

Machine Learning in the Game of Breakthrough

Richard J. Lorentz^(✉) and Teofilo Erin Zosa IV

Department of Computer Science, California State University,
Northridge, USA

lorentz@csun.edu, tzosa@ucsd.edu

Abstract. Motivated by recent activity in using machine learning techniques to game programming we present a study of applying these techniques to the game of Breakthrough. Specifically, we show that by using temporal difference learning in an MCTS setting we are able to achieve results almost equal to those obtained by WANDERER, a very strong program with a highly tuned evaluation function. We also show that convolutional neural networks trained by using WANDERER as a provider of expert moves can produce a program much stronger than the original. Even in an environment with very slow execution speeds, we can achieve excellent results.

1 Introduction

Breakthrough is a fairly new game, invented in 2000 by Dan Troyka and was the winner of the 2001 about.com 8×8 Game Design Competition. It provides an interesting mix of strategy and tactics and so is a popular game for games researchers [1–5].

It is played on an 8×8 board. The first two rows are populated with White's pieces and the last two rows with Black's. The first row is White's home row and the eighth row is White's goal. White moves their pieces towards their goal, one square forward at a time, either straight or diagonally. White may not move to a square already occupied by a white piece and may only capture a black piece if it is a diagonal move. White cannot move straight forward to a square occupied by a black piece. Black moves similarly. The first player to reach their goal row or capture all the opponent's pieces is the winner. A draw is impossible in Breakthrough.

Figure 1 shows two Breakthrough positions, both with White to move.¹ The one on the left is the starting position and the one on the right is after 22 moves have been played. From the second position the white piece on g2 may move either to g3 or h3, the piece on f3 may either move to g4 or capture on e4, and the piece on f4 may make either diagonal move.

Starting with WANDERER, a strong MCTS/EPT (Monte-Carlo Tree Search with Early Plyout Termination) based program that plays Breakthrough [3], we

R. J. Lorentz and T. E. Zosa IV—Have contributed equally to this work.

¹ Images are taken from the Little Golem website [6].

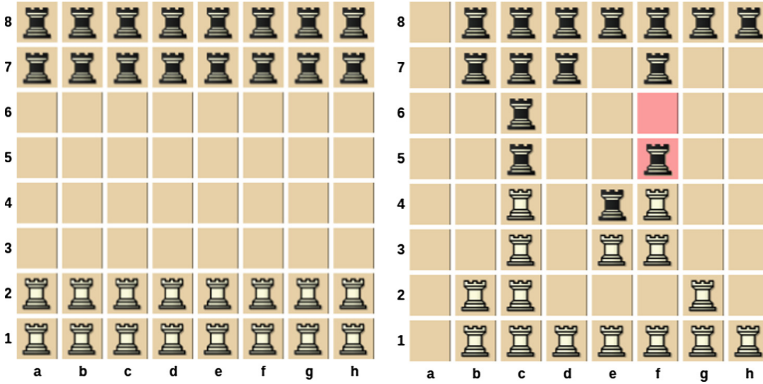


Fig. 1. Sample Breakthrough positions.

created two new versions: WAN-TDL uses temporal difference learning (TDL) techniques to create an evaluation function and WAN-CNN uses convolutional neural networks (CNNs) to generate a probability distribution over moves with respect to an expert WANDERER given a particular board configuration.

Machine learning techniques have long been used in game programming. In 1995 Tesauro used TDL to create a world class Backgammon playing program [7] and over the years many other game playing programs have used TDL in one form or another [8,9]. We use a straightforward implementation of TDL to train an evaluation function that replaces the hand-tuned evaluation function in WANDERER in order to create WAN-TDL. In Sect. 2 we describe the various patterns (n -tuples) we worked with, our reasons for choosing those patterns, and the results associated with each pattern choice.

Last year DeepMind famously created ALPHAGo, a program that uses CNNs and played the game of Go at a level even higher than the world’s best human players [10,11]. In Sect. 3 we explain how we apply these ideas to create WAN-CNN. First we created CNNs for policy networks that can rank the possible moves from any board position. This was done by using moves generated by WANDERER as there are insufficient publicly available games played by strong players to draw from. Then we created a new MCTS program in Python that could use the policy net in an MCTS setting.

Finally in Sect. 4 we summarize our results and discuss future work.

