e-PG Pathshala

Subject : Computer Science

Paper: Machine Learning

Module: Temporal Difference Learning

Module No: CS/ML/40

Quadrant 1- e-text

Welcome to the e-PG Pathshala Lecture Series on Machine Learning. In this module we will discuss in detail the concept of Temporal Difference Learning which we introduced in the last module.

Learning Objectives

The learning objectives of the module are as follows:

- To understand Temporal Difference Learning
- To describe the Optimal Value Functions
- To analyse TD-Prediction
- To describe Learning an action value function

40.1 Introduction

Policy evaluation algorithms are intended to estimate the value functions V^{π} or Q^{π} for a given policy π. Typically these are on-policy algorithms, and the considered policy is assumed to be stationary (or "almost" stationary). There are three classes of policy evaluation algorithms namely Monte Carlo methods, Dynamic Programming, and Temporal Difference Learning or TD learning. Monte Carlo methods do not need a model of the learning environment. From experience in form of sequences of state-action-reward-samples they can approximate future rewards. Monte-Carlo methods are based on the simple idea of averaging a number of random samples of a random quantity in order to estimate its average. Dynamic Programming is based on the Bellman Equation where the problem is broken down into sub problems and depends on a perfect model of the environment. However both Dynamic Programming and Monte Carlo methods only update after a complete sequence that is when the final state is reached. The main issues are whether it is possible to avoid the computational expense and space requirements for storing the transition model estimate of a full Dynamic programming policy evaluation.

TD learning is considered as the most novel idea in reinforcement learning. Temporal Difference Learning is a model free approach which does not store an estimate of entire transition function but instead stores estimate of V^{π}, which requires only O(n) space. It carries out local, cheap updates of utility/value function on a per-action basis. As you may recall Value Functions are state-action pair functions that estimate how good a particular action will be in a given state, or what the return for

that action is expected to be a Temporal Difference (TD). Learning methods can be used to estimate these value functions. If the value functions were to be calculated without estimation, the agent would need to wait until the final reward was received before any state-action pair values can be updated. Once the final reward was received, the path taken to reach the final state would need to be traced back and each value updated accordingly.

40.1.1 Comparison – Monte Carlo, Dynamic Programming and TD Approaches

Before we discuss TD method let us compare Monte Carlo and Dynamic Programming methods. We will discuss two aspects namely bootstrapping and sampling. In bootstrapping update involves an estimate from a nearby state. While Monte Carlo method does not bootstrap that is each state is independently estimated, Dynamic Programming methods bootstraps that is estimates for each state is dependent on nearby states. In sampling update involves an actual return Monte Carlo method does sampling that is learns from experience in the environment and gets an actual return. Dynamic programming does not sample but learns by spreading constraints until convergence through iterating the Bellman equations without interaction with the environment.

TD bootstraps that is learning involves an estimate of a nearby state. TD also samples where update involves an actual return.

40.2 Optimal Value Functions

Before we discuss TD learning let us recall that for finite MDPs, policies can be **partially ordered** that is there is always at least one (and possibly many) policy that is better than or equal to all the others. This is an **optimal policy**. We denote them all π^*. Optimal policies share the same **optimal state-value function**:

$$V^*(s) = \max_{\pi} V^{\pi}(s) \quad \text{for all} \quad s \in S$$

Optimal policies also share the same optimal action-value function:

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a) \quad \text{for all} \quad s \in S \text{ and } a \in A(s)$$

This is the expected return for taking action '*a*' in state '*s*' and thereafter following an optimal policy. Now let us discuss the Bellman optimality equation for V*.

