PUNISH/REWARD: LEARNING WITH A CRITIC IN MULTILAYER NEURAL NETWORKS

GREGORY L. PLETT
Department of Electrical and Computer Engineering
University of Colorado at Colorado Springs
Colorado Springs, Colorado

ABSTRACT

One of the earliest machine-learning algorithms in the class we now refer to as "reinforcement learning" was the learning-with-a-critic method by Widrow, Gupta, and Maitra. In their work, a punish and reward scheme is used to train a single adaptive neuron. The novel feature of this method is that an error signal is not used as is required with supervised-learning algorithms; neither is the algorithm entirely unsupervised. Instead, an external observer called the critic evaluates the neuron's performance in a qualitative way and decides to either "reward" or "punish" its behavior. The neuron is "rewarded" if its performance is good, and is "punished" if its performance is bad.

In this paper, the punish/reward mechanism is generalized in order to train multilayer neural networks. An example is presented where a neural network is trained to play the childrens' game "tic-tac-toe."

INTRODUCTION

In the early 1960s, the LMS algorithm (Widrow and Hoff, 1960; Widrow and Stearns, 1985) was developed for training a single neuron. This is a supervised learning algorithm which requires a "teacher" to supply a *desired response* for every input training pattern. The difference between the neuron's output and the desired response is used to adapt the neuron's internal parameter values to cause learning. The backpropagation algorithm (Werbos, 1974; Rumelhart et al., 1986) was later developed for supervised learning with multilayer neural networks.

An alternative to supervised learning is unsupervised learning. An example is Kohonen's self-organizing feature map (Kohonen, 1982). Without a teacher, the neurons are only able to identify patterns in the data set and perform data clustering.

A final paradigm is the class of reinforcement learning algorithms (Haykin, 1994). Within this class, Widrow, Gupta, and Maitra (Widrow et al., 1973) invented the concept of learning-with-a-critic for training a single neuron. Critic learning does not require a precise desired response for each input training pattern. An external observer, called the critic, decides whether the neuron's decisions are good or bad, and *rewards* or *punishes* accordingly. Rewarded decisions are more likely to be repeated, and punished decisions are less likely to be repeated.

In this paper, the original critic algorithm is first discussed, and is then extended to be able to train a larger neural network.

ADAPTING A SINGLE NEURON WITH CRITIC LEARNING

We consider a neuron with a signum activation function (see Fig. 1(a)). The neuron computes $y = \text{sgn}(s) = \text{sgn}(W^T X)$, where y is the output of the neuron for input vector X,

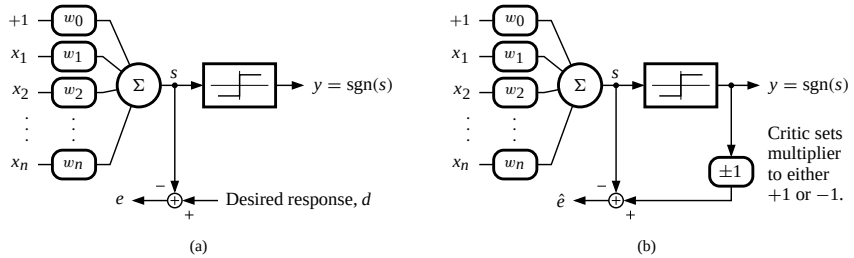

Figure 1. Single-neuron neural network: Adapted (a) using LMS; (b) using critic.

and W are the weight values of the neuron. (X is augmented with a bias value of 1, and W is likewise augmented with a weight to multiply the bias value.) When a desired response is available, an error signal may be computed and used with the LMS algorithm to update the neuron's weight values: $\Delta W = 2\mu e X$, where $e = d - s$ is the difference between the desired response and the neuron's internal sum, and μ is a small positive constant that controls the learning rate.

If no desired response is available, we cannot compute the error e and so cannot use LMS to adapt the weights. Critic learning estimates the desired response. The neuron's output is thresholded and interpreted to be a binary decision. An external observer called the critic judges the the result of the decision to be "good" or "bad."

