

Deep or Wide? Learning Policy and Value Neural Networks for Combinatorial Games

Stefan Edelkamp^(✉)

Faculty of Mathematics and Computer Science,
University of Bremen, Bremen, Germany
edelkamp@tzi.de

Abstract. The success in learning how to play *Go* at a professional level is based on training a deep neural network on a wider selection of human expert games and raises the question on the availability, the limits, and the possibilities of this technique for other combinatorial games, especially when there is a lack of access to a larger body of additional expert knowledge.

As a step towards this direction, we trained a *value network* for Tic-Tac-Toe, providing perfect winning information obtained by *retrograde analysis*. Next, we trained a *policy network* for the SameGame, a challenging combinatorial puzzle. Here, we discuss the interplay of deep learning with *nested rollout policy adaptation* (NRPA), a randomized algorithm for optimizing the outcome of single-player games.

In both cases we observed that ordinary feed-forward neural networks can perform better than convolutional ones both in accuracy and efficiency.

1 Introduction

Deep Learning¹ is an area of AI research, which has been introduced with the objective of moving the field closer to its roots: the creation of human-like intelligence. One of its core data structures is a convolutional neural network (CNN). As in conventional NNs, CNNs are trained with reinforcement learning (back-propagation and stochastic gradient decent). The major advance in learning larger NNs are growing resources in computational power, especially in graphics processing units (GPUs) found on the computer's graphics card.

Prototypical applications for deep learning are Computer Vision, but also Language Understanding. In Game Playing, deep learning has made its way to play real-time strategy games by just looking at the screen and score data [6]. More formally the aim of this reinforcement technique is to learn a *policy network*, which outputs a probability distribution of the next move to play. Alternatively, in a *value network*, learning is used to predict the *game-theoretical value* in a single network output node, i.e., its expected outcome assuming perfect play.

¹ <http://deeplearning.net>.

As a sign of industrial relevance, Google bought the deep learning specialist DeepMind. DeepMind's *AlphaGo*² is a Go game playing program that applies a combination of neural network learning and Monte Carlo tree search. In March 2016, it won 4:1 against Lee Sedol in a match; proving itself to be the first computer program to ever beat a professional human player in Go [9]. This achievement is widely considered a landmark result in AI, previously estimated to become true only in the far future. The program *AlphaGo* was trained both with games played by humans and with ones played by earlier versions of *AlphaGo*. It is amazing that deep learning of thousands of expert games (in matters of days of GPU computation time) made the program *understand* the strategic concepts and tactics of Go.

AlphaGo learned to match the moves of expert players from recorded historical games. Once it had reached a certain degree of proficiency, it was trained further by playing games against other instances of itself. The input was a random permutation of expert game positions, made available in a number of Boolean input matrices of size 19×19 (some for the occupation and the colors that play, some for further features like liberty). The output was 19×19 matrix as a predictor for the next move. The convolutional neural network (CNN) that was trained was a *policy network*.

Cazenave [3] could reproduce *AlphaGo*'s results by training a deep neural network for the same set of expert games on a GPU-lifted PC infrastructure. The minimized error evaluation was comparable to the one obtained and reported by DeepMind [9]. The accuracy of finding the correct expert move was 55.56%, while *AlphaGo* had success rate of 57.0%.

Collocated to a human top tournament (79th Tata Steels) for the Google Alpha Chess Classics in Wijk an Zee 2017³ an entire tournament was set, where Alpha-Go-inspired chess engines will try playing close to the humans from whose games they has been trained, e.g., Anderssen, Pillsbury, Tarrasch, Capablanca, Tal, Smyslov, Fischer, some of which have not played against each other in real live.

Following Rojas [7], a neural network is graph representation of a function with input variables in \mathbb{R}^l and output variables in \mathbb{R}^k . The internal working is described through an activation function and a threshold applied at each network node. The input vector is a number of l features (e.g., in board game the board itself is often included in the feature vector). In a value network we have $k = 1$, while in policy networks we get a probability distribution for the successor set. Learning is the process of computing weights to the network edges to find a close approximation of the true network function via the (incremental) change of the weights through labeled input/output sample pairs. In *multi-layer feed-forward neural networks* (MLNN) there are input and output, as well as fully connected intermediate hidden layers, while for CNNs the input layers are more loosely connected through a number of *planes*.

