# Learning Richman Games on Neural Networks Using Temporal Difference Learning

**Honors Thesis**

**by**

**Gil Carmel**

# 1. Introduction

The purpose of this project was to quantitatively analyze the performance of *temporal-difference learning*, a training algorithm for neural networks. The idea was to choose a non-trivial game for which an optimal strategy is known (I chose Richman games), then analyze the learning of the neural network using the optimal strategy as a barometer of performance. While I never had time to train the network correctly and so never achieved any encouraging training results, I have learned about and integrated several different research areas from recent research publications.

The project consisted of the following primary tasks:

- Learn about Richman games
- Learn about neural networks in general (temporal difference learning and back-propagation in particular)
- Devise a scheme to integrate the above two into a game-learning neural network
- Write software to implement these concepts

The following sections describe my work pertaining to each of these tasks. Section 2 provides an overview of Richman games. Section 3 explains the concepts I acquired in the field of neural networks. Section 4 describes my attempts to integrate the two into a game-learning neural network.

# 2. Richman Games

## 2.1 Description

The game Tic-Tac-Dough – which belongs to the class of *Richman games* for which Lazarus, et. al. have devised an optimal strategy [1] -- was chosen as the non-trivial game which the neural network was to learn. Richman games are two-player contests in which the players, instead of alternating turns, each bid a sum of money for each turn, with the higher bidder winning the move and paying his bid to the other player. Tic-Tac-Dough, then, is very similar to Tic-Tac-Toe, except that, again, the players bid for each turn instead of alternating. Thus before making the bid, the player has to weigh the importance of the current move versus the importance of conserving money for later turns. The game can be regarded as a finite directed graph (representing the possible game positions and the transitions between them), a token (representing the current game position), and two distinguished vertices on the graph (say *blue* and *red*, where one player's goal is to reach *blue* and the other's to reach *red*). The player that wins the bid moves the token along a directed edge of the graph from the current position to the next position. Lazarus et. al. prove that for each vertex on the graph there a critical ratio (the *Richman cost*) such that if a player's share of the money is greater than the Richman cost, then the player has a winning strategy from this vertex. For more information, see [1].

## 2.2 Building a Perfect Player

### 2.2.1 Generating the Game-State Graph

#### 2.2.1.1 The basic algorithm

Calculating the Richman costs and the winning strategy is a fairly simple recursive process once the game-state graph is generated. Therefore, my main challenge in implementing a perfect Tic-Tac-Dough player was to build this graph. The basic algorithm is as follows:

→ Generate two nodes, *blue* and *red*, each representing *all* winning boards for Blue and Red, respectively

→ Add a node, *start*, representing the starting board to a queue, *q*.

→ while *q* not empty

  → *curr* = dequeue(*q*)

  → for each successor board *succboard* (representing the result of a possible move *m* from the board represented by *curr*)

    → if *succboard* is a blue (red) win, connect *curr* to *blue* (*red*) with an edge representing move *m*

    → else if a node *succ* representing *succboard* already exists[1], connect *curr* to *succ* with an edge representing move *m*

    → else create a new node *succ* representing board *succboard*, then connect *curr* to *succ* with an edge representing move *m*, and enqueue *succ* in *q*

The task of discovering if a node *succ* than represents *succboard* already exists adds some complication. Searching the whole graph would be inefficient. Instead, I created a simple routine that picks, from among all the predecessors of a board, a *standard predecessor*. When a new node is created, the standard predecessor of its board is found in the graph[2], and an edge representing the appropriate move is added between the standard predecessor node and the new node. Thus, when we want to search for a node representing a certain board, all we have to do is recursively search for standard predecessors.

#### 2.2.1.2 Reflections and Rotations of the Same Board

I tried to make this graph as memory efficient as possible. I decided to represent all rotations and reflections of the same board by the same node in the graph (since the strategy is obviously the same from each these boards). To do this, all boards need to be stored in their *canonical form*. In addition, edges now represented an orientation change from one board to the next in addition to the move.

The canonical form of a board is defined as follows. Boards are represented as two integers in my program. The canonical form of a board is simply the board that has the smallest two integers (lexicographically) out of all the possible transformations of the board. The above algorithm still works if we bring each board to its canonical form before we process it and take special care to associate the appropriate orientation change to each edge.

#### 2.2.1.3 Tie-Breaking

A game like Tic-Tac-Dough can end with a tie (i.e. no player gets three in a row). We can arbitrarily define a tie-breaker that has to do with the characteristics of the board (e.g. the player with the most pieces wins), or we can break the tie according to who has the most money.

In the former case, we just treat a tie like any other win according to who wins the tie breaker: If Blue (Red) wins the tie-breaker, than we connect the predecessor to the node *blue* (*red*).

But what if we want the player with the most money to win? To what node should the predecessor of the tie be connected? To solve this problem we use the following trick: create a special node, *most_money*, that is attached

---

[1] This can happen when a single board, *b*, can be reached from two different boards *p* and *q*. If *p* is dequeued before *q* then a node representing *b* will already exist by the time *q* is dequeued (because it will have been generated from *p*).

