Training an artificial neural network to play tic-tac-toe

Sebastian Siegel

ECE 539 Term Project

12/20/2001

This PDF file contains hyperlinks.

Contents

Figures

1 Introduction

1.1 Problem Statement

In this project I will train an artificial neural network (ANN) to play tic-tac-toe (see 1.3 for further details on the game). The learning method involved will be reinforcement learning.

1.2 Importance of the topic

"The ability to play games is important for management information systems. Input will be a board position[1]. Number of input neurons is $\log_2$ of the number of different positions to be considered. Output is a move, suitably encoded [DeWilde95, p.48-49]."

"Reinforcement learning is learning what to do – how to map situations to actions – so as to maximize a numerical reward signal. The learner is not told which actions to take, as in most forms of machine learning, but instead must discover which actions yield the most reward by trying them [Sutton98, p. 3-4]." Tic-tac-toe is an illustrative application of reinforcement learning.

1.3 Tic-Tac-Toe

Usually, tic-tac-toe is played on a three-by-three grid (see figure 1). Each player in turn moves by placing a marker on an open square. One player's marker is "X" and the other's is "O". The game is over as soon as one player has three markers in a row: horizontally, vertically, or diagonally (an example is shown in figure 2). The game can also end as a draw (see figure 3), if there is no chance of winning for either player.

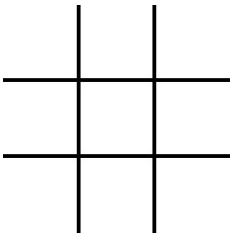


figure 1: Plain grid

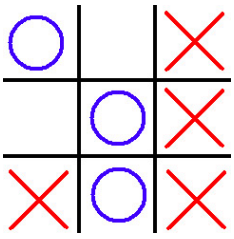


figure 2: X's won

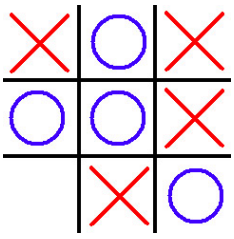


figure 3: Draw

[1] This method was suggested for the game of backgammon. It is quoted here because tic-tac-toe shares many properties.

2 Reinforcement learning

2.1 Method

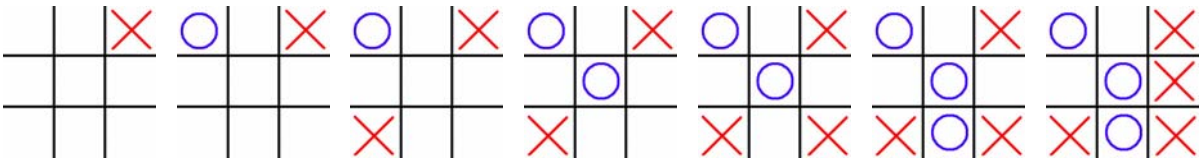
Supposed the exact probability of winning from a particular state is know for all possible situations that one player can encounter after it made a move, it could always make a perfect decision (by playing in such a way that any the next state has the highest probability out of all possible next states). But how can we get these probabilities? As mentioned in 1.2, the learning process is based upon the reward gained after every game. So, we already know the probability of winning for every final state. It will either be zero for losing or one for winning[2].

After every game, all intermediate states can be updated for each player. Note that these states are "afterstates". This means that for each player, only the states that result from a just completed move matter. As an example, figure 2 would represent an afterstate for the player marking X's while figure 3 may be afterstate of both players, depending upon who began.

Initially, all states are assigned a probability of winning of 0.5. The updating procedure after every games replays the game backward starting from the final state. Every previous afterstate's winning probability will be moved a little bit closer to the one of the current afterstate using the following formula [Sutton98, p. 12]:

$$P(s) \leftarrow P(s) + \alpha\left[P(s') - P(s)\right]$$

s' … current afterstate, s … previous afterstate
α ... learning rate (small positive fraction)



P(1a)=0.5625 P(1b)=0.125 P(2a)=0.625 P(2b)=0.25 P(3a)=0.75 P(3b)=0 P(4a)=1

figure 4: Updating afterstates

[2] I intended to train a player that will never lose. So I also rewarded any final state that represented a draw with +1, since it is also desirable.

