Reinforcement Learning for Village Game

Megala Anandakumar nmegala9@stanford.com
Rakesh Talanki rakeshTL@stanford.edu
Bohdan Metchko Junior bohdanjr@stanford.edu

Stanford University
Project Paper

Abstract

In the present study, reinforcement learning agents are created for a popular strategy-based board game called Village Game [1]. The game environment consists of a family in a village, farming land for different crops, barns to store the harvested crops, markets to sell the crops, initial cash in hand. Player can play the game for a specified number of weeks by taking different action: plant crops, harvest crops, sell crops in the market. In addition to the actions taken, family expenses are incurred every week. Stochasticity is involved in every action. The objective of the game is to find the optimal strategy that would maximize the value earned at the end of a game episode. Reinforcement learning models: Q-Learning, Function approximation, 3 different Deep Q Networks are developed for a computer agent to learn and play the game.

1. Introduction

Motivation for the present work is that providing practical education in a game environment can promote social equality, and can have a profound effect especially when done at a young age. The village game when developed completely can teach the children the responsibilities of a family and how various decisions can affect their livelihood. In addition. We want to pick a problem in reinforcement learning space using Q-Learning technique applying what is taught in CS221 course.

Over the last few years, Q-learning and Deep reinforcement learning (DRL) techniques have seen great success in solving many sequential decision problems involving high-dimensional state-action spaces. Deep learning (DL) techniques allow Reinforcement learning (RL) agents to better correlate actions and delayed rewards by modeling the relationship between states, actions, and their long-term impact on rewards, leading to better agent generalization. The goal of the present study is to build a RL agent for playing Village Game. It is an interesting topic to explore in that when considering the optimal reward, there are constraints on the crops where certain actions can be done only in certain periods of the year (such as planting and harvesting) and the reward is determined stochastically which relates to real life where the price of the crop can depend on many factors outside the purview of the farmer. The problem is modeled as a Markov Decision Process (MDP), and five different reinforcement learning agents: Q-Learning, Function Approximation, three Deep Q-Networks (DQN) are created to play the game. Using the current state of the game table as an input at each time step (week), agent takes optimal action to develop a good policy that leads to achieve high long-term reward and maximize value.

2. Literature Review

Strategy games are one of the important and difficult subclasses of video games. Various algorithms are addressed in the literature for strategy-based games. One of the most commonly used approach to solve strategy games optimally is dynamic programming. Smith [2] has done a literature survey on the application of dynamic programming for various strategy-based board games. The large state space and action set, uncertainty about game conditions as well as multiple cooperative and competitive agents make strategy games realistic and challenging [3]. Amato and Shani [3] developed a single agent reinforcement learning approach to learn a policy for switching high-level strategies under the fixed opponent strategy assumption. Vincent-Pierre [4] attempted different reinforcement learning models: SARSA, SARSA(λ), and Q learning, and Policy Gradient for classic Atari 2600 game. Tapio [5] applied reinforcement learning in a turn-based strategy game called King of Thule. The author demonstrated the application of AI based reinforced learning approach to play an adversarial complex strategy-based game.

The DQN is the recent advancement in deep learning networks [6]. Mnih, et.al., [6] developed an agent called deep Q-network that can learn policies directly from high-dimensional sensory inputs. The deep Q-network was learned for classic Atari 2600 games. The authors reported that the deep Q-network performance surpasses all previous algorithms and achieved a level comparable to that of a professional games-tester across a set of 49 games. Zarkias et. al [7] proposed a deep reinforcement model to predict the behavior of the financial markets that are extremely volatile and unstable.

The first village game is created by Stephan Schmitt-Degenhardt [1]. The original version was officially released in July 1995 in Fiji, Vanuatu, Tuvalu and Western Samoa under the name "Navunavici". For international distribution, the game is named Village. The game has been introduced in several countries by DED - Deutsche Entwicklungs Dienst. Many countries including South Africa, Nepal, Sri Lanka and Brazil have adopted the game to promote development using the local culture. In the present work we have proposed various reinforcement models: Q-learning, Function approximation, DQNs for the village game. Ernest and Dormans [8] developed Machinations gaming tool to design and prototype the mechanics of games, and introduced 5 different types of game mechanics for more common video game genres. In the present work prototype of the village game is created using Machinations gaming tool [8], and the game is played manually to get the Oracle score.

3. Problem Description

The details of the village game, constraints & stochasticity in the game and MDP are discussed in this section.

3.1 Problem Formulation

In the original village game, the farmer has to decide to plant corn, bean or cotton at a time. Only during certain times of the year the farmer can plant, harvest, store and sell. The farmer starts with certain dollars, has certain labor units to spend and has to take actions during each episode to maximize the earnings. The farmer can also decide to buy cows. The milk from cows can be sold to earn additional money. Like in real life there is an expense to raise the family. At any point in the game when the farmer has no money the farmer loses the game. Our objective is to build an agent using reinforcement learning techniques to

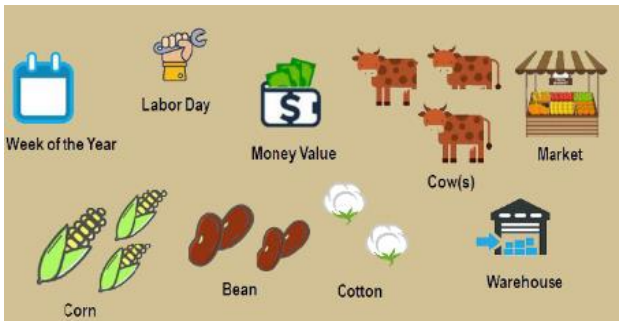

Figure 3.1: Assets in Village Game

maximize expected utility (in this case the earnings) by finding an optimal policy and coming close to human achievable solution also called as oracle solution.