[2] The queue ensures that the standard predecessor has already been created, since by the time we start adding boards with *x* pieces to the graph, all the boards with (*x-1*) pieces are already in the graph

as a predecessor to *blue* and to *red*. Attach each tied board's predecessor to *most_money*. If the token ever reaches *most_money*, it is obvious that the player with the most money has a winning strategy: bid all her money and move to the appropriate final node. Thus the calculation of the Richman costs will be correct.

#### 2.2.1.4 Consecutive Boards

We may wish to break the first *n* ties by starting a new game rather than resorting to a tie-breaker. To accomplish that, we run the basic algorithm *n+1* times. The first iteration builds the lowest game-graph (the one whose ties are broken as before). The next iteration builds the next lowest game-graph. Tied board's predecessors in this graph are attached to the lowest graph's *start* node, etc. Note that in each of these iterations, *blue*, *red*, and *money_tie*, remain the same (they are not generated anew for each game-graph). When we run the algorithm for calculating Richman costs, we start at the very top and let the recursion go all the way to the very bottom.

### 2.2.2 Calculating the Winning Strategy from the Game-State Graph

#### 2.2.2.1 The Basic Strategy

The winning strategy of Richman games is closely tied to a function defined on the vertices of the graph, V.

Lazarus et. al. prove that every game-state graph has a unique function (the *Richman cost function*) R: V→[0,1] such that R(*blue*)=0, R(*red*)=1, and for every other vertex *v*, $R(v) = (R^+(v) + R^-(v)) / 2$ (where $R^+(v)$ ($R^-(v)$) is the Richman cost of the successor node of *v* with the highest (lowest) Richman cost) [1]. In other words, the Richman cost of *v* is the average of the lowest and highest Richman costs among its successor nodes.

If the graph is acyclic (as is the case in Tic-Tac-Dough), we can easily calculate the Richman costs for each node using a recursive function that starts at the *start* node.

Once we have the Richman cost for each node, we can figure out the winning strategy. By the way we assigned Richman costs to *blue* and *red*, we know Blue has a winning strategy from these nodes if she has more than R(*blue*) and R(*red*), respectively. Now we go inductively up the tree (assuming that from each node there is a path to at least one of the winning vertices, and that the graph is acyclic[3]). Suppose Blue is on a node *n* and her share of the money is greater than R*(n)*. Blue then bids $\frac{R^+(n) - R^-(n)}{2}$. If she wins the bid, she moves to the successor node *v* whose Richman cost is $R^-(n)$. In this case, she has more than

$R(n) - \frac{R^+(n) - R^-(n)}{2} = \frac{R^+(n) + R^-(n)}{2} - \frac{R^+(n) - R^-(n)}{2} = R^-(n) = R(v)$ left, which is enough to win from node *v*.

Similarly, if she loses the bid, the worst her opponent can do is go to the node *u* with the largest Richman cost (i.e. $R^+(n)$). But if she loses the bid, she has more than $R^+(n) = R(u)$, which is enough to win from node *u*.

#### 2.2.2.2 A Player Who Likes to Win Pretty

The strategy as we have described it does not distinguish between winning legitimately or by a tie-breaker. To make a player that prefers to win pretty, we need to generate two game-state graphs. The first, T, has tie-breakers as described before. The second, L, considers all ties as Red wins. The Richman costs for Blue on L are obviously higher (it takes more money to guaranty a win rather than a non-loss). The computer, playing as Blue, advances the token along both graphs as the game is played, but the minute its share of the money exceeds the Richman cost of the current node on L, it starts playing according to the strategy as dictated by the Richman costs on L.

---

[3] For other cases, include cyclic digraphs, infinite digraphs, etc., see [1] and [2]

# 3. Neural Networks

I tried to use a combination of two techniques to train a *feed-forward* network: *back-propagation* and *temporal-difference learning*. The following is a brief overview of feed-forward neural nets, followed by a description of each of the two learning methods.

## *3.1 Feed-Forward Neural Networks*

Feed-forward networks are composed of several layers of units (see Figure 1). The *input layer* outputs some form of encoded data representing the current state of our system (in our case the game-state – the Tic-Tac-Dough board and the money each player has).

Each unit $u$ on a non-input layer receives as input the output of all units on the previous layer. $u$ also stores a weight associated with each connection to a unit on the previous level. To produce $O(u)$, the output of $u$, we apply an *activation function*, $f(x)$, to the weighted sum $h$ of $u$'s inputs (each input is multiplied by the weight of its connection):

$O(u) = f\left(\sum_i x_i w_i\right)$, where $x_i$ is the input from the $i^{th}$ unit on the previous level, and $w_i$ is the weight of the connection between the two units.

Our "learning algorithms" are simply ways of changing each connection's weight so that the output of the output units matches what we want it to be. For example, suppose we want the network to output, given a certain Tic-Tac-Dough game state, the expected outcome of that game, e.g. a number between 0, representing a loss, and 1, representing a win (in this case the network would have only one unit in its output layer). A good network would have weights such that the output for a favorable game state is high and the output for an undesirable game state is low.

