The *Layered Learning* method and its application to generation of evaluation functions for the game of checkers

Karol Walędzik and Jacek Mańdziuk

Warsaw University of Technology,
Faculty of Mathematics and Information Science,
Plac Politechniki 1, 00-661 Warsaw, Poland
{K.Waledzik, J.Mandziuk}@mini.pw.edu.pl

Abstract. In this paper we describe and analyze a Computational Intelligence (CI)-based approach to creating evaluation functions for two player mind games (i.e. classical turn-based board games that require mental skills, such as chess, checkers, Go, Othello, etc.). The method allows gradual, step-by-step training, starting with end-game positions and gradually moving towards the root of the game tree. In each phase a new training set is generated basing on results of previous training stages and any supervised learning method can be used for actual development of the evaluation function.
We validate the usefulness of the approach by employing it to develop heuristics for the game of checkers. Since in previous experiments we applied it to training evaluation functions encoded as linear combinations of game state statistics, this time we concentrate on development of artificial neural network (ANN)-based heuristics.

Games provide cheap, reproducible environments suitable for testing new search algorithms, pattern-based evaluation methods or learning concepts. Since the seminal papers devoted to programming chess [1–3] and checkers [4] in the 1950s., games remained through decades an interesting topic for both classical AI and CI-based approaches.

Most examples of application of CI methods to mind game playing make use of either reinforcement learning methods, neural networks-based approaches, evolutionary methods or hybrid neuro-genetic solutions, e.g. in chess [5–7], checkers [8–11], Go [12], Othello [13], or give-away-checkers [14, 15].

The main focus of this paper is on testing the efficacy of what we call *Layered Learning* - a generally-applicable approach to building the evaluation function for two-player games (checkers in here) which can be implemented either in the evolutionary mode or as a gradient backpropagation-type neural network training. The method, originally proposed in [16], was used previously by the authors in the case of linear heuristic composed of checkers-specific components [17, 18]. In this paper a more detailed description of the method is provided along with some modifications to the previously used version. Furthermore, as opposed to [17] and [18], where evolutionary learning methods were employed, this work concentrates on applicability of Layered Learning to the case of artificial neural networks (ANNs) based evaluation functions with much lesser use of pre-defined domain knowledge.

The reminder of the paper is organized as follows: in section 1 the basic idea of the proposed Layered Learning method is described along with its several modifications and enhancements. The next two sections present the experimental setup and the results of experiments, respectively. Conclusions and summary of possible research prospects are placed in section 4.

1 Layered Learning

1.1 Learning method

Layered Learning (LL), schematically depicted in fig. 1, is an end-game first method. Similarly to TD(λ) [19] it attempts to propagate the knowledge of final game results from endgame positions up the game tree. Still, we believe that it differs enough to be worth separate analysis and evaluation.

This learning scheme starts with division of the game tree into a number of disjoint stages, depending on the game progress. The simplest criterion that can be employed here in case of checkers is the number of moves performed. The whole process starts with positions expected to be very close to the end of the game. They are analyzed by a minimax algorithm (typically employing alpha-beta pruning) with a null evaluation function. It is assumed that in most cases the analysis will be able to reach the leaves of the game tree and the heuristic evaluation function will not be needed. The other cases are treated as draws and have neutral value assigned by the evaluation function.

Once a set of assessed game positions is obtained, it can be used as training data for any supervised learning approach, so as to create an evaluation function able to assess those endgame positions. In our experiments we employed both evolutionary methods (with various representations of heuristic evaluation functions) and backpropagation learning methods (in case of ANNs).

Having trained an evaluation function for one stage, the algorithm moves to the next stage, closer to the beginning of the game. A number of game states from this new stage are generated and, again, they are analyzed with a minimax algorithm. It is expected that its search depth will be enough to always reach positions from the previously trained stage. In that case, the result of previous stage can be used as the evaluation function for this analysis and another training set can easily be generated. This process, repeated for all game stages, should lead to creation of an evaluation function capable of assessing positions from all game stages (or an ensemble of such functions covering the whole game).

