	30637	88	**10536**	**45**
c7	93673	278	60134	114	**42389**	**90**
c8	40562	183	**14157**	**40**	20327	63
d1	44295	192	14858	35	**10649**	**34**
d2	51451	147	89900	140	**16309**	**37**
d3	62776	269	24926	75	**12177**	**48**
d4	20368	94	13265	43	**8469**	**35**
d5	13821	85	6914	22	**4068**	**20**
d6	63014	345	93313	256	**13876**	**59**
d7	63574	192	79214	137	**40337**	**87**
d8	44634	198	15472	**37**	**14363**	40
SUM	14132209	43861	8509140	16864	**4534084**	**10752**

4.3 Accuracy of the value neural network

When a state is solved, it means a *solution tree* (more precisely, solution graph) has been found and the optimal value of every node in this solution tree is settled. This provides us an opportunity to exactly inspect the real estimation accuracy of our value neural network.

We have seen from Table 2 that opening $a5$ requires the most computation time and most number (more than 1×10^6) of node expansions to find a solution, thus we present in Table 3 the detailed value estimations by v_θ on all nodes in the solution graph to opening $a5$.

Summing the estimations in Table 3, we obtain 188454

Table 3: Value estimations by v_θ of nodes in the solution graph to opening $a5$. Boldface indicates the prediction by v_θ matches the true value.

$v_\theta(s)$	wining	losing	$v_\theta(s)$	wining	losing
$[-1.0, -1.0]$	2	**22**	$(0, 0.1]$	**806**	345
$(-1.0, -0.9]$	10068	**46397**	$(0.1, 0.2]$	**840**	367
$(-0.9, -0.8]$	1779	**2144**	$(0.2, 0.3]$	**911**	332
$(-0.8, -0.7]$	1281	**1236**	$(0.3, 0.4]$	**1097**	341
$(-0.7, -0.6]$	1022	**844**	$(0.4, 0.5]$	**1256**	336
$(-0.6, -0.5]$	924	**667**	$(0.5, 0.6]$	**1525**	383
$(-0.5, -0.4]$	795	**520**	$(0.6, 0.7]$	**1895**	467
$(-0.4, -0.3]$	799	**484**	$(0.7, 0.8]$	**2825**	522
$(-0.3, -0.2]$	778	**428**	$(0.8, 0.9]$	**4987**	759
$(-0.2, -0.1]$	779	**380**	$(0.9, 1.0]$	**91953**	4052
$(-0.1, 0]$	752	**354**	SUM	127074	61380

– it represents the size of the discovered *solution*, in which 127074 are wining nodes, 61380 are losing. Suppose there is a simple classifier that classifies s as wining if $v_\theta(s) > 0$, otherwise as losing, then it is easy to calculate the accuracy of this classifier: $\frac{22+46397+...+354+806+840+...+91953}{188454} = 85.7\%$ – such a result reaffirms the good quality of our value network. Nonetheless, the total node expansion for solving $a5$ is 1226009. This suggests that there is, perhaps, still a great potential for algorithmic improvement.

4.4 Effectiveness of the policy neural network

We further investigate the effectiveness of the policy network by conducting yet another comparison between FDFPN and FDFPN-p_σ, a program that only replaces the move ordering function as policy neural network. Both programs use the same $base = 1$ and widening factors varying from 0.1 to 1.0.

Figure 3 shows the comparison. FDFPN-*resistance* is the original FDFPN that uses resistance-based evaluation function, and FDFPN-p_σ is with the policy network. Apparently, both methods achieve best performance with widening factor of 0.2, while the consumed time of FDFPN-p_σ is consistently less than that of FDFPN-resistance when the widening factor is small (≤ 0.5). FDFPN-p_σ becomes slightly worse than FDFPN-resistance when the widening factor is intermediately large, perhaps because the policy network is poor at ranking weaker moves that are seldom played by expert players. We note that FDFPN-p_σ is 17.3% faster than FDFPN-resistance with widening factor 0.2, while in terms of node expansion, it is 14.3% less than that of FDFPN-resistance. Combing the results in Table 2, we can also notice that after applying value neural network, the node expansion compared to FDFPN-resistance was futher reduced by 32.4%.