In figure 4, an example of a possible game is shown. The final state for player A (plays X's) is the rightmost. Since player A won, that state will be rewarded with P(4a)=1. The previous afterstate would be state 3a. According to the updating rule,

$P(3a) \leftarrow P(3a) + \alpha\,[P(4a)-P(3a)]=0.5+0.5[1-0.5]=0.75$. Note that I assigned $\alpha=0.5$ and that P(3a) was initially 0.5. All other P-values are computed in a similar way. A lookup table[3] contains all the states and their corresponding P-values.

If player A once again goes first, all afterstates for an empty board would have a chance of winning of 0.5 except for state 1a from figure 4. Therefore, its first move would be similar. Again, all afterstates for player B in this case are 0.5 (were never met yet) except for state 1b (see figure 4). Therefore, player B would make a different choice this time.

Playing always the "best move" according to the P-values will not completely cover all possible states. During the process of learning, exploratory moves become necessary. This means, that each player makes a random move every once in a while. According to [Sutton98, p. 11 ff.] the updating has to be skipped for random moves. This means, that an afterstate that was the result of a random move will not be used to update the previous afterstate. However, I decided to still update completely, if the player managed to win or played a draw, even if random moves were involved for it was most likely a good "decision".

2.2 Convergence of the P-values

The method described in 2.1 is known to converge [Sutton98, p. 13] if α decreases within the course of training. I initially set $\alpha=0.5$ and decreased it by a constant value after every game such that after the last game it would be zero. Note, that in this case the total number of games for the training need to be known initially. Another approach would be the multiplication with a positive constant that is less than one.

The convergence of the P-values is shown in figure 5. There, 20000 games between two lookup tables took place. On average, 15 out of 100 moves were randomly chosen. Every ten games, the P-values for three example states were recorded. All three states are afterstates for the player that plays X's. So, state two is not desirable because of the fact that the opponent can easily win with its next move. That is why the P-value dropped down to 0 from its initial value of 0.5 (after approximately 4000 games). State 1 is highly desirable and has therefore a P-value that converges to almost one. The P-

[3] see 3 for more details

value of state three converges to 0.8. Yet this state is not desirable at all. Because the opponent (playing O's) can easily win from that state as well. However, this afterstate is only one of six possible afterstates that player A (playing X's) can chose from. All other five afterstates had higher P-values. This means that player A will always avoid example state 3 in figure 5, unless it makes random decisions.

