Decision Tree Function Approximation in Reinforcement Learning

Larry Pyeatt
Computer Science
Texas Tech University
Lubbock, TX 79409
pyeatt@cs.ttu.edu
Reinforcement learning associates a value with each action over all states, and selects actions that will give the highest rewards.
Q-Learning

In the case of Q-learning, the function $Q$ that is learned is a direct approximation of the optimal action-value function $Q^*$. At each time step, we update the value of the previous state-action pair:

$$
\Delta Q_t(s_t, a_t) = \alpha [r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)].
$$
Common Methods for Representing $Q$

**Table Lookup:** Each dimension of the $s$ vector is divided into discrete intervals, forming a table in which a discrete representation of the $Q$ function is stored.

**Neural Network:** A backpropagation neural network with one input for each dimension of the $s$ vector is used to store the $Q$ function.

**Decision Tree:** A decision tree is defined over the $s$ space with each leaf node representing a region in the space for which a discrete representation of the $Q$ function is stored.
Problem with Table Lookup Reinforcement Learning

**Dimensionality:** The size of the table grows exponentially with $|s|$, leading to very large tables and slow learning or, in the worst case, not enough memory to store the table.

**Fixed representation:** The discretization is usually set to a fixed number of intervals, possibly resulting in better resolution than needed in some areas and inadequate resolution in other areas.
Problem with Neural Reinforcement Learning

**Overtraining:** Once a good control policy is learned, the network only sees states that are on-policy, and overtrains on a small part of the input space.

**Nonlocal Update:** Because non-local changes are made to the approximated value function, the network forgets the correct action to take for states that it does not see very often. This eventually causes it to fail when one of these “forgotten” states is encountered.

The network displays cyclical behavior where it performs well for a time, and then performs poorly until it re-learns the less common states.
DT Function Approximation for $Q(s, a)$

DTRL uses a decision tree to partition the input space. Each leaf node uses reinforcement learning to approximate the value of the state space partition that it represents.

This approach provides robust convergence while avoiding the curse of dimensionality that plagues table lookup reinforcement learning.
Question: When and where should nodes be split?

- Leaf nodes gather information in history lists.

- When the list for a node reaches a threshold length, a test is performed to determine whether the leaf node should be split.

- We investigated four methods of selecting where the node should be split: Information Gain, Gini Index, and Twoing Rule, and t-test.
DTRL algorithm

1. Receive input vector \( v \) and reward \( r_t \) for time \( t \).
2. Use input vector \( v \) to find a leaf node representing state \( s_t \).
3. Select the action \( a \) with the largest value of \( Q(s_t, a) \), or select a random action with some small probability.
4. If the action was not chosen at random, calculate \( \Delta Q(s_{t-1}, a_{t-1}) \) and update \( Q(s_{t-1}, a_{t-1}) \).
5. Add \( \Delta Q(s_{t-1}, a_{t-1}) \) and \( v \) to the history list for the leaf node corresponding to \( s_{t-1} \).
6. Decide if \( s_{t-1} \) should be divided into two states by examining the history list for \( s_{t-1} \).
   (a) if history_list_length < history_list_min_size then split := False
   (b) else
      i. calculate average \( \mu \) and standard deviation \( \sigma \) of \( \Delta Q(s_{t-1}, a_{t-1}) \) in the history list
      ii. if \( |\mu| < 2\sigma \) then split := True
      iii. else split := False
7. Perform split if required using Information Gain, Gini Index, Twoing Rule, or T-statistic to select the decision boundary.
8. Save \( a_t \) and \( s_t \) so that they can be used for training on the next iteration.
9. Return \( a_t \).
DTRL Results for Two Other Domains

Pole Balancing
(high score is better)

Mountain Car
(low score is better)
DTRL Results for Three Domains

Performance during the fourth period: each column shows average and standard deviation.

<table>
<thead>
<tr>
<th>Method</th>
<th>Mountain Car $\mu$</th>
<th>Pole Balance $\mu$</th>
<th>RARS Crashes $\mu$</th>
<th>RARS Times $\mu$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\sigma$</td>
<td>$\sigma$</td>
<td>$\sigma$</td>
<td>$\sigma$</td>
</tr>
<tr>
<td>Gini</td>
<td>206</td>
<td>No result</td>
<td>15.7</td>
<td>3617</td>
</tr>
<tr>
<td>Info Gain</td>
<td>170</td>
<td>2000</td>
<td>1.1</td>
<td>2956</td>
</tr>
<tr>
<td>NN</td>
<td>328</td>
<td>1999</td>
<td>2.8</td>
<td>1172</td>
</tr>
<tr>
<td>Table</td>
<td>160</td>
<td>2000</td>
<td>No result</td>
<td>No result</td>
</tr>
<tr>
<td>T-test</td>
<td>170</td>
<td>2000</td>
<td>0.01</td>
<td>723</td>
</tr>
<tr>
<td>Twoing</td>
<td>219</td>
<td>1848</td>
<td>1.4</td>
<td>1869</td>
</tr>
</tbody>
</table>


Conclusions

Decision tree based reinforcement learning:

- meets our needs for more reliable convergence than the neural network approach,
- has lower memory requirements than the table lookup method and scales better to large input spaces, and
- gave the best performance overall in the domains that we studied.
Future Work

• Use oblique instead of axis parallel decision boundaries. Oblique boundaries lead to smaller decision trees by allowing each node to use several input variables.

• Explore some alternative approaches for selecting the decision boundaries.

• Explore some alternative approaches for deciding when to split a leaf node.