2 Temporal Difference Learning and Breakthrough

TDL is an unsupervised reinforcement learning technique whose history goes back at least to Samuel’s checker program [12,13]. Indeed, in 1959 Samuel captured the spirit of TDL when he said that a strong program to play a game (of checkers) can be produced using machine learning “...when given only the rules of the game, a sense of direction, and a redundant and incomplete list of parameters which are thought to have something to do with the game, but whose

correct signs and relative weights are unknown and unspecified” [12]. It is in this spirit that we used TDL to program Breakthrough. To be a bit more precise, we employed $\text{TD}(\lambda)$ a more recent and probably the most used version of TDL in game programming [14].

2.1 TDL Parameters

Corresponding to the parameters mentioned in the above quote we use patterns of board squares on the Breakthrough board. This technique is usually referred to as the n -tuple approach, however we feel the term *pattern* is more suggestive in this context.

We experimented with numerous pattern shapes, but all were simply small rectangular regions. For example, one pattern shape we used was the 3×3 rectangle. We considered every possible 3×3 rectangle on the board, for a total of 36 different rectangles. However, there is vertical symmetry on the Breakthrough board, e.g., the rectangle in the lower left corner is the same as the reflected rectangle in the lower right corner. This means there are actually only 18 unique rectangles.

Continuing this example, each rectangle contains 9 board squares and each board square can either be occupied by White or Black, or can be empty. Hence each 3×3 rectangle defines a total of $3^9 = 19,683$ distinct patterns. Since there are 18 different rectangles, there are a total of $36 \times 19,683 = 354,294$ different patterns and thus the same number of parameters. The board evaluation is then calculated simply as the sum of all the weights of all the patterns that appear on the board. We store the weights for all of these patterns as a file of float variables. In this case all of the weights required approximately 2 MB of memory (see Table 1).

Besides the weights on the patterns there are three other parameters that control the behavior of $\text{TD}(\lambda)$. TDL updates the parameter weights of patterns seen in previous board states based on the evaluation of the current board state. The trace decay parameter, λ , determines how many previous board states get updated. In our experiments we always set $\lambda = 0$, that is, only parameters from the previous board state are updated.

The second parameter, α , sometimes called the learning rate, is a multiplicative constant for the update values thus regulating the speed in which parameters can change. After experimenting with different values and changing schemes we settled on a constant value of $\alpha = 0.01$ for all of our experiments.

Finally, the standard method of ensuring sufficient exploration in the learning process is to use an ε -greedy policy where a random action is taken with probability ε . We chose $\varepsilon = 0.1$ for all of our tests.

One of the difficulties with TDL is determining if and when the weights have converged. The larger the pattern size the longer it will take because (1) there are more weights to train and (2) training games take longer due to the extra time required to calculate the evaluation with so many weights. Empirically, we found that training with one hundred million games worked well, though that by no means guarantees that the weights have to (or ever will) converge. For the

smallest pattern size this would take about an hour while for the largest it would take about 2 days. Pattern sizes are discussed in detail in the next section.

2.2 TDL Wanderer

Our goal is to see how TDL performs in an MCTS setting. WANDERER is a very strong program that uses MCTS with EPT. Specifically, WANDERER playouts run for 6 moves and then an evaluation function is called to determine the status of the position. In our experiments we simply replaced the finely tuned, hand coded evaluation with the one created using TDL. When testing, the first two moves of both programs are random. This, along with the nondeterminism inherent to MCTS provides sufficient variety in the games.

As can be seen in Table 1 we experimented with eight different rectangular patterns of various shapes of type $m \times n$ where m is the number of rows and n is the number of columns. If two sizes are mentioned that means all rectangles of both sizes were used. For example, 5×3 & 3×5 means that all rectangles of both size 5×3 and size 3×5 that fit on the board were used where all weights for every pattern are treated equally. It is worth comparing the number of patterns in this second example with the 3×3 case. Recalling the vertical symmetry, for 3×5 rectangles there are $2 \times 6 = 12$ possible placements and $3 \times 4 = 12$ possible placements of 5×3 rectangles. These rectangles contain 15 squares, each with 3 possible values, for a total of $12 \times 12 \times 3^{15} = 172,186,884$ weights compared with 354,294 in the 3×3 case, making it about 500 times bigger.

