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

![Diagram of combining TD and function approximation]

- 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 uses for draughts illustrates a gradient descent method
  - (to approximate a linear function)
- Goal: to learn the parameter vector

\[ \vec{\theta}_t = (\theta_t(1), \theta_t(2), \theta_t(3), \ldots, \theta_t(m)) \]  

by adjusting them at each iteration towards reducing the error:

\[ \vec{\theta}_{t+1} = \vec{\theta}_t - \frac{1}{2} \alpha \nabla_{\vec{\theta}_t} (V^\pi(s_t) - V_t(s_t))^2 \]  

\[ = \vec{\theta}_t + (V^\pi(s_t) - V_t(s_t))\alpha \nabla_{\vec{\theta}_t} V_t(s_t) \]

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 $R^\lambda_t$:

$$\hat{\theta}_{t+1} = \hat{\theta}_t + \alpha(R^\lambda_t - V_t(s_t))\nabla_{\hat{\theta}_t}V_t(s_t)$$  \hspace{1cm} (4)

- The backward view is given by:

$$\bar{\theta}_{t+1} = \bar{\theta}_t + \alpha \delta_t \bar{e}_t$$ \hspace{1cm} (5)

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

$$\bar{e}_t = \gamma \lambda \bar{e}_{t-1} + \nabla_{\bar{\theta}_t}V_t(s_t)$$ \hspace{1cm} (6)
Value estimation with approximation

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

1. Initialise \(\vec{\theta}\) arbitrarily
2. \(\vec{e} \leftarrow 0\)
3. \(s \leftarrow\) initial state of episode
4. repeat (for each step of episode)
   5. choose \(a\) according to \(\pi\)
   6. perform \(a\), observe \(r, s'\)
   7. \(\delta \leftarrow r + \gamma V(s') - V(s)\)
   8. \(\vec{e} \leftarrow \gamma \lambda \vec{e} + \nabla \vec{\theta} V(s)\)
   9. \(\vec{\theta} \leftarrow \vec{\theta} + \alpha \delta \vec{e}\)
10. \(s \leftarrow s'\)
11. 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:

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

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

- So the backward view can be expressed as before:

\[ \vec{\theta}_{t+1} = \vec{\theta}_t + \alpha \delta_t \vec{e}_t \]  

where

\[ \vec{e}_t = \gamma \lambda \vec{e}_{t-1} + \nabla_{\vec{\theta}_t} Q_t(s_t, a_t) \]
Algorithm 2: Linear Gradient Descent Q(λ)

1. Initialise $\theta$ arbitrarily
2. for each episode
   \[\tilde{e} \leftarrow 0; \text{ initialise } s, a\]
   \[F_a \leftarrow \text{set of features in } s, a\]
   repeat (for each step of episode)
   3. for all $i \in F_a$: $e(i) \leftarrow e(i) + 1$
   4. perform $a$, observe $r, s$
   5. $\delta \leftarrow r - \sum_{i \in F_a} \theta(i)$
   6. for all $a \in A$
      \[F_a \leftarrow \text{set of features in } s, a\]
      \[Q_a \leftarrow \sum_{i \in F_a} \theta(i)\]
      \[\delta \leftarrow \delta + \gamma \max_a Q_a\]
      \[\tilde{\theta} \leftarrow \tilde{\theta} + \alpha \delta \tilde{e}\]
      with probability $1 - \epsilon$
   7. for all $a \in A$
      \[Q_a \leftarrow \sum_{i \in F_a} \theta(i)\]
      \[a \leftarrow \arg\max_a Q_a\]
      \[\tilde{e} \leftarrow \gamma \lambda \tilde{e}\]
   8. else
      \[a \leftarrow \text{a random action}\]
      \[\tilde{e} \leftarrow 0\]
   9. 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)
  - \((4 \text{ (white)} + 4 \text{ (black)}) \times 24 \text{ points} = 192 \text{ units}\)
  - 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 \( i \)
    \[
    h(j) = \sigma \left( \sum_i w_{ij} f(i) \right)
    \] (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])

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

Neuro-Dynamic Programming.
Athena Scientific, Belmont.

Reinforcement Learning: An Introduction.
MIT Press, Cambridge, MA.

TD-gammon, a self-teaching backgammon program, achieves master-level play.