# Combining Dynamic programming and approximation architectures

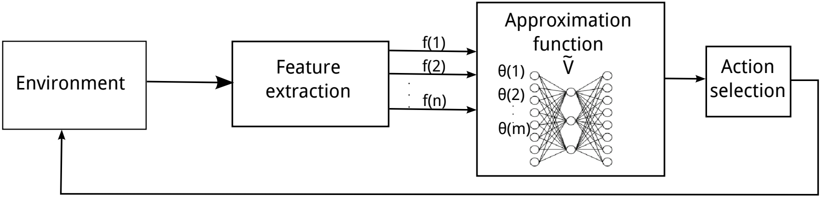
AI & Agents for IET
Lecturer: Liliana Mamani Sanchez

http://www.scss.tcd.ie/~mamanisl/teaching/cs7032/

November 30, 2015

# Combining TD and function approximation

- Basic idea: use supervised learning to provide an approximation of the value function for TD learning
- The approximation architecture is should generalise over (possibly unseen) states




- In a sense, it groups states into equivalence classes (wrt value)

# Why use approximation architectures

- To cope with the curse of dimensionality
- by generalising over states
  - Note that the algorithms we have seen so far (DP, TD, Sarsa, Q-learning) all use tables to store states (or state-action tuples)
  - This works well if the number of states is relatively small
  - But it doesn't scale up very well
- (We have already seen examples of approximation architectures: the draughts player, the examples in the neural nets lecture.)

# Gradient descent methods

- The LMS algorithm use for draghts illustrates a gradient descent method
  - (to approximate a linear function)
- Goal: to learn the parameter vector

$$\overrightarrow{\theta_t} = (\theta_t(1), \theta_t(2), \theta_t(3), \ldots, \theta_t(m)) \tag{1}$$

by adjusting them at each iteration towards reducing the error:

$$\begin{aligned} \vec{\theta}_{t+1} &= \vec{\theta}_t - \frac{1}{2}\alpha \nabla_{\vec{\theta}_t} (V^\pi(s_t) - V_t(s_t))^2 && (2) \\ &= \vec{\theta}_t + (V^\pi(s_t) - V_t(s_t))\alpha \nabla_{\vec{\theta}_t} V_t(s_t) && (3) \end{aligned}$$

where $V_t$ is a smooth, differentiable function of $\vec{\theta}_t$.

# Backward view and update rule

- The problem with (2) is that the target value ($V^\pi$) is typically not available.
- Different methods replace their estimates for this value function:
  - So Monte Carlo, for instance, would use the return $R_t$
  - And the $TD(\lambda)$ method uses $R_t^\lambda$:

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha(R_t^\lambda - V_t(s_t))\nabla_{\vec{\theta}_t} V_t(s_t) \tag{4}$$

  - The backward view is given by:

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha\delta_t\vec{e}_t \tag{5}$$

    where $\vec{e}_t$ is a vector of eligibility traces (one for each component of $\vec{\theta}_t$), updated by

$$\vec{e}_t = \gamma\lambda\vec{e}_{t-1} + \nabla_{\vec{\theta}_t} V_t(s_t) \tag{6}$$

# Value estimation with approximation

**Algorithm 1:** On-line gradient descent TD($\lambda$)

Initialise $\vec{\theta}$ arbitrarily
$\quad\vec{e} \leftarrow 0$
$\quad s \leftarrow$ initial state of episode
repeat (for each step of episode)
$\quad$choose $a$ according to $\pi$
$\quad$perform $a$, observe $r, s'$
$\quad\delta \leftarrow r + \gamma V(s') - V(s)$
$\quad\vec{e} \leftarrow \gamma\lambda\vec{e} + \nabla_{\vec{\theta}} V(s)$
$\quad\vec{\theta} \leftarrow \vec{\theta} + \alpha\delta\vec{e}$
$\quad s \leftarrow s'$
until $s$ is terminal state

- Methods commonly used to compute the gradients $\nabla_{\vec{\theta}} V(s)$:
  - error back-propagation (multilayer NNs), or by
  - linear approximators (for value functions of the form $V_t(s) = (\vec{\theta}_t)^T \vec{f} = \sum_{i=1}^{n} \theta_t(i) f(i)$. (where $(\vec{\theta}_t)^T$ denotes the transpose of $\vec{\theta}_t$)

# Control with approximation

- The general (forward view) update rule for action-value prediction (by gradient descent) can be written:

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha(R_t^\lambda - Q_t(s_t, a_t))\nabla_{\vec{\theta}_t} Q_t(s_t, a_t) \tag{7}$$

(recal that $V_t$ is determined by $\vec{\theta}_t$)

- So the backward view can be expressed as before:

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha\delta_t\vec{e}_t \tag{8}$$

where

$$\vec{e}_t = \gamma\lambda\vec{e}_{t-1} + \nabla_{\vec{\theta}_t} Q_t(s_t, a_t) \tag{9}$$

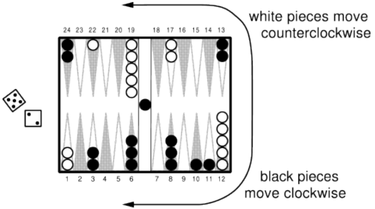
# An algorithm: gradient descent Q-learning

