Chapter 10

Temporal-Difference Learning

10.1 Temporal Patterns and Prediction Problems

In this chapter, we consider problems in which we wish to learn to predict the future value of some quantity, say z, from an n-dimensional input pattern, $\mathbf{X}$. In many of these problems, the patterns occur in temporal sequence, $\mathbf{X}_1, \mathbf{X}_2, \ldots, \mathbf{X}_i, \mathbf{X}_{i+1}, \ldots, \mathbf{X}_m$, and are generated by a dynamical process. The components of $\mathbf{X}_i$ are features whose values are available at time, $t = i$. We distinguish two kinds of prediction problems. In one, we desire to predict the value of z at time $t = i + 1$ based on input $\mathbf{X}_i$ for every i. For example, we might wish to predict some aspects of tomorrow's weather based on a set of measurements made today. In the other kind of prediction problem, we desire to make a sequence of predictions about the value of z at some *fixed* time, say $t = m + 1$, based on each of the $\mathbf{X}_i$, $i = 1, \ldots, m$. For example, we might wish to make a series of predictions about some aspect of the weather on next New Year's Day, based on measurements taken every day before New Year's. Sutton [Sutton, 1988] has called this latter problem, *multi-step prediction*, and that is the problem we consider here. In multi-step prediction, we might expect that the prediction accuracy should get better and better as i increases toward m.

10.2 Supervised and Temporal-Difference Methods

A training method that naturally suggests itself is to use the actual value of z at time $m+1$ (once it is known) in a supervised learning procedure using a sequence of training patterns, $\{\mathbf{X}_1, \mathbf{X}_2, \ldots, \mathbf{X}_i, \mathbf{X}_{i+1}, \ldots, \mathbf{X}_m\}$. That is, we seek to learn a function, f, such that $f(\mathbf{X}_i)$ is as close as possible to z for each i. Typically, we would need a training set, Ξ, consisting of several such sequences. We will show that a method that is better than supervised learning for some important problems is to base learning on the difference between $f(\mathbf{X}_{i+1})$ and $f(\mathbf{X}_i)$ rather than on the difference between z and $f(\mathbf{X}_i)$. Such methods involve what is called *temporal-difference (TD) learning*.

We assume that our prediction, $f(\mathbf{X})$, depends on a vector of modifiable weights, $\mathbf{W}$. To make that dependence explicit, we write $f(\mathbf{X}, \mathbf{W})$. For supervised learning, we consider procedures of the following type: For each $\mathbf{X}_i$, the prediction $f(\mathbf{X}_i, \mathbf{W})$ is computed and compared to z, and the learning rule (whatever it is) computes the change, $(\Delta\mathbf{W}_i)$, to be made to $\mathbf{W}$. Then, taking into account the weight changes for each pattern in a sequence all at once after having made all of the predictions with the old weight vector, we change $\mathbf{W}$ as follows:

$$\mathbf{W} \longleftarrow \mathbf{W} + \sum_{i=1}^{m} (\Delta\mathbf{W})_i$$

Whenever we are attempting to minimize the squared error between z and $f(\mathbf{X}_i, \mathbf{W})$ by gradient descent, the weight-changing rule for each pattern is:

$$(\Delta\mathbf{W})_i = c(z - f_i)\frac{\partial f_i}{\partial \mathbf{W}}$$

where c is a learning rate parameter, f_i is our prediction of z, $f(\mathbf{X}_i, \mathbf{W})$, at time $t = i$, and $\frac{\partial f_i}{\partial \mathbf{W}}$ is, by definition, the vector of partial derivatives $(\frac{\partial f_i}{\partial w_1}, \ldots, \frac{\partial f_i}{\partial w_i}, \ldots, \frac{\partial f_i}{\partial w_n})$ in which the w_i are the individual components of $\mathbf{W}$. (The expression $\frac{\partial f_i}{\partial \mathbf{W}}$ is sometimes written $\nabla_{\mathbf{W}} f_i$.) The reader will recall that we used an equivalent expression for $(\Delta\mathbf{W})_i$ in deriving the backpropagation formulas used in training multi-layer neural networks.