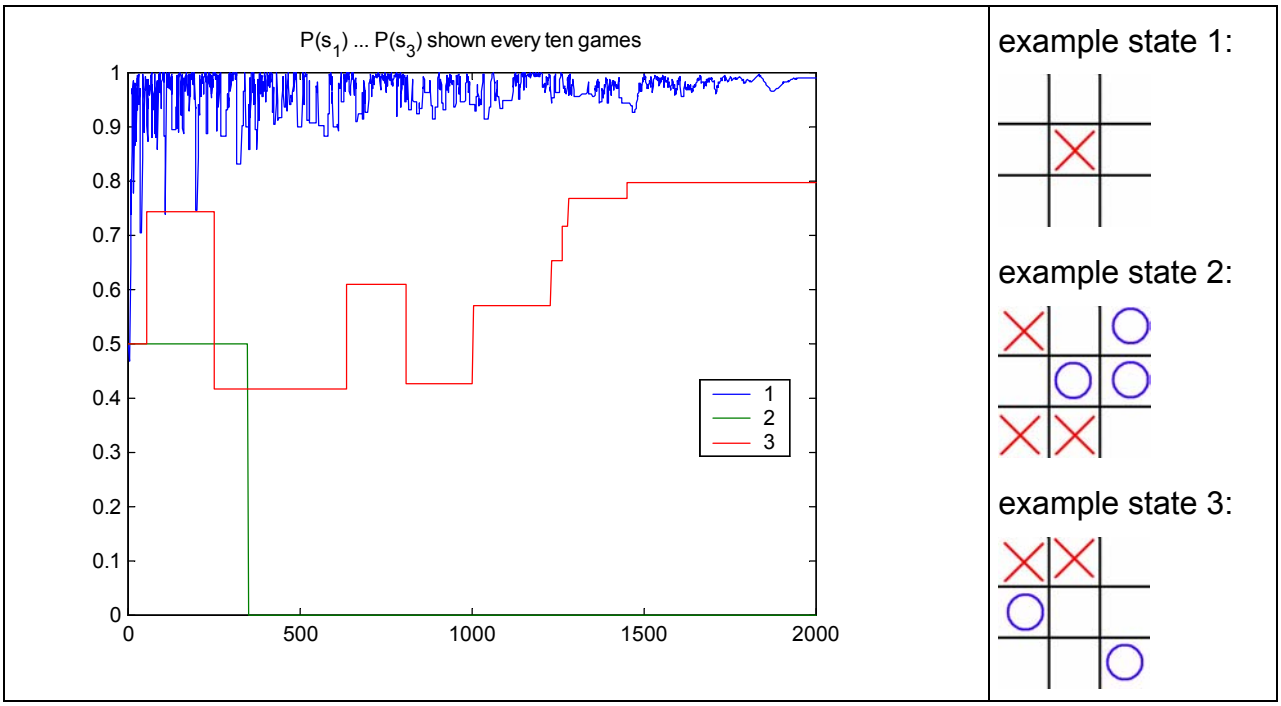


figure 5: Convergence of P-values for three sample states (observed every 10 games)

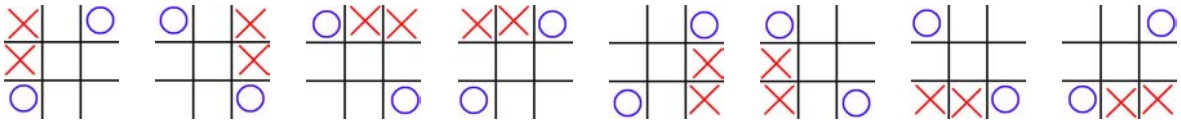
3 Generating the training data for the ANN

3.1 Setting up a lookup table

One approach is, to generate all possible afterstates for each player and initialize their corresponding P-values with 0.5. I decided to start with an empty lookup table. Since I knew that any state that the lookup table does not contain yet would have a P-value of 0.5, I just assumed that and set up the lookup table by adding new states met (or updated values that the lookup table contained already).

I simulated[4] all possible afterstates for one player and figured, that there is a total of 5890 for a three-by-three board. Many entries will be somewhat similar due to symmetry. In order to exploit the symmetry, a unique mapping is necessary that maps all (up to eight) states to one and only one. For the lookup table approach, this is not really necessary. But there are two main advantages involved: Since there are less states to learn, the algorithm converges much faster. Also, the ANN (to be trained) needs to learn less states and corresponding targets. This will either raise the classification rate or allow a smaller ANN.

There are as many as eight symmetric states that are considered to be equal (regarding the P-value). For any state, all the other symmetric ones can be found by turning the board 90 degrees clockwise (or counterclockwise). For each of those 4 states exists another one (flip either horizontally or vertically), as shown in figure 6.



S(X):	0.5625	0.1406	0.375	0.75	0.0176	0.0703	0.1172	0.0059
S(O):	0.1328	0.5020	0.5020	0.1328	0.1328	0.5020	0.5020	0.1328

figure 6: symmetrical states and their corresponding S(X) and S(O)

Now, the question is: Which of the eight symmetric states in figure 6 shall be the unique one? For this reason I introduce a function S(X) and S(O) that will assign all X's a real number S(X) and all O's a real number S(O), such that the state can be reproduced from

[4] findafterstates.m

these two numbers. The encoding is done as follows. Assign numbers 1, 2, 3, … on the board[5] (see figure 7):

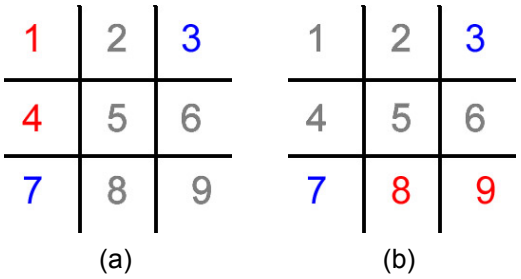


figure 7: Assigning numbers 1, 2, 3, … on the board

Furthermore, form a sum S(X) such that $S(X)=\sum_i 0.5^i I_X(i)$, i=1..9 for the three-by-three tic-tac-toe. $I_X(i)$=1 if there is an X on the i^{th} square on the board and zero otherwise. Similarly, $S(O)=\sum_i 0.5^i I_O(i)$, where $I_O(i)$=1 if there is an O on the i^{th} square on the board and zero otherwise. In figure 7, all red numbers correspond to X's on the board and all blue numbers correspond to O's respectively. Situation (a) represents the first state shown in figure 6 and situation (b) represents the rightmost one. As an example, I will compute S(X) for both states just mentioned:

$S_{(a)}(X)=0.5^1+0.5^4=0.5625$ and $S_{(b)}(X)=0.5^8+0.5^9=0.005859375$.