From a human point of view there are three main strategic concepts that one must know to play the game well: (1) Pieces are important – do not give them away; (2) Pieces in your home row in columns b, c, f, and g should not move until absolutely necessary because they play an important defense role; (3) Do not trade pieces on the opponent’s side of the board because that usually means you have given up important tempo. We were surprised to see that even with the smallest patterns WAN-TDL was able to learn these concepts. We find this quite remarkable considering the abstract and global nature of these rules compared with the size and local nature of the patterns.

Still, 3×3 patterns were insufficient to produce a player as strong as WANDERER so we added more patterns. We did this using the obvious idea of increasing the size of the rectangle, by allowing multiple sized rectangles, and by differentiating patterns according to what phase the game was in. For eight variations of these ideas we ran tests against basic WANDERER. In each case we played 1000 games, where each player was white half the time and both players were given 10 secs/move. Table 1 summarizes our results and is explained in more detail below.

We have included a column showing the actual size of the weights file to show that in some cases we are approaching the limits of reasonable file sizes. The win rate column shows the percentage of games won by WAN-TDL and the 95% confidence bounds.

Game progress means that a different collection of weights are given depending on the stage of the game. Stages are defined according to how many pieces

Table 1. Basic WANDERER vs. TDL with various patterns at 10secs/move

Pattern shape	File size	Win rate of TDL version
3×3	2 MB	$23.5\% \pm 2.7$
3×5	557 MB	$34.9\% \pm 3.0$
4×3	35 MB	$43.0\% \pm 3.1$
5×3	490 MB	$43.9\% \pm 3.1$
5×3 & 3×5	1.1 GB	$44.8\% \pm 2.9$
4×4	1.9 GB	$46.1\% \pm 3.1$
4×3 & 3×4 + game progress	418 MB	$46.3\% \pm 3.1$
4×3 & 3×4	81 MB	$46.6\% \pm 3.1$

are on the board for a total of seven stages with piece count cutoffs at 32, ≥ 30 , ≥ 28 , ≥ 24 , ≥ 20 , ≥ 16 , and < 16 .

Though it is difficult, if not impossible, to articulate what additional strategic concepts have been learned when using more patterns, we can see from the table that the program has clearly improved. It seems that patterns of sizes 4×3 & 3×4 provide the best results, especially relative to their size.

3 Convolutional Neural Networks and Breakthrough

Traditional MCTS induces a policy from a directed search of the state space. While MCTS is a flexible approach that has been shown to converge to an optimal policy in the limit [15], it seldom yields such a policy in scenarios that require real-time decision making.

Conversely, heuristic models such as CNNs have the benefit of producing predictions in far less time. The main drawback with these models is that the value of the prediction may be specious, heavily dependent on its architecture, how it is trained, and what data it is trained on.

While these two methods have been used in different domains with varying levels of success, the combination of both has produced powerful agents that rival expert human players [10] or far more sophisticated programs [16] in the game of Go. However, the neural networks used in these agents were trained on data from expert human players. As Breakthrough is fairly nascent, there is an absence of readily available human generated data from which to train a neural network.

Related work by Guo et al. [17] remedied the data dearth problem in the context of Atari games, by training neural networks on data generated by an external agent that utilized MCTS. They found that this data was sufficient to produce neural networks that could successfully play games such as Breakout, Pong, and Q*bert. Based on these findings, we sought to investigate whether this same approach would hold true in the domain of Breakthrough.

3.1 Methods

We began by training two four-layer CNNs to play Breakthrough by classifying which move an expert WANDERER would make in a given position. The *player* neural network, which predicts the player’s moves, would be trained only on moves from winning players while the *opponent* neural network, which predicts the opponent’s moves, would be trained on all moves.

Training Data. Training data consisted of Breakthrough games played between two instantiations of WANDERER, allowing the training set to consist of moves from both players. Each search expanded 93,750,000 nodes, taking between 180 and 420 s. This specific number of nodes was chosen due to the fact that experiments at higher numbers of nodes (up to 200,000,000) would typically (~90% of the time) converge to an optimal move at approximately 93,000,000 nodes. Limiting WANDERER to 93,750,000 nodes ensured that a dataset could be generated in a reasonable amount of time while not sacrificing move quality.