The Widrow-Hoff rule results when $f(\mathbf{X}, \mathbf{W}) = \mathbf{X} \bullet \mathbf{W}$. Then:

$$(\Delta\mathbf{W})_i = c(z - f_i)\mathbf{X}_i$$

An interesting form for $(\Delta\mathbf{W})_i$ can be developed if we note that

$$(z - f_i) = \sum_{k=i}^{m}(f_{k+1} - f_k)$$

where we define $f_{m+1} = z$. Substituting in our formula for $(\Delta\mathbf{W})_i$ yields:

$$(\Delta\mathbf{W})_i = c(z - f_i)\frac{\partial f_i}{\partial \mathbf{W}}$$

$$= c\frac{\partial f_i}{\partial \mathbf{W}}\sum_{k=i}^{m}(f_{k+1} - f_k)$$

In this form, instead of using the difference between a prediction and the value of z, we use the differences between successive predictions—thus the phrase *temporal-difference (TD)* learning.

In the case when $f(\mathbf{X}, \mathbf{W}) = \mathbf{X} \bullet \mathbf{W}$, the temporal difference form of the Widrow-Hoff rule is:

$$(\Delta\mathbf{W})_i = c\mathbf{X}_i\sum_{k=i}^{m}(f_{k+1} - f_k)$$

One reason for writing $(\Delta\mathbf{W})_i$ in temporal-difference form is to permit an interesting generalization as follows:

$$(\Delta\mathbf{W})_i = c\frac{\partial f_i}{\partial \mathbf{W}}\sum_{k=i}^{m}\lambda^{(k-i)}(f_{k+1} - f_k)$$

where $0 < \lambda \leq 1$. Here, the λ term gives exponentially decreasing weight to differences later in time than $t = i$. When $\lambda = 1$, we have the same rule with which we began—weighting all differences equally, but as $\lambda \rightarrow 0$, we weight only the $(f_{i+1} - f_i)$ difference. With the λ term, the method is called TD(λ).

It is interesting to compare the two extreme cases:

For TD(0):

$$(\Delta\mathbf{W})_i = c(f_{i+1} - f_i)\frac{\partial f_i}{\partial \mathbf{W}}$$

For TD(1):

$$(\Delta\mathbf{W})_i = c(z - f_i)\frac{\partial f_i}{\partial \mathbf{W}}$$

Both extremes can be handled by the same learning mechanism; only the error term is different. In TD(0), the error is the difference between successive predictions, and in TD(1), the error is the difference between the finally revealed value of z and the prediction. Intermediate values of λ take into account differently weighted differences between future pairs of successive predictions.

Only TD(1) can be considered a pure *supervised* learning procedure, sensitive to the final value of z provided by the teacher. For $\lambda < 1$, we have various degrees of unsupervised learning, in which the prediction function strives to make each prediction more like successive ones (whatever they might be). We shall soon see that these unsupervised procedures result in better learning than do the supervised ones for an important class of problems.

10.3 Incremental Computation of the $(\Delta\mathbf{W})_i$

We can rewrite our formula for $(\Delta\mathbf{W})_i$, namely

$$(\Delta\mathbf{W})_i = c\frac{\partial f_i}{\partial \mathbf{W}}\sum_{k=i}^{m}\lambda^{(k-i)}(f_{k+1} - f_k)$$

to allow a type of incremental computation. First we write the expression for the weight change rule that takes into account all of the $(\Delta\mathbf{W})_i$:

$$\mathbf{W} \longleftarrow \mathbf{W} + \sum_{i=1}^{m} c\frac{\partial f_i}{\partial \mathbf{W}}\sum_{k=i}^{m}\lambda^{(k-i)}(f_{k+1} - f_k)$$

Interchanging the order of the summations yields:

$$\mathbf{W} \longleftarrow \mathbf{W} + \sum_{k=1}^{m} c\sum_{i=1}^{k}\lambda^{(k-i)}(f_{k+1} - f_k)\frac{\partial f_i}{\partial \mathbf{W}}$$

$$= \mathbf{W} + \sum_{k=1}^{m} c(f_{k+1} - f_k)\sum_{i=1}^{k}\lambda^{(k-i)}\frac{\partial f_i}{\partial \mathbf{W}}$$

Interchanging the indices k and i finally yields:

$$\mathbf{W} \longleftarrow \mathbf{W} + \sum_{i=1}^{m} c(f_{i+1} - f_i) \sum_{k=1}^{i} \lambda^{(i-k)} \frac{\partial f_k}{\partial \mathbf{W}}$$

If, as earlier, we want to use an expression of the form $\mathbf{W} \longleftarrow \mathbf{W} + \sum_{i=1}^{m} (\Delta\mathbf{W})_i$, we see that we can write:

$$(\Delta\mathbf{W})_i = c(f_{i+1} - f_i) \sum_{k=1}^{i} \lambda^{(i-k)} \frac{\partial f_k}{\partial \mathbf{W}}$$