3.2 Game Details

Defining a simplified version of the games as our MDP model, following are the definitions of states, start state, actions, rewards, end states, extra info and discount factor:

- States: $[week, plantedcorn, harvestedcorn, moneyvalue, labordaysvalue\ ]$
- $s_{START} \in$ States: [0,0,0,3,3]
- Action $a \in$ Actions: (0,1,2,3)
- $R_{(s,a,s')} = 0 + 1[week == 13] - 1[money == 0]$
- $IsEnd(s)$: $[week == 13] \vee [money == 0]$
- $info = [money, labordays]$
- $0 \leq \gamma \leq 1$: discount factor (0.95)

In the given game environment, $week$ is the current week the player is in, $plantedcorn$ is total units of planted corn (up to 3), $harvestedcorn$ is the total units of harvested corn (up to 3), $moneyvalue$ is a qualitative indicator of the amount of money the player has (poor, medium, rich), and $labordaysvalue$ is an indicator of the amount of labor days the player has (low, medium, high). Instead of letting the player acts for an entire year, the reduced game allows to play during 1/4 of the year (13 weeks) and there is only one crop: corn. Then the permitted actions (one per week) are the following:

- 0: idle (do nothing)
- 1: plant corn (permitted between week 1 and 3 with a cost of 25 $labordays$)
- 2: harvest corn (permitted between week 6 and 8 with a cost of 5 $labordays$
- 3: sell corn (permitted between week 9 and 13)

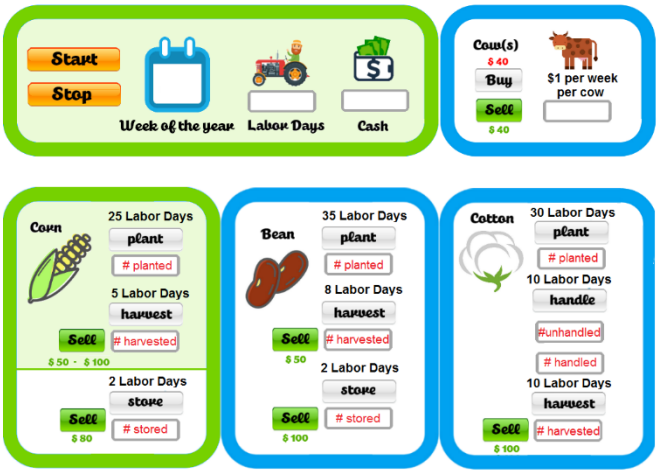

Figure 3.2: Village Game Environment Original game (blue), and simplified version (green)

At the beginning of the game, the player has an initial amount of \$250, 100 $LaborDays$, \$25/week to maintain family played for 13 weeks. Reward +1 indicates that the player reached the end of the game with a certain amount of money (Win Situation), reward -1 indicates that the player lost their money (Loose Situation) and a 0 reward indicates player continues to play. The price of corn when sold is stochastically defined by following probability:

- Value of 50 (50%) or 100(50%) in week 9;
- Value of 60 (40%) or 100(60%) in week 10;
- Value of 70 (30%) or 100(70%) in week 11;
- Value of 80 (20%) of 100(80%) in week 12;
- Value of 90 (10%) or 100(90%) in week 13;

In OpenAI gym environment, we have added additional details: Initial total amount of money; total $LaborDays$.

4. Solutions

In the present work, we proposed 5 solution methods for the strategy-based village game.

- Q-Learning including TD & Epsilon Greedy
- Function Approximation
- DQN-1: Deep Q-Network with the combination of state and action as a model input
- DQN-2: Deep Q-Network with state as a model input
- DQN-3: Deep Q-Network with state as a model input with the increased game scope (game with 3 crops).

In order to solve the problem, the modeling, learning and inference paradigm is adopted. OpenAI gym is used for Q-Learning (TD) and Function Approximation. Keras library is used to develop DQN-1. The DQN-2 and DQN-3 models are developed from the scratch in python using the OpenAI gym model architecture.