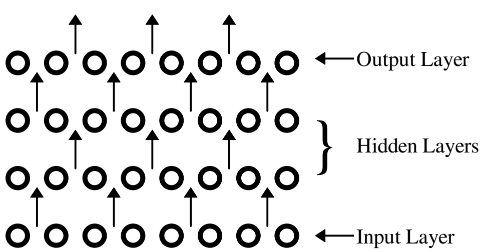


**Figure 1 - Feed Forward Network**

## *3.2 Back-Propagation Learning*

The idea, then, is to change each connection's weight so that it minimizes the difference between the desired output and the actual output of the network. Back-Propagation is a way to determine what these weight changes should be, even for a multilayer network. We will develop this technique for a three-layer network (input, output, and one hidden layer). It can easily be generalized to more layers. We introduce the following terminology (the derivation is based on [3]):

$S_i$ - actual output of the $i^{th}$ output unit

$\zeta_i$ - desired output of $i^{th}$ output unit

$s_j$ - output of the $j^{th}$ hidden unit

$\sigma_k$ - $k^{th}$ input (i.e. output of the $k^{th}$ input unit)

$w_{ij}$ - weight of connection between $i^{th}$ output unit and $j^{th}$ hidden unit

$\overline{w}_{jk}$ - weight of connection between j[th] hidden unit and k[th] input unit

$f$ - activation function

$h_i$ - weighted sum of inputs to i[th] output unit: $h_i = \sum_j w_{ij} s_j$

$\overline{h}_j$ - weighted sum of inputs to j[th] hidden unit: $\overline{h}_j = \sum_k \overline{w}_{jk} \sigma_k$

Tying these together, we have:

$S_i = f(h_i)$, and $s_i = f(\overline{h}_j)$

Our goal is to minimize the differences between each $S_i$ and $\zeta_i$ pair. Thus, we define our error as follows:

$$D = \frac{1}{2}\sum_i [\zeta_i - S_i]^2 = \frac{1}{2}\sum_i [\zeta_i - f(h_i)]^2$$ (the ½ coefficient eases later calculations).

If we expand the expression for D using the above definitions, we see that it depends on $w_{ij}$ and $\overline{w}_{jk}$ (i.e. each connection weight affects D). We can regard D as a surface over a (*JK*+*IJ*)-dimensional coordinate space (where *I*, *J*, and *K* are the number of output, hidden, and input units, respectively) for which we want to find a minimum – we wish to find the point in the coordinate space (i.e. the combination of connection weights) where the error is smallest. To do this, we update the weights along the negative of the gradient of D with respect to the weights. Thus the change to the weights of connections between hidden unit *j* and output unit *i* is as follows:

$$\delta w_{ij} = -\frac{\partial D}{\partial w_{ij}} = [\zeta_i - f(h_i)] f'(h_i) \frac{\partial h_i}{\delta w_{ij}} = [\zeta_i - f(h_i)] f'(h_i) s_j = \Delta_i s_j,$$

with the shorthand notation $\Delta_i = [\zeta_i - f(h_i)] f'(h_i)$. The intuitive explanation of the above formula is as follows: note that $\Delta_i$ is simply the negative of the partial derivative of *D* (the error) with respect to $h_i$ (the weighted sum of the inputs to the i[th] output unit). It signifies how we would modify $h_i$ in order to reduce *D*. Since we cannot modify $h_i$ directly, we modify it indirectly by changing the connection weights into unit *i* according to the formula. The formula states that if the input $s_j$ has the same (opposite) sign as $\Delta_i$, we need to increase (decrease) $w_{ij}$ in order to achieve the desired effect on $h_i$.

Now we extend the calculation to the hidden units. The change to the weights of connections between the input and hidden layer is as follows:

$$\delta \overline{w}_{jk} = -\frac{\partial D}{\partial \overline{w}_{jk}} = \sum_i [\zeta_i - f(h_i)] f'(h_i) \frac{\partial h_i}{\partial s_j} \frac{\partial s_j}{\partial \overline{w}_{jk}} = \sum_i \Delta_i w_{ij} f'(\overline{h}_j) \frac{\partial \overline{h}_j}{\partial \overline{w}_{jk}}$$

$$= \sum_i \Delta_i w_{ij} f'(\overline{h}_j) \sigma_k = \overline{\Delta}_j \sigma_k$$

where $\overline{\Delta}_j = \left(\sum_i \Delta_i w_{ij}\right) f'(\overline{h}_j)$. The intuitive interpretation of $\overline{\Delta}_j$ is analogous to the one given above for $\Delta_i$ – it signifies our desired change to $\overline{h}_j$ in order to reduce the overall error, because it is the negative of the partial derivative of *D* with respect to $\overline{h}_j$. Here, since we are dealing with hidden units, this "desired change" is proportional to the sum of the "desired changes" of each of the output units (i.e. $\Delta_i$), with each term in the sum weighted according to the weight of the connection between the hidden unit *j* and the output unit *i*.

Since the weight changes $\delta \overline{w}_{jk}$ depend on the values of $\Delta_i$, we first calculate the weight changes $\delta w_{ij}$, then **propagate** the $\Delta_i$ values backward to the previous level, so that they can be used for the calculation of weight changes there (hence the name back-propagation).

For more information on back-propagation, see [3].