1.2 Method variations

The general approach described in previous section can be implemented in several different ways and its quality may be influenced by a number of fine details of the algorithm. Implementors of LL scheme have, of course, to tackle all typical hurdles of CI

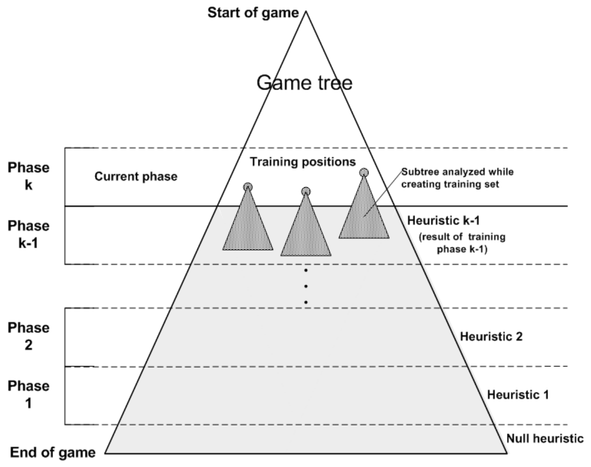


Fig. 1. *Layered Learning* method - an overview

methods application, such as choosing learning coefficients, designing ANN architecture, or defining evolutionary operators etc. There are, however, also several decisions typical for LL learning that must be made.

First of all, implementors should settle on supervised learning method. Our first experiments concentrated on using evolutionary methods - initially with evaluation functions represented as linear combinations of simple game state features. More sophisticated checkers position description features were introduced afterwards, and the definition of evaluation function was modified so as to allow dynamic switching of linear combinations' coefficients depending on game progress.

In the experiments described in this document we concentrated on evaluation functions represented by ANNs in the form of fully connected feed-forward multi-layer perceptrons. Input vectors would contain either only board content representation (with no preprocessing applied) or, alternatively, also values of a number of simple game state features. The ANNs would be trained either by backpropagation (RPROP [20]) or evolutionary methods.

Another problem faced by implementors of the LL method may be the risk of trained evaluators (be it ANN or any other representation) 'forgetting' knowledge learned during previous stages. There are several ways this issue can be dealt with. It is possible to train a separate evaluator in each stage and treat the resulting ensemble as the output of the training process. Each evaluator would then be used only in the stage it was trained for. Otherwise, special care must be taken to ensure that no (or next to no) 'forgetting' takes place. One way to achieve that is to make sure, that in each phase, evaluator is trained not only on positions from the current stage but also a number of game states from previous stages. These historical positions can be either regenerated each time they are needed (to improve the diversity of training positions), or reused in all subsequent training phases (to save time required for their regeneration and minimax analysis).

Whatever the choice, the current stage positions should be slightly over-represented in the training set, as they introduce new knowledge not yet acquired by trained evaluator.

One of the problems obvious in most training approaches in the domain of CI application to mind games is the selection of training games, positions or opponents. Since some players can be very successful against specific opponents while at the same time being of inferior quality to all the others, it is important to train them on a wide selection of game strategies they should be able to deal with. In the case of LL, training positions are in the simplest case generated by playing random games till given depth in game tree is reached.

This approach, however, brings about the risk that the training game states will not be representative of positions encountered in real games against intelligent players. One of the possible ways to circumvent this risk is modification of the positions generation process. Instead of random players, a set of varied intelligent agents can be used to play the games (possibly changing playing agent after each move) in order to generate the required collection of game states. Results of preliminary tests of this approach proved, however, to be unsatisfactory. This may, nevertheless, have been caused by poor selection of playing agents and we still consider this idea worth further testing.

2 Experiments setup

First of all, it should be stressed that the aim of the experiment was not to create a master level player capable of competing with commercial checkers applications. Our solution was not fully optimized for speed, employed only basic alpha-beta pruning algorithm with no further modifications and used only simple fully-connected ANNs for evaluation function representation.

During our experiments we tested several different sets of control parameters in combination with varied evaluation function architectures, which makes it impossible to list all of them in such a short document. Still, we will point out the most typical values (or intervals) of the coefficients used during training and indicate whenever an atypical value was employed. We also believe that there is a huge potential for results improvement by further tuning all the learning parameters and coefficients, considering how little attention has yet been devoted to the LL method.