Now, if we let $e_i = \sum_{k=1}^{i} \lambda^{(i-k)} \frac{\partial f_k}{\partial \mathbf{W}}$, we can develop a computationally efficient recurrence equation for e_{i+1} as follows:

$$e_{i+1} = \sum_{k=1}^{i+1} \lambda^{(i+1-k)} \frac{\partial f_k}{\partial \mathbf{W}}$$

$$= \frac{\partial f_{i+1}}{\partial \mathbf{W}} + \sum_{k=1}^{i} \lambda^{(i+1-k)} \frac{\partial f_k}{\partial \mathbf{W}}$$

$$= \frac{\partial f_{i+1}}{\partial \mathbf{W}} + \lambda e_i$$

Rewriting $(\Delta\mathbf{W})_i$ in these terms, we obtain:

$$(\Delta\mathbf{W})_i = c(f_{i+1} - f_i) e_i$$

where:

$$e_1 = \frac{\partial f_1}{\partial \mathbf{W}}$$

$$e_2 = \frac{\partial f_2}{\partial \mathbf{W}} + \lambda e_1$$

etc.

Quoting Sutton [Sutton, 1988, page 15] (about a different equation, but the quote applies equally well to this one):

> "... this equation can be computed incrementally, because each $(\Delta\mathbf{W})_i$ depends only on a pair of successive predictions and on the [weighted] sum of all past values for $\frac{\partial f_i}{\partial \mathbf{W}}$. This saves substantially on memory, because it is no longer necessary to individually remember all past values of $\frac{\partial f_i}{\partial \mathbf{W}}$."

10.4 An Experiment with TD Methods

TD prediction methods [especially TD(0)] are well suited to situations in which the patterns are generated by a dynamic process. In that case, sequences of temporally presented patterns contain important information that is ignored by a conventional supervised method such as the Widrow-Hoff rule. Sutton [Sutton, 1988, page 19] gives an interesting example involving a random walk, which we repeat here. In Fig. 10.1, sequences of vectors, $\mathbf{X}$, are generated as follows: We start with vector $\mathbf{X}_D$; the next vector in the sequence is equally likely to be one of the adjacent vectors in the diagram. If the next vector is $\mathbf{X}_C$ (or $\mathbf{X}_E$), the next one after that is equally likely to be one of the vectors adjacent to $\mathbf{X}_C$ (or $\mathbf{X}_E$). When $\mathbf{X}_B$ is in the sequence, it is equally likely that the sequence terminates with $z = 0$ or that the next vector is $\mathbf{X}_C$. Similarly, when $\mathbf{X}_F$ is in the sequence, it is equally likely that the sequence terminates with $z = 1$ or that the next vector is $\mathbf{X}_E$. Thus the sequences are random, but they always start with $\mathbf{X}_D$. Some sample sequences are shown in the figure. This random walk is an example of a *Markov process*; transitions from state i to state j occur with probabilities that depend only on i and j.

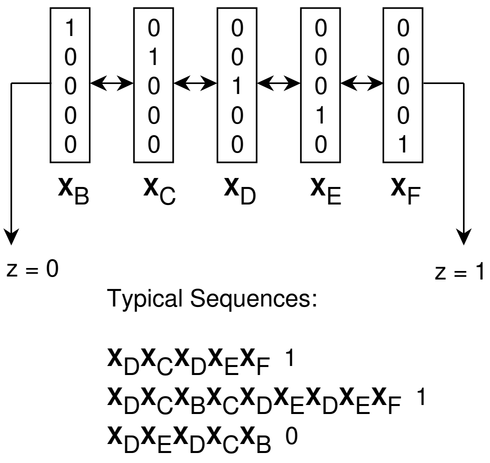

Figure 10.1: A Markov Process

Given a set of sequences generated by this process as a training set, we want to be able to predict the value of z for each $\mathbf{X}$ in a test sequence. We assume that the learning system does not know the transition probabilities.

For his experiments with this process, Sutton used a linear predictor, that is $f(\mathbf{X}, \mathbf{W}) = \mathbf{X} \bullet \mathbf{W}$. The learning problem is to find a weight vector, $\mathbf{W}$, that minimizes the mean-squared error between z and the predicted value of z. Given the five different values that $\mathbf{X}$ can take on, we have the following predictions: $f(\mathbf{X}_B) = w_1$, $f(\mathbf{X}_C) = w_2$, $f(\mathbf{X}_D) = w_3$, $f(\mathbf{X}_E) = w_4$, $f(\mathbf{X}_F) = w_5$, where w_i is the i-th component of the weight vector. (Note that the values of the predictions are not limited to 1 or 0—even though z can only have one of those values—because we are minimizing mean-squared error.) After training, these predictions will be compared with the optimal ones—given the transition probabilities.

