CS221 Project Progress Report: Village Game

Fall 2019

Megala Anandakumar
nmegala9@stanford.com

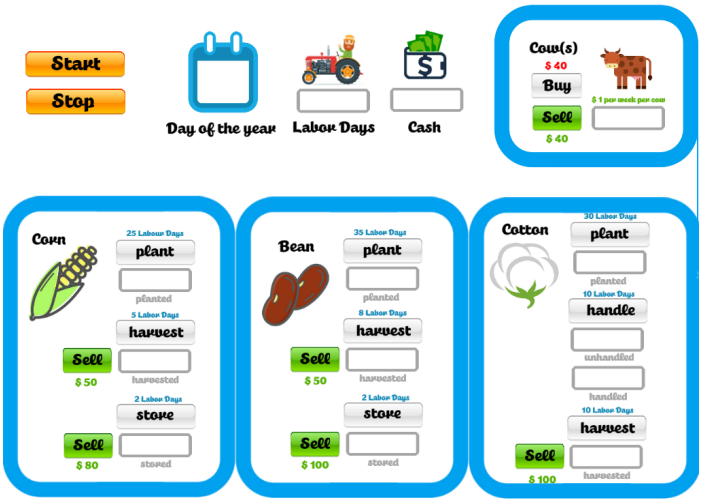
Rakesh Talanki
rakeshTL@stanford.edu

Bohdan Metchko Junior
bohdanjr@stanford.edu

Introduction

Village Game has a rich state space and is aptly suited for a reinforcement learning (RL) approach. By applying variants of the well-known Q-learning algorithm, we seek to deepen our understanding of the problem and solution space. Since our project proposal, we have completed a custom implementation of the Village game using Temporal Difference (TD). In this first version of the game, the problem is simplified so that we can define the project structure using OpenAI Gym and obtain initial results. We have begun experimenting with various reinforcement learning algorithms (epsilon greedy algorithm), function approximators, and feature sets.