We concentrate in this paper on analysis of the learning method itself and try to prove its applicability to mind games such as checkers, especially in zero-initial-knowledge training scheme. We hope to present LL method to wider audience and point out possible directions for further research.

2.1 Neural networks architecture

All our ANN-based experiments (as opposed to earlier experiments described in [17, 18]) involved feed-forward fully-connected multi-layered perceptrons. Ideally, we wanted their input vectors to contain board description only. They would, therefore, consist of 32 neurons representing individual board squares, each with one of five values: -2, -1, 0, 1 and 2 representing, respectively, opponent's king, opponent's checker, empty square and current player's checker or king. In some of the experiments the input layer would further be extended to contain a number of simple game state description features:

– differences between player's and opponent's checkers and kings counts;
– differences between player's and opponent's safe (i.e. adjacent to the edge of the board) checkers and kings counts;
– differences between player's and opponent's moveable (i.e. able to perform move other than capturing, ignoring capturing priority) checkers and kings counts;
– difference between player's and opponent's aggregated distances of checkers to promotion line;
– difference between player's and opponent's unoccupied fields on promotion line count.

For comparison, some experiments with input vectors containing only the game state features (without raw game state representation) were performed as well. In most experiments, the neural networks would contain one hidden layer of up to 10 neurons. Output layer would always contain a single neuron expected to output game state evaluation within the interval $[-1,1]$.

2.2 RPROP training

During the first phase of our experiments evaluators were trained using RPROP backropopagation method. In order to minimize the chance of random factors hindering the learning process, learning process was augmented by elements of evolutionary training procedures. At every stage of the algorithm 8 networks were trained simultaneously on the same training set. Each training phase consisted of 4 generations. After each generation, the quality of all candidate evaluators was tested by computing their mean square errors on a test set. Test set used for this task was separate from training sets and contained a number of game positions from all game stages trained on so far (including the current one). Candidate solutions were afterwards sorted based on their thus measured quality and the worse half of them was replaced by mutated copies of the best networks.

During the RPROP learning, training patterns were presented to the networks in random order (independent for each network). After each training phase, a more significant modification of the population took place. Only two best networks survived intact to the next phase. Additional 4 were generated by mutating them - once with lower (0.01 to 0.03) and once with higher (0.1) mutation probabilities. Further two candidate solutions were created with fully random weight values.

Mutation was applied independently to each weight in the mutated network, with each connection having equal probability to mutate. Once it was decided that given connection value should change, one of four possible mutations would be applied to it (each with equal probability): multiplication by 2, division by 2, sign change (multiplication by -1) or replacement with random value.

Results of preliminary experiments comparing training for varied number of epochs and with varied training set sizes (not presented here in details due to lack of space), suggest that one of the main problems hindering further improvement of the solutions generated by backpropagation method was the risk of overtraining. With high ANN capacities, small training sets and long training the networks would quickly loose their generalization capabilities. This would result in low training set errors but higher test set errors and poor performance in actual comparison games.

In order to overcome this hurdle, in the subsequent experiments we limited the ANNs' sizes and attempted to use training sets as big as possible, which, of course, resulted in slower training process. This meant, however, that in order to keep the training time within reasonable limits we had to reuse the same training boards across multiple training phases (continuously increasing the training set size with positions from lower depths in game tree).

Finally, we decided to employ early stopping routine as a way to define stop condition for training, so that the probability of overtraining is reduced. It proved, however, less successful than we expected. It turned out that the specificity of the problem caused the validation set error to fluctuate significantly - sometimes rising for several epochs only to drop afterwards. Early stopping, even modified to accept temporary rise of validation set error, was prone to ceasing the training too early, which forced us to train all networks for a preset number of epochs before the technique was employed.

2.3 Evolutionary training

Second phase of our experiments made use of a simple evolutionary approach. The trained population would, in this case, contain several dozen (up to 100, depending on individual experiment settings; 40 in most runs) candidate networks that would be modified over several hundred generations in each training phase.