The mapping always maps to the state that has a minimum S(X). If there are more than one S(X) that are a minimum and equal, then the state with a minimum S(O) out of these would be chosen. So in figure 6, all eight states would be mapped to the rightmost one for it has a minimum S(X). It turns out[6], that there are only 825 (from 5890 originally) afterstates left for a three-by-three tic-tac-toe game.

The reason I chose 0.5^i in the previous sum is this: Most computer programs store real numbers as single or double precision floating point numbers (see IEEE 754: Floating Point). This format consists of a sign bit, some bits for the exponent and many bits for the mantissa. The sum S(X) or S(O) can be perfectly stored in the mantissa. The

[5] Start at the upper left corner and go right until the end of the row. Move on to the next row and start from the leftmost position. Go right again…

[6] findafterstates.m computes all possible afterstates for either player

number of bits reserved for the mantissa equals the maximum dimension[7] of the board, that can be handled in this way. A single precision floating point format (4 bytes: 23 precision bits) could perfectly store all X's or all O's on a board as large as 4 by 5. For the double precision floating format (8 bytes: 52 precision bits), this method works for boards as large as 7 by 7 (among other combinations).

I implemented a function[8] that would return S(X) and S(O) for any state. These two values and the corresponding P-value of that state were stored[9] in a lookup table. Each player had its own lookup table.

In order to obtain a lookup table that would have entries such that it could not lose any more, I implemented a program[10] that started out with two virtual players and two empty lookup tables (one for each). Both players competed for 20000 games. They each learned within the progress of the playing. However, they both did not encounter all possible afterstates. They only learned 711 and 707 out of 825 possible afterstates.

Now, the question was, are they already perfect? For this reason I (player B) played against the lookup table (player A). I was not able to win. But humans fail and therefore I decided to let A play against itself[11] by giving player B the same lookup table. Of course, all games would end as a draw. For this reason I let player B make some random (20 %) moves. If player A would lose once, it means that it does not play perfect. The result[12] after 10,000 games was this: player A won 2728 games, 7272 draws, player B did not win a single game. So I decided that from now on, I had a perfect playing lookup table[13].

3.2 Converting the lookup table to training data

Once a lookup table with converged P-values (probability of winning from that state) is available, its S(X) and S(O) could be expanded to a matrix representing the board, such that all X's are +1, all O's are –1 and all empty squares are 0. This could be the input to an ANN. The output of the ANN would be the corresponding P-value. It turns out, that it

---

[7] dimension in this case means number of squares on the board (e.g. three-by-three tic-tac-toe: dimension=3 rows x 3 columns =9 positions)

[8] framevalue.m, works for any board dimension (also of rectangular shape: only 4 symmetric states) according to the floating point number format

[9] setframeprob.m (getframeprob.m must be run in advance, see function for details)

[10] generatelookuptable.m

[11] test.m

[12] logfile: test1.log

[13] lookuptablesmat (holds a variable "lookuptablepos" and "lookuptableneg"), here I only consider "lookuptablepos"

is extremely difficult to train such an ANN to a degree that the output (corresponding P-value) is computed precise enough, such that the state of all possible afterstates with a maximum P-value according to the lookup table matches the state that has a maximum P-value according to the ANN. Furthermore, this implementation has another disadvantage: For a given state in the game, all possible afterstates would have to be figured out and applied to the ANN. All the P-values would have to be stored and the maximum needs to be determined. After that, the desirable move would be available.

In order to improve the training data and make it easier to actually train an ANN, I decided to use a different strategy: implement an ANN for "position-move" pairs (as suggested by the first quote in 1.2). This means that the input of the ANN would once again be a state in the game, but not an afterstate. It would be the state before a move was made ("beforestate"). The output neurons that has the highest value would represent the position for the best move. Therefore, the ANN would have as many output neurons as the board has squares (for the three-by-three tic-tac-toe it would be nine output neurons). Now, we have a pattern classification problem which in this case promises better results.

In order to convert the lookup table to the training data that would be used for training the latter ANN, all beforestates[14] (not afterstates) have to be computed. Note that there are less beforestates (609) than there are afterstates (825). This has to do with the fact that all the final states in the game (someone won, lost, or game ended as a draw) are no longer necessary. This is another advantage of the second approach (regarding the kind of ANN) mentioned earlier.