4.1 Q-Learning

Q-learning technique is attempted to learn the game. It is an off-policy method that includes policy evaluation and improvement to learn an optimal policy to maximize the earnings of the player. TD Learning is very similar to Q-Learning, but it is an on-policy (where usually the policy is set to be the best policy π(s)). Both are used in training, and the focus in on TD-learning. To learn the Q-value, which is the expected total future reward by taking a certain action at a certain state s, we iteratively update the Q-value by means of probing the environment and receiving reward r and new state s'. Both Q-learning and TD-learning learn from the same data. The Q-value in Q-Learning, is updated with the following equation upon each learning episode

$$\hat{Q}_{opt}(s,a) \leftarrow (1-\eta)\hat{Q}_{opt}(s,a) + \eta(r + \gamma\hat{V}_{opt}(s'))$$
$$\hat{V}_{opt}(s') = max_{a' \in Actions(s')}\hat{Q}_{opt}(s',a')$$

where $\hat{Q}_{opt}(s,a)$is the current estimate of the Q-value, s is the current state, a is the current action, r is the observed reward received by taking action a at state s, s' is the transitioned next state, η is the learning rate, γ is the discount factor of the rewards and $\hat{V}_{opt}(s')$ is the estimated optimal value. In TD-learning, a very similar equation is used (update a value for each state), however It is necessary to know the next state, what is feasible, since the rules of the game are known:

$$\hat{V}_{\pi}(s) \leftarrow (1-\eta)\hat{V}_{\pi}(s) + \eta(r + \gamma\hat{V}_{opt}(s'))$$
$$\hat{V}_{opt}(s') = max_{a' \in Actions(s')}\hat{V}_{\pi}(s')$$

where $\hat{V}_\pi(s)$ is the current estimate of Value, s is the current state, r is the observed reward received by choosing policy π(s) (in our case the best policy), s' is the transitioned next state, η is the learning rate, γ is the discount factor of the rewards and $\hat{V}_{opt}(s')$ is the estimated optimal value (best policy). If we choose the actions deterministically, not too many values are updated and most of the Q (or V) are untouched. To avoid this situation epsilon-greedy update is used with a dynamic value of epsilon which allows to explore in the beginning (random value with probability ε) and exploit towards the end of learning (optimal Q-function with probability 1 - ε). The epsilon-greedy policy is defined as:

$$\pi_{act}(s) = \begin{cases} \arg max_{a \in Actions(s)} \hat{Q}_{opt}(s,a) & probability\ 1-\varepsilon, \\ random\ from\ Actions(s) & probability\ \varepsilon. \end{cases}$$

To obtain the optimum policy π_{opt}, the hyperparameters values are the following: η = 0.01, γ = 0.95 and ε = 0.03. It takes 10,000 episodes to obtain Q-values, and a look-up table (Q-Table) keeps the value for each discrete state-pair (Q-learning) as well as the number of times it has been updated. If the algorithm is changed to TD-learning we measure how good is to be in a state compared to Q Learning which measures how good an action is to take in a state. The values in Q-Tables are values for a particular state. The Q-Table is then used to do inference (playing the game). After playing 1,000 episodes, the averaged value of earnings (average expected utility) is around 184, which corresponds to almost 82 % of oracle's value.

4.2 Function Approximation

Q-Tables works well on small number of states and actions. And it does not extend to unseen states thus another approach is necessary. Unlike Q-learning (and TD-learning), Q-learning with Function Approximation deals with generalization, one of the most important aspects of learning.

Function approximation parametrizes $\hat{Q}_{opt}$ by a weight vector and a feature vector just like linear regression: $\hat{Q}_{opt}(s,a;w) = w \cdot \Phi(s,a)$. The algorithm is given by:

$On\ each\ (s,a,r,s')$:
$w \leftarrow w - \eta[\hat{Q}_{opt}(s,a;w) - (r + \gamma \hat{V}_{opt}(s'))\Phi(s,a)]$
$\hat{V}_{opt}(s') = max_{a' \in Actions(s')} \hat{Q}_{opt}(s',a';w)$

$Feature\ Vector\ \emptyset(x) \in \mathbb{R}^d$

action, money	: 1.0
action, plantedcorn	: 1.0
action, plantedcorn, cornprice	: 1.0
action, harvestedcorn	: 1.0
action, harvestedcorn, cornprice	: 1.0
action, harvestedcorn, money	: 1.0

Figure 4.2.1: Definition of the feature vector for function approximation

Features are supposed to be the properties of state-action (s,a) pairs that are indicative of quality of taking action a in state s. The first approach is to consider directly each (s,a) pair as a feature. After training the model the agent stores weights for each feature (s,a), and it is possible to see that higher weights indicates better actions: The figure shows Learned values of weights for pairs of states and actions (s,a).

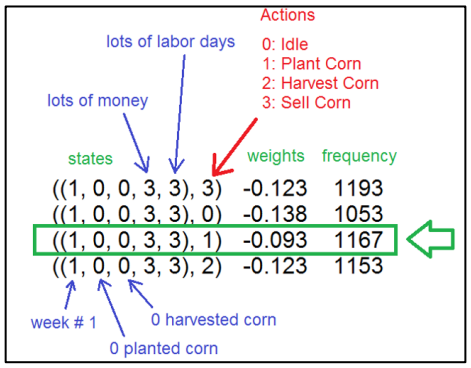

Figure 4.2.2: Weights for different actions in same state

It is possible to see by the example illustrated in the figure that for state s = [1,0,0,3,3], the best action is to plant (related best weight). It does make sense since it is the beginning of the game (week = 1), and the player is rich ($moneyvalue$ = 3).

To obtain the optimum values for weights, the training process takes 10,000 episodes with η = 0.03, γ = 0.95 and ε = 0.025, and it is a little bit unstable (too much variation). After playing 1,000 episodes, the averaged value of earnings (maximum expected utility) is around 150, which corresponds to almost 67 % of oracle's value.