In each generation mean square error of training boards assessment was calculated for each candidate ANN. Afterwards, a number (55% of population size in the most successful experiments) of the worst performing solutions were discarded. The remaining individuals were replicated with mutation, with a subset of them (typically the top 5% of the original population size) being replicated twice (once with lower and once with higher mutation probability coefficients). In case the resulting number of individuals was still lower than the requested population size, additional candidate solutions were generated randomly.

After each phase, population was refreshed in similar manner but with different control parameters. In that case, only 30% of the population would survive. Thus, after each generation a significant number of candidate solutions was regenerated randomly.

In most experiments the evolutionary method was additionally augmented by element of RPROP training. Namely, each newly created (be it randomly or via replication and mutation) network was first once trained on all training patterns. The RPROP training was also repeated for all networks after each training set change, i.e. at the beginning of each training phase.

3 Results analysis

3.1 Evaluators comparison

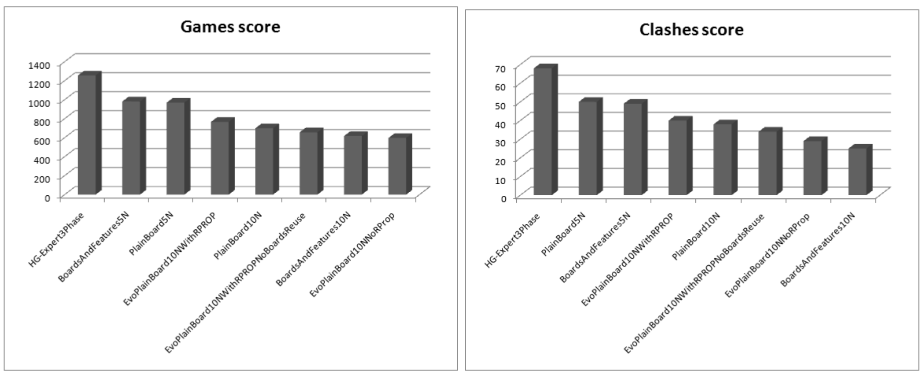
In order to analyze the results of our experiments we first decided to perform direct comparison of resulting evaluation functions by means of tournament, in which each agent played 20 games against every other one (with sides swapped after each game). Search depth limit for alpha-beta algorithm used in these comparison was set to relatively low limit of 4 - thus increasing the influence of evaluation function quality on the final score (in the case of greater search depths, score differences might prove less significant). Games played by each selected pair of agents were pairwise different thanks to the fact that, in each position, available moves were considered by the alpha-beta algorithm in random order. In order to make the results of the tournament as representative of the true quality as possible, it included a significant number of various agents trained in these and earlier experiments. Two scoring schemes were used in the tournament: games-based and clashes-based. In the former, agents were assigned points for each individual game: 2 points for a win and 1 point for draw. In the latter, contestants were scored analogically based on 20-game clashes (series of games against a single opponent).

Figure 2 presents results of the tournament for selected most important evaluation functions:

- *HG-Expert3Phase* the most successful evaluation function generated in the first Layered Learning experiment described in [18], consisting of 3 linear combinations of advanced game position features - each applied to one of disjoint phases of the game;
- *BoardsAndFeatures5N* - ANN trained with backpropagation method (RPROP), with 5 neurons in its single hidden layer and input vector containing both plain board description and basic game position numerical features;
- *BoardsAndFeatures10N* - ANN similar to the previous one but with doubled number of neurons in hidden layer;
- *PlainBoard5N* - ANN, differing from *BoardsAndFeatures5N* only in size of its input layer, as it did not include precalculated checkers position features;
- *PlainBoard10N* - ANN similar to the previous one but with doubled number of neurons in hidden layer;
- *EvoPlainBoard10NWithRPROP* - ANN trained using evolutionary approach (augmented by RPROP procedure), with hidden layer of 10 neurons;
- *EvoPlainBoard10NNoRPROP* - ANN with architecture identical to the previous one, but trained with pure evolutionary approach (with no backpropagation learning component);
- *EvoPlainBoard10NWithRPROPNoBoardsReuse* - yet another identical ANN, but this time trained with training boards set fully regenerated after each phase.

