Computational Learning Theory	**Fall Semester, 1992/3**

Lecture 10: December 30

Lecturer: Yishay Mansour *Scribe: Libi Hertzberg, Uri Schneider*

10.1 TD-Gammon

10.1.1 MDPs with a very large number of states

What happens when the number of states is too large to enumerate?

For example: the game of backgammon.
Board description: there are 24 positions on the board.
Pieces: each player has 15 pieces.
The goal: to bring your pieces to your side of the board, and then remove them.
Game play: In each turn a player throws two dice, and moves his pieces accordingly.
Interaction:

1. If a white piece is the only one in its position, and a black piece reaches that position, the white piece is removed, and needs to start from the beginning.
2. If two or more white pieces are in the same position, then a black piece can not "land" there.

End of game: The one who removed all his pieces out of the board first wins the game.

Lets try to estimate the number of states in backgammon: Descrption of a state: For every position of 24 + 2, remember the number of pieces of each color which are located there. Number of states for one player's pieces: $\cong 2.5 \times 10^{10}$.
Total number of states $\cong 10^{20}$.

10.1.2 State encoding TD-Gammon

TD-Gammon uses a neural network with 198 inputs. For each position and for each color there are four inputs:

1. equals true if there is at least one piece present.
2. equals true if there are at least two pieces present.
3. equals true if there are at least three pieces present.

4. has a value of $\frac{n-3}{2}$ if there are at least four pieces present.

If no piece is present then all four inputs are false.
Two additional inputs encode the number of pieces that were "taken" for each color. Each one has a value of $\frac{n}{2}$ where n is the number of eaten pieces. Two other inputs encode the number of pieces removed. Each one has a value of $\frac{n}{15}$ where n is the number of pieces removed. Two last boolean inputs encode for each player whether it is his turn currently.

10.1.3 MDP description

1. The discount factor is set to $\lambda = 1$.
2. Immediate rewards:
   (a) In non-terminal states the immediate rewards are equal to 0.
   (b) In a winning terminal state the immediate reward equals 1.
   (c) In a losing terminal state the immediate reward equals 0.

I.e. the TD has a difference of $V(s_{t+1}) - V(s_t)$ for all non-terminal states.

In every move the parameters are changed in the direction of the TD. Generally, assume we have a function $F(s,r) = V_r(s)$,which gives each state s a value according to r (r is actually a program which gets a state s as an input). We will update $\overrightarrow{r}$ by the derivative of $V_r(s)$ according to $\overrightarrow{r}$. I.e. according to the vector $[\frac{\partial}{\partial r_1}V_r(s), \ldots, \frac{\partial}{\partial r_k}V_r(s)]$. Updating $\overrightarrow{r}$ in this direction will hopefully change the value of $V_r(s)$ in the "right" direction. TD tries to minimize the difference between two succeeding states: assuming that $V_r(s_{t+1}) > V_r(s_t)$, we would like to "strengthen" the weight of the action taken, and update in the direction of $[V_r(s_{t+1}) - V_r(s_t)]\nabla_{\overrightarrow{r}} V_t(s_t)$

For example, if r is a table: $\overrightarrow{r} = (r_1, \ldots, r_k)$, then $\nabla_{\overrightarrow{r}} V_r(s) = (0, 0, \ldots, 1, 0, \ldots, 0)$, and the update will occur only in the s'th entry of r.

TD Gammon updates $\overrightarrow{r}$ while running the system, where the current policy is the greedy policy with respect to the function $V_r(s)$.

Specifically, $\overrightarrow{r_{t+1}} = \overrightarrow{r_t} + \alpha[V_t(s_{t+1}) - V_t(s_t)] \cdot \overrightarrow{e_t}$, where $\overrightarrow{e_t} = \gamma\lambda\overrightarrow{e_{t-1}} + \nabla_{\overrightarrow{r_t}} V_t(s_t)$

TD-Gammon simply "learns" by playing against itself, i.e it updated $\overrightarrow{r_t}$ while playing against itself. After about $300,000$ games the system achieved a very good playing skill (equivalent to other backgammon playing programs).