40.2.1 Bellman Optimality Equation for V*

The value of a state under an optimal policy must equal the expected return for the best action from that state where **V*** is the unique solution of this system of nonlinear equations.

$$\begin{aligned} V^*(s) &= \max_{a \in A(s)} Q^{\pi^*}(s,a) \\ &= \max_{a \in A(s)} E\left\{ r_{t+1} + \gamma V^*(s_{t+1}) \middle| s_t = s, a_t = a \right\} \end{aligned}$$

$$= \max_{a \in A(s)} \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V^*(s') \right]$$

The relevant backup diagram is shown in Figure 40.1.

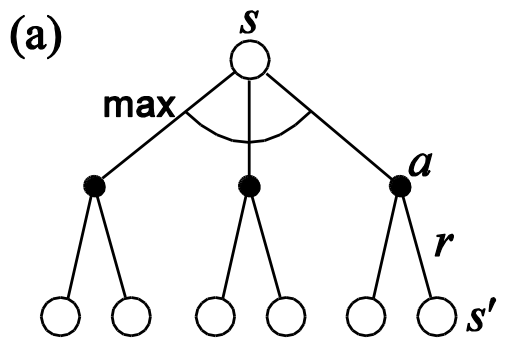

Figure 40.1 Backup diagram

40.2.2 Bellman Optimality Equation for Q*

The unique solution of this system is given by the nonlinear equations, where **Q*** is the unique solution of this system of nonlinear equations

$$Q^*(s,a) = E\left\{ r_{t+1} + \gamma \max_{a'} Q^*(s_{t+1}, a') \middle| s_t = s, a_t = a \right\}$$

$$= \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma \max_{a'} Q^*(s', a') \right]$$

The relevant backup diagram is shown in Figure 40.2.

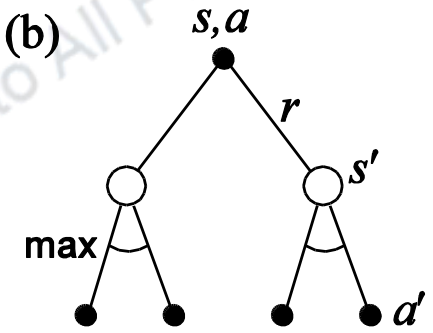

Figure 40.2 Backup diagram

Given Q^*, the agent does not even have to do a one-step-ahead search in order to find the optimal Action-Value functions.

$$\pi^*(s) = \arg\max_{a \in A(s)} Q^*(s,a) \qquad (6)$$

40.2.3 Solving the Bellman Optimality Equation

Finding an optimal policy by solving the Bellman Optimality Equation requires the following that is accurate knowledge of environment dynamics, enough space and time to do the computation, and the Markov Property must be satisfied. The space

required is polynomial in number of states but the number of states is often huge (e.g., backgammon has about 10**20 states). Therefore usually we have to settle for approximations. Many reinforcement learning methods can be understood as approximately solving the Bellman Optimality Equation.

40.3 Policy Evaluation

Policy Evaluation (the prediction problem) is the computation of the state-value function V^{π} for a given policy π

$$V(s_t) \leftarrow V(s_t) + \alpha\left[R_t - V(s_t)\right]$$

where R_t is the actual return following state s_t.

The simplest TD method, TD(0):

$$V(s_t) \leftarrow V(s_t) + \alpha\left[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)\right]$$

where $r_{t+1} + \gamma V(s_{t+1})$ is the target.

Monte Carlo methods do not need full knowledge of environment, but just needs experience, or simulated experience (Figure 40.3). One method is by averaging sample returns which is generally defined only for episodic tasks. This method is similar to DP in terms of policy evaluation, policy improvement.