This comparison shows that with small widening factors, which are desirable for the idea of focused search, the policy network is indeed more accurate at selecting promising moves than the circuit resistance-based evaluation function. The dynamic widening adjustment provided by the value neural network estimation further improves solving efficiency.

5 Conclusion and future work

We have presented a focused depth-first proof number search using deep convolutional neural networks. Experimental results on 8x8 Hex show the convolutional neural networks can effectively grasp knowledge from data accumulated by expert computer players, then we proposed a redefinition of FDFPN using CNNs, resulting a solver that is able to solve game positions with much less node expansion.

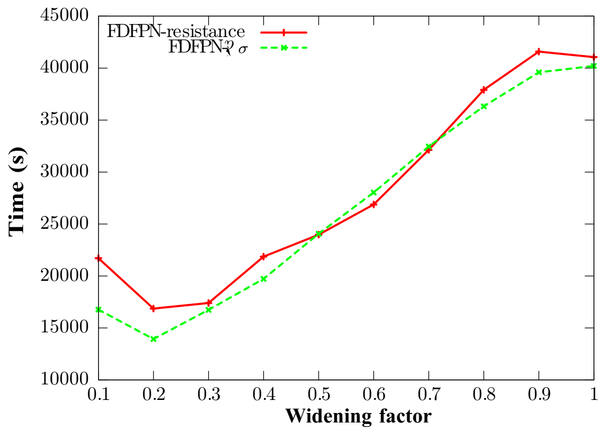

Figure 3: Comparison of *resistance* and p_σ

Nevertheless, it is observed that the search effort is still substantially larger than size of the final solution tree. In the future, we intend to work in the following directions. First, it is necessary to investigate other input features to CNNs, which may further improve the accuracy of the neural networks. Second, our policy network was trained to maximize the likelihood of playing a single expert move, but it is more desirable to adjust the training targets so as to discern frequently played good moves and occasional blunder moves. Linking log-likelihood with task rewards [Norouzi *et al.*, 2016] might be helpful. Third, proof number search has nice theoretical properties on trees, but the state space of Hex is essential a directed acyclic graph (DAG), mis-selection of MPN due to exponential over-counting is an issue [Kishimoto *et al.*, 2012], which should be addressed. Fourth, the focused search is somewhat sensitive to the accuracy of the value neural network, the possibility of better estimating the value of a node by considering also its descendants (i.e., utilize the the hierarchical AND/OR structure) should be investigated. With more accurate estimation, a more aggressive focusing scheme may be possible.

Since in Hex, automatic playing on board sizes up to 11x11 has been very strong, but solving even 9x9 openings still takes very long parallel computation. On the other hand, playing and solving share the same aspiration of exploring promising regions preferably. In the future, we would also like to work on solving larger board sizes where learning from strong players is also possible. It is interesting to adapt our framework for solving other games as well.

Acknowledgments

We thank Broderick Arneson, Jakub Pawlewicz for their feedbacks about benzene. Thank Noah Weninger for helping produce training data.

References

[Allis *et al.*, 1994] L Victor Allis, Maarten van der Meulen, and H Jaap Van Den Herik. Proof-number search. *Artificial Intelligence*, 66(1):91–124, 1994.

[Anshelevich, 2002] Vadim V Anshelevich. A hierarchical approach to computer Hex. *Artificial Intelligence*, 134(1):101–120, 2002.

[Arneson *et al.*, 2010] Broderick Arneson, Ryan B Hayward, and Philip Henderson. Monte carlo tree search in Hex. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):251–258, 2010.

[Cazenave, 2004] Tristan Cazenave. Generalized widening. In *ECAI*, volume 16, page 156, 2004.

[Clark and Storkey, 2014] Christopher Clark and Amos Storkey. Teaching deep convolutional neural networks to play go. *arXiv preprint arXiv:1412.3409*, 2014.

[Hayward *et al.*,] Ryan Hayward, Jakub Pawlewicz, Kei Takada, and Tony van der Valk. Mohex wins 2015 Hex 11x11 and Hex 13x13 tournaments. *ICGA Journal*, (Preprint):1–6.

[Hayward *et al.*, 2004] Ryan Hayward, Yngvi Björnsson, Michael Johanson, Morgan Kan, Nathan Po, and Jack van Rijswijck. Solving 7×7 hex: Virtual connections and game-state reduction. In *Advances in Computer Games*, pages 261–278. Springer, 2004.