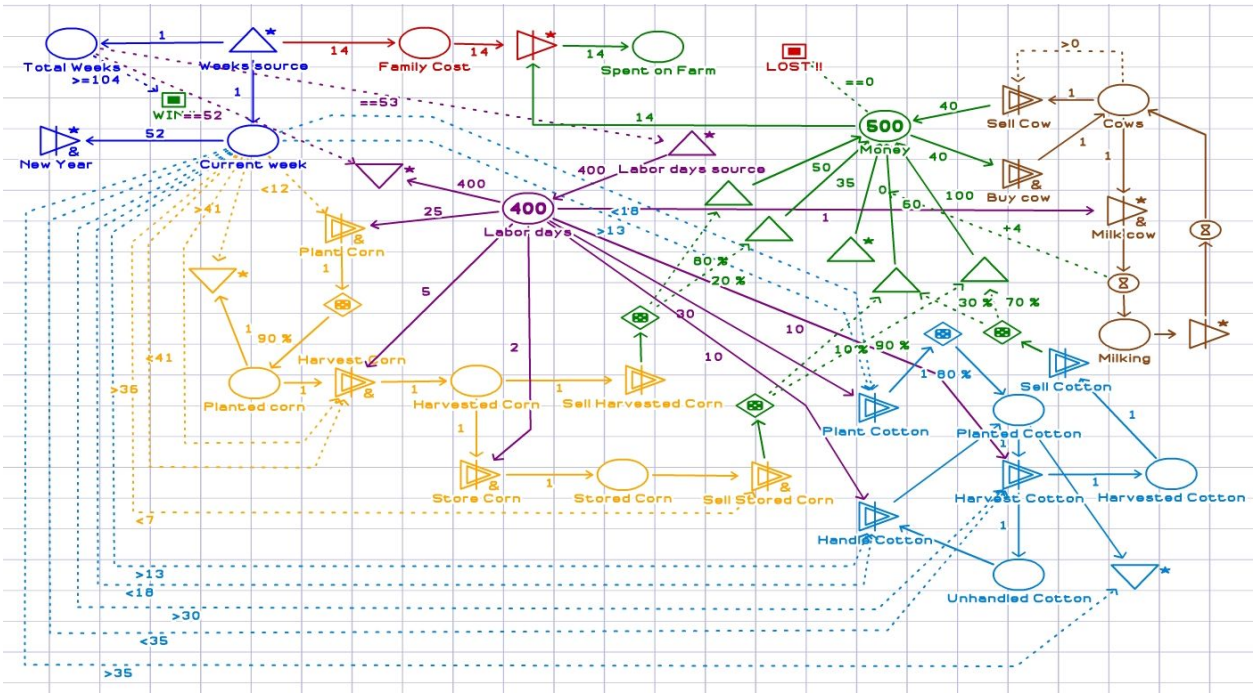
Features



The environment in the village game consists of a family, farm land for bean, farm land for corn, barn for bean, barn for corn, initial money value, and labor units. In order to survive, every week the family needs 1 unit of money value. To make revenue from the farm land, the family can plant corn or bean in the respective farming land. One labor unit is required to plant either one unit of corn or one unit of bean. The planting can happen only during the first 12 weeks of the calendar year. After 3 weeks of the plantation, the corn and bean will be ready for harvesting. One labor unit is required to harvest one unit of bean or corn. The harvested corn or bean will be stored in the barn. The stored unit can be sold in the market at any time. The selling price of the bean or corn is based on the market price. The market price is seasonal and stochastic in nature. The goal of the game is to make more money at the 52 weeks. The game will be played for 52 weeks. The game will come to end either at the end of 52 weeks or when money values becomes less than or equal to zero.

Model

States and Actions



In our first version, the game environment will have the following set-up.

$w = \{1, \ldots..13\}$ - w is the week that can take values from 1 to 13.

$pc_w = \{0,1,2,3\}\ for\ \forall_w$ - pc_w *is the planted* corn at week w that can take a value from 0 to 3 units at any week.

$hc_w = \{0,1,2,3\}\ for\ \forall_w$ - sc_w *is the harvested* corn at week w that can take a value from 0 to 3 units at any week. The harvested corn unit at any week is stored at barn and will be available to sell.

$lu_w = \{1,2,3\}\ for\ \forall_w$ - where lu_w is the labor units value at week w.

$m_w = \{1,2,3\}\ for\ \forall_w$ - where m_w is the money value at week w

The value of N and M can increase the state space to a large value. In order to restrict the state space, the value of lu_w and m_w are grouped into 3 categories.

$luc_w = \{0,1,2\}\ for\ \forall_w$, luc_w is the labor unit category at week w. The group 0 indicates 0 - labor units, 1 indicates the labor unit between 1 to 100 and 2 indicates labor unit greater than 100.

$mc_w = \{0,1,2\}\ for\ \forall_w$ - mc_w money value category at week w. The group 0 indicates 0 – money value, 1 indicates the money value between 1 to 100 and 2 indicates money value greater than 100.

The terminal state of the game is when $mc_w = 0$ at any week w or $w = 13$.

Algorithms

The problem is modeled as a non-adversarial partially observable stochastic game with discrete space. In the given game setup, the agent plays the game with the objective of maximizing the value of money value at the end of the 13 weeks. Q-learning technique is attempted to learn the game to maximize the value of money value. In Q-learning[1] applying Temporal Difference of order 0 ($TD(0)$) for control problem involves using generalized policy iteration using greedy action selection. Q is

used in-place and improve policy after every change. We want to use Q because it is indexed by state (s) and action (a).

The main theme in the Q learning is generalized policy iteration (which includes policy evaluation and policy improvement). Q learning is the off-policy method where random action is taken and still Q^* is obtained. This approach required 5 tuples (s, a, r, s', a') to update the Q value and it is called as SARSA. Similar to Monte Carlo SARSA requires to know $Q(s, a)$ for all a in order to choose argmax of Q. If we use the deterministic policy using argmax, only 1/|A| values are updated in each episode and most of Q are untouched. To avoid this epsilon-greedy update is used that includes both exploitation and exploration.

$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\left[r_{t+1} + \gamma max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\right]$
where $Q(s_t, a_t)$ *is the value of action − state pair*
$s_t -$ *game state*
$a_t -$ *action at time t*
$\alpha -$ *is the learning rate*
$r -$ *is the reinforcement value* (*return value*)
$\gamma -$ *is the discount factor*
The above equation can be written with simple notation
$Q(s, a) \leftarrow Q(s, a) + \alpha\left[r + \gamma max_{a'} Q(s', a') - Q(s, a)\right]$ where $s,\ a$ represent the current state s_t, a_t and s', a' represents the next state s_{t+1} , a_{t+1}

In Q learning the value of $Q(s, a)$ is defined arbitrary for all sates 's' except the terminal state. of $Q(terminal, a) = 0$. For a given number of iteration N, the game is played and $Q(s, a)$ is updated. In a given game, for a chosen state s, action a is chosen from epsilon greedy approach. The reward r and the next state s' are updated for the chosen a. Then the $Q(s, a)$ is updated
$Q(s, a) \leftarrow Q(s, a) + \alpha\left[r + \gamma Q(s', a') - Q(s, a)\right]$ This process is repeated until the game is over. The learning process is repeated for the specified number of iterations N. The learning process will converge as we use epsilon greedy approach where the hyperparameter epsilon ξ is used to choose the action a and learning rate α is used.