The classification rates I achieved did not satisfy me at first. Instead of increasing the size of the ANN, I thought about further pruning the training data. Supposed the opponent does not miss a chance to win whenever it has the opportunity to complete a row on its turn, I do not really need to train all "position-move" pairs that can result in such a situation for the opponent. This way of pruning the training data means that I actually evaluate future moves of the opponent. One could do a further evaluation, but that would pretty much mean an implementation of the rules of the game, which was not my intension. However, by doing the one step evaluation, I was able to reduce the size of the training data to 537 entries[15] (was 609 before) without losing a perfect player[16].

---

[14] part of lookuptable2trainingdata.m

[15] that is the output of lookuptable2trainingdata.m

[16] test(trainingdata,2) will "prove" that

4 Introducing an artificial neural network

4.1 Training a Multilayer Perceptron (MLP)

Once the training data is available, an MLP can be trained. The input data will be the beforestates[17]. The output neuron with the highest value will represent the next suggested move by its position (see 3.2 for further details). This is a classification problem. For this reason I wrote a function[18] (based upon "bp.m[19]") that trains an MLP using the back propagation algorithm[20]. The input was the training data and several parameters regarding the process of training the MLP. The output was the best weights obtained.

I tried different sizes of MLPs several times (see figure 8) and varied parameters such as the learning rate α, the momentum μ and others. The activation function for each hidden layer was the hyperbolic tangent and for the output neurons it was the sigmoidal function. It turned out that the best results could be obtained with the following parameters (besides those mentioned in figure 8):

α=0.1, μ=0, epoch size=64, convergence test every 8 epochs,

hidden layers	neurons within hidden layers	classification rate
1	9	80.26 %
1	27	93.30 %
2	9 – 9	88.27 %
2	9 – 27	93.11 %
2	27 – 27	95.72 %
3	27 – 27 – 27	98.51 %
4	9 – 27 – 81 – 27	94.41 %

figure 8: best classification rates obtained for different MLP configurations

[17] input values: squares with an X: +1, O: -1, empty square: 0

[18] mlpbp.m

[19] Implementation of backpropagation algorithm © copyright 2001 by Yu Hen Hu

[20] using class notes and [Haykin99]

It turns out that it is most likely impossible to achieve a classification rate of 100% with a reasonable sized MLP. Also, large MLPs tend to get stuck at a certain classification rate without improving during the process of training which makes it even harder to reduce misclassification.

In order to still achieve perfect playing, I simply computed[21] a new lookup table that contained all those "position-move" pairs that the MLP would misclassify. The function[22] that I implemented to compute the ANN output for a given beforestate will automatically use that new lookup table to make a decision.

4.2 Comparison: MLP vs. lookup table

In this chapter I define the lookup table as a table that contains all the "position-move" pairs (this was the training data before). So the MLP (expanded by a new lookup table to deal with misclassifications) and the lookup table would have the same input and the same output.

I compared both in terms of speed and memory. For this reason I used my test[23] function that would play a large number of games. When it tested the lookup table (537 "position-move" pairs), it took 262 seconds to play 1000 games while the MLP[24] finished the same task in 108 seconds.

For the memory comparison I assume that I need 24 bytes for each entry in the lookup table ([S(X) S(O) target], 8 bytes each). So, the lookup table has a size of 12,888 bytes. The MLP consists of 27 hidden neurons and 9 output neurons. That makes a total of 27x10+9x28=522 weights (including bias terms). They will need 522x8 bytes= 4176 bytes. Furthermore I needed a new lookup table with 36 entries. That adds an additional 36x24 bytes= 864 bytes.

The following figure 9 will summarize the just mentioned comparison:

---

[21] errortable.m

[22] comp_mlp(board matrix, player (-1 or +1), random play (0=no, 1=yes), weights, lookup table (optional))

[23] test.m, first introduced in 3.1

[24] 1 hidden layer with 27 neurons, „expanded" by a new lookup table (36 entries) to achieve perfect classification

	lookup table	MLP	
speed (1000 games)	262 sec.	108 sec.	(saves 59% computing time)
memory (bytes)	12,888	5,040	(saves 61% memory)

figure 9: comparing a lookup table and an MLP in terms of speed and memory

The comparison in terms of memory usage is based upon double precision floating point numbers. With access to the bits of a variable, one could easily prune the size of the lookup table to 22 bit per entry (18 bits for the current board state and 4 bits for the target). Using Huffman encoding, this could even be further reduced because the board contains only three different marks (and empty squares appear with a chance of 41.3% on the board regarding the 537 entries). This would result in a bit stream of 1,996x1 bits+2,837x2 bits=7,670 bits for all 537 board states. Allowing 4 bits for the target (could also be reduced by Huffman encoding), the lookup table could be stored within a bit stream of 9,818 bits. However, implementing such techniques in Matlab is not advisable. The major disadvantage would be time consuming calculations.