[Henderson *et al.*, 2009] Philip Henderson, Broderick Arneson, and Ryan B Hayward. Solving 8×8 Hex. In *Proc. IJCAI*, volume 9, pages 505–510. Citeseer, 2009.

[Henderson, 2010] Philip Thomas Henderson. *Playing and solving the game of Hex*. PhD thesis, University of Alberta, 2010.

[Huang *et al.*, 2013] Shih-Chieh Huang, Broderick Arneson, Ryan B Hayward, Martin Müller, and Jakub Pawlewicz. Mohex 2.0: a pattern-based MCTS hex player. In *International Conference on Computers and Games*, pages 60–71. Springer, 2013.

[Kingma and Ba, 2014] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[Kishimoto and Müller, 2005] Akihiro Kishimoto and Martin Müller. Search versus knowledge for solving life and death problems in Go. In *AAAI*, pages 1374–1379, 2005.

[Kishimoto *et al.*, 2012] Akihiro Kishimoto, Mark HM Winands, Martin Müller, and Jahn-Takeshi Saito. Game-tree search using proof numbers: The first twenty years. *ICGA Journal*, 35(3):131–156, 2012.

[Kishimoto, 2005] Akihiro Kishimoto. *Correct and efficient search algorithms in the presence of repetitions*. PhD thesis, University of Alberta, 2005.

[Kishimoto, 2010] Akihiro Kishimoto. Dealing with infinite loops, underestimation, and overestimation of depth-first proof-number search. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, pages 108–113. AAAI Press, 2010.

[Krizhevsky *et al.*, 2012] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[Maddison *et al.*, 2014] Chris J Maddison, Aja Huang, Ilya Sutskever, and David Silver. Move evaluation in Go using deep convolutional neural networks. *arXiv preprint arXiv:1412.6564*, 2014.

[McAllester, 1988] David Allen McAllester. Conspiracy numbers for min-max search. *Artificial Intelligence*, 35(3):287–310, 1988.

[Nagai, 2002] Ayumu Nagai. *Df-pn algorithm for searching AND/OR trees and its applications*. PhD thesis, PhD thesis, Department of Information Science, University of Tokyo, 2002.

[Norouzi *et al.*, 2016] Mohammad Norouzi, Samy Bengio, Navdeep Jaitly, Mike Schuster, Yonghui Wu, Dale Schuurmans, et al. Reward augmented maximum likelihood for neural structured prediction. In *Advances In Neural Information Processing Systems*, pages 1723–1731, 2016.

[Pawlewicz and Hayward, 2013] Jakub Pawlewicz and Ryan B Hayward. Scalable parallel DFPN search. In *International Conference on Computers and Games*, pages 138–150. Springer, 2013.

[Pawlewicz *et al.*, 2015] Jakub Pawlewicz, Ryan Hayward, Philip Henderson, and Broderick Arneson. Stronger Virtual Connections in Hex. *IEEE Transactions on Computational Intelligence and AI in Games*, 7(2):156–166, 2015.

[Reisch, 1981] Stefan Reisch. Hex ist PSPACE-vollständig. *Acta Informatica*, 15(2):167–191, 1981.

[Schaeffer *et al.*, 2007] Jonathan Schaeffer, Neil Burch, Yngvi Björnsson, Akihiro Kishimoto, Martin Müller, Robert Lake, Paul Lu, and Steve Sutphen. Checkers is solved. *science*, 317(5844):1518–1522, 2007.

[Silver *et al.*, 2016] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[Tian and Zhu, 2015] Yuandong Tian and Yan Zhu. Better computer Go player with neural network and long-term prediction. *arXiv preprint arXiv:1511.06410*, 2015.

[van den Herik and Winands, 2008] H Jaap van den Herik and Mark HM Winands. Proof-number search and its variants. In *Oppositional Concepts in Computational Intelligence*, pages 91–118. Springer, 2008.

[Yoshizoe *et al.*, 2007] Kazuki Yoshizoe, Akihiro Kishimoto, and Martin Müller. Lambda Depth-First Proof Number Search and Its Application to Go. In *IJCAI*, pages 2404–2409, 2007.

[Yoshizoe, 2008] Kazuki Yoshizoe. A new proof-number calculation technique for proof-number search. In *International Conference on Computers and Games*, pages 135–145. Springer, 2008.