This confirms that the defined set of features is a good one however it still lacks the extensibility. In other words, there is a scope for improvement. If the feature vector is changed to more meaningful ones, it is possible to observe better results.

To obtain the optimum values for weights, the training process takes 10,000 episodes with η = 0.01, γ = 0.95 and ε = 0.06, and it very stable (little variation). After playing 1,000 episodes, the averaged value of earnings (maximum expected utility) is around 174, which corresponds to almost 78 % of oracle's value. We can infer that all states with similar features will have similar Q-values.

4.3 Deep Q-Networks (DQN)

Deep Q Learning Networks (DQN) are used to approximate Q-Value function. DQN models overcome the problem of defining very large size of state space Q table in Q-learning. The input layer in DQN represent a state of the village game, and the output layer represents the Q-value of all possible actions for the given input (which allows the agent to choose the best action). The next action for a given state is determined by the maximum output of the DQN-network. During the learning process of the DQN network, for a given state in the entire state space, the network predicts its best possible actions. The DQN updates its gradient using backpropagation to converge to optimal weights. The challenge in the learning of DQN is the target is continuously changing with each iteration, whereas the target is not changing in Deep Learning. We propose three 3 DQN models in the present work.

1. DQN-1: DQN (State, Action) 1 Crop Model
2. DQN-2: DQN (State) 1 Crop Model
3. DQN-3: DQN (State) 2 Crop Model

4.3.1 DQN-1: DQN (State, Action) 1 Crop Model

DQN-1 consists of one embedding layer, 3 dense layers and one output layer. The entire combination of input and action (2227680) is defined as the input to the network. The embedding layer converts the input space into 10 embedded units. Each dense layer has 50 neurons and the output layer has 204 neurons. The output layer represents all possible action for a given state. The out of the DQN-1 is the Q function that is defined as a linear combination of input and output layers.

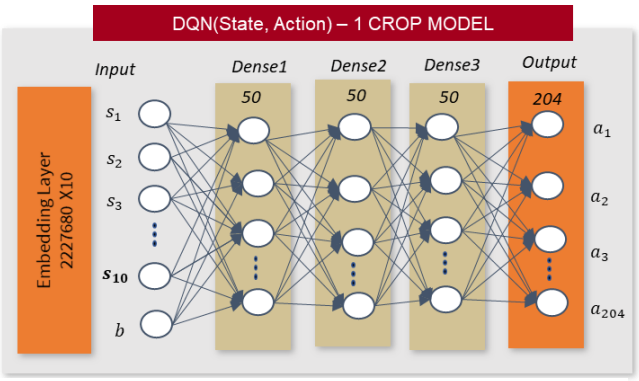

Figure 4.3.1: Architecture of DQN-1 (state, action)

The target value of each output unit is a scalar

$$target_s = \begin{cases} r & if\ state\ is\ terminal \\ r + \gamma Q(s', a') & otherwise \end{cases}$$

Where target is the final target value at a given state s;

$r - reward;\ \gamma\ - discount\ factor;$

$Q(s', a')\ - max\ return\ for\ state\ s',\ and\ action\ a'$

The output layer is a vector of size $|A|$.

The hyperparameters of DQN-1: Optimizer - Adam optimizer with learning rate 0.001, & error metric mae. Activation function in the hidden layers: $relu$ & output layer: $linear$; The learning process took more than 12 hours. After the learning, the agent played for 1,000 episodes, and the averaged value of earnings (maximum expected utility) achieved was around 150, which corresponds to almost 67 % of oracle's value.

4.3.2 DQN-2: DQN(State) 1 Crop Model

DQN-2 model consists of only input and output layers. The input layer represents state s of the game with one bias unit and the output layer represents all possible action a for the given state. Q function is defined as a linear combination of input and output layers. The state of the village game is defined as:

$$S = [money\ value, labor\ units, week, \\ corn\ price, planted\ corn, \\ harvested\ corn, stored\ corn]$$

The set of possible all actions is:
$A = [Hold, Plant, Harvest, Sell]$.

The DQN-2 architecture is defined as follows: $W \in R^{7X4}$; $B \in R^4$; $S \in R^4$; $A \in R^7$ and $Q(S,:) = W^T S + B$. Reward is defined as a function of wealth gained by moving to next state s' by action a. The wealth is calculated as:

$$Wealth = M + 1.2 * UC,$$
$$UC = unit\ of\ planted\ corn + unit\ of\ corns\ in\ barn.$$

where UC is the total unit of corn.