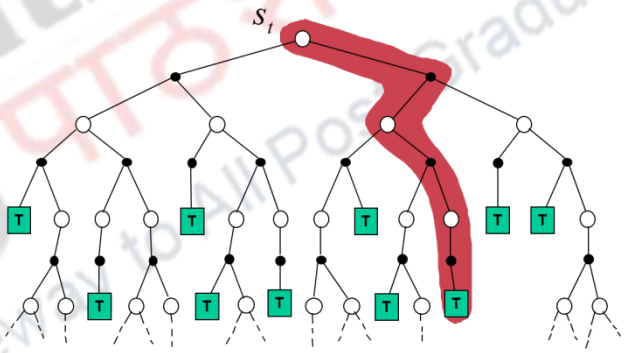

Figure 40.3 Simple Monte Carlo Method

Dynamic Programming (Figure 40.4) needs complete model of the environment and rewards. DP bootstraps that is updates estimates on the basis of other estimates

$$V(s_t) \leftarrow E_\pi \{r_{t+1} + \gamma V(s_t)\}$$

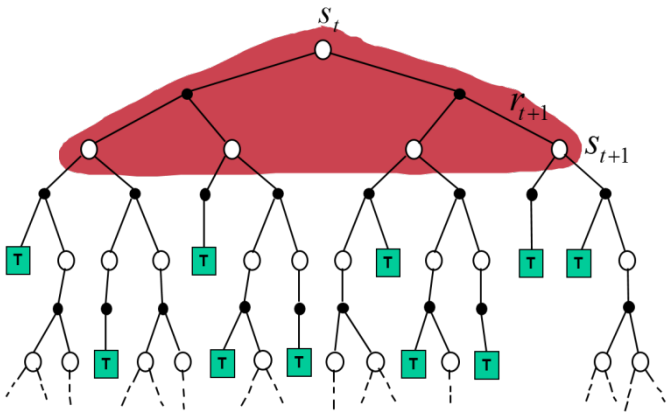

Figure 40.4 The Dynamic Programming Method

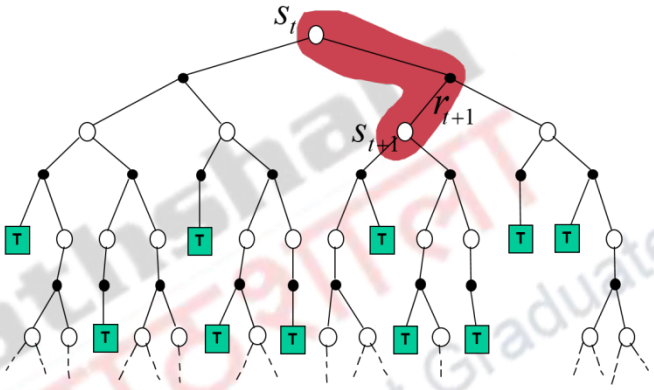

Figure 40.5 Temporal Difference Method

In TD prediction we estimate the actual return as the sum of the next reward plus the value of the next state (Figure 40.5). As we discussed earlier when the environment, $P(s_{t+1} \mid s_t, a_t)$, $p(r_{t+1} \mid s_t, a_t)$, is not known it is called as model-free learning. The temporal difference is defined as the difference between the value of the current action and the value discounted from the next state.

40.3 TD Prediction

TD update for transition from s to s' is given in Figure 40.6.

$$V^\pi(s) \leftarrow V^\pi(s) + \alpha\,(R(s) + \gamma V^\pi(s') - V^\pi(s))$$

updated estimate

learning rate

(noisy) sample of value at s based on next state s'

Figure 40.6 TD Update

The update is maintaining a "mean" of the (noisy) value samples. If the learning rate decreases appropriately with the number of samples (e.g.1/n) then the value estimates will converge to true values as shown in Figure 40.7.

$$V^{\pi}(s) = R(s) + \gamma \sum_{s'} T(s,a,s')V^{\pi}(s')$$

Figure 40.7 Converged Value Estimates

We estimate the actual return by n-step return

$R_t^{(n)} = r_{t+1} + \gamma\, r_{t+2} + \ldots + \gamma^{n-1}\, r_{t+n} + \gamma^n\, V(s_{t+n})$

In other words TD(λ) is a weighted average of n-step returns

$R_t^{\lambda} = (1 - \lambda) \sum_n \lambda^{(n-1)}\, R_t^{(n)}$

In case λ=0, we only look at next reward, when λ=1, we have the constant-α Monte Carlo. Monte Carlo methods normally wait until the return following the visit is known, then use that return as a target for $V(s_t)$. Constant-α Monte Cralo methods wait until a constant time and then use that return.

