AI Programming (IT-3105)
Project Module #1:
Deep Reinforcement Learning for Peg Solitaire

Due Date: Thursday, February 18, 2021 (Demonstration Session)

Purposes:

1. Gain hands-on familiarity with Reinforcement Learning (RL) and, in particular, the actor-critic model of RL.
2. Learn to use one of the popular deep-learning systems (Tensorflow or PyTorch) and how to integrate it into an RL system as a function approximator.

1 Introduction

In this project, you will build a general-purpose Actor-Critic Reinforcement Learner and apply it to assorted instances of a puzzle type known as *Peg Solitaire*, *Hi-Q*, or a variety of other names in other languages. For a complete description of the game, see Wikipedia, and for a description relevant (and vital) to this assignment, see the accompanying document, hex-board-games.pdf.

In general, all of the documents listed below (found in the *Materials* section of the course web page) contain critical information for this project:

1. *Board Games on Hexagonal Grids* (hex-board-games.pdf)
2. *Implementing the Actor-Critic Model of Reinforcement Learning* (actor-critic.pdf)
3. *Basic Expectations of the Structure and Behavior of Project Code* (code-expectations.pdf)

In addition, the Python file (splitgd.py), also found in the *Materials* section and mentioned in actor-critic.pdf, may provide assistance.

2 System Overview

In standard terminology, an RL system consists of an *agent* and an *environment*, where the agent houses all of the core RL processes, while the environment contains *everything else.* For example, in an RL-controlled robot, the environment is comprised of the robot's surroundings plus its body and any other elements of its cognitive machinery other than the RL code.

Figure 1 provides a high-level view of the system that you will design and implement for this project. In relation to the classic breakdown of an RL system into *agent* and *environment*, the decomposition for this project consists of a) the RL system (i.e. agent) composed of an actor and critic, and b) the SimWorld, which incorporates the environment and all knowledge about states and their relationships in that environment. As shown in the diagram, it may also house a structure representing the actual player of the game, which is probably a very simple entity, given that the RL agent does most of the *thinking* in this project.

The SimWorld's main jobs include maintaining the current state of the game board, providing the RL system with both initial board states and successor states (a.k.a. child states) of any parent state, giving the reward associated with the transition from one state to another, determining whether a state represents a final state, etc. In general, the SimWorld understands the game of Peg Solitaire (and little else), while the Reinforcement Learner understands the RL algorithms germane to the actor-critic model (and little else). When deciding upon the next action to take from game state s, the actor may consult the critic to get the values of all child states of s. Through learning, the critic will have developed a thorough set of associations between states and values, and these will form the basis of the actor's decision.

The vast majority of these state-value associations will not be known by the SimWorld, but all of them will be **derived** from the basic information (supplied by the SimWorld) concerning the identity of final states, the transitions between states governed by the game-playing operators (e.g. peg hopping), and the concrete rewards associated with all transitions, including (most importantly) those into a final state. In short, the SimWorld provides the raw materials for the Reinforcement Learner, which manipulates and combines them in accordance with the standard RL algorithms to eventually create accurate evaluations for many states and an intelligent *policy* (i.e. mapping from states to actions).

There are many ways to implement the SimWorld, and for this project the only constraint is that the SimWorld and Reinforcement Learner must be very distinct modules such that the SimWorld does not know that it is producing states for a Reinforcement Learner, and the Reinforcement Learner does not make assumptions about the particular game (Peg Solitaire) being played. See the section entitled *The Critical Divide* in code-expectations.pdf for further discussion of the separation between AI system and SimWorld. **Failure to display this modularity can result in significant point during project evaluation.**

3 The Actor-Critic Model

Your code should follow the basic algorithm outlined in *A Generic Actor-Critic Algorithm* in actor-critic.pdf. Discuss any significant deviations from that model with the course instructor before spending time coding them up, as not all variations are acceptable for this assignment.