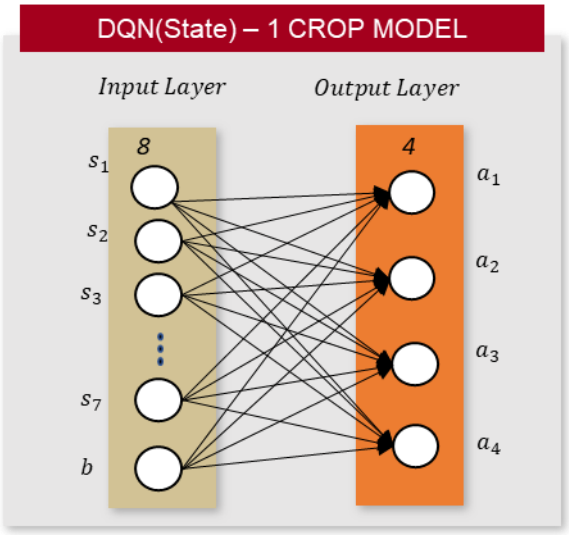

Figure 4.3.2: Architecture of DQN-2 (state)

DQN-2 uses Adam Optimizer with Stochastic Gradient Descent (SGD). The momentum term is used in SGD to speed up the training of the network significantly. The weights are updated as follows:

$w(t) \leftarrow w(t-1) + v(t)$; and
$v(t) \leftarrow \mu v(t-1) + \eta g(t)$;
$\mu = momentum\ term$ (0.99) and
$\eta = learning\ rate$ (0.0001).

One episode of the game is consisting of 13 weeks and the network is trained for 10,000 episodes to learn the best possible combination of weights. Once the network is trained the agent plays the game for 1000 episodes.

4.3.3 DQN-3: DQN(State) 3 Crop Model

DQN-3 model consists of only input and output layers. The architecture of DQN-3 is similar to DQN-2 where input layer represents state s of the game with one bias unit and the output layer represents all possible action a for the given state.

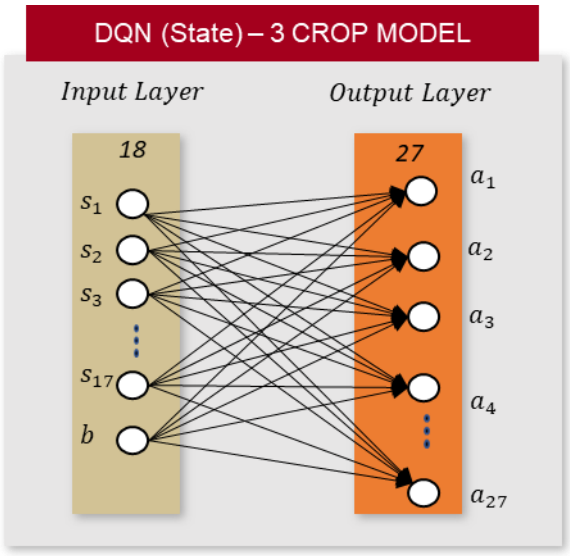

Figure 4.3.3: Architecture of DQN-3 (state) 2

The MDP of the village game for 3 crops is defined as follows:
The game observations: week, family expenses, plant age. Basic action of the game is: plant, harvest, and sell. Hence the Number of possible actions is defined as 3^3 = 27, this will not tell how many to buy or sell. The state of the game is defined as a set of Money Value, Labor Units, Market Prices of each crop, and planted crops of each type of crops in barns.

$M - money\ value$; $L - Labor\ units$;
$CP - Corn\ Price$; $BP - Bean\ Price$;
$CTP - Cotton\ Price$
$PC1 - Planted\ corn\ at\ week\ 1$;
$PC2 - Planted\ corn\ at\ week\ 2$;
$PC3 - Planted\ corn\ at\ week\ 3$
$PB1 - Planted\ bean\ at\ week\ 1$;
$PB2 - Planted\ bean\ at\ week\ 2$;
$PB3 - Planted\ bean\ at\ week\ 3$
$PCT1 - Planted\ cotton\ at\ week\ 1$;
$PCT2 - Planted\ cotton\ at\ week\ 2$
$PCT3 - Planted\ cotton\ at\ week\ 3$;
$SC - Stored\ corn$; $SB - Stored\ bean$;
$SCT - Stored\ cotton$
State $(S = \{s: (1,2,3 \ldots, 17)\})$;
Action $A = \{a: (1,2,3 \ldots, 27)\}$.

Reward is defined as a function of wealth gained by moving to next state s' by action a. The wealth gained is the difference between the wealth in previous state and the current state. The wealth is calculated as

$Wealth = M + 1.2 * UC + 1.4 * UB + 1.6 * UCT$. where $M : Money$ which is the cash in hand; UC is the total unit of corn which is $unit\ of\ planted\ corn + unit\ of\ corns\ in\ barn$; UB is the total unit of beans which is $unit\ of\ planted\ beans + unit\ of\ beans\ in\ barn$; and UCT is the total unit of cottons which is:

$UCT = unit\ of\ planted\ cottons + unit\ of\ cottons\ in\ barn$.
$reward\ r = wealth' - wealth$;
$Wealth = U^T P$;
$U = vector\ of\ corp\ units\ available$, and
$P = vector\ of\ market\ prices\ of\ the\ corps$.

DQN-3 uses Adam Optimizer with Stochastic Gradient Descent (SGD). The momentum term is used in SGD to speed up the training of the network significantly. The gradient updates of DQN-3 is same as DQN-2. One episode of the game is consisting of 13 weeks and the network is trained for 10,000 episodes to learn the best possible combination of weights. Once the network is trained the agent plays the game for 1,000 episodes.