## *3.3 Temporal-Difference Learning*

Temporal-difference learning deals with the problem of predicting the outcome of a series of events. In the case of Tic-Tac-Dough, the "series of events" is a game progression $\sigma^1, \sigma^2, \sigma^3, \ldots \sigma^m$ (where each $\sigma^\mu$ is an input vector representing the current game state – a board position and the cash total for each player), and the outcome $\zeta$ is a number (1 for a win, 0 for a loss). From each game-state $\sigma^\mu$, as we progress through the series, we want to make a prediction $S^\mu$, a number between zero and one, of the expected outcome. The prediction, which is the output of the network's sole output unit, is based on some function of the input vector $\sigma^\mu$ and a weight vector $w$ (this function is the *extremely* long expression that represents the complete operation of the network, top to bottom). Learning is accomplished by changing $w$ in order to produce more accurate predictions; after a complete series and an outcome are observed, we change the weights by the sum of the weight changes produced at each time step $t$, called $\delta w^t$:

$$w \leftarrow w + \sum_{t=1}^{m} \delta w^t$$

Traditional supervised-learning algorithms are based on the error between each prediction and the final outcome $\zeta$. Temporal-difference learning, on the other hand, is based on the error between successive predictions, as well as the final outcome. For example, suppose we want to predict, at each day of the week, the closing price for a certain stock on Friday. Supervised-learning would change each day's prediction based on the difference between that day's prediction and the actual closing price on Friday. Temporal-difference learning would change the prediction for each day based on the next day's prediction (the actual outcome is treated as $S^{m+1}$, so that $S^m$ is changed according to the actual outcome).

The following explanation of temporal-difference learning is based on Sutton's paper [4], which provided the first formalization of temporal-difference methods. More information about the intuition of temporal-difference learning, as well as proofs of convergence and efficiency for certain cases, can be found in that paper.

### 3.3.1 Supervised-Learning

First we formally define the weight update rule for supervised-learning. Again we regard the error $D$ as a surface over an $n$-dimensional coordinate space (where $n$ is the number of weights in the vector $w$ – the total number of weights in the network). Supervised-learning pairs each day's prediction with the final outcome, and tries to reduce the difference between them. Since we want to minimize $D = \frac{1}{2}(\zeta - S^t)^2$, the update-rule needs to change the weight vector in the direction opposite the gradient of $D$ with respect to $w$:

$$\delta w^t = -\nabla_w D = \left(\zeta - S^t\right)\nabla_w S^t$$

Note that this is the same update-rule used in back-propagation, only in vector form (instead of explicitly for each weight). Also, note that in order to calculate the weight changes, we need to know the final outcome $\zeta$, so we are forced to wait until the end of the series in order to calculate all the weight changes.

### 3.3.2 Incremental Supervised-Learning

Now we present an instance of temporal-difference learning that produces the same results as supervised-learning, but calculates the weight changes incrementally.

The key is to notice that the overall error ( $\zeta - S^t$ ) can be represented as a sum:

$\zeta - S^t = \sum_{\mu=t}^{m} S^{\mu+1} - S^{\mu}$ , where we define $S^{m+1} = \zeta$ . Now we rewrite our end-of-series update rule:

$$
\begin{aligned}
w \leftarrow w + \sum_{t=1}^{m} \delta w^t = w + \sum_{t=1}^{m} \left(\zeta - S^t\right) \nabla_w S^t \\
= w + \sum_{t=1}^{m} \sum_{\mu=t}^{m} \left(S^{\mu+1} - S^{\mu}\right) \nabla_w S^t \\
= w + \sum_{\mu=1}^{m} \sum_{t=1}^{\mu} \left(S^{\mu+1} - S^{\mu}\right) \nabla_w S^t \\
= w + \sum_{t=1}^{m} \left(S^{t+1} - S^t\right) \sum_{\mu=1}^{t} \nabla_w S^{\mu} ,
\end{aligned}
$$

so our new definition of "$\delta w^t$" is $\delta w^t = \left(S^{t+1} - S^t\right) \sum_{\mu=1}^{t} \nabla_w S^{\mu}$ . This is an incremental update-rule that produces the same results as the supervised-learning update-rule. We can calculate the weight updates at the end of each event in the series because the rule depends only on the next prediction and the sum of the gradients up to that event.

### 3.3.3 General TD(λ)

The above update-rule is a specific instance of a more general class of update-rules, the TD(λ) family. Here, $\delta w^t = \left(S^{t+1} - S^t\right) \sum_{\mu=1}^{t} \lambda^{t-\mu} \nabla_w S^{\mu}$ . The incremental supervised-learning update-rule, where the update depends solely on the difference between the current prediction and the actual outcome, is simply TD(1). In TD(0), the update-rule depends solely on the difference between the current prediction and the next prediction (plug it in and see). In between, when $0<\lambda<1$, we have a smooth interpolation between the two. Tesauro has achieved good results using TD-learning to train a backgammon player [4].