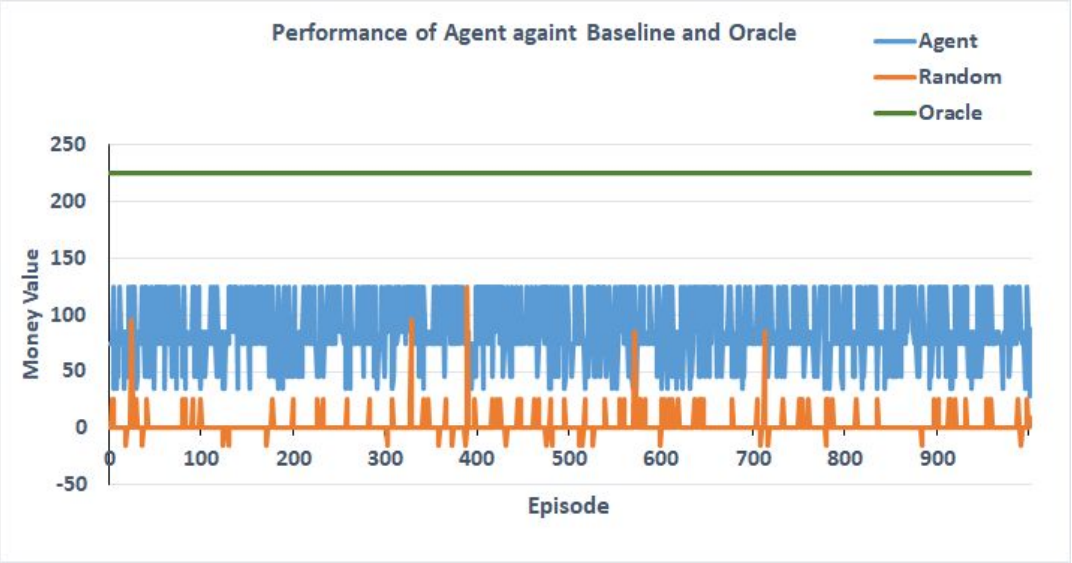
Q-learning pseudocode

Initialize $Q(s,\ a)$ arbitrarily
Repeat the following for N episodes:
    Initialize s
    Repeat the following for each step in episode n:
        Choose action a from s using $\xi - greedy$ approach
        // With probability ξ*, choose random from Actions(s) - Explore,*
        // With probability 1 - ξ*, choose arg max* $_{a \in Actions(s)}$ Q_{opt}*(s,a) - Exploit*
        Take action a, observe $r,\ s'$
        Update
        $Q(s, a) \leftarrow Q(s, a) + \alpha\left[r + \gamma max_{a'} Q(s', a') - Q(s, a)\right]$
        $s \leftarrow s'$
    Until s is terminal

Implementation

Following is our first version of Village game implemented using Python3 on OpenAI gym platform. https://github.com/rakeshtl/artificial_intelligence/tree/master/Village-Gym [2][3]

Preliminary Results



We trained the game over 34,000 episodes with $\xi = \{0,1\}$, $\alpha = 0.2$ and $\gamma = 1.$, The domain of ξ is also changing during the training with learning beginning at ($\xi = 1$) and exploiting to ($\xi = 0$) at the end. We then allowed the agent to play the game. We observed the average reward over 1000 episodes is $87.40 with standard deviation equal to 28.42. It means that in every game played the agent managed to reach the end with certain amount of money. Our Oracle solution where a good player would reach the end of the game with $225. However it is better than a random player (choosing random actions) who after 1000 played episodes reached an average of $1.95 with standard deviation equal to 9.67. As such we can conclude that the agent played better than a random player but still has the potential to improve to play as good as Oracle solution.

References

1. Practical Reinforcement Learning — 02 Getting started with Q-learning
2. Making a custom environment in gym
3. gym-tictactoe

Next Steps

In the next version of the game our plan is to generalize the code to extend the game to 104 iterations (corresponding to a period of two years) , introduce more crops and try different variants of Q-learning algorithm. And, of course, improve the reward as well as minimize the standard deviation from episode to episode.

In the final version of the game our plan is to have all the planned crops like corn, beans, cotton and also have cow which will earn money to the farmer. In this final version we hope our agent can be as close to the Oracle winning the game. In order to do so, we will be trying different algorithms and choose the best algorithm to maximize the reward.