The experimental setup was as follows: ten random sequences were generated using the transition probabilities. Each of these sequences was presented in turn to a TD(λ) method for various values of λ. Weight vector increments, $(\Delta\mathbf{W})_i$, were computed after each pattern presentation but no weight changes were made until all ten sequences were presented. The weight vector increments were summed after all ten sequences were presented, and this sum was used to change the weight vector to be used for the next pass through the ten sequences. This process was repeated over and over (using the same training sequences) until (quoting Sutton) "the procedure no longer produced any significant changes in the weight vector. For small c, the weight vector always converged in this way, and always to the same final value [for 100 different training sets of ten random sequences], independent of its initial value." (Even though, for fixed, small c, the weight vector always converged to the same vector, it might converge to a somewhat different vector for different values of c.)

After convergence, the predictions made by the final weight vector are compared with the optimal predictions made using the transition probabilities. These optimal predictions are simply $p(z = 1|\mathbf{X})$. We can compute these probabilities to be 1/6, 1/3, 1/2, 2/3, and 5/6 for $\mathbf{X}_B$, $\mathbf{X}_C$, $\mathbf{X}_D$, $\mathbf{X}_E$, $\mathbf{X}_F$, respectively. The root-mean-squared differences between the best learned predictions (over all c) and these optimal ones are plotted in Fig. 10.2 for seven different values of λ. (For each data point, the standard error is approximately $\sigma = 0.01$.)

Notice that the Widrow-Hoff procedure does not perform as well as other versions of TD(λ) for $\lambda < 1$! Quoting [Sutton, 1988, page 21]:

> "This result contradicts conventional wisdom. It is well known that, under repeated presentations, the Widrow-Hoff procedure minimizes the RMS error between its predictions and the actual outcomes in the training set ([Widrow & Stearns, 1985]).

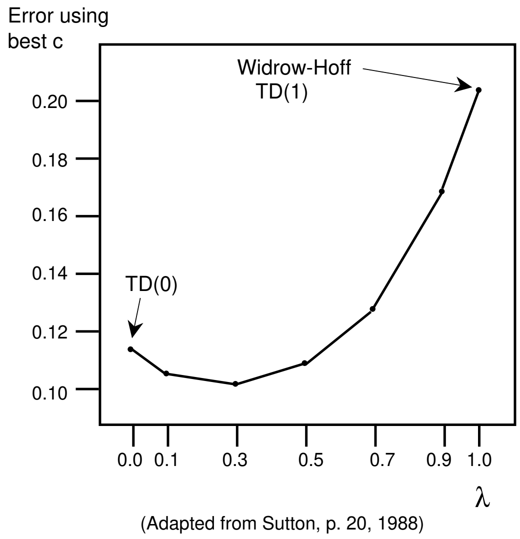

Figure 10.2: Prediction Errors for TD(λ)

How can it be that this optimal method peformed worse than all the TD methods for $\lambda < 1$? The answer is that the Widrow-Hoff procedure only minimizes error *on the training set*; it does not necessarily minimize error for future experience. [Later] we prove that in fact it is linear TD(0) that converges to what can be considered the optimal estimates for matching future experience—those consistent with the maximum-likelihood estimate of the underlying Markov process."

10.5 Theoretical Results

It is possible to analyze the performance of the linear-prediction TD(λ) methods on Markov processes. We state some theorems here without proof.

Theorem 10.1 (Sutton, page 24, 1988) *For any absorbing Markov chain, and for any linearly independent set of observation vectors* $\{\mathbf{X}_i\}$ *for the non-terminal states, there exists an* $\varepsilon > 0$ *such that for all positive* $c < \varepsilon$ *and for any initial weight vector, the predictions of linear TD(0) (with* *weight updates after each sequence) converge in expected value to the optimal (maximum likelihood) predictions of the true process.*

Even though the expected values of the predictions converge, the predictions themselves do not converge but vary around their expected values depending on their most recent experience. Sutton conjectures that if c is made to approach 0 as training progresses, the variance of the predictions will approach 0 also.

Dayan [Dayan, 1992] has extended the result of Theorem 9.1 to TD(λ) for arbitrary λ between 0 and 1. (Also see [Dayan & Sejnowski, 1994].)