40.4 Error signal

Another aspect we need for estimation and bootstrapping is the error signal. Error signal is defined as the difference between current estimate and improved estimate. It drives change of current estimate. There are many types of errors as discussed below:

- Supervised learning error:

  error(x) = target_output(x) - learner_output(x)

- Bellman error (DP):

  $$error(s) = \max_a \sum_{s'} P_{ss'}^a \left[R_{ss'}^a + \gamma V(s')\right] - V(s)$$

  "1-step full-width lookahead" - "0-step lookahead"

- Monte Carlo error:

  error(s) = <R_t> - V(s)

  "many-step sample lookahead" - "0-step lookahead"

40.4.1 TD error signal

Here we take one step using current policy, observe r and s', then:

$$error(s) = \left[r + \gamma V(s')\right] - V(s) \qquad (3)$$

In particular, for undiscounted sequences with no intermediate rewards, we have simply:

$$error(s) = V(s') - V(s) \qquad (4)$$

Self-consistent prediction goal is to predict returns that should be self-consistent from one time step to the next (true of both TD and DP).

Learning using the Error Signal: we could just do a reassignment:

$$V(s) \leftarrow [r + \gamma V(s')] \tag{5}$$

But it is often a good idea to learn incrementally:

$$\Delta V(s) = \alpha([r + \gamma V(s')] - V(s)) \tag{6}$$

where '**α**' is a small "learning rate" parameter (either constant, or decreases with time). The above algorithm is known as "TD(0)" .

40.5 TD Learning

TD Learning combines the "bootstrapping" (1-step self-consistency) idea of DP with the "sampling" idea of Monte Carlo (MC). Like MC, it doesn't need a model of the environment, only experience. However TD, but not MC, can be fully incremental. Here we can learn before knowing the final outcome and we can learn without the final outcome (from incomplete sequences). The incorporation of bootstrapping into TD has reduced its variance compared to Monte Carlo, but possibly introduced greater bias.

40.5.1 Example – Driving Home

Initially we will consider TD(0) that is the case where λ =0 in TD(λ). In this case we only look at the next reward. To understand the concept we consider the example shown in Figure 40.8. Value of each state in this example is the expected time-to-go.

State	Elapsed Time (minutes)	Predicted Time to Go	Predicted Total Time
leaving office	0	30	30
reach car, raining	5	35	40
exit highway	20	15	35
behind truck	30	10	40
home street	40	3	43
arrive home	43	0	43

rewards

Returns from each state

Figure 40.8 Driving Home Example

Figure 40.9 shows the estimated values when we use Monte Carlo and TD approaches.