If the critic decides "good," then the neuron's decision is assumed correct and the weight values are reinforced. This is done by estimating the desired response to be equal to the neuron's output: $\Delta W = 2\mu \hat{e} X$, where $\hat{e} = \text{sgn}(s) - s$. *If the critic decides "bad,"* then the neuron's decision is assumed to be incorrect and the weight values are punished. The desired response is estimated to be the negative of the neuron's output. The same weight update formula is used, but $\hat{e} = -\text{sgn}(s) - s$. Figure 1(b) shows a neuron with error $\hat{e}$. An internal desired response signal is computed with the aid of the external critic which sets the ± 1 switch.

Critic learning was devised for a single neuron, so has limited usefulness. Here, we extend the method so that it is able to train multilayered neural networks and therefore solve more difficult problems. In keeping with the original work by Widrow and colleagues, the resulting algorithm does not require a desired response—a simple reward/punish signal will adapt the entire network.

ADAPTING A NEURAL NETWORK WITH CRITIC LEARNING

A multi-layer neural network is shown in Fig. 2(a). Each neuron in the network computes a nonlinear function of a weighted sum of its inputs. The nonlinear function is usually chosen to be either the sigmoid or the $\tanh(\cdot)$ function. Layers of neurons are able to compute very general nonlinear functions, and the network may be trained (with a teacher) using the backpropagation algorithm.

To extend critic learning to train multi-layered neural networks, it is important to notice that Widrow and colleagues' algorithm can be divided into two parts. The first part estimates a desired response for the neuron based on whether the neuron is being rewarded or punished. The error $\hat{e}$ is then computed to be the estimated desired response minus the actual output. The second part of the algorithm uses LMS to adapt the neurons' weights using this error.

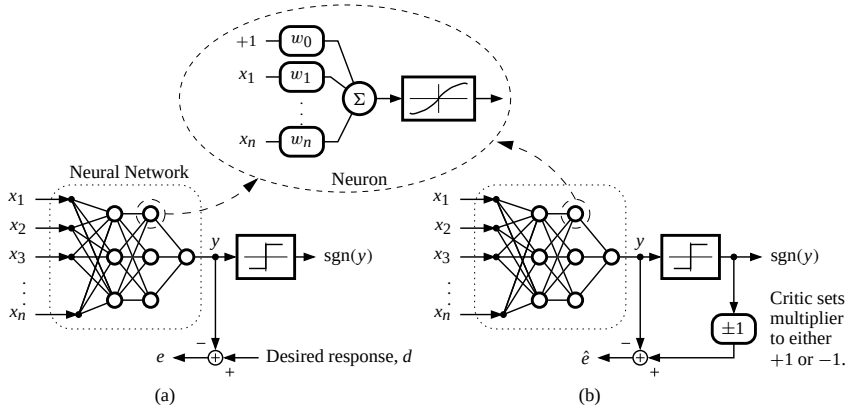

Figure 2. Multilayer neural network: Adapted (a) using backpropagation; (b) using critic.

From this point of view, it is easy to extend critic learning to more general neural-network structures. First, as before, an external observer called the critic supplies a reward/punish signal which is used to estimate a desired response. The network's error is estimated to be $\hat{e} = [\text{sgn}(y) - y]$ for rewarding or $\hat{e} = [-\text{sgn}(y) - y]$ for punishing. This error is used with the backpropagation algorithm to adapt the weights in the neural network. The resulting scheme is shown in Fig. 2(b).

When dealing with multi-output neural networks the reward/punish mechanism needs to be chosen to conform to the interpretation of the network outputs. If the outputs compete in a one-of-many scheme (where the output with the highest value is the selected output), then the error for all neurons in the output layer should be zero except for the neuron that wins the competition. That specific neuron should have its desired response set to either $+1$ or -1 by the critic. If the neurons in the output layer operate independently, then each should either be rewarded or punished based on individual performance.

One final scenario needs to be considered. It is possible that the critic is unable to judge whether the network's performance is good or bad. The performance may be "fair" or "tie." One possible method to generate a desired response when the critic sees no clear winner is described below.

APPLICATION OF CRITIC LEARNING TO PLAYING TIC-TAC-TOE

Problem Specification

Multiple-player games are problems which have an optimal strategy which may not be known a priori. Using a supervised-learning algorithm to train the network is impractical since a desired response cannot be efficiently generated at the end of each turn to function as a training signal. Knowledge of the system's performance is only available at the end of the game, after the players take a number of turns.