Next, 80 instances of self-play were run in parallel for approximately two months, yielding 3,735,662 moves from 37,688 games. As Breakthrough boards exhibit vertical symmetry, reflected moves were added to the training set, doubling the original number of training examples for a final training set total of 7,471,324 moves (3,735,662 of which were winning moves). Horizontal symmetry was also exploited, explained further in Sect. 3.1.

Data Pre-processing. For each game, moves were used to generate a sequence of board states, with each board state having an associated transition (move) to the next state. To take advantage of horizontal symmetry, we transformed the board representations so that the player’s pieces started in the first two rows while the opponent’s pieces started in the last two rows, and moves were made relative to the player’s location on the board. We will refer to this representation as the player point-of-view representation. This had the benefit of minimizing the amount of information the neural net had to learn (viz. color differentiation) and cut the number of possible classes in half (e.g., a1–a2 and h8–h7 map to the same move, relative to the player).

For neural network input, we transformed each board representation into a one-hot tensor with dimensions $8 \times 8 \times 4$, with an on-value of 1 for positions that correspond to player, opponent, empty, or bias positions, respectively. The bias is an on-value for every position.

For labeling of training examples, each move is mapped to a unique numerical index in the range 0 to 154, and this index serves as the label for the associated board representation. (With a player point-of-view representation, there are a total of 154 possible moves, according to start square and destination square plus a no-move position if the game is over.)

Network Architecture. The network’s input layer takes in a $b \times 8 \times 8 \times 4$ tensor where b is the mini-batch size, or the number of samples to be evaluated

in parallel. Each hidden layer has $192 \ 3 \times 3$ filters, with the exception of the final hidden layer which has a single 1×1 filter, and uses a rectified linear unit (ReLU) activation function [18]. As input is passed to each hidden layer, it is zero padded to the dimensions that ensure the resulting output volume has 8×8 planes. Hidden layer weights were randomly initialized with 0 mean and standard deviation $\sqrt{2/n_l}$ as described by He et al. [19, p. 4], where n_l is the number of incoming connections for that layer. The biases were initialized to 0.01 to ensure all ReLUs fired in the beginning [20]. The output layer is a fully-connected layer which uses a Softmax activation function [21] to produce the predicted probability distribution over the 155 possible moves. The output layer’s weights were initialized by the Xavier initialization algorithm [22] and biases were initialized to 0. The network was implemented in TensorFlow, Google’s machine learning framework, as TensorFlow offers a high level of performance, portability, and an interface that facilitates quick debugging and iterative development [23].

Network Training. The network was trained for five epochs via adaptive moment estimation, also known as Adam [24], using cross entropy with a learning rate of 0.1 and a mini-batch size of 128. Training took approximately one hour on an Nvidia GTX 1080 GPU.

3.2 Results

Figure 2 shows the *player* policy network’s accuracy throughout training (the *opponent* policy network’s training showed a similar pattern).² On a held-out test set³, the *player* policy network achieves 50.6% accuracy, with the correct move being in the networks top ten predictions 95.3% of the time. The *opponent* policy network achieves 50.9% accuracy, with the correct move being in the network’s top ten predictions 96.0% of the time. The four-layer network with 192 filters per layer represented an optimal tradeoff between accuracy and complexity. Experiments with larger networks (up to 12 hidden layers and 512 filters) did not yield significantly higher accuracy and came at the expense of increased memory demands and computation time.

Simply using the policy net to suggest moves produces a player of nearly equivalent strength, achieving a $44.0\% \pm 3.14$ win-rate against WANDERER at 10 seconds/move (approximately 800,000 playouts).⁴ This is impressive given the fact that the policy net takes approximately 2 ms to generate a prediction.

² The training accuracy graph was generated from Tensorboard, TensorFlow’s data visualization package [25].

³ roughly 5% of the size of our training set, consisting entirely of moves from games excluded from the test set.

⁴ Unless otherwise noted, all games are out of 1000, 500 as White and 500 as Black, with 95% confidence intervals.



Fig. 2. Training accuracy over the number of training steps.

3.3 Integration into MCTS

In our CNN/MCTS paradigm, WAN-CNN utilizes a Synchronous Replace by Threshold [16, p. 20] expansions with $T = 0$ and a thread pool (minimal locking) for tree-level parallelism.⁵ Our implementation is basically UCT, with some differences in how moves are chosen, the expansion policy, and the rollout policy.