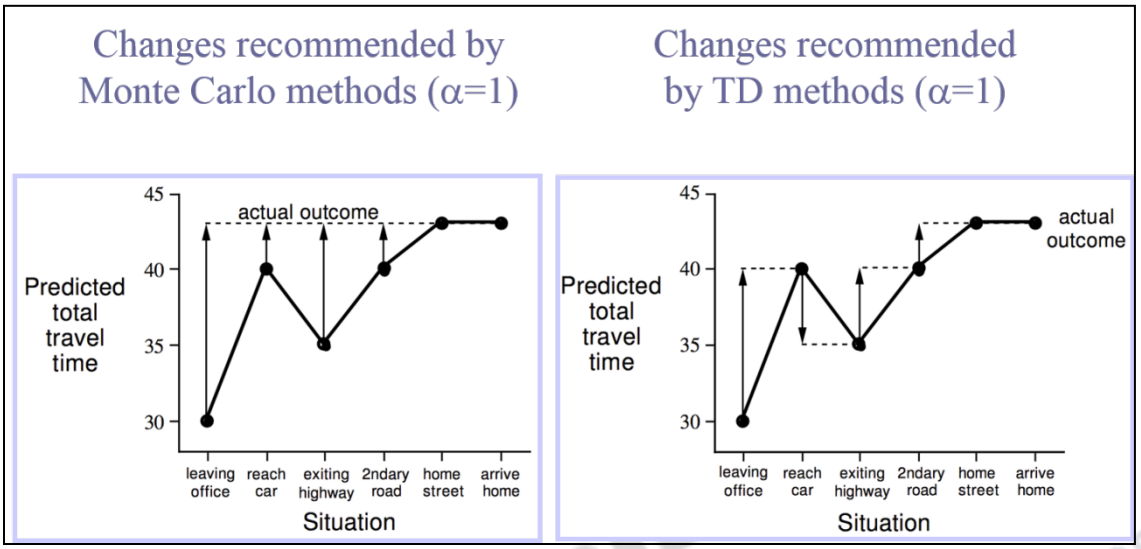

Figure 40.9 Predicted Values for Monte Carlo and TD Methods

Now the question is whether it is really necessary to wait until the end of the episode (here – arrive home) to start learning. According to Monte Carlo method it is necessary to wait. However TD learning argues that learning can occur on-line. Suppose, on another day, you again estimate when leaving your office that it will take 30 minutes to drive home, but then you get stuck in a massive traffic jam. In TD the initial estimate would be changed. The algorithm for TD(0) is given in Figure 40.10. This can be considered a control method where the policy is always updated using a greedy approach with respect to the current estimate. This is also called Sarsa, the On-Policy TD Control algorithm.

Initialize $Q(s, a)$ arbitrarily
Repeat (for each episode):
  Initialize s
  Choose a from s using policy derived from Q (e.g., ε-greedy)
  Repeat (for each step of episode):
    Take action a, observe r, s'
    Choose a' from s' using policy derived from Q (e.g., ε-greedy)
    $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$
    $s \leftarrow s'$; $a \leftarrow a'$;
  until s is terminal

Figure 40.10 TD(0) Algorithm

40.5.2 Optimality of TD(0)

Suppose only a finite amount of experience is available, say 10 episodes or 100 time steps. Intuitively, we repeatedly present the experience until convergence is achieved. Updates are made after a batch of training data. For any finite Markov prediction task, under batch updating, TD(0) converges for sufficiently small value of λ. MC method also converges deterministically but to a different answer

40.6 TD(λ)

Figure 40.11shows the TD prediction for n steps and the mathematics behind that is given in Figure 40.12.