4.3 Tic-Tac-Toe on a larger board

My original intension was, to train an artificial neural network to play tic-tac-toe on a larger board (e.g. four-by-five, four in a row win). But there is an immediate problem involved: The lookup table becomes tremendously large (millions of entries even after symmetry pruning). For this reason, a different approach would have to be considered. The MLP could now be used to learn the P-value of a given state immediately without storing it in a lookup table. Training such an artificial neural network requires a different approach[25].

As there would be millions of states to be evaluated, the P-values will only converge after millions of games. A simulation of such a process with Matlab will take way too long these days. Therefore, I could not try this approach.

[25] see [Sutton98] chapter 8.2, TD-learning, Gradient-Descent Methods

5 Discussion

In this project, I used the method of reinforcement learning to learn how to play a game: tic-tac-toe. It is rather interesting that without implementing any rules[26] or strategies, it was possible to learn a perfect strategy from the reward after each game only. For this reason, a lookup table was introduced that would contain the information gained from the previous games (afterstates and corresponding P-values).

Training an MLP with the information gained from the lookup table was possible with a high classification rate (see figure 8). Transferring the lookup table to an MLP may not yield perfect recall. In those cases, a combination of a strongly reduced lookup table (contains all entries that would be misclassified by the MLP) and an MLP can be applied.

Especially for large lookup tables (e.g. thousands of entries), an MLP can give a much faster answer. It may also reduce the required memory for storing all entries. All MLPs had nine input neurons. According to [DeWilde95][27], the number of input neurons should be $\log_2$ of the number of different positions to be considered. I trained an ANN with 537 "position-move" pairs. So, nine input neurons are a reasonable approach.

Further investigation possible:

1. Train an ANN with a teaching algorithm such that the ANN learns all tricky "position-move" pairs only.
2. Try to teach an ANN those "position-move" pairs encountered in playing against a human and train a method of defense (e.g. always move where the human went next in a previous game). This may actually pick up the strategy the human follows (if there was one).

[26] The only things known were whose turn it was and when the game was over.

[27] see first quote in 1.2

6 Fun facts

- One can actually play against the MLP (extended by the lookup table to take care of misclassification) by running “play.m”.
- All the steps[28] mentioned that are necessary to train the MLP are performed by starting “batch.m”. This process will take two to three minutes. At the end, you can play against the MLP (“play.m” will be called). Note that the lookup table with converged P-values will be loaded. It was generated by “generatelookuptable.m” (takes about two to three hours).
- Most functions are general such that they would work on a larger board (e.g. 4x5 as well).
- I implemented a function winnertest.m that tests if the game is over and for this reason introduced a matrix mc[29] that contains all possible chains of squares that need to be tested:

2	2	2	1	3	3	3
2	2	2	1	3	3	3
5	5	5	5	0	0	0
5	5	5	5	0	0	0
5	5	5	5	0	0	0

Example of mc for a 5x7 board and 4 in a row win

2	4	3
5	0	0
5	0	0

Example of mc for a 3x3 board and 3 in a row win

Whereas the number on each position would indicate in what directions the “four in a row test” or “three in a row test” would have to be considered:

1: ↙↓↘→, 2: ↓↘→, 3: ↙↓, 4: ↓, 5: →

[28] convert lookuptable with converged P-values to training data, train an MLP, generate a new lookup table for those “position-move” pairs that the MLP would misclassify

[29] generated by init_mc.m (given dr=# of rows, dc=# of columns, dwin=# of marks that indicate a winner)

7 References

Sutton98	Sutton, Richard S., and Barto, Andrew G. (1998). *Reinforcement Learning: An Introduction, "*A Bradford Book", MIT Press online: http://www-anw.cs.umass.edu/~rich/book/the-book.html
Haykin99	Haykin, Simon (1999). *Neural networks: a comprehensive foundation,* 2nd edition, Prentice Hall, New Jersey
DeWilde95	De Wilde, Philippe (1995). *Neural network models: an analysis,* Springer-Verlag