5. Experiments and Analysis

Experiments set up were created in order to make all our agent-based models (Q-Learning, Function approximations, DQNs) to learn from the game environment. The hyperparameters of the models are tuned to achieve the best performance during learning period. The cash in hand at the end of the episode (value of the game) is measured during the learning process of the agents. The learning period for DQN-1 is 500,000 episodes, and for the remaining models the learning period is 10,000 episodes. The average value of the game at the end of every 50 episodes are computed to measure the average performance of the agents during the learning period. After the learning process, the agents of the all the models are allowed to play for 1000 episodes. The value of the game at the end of each episode and the average value of the game at the end of every 50 episodes are measured.

5.1 Q-Learning

The hyper parameters tuned for the Q-Learning agent are: Number of Episodes *N*; Epsilon for exploration vs exploitation ε; Discount Factor γ; Learning rate η. The search space of the hyperparameters, best parameters learned from the Q-learning agent are given in Table 5.1.1. Agent received the best value (case in hand) of $225, and the average value of $184 during the learning process.

Table 5.1.1 Hyper parameters we tuned to achieve the final result.

#	Hyper Parameters Tuned	Avg $
1	η = 0.4, ε = 0.08, N = 17000, γ = 1.00	85
2	η = 0.2, ε = 0.04, N = 17000, γ = 1.00 e < ε episode_rate	120
3	η = 0.01, ε = 0.03, N = 10000, γ = 0.95 e < ε /(1.2 * episode_rate)	184

5.2 Function Approximation

The hyper parameters tuned for the function approximation agent (using custom features) are: Number of episodes N; Epsilon for exploration vs exploitation ε; Discount Factor γ; Learning rate η. The search space of the hyperparameters, best parameters learned from the Function approximations are given in Table 5.2.1. Agent received the best value (case in hand) of $225, and the average value of $174 during the playing process (custom features).

Table 5.2.1 Features tuned to achieve the final result.

#	Features Evaluated	Hyper Parameters Tuned	Avg $
1	(state, action)	η = 0.03, ε = 0.025, N = 10000, γ = 0.95	$151
2	(action, money) (action, plantedcorn) (action, harvestedcorn) (action, plantedcorn, cornprice) (action, harvestedcorn, cornprice) (action, harvestedcorn, money)	η = 0.01, ε = 0.06, N = 10000, γ = 0.95	$174

5.3 Deep Q-Networks

The hyper parameters tuned for the DQN agents are: Number of episodes N; Epsilon for exploration vs exploitation ε; Discount Factor γ; Learning rate η. Hyperparameters of Deep Learning: optimizer; learning rate lr; activation function h;

The search space of the hyperparameters, best parameters learned from the DQN-1, DQN-2 and DQN-3 are given in Table 5.3.1. In addition to the fine-tuned hyperparameters: DQN-1 uses, *relu* activation function in hidden layers; linear activation in output layer and Adam optimizer to optimize the network weights; DQN-2 and DQN-3 use linear activation with argmax to predict the action; SGD optimizer with momentum value 0.99. The network weights are initialized with zeros in all DQNs.

Table 5.3.1 Tuned Hyper parameters of DQN-1, DQN-2 & DQN-3

Hyper Parameter	Range	Best Parameter		
		DQN-1	DQN-2	DQN-3
ε	0.02 - 0.08	0.06	0.02	0.01
γ	0.9 - 1.0	1	0.95	0.95
η	0.01 - 0.03	0.01	0.01	0.01
N	10,000 - 500,000	500,000	10,000	10,000
lr	0.01 - 0.001	0.001	0.0001	0.0001

5.4 Analysis

Q-Learning performed well during learning after tuning hyperparameters (η, ε, γ and N) and changing the verification condition for epsilon-greedy algorithm (exploration and exploitation). After reaching a stable solution Q-Learning performed the best during playing.

At first, Function Approximation methods performed poorly during learning, but it performed very well during playing. Using (state, action) pairs as features the learning performance is not very stable (great variance of expected utility). By changing the features, and tuning hyperparameters (η, ε, γ and N) the Learning process becomes stable and better results are achieved during playing.

DQN-1 performed well both during learning as well as playing, but the disadvantage is that the number of state action combination (s, a) grows significantly. Hence the number of model and hence the learning slows down. Whereas DQN 2 with Function Approximation performed equally well as DQN -1 with lesser time and a smaller number of episodes (10,000). This is very encouraging as the learning is much faster and model scales well as we build the full scope of the game.

Average performance of all the models during the learning and play times are shown in Figure 5.4.1 and Figure 5.4.2. The comparison of the model performance shows that the Q-learning outperformed during learning as well as play period. The learning of DQN-1 is converged in less than 500 episodes, and have stable result after 1,000 episode. Average performance of DQN-2 is better than function approximation models and closer to DQN-1 performance.