² <https://deepmind.com/alpha-go.html>.

³ See de.chessbase.com/post/google-alpha-chess-classic.

The rest of the paper is structured as follows. First we introduce the concept learning of games and take TicTacToe as a first case study. We explain the steps taken to a trained neural network that can be used as an evaluation function to play the game. By the small problem size, this part of the paper has a tutorial character to guide the reader through the practical steps needed to construct a neural network game player. Nonetheless, the learning results are interesting, in the sense that the CNNs had problems to match the efficiency of MLNNs. Next, we turn to the SameGame, a single-agent maximization problem. We provide a state-of-the art NRPA player implementation of the game to compute the training sets for the neural network. Results of training CNNs and MLNNs are shown and discussed.

2 Case Study: TicTacToe

We exemplify the learning setup in TicTacToe (Fig. 1), where we construct and train a *value network*. The game is a classic puzzle that results in a draw in optimal play⁴.

X	_	X		1	0	1		1	0	1		0	0	0
_	0	_	->	0	1	0		0	0	0		0	1	0
0	_	X		1	0	1		0	0	1		1	0	0

Fig. 1. A TicTacToe position won for the X player, and its representation in form of input planes.

We used the prominent tensor and optimization framework *torch7*, which provides an interactive interface for the programming language LUA⁵. *Tensors* featured by the programming framework are numerical matrices of (potentially) high dimension. It already offers the support for optimizers like stochastic gradient decent, as well as neural network designs and training. For fast execution of tensor operations, *torch7* supports the export of computation to the graphic card (GPU) in CUDA⁶, a GPU programming framework that is semantically close to and finally links to C. The changes to the LUA code are minimal.

2.1 Automated Generation of Games

We kicked off with generating all 5478 valid TicTacToe positions, and determined their true game value by applying *retrograde analysis*, a known technique for constructing strong solutions to games. The according code is shown in Fig. 2. All classified TicTacToe positions are stored in *comma separated value* (CSV) files.

⁴ This has lead movies like *war games* to use it as an example of a game that consumes unlimited compute power to solve.

⁵ <http://torch.ch/>, the Linux installation is simple if the firewall does not block Github access.

⁶ <https://developer.nvidia.com/cuda-zone>.

```

retrograde()
  change = 1
  while (change)
    change = 0
    for each c = 1 .. 5478
      if (solved[c] == UNSOLVED)
        unpack(c)
        succs = successors(moves)
        if (moveX())
          onesuccwon = 0;
          allsuccslost = 1
          for each i = 1 .. succs
            apply(moves[i], 'X')
            onesuccwon |=
              (solved[pack()] == WON)
          allsuccslost &= (
            solved[pack()] == LOST)
          apply(moves[i], '_')
          if (succs & onesuccwon)
            solved[c] = WON; change = 1
          if (succs && allsuccslost)
            solved[c] = LOST; change = 1
        else
          onesucclost = 0;
          allsuccswon = 1
          for each i = 1 .. succs
            apply(moves[i], 'O')
            onesucclost |=
              (solved[pack()] == LOST)
          allsuccswon &=
            (solved[pack()] == WON)
          apply(moves[i], '_')
          if (succs && onesucclost)
            solved[c] = LOST;
            change = 1
          if (succs && allsuccswon)
            solved[c] = WON;
            change = 1
    for each c = 1 .. 5478
      if (solved[c] == UNSOLVED)
        solved[c] = DRAW

```

Fig. 2. Finding the winning sets in TicTacToe.

In one *network output* file the *values* for the value network are kept (for a policy network a suitable policy has to be used). In the other *network input* file, we recorded the according 5478 intermediate game positions. For each position, we took three 3×3 Boolean *planes* to represent the different, boards, one for the free cells, one for the X player and one for the O player.