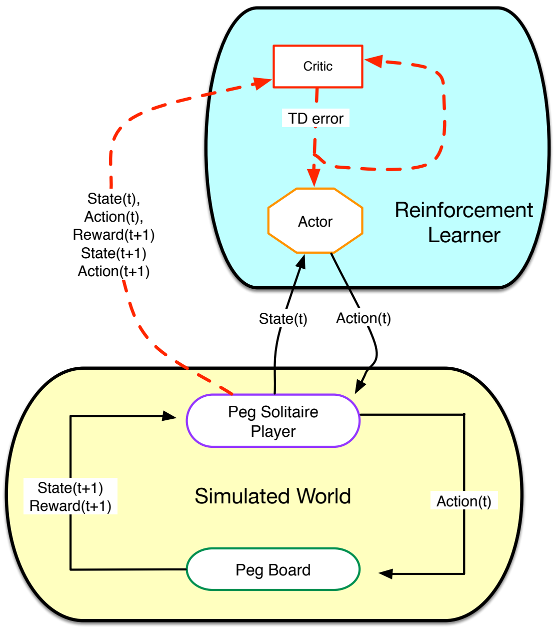

Figure 1: Basic components of an Actor-Critic Reinforcement Learning system for playing Peg Solitaire.

4 The Actor

This project will use *on-policy* RL, meaning that the policy used for searching through state space during each puzzle-solving episode (a.k.a. the *behavior policy*) is also the policy that the system is trying to learn/improve (a.k.a. the *target policy*). Hence, search behavior should change throughout the run (as the policy improves), and hopefully lead to an increased number of solved puzzles toward the end of the run. [1]

To insure a balance between exploration and exploitation, the actor should use its policy in an ϵ-greedy manner (see actor-critic.pdf), where ϵ is either a constant, user-supplied parameter, or a dynamic variable that changes (i.e. decreases) from earlier to later episodes. By setting $\epsilon = 0$ at run's end, the behavior policy essentially becomes the target policy. By displaying one game played with this policy, the user sees the *best* moves that the actor has found for the states of an episode.

The actor's policy should be represented as a table (or Python dictionary) that maps state-action pairs (s,a) to values that indicate the desirability of performing action a when in state s. For any state s, it is wise to normalize the values across all legal actions from s, thus yielding a probability distribution over the possible actions to take from s. The ϵ greedy policy would then choose the action with the highest probability, or a random value.

5 The Critic

Your system must provide two different implementations for the critic: a simple table of assocations (e.g., Python dictionary) between states and values, and a neural network that maps states to values.

[1]Here, a *run* denotes the complete set of episodes simulated by your system.

The neural network must be implemented in one of the standard deep-learning packages, such as Tensorflow or PyTorch. See the section entitled *Function Approximators in the Critic* in actor-critic.pdf for details of this implementation.

6 The Peg Solitaire Interface: Pivotal Parameters

Refering to the secton entitled *The System Interface* in code-expectations.pdf, the following are the *pivotal parameters* for this project:

1. The type of Solitaire board: diamond or triangle.
2. The size of the board. See hex-game-boards.pdf for definitions of *size* for trianguler versus diamond boards.
3. The open cell (or cells) in the puzzle's start state. NOTE: It should be possible to create an initial state with one **or more** holes.
4. The number of episodes that the system will run.
5. Whether the critic should use table lookup or a neural network.
6. The dimensions of the critic's neural network: the number of layers and the number of neurons in each layer. For example, given the specifier (15, 20, 30, 5, 1) as a parameter, your code must be capable of producing a 5-layered network with 15, 20, 30, 5 and 1 node(s) in the successive layers.
7. The learning rates for the actor and critic – you may have separate values for each. When the critic uses a neural network, it is often wise to have a much lower learning rate for the critic than for the actor.
8. The eligibility decay rate for the actor and critic – you may have separate values for each.
9. The discount factor for the actor and critic – you may have separate values for each.
10. The initial value of ϵ for the actor's ϵ-greedy strategy. This value may change during a system run, in which case you may want to include a parameter for the ϵ decay rate.
11. A display variable indicating when actual games will be visualized. Normally, learning should run without visualization – since learning may involve hundreds of episodes – but then when learning finishes, the system should display at least one game, to show that the system has learned to solve the puzzle. For this final display, set $\epsilon = 0$ in the actor such that the completely-greedy strategy is on display.
12. The delay between frames of the game viewer.

7 Displays for Peg Solitaire

At the end of a complete run, your system will need to display two key items:

1. The Progression of Learning - This is a simple plot with the episode number on the x axis and the number of pegs remaining on the board on the y axis. This should be easy to do in matplotlib, for example.

2. A display of all moves made during a game of Solitaire. It suffices to show a single board whose state advances from start to end, with possibilities for varying the display speed. This is best done using a tool for drawing graphs/networks, such as Python's *networkx* (see Figure 2). A command-line print is **not** acceptable for this particular display.