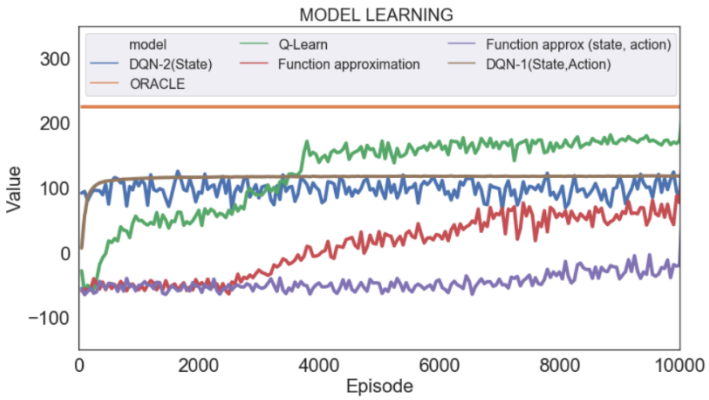

Figure 5.4.1: Comparison of average learning performance of agents

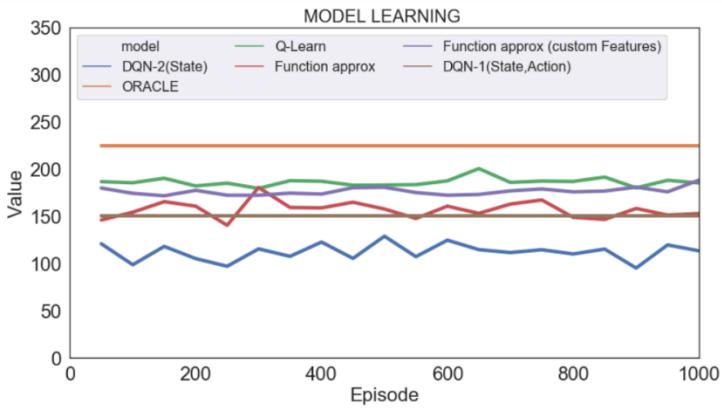

Figure 5.4.2: Comparison of average performance of agents during play

The learning and play performances of DQN-3 are shown in Figure 5.4.3 and Figure 5.4.4. The learning curves shows that average performance for every 50 episodes are varying between 400 and 1200. As our attempt to play the village game for 3 crops was not successful, we do not have oracle value for 3 crops game.

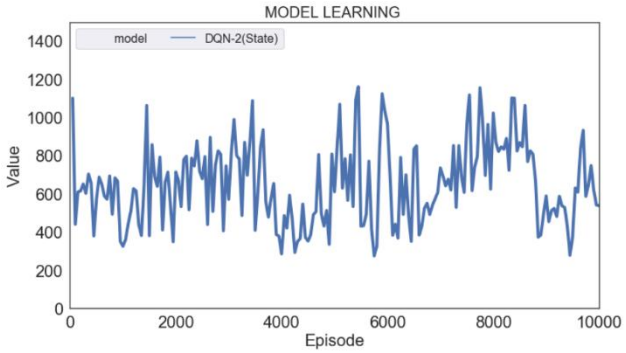

Figure 5.4.3: DQN-3 Learning Performance

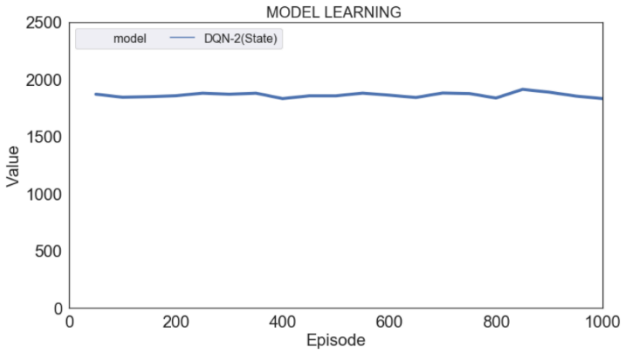

Figure 5.4.4: DQN-3 Play Performance

6. Results and Discussion

In addition to the average performance in every 50 episodes, the overall performance of the agents is analyzed. The box plots shown Figure 5.4.3 and Figure 5.4.4 describes the overall performance of the agents during learning and play period respectively. The performance plots show that Q-learning agent outperforms all other agent during learning and play time. The reason is that the state space is of manageable size for one crop. Hence Q-learning learns the optimal policy by updating the Q-table and the searching is guided by reward function. The maximum value achieved during learn and play time is $225. The learning performance of Function Approximation agent using state and action pair as features is low during learning time (median is below $50, and 3rd quartile is below $100). But interestingly, it showed good performance during play time (median value above $150) and reached the oracle value of $225. The reason could be during the initial stage of the learning time, the target value might be less accurate and agents takes time to approximate the Q-value function. Using custom features improve both training, which is more stable, and playing (median value above $174). This happens because custom features are better than state-action features (they extend better to unseen states). The overall performance of DQN-2 (state) agent is consistent in both learning as well as play time. The maximum value achieved is $200 in both learn and play time. DQN-1 agent shows poor performance in both learning and play time. The median performance during learning time is about $120, and during play time is about $150. The reason for the low performance is that that the DQN-2 has very large number of input and output parameters, hence the agent is not able to learning the network weights properly. The learning plot shows that there is some vanishing gradient problem in the DQN-2 network. After 200 episodes, there is no gradient update in the network.

The DQN-3 agent is built to overcome the state action space issue in the Q-learning agent. DQN-3 agent is trained to learn for the increased scope of the model. DQN-3 showed the good performance in both learning and play time. The Figure 6.3 shows that DQN-3 reaches $3000 at maximum with median performance of $500 during learning time. During play time, DQN-3 reached at maximum of $3750.