2.2 Defining the Network

Next, we produced the input and output files for the neural network to be trained and tested. As shown in Fig. 3 we used *torch7* for the compilation of entries from the CSV input to the required binary format.

The NN consists of layers that are either fully connected (MLNN) or convoluted (CNN). The according LUA code is shown in Figs. 4 and 5. For CNNs it consists of a particular layered structure, which is interconnected through the

```

local Planes = 3
local csvFile = io.open('ttt-input.csv','r')
local input = torch.Tensor(5478,nPlanes,3,3)
local nb = 0
local currentnb = 0
for line in csvFile:lines('*l') do
    nb = nb + 1
    currentnb = currentnb + 1
    local l = line:split(',')
    local plane = 1
    local x = 1
    local y = 1
    for key, val in ipairs(l) do
        input[currentnb][plane][x][y] = val
        y = y + 1
        if y == 4 then
            y = 1
            x = x + 1
        end
        if x == 4 then
            x = 1
            plane = plane + 1
        end
    end
    if currentnb == 5478 then
        currentnb = 0
        nameInputFile = 'ttt-input.dat'
        torch.save (nameInputFile, input)
    end
    if nb == 5478 then
        break
    end
end
csvFile:close()

```

Fig. 3. Converting TicTacToe CSV to a tensor file.

```

require 'nn'
local net = nn.Sequential ()
net:add (nn.Reshape(27))
net:add (nn.Linear(27,512))
net:add (nn.Tanh())
net:add (nn.Linear(512,1))
local nbExamples = 5478
local input = torch.load ('ttt-input.dat')
local output = torch.load ('ttt-output.dat')
dataset = {};
function dataset:size() return nbExamples end
for j = 1, dataset:size() do
    dataset[j] = {input[j], output[j]};
end
criterion = nn.MSECriterion()
trainer = nn.StochasticGradient(net,criterion)
trainer.maxIteration = 1500
trainer.learningRate = 0.00005
trainer:train(dataset)

```

Fig. 4. Learning TicTacToe with an MLNN.

```

require 'nn'
local nPlanesInput = 3
local net = nn.Sequential ()
local nplanes = 25
net:add (nn.SpatialConvolution
  (nPlanesInput, nplanes, 3, 3, 1, 1, 0, 0))
net:add (nn.ReLU ())
net:add (nn.SpatialConvolution
  (nplanes, nplanes, 2, 2, 1, 1, 1, 1))
net:add (nn.ReLU ())
net:add (nn.SpatialConvolution
  (nplanes, nplanes, 2, 2, 1, 1, 1, 1))
net:add (nn.ReLU ())
net:add (nn.SpatialConvolution
  (nplanes, 1, 3, 3, 1, 1, 1, 1))
net:add (nn.ReLU ())
print(net)
net:add (nn.Reshape(1*3*3))
net:add (nn.Linear(9,1))
local nbExamples = 5478
local input = torch.load ('ttt-input.dat')
local output = torch.load ('ttt-output.dat')
dataset = {};
function dataset:size() return nbExamples end
for j = 1, dataset:size() do
  dataset[j] = {input[j], output[j]};
end
criterion = nn.MSECriterion()
trainer = nn.StochasticGradient(net,criterion)
trainer.maxIteration = 1500
trainer.learningRate = 0.00005
trainer:train(dataset)

```

Fig. 5. Learning to play TicTacToe with a CNN.

definition of planes in form of tensors. The hidden units were automatically generated by the tensor dimensions. This was done though defining sub-matrices of certain sizes and some padding added to the border of the planes. After having the input planes T_I represented as tensors and the output planes represented as tensors T_O (in our case a singular value) there are $k - 2$ spatial convolutions connected by the tensors $T_I = T_1, \dots, T_k = T_O$. The information on the size of sub-matrices used and on the padding to the matrix was used as follows. All possible sub-matrices of a matrix for a plane (possibly extended with the padding) on both sides of the input are generated. The sub-matrices are fully connected and the matrices themselves.