### 3.3.4 Combining TD-Learning and Back-Propagation

TD and back-propagation learning are easily combined, since TD(λ) is geared toward *temporal* credit-assignment, while back-propagation solves the problem of *structural* credit assignment. TD(λ) gives a general update-rule for the weight vector at the end of each event in the series. This update-rule is presented in terms of the gradient of the prediction $S^{\mu}$ with respect to the weight vector. Back-propagation simply presents an efficient way to calculate this gradient – a non-trivial task due to the hidden units in the network. The combined update-rule looks like this:

For connections between hidden units and the output unit[4]:

[4] We now combine the notations from both discussions; subscripts are the same as defined for back-propagation and serve to topographically identify units in the network, superscripts (with the exception of the exponent of λ) are the same as defined for TD(λ) and serve to temporally identify the inputs and predictions of the network. Note that we now have only one output unit (the one predicting the outcome), so the *i* subscript will always be 1 in these formulas.

$$\delta w_{1j}^{t} = \left(S_1^{t+1} - S_1^{t}\right)\sum_{\mu=1}^{t}\lambda^{t-\mu}\frac{\partial S_1^{\mu}}{\partial w_{1j}} = \left(S_1^{t+1} - S_1^{t}\right)\sum_{\mu=1}^{t}\lambda^{t-\mu} f'\left(h_1^{\mu}\right)\frac{\partial h_1^{\mu}}{\delta w_{1j}} = \left(S_1^{t+1} - S_1^{t}\right)\sum_{\mu=1}^{t}\lambda^{t-\mu} f'\left(h_1^{\mu}\right) s_j^{\mu}$$

For connections between input units and hidden units:

$$\delta \overline{w}_{jk}^{t} = \left(S_1^{t+1} - S_1^{t}\right)\sum_{\mu=1}^{t}\lambda^{t-\mu}\frac{\partial S_1^{\mu}}{\partial \overline{w}_{jk}} = \left(S_1^{t+1} - S_1^{t}\right)\sum_{\mu=1}^{t}\lambda^{t-\mu} f'\left(h_1^{\mu}\right)\frac{\partial h_1^{\mu}}{\partial s_j^{\mu}}\frac{\partial s_j^{\mu}}{\partial \overline{w}_{jk}}$$

$$= \left(S_1^{t+1} - S_1^{t}\right)\sum_{\mu=1}^{t}\lambda^{t-\mu} f'\left(h_1^{\mu}\right) w_{ij} f'\left(\overline{h}_j^{\mu}\right)\frac{\partial \overline{h}_j^{\mu}}{\partial \overline{w}_{jk}} = \left(S_1^{t+1} - S_1^{t}\right)\sum_{\mu=1}^{t}\lambda^{t-\mu} f'\left(h_1^{\mu}\right) w_{ij} f'\left(\overline{h}_j^{\mu}\right)\sigma_k^{\mu}$$

# 4. Learning Richman Games on Neural Networks

Unfortunately, although a lot of the groundwork is already there, I never programmed a correct version of a TD($\lambda$) learner. Instead, I programmed convoluted versions of it, based on my convoluted understanding of the subject of neural nets until I had to sit down and explain it in this paper. These mentally-challenged learners never really amounted to anything, but they were enough to make me consider several implementation issues and how to analyze performance.

## *4.1 Implementation Issues*

### 4.1.1 Purpose of the Network

The explicit purpose of the network is to function as a game-state evaluator; given a game-state as input (see section 4.1.2), the network is to output a real number between 0 and 1 that provides an estimate of the expected outcome of a game in that state, where values close to 1 (0) represent a high probability of winning (losing). The network's output can be used to select moves and bids, as explained later, but it is important to realize that the network is trained as a game-state evaluator only, not as a game-player. The network is trained through game observation (i.e. a series of game-states plus an outcome – 0 for a loss and 1 for a win), and the weights are updated as described in section 3.3.3.

### 4.1.2 Board Encoding

One important consideration is how to convert the game-state (board and money) into a numerical input vector on which the network can operate.

First, we need to consider the trade-off between having a large number or a small number of input units. Take as an example how we encode the amount of money a player has. Let's compare the following two options: use a single input unit, valued 0 to 1, to indicate the percentage of the total money held by the player; or have 101 input units, with units 1-100 being binary units with the appropriate one set to indicate the integer portion of the player's percentage of the total money, and unit 101 being a real number between 0 and 1 and representing the fraction portion of the player's percentage of the money (e.g. if the player has 23.5 percent of the money, then unit 23 would be 1, unit 101 would be 0.5, and the rest would be 0). In the first case we risk over-encoding, i.e. making the input space so cryptic that the network would have difficulty distinguishing between states that are significantly different (having 49 percent of the money vs. having 51 percent of the money when both players are one move away from a win). In the second case we risk making the input space so large and complex that learning would be hard to accomplish and would take an extremely long time (because we would have to cover many of the different input states during training in order not to leave any training "blind spots", and because this would make it harder for the network to see the similarities between similar game-states, thus decreasing its ability to "generalize").