10.1.4 "Encoding" Functions $F : S \rightarrow \Re$

One way to implement F is by using a table. This enables us to implement *any* function F. The drawback in this approach is that the space needed is proportional to the number of states. Since, in backgammon, the number of states is very large ($\cong 10^{20}$), using a table is impractical.

We will implement F in way which does *not* enable any mapping $S \to \Re$. First, we choose a mapping $H : S \to U$, which for each state s maps a vector $\overrightarrow{u}$, which will "represent" it. Choosing such appropriate mapping h determines the performance of the algorithm. For example:

1. TD-Gammon - $\overrightarrow{u}$ is a vector with 198 slots.

2. 21 (blackjack) - u is the sum of cards.

From here on, we exchange s with $\overrightarrow{u} = H(s)$, and perform the calculations on the vector $\overrightarrow{u}$.

In the game of 21 there were few such vectors $\overrightarrow{u}$, and we could build a table for all of them. It is not the case for backgammon.

methods of encoding

1. *Linear Function*

   We choose a weight function, $\overrightarrow{w}$, for which the value function is $V_w(s) = \overrightarrow{w} \cdot \overrightarrow{H(s)} = \sum w_i u_i$. Naturally, we cannot calculate *every function this way, but in many cases it gives good results. Another advantage is the simplicity of calculating the derivative* $\nabla_w V_w(s) = \nabla_w \overrightarrow{w} \cdot \overrightarrow{u} = \overrightarrow{u}$. *The derivative is the encoding of the state, which is very convenient computation-wise.*

2. Neural Networks *(figure 10.1)*

Calculation of a gate (see figure 10.2):
First, we compute $\alpha = \sum w_i z_i$. Then give α as an argument to a non linear function.

1. perceptron : $h(\alpha) = \{\begin{smallmatrix} 1 & ,\alpha>0 \\ 0 & ,\alpha\leq 0 \end{smallmatrix}$
   The problem: it isn't a continuous function.

2. sigmoid function: a continuous perceptron approximation,

   $$\sigma(\alpha) = \frac{1}{1+e^{-\alpha}} .$$

   Note that, $\sigma(0) = \frac{1}{2}$, and when $\alpha \to \infty$, $\sigma(\alpha) \to 1$, and when $\alpha \to -\infty$, $\sigma(\alpha) \to 0$.

   It is possible to connect a large number of such gates, each gate has its own weight vector w_i. There are simple algorithms for computing the derivative by using the chain rule.

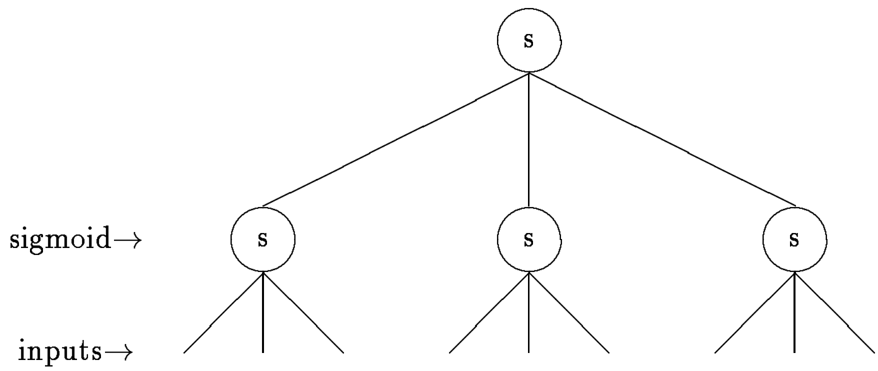

Figure 10.1: A Neural Network

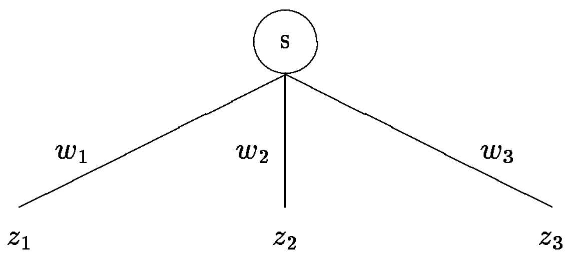

Figure 10.2: Calculating The Value Of A Gate

Basically, we are left with a learning problem: finding $F(r,s)$ that corresponds to V^*. We are interested in two things:

- $V^* = \max_\pi\{V^\pi\}$ equivalent to $\max_r\{F(r,s)\}$
- $V^* \cong F(r^*, s)$

10.1.5 Choosing the parameters for $F(r,s)$

If it is possible, we would like to find a vector r such that:

$$\forall s: \; F(r,s) = V^\pi(s)$$

In most cases, it isn't possible since there aren't sufficient computing resources. Therefore, we should discuss the distance between F and V^π. One way to measure this distance is to calculate the minimum square error (MSE):

$$\min_r E_s[(F(r,s) - V^\pi(s))^2]$$

If it is possible to encode V^π using $F(r,s)$, this minimum equals to 0. Otherwise, we try to get as close as possible to $V^\pi(s)$. For linear functions it is sometimes possible to compute $\overrightarrow{r}$ which minimizes the MSE, but generally, it is a difficult task.

Let's look at the problem in the following manner: we try to minimize

$$G(r,s) = E_s[(F(r,s) - V^\pi(s))^2]\ .$$

Finding the global minimum cannot always be done efficiently, and it is much easier to find a local minimum (and hope it is good enough).To find a local minimum, we can follow the derivative of $G(r,s)$. As long as the derivative is non zero, we have a direction, in which $G(r,s)$ is descending. When the value of the derivative equals to zero, we are done. We now only need to establish that this is a minimum (and not a maximum or an inflection point). The reason that this is not a maximum value is that the function descends to this point. However, we could be at an inflection point, and remain there (because the derivative could be fixed on zero in that neighborhood).

$$\overrightarrow{r_{t+1}} = \overrightarrow{r_t} - \frac{\alpha}{2}\nabla_{\overrightarrow{r_t}}[V^\pi(s_t) - V_t(s_t)]^2 = \overrightarrow{r_t} + \alpha[V^\pi(s_t) - V_t(s_t)] \cdot \nabla_{\overrightarrow{r_t}} V_t(s_t)$$

Recall that:

$$\nabla_{\overrightarrow{r_t}} V_t(s_t) = (\frac{\partial}{\partial r_1}V_t(s_t), \ldots, \frac{\partial}{\partial r_k}V_t(s_t))\ .$$

Of course, we don't have access to $V^\pi(s_t)$, but rather to a sample of it v_t (which could be noisy). Thus, we have:

$$\overrightarrow{r_{t+1}} = \overrightarrow{r_t} + \alpha[v_t - V_t(s_t)] \cdot \nabla_{\overrightarrow{r_t}} V_t(s_t)$$

For example, v_t could be a Monte-Carlo estimate, i.e. the total reward from time t onwards R_t.

In the case of backgammon, v_t denotes whether we won or lost the game. Similarly,

$$\overrightarrow{r_{t+1}} = \overrightarrow{r_t} + \alpha[R_t^\lambda - V_t(s_t)] \cdot \nabla_{\overrightarrow{r_t}} V_t(s_t)$$

10.2 TD-Gammon

Let $V^*(s,0)$ be the probability of white winning from state s and it is white's turn (assuming white and black are playing optimally). Let $V^*(s,1)$ be the probability of white winning from state s and it is black's turn.

We estimate $V^*(s,l)$ using a neural network which calculates $\tilde{V}(s,l,r)$.

The Neural Network:

- 198 inputs.
- 40 nodes in the second level.
- 1 output node in the third level.

Network Initialization: (small) random weights.

Training Method: The program plays for both sides. For each point in time we have a state s_t, a vector r_t, and a turn l_t.

For each state s' accessible from s_t (according to the dice), we calculate $\tilde{V}(s', l_t, r_t)$, and choose the best state. (For white's turn choose the state corresponding to the maximum value; for black's - the minimum.)