2.3 Training the Network

Deep learning in CNNs is very similar to learning in classical NNs. The main exception is an imposed particular network structure and the computational power to train even larger networks to a small error. For global optimization, usually stochastic gradient decent is used [1].

Figures 4 and 5 also show the LUA code for training the network. We experimented with an alternative formulation for the optimization process, but while other NN experts insist on batched learning to be more effective, for us it did not made much of a difference.

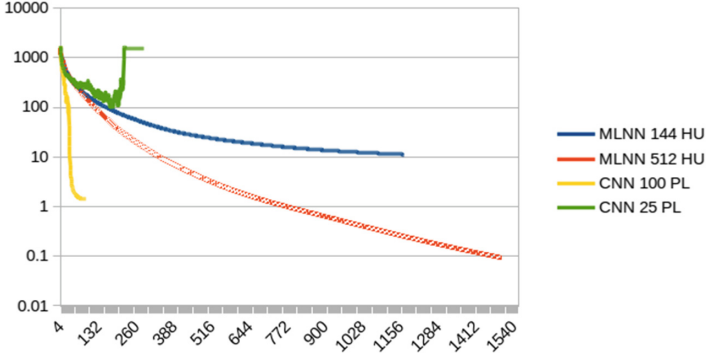


Fig. 6. Learning results in TicTacToe displaying the training error for the full set with multi-layer neural nets (MLNN) and convolutional neural nets (CNN).

Figure 6 shows the effect of learning different NNs design given the pre-computed classification of all valid TicTacToe positions. We see that larger networks (number of hidden units - HU, number of intermediate planes - PL) yield better learning. Moreover, CNNs tend to have the smaller number of learning epochs CNNs compared to MLNNs. However, each optimization step in a CNN is considerably slower than in a MLNN⁷. While learning MLNNs yields a rather smooth monotone decreasing curve the learning in a CNN has more irregularities. Moreover, CNNs tend to saturate. Worse, we observe that after reaching some sweet spot CNNs can even deviate back to very bad solutions.

2.4 Using the Network

A trained value network can be used as an estimator of game positions (usually called evaluation function) and integrated in other game playing programs. As TicTacToe is a known draw we are more interested in the average value accuracy in training vs. test data.

The following small test shows that the network has learned the value of the game for which we chose 0 for a won game, 50 for draw, and 100 for a lost game. The value network it can be used as an evaluation function and, thus, immediately leads to playing engines. In fact, by fixing the class ranges to 25 and 75, we could use the trained net as an optimal predictor of the true winning classes.

⁷ All experiments are executed one core of an Intel[®] Core[™] i5-2520M CPU @ 2.50GHz \times 4. The computer has 8 GB of RAM but all invocations of the algorithm to any problem instance used less than 1 GB of main memory. Moreover, we had the following software infrastructure. Torch7 with LUA, Operating system: Ubuntu 14.04 LTS, Linux kernel: 3.13.0-74-generic.

3 Case Study: SameGame

The SameGame is an exciting interactive game for humans and for computers played on an $n \times n$ (usually, $n = 15$ and $k = 5$) board with k colored tiles. Tiles can be removed, if they form a connected group of $l > 1$ elements. The score of a move is $(l - 2)^2$ points. If a group of tiles is removed, others fall down. If a column becomes empty, others move to the left, so that all non-empty columns are aligned. Total clearance yields an additional bonus of 1,000 points. The objective is to maximize the score.

Successor generation and evaluation has to find the tiles that have the same color. Figure 9 shows a simplified implementation for generating the successors. We used an explicit stack for building moves. Termination is checked by looking into each of the 4 directions for a tile of the same color.