Second, we need to consider how much help we give the network. It has been shown that transforming the board into an input vector using several "hand-crafted features" based on certain intermediate goals of the game (such as the number of two-in-a-rows held, for example) can help the network learn [4]. But it seems more intellectually appealing to let the network learn "from scratch", without this additional help, using a straightforward transformation from game-state to input vector. However, since there are a myriad of possibilities even for such "straightforward" transformations, our biases in selecting this transformation inevitably play a role in the network's learning. A good guideline is to choose a transformation that (1) simply encodes all of the information in the game-state without adding any non-trivial features, and (2) still affords the network the opportunity to learn given the network structure and training algorithm. A example of a transformation that satisfies (1) but does not satisfy (2) is the one discussed above that uses one real-valued unit to represent the money held.

### 4.1.3 Values of $\lambda$

What value of $\lambda$ should we use? Values close to 1 (i.e. close to supervised-learning) might provide faster initial learning, since at the beginning of training weights are random, so using the next prediction as a barometer of performance would work very slowly. However, as the network becomes more trained, using a large $\lambda$ could result in bad training: if we match the start state, which has an expected outcome of .5, with the actual outcome (0 or 1), each game would cause the prediction of the start state to vary, causing somewhat of an instability. Matching the start state with the prediction from the next state would cause less instability in the predictions. Tesauro did not find a bid difference in learning between different values of $\lambda$.

I should mention that in the discussion of both back-propagation and TD-learning, I omitted, for simplicity's sake, the learning rate parameter, the scalar $\alpha$. It is used simply to regulate how far along the gradient we want to go; the rate-update vector is multiplied by $\alpha$ to change its length. Thus if $\alpha$ is large, we may overshoot the minimum, but if it is small, training may take longer since we approach the minimum in very small steps. Again a sensible solution might be to use a large $\alpha$ at the beginning, then reduce it throughout training to reduce the step size (but what is large? What is small? There's probably a whole lot of discussion of this subject out there.)

### 4.1.4 Activation Function

I have not studied the different possible choices of activation functions for the network. There is some discussion of this topic in [3]. I chose the sigmoid function, $f(x) = \frac{1}{1+e^{-x}}$, simply because it was the one used successfully by Tesauro.

### 4.1.5 Move Selection

First we discuss how to select a move once a bid is already won. Like Tesauro, I used the network's output to select the next move – of all the available moves, we pick the one whose resultant game-state, when fed to the network as a test-input, produces the highest expected outcome[5].

### 4.1.6 Bid Selection

Next we discuss bid selection. The strategy is to choose the bid which would result in the best worst-case scenario (i.e. the worst-case scenario with the highest network output). We pick a bid examination resolution (to be discussed in Section 4.1.8), and for every possible bid from 0 up to the amount of money held we check what is the best that could happen if we win the bid and what is the worst that could happen if we lose the bid:

---

[5] Tesauro notes that this is not necessarily the best move selector. The function we are trying to minimize through training is the difference between the actual expected outcome and the network's estimate of the expected outcome. The only thing we care about for move selection, on the other hand, is the relative difference between the predictions for the expected outcome from each state. This means that for move selection purposes, we do not care if the predictions are completely off-target, as long as they are all off by the same amount, while for training purposes we want the estimates overall to vary as little as possible from the actual expected outcomes.

→ *best_worst_case* := -1

→ for each possible *bid*

  → pick the best move to make if bid is won as discussed in Section 4.1.5 (test-input boards will be the results of making each move on the current board, and test-input cash will be the current cash minus the bid); *max_winbid* := output of network for resultant game-state

  → pick best move for opponent to make if bid is lost (here test-input cash will be the current cash plus the bid); *min_losebid* := output of network for resultant game-state

  → if min(*max_winbid, min_losebid*) > *best_worst_case*

    → *best_bid* = *bid*

    → *best_worst_case* = min(*max_winbid, min_losebid*)

→ return *best_bid*

### 4.1.7 Winning Pretty

It is a rather complex task to teach the network to prefer to win outright and not by virtue of a tie-breaker. Tesauro was faced with a similar problem in his TD-gammon program, and came up with a solution I do not completely understand. In backgammon, you can either win (worth 1 point) or gammon (worth 2 points). Tesauro uses 4 output units and a 4-component outcome signal (I am guessing one binary component for each possible outcome) to train the network. Each output is regarded as the probability of its corresponding outcome. For move selection purposes, he uses these probabilities to calculate expected payoff according to the following equation: $-2p_1 - p_2 + 2p_3 + p_4$ (where $p_1$, $p_2$, $p_3$, $p_4$, are the probabilities of a gammon loss, a loss, a gammon win, and a win, respectively), then chooses the move that produces the highest expected payoff. The part I do not understand is how to generalize TD($\lambda$) to several output units. The update-rule is based on the partial derivative of **the** network output with respect to each weight. Here we have four outputs. It is clear which output should be differentiated to determine weight changes for connection weights between hidden units and output units – simply the output produced by the unit at the end of the connection. But which output is to be differentiated for connections between input and hidden units? Each such connection weight affects *all* the outputs.

If we knew the error function that TD($\lambda$) tries to minimize, we could perhaps generalize this error to multiple output units. To find this error, all we need to do is integrate the TD($\lambda$) update-rule. Let me know when you're done.

My suggested solution (admittedly heuristic) is to pick two numbers, $x_1$ and $x_2$, $0<x_1<x_2<1$, and set the outcome signals for an outright loss, a tie-break loss, a tie-break win, and an outright win to 0, $x_1$, $x_2$, and 1, respectively. It would be interesting to see if the network learns to evaluate this function correctly, and also what effect the two values chosen for $x_1$ and $x_2$ would have on learning.

