Haskell package for Reinforcement Learning

Gabriel V. de la Cruz Jr.
School of Electrical Engineering and Computer Science
Washington State University
Pullman, Washington 99163
Email: gabrieledcjr@gmail.com

I. Creating Haskell Package

Reinforcement Learning (RL) is a powerful machine learning paradigm that allows agents to autonomously learn a cumulative maximum scalar reward. It uses a trial and error approach where agents learn to model it's environment over multiple episodes. Basically, it learns a optimal policy π^* that maps all possible state to action: $\pi^*(s) = a$.

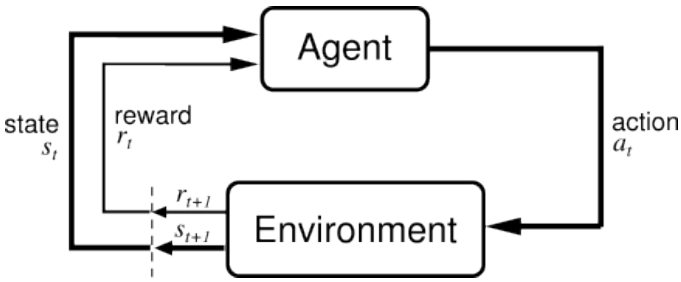

Fig. 1. The agent-environment interaction in RL.

The goal of this project is to develop a Haskell package for three well-known RL algorithms: 1) TD-Learning 2) Q-Learning and 3) SARSA.

Algorithm 1 TD-Learning

Initialize $V(s)$ arbitrarily, π to the policy to be evaluated
Repeat (for each episode):
  Initialize s
  Repeat (for each step of episode):
    $a \leftarrow$ action given by π for s
    Take action a; observe reward, r, and next state, s'
    $V(s) \leftarrow V(s) + \alpha[r + \gamma V(s') - V(s)]$
    $s \leftarrow s'$
  until s is terminal

Algorithm 2 Q-Learning

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

Algorithm 3 SARSA

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 reward, r, s'
    Choose a from s using policy derived from Q (e.g., ϵ-greedy)
    $Q(s,a) \leftarrow Q(s,a)+\alpha[r+\gamma max_{a'}Q(s',a')-Q(s,a)]$
    $s \leftarrow s'; a \leftarrow a';$
  until s is terminal

Although the algorithms 1, 2 and 3 are limited to a tabular implementation of the states; as an extension to the algorithms, we can implement additional parameters to allow problems that deals with continuous states. We can either extend the algorithms to use linear function approximation, or search in *hackage* if there are any existing function approximator we can use.

II. Implementation of Haskell package

To test our Reinforcement Learning Haskell package, we will implement two programs: 1) Grid World and 2) Tic-Tac-Toe. Since time is limited, the goal is just to create a console-based game, but if time permitted, we will extend it to a GUI-based.

A. Grid World

The Grid World's map can be setup from a file that defines the environment. The agent will be have a pre-defined start and end goal. The agent is also limited to four actions: 1) up, 2) down, 3) left and 4) right. The goal is for the agent to be able to determine the optimal policy that brings the agent from start to end.

B. Tic-Tac-Toe

Tic-Tac-Toe is a 3x3 grid where the human agent and the computer agent plays alternately by playing the symbols: 1) 'X' or 2) 'O'. The game ends in two ways: 1) draw or 2) either agent wins by having three consecutive symbols in vertical, horizontal or diagonal pattern. The goal is to create an intelligent computer agent that will mostly win against a human agent or cause the game to a draw.

III. Learning Objectives

1) Learn how to create a Haskell package.
2) Determine how to represent states and actions to be generic as each problem is different.
3) Determine how the reward function in the algorithms can be implemented outside the package as each problem have unique rewards.
4) Determine if Haskell is a viable programming language alternative for machine learning projects.