3.1 Randomized State-Space Search

As the access to high-quality expert games in many domains is limited, the research question addressed in this paper is how to apply (deep) learning in state-space search without the input of human knowledge, where *state-space search* is a general term for the exploration of problem domains to search a solution that optimizes a given cost function [4]. State spaces are generated if the problem are intrinsically hard, which is often the case in games, due to the inherent combinatorial structure. The enumeration of state spaces, however, often suffers from the *state-explosion problem*, which states that the sizes of the spaces are exponential in the number of state variables (Fig. 7).

Randomized algorithms often show performance advantages to deterministic algorithms, as in the randomized test for primality [10, 11]. In *Roshambo*, random strategies are superior to deterministic ones. Randomization often turns out to be conceptually simple and, frequently, successful in large state-spaces to find the *needle in-the-haystack*. For example, most successful SAT solvers like *Lingeling*⁸ rely on randomized search (Fig. 8).

In the domain of single-agent combinatorial games, nested rollout policy adaptation (NRPA) is a randomized optimization algorithm that has found new records for several problems [8]. Besides its excellent results in games, it

Predicted Value	True Value
95.6280	100
-3.1490	0
50.8897	50
0.6506	0
⋮	⋮

Fig. 7. Accuracy of value neural network for TicTacToe.

⁸ <http://fmv.jku.at/lingeling>, with parallel implementations Plingeling & Treengeling.

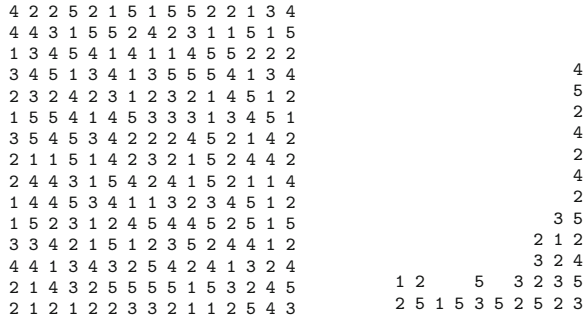


Fig. 8. Initial and final position in the SameGame.

```

legalMoves[m[]]
    succs = 0;
    visited.clear()
    if (!moreThanOneMove (tabu))
        tabu = blank;
    for each i = 1 .. n^2-1
        if (color[i] != blank)
            if (visited[i] == 0)
                buildMove (i, m[succs]);
                succs += |m[succs].tiles| > 1
    return succs

play (move)
    move.sort()
    for each i = 0 .. |move.tiles|-1
        remove(move.locations[i])
    column = 0
    for each i = 0 .. n-1
        if (color[n*(n-1) + column] == blank)
            removeColumn (column);
        else
            column++;
    currentScore += (|move.tiles|-2)^2
    + 1000 * color[n*(n-1)] == move
    rollout [length++] = move

```

Fig. 9. Generating the successors and executing a move in the SameGame.

has been effective in the applications, for example in IT logistics for solving constrained routing problems, or in computational biology for computing and optimizing multiple sequence alignments. NRPA belongs to the wider class of so-called Monte Carlo tree search (MCTS) algorithms, where Monte Carlo stands as an alias for random program execution. The main concept of MCTS is the *playout* (or rollout) of positions, with results in turn change the likelihood of the generation of successors in subsequent trials. Other prominent members in this class are *upper confidence bounds applied to trees* (UCT) [5] (that was applied to in *AlphaGo*), and *nested monte-carlo tree search* (NMCTS) that has been used successfully for combinatorial single-agent games with large branching factor [2]. What makes NRPA different to UCT and NMCTS is the concept of training a policy through a mapping of encoded moves to probabilities (in NMCTS the

```

Nrpa(level)
  Solution best
  if (level == 0)
    best.score = playout(global)
    best.rollout = rollout
  else
    best.score = Init
    backup[level] = global
    for(i=0..iteration)
      Solution r = Nrpa(level - 1)
      if (better(r.score,best.score))
        best = r
        adapt(r.score,r.rollout,backup[level])
    global = backup[level]
  return best

```

Fig. 10. Nested rollout policy adaptation.

policy is hidden in the recursive structure of the program’s decision-making, while in UCT the policy is represented partitioning in the nodes of the top tree that is stored).