Selection. For move selection, we use the PUCT variant described in [10] where we choose action a at state s where $a = \operatorname{argmax}_{a'} \left(\frac{w_{a'}}{1+v_{a'}} + c_{puct} P(a'|s) \frac{\sqrt{\sum_b w_b}}{1+v_{a'}} \right)$, with w_n and v_n corresponding to the total wins and visits passing through node n , $\sum_b w_b$ to the wins over all sibling node of n , and c_{puct} to the exploration constant.

Expansion and Pruning Policy. For a given iteration, the tree is searched as in normal MCTS. Once we find a node to expand, we evaluate it with the respective neural network (while this thread is waiting for results from the GPU, another search thread can continue its execution). Once the neural network predictions are returned, only children that meet specified threshold criteria are retained – all other nodes are taken out of consideration.⁶ This corresponds to the top two actions for the player, the top ten for the opponent.⁷ The neural net

⁵ Note: As WAN-CNN was implemented in Python, it does not utilize true parallelism (Python’s Global Interpreter Lock prevents multiple threads from running in the same process simultaneously). The advantage of multithreading is that threads waiting for I/O (viz. GPU) can yield the processor to other idle search threads.

⁶ These children are added back into consideration if the initial set of children are solved for a loss.

⁷ Parameters chosen based on relative performance.

output is then used to set the prior values on a node’s children, with probability of move selection corresponding to a child’s number of losses. All visits are initialized to 100.

Rollout Policy. WAN-CNN utilizes quasi-rollouts where, immediately after expansion, the highest-ranking child is marked for expansion. This continues at each subsequent level until a terminal state is reached and the resulting win/loss value is backpropagated up the tree.

3.4 Wan-CNN Performance

In games against WANDERER, WAN-CNN achieved a $63.4\% \pm 2.99$ win-rate, meaning WAN-CNN’s exceeds that of WANDERER’s, despite the poor execution performance that Python offers. This result was directly attributable to our move selection and expansion policies. Without these modifications, WAN-CNN yielded a disappointing $21.6\% \pm 4.31$ win-rate.

3.5 Limitations

As the policy network was trained on games played by WANDERER, it may be overfitting to WANDERER’s policy. In other words, its win-rate may be due to the fact that it was trained to emulate WANDERER’s playstyle. In the future, we plan to play it against a broader array of expert players to validate our findings.

Additionally, as the maximum search time of our tests was 10s, our results may not hold at higher search times.

4 Conclusions and Future Work

Using an unsupervised learning paradigm (TDL) and starting with no domain knowledge we were surprised that we could create a Breakthrough evaluation function that was on par with the mature, hand-tuned one in WANDERER, giving a 46.6% win rate when played against normal WANDERER.

Using a supervised learning paradigm (CNN), we trained four-layer CNNs to classify Breakthrough moves made by WANDERER. When these networks were used to directly dictate a policy, they rivaled WANDERER at search times of up to 10 seconds/move. This is impressive given the CNNs’ inference speed (2 ms). Integrating the neural network into a simple MCTS (no domain knowledge) yielded a program that exceeded the performance of normal WANDERER, achieving a $63.4\% \pm 2.99$ win-rate.

Going forward, to improve WAN-TDL we plan to try refinements on our game progress idea which creates patterns based on the state of the game. Other global properties such the left/right relative balances of the pieces between the two sides, piece counts on black vs. white squares, etc. could provide further benefit, keeping in mind the tradeoff between the number patterns and memory and training time constraints.

We also plan to generate a neural network to act as a value function approximation and compare against both our simplified evaluation function as well as WANDERER's full evaluation function.