**Algorithm 2: Linear Gradient Descent Q($\lambda$)**

Initialise $\theta$ arbitrarily
for each episode
  $\vec{e} \leftarrow \vec{0}$; initialise $s, a$
  $\mathcal{F}_a \leftarrow$ set of features in $s, a$
  repeat (for each step of episode)
    for all $i \in \mathcal{F}_a$ : $e(i) \leftarrow e(i) + 1$
    perform $a$, observe $r, s$
    $\delta \leftarrow r - \sum_{i \in \mathcal{F}_a} \theta(i)$
    for all $a \in A$
      $\mathcal{F}_a \leftarrow$ set of features in $s, a$
      $Q_a \leftarrow \sum_{i \in \mathcal{F}_a} \theta(i)$
    $\delta \leftarrow \delta + \gamma \max_a Q_a$
    $\vec{\theta} \leftarrow \vec{\theta} + \alpha\delta\vec{e}$
    with probability $1 - \epsilon$
      for all $a \in A$
        $Q_a \leftarrow \sum_{i \in \mathcal{F}_a} \theta(i)$
      $a \leftarrow \arg\max_a Q_a$
      $\vec{e} \leftarrow \gamma\lambda\vec{e}$
    else
      $a \leftarrow$ a random action
      $\vec{e} \leftarrow 0$
  until $s$ is terminal state

# A Case Study: TD-Gammon

- 15 white and 15 black pieces on a board of 24 locations, called points.
- Player rolls 2 dice and can move 2 pieces (or same piece twice)



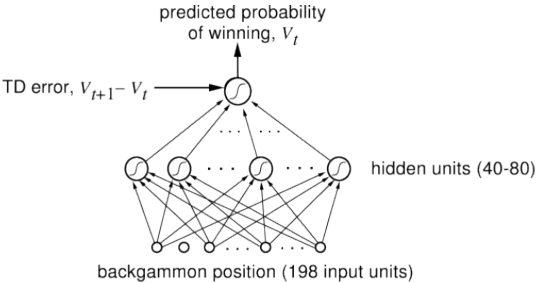

  - Goal is to move pieces to last quadrant (for white that's 19-24) and then off the board
  - A player can "hit" any opposing single piece placed on a point, causing that piece to be moved to the "bar"
  - Two pieces on a point block that point for the opponent
  - + a number of other complications

# Game complexity

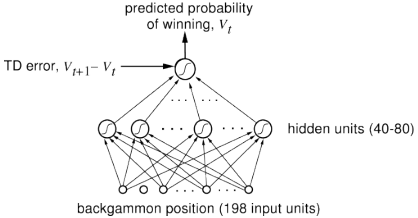
- 30 pieces, 26 locations
- Large number of actions possible from a given state (up to 20)
- Very large number of possible states ($10^{20}$)
- Branching factor of about 400 (so difficult to apply heuristics)
- Stochastic environment (next state depends on the opponent's move) but fully observable

# TD-Gammon's solution

- $V_t(s)$ meant to estimate the probability of winning from any state $s$
- Rewards: 0 for all stages, except those on which the game is won
- Learning: non-linear form of TD($\lambda$)
  - like the Algorithm presented above, using a multilayer neural network to compute the gradients



# State representation in TD-Gammon




- Representation involved little domain knowledge
- 198 input features:
  - For each point on the backgammon board, four units indicated the number of white pieces on the point (see [Tesauro, 1994] for a detailed description of the encoding used)
  - (4 (white) + 4 (black)) x 24 points = 192 units
  - 2 units encoded the number of white and black pieces on the bar
  - 2 units encoded the number of black and white pieces already successfully removed from the board
  - 2 units indicated in a binary fashion whether it was white's or black's turn to move.

# TD-Gammon learning

- Given state (position) representation, the network computed its estimate in the way described in lecture 10.
  - Output of hidden unit $j$ given by a sigmoid function of the weighted sum of inputs $i$

$$h(j) = \sigma(\sum_i w_{ij} f(i)) \tag{10}$$

  - Computation from hidden to output units is analogous to this
- TD-Gammon employed TD($\lambda$) where the eligibility trace updates (equation (9),

$$\vec{e}_t = \gamma\lambda\vec{e}_{t-1} + \nabla_{\vec{\theta}_t} V_t(s_t)$$

  were computed by the back-propagation procedure
- TD-Gammon set $\gamma = 1$ and rewards to zero, except on winning, so TD error is usually $V_t(s_{t+1}) - V_t(s_t)$

# TD-Gammon training

- Training data obtained by playing against itself
- Each game was treated as an episode
- Non-linear TD applied incrementally (i.e. after each move)
- Some results (according to [Sutton and Barto, 1998])

| Program | Hidden Units | Training Games | Opponents | Results |
|---|---|---|---|---|
| TD-Gam 0.0 | 40 | 300,000 | other programs | tied for best |
| TD-Gam 1.0 | 80 | 300,000 | Robertie, Magriel, ... | -13 pts / 51 games |
| TD-Gam 2.0 | 40 | 800,000 | various Grandmasters | -7 pts / 38 games |
| TD-Gam 2.1 | 80 | 1,500,000 | Robertie | -1 pt / 40 games |
| TD-Gam 3.0 | 80 | 1,500,000 | Kazaros | +6pts / 20 games |

# References

Notes based on [Sutton and Barto, 1998, ch 8, 9]. Further details on TD-Gammon can be found in Tesauro's papers [Tesauro, 1994]. Other interesting case studies can be found in [Sutton and Barto, 1998, ch 10] and [Bertsekas and Tsitsiklis, 1996].


Bertsekas, D. P. and Tsitsiklis, J. N. (1996).
*Neuro-Dynamic Programming*.
Athena Scientific, Belmont.

Sutton, R. S. and Barto, A. G. (1998).
*Reinforcement Learning: An Introduction*.
MIT Press, Cambridge, MA.

Tesauro, G. (1994).
TD-gammon, a self-teaching backgammon program, achieves master-level play.
*Neural Computation*, 6:215–219.