NRPA was introduced in the context of generating a new world record in *Morpion Solitaire*. While the algorithm is general and applies to many other domains, we keep the notation close to games and will talk about boards, roll-outs, moves. The recursive search procedure is shown in Fig. 10. It requires a proper initialization value *Init* and comparison function *Better*, depending on whether a maximization or a minimization problem being solved. Different to UCT and NMCS, in NRPA every playout starts from an empty board. Two main parameters trade exploitation vs. exploration: the number of *levels* and the branching factor *iteration* of successors in the recursion tree. There is a training parameter α , usually kept at $\alpha = 1$. Successor selection refers to probabilities for each move are computed and recorded in a distribution vector. We implement *playout* and *adapt* based on domain-dependent successor generation and move encoding rules, functions `terminal`, `legalMoves`, `code`. The function `code` maps the move to an integer that addresses the value in the policy table. As it is called only during a playout it has access to all other information of the state that is produced. This way *code* realizes a mapping of state and move to a floating-point value.

The implementation for policy adaptation in Fig. 12 records the codes and the length of the playout in the successor selection *Select* (Fig. 11). This leads to the implementation of the generic *Playout* function (Fig. 13): each time a

```

Select(board, moves, pol)
  for each i .. moves.size
    c = board.code(moves[i]);
    probaMove[i] = exp(pol[c]);
    bestCode[0][board.length][i] = c;
  return probaMove

```

Fig. 11. A fitness selection module.

```

Adapt(length, level, p)
for each i = 0 .. length
    backup[level][bestCode[level][i]] =
        bestCode[level][i] + ALPHA
    z = 0
    for each j = 0 .. |bestCode[level][i]|
        z += exp(pol[bestCode[level][i][j]])
    for each j = 0 .. |bestCode[level][i]|
        p'[level][bestCode[level][i][j]] -=
            ALPHA * exp(p[bestCode[level][i][j]])/z

```

Fig. 12. An implementation of policy adaptation.

```

Playout(pol)
Board b
while(1)
    if (board.terminal())
        score = board.score ()
        scoreBestRollout[0] = score
        lengthBestRollout[0] = board.length
        for each k = 0 .. board.length
            bestRollout[0][k] = board.rollout[k]
        if (Better(score,bestScore))
            bestScore = score
            bestBoard = board
        return score
    moves = board.legalMoves(moves)
    nbMovesBestRollout[0][board.length] =
        moves.size
    probaMove = Select(board,moves,pol)
    sum = probaMove[0]
    for each i = 1 .. |provaMove|
        sum += probaMove[i]
    r = rand(0,sum)
    j = 0
    s = probaMove[0]
    while (s < r)
        s += probaMove[++j]
    bestCode[0][board.length] =
        bestCode[0][board.length][j]
    board.play(moves[j])

```

Fig. 13. The generic playout function.

new problem instance in form of an initial board is created. With *Select* the procedure calls the fitness evaluation.

3.2 Generating the Training Data

As a first step we generate input data for training the CNN using our Monte Carlo tree search solver. We used the benchmark set of 20 problem instances⁹ with board sizes $n = 15$. Each tile has one of 5 colors.

We ran a level-3 iteration-100 NRPA search 30 times. To compare with we also ran one NRPA(4,100) for each problem. All individual games were recorded, merged and subsequently split into 33972 partial states, one after each move, The

⁹ <http://www.js-games.de/eng/games/samegame>.

partial state were stored into an input file. For each partial state the move executed was stored into another file. The 33972 partial states were chosen randomly to avoid a bias in training the network.

3.3 Defining the Network

To specify a policy network for the SameGame the set of input planes were defined as follows. For each of the 5 colors plus 1 for the blank, we defined an

Table 1. Parameter finding for deep learning in the SameGame using 1000 of 33972 randomly chosen training examples, minimizing the MSE in stochastic gradient decent according to different learning rates λ .