10.6 Intra-Sequence Weight Updating

Our standard weight updating rule for TD(λ) methods is:

$$\mathbf{W} \longleftarrow \mathbf{W} + \sum_{i=1}^{m} c(f_{i+1} - f_i) \sum_{k=1}^{i} \lambda^{(i-k)} \frac{\partial f_k}{\partial \mathbf{W}}$$

where the weight update occurs *after* an entire sequence is observed. To make the method truly incremental (in analogy with weight updating rules for neural nets), it would be desirable to change the weight vector after every pattern presentation. The obvious extension is:

$$\mathbf{W}_{i+1} \longleftarrow \mathbf{W}_i + c(f_{i+1} - f_i) \sum_{k=1}^{i} \lambda^{(i-k)} \frac{\partial f_k}{\partial \mathbf{W}}$$

where f_{i+1} is computed before making the weight change; that is, $f_{i+1} = f(\mathbf{X}_{i+1}, \mathbf{W}_i)$. But that would make $f_i = f(\mathbf{X}_i, \mathbf{W}_{i-1})$, and such a rule would make the prediction difference, namely $(f_{i+1} - f_i)$, sensitive both to changes in $\mathbf{X}$ and changes in $\mathbf{W}$ and could lead to instabilities. Instead, we modify the rule so that, for every pair of predictions, $f_{i+1} = f(\mathbf{X}_{i+1}, \mathbf{W}_i)$ and $f_i = f(\mathbf{X}_i, \mathbf{W}_i)$. This version of the rule has been used in practice with excellent results.

For TD(0) and linear predictors, the rule is:

$$\mathbf{W}_{i+1} = \mathbf{W}_i + c(f_{i+1} - f_i)\mathbf{X}_i$$

The rule is implemented as follows:

1. Initialize the weight vector, $\mathbf{W}$, arbitrarily.
2. For $i = 1, ..., m$, **do**:
   (a) $f_i \longleftarrow \mathbf{X}_i \bullet \mathbf{W}$
   (We compute f_i anew each time through rather than use the value of f_{i+1} the previous time through.)
   (b) $f_{i+1} \longleftarrow \mathbf{X}_{i+1} \bullet \mathbf{W}$
   (c) $d_{i+1} \longleftarrow f_{i+1} - f_i$
   (d) $\mathbf{W} \longleftarrow \mathbf{W} + c\ d_{i+1}\mathbf{X}_i$
   (If f_i were computed again with this changed weight vector, its value would be closer to f_{i+1} as desired.)

The linear TD(0) method can be regarded as a technique for training a very simple network consisting of a single dot product unit (and no threshold or sigmoid function). TD methods can also be used in combination with backpropagation to train neural networks. For TD(0) we change the network weights according to the expression:

$$\mathbf{W}_{i+1} = \mathbf{W}_i + c(f_{i+1} - f_i)\frac{\partial f_i}{\partial \mathbf{W}}$$

The only change that must be made to the standard backpropagation weight-changing rule is that the difference term between the desired output and the output of the unit in the final (k-th) layer, namely $(d - f^{(k)})$, must be replaced by a difference term between successive outputs, $(f_{i+1} - f_i)$. This change has a direct effect only on the expression for $\delta^{(k)}$ which becomes:

$$\delta^{(k)} = 2(f'^{(k)} - f^{(k)})f^{(k)}(1 - f^{(k)})$$

where $f'^{(k)}$ and $f^{(k)}$ are two successive outputs of the network.

The weight changing rule for the i-th weight vector in the j-th layer of weights has the same form as before, namely:

$$\mathbf{W}_i^{(j)} \longleftarrow \mathbf{W}_i^{(j)} + c\delta_i^{(j)}\mathbf{X}^{(j-1)}$$

where the $\delta_i^{(j)}$ are given recursively by:

$$\delta_i^{(j)} = f_i^{(j)}(1 - f_i^{(j)}) \sum_{l=1}^{m_{j+1}} \delta_l^{(j+1)} w_{il}^{(j+1)}$$

and $w_{il}^{(j+1)}$ is the l-th component of the i-th weight vector in the $(j+1)$-th layer of weights. Of course, here also it is assumed that $f'^{(k)}$ and $f^{(k)}$ are computed using the same weights and *then* the weights are changed. In the next section we shall see an interesting example of this application of TD learning.