Fortunately, there is no need to delve too deeply into this issue for now. The perfect player that does not care how it wins is just as effective a standard to compare the network against as the player that prefers to win pretty.

### 4.1.8 Bid Examination Resolution

The reason Richman games were chosen is that they offer a perfect strategy to compare the network's learning against. Therefore, we must not handicap the network's learning by imposing some arbitrary bid examination resolution and hindering it from making the correct bids to begin with (e.g. if the bid examination resolution is 1, while the perfect strategy for a certain game-state calls for a bid of 32.8125, there is no way the network could play the game perfectly).

Unfortunately, even for the simple game of tic-tac-dough, the Richman costs of the graph node lie on a very fine scale – they are multiples of $2^{-9}$ between 0 and 1 (9 is the maximum number of steps to either the *red* or *blue* node). Therefore, the bids lie on an even finer scale: multiples of $2^{-10}$ between 0 and 1 times the total amount of money[6]. Training time obviously scales linearly with the number of bids examined. I am sure it is possible to achieve reasonable results even without such fine resolution, but if that approach is taken we lose the advantage of being able to see how closely the neural network approaches the perfect player.

### 4.1.9 Training Opponent

There are four obvious possibilities for training opponents for the network:

1. A human player (let me know when you're done)
2. Itself
3. A different network
4. A perfect player

#### 4.1.9.1 A Network that Plays a Human Player

The longest reported lifetime of a human being is approximately 119 years. The holder of this record might have begun to put a dent in the training. Unfortunately, he is also dead. Since we cannot know in advance a specific individual's lifetime, it would seem futile to invest the time in just anyone who might die tomorrow, for all we know. In this author's opinion, research should be directed toward teaching giant tortoises, whose expected lifetime is around 400 years, to play tic-tac-dough.

#### 4.1.9.2 A Network that Plays Itself

First we examine the possibilities for implementing a network that plays itself. We assume that game-states are transformed into input vectors as follows: 9 binary bits each indicating whether the proponent has a piece in a board position, 9 binary bits each indicating whether the opponent has a piece in a board position, some kind of representation of the proponent's money (see Section 4.1.2), and the same representation for the opponent's money. As I see it, there are two possibilities for picking moves this way. The first is to select proponent moves with the highest expected outcome and opponent moves with the lowest expected outcome. The second is to select proponent moves the same way, but when it is time to select the opponent's move, reverse the definitions of "proponent" and "opponent" when converting the game-state to an input vector, and pick the move with the highest expected outcome.

The problem with the first approach is that the network may get stuck in a local minimum, where some of the games are won, some are lost, but any small change to the weights reduces the fraction of games won, propelling the weight vector back to the local minimum. This may also happen in the second approach, but it seems less likely, because we are, in effect, training two "separate" networks at the same time – every game produces two series & output pairs, one for each player. Although input vectors where opponent/proponent are reversed are obviously strategically identical, a random network will most likely not choose the same bid and the same move location for two "reversed" versions of the same game-state – as far as the network is concerned, these are two completely different game-states. Here, unless "both" players get stuck in a local minimum, they push each other to learn in a fashion similar to having two separate networks (see Section 4.1.9.3).

One advantage of this second approach over having two separate networks, is that the players can learn from each other's experiences. If a game-state is reached that is approximately the reverse of a previously trained-on game-state, we will have the advantage of having already trained the network on this portion of the input space.

---

[6] It is possible that a somewhat coarser resolution is actually necessary, but I have not given too much thought to this. I know for a fact that the start node's bid is an odd multiple of $2^{-7}$, so the resolution needs to be at least that.

#### 4.1.9.3 A Network that Plays Another Network

In pitting a network against another network, we would start with two different random networks, and let them both train at the same time. As was mentioned before, the likelihood of two networks getting stuck in a local minimum is probably lower than that of a single network. If one network gets stuck, the other will probably continue to improve until it is winning almost every game (at which point it has no more reason to improve – the reward signal is always 1). However, once the first network is losing all its games (i.e. the reward signal is always 0), the weights will probably slowly drift back towards randomness, and eventually find another minimum. In this way, the two networks can "push" each other. This is the optimist's view. The pessimist's view goes like this: In the same way the inferior network starts to drift toward randomness when the reward signal stays the same no matter what it does, the good network starts to drift as well. This may lead to perpetual mediocrity, since now instead of getting better, the inferior network just waits for the superior network to "play down to the competition". In order to avoid this phenomenon, we need to somehow ensure that one network never gets dramatically better than the other (perhaps an elimination tournament among many networks, a kind of genetic algorithm). This issue leads us to…

#### 4.1.9.4 A Network that Plays a Perfect Player

For the above reasons, playing a perfect player would lead to completely random weight updates, since the reward signal will almost always be 0 no matter how close the network gets to the perfect player. It could happen that the network will eventually stumble upon an appropriate weight vector, but this is somewhat unlikely to happen in any reasonable amount of time. It is interesting to note this difference between a human learner and the network learner – a human could definitely learn by playing a perfect player. One of the reasons for this may be that, as humans, we learn to define intermediate goals for ourselves when we play a game (getting two in a row, for example). All the network has to go by is the final reward signal.