References

1. Finnsson, H., Björnsson, Y.: Game-tree properties and MCTS performance. In: IJCAI Workshop on General Game Playing (GIGA11), AAAI (2011)
2. Isaac, A., Lorentz, R.: Using partial tablebases in breakthrough. In: Plaat, A., Kusters, W., van den Herik, J. (eds.) CG 2016. LNCS, vol. 10068, pp. 1–10. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-50935-8_1
3. Lorentz, R., Horey, T.: Programming breakthrough. In: van den Herik, H.J., Iida, H., Plaat, A. (eds.) CG 2013. LNCS, vol. 8427, pp. 49–59. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09165-5_5
4. Saffidine, A., Jouandeu, N., Cazenave, T.: Solving breakthrough with race patterns and job-level proof number search. In: van den Herik, H.J., Plaat, A. (eds.) ACG 2011. LNCS, vol. 7168, pp. 196–207. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31866-5_17
5. Skowronski, P., Björnsson, Y., Winands, M.H.M.: Automated discovery of search-extension features. In: van den Herik, H.J., Spronck, P. (eds.) ACG 2009. LNCS, vol. 6048, pp. 182–194. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12993-3_17
6. Little Golem: <http://www.littlegolem.net/jsp/index.jsp>. Accessed 10 May 2017
7. Tesauro, G.: Temporal difference learning and TD-Gammon. *Commun. ACM* **38**, 58–68 (1995)
8. Schraudolph, N.N., Dayan, P., Sejnowski, T.J.: Temporal difference learning of position evaluation in the game of Go. In: *Advances in Neural Information Processing Systems*, NIPS, pp. 817–824 (1994)
9. Oka, K., Matsuzaki, K.: Systematic selection of N -tuple networks for 2048. In: Plaat, A., Kusters, W., van den Herik, J. (eds.) CG 2016. LNCS, vol. 10068, pp. 81–92. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-50935-8_8
10. 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**(7587), 484–489 (2016)
11. Mozur, P.: Google's AlphaGo defeats Chinese Go master in win for A.I. *New York Times*, 23 May 2017. <https://www.nytimes.com/2017/05/23/business/google-deepmind-alphago-go-champion-defeat.html>
12. Samuel, A.L.: Some studies in machine learning using the game of checkers. *IBM J. Res. Dev.* **3**, 210–229 (1959)
13. Sutton, R.S., Barto, A.G.: *Introduction to Reinforcement Learning*, 1st edn. MIT Press, Cambridge (1998)
14. Sutton, R.S.: Learning to predict by the methods of temporal differences. *Mach. Learn.* **3**, 9–44 (1988)
15. Kocsis, L., Szepesvári, C.: Bandit based Monte-Carlo planning. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) ECML 2006. LNCS, vol. 4212, pp. 282–293. Springer, Heidelberg (2006). https://doi.org/10.1007/11871842_29

16. Graf, T., Platzner, M.: Using deep convolutional neural networks in Monte Carlo tree search. In: Plaata, A., Kusters, W., van den Herik, J. (eds.) CG 2016. LNCS, vol. 10068, pp. 11–21. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-50935-8_2
17. Guo, X., Singh, S., Lee, H., Lewis, R., Wang, X.: Deep learning for real-time Atari game play using offline Monte-Carlo tree search planning. In: Advances in Neural Information Processing Systems 27 (NIPS 2014), pp. 3338–3346. IEEE (2014)
18. Nair, V., Hinton, G.: Rectified linear units improve restricted Boltzmann machines. In: Proceedings of the 27th International Conference on Machine Learning, ICML-2010, IMLS, pp. 807–814 (2010)
19. He, K., Zhang, X., Ren, S., Sun, J.: Delving deep into rectifiers: surpassing human-level performance on ImageNet classification. CoRR abs/1502.0 (2015)
20. Karpathy, A.: Convolutional neural networks for visual recognition: weight initialization (website) (2016)
21. Dunne, R., Campbell, N.: On the pairing of the softmax activation and cross-entropy penalty functions and the derivation of the softmax activation function. In: Proceedings of the 8th Australian Conference on Neural Networks, Clayton, pp. 181–185 (1997)
22. Glorot, X., Bengio, Y.: Understanding the difficulty of training deep feedforward neural networks. In: Proceedings of the 13th International Conference on Artificial Intelligence and Statistics (AISTATS), JMLR, pp. 249–256 (2010)
23. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D.G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., Zheng, X.: TensorFlow: a system for large-scale machine learning. arXiv preprint [arXiv:1605.08695](https://arxiv.org/abs/1605.08695) (2016)
24. Kingma, D.P., Ba, J.L.: Adam: a method for stochastic optimization. In: International Conference on Learning Representations, ICLR, pp. 1–15 (2015)
25. Google: TensorBoard: visualizing learning (2017). https://www.tensorflow.org/get-started/summaries_and_tensorboard