Deep reinforcement learning for the board game Dominion

An assignment for the course AI tools

written by

Peter Khiem Duc Tinh Nguyen
Pengu20@student.sdu.dk

Course lector: Xiaofeng Xiong
ECTS: 5

The code for this project is available at
when the code is public put the github link in the curly brackets
`https://gitlab.sdu.dk/sdurobotics/medical/student-projects/2023/Peter_Duc_Bachelor_NLP`

the Faculty of Engineering (TEK)
University of Southern Denmark

Date of Hand In 31. of May

Contents

Chapter 1

Project specification

For this project, an AI agent must be designed, constructed and evaluated, based on a board game called "Dominion". The main problems posed, is the navigation and optimization of trajectories in large discrete state spaces. The aim for this project is therefore to construct a Reinforcement Learning method which can be used for large discrete spaces.

3 Different methods will be used to train a Reinforcement Learning agent to play the game Dominion. These methods will then be compared to each other as the conclusion of this paper. The structure of the entire project is shown as following: The project is divided into the following tasks:

- Create a Dominion engine that can be used to simulate games of Dominion.
- Design and create an AI agent that can play the game Dominion.
- Implement a deep learning approach for the Q-table.
- Create a training scheme for the AI agent.
- Evaluate the AI agent.
- Discuss the results of the AI agent.
- Conclude on the work.

1.1 Problem constraint

For this project there are some constraints which must be taken into consideration. These constraints are:

- The Dominion game can only support 2 players (The trained player, and the test player)
- The tested method are all within the category of 1-step temporal difference learning methods.
- The game normally varies for each game, as the cards present are randomized. For this project, the agent will train on a fixed set of cards.

The reason for only supporting 2 players, is that the game poses the problem of large variance due to the nature of the game. As the game already has large variance, it would be difficult to train and visualize progress, if the player had to battle more than one opponent. The reason for only evaluating methods within the category of 1-step temporal difference learning methods, is due to the limits of the time frame of the project. Furthermore, the main focus is to evaluate the differences between 3 different methods. Diverging from the specified category comparing only 3 methods, would pose too many variables to consider, if the methods yield different results. It would therefore be difficult to determine the cause of the difference in results. Is the winrate affected by the use of 10 steps instead of 1-step SARSA? Is it because of the low bias of monte carlo non-bootstrapping methods? Or is it because of the high bias of the 1-step SARSA method? These are questions that would be difficult to answer if the methods were not within the same category. For future work, more methods can be evaluated, and the category can be expanded to include more methods. The game features randomization in the game setup, in which the AI must be capable of generalizing its knowledge to other domains which are similar the learned domain. Due to the time constraint of

the project, and a lack of computational power, the game setup has been set to a fixed game setup for the AI to train on. The concept of domain variation due to randomization in the board game, will be discussed at the rules of the game.

Chapter 2

Task: Creating a dominion engine suited for a RL agent

The first step for the project is to create the board game engine which both can be used to play the game of dominion while supporting the playability from a Reinforcement Learning agent. Before going into the specifications of the criteria for the engine, the game of Dominion will be explained in short.

2.0.1 General rules of the board game Dominion

Each player starts with a deck of 10 cards, that are identical between players. At the start of each turn, draw 5 cards, at the end of each turn, put all cards from the hand into the discard pile. If there is not enough cards in the deck to draw all 5 cards, then shuffle the discard pile into the deck and draw until 5 cards are drawn. The main goal of the game is to reach terminal state with the most Victory points. These victory points are gained by buying victory cards.
All cards in the game is categorized into 3 different types of cards. The types are as follows:

- Action cards
- Treasure cards
- Victory cards

Action cards: Action cards are cards that can be played to perform an action, that manipulates the state space in some way or form. A typical example, could be the "smithy" that makes the player draw 3 cards. Based on the price of the
Treasure cards: These cards have assigned a currency value, which is used to buy other cards. The process of buying a card is discussed later in the rules
Victory cards: These cards are expensive card, which gives the player victory points. The cards hold no tactical value, other than to be bought to gain victory points.

A turn consists of an action phase, and a buy phase. The player is granted a single action that the player can do, and a single buy action, which enables the player to buy a single card. It is possible, through, action cards to gain more actions, and more buys. The cards that can be bought are shown as a shared pool of cards that are structured as piles of cards. Every pile represents all the copies of a unique card which can be bought. A typical game setup is shown in the illustration below:

Fig. 2.1: A typical setup of a Dominion game

The usual process is a randomization of the action card piles present. Only 10 action card piles are at play at a time, but around 25 unique action cards exists. For the sake of simplicity, the game will be played with a unique set of cards, that is fixed.

The process of buying a card in the buy phase, is done by showing the currency in hand that is enough to buy the specified card, and then putting the bought card into the discard pile, so it can be drawn in future turns. The treasure cards used does not disappear, and will be discarded at the end of your turn as all other cards.

The game ends when the highest cost victory card pile is empty (Card is named "Province"), or when 3 piles of cards are empty. The player with the most victory points wins the game.

2.0.2 Specifications of criteria for the Dominion engine

The game can be found for free on the internet. Therefore, the reason for creating a new engine, is to have a more flexible engine that supports the use of an AI agent, which does not need to extract information from an image. The advantage of creating a new engine, is that the engine can be designed to ensure that it is capable of giving the entire state space of the game to the AI through a single command call. All actions can also be given to the AI as a list of integers, which the AI must choose from. For the final result of the Dominion game engine, given the nature of the information delivered, it is observed that the game is far more intuitive for the AI than it would be for a human player.
The specific criteria for the Dominion engine are as follows:

- The engine must be able to simulate a game of Dominion. (The simulation speed must ideally be faster than the real-time average game length of 20-30 minutes)
- The engine must pass a game state object to the agent player, which contains all the information about the current game state.

- All actions that can be made in the game must exclusively be run by the agent that is inserted into the game.

2.0.3 Finished Implementation of the Dominion Engine

The finished implementation of the Dominion engine is a python class, which can input arbitrary players and simulate the game. Figure 2.2 shows that structure of the entire progran. Essentially it consists of the Dominion engine, the AI agents, a deck generator and an evaluation program. Figure 2.3 is the class structure of the Dominion engine.

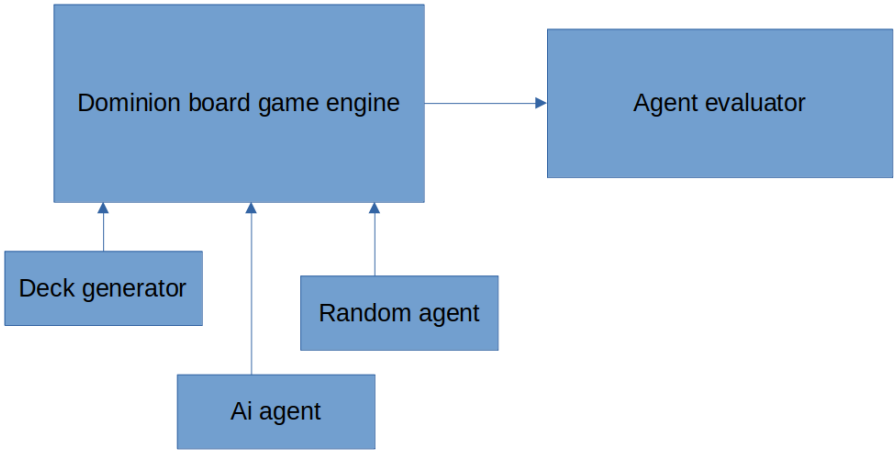

Fig. 2.2: This is the class structure for the entire project

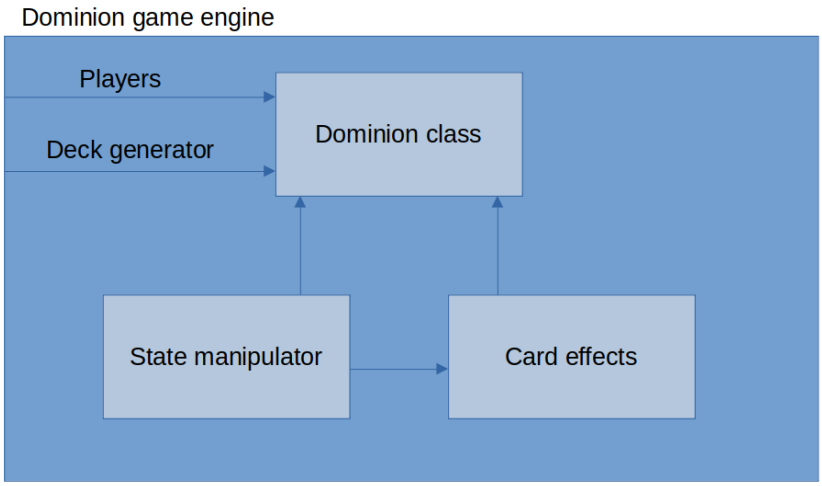

Fig. 2.3: The class structure of the Dominion engine.

The only constriction for the player object, is that the player object must have a function called "choose_action" with inputs action_list and game_state, that returns on of the actions. The game needs to auxiliary classes, that helps the class compute certain aspects of the game. An entire class has been dedicated to the manipulation of the state object. This might reflect the complexity and size of the programs that is needed to handle the state object. The card effects class is used to give each card their effect. Each card effect is exclusively limited to effects that alter the state object in some way or form. An explanation of the gamestate object is as follows:

The game state is a dictionary object, which contains the following keys:

game_state:

- The unique Dominion card set
- The supply amount of each card
- The type of action that can be done
- The parameter that the action needs to be executed
- The main player's state
- The adversary player's state

player_state:

- Player won
- Cards in hand
- Cards in deck
- Known cards on top of deck
- Cards in discard pile
- Owned cards
- Played cards
- Action points
- Buy points
- Treasure value
- Victory points

It should be observed that each listed key is a unique dimension that the value function must account for. As it is known that there are 17 piles in the game, then the first two keys correspond to 34 dimensions, with approximately 20 values for each. That in itself spans a state space that is 1.71e+44 unique states large. As a normal value function would be too impractical to train, a neural network must be used to approximate the value function. The neural network will be discussed in chapter 4.

An interesting note, is that information regarding the opponent player is partially observable. An example, is that cards in the opponent's hand is not shown, but the amount of cards in the hand is. In the same way, both the order of the cards in the opponent and the player's deck is not shown, but the amount of cards in the deck is.

Another aspect of the Dominion game, is that since the game only had to support AI agent training, then no GUI is created. This in turn also makes the game go significantly faster than a normal average game. The average game length of the engine using two agents is around 5-10 seconds, instead of 20-30 minutes. The game engine is now ready to be used to train an AI.

Chapter 3

Task: choosing and creating RL agents

For this part of the project, it has to be decided which kind of Reinforcement Learning method must be used. Due to the constraints of the assignment, it must either be a Monte Carlo based approach, or a Temporal Difference (TD) based approach. Between these to categories, it is decided that it would make more sense to focus on different Temporal Difference methods, since these offers more variety to work with. As stated earlier also, the project will focus on 1-step version of the most known TD-methods. The TD-methods that will be implemented are SARSA, Q-learning and Expected SARSA. Furthermore, due to the size of the state space of the board game Dominion. Then it is decided that the value function will be approximated using a neural network. This will be discussed further in chapter 4.
The next three are an introduction to all three methods, along with some implementation details.

3.0.1 SARSA

This method is the most fundamental TD-method which will be the fundament which Q-learning and Expected SARSA will work on. SARSA is an on-policy method that introduces a concept called bootstrapping in comparison to Monte Carlo type methods. The bootstrapping concept works on updating the value function based on the estimation of the next state and action. Meaning that the value function is updated based on an estimate instead of the actual reward. This method theoretically lowers the variance of expected returns in comparison to Monte Carlo methods. The update function for SARSA is as follows:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t) \right] \tag{1}$$

Where $Q(S_t, A_t)$ is the value function for state S_t and action A_t, α is the learning rate, R_{t+1} is the reward received after taking action A_t in state S_t, γ is the discount factor, $Q(S_{t+1}, A_{t+1})$ is the value function for the next state and action.
It should be noted, that for SARSA, then the next state is chosen based on an action taken with the current policy, hence making this method on-policy.
The specific hyperparameters that was tuned to fit SARSA best, was epsilon = 0.2, alpha = 0.1 and gamma = 0.45. The epsilon value decides the probability of exploring instead of searching as a greedy algorithm. The alpha value decides the learning rate of the agent, and the gamma value decides the discount factor of the agent. Lowering the discount factor was observed to increase the win-rate of the agent. A more detailed explanation of the hyperparameters will be discussed in chapter 7.

3.0.2 Q-learning

Q-learning is an alternative to SARSA that was designed to be a temporal difference method that is off-policy. The term off-policy refers to the feature, that Q-learning can use a different policy to choose the next action, when updating the value function. This means that Q-learning can explore the environment more freely, and is not bound to the current policy. The tradeoff between exploration and exploitation, is then not as strict as in SARSA. The update function for Q-learning is as follows:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right] \tag{2}$$

In this instance, the update function is similar to SARSA, but the next action is determined by a greedy policy. In our implementation, an epsilon-greedy policy is chosen instead. This will be discussed further in chapter 7. A noteworthy disadvantage with Q-learning, is that it can be more unstable than SARSA, as it be affected by maximization bias. This is due to the fact that the value function updated based on the best outcome in the next state. Therefore, due to Stochasticity, the value function will be overestimated. It will therefore need more training to converge to the optimal policy. To counter maximization bias, one can make use of something akin to double Q-learning. Double Q-learning is used to reduce the overestimation of the value function, by using two value functions instead of one. In our case, it would then mean that Q-learning will make use of both a Q-value approximator, and a target neural network which will serve as the target for the Q-value approximator.
For this method the following hyperparameters was chosen. epsilon = 0.8, alpha = 0.05 and gamma = 0.45. Due to the instability of the Q-learning method, it was decided to lower the learning rate.

3.0.3 Expected SARSA

Another way to avoid maximization bias, is to avoid using the maximum function all together. Expected SARSA is an off-policy method as Q-learning, that uses the average of all possible actions in the next state, instead of the maximum. For this project, it is used as midpoint, between SARSA and Q-learning. The update function for Expected SARSA is as follows:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t) \right] \tag{3}$$

As the equation shows. Instead of choosing an action, then the value function is updated based on the value of all actions multiplied by the probability of taking each action. Since the agent will be using an epsilon-greedy policy, then the probability value for each action will be, $\epsilon/|A|$ for all actions except the best action and $1 - \epsilon + \epsilon/|A|$ for the best action. Where |A| is the amount of actions given the state.

Chapter 4

Task: implementing a neural network Q-table

For this project a neural network must be used to approximate the value function for the agent. This is due to the large size of the dominion state space as discussed in chapter 2.0.3. This chapter will discuss all the details of the implemented neural network, which is identical for all agents.

4.1 Neural network structure

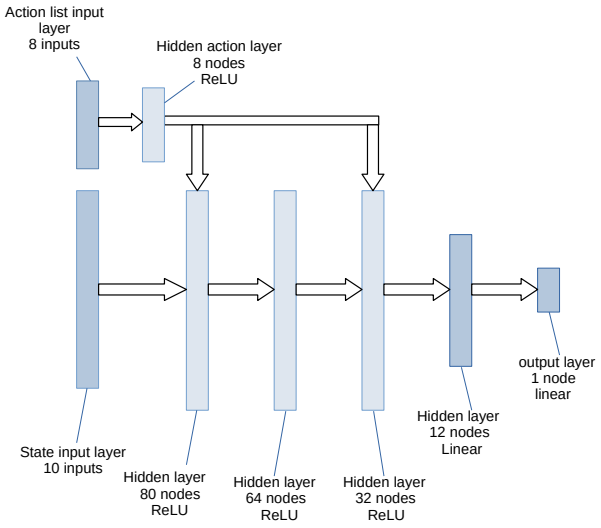

Fig. 4.1: The neural network structure used for the agents.

As shown in the neural network then the action values are fed into the neural network, as a residual connection. This is due to the fact, that it was observed that the actions has little value for the final result of the value function. As the actions are also passed in later in the neural network, the symbolic representation of the actions are represented stronger in the final output. Furthermore, it should also be noted, that a significant finding, was that the ReLU activation function was better than any other activation function tested. This is most likely due, to the fact, that the ReLU has an infinite range of positive values, which is beneficial for the value function. Though it should also be noticed, that the two last layers use linear layers, as the output of the value function should support being any number between -∞ and ∞.

Using this neural network structure then it is time for the agents to train on the Dominion game. This will be discussed in the next chapter. For the rest of this chapter, then the design choices for the neural network will be explained.

4.1.1 Residual action layer connection

It was shown through experience, that the influence of the action input was insignificant for all states that the agent was trained in. Meaning that given a state, then all actions would suffice to the same Q-table value. This feature

prohibited learning the concept of good actions, and had to be dealt with. To enlarge the importance of the action input, then the final neural network design made use of residual connections in which the action input was also added into the neural network later in the hidden layers. The intuitive idea, was that the action input would influence the final output greater, if the action input was also given at a later stage of the neural network, to make sure it does not vanish. This approach solved the problem of identical Q-table values for all actions.

4.1.2 Binarization of the action layer input

At this point of the neural network design, it was decided to use one integer value as the action input, instead of the 8 values used now. The agent was now capable of learning the value difference between actions. But another problem was observed. It was observed, that the neural network would learn a dependency between action index and the value of the action. This dependency would incorrectly influence the value function. An example was the action to buy a "curse" card, which in practically all cases would be a bad choice. This action is numerically adjacent to buying the "province" card, which in practically all cases would be a good choice to buy if possible. Since the "province" card rarely was bought due to its expensive price, then the neural network would learn that the "province" card probably was a bad choice, since it was numerically close to the "curse" card.

To mitigate the spatial relation between actions, then the action input was binarized. action 1 would then be represented as [1, 0, 0, 0, 0, 0, 0, 0], action 3 as [1, 1, 0, 0, 0, 0, 0, 0] and so on. Furthermore, a hidden layer, exclusively for the action input was added. This approach removed the spatial relation between actions giving each action an appropriate value.

4.1.3 Optimistic initialization

To improve the explorative nature of the agent, then the weights of the neural network was initialized optimistically. Intuitively, then the agent would try new states with greater probability before realizing that the state was bad. Specifically the neural network weights was optimistically initialized by setting them based on a normal distribution with variance 0.01 and mean 0.05

Chapter 5

Task: Training scheme

For this section, the general training structure will be discussed. This training scheme serves the purpose of training the agent, such that its value function will converge at an optimal policy. The agent was faced with a random walk policy at each game. But to even the odds then the random walk policy was never allowed to buy curses which significantly boosted its win chance.

5.1 Training amounts

Training a dominion gamed until convergence usually lasted around 48 hours. This meant that training was doable, but not replicable, as it would take too long. Convergence happened at around 2000 games. To get reliable data, then instead of training a single agent op to 2000 games, the agent is trained in 20 epochs with 150 games. The return gained from each game will then be averaged to reduce variance in the results of the methods.

Chapter 6

Results

The section is dedicated to the results of this project. Figure 6.1 shows the average winrate graph of all agents. The agent has trained in 150 games, and 15 epochs. The performance is evaluated after every 4 training games. The amount of performance games is therefore $150/4 = 37$ games.

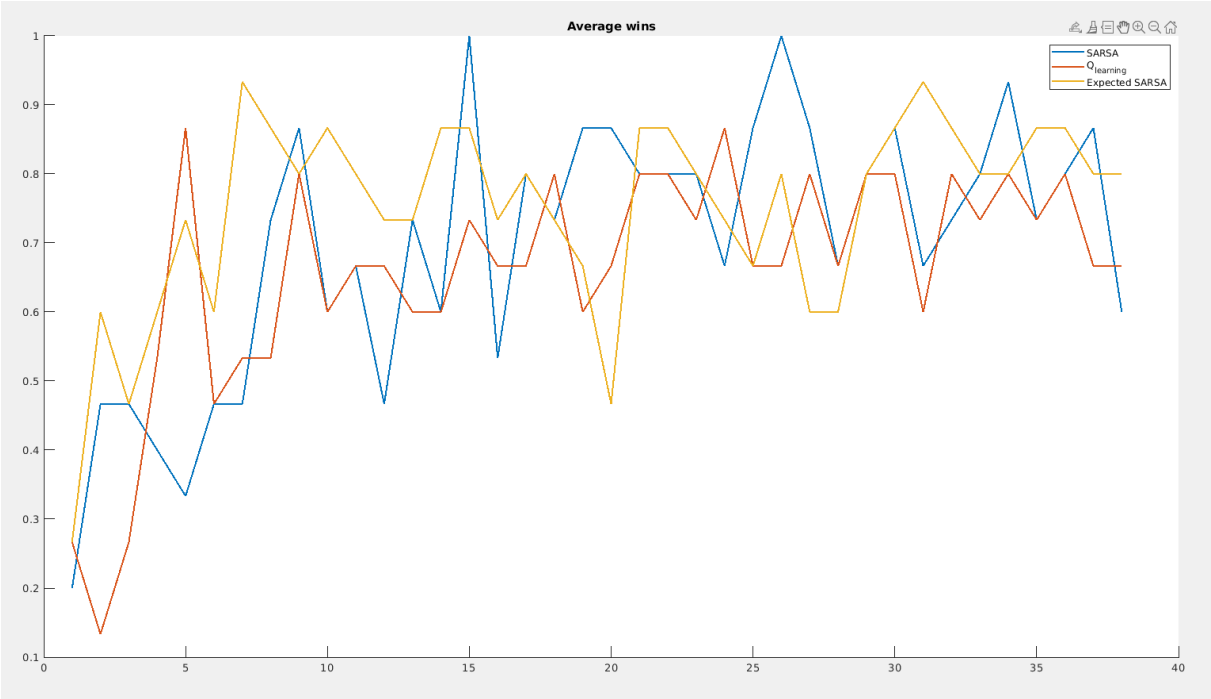

Fig. 6.1: This figure shows the winrate of all three methods based on 20 epochs with 150 games each. The x-axis is performance game, and y axis is winrate

Figure 6.2 shows the discounted returns of all three agents.

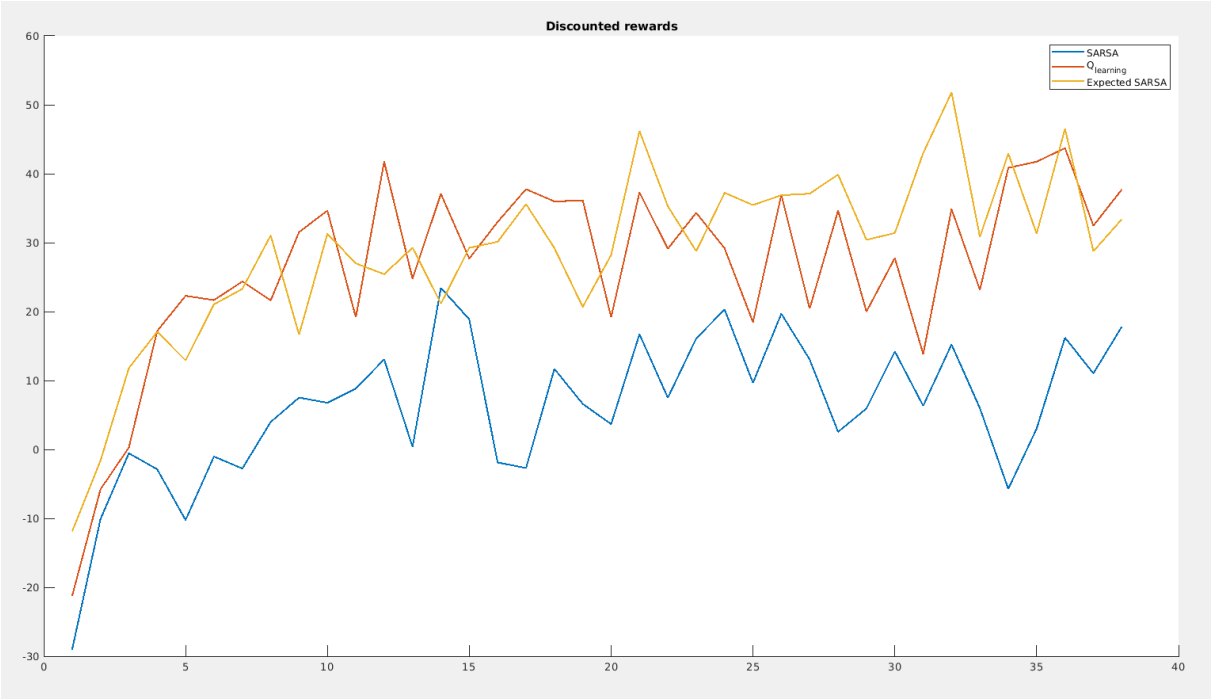

Fig. 6.2: This figure shows the discounted returns for all three methods based on 20 epochs with 