10.7 An Example Application: TD-gammon

A program called TD-gammon [Tesauro, 1992] learns to play backgammon by training a neural network via temporal-difference methods. The structure of the neural net, and its coding is as shown in Fig. 10.3. The network is trained to minimize the error between actual payoff and estimated payoff, where the actual payoff is defined to be $d_f = p_1 + 2p_2 - p_3 - 2p_4$, and the p_i are the actual probabilities of the various outcomes as defined in the figure.

TD-gammon learned by using the network to select that move that results in the best predicted payoff. That is, at any stage of the game some finite set of moves is possible and these lead to the set, $\{\mathbf{X}\}$, of new board positions. Each member of this set is evaluated by the network, and the one with the largest predicted payoff is selected if it is white's move (and the smallest if it is black's). The move is made, and the network weights are adjusted to make the predicted payoff from the original position closer to that of the resulting position.

The weight adjustment procedure combines temporal-difference (TD(λ)) learning with backpropagation. If d_t is the network's estimate of the payoff at time t (before a move is made), and d_{t+1} is the estimate at time $t+1$ (after a move is made), the weight adjustment rule is:

$$\Delta \mathbf{W}_t = c(d_{t+1} - d_t) \sum_{k=1}^{t} \lambda^{t-k} \frac{\partial d_k}{\partial \mathbf{W}}$$

where $\mathbf{W}_t$ is a vector of *all* weights in the network at time t, and $\frac{\partial d_k}{\partial \mathbf{W}}$ is the gradient of d_k in this weight space. (For a layered, feedforward network, such as that of TD-gammon, the weight changes for the weight vectors in each layer can be expressed in the usual manner.)

To make the special cases clear, recall that for TD(0), the network would be trained so that, for all t, its output, d_t, for input $\mathbf{X}_t$ tended toward its expected output, d_{t+1}, for input $\mathbf{X}_{t+1}$. For TD(1), the network would be trained so that, for all t, its output, d_t, for input $\mathbf{X}_t$ tended toward the expected final payoff, d_f, given that input. The latter case is the same as the Widrow-Hoff rule.

After about 200,000 games the following results were obtained. TD-gammon (with 40 hidden units, $\lambda = 0.7$, and $c = 0.1$) won 66.2% of 10,000 games against SUN Microsystems Gammontool and 55% of 10,000 games against a neural network trained using expert moves. Commenting on a later version of TD-gammon, incorporating special features as inputs, Tesauro said: "It appears to be the strongest program ever seen by this author."

10.8 Bibliographical and Historical Remarks

To be added.

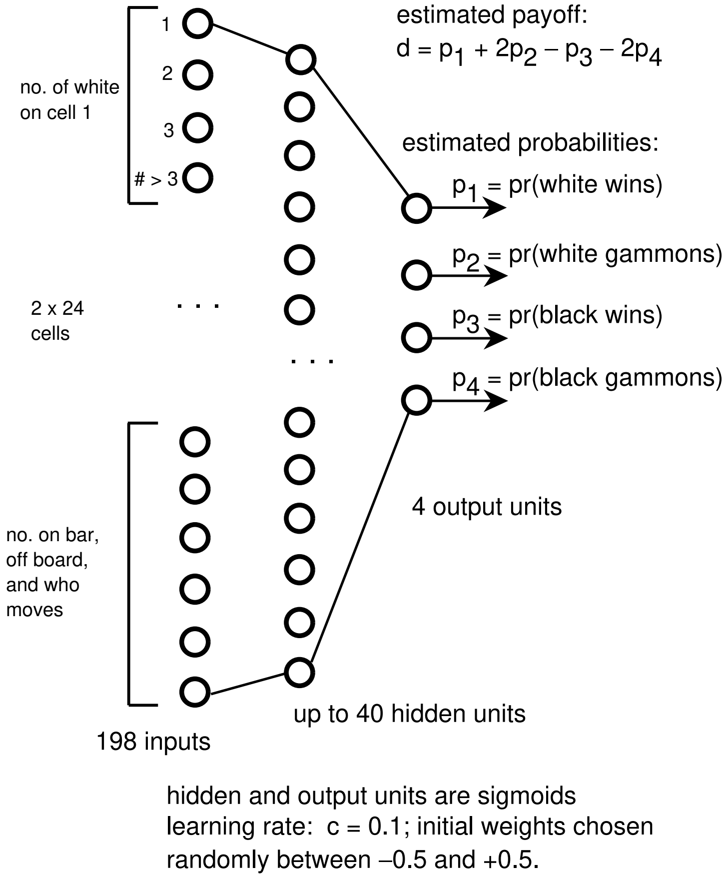

Figure 10.3: The TD-gammon Network