Based on the results of the described tournament and several minor comparisons performed independently, several conclusions can be drawn. First of all, it can easily be spotted that none of the evaluation functions generated in the current experiment managed to surpass the best results of the original experiment based on game state description features. The explanation of this fact is twofold. Firstly, the experiments differed in their focus and amount of learning parameters tuning applied. More importantly, however, it should not be forgotten that the most successful linear heuristics operated on manually defined advanced game positions characteristics. At the same time ANN had access to either raw board position or the simplest game position features only.

What is interesting, no significant difference in quality was observed between evaluators being provided with raw board representation only and those having additionally access to simple game state statistics. Since earlier experiments confirmed that those statistics are actually important in evaluation function building, it can be inferred that



(a) Individual games (b) 20-game clashes

Fig. 2. Comparison of evaluators based on individual games scores or clashes scores

the trained neural networks were actually able to successfully learn to compute at least some of them basing on the raw position description only.

Better results of smaller networks confirmed our preliminary expectations of overfitting being a common and serious problem for our training process. It should also not be omitted here that our decision to mix elements of evolutionary and backpropagation training was a highly successful one. Both training methods yielded far weaker solutions when used independently. In case of evolutionary method this fact is clearly visible in figure 2 with *EvoPlainBoard10NNoRPROP* being scored more than 20% lower than *EvoPlainBoard10NWithRPROP*. At the same time, analysis of the training logs of the backpropagation-based processes clearly indicates that in this case a significant number of mutations led to improvement of ANNs' mean square errors.

3.2 Training progress analysis

In order to verify the training process itself and evaluators' quality improvement from phase to phase, for several selected individual experiments we decided to perform further tournaments comparing solutions generated in subsequent phases of the same training process. We were aware that poor choice of training configuration might cause the evaluators to loose during the training knowledge gathered in earlier phases. What is more, any errors in heuristic generated in one of the early phases, would be repeated or even magnified during the training process, because the results of previous phases are used to generate training sets for further training.

It should also be stressed that, even if none of the above dangers actually applied, in case of such a comparison we had no reason to expect a monotonous increase in scores, as all but the last few evaluation functions were in no way prepared to play full games, having been trained only on their final stages. This fact might have lead them to choosing seemingly random moves in the first parts of games which could in consequence cause the objectively better end-game players to arrive at very disadvantageous positions. We expected, however, the heuristics generated in last training phases to play significantly better than the earlier ones.

Since in our experiments we decided to divide the game into 14 stages, we expected the winner to be one of evaluation functions generated in stages 11 to 14. This assumption proved, in general, to be true. The results of further classification proved, however, sometimes surprising. In some cases (for example for *PlainBoard5N* as visible in figure 3) one or more of the early stage evaluators turned out to be unexpectedly strong players as well. This can be attributed to the fact that it can be expected for such early heuristics to rely heavily on material differences and such a simple approach may be enough to beat opponents using evaluation functions being more sophisticated but applicable to mid-game positions only (with no ability to play any reasonable opening moves).

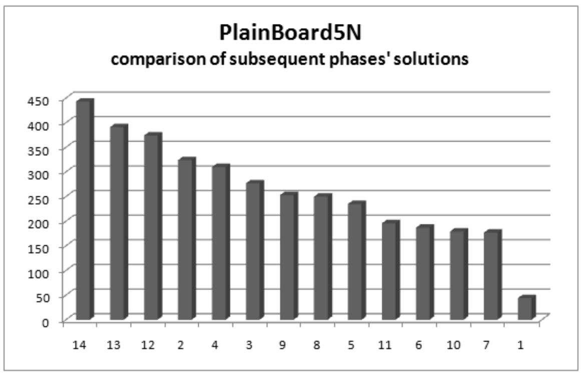


Fig. 3. Comparison of results of subsequent phases

4 Conclusions

In this paper a generally-applicable game learning approach to creating evaluation function for two-player games has been described.

To verify usefulness of this training method, we decided to apply it to the game of checkers. Following our previous experiments, in which we evolved linear-combination-based heuristics, this time we concentrated on training ANNs.