$\lambda = 0.0005$	$\lambda = 0.005$	$\lambda = 0.05$	$\lambda = 0.5$	$\lambda = 0.2$
0.1429	0.1548	0.1437	0.2084	0.1567
0.1409	0.1418	0.1414	0.2088	0.1537
0.1404	0.1409	0.1398	0.2088	0.1533
0.1400	0.1408	0.1392	0.2088	0.1531
0.1395	0.1408	0.1388	0.2088	0.1527
0.1391	0.1407	0.1384	0.2088	0.1523
0.1387	0.1407	0.1382	0.2088	0.1521
0.1384	0.1406	0.1380	0.2088	0.1519
0.1382	0.1406	0.1378	0.2088	0.1517
0.1380	0.1406	0.1376	0.2088	0.1515
0.1378	0.1405	0.1375	0.2088	0.1515
0.1376	0.1405	0.1374	0.2088	0.1501
0.1374	0.1405	0.1373	0.2088	0.1457
0.1372	0.1404	0.1371	0.2088	0.1423
0.1370	0.1404	0.1349	0.2088	0.1434
0.1368	0.1404	0.1320	0.2088	0.1416
0.1366	0.1403	0.1291	0.2088	0.1373
0.1365	0.1403	0.1327	0.2088	0.1359
0.1363	0.1403	0.1324	0.2088	0.1353
0.1362	0.1402	0.1325	0.2088	0.1348
0.1361	0.1402	0.1323	0.2088	0.1338
0.1360	0.1402	0.1323	0.2088	0.1333
0.1359	0.1401	0.1322	0.2088	0.1344
0.1358	0.1401	0.1322	0.2088	0.1346
0.1357	0.1401	0.1321	0.2088	0.1352
\vdots	\vdots	\vdots	\vdots	\vdots

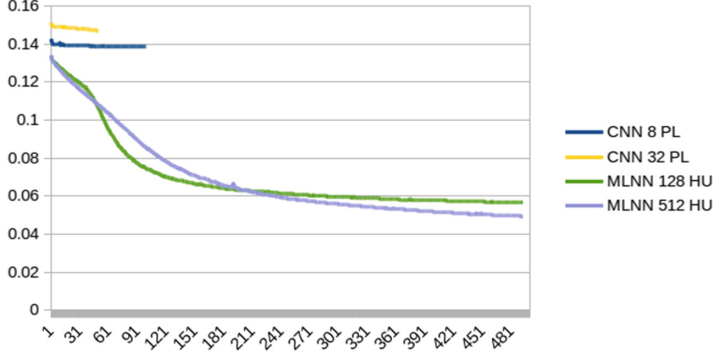


Fig. 14. Learning results in the SameGame displaying the change of the network error on the full training set of 33972 game positions for multi-layer neural nets (MLNN) and convolutional neural nets (CNN).

indicator matrix of size 15×15 for the board, denoting if a tile is present in a cell or not. This amounts to 6 planes of size 225, so that we had 1350 binary input features to the neural network. The output tensor file refers to one binary plane of size 15×15 representing a board, with the matrix entries denote the cells affected by the move. What was learned in this *policy network* is the distribution values on where to click.

3.4 Training the Network

Table 1 shows the effect of varying the learning parameter for the learning process on a fraction of all training examples. In Fig. 14 we see the effect of learning different neural networks given the precomputed randomly perturbed set of all SameGame training positions. The learning rate was 0.1–0.2 and the first 50–500 iterations of the optimization process are plotted. Again, it seemingly looks like that MLNNs can perform better in comparison with convolutional structures. Moreover the convergence was much faster, the largest MLNN took about 5 h and the smallest about 1.5 h, while the CNN took about 2 days of computational time on our CPU.

3.5 Using the Network

To validate our solution, we compared the MLNN network output after 1000 learning epochs (having an error of 0.0430463278) with the real output. In the visualization of Fig. 15 we used the threshold of 0.2 for a bit being set.

We see that more time is needed to reduce the error to a value in which can be used for playing well.

The subsequent integration of the neural network into the randomized NRPA engine, however, is simple, as we only need to change the initialization or rollout functions. There are three main implementation options.