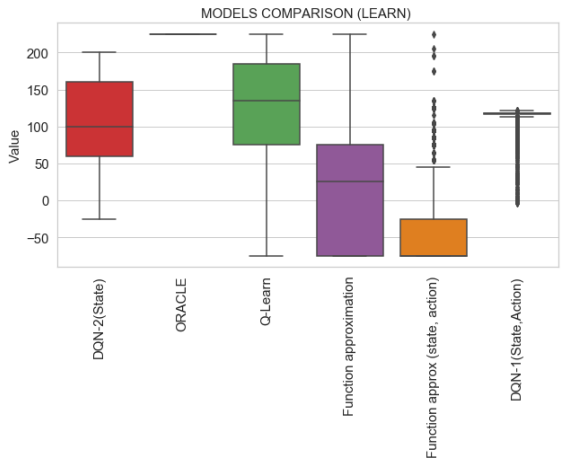

Figure 6.1: Agents overall learn performance

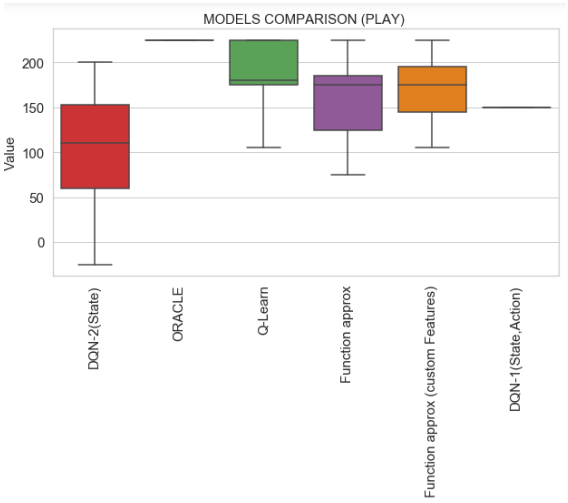

Figure 6.2: Agents overall play performance

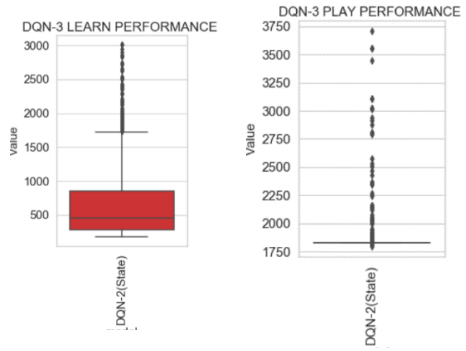

Figure 6.3: Over all learning and play performanceDQN-3

7. Future Work

In the present work, we demonstrated the application of different reinforcement learning agents for the village games. We have DQN-3 agent for the increased scope of the game and adding more constraints (more crops and more stochastic elements). DQN-3 agent shows that deep Q network can be built for the village game with all real time complexities. The current DQN-3 can be extended for the village game that includes cows, increased planning period (one or more years) and even more constraints (periods to plant and harvest). The same game can be modeled as an adversarial problem and create Deep Q-Network agent to play against a human player.

Acknowledgement:

Our professors Percy Liang and Dorsa Sadigh have helped us to get a solid foundation in Principles and Techniques of Artificial Intelligence. We would like to acknowledge our mentor Zach Barnes for his guidance to implement our solutions.

Contributions:

Our team of Megala Anandakumar, Rakesh Talanki and Bohdan Junior had amazing time collaborating from New York City, Atlanta GA and Curitiba, Brazil. We are thrilled to have gone through this journey together and delivered a novel modeling solution for the village game. The coding for the project is done in Python3, OpenAI Gym, Jupyter Notebook, Keras-rl and Tensorflow on GitHub. The video presentation of this project can be found in Youtube.

References

[1]. Stephan Schmitt-Degenhardt. (1997) Village Game - Facilitator's Manual. *Instituto Centro CAPE, Belo Horizonte, MG, Brazil.*

[2]. D.K. (2007) Dynamic programming and board games: A survey. *European Journal of Operational Research* 176 1299 –1318.

[3]. Amato. C., and Shani. G. (2010) High-level Reinforcement Learning in Strategy Games. *Proc. of 9th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2010)*, Lespérance, Luck and Sen (eds.), May, 10–14, Toronto, Canada.

[4]. Vincent-Pierre. B. et.al., Reinforcement Learning for Atari Breakout, *CS 221 Project Paper, Stanford University.*

[5]. Tapio, H.J.J. (2015) Reinforcement Learning in a turn-based strategy game. *Bachelor of Business Administration, Business Information Technology, project report, Theseus, Finland.*

[6]. Mnih V., et.al., (2016) Human-level Control through Deep Reinforcement Learning. *GoogleDeepMind,5, Nature, Feb 2016, Vol 518.*

[7]. Zarkias. K.S., et. al., (2019) Deep Reinforcement Learning for Financial Trading Using Price Trailing. *IEEE, ICASSP 2019.*

[8]. Ernest. A., and Dormans, J. (2012) Game Mechanics Advanced Game Design. *New Riders, Berkeley.*