### 4.1.10 Number of Hidden Units

The network is, in fact, a function evaluator; we want it to evaluate the expected outcome from each game-state. The parameters the network has to play with in order to produce a good evaluation are the connection weights. Obviously, there is some lower limit to the number of connection weights such that a network with fewer weights simply could not store enough information to evaluate the function correctly. Since the number of weights is determined by the number of input units and hidden units, it is important to choose a good number for each. As was mentioned before, having too many input units would increase the size of the input space, and could thus leave training "blind spots". Adding hidden units adds more weights while not affecting the size of the input space. Tesauro found that increasing the number of hidden units in his network increased performance. However, it should be mentioned that the training time increases linearly with the number of hidden units (each units adds a set number of connections whose weights need to be updated).

As a side note, it seems to me that it should be possible to come up with a rough estimate of how many connection weights the network needs in order to have the ability to "capture" all the information contained in the game-state graph for tic-tac-dough. Also, it would be interesting to see whether the network could do with fewer weights by generalizing rather than in effect storing all of the game-state graph information in its weights.

## *4.2 Analysis Issues*

There are a few interesting points to mention regarding analysis of the network's performance:

- Throughout the training of the network, it is possible to pause training every $x$ training games, go through a sample set of game-states, compare the bids and moves generated by the network to the perfect bids and moves, and come up with some sort of metric for network performance. In this fashion one could plot a learning curve for the network, and quantitatively analyze the effects of the implementation decisions discussed above (values of $\lambda$ and $\alpha$, game-state$\rightarrow$input-vector transformation, number of hidden units, etc.).
- It will be interesting to see whether the network learns to generalize certain aspects of the game (for example, do rotations of the same board always produce similar outputs; do inverses of the same board always produce

the same bid; is the move selected the same regardless of the amount of money held; etc.), or whether it simply knows to repeat what it has already seen.

- In addition to looking at the network's move and bid selection, we should also look at the network output, which we hope will be the expected outcome from each board position. What we can expect this output to be, however, depends on who the network has played throughout its training. Suppose the network has somehow converged to a perfect player. If it played a perfect player throughout its training, then we might expect the network output to be random but greater than 0.5 for *winnable positions*, which are game-states in which the Richman cost for the board position is less than the fraction of the money held (random because playing a perfect player would never train the network on these positions and greater than 0.5 because we assumed that this is a perfect player); close to 0 for *unwinnable positions*, which are game-states in which the Richman cost for the board position is more than the fraction of the money held (because the perfect player would always win from those positions); and 0.5 for *tied positions*, game-states in which the two are equal (because tied bids are resolved by a coin toss). If, on the other hand, the network trained against a non-perfect player, then its outputs for winnable positions would all be close to 1 (because it would always win from these positions); and its output from unwinnable and tied positions would be greater than those of the previous network (because it still would win sometimes even from those positions);

### 4.3 Pitfalls

Temporal-difference learning is intended for a somewhat different application than the one suggested here. In Sutton's paper, the neural network simply observes a series of states and an outcome, and eventually learns to predict the expected outcome from each of the states in this series. In effect, the network learns to evaluate a function that maps a state to an expected outcome. In our application, as in Tesauro's, the network's output is used to select moves, a practice that affects the observed series and outcome. As the network becomes a better evaluator of game-states, the network selects better moves, and the expected outcome changes. The network is no longer just converging to a certain function, because the function it is supposed to evaluate is constantly changing. This effect may be amplified if the network's *opponent* is also learning, or it may be neutralized if both players improve at about the same rate while acquiring roughly the same "skills" (in this case the players' improvements could cancel each other out, maintaining the expected outcomes at somewhat constant levels). It should be mentioned that Tesauro achieved good results despite this (perhaps because the effect is not so significant for the purposes of move selection).

In order to apply temporal-difference learning strictly, we would need to have the network observe two expert, unchanging players, so that it may acquire the true "expected outcome" function. Because of the nature of Richman games, however, this is an impractical approach in our case.

## 5. The Software

I wrote some pretty neat and object-oriented software to help implement all these concepts. I will be glad to document it in more detail in case anyone wants to play with it.

## References


[1] A.J Lazarus, D.E. Loeb, J.G. Propp, D. Ullman, *Richman Games*, Proceedings of the 1994 MSRI Workshop on Combinatorial Game Theory. See *http://www-math.mit.edu/~propp/articles.html*.

[2] A.J Lazarus, D.E. Loeb, J.G. Propp, D. Ullman, *Combinatorial Games under Auction Play*, submitted to *Games and Economic Behavior*. See *http://www-math.mit.edu/~propp/articles.html*.

[3] B. Muller, J. Reinhardt, M.T. Strickland, *Neural Networks: An Introduction,* second edition, Springer-Verlag, Berlin (1995).

[4] G. Tesauro, “Practical Issues in Temporal Difference Learning”, *Machine Learning, 8,* p. 257-277, Kluwer Academic Publishers, Boston (1992).