We believe that our experiments prove the method is worth further analysis, testing its various aspects and applicability to other mind games. We identified some of the most troublesome aspects of the approach and proposed several modifications to it, that we think are worth further research.

Although the method was introduced some time ago [16] it hasn't been extensively researched yet. Since only few experiments utilizing LL method have been performed so far, we think that its true potential is still yet to be discovered. We also believe that the name Layered Learning coined in this paper aptly describes the general idea of the method.

Our current research is focused on direct comparison of the LL method with the TD(λ) learning scheme.

References

1. Shannon, C.E.: Programming a computer for playing chess. Philosophical Magazine **41 (7th series)** (1950) 256–275
2. Turing, A.M.: Digital computers applied to games. In Bowden, B.V., ed.: Faster than thought: a symposium on digital computing machines. Pitman, London, UK (1953)
3. Newell, A., Shaw, J., Simon, H.: Chess-playing programs and the problem of complexity. IBM Journal of Research and Development **2** (1958) 320–335
4. Samuel, A.L.: Some studies in machine learning using the game of checkers. IBM Journal of Research and Development **3** (1959) 210–229
5. Kendall, G., Whitwell, G.: An evolutionary approach for the tuning of a chess evaluation function using population dynamics. In: Proceedings of the 2001 Congress on Evolutionary Computation CEC2001, IEEE Press (2001) 995–1002
6. Fogel, D., Hays, T., Hahn, S., Quon, J.: A self-learning evolutionary chess program. Proceedings of the IEEE **92** (2004) 1947–1954
7. J. Baxter, A. Tridgell, and L. Weaver. Learning to play chess using temporal differences. *Machine Learning*, 40(3):243–263, 2000.
8. Fogel, D.B.: Blondie24: Playing at the Edge of Artificial Intelligence. Morgan Kaufmann (2001)
9. Chellapilla, K., Fogel, D.: Evolution, neural networks, games, and intelligence. Proceedings of the IEEE **87** (1999) 1471–1496
10. Aleksander, I.: Neural networks - evolutionary checkers. Nature **402** (1999) 857
11. J. Schaeffer, M. Hlynka, and V. Jussila. Temporal difference learning applied to a high-performance game-playing program. In *International Joint Conference on Artificial Intelligence (IJCAI-01)*, pages 529–534, 2001.
12. N. N. Schraudolph, P. Dayan, and T. J. Sejnowski. Learning to evaluate Go positions via Temporal Difference methods. In N. Baba and L. C. Jain, editors, *Computational Intelligence in Games*, volume 62, pages 77–98. Springer Verlag, Berlin, 2001.
13. Moriarty, D.E., Miikkulainen, R.: Discovering complex othello strategies through evolutionary neural systems. Connection Science **7** (1995) 195–209
14. J. Mańdziuk and D. Osman. Temporal difference approach to playing give-away checkers. Proc. ICAISC 2004, Zakopane, Poland, vol. 3070 of *LNAI*, 909–914. Springer, 2004.
15. D. Osman and J. Mańdziuk. Comparison of tdleaf(λ) and td(λ) learning in game playing domain. Proc. ICONIP 2004, Calcutta, India, vol. 3316 of *LNCS*, 549–554. Springer, 2004.
16. M. Borkowski. Analysis of algorithms for two-player games. M.Sc. Thesis, Warsaw University of Technology (in Polish), 2000.
17. Kusiak, M., Walędzik, K., Mańdziuk, J.: Evolution of heuristics for give-away checkers. Proc. ICANN 2005, Part 2, Warszawa, Poland, vol. 3697 of *LNCS*, 981–987, Springer, 2005
18. J. Mańdziuk, M. Kusiak, K. Walędzik: Evolutionary-based heuristic generators for checkers and give-away checkers. Expert Systems, 24(4): 189–211, Blackwell-Publishing, 2007.
19. R. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.
20. M. Riedmiller and H. Braun, Rprop- a fast adaptive learning algorithm, 1992. [Online], *citeseer.ist.psu.edu/riedmiller92rprop.html*