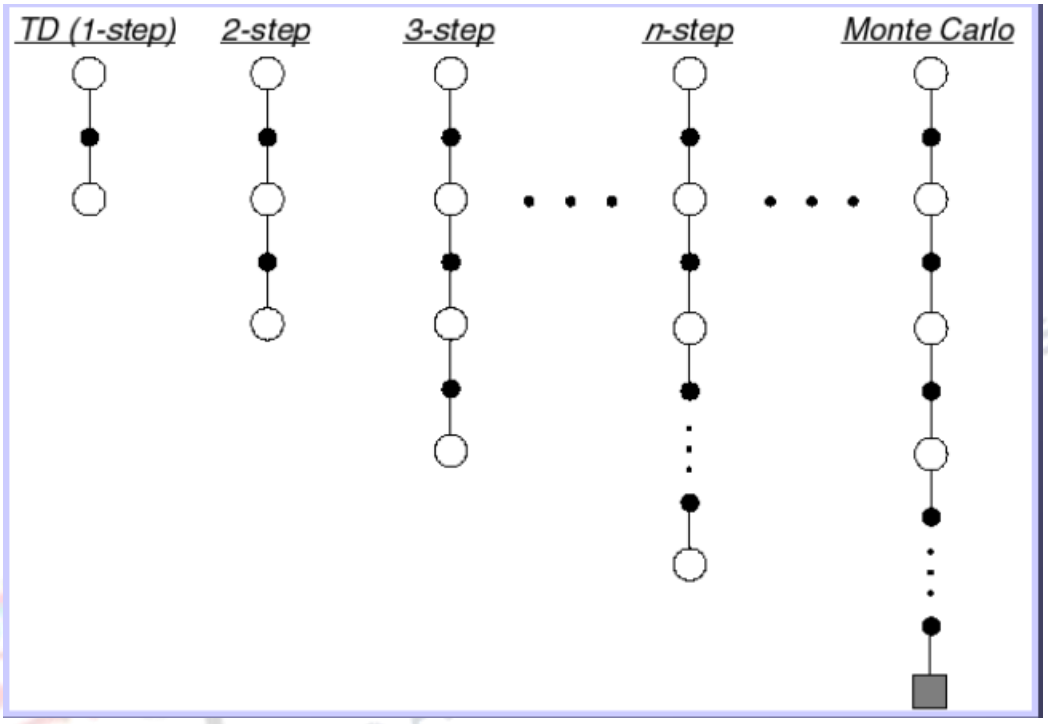

Figure 40.11 TD(λ) Prediction

- Monte Carlo: $R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^{T-t-1} r_T$
- TD: $R_t^{(1)} = r_{t+1} + \gamma V_t(s_{t+1})$
  - Use V to estimate remaining return
- n-step TD:
  - 2 step return: $R_t^{(2)} = r_{t+1} + \gamma r_{t+2} + \gamma^2 V_t(s_{t+2})$
  - n-step return: $R_t^{(n)} = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^{n-1} r_{t+n} + \gamma^n V_t(s_{t+n})$

Figure 40.12 Mathematics of N steps

40.5.1 λ Parameter

The λ in TD(λ) provides a smooth interpolation between λ=0 (pure TD) and λ=1 (pure MC), For many toy grid-world type problems, can show that intermediate values of λ work best. However for real-world problems, best λ will be highly problem-dependent.

TD(λ) converges to the correct value function V^o (s) with probability 1 for all λ. For which a lookup table representation (V^o (s) is a table), it must visit all states an infinite # of times, and a certain schedule for decreasing α(t). However TD(k) converges only for a fixed policy.

40.6 Learning backgammon using TD(λ)

Neural net observes a sequence of input patterns x_1, x_2, x_3, …, x_f which is the sequence of board positions occurring during a game and let x_f be the final position. The representation of the board game can be raw board description that is the number of White or Black checkers at each location. We can use simple truncated unary encoding for this representation. At final position x_f, reward signal z given can be z = 1 if White wins and z = 0 if Black wins. We can train the neural net using gradient version of TD(λ). The trained NN output V_t=V(x_t,w) should estimate probability of (White wins | x_t) that is white win given the board position x_t.

40.6.1 Multi layer Neural Network

The multilayer neural network for learning Backgammon is shown in Figure 40.13.