From the progression-of-learning plot, the user should be able to easily assess whether or not the system has converged upon a good strategy. Successful convergence will be indicated by an abundant occurrence of 1's on the y axis for the latter episodes. In addition, if the actor's greedy strategy ($\epsilon = 0$) yields a solution, then your system has learned the proper target strategy.

Figure 2 shows the learning progression for a successful convergent policy. Since the policy is ϵ-greedy, the actor still fails to win the game during some episodes. However, if ϵ is set to 0, the actor will solve the puzzle every time, assuming the same starting board state (i.e. that which the actor trained on).

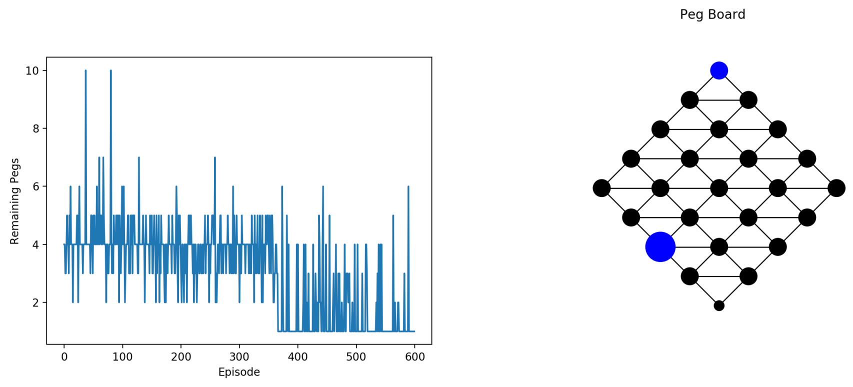

Figure 2: (Left) Plot of the learning progression for a successfully-converged policy for Peg Solitaire. (Right) Graphic display (rendered using Python's networkx) of the final state of a diamond-shaped peg board, where black nodes are empty cells, blue nodes are pegs, the large blue node denotes the peg that most recently hopped, and the small black node represents the cell from which it hopped.

See the "Visualization" section in code-expectations.pdf for important, general information on visualizations for all projects in this course.

8 Deliverables

1. Well-structured code (read code-expectations.pdf carefully) that you can discuss and explain (in general and in detail) with a reviewer on demonstration day. (**20 points**).

2. Show successful convergence (to a perfect solution: 1 peg remaining) on a size-5 triangle puzzle using an initial configuration with one open cell in the *center* (i.e. one of the three cells in the central triangle). Perform this twice: once with a table-based critic, and once with the neural-net based critic. (**6 points**)

3. Show successful convergence (to a perfect solution: 1 peg remaining) on a size-4 diamond puzzle using an initial configuration with one open cell in the *center* (i.e., the 4 cells in the middle of the diamond). As it turns out, this particular puzzle is only fully solveable when the initial hole is in one of two center locations, C_a or C_b; the other 2 center locations do not permit a full solution. You should be able to discover C_a and C_b by experimentation before the demonstration. Again, your system will need to show successful convergence twice: once with a table-based critic, and once with the neural-net based critic. (**6 points**)

4. Show *reasonable behavior* on other versions of Peg Solitaire, where versions may vary with respect to a) the shape (triangle or diamond) and its size, b) the open cell (or cells) in the initial configuration, c) the critic's basis (table or neural network), and d) any of the other pivotal parameters. All versions will be determined by the reviewer of your project. (**8 points**)

   Reasonable behavior means that the system does not crash and shows some indication of policy improvement via learning. For boards larger than 5 (triangle) or 4 (diamond), the expectation is minimal: no crash. However, your system should handle the range of board sizes given in hex-board-games.pdf and variations on the size of the critic's neural network, the value of ϵ, and the learning rates, discount factors, eligibility decay rates, etc.

The 40 total points for this module are 40 of the 100 points that are available for the entire semester.

A zip file containing your commented code must be uploaded to BLACKBOARD prior to the FIRST demonstration time slot on demonstration day. You will not get explicit credit for the code, but it is crucial that we have the code online in the event that you decide to register a formal complaint about your grade (for the entire course).

9 Warning

When coding up AI systems, whether from scratch (as in this class) or using advanced libraries, the difference between error-free functioning of your system and *intelligent* functioning is often very significant. Getting the system up and running (without crashes) may take a week or two, but another few days (or weeks) may be needed to tune the system parameters to successfully solve problems. Unfortunately, the combination of RL and neural networks only widens this gap: it is not uncommon to spend weeks fiddling with parameters. Budget your time accordingly.