Updating Parameters: At the end of each turn, we compute:

$$d_t = \tilde{V}(s_{t+1}, l_{t+1}, r_t) - \tilde{V}(s_t, l_t, r_t) \ ,$$

which is the TD (temporal difference). (In the final state we replace $\tilde{V}$ with the game outcome.) In addition, we update $\overrightarrow{r_t}$:

$$\overrightarrow{r_{t+1}} \leftarrow \overrightarrow{r_t} + \alpha \cdot d_t \underbrace{\Sigma_{k=0}^{t} \gamma^{t-k} \nabla_{r_k} \tilde{V}(s_k, l_k, r_k)}_{\overrightarrow{e_t}} \ .$$

At the end of each game a new game is started, and r_0 is set to the previous game's parameter vector.

1. α is set to a constant (determined by experiments).

2. γ does not affect the results significantly in this case.

At the end of the training phase, we get a function $\tilde{V}(s, l, r)$ (r is fixed) which we can use to play backgammon.

Improvements:

- Instead if 40 nodes at the second level, we add 40 more (80 total), the additional 40 units are set to represent important patterns for backgammon.
- After $\tilde{V}$ is set, it can be used immediately (one step), or, alternatively, we can look a few steps ahead, by building a game search tree.

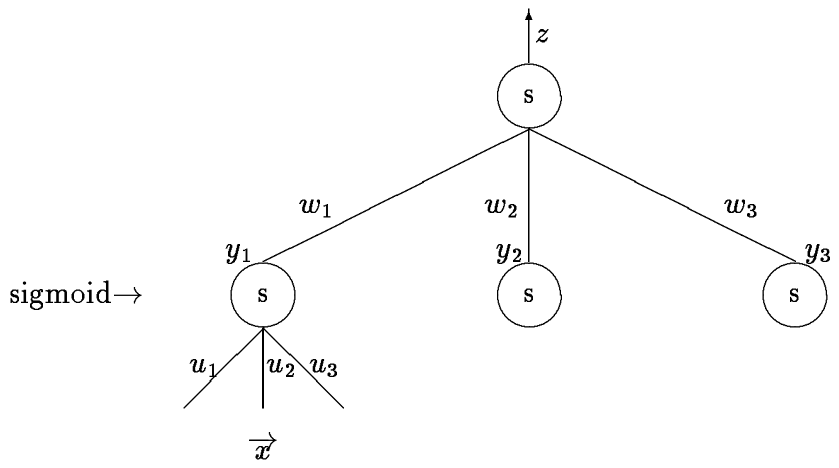

Figure 10.3: Calculating the Derivative of a Neural Network

Comments:

- $\tilde{V}$ wasn't a good estimate for the probability of white winning, but the policy derived from it is a very good one. (the reason for this is as yet unclear.)
- The chosen policy is totally greedy (selected deterministically). However, in backgammon there is a lot of randomness, because of the dice. This enables us to "explore", although we do not perform this explicitly.
- TD-Gammon has achieved a skill level close to the level of the best players in the world today!

10.3 Calculating the Derivative of a Neural Network

Let z be the output of the neural network.
Let $w_1, \ldots, w_l$ be the weights of the inputs from the second level to the output gate.
Let $y_1, \ldots, y_l$ be the outputs of the gates in the second level.
Let $u_{i1}, \ldots, u_{ik_i}$ be the weights of the inputs to gate y_i.
Let $x_1, \ldots, x_k$ be the inputs to the neural network.
(see figure 10.3)

$$z = F(\overrightarrow{x}) = \sigma(\Sigma w_i \sigma(\overrightarrow{u_i} \overrightarrow{x_i}))$$

$$\frac{\partial z}{\partial u_{ij}} = \frac{\partial z}{\partial y_i} \cdot \frac{\partial y_i}{\partial u_{ij}}$$

In this case:

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

$$\frac{\partial}{\partial u_{ij}} \sigma(x) = \frac{e^{-x}}{(1+e^{-x})^2} = e^{-x} \cdot \sigma^2(x)$$

$$\frac{\partial}{\partial y_i} \sigma(\Sigma y_j w_j) = w_i \cdot e^{-\Sigma y_j w_j} \cdot \sigma^2(\Sigma y_j w_j)$$