- The distribution information is used as an initial policy matrix prior to the search.
- The NN recommendation and the learned policy are alternated by flipping a coin with probability p .
- The distribution of successors computed by the policy are merged with the NN recommended ones. If p_i and p'_i are the two probabilities for choosing the i -th of r successors, then the new probability is $p_i \cdot p'_i / \sum_{k=1}^r p_k \cdot p'_k$.

[illegible]

Fig. 15. Validation of learning result.

4 Conclusion

This paper explains the working of (deep) learning for training value and policy (neural) networks, to reflect its usage in game playing programs. In both of our case studies, we excluded human expert knowledge and used accurate and approximate computer exploration results instead.

Deep learning for TicTacToe is like shooting large bullets on too small animals, especially given that we have computed exact information on the game theoretical value for all reachable states beforehand. Nonetheless, we see the results of the learning process as being insightful. We were able to train the network to eventually learn the exact winning information in TicTacToe, and likely due to better separation, the wider the hidden layer(s), the better the learning.

Next, we turned to the SameGame, for which we applied a fast randomized solver. We used it to generate a series of good games (30 for each of the considered 20 instances). We obtained better learning curve with shallow MLNNs, which also lead to a drastic performance gain (about 20–30 fold speedup) compared to our CNN designs.

The sparser form of convolutions are often reported to perform better than ordinary multi-layerd neural networks that have fully connected hidden layers. The sparser interconnection between the network levels is balanced by a deeper network. To some extend our results can be interpreted in the sense that good neural learning does not always has to be *deep*, but sometimes *wide* and *shallow*.

References

1. Bottou, L.: Stochastic learning. In: Bousquet, O., Luxburg, U., Rätsch, G. (eds.) *ML -2003. LNCS (LNAI)*, vol. 3176, pp. 146–168. Springer, Heidelberg (2004). doi:[10.1007/978-3-540-28650-9_7](https://doi.org/10.1007/978-3-540-28650-9_7)
2. Cazenave, T.: Nested Monte-Carlo Search. In: *IJCAI*, pp. 456–461 (2009)
3. Cazenave, T.: Combining tactical search and deep learning in the game of go. In: *IJCAI-Workshop on Deep Learning for Artificial Intelligence (DLAI)* (2016)
4. Edelkamp, S., Schrödl, S.: *Heuristic Search - Theory and Applications*. Academic Press, London (2012)
5. Kocsis, L., Szepesvári, C.: Bandit based Monte-Carlo planning. In: *ECML*, pp. 282–293 (2006)
6. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., Hassabis, D.: Human-level control through deep reinforcement learning. *Nature* **518**(7540), 529–533 (2015)
7. Rojas, R.: *Neural Networks: A Systematic Introduction*. Springer, New York (1996)
8. Rosin, C.D.: Nested rollout policy adaptation for Monte-Carlo tree search. In: *IJCAI*, pp. 649–654 (2011)
9. Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., Hassabis, D.: Mastering the game of Go with deep neural networks and tree search. *Nature* **529**, 484–503 (2016)
10. Solovay, R.M., Strassen, V.: A fast Monte-Carlo test for primality. *SIAM J. Comput.* **6**(1), 84–85 (1977)
11. Solovay, R.M., Strassen, V.: Erratum a fast Monte-Carlo test for primality. *SIAM J. Comput.* **7**(1), 118 (1978)

Computer Games

5th Workshop on Computer Games, CGW 2016, and 5th
Workshop on General Intelligence in Game-Playing
Agents, GIGA 2016, Held in Conjunction with the 25th
International Conference on Artificial Intelligence, IJCAI
2016, New York, USA, July 9-10, 2016, Revised Selected
Papers

Cazenave, T.; Winands, M.; Edelkamp, S.; Schiffel, S.;
Thielscher, M.; Togelius, J. (Eds.)

2017, XII, 179 p. 59 illus., Softcover

ISBN: 978-3-319-57968-9