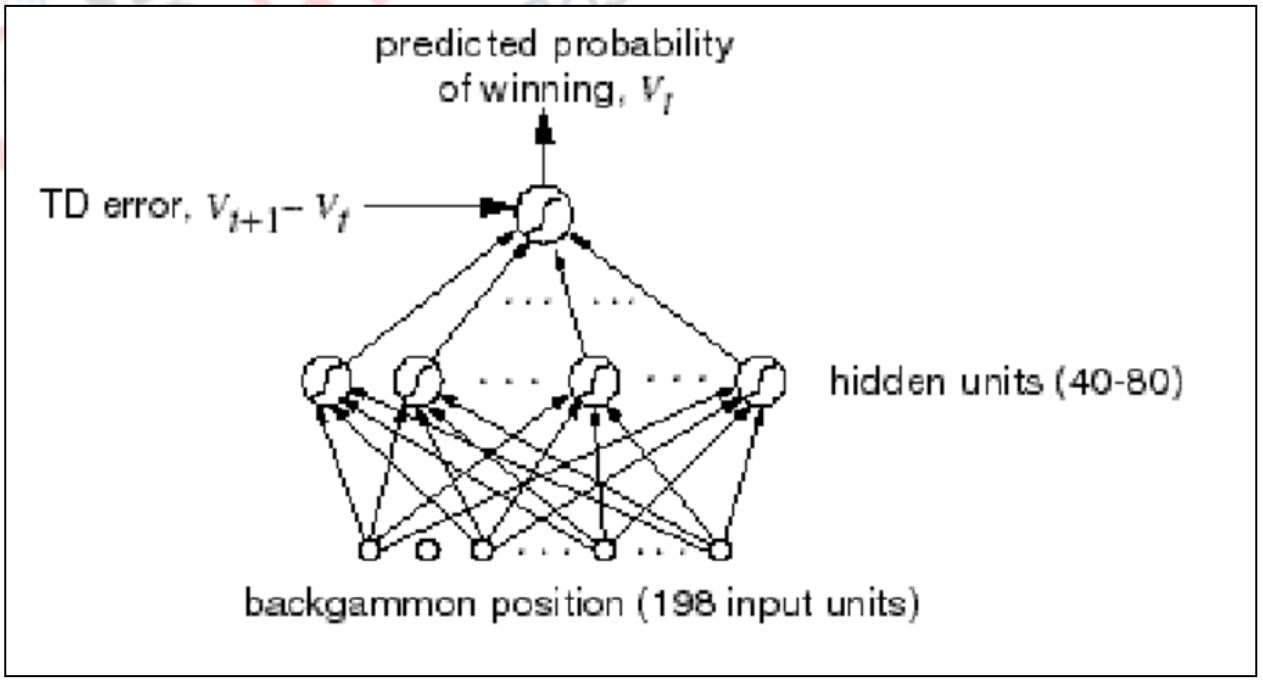

Figure 40.13 Multilayer Network

Let the neural net make the moves **itself**, using its current evaluator where all legal moves are scored and pick maximum value V_t for White, and minimum V_t for Black.

TD-Gammon can teach itself by playing games against itself and learning from the outcome. Works even starting from random initial play and zero initial expert knowledge (surprising). The program achieves strong intermediate play, if we add hand-crafted features it is able to achieve advanced level of play, 2-ply search-strong master play and 3-ply search almost superhuman play. “TD-Leaf” that is n-step TD backups in 2-player games is able to achieve great results for checkers and chess.

40.7 Advantages of TD Learning

TD methods do not require a model of the environment, only experience. TD methods can be fully incremental. With TD learning can happen before knowing the final outcome and hence requires less memory and less peak computation. We can in fact learn without even knowing the final outcome that is we can learn from incomplete sequences. This type of learning helps with applications that have very long episodes. Both MC and TD converge under certain assumptions but generally TD does better on stochastic tasks.

Summary

- Outlined the TD Learning, optimal value functions, and TD prediction
- Explained advantages of TD Learning
- Reviewed the Bellman's' optimal value functions and TD Prediction
- Discussed TD-learning steps and examples

Summary of the Complete paper

- Machine learning – an important approach to solve many of today’s problems
- In this course we discussed the following topics
    - Basics of Machine Learning
    - Mathematical Foundations – Probability, Probability Distributions, Linear Algebra, Decision Theory and Information Theory
    - Supervised Techniques & Classification – KNN, Decision Trees, SVM, Neural Networks & Genetic Algorithms
    - Clustering – K-Means
    - Semi-supervised Learning
    - Baye’s Learning – Naïve Bayes, Bayes Belief Networks, EM method, HMM
    - Reinforcement Learning