"Tic-tac-toe" is a simple two-player game which is played on a grid of three-by-three squares. At the start of the game, the grid is empty, and two players alternately place their mark in an empty position in the grid. One player uses an "$\times$" to mark his or her position, and the other player uses an "$\circ$." The first player to get three of his or her marks in a row (horizontally, vertically or diagonally) wins the game. If neither player has three adjacent marks when the grid is full, the game is considered to be tied.

Figure 3. Configuration of the network tic-tac-toe player.

By attempting a few games, the reader will become convinced that a skillful player will never lose. While the player who goes first has an advantage, two expert players will always play to a draw. We can anticipate that different applications will also have scenarios where it is difficult to determine whether the neural network did a good or a bad job. Learning how to handle a tie game of tic-tac-toe will give us insight into how to handle other "tie" situations.

The Linear Player

The network configuration used to play tic-tac-toe is shown in Fig. 3. The nine board squares are represented in the computer as a "board state vector" of nine elements. An element is 1, 0, or -1, when the corresponding board position is "$\times$", empty or "$\circ$", respectively. The board state vector is the network input. The output of the network consists of nine floating-point numbers. The index (into the output vector) of the maximum output is chosen to be the network's move.

Our first attempt to teach a network to play tic-tac-toe was to train a single-layer network of nine *linear* neurons—ones without sigmoidal activation functions. Each linear neuron computes a weighted sum of the input board state vector, plus an adaptable bias value: $y_i = W_i^T X$. The opponent's strategy is to automatically make a move by selecting at random one of the remaining available positions. One might envision a different scenario where two networks are trained to play against each other (or, where a single network is trained by playing against itself). We found that the network quickly learned a single winning or tying scenario which was always repeated. We wish to learn a more general strategy. This being the case, a perfectly-trained network competing against a random player will generally win (as opposed to tie), and will never loose. Either the network or the random player may make the first move of the game. Since the strategies for the odd-turn player and the even-turn player are different, two different networks were trained.

Notice that it is possible for the network to make an invalid move: the square it chooses for its move may already be marked. *If the network makes an invalid move*, then the game is terminated and that single invalid move is punished. *If the network makes no invalid moves and wins the game*, then each move made by the network is rewarded: each move of the game is replayed and the board configuration at the beginning of the turn is again used as input to the network. The output neuron corresponding to the move made by the network is rewarded, and all other neurons are unchanged. The amounts of the weight updates for all neurons in the network are computed (using backpropagation) and stored. This procedure is repeated for all of the moves that the network made in the game. Finally, the weights of the network are updated by the amounts calculated in each of the individual sub-steps. *If the network makes no invalid moves and looses the game*, all of the above steps are performed, with the exception that the output neurons corresponding to the moves made by the network are punished. In general, the adaptation constant μ is may be one value when rewarding, and a different value when punishing. Since punish events happen either when the network looses a game or when it makes an invalid move, and reward events happen only when the network wins a game, punish events tend to occur more frequently, especially at the

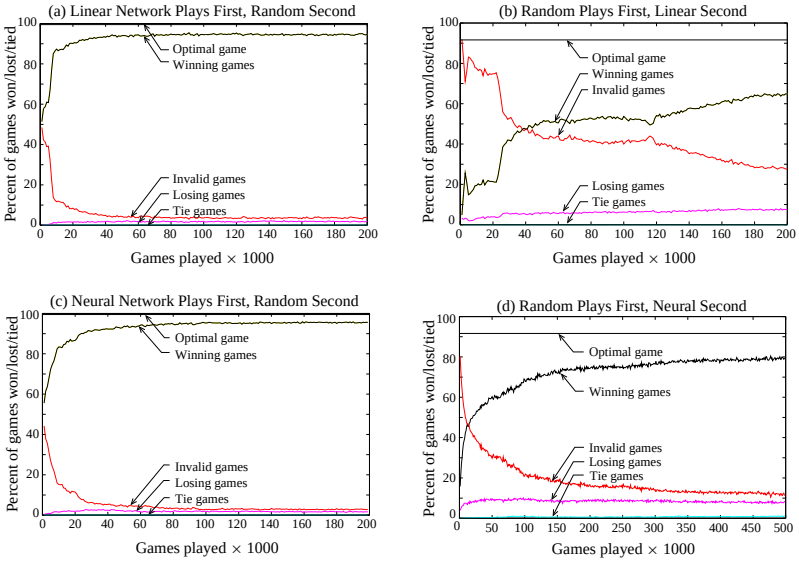

Figure 4. Learning curves: (a) and (b) Linear network; (c) and (d) Neural network.

beginning of training. To compensate, we found that the reward constant should be slightly larger than the punish constant. All that remains is to determine what should be done if the game results in a tie. Because the network played against a random player, we found that tie games were very rare. They were neither rewarded nor punished.

Learning curves resulting from simulations are shown in Figs. 4(a) and (b). Each curve portrays an ensemble average of 10 simulation runs, plotted against games played, where the initial network weights were different in each run. The solid horizontal line in each plot corresponds to the theoretical maximum average performance (too close to 100% in Fig. 4(a) to be clearly seen). In Fig. 4(a), the network was given the advantage of taking the first move of the game. In Fig. 4(b), the network played second. The curves are labeled and represent the ensemble-average percentages of games won, invalid, lost and tied.

When the network played first, it performed better than if it played second, as would be expected. When playing first, it wins about 94% of its games with the random player, forfeits about 4% due to invalid moves, loses about 2% and never ties; when playing second, it wins about 64% of its games, forfeits about 28%; loses about 8% and never ties. To give these results some meaning, simulations were done with two random players playing against each other. The random first-turn player won 58% of the games, the random second-turn player won 29%, and the players tied 13%. Clearly, the network is doing *much* better than random.

Playing against a random opponent, the optimal strategy wins 99.5% of its games and ties the rest, if it takes the *first* turn. If it takes the *second* turn, the optimal strategy wins 91.6% of its games, and to ties the rest.

The Neural-Network Player

While the linear system just described performed well, we expect that a multi-layer neural network can do better. The neural-network trained to play tic-tac-toe had three fully-

connected processing layers of neurons. The first layer had 18 neurons; the second had 12; and the third had nine.

Learning curves for the neural network trained by critic are shown in Figs. 4(c) and (d). When the network played first, it won about 95% of its games, forfeited 3% due to invalid moves, lost 2% and never tied; when it played second, it won 80%, forfeited 11%, lost 8% and tied 1%. Furthermore, the best neural network (out of the ten trained) came *very* close to learning the optimal game.

CONCLUSIONS

This work extends the original learning-with-a-critic methods so that critic learning may now be applied to train neural networks instead of only single neurons. Critic learning can be used to train the weights of a neural network when no teacher exists to provide a desired response. The algorithm was tested by training a neural network to play tic-tac-toe against an opponent who always chose a random available square. Using the proposed algorithm, the network was observed to learn a good strategy, and was able to play a near-perfect game.

NOMENCLATURE

W	Neuron weight vector.
ΔW	Change in weight vector due to adaptation.
X	Input vector to neuron.
e	Network or neuron output error.
$\hat{e}$	Network or neuron output error, estimated by critic.
s	Neuron internal sum (before nonlinearity).
y	Neuron or network output.
μ	Small positive learning constant.

ACKNOWLEDGMENTS

This work was supported in part by funding from the National Science Foundation under contract ECS–9522085, and the Electric Power Research Institute under contract WO8016–17. Thanks to Dr. Raymond T. Shen for discussions relating to similar work, and to Dr. Bernard Widrow for providing the inspiration for this work, and for reviewing initial drafts of this paper.

REFERENCES

Haykin, S. 1994. *Neural Networks: A Comprehensive Foundation*, MacMillan Publishing Company, New York.

Kohonen, T. 1982. Self-organized formation of topologically correct feature maps, *Biological Cybernetics* **43**: 59–69.

Rumelhart, D. E., Hinton, G. E. and Williams, R. J. 1986. Learning internal representations by error propagation, *in* D. E. Rumelhart and J. L. McClelland (eds), *Parallel Distributed Processing*, Vol. 1, The MIT Press, Cambridge, MA, chapter 8.

Werbos, P. 1974. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*, PhD thesis, Harvard University, Cambridge, MA.

Widrow, B., Gupta, N. K. and Maitra, S. 1973. Punish/reward: Learning with a critic in adaptive threshold systems, *IEEE Transactions on Systems, Man, and Cybernetics* **SMC-3**(5): 455–65.

Widrow, B. and Hoff, M. E. 1960. Adaptive switching circuits, *1960 IRE WESCON Convention Record*, Vol. 4, IRE, New York, pp. 96–104.

Widrow, B. and Stearns, S. D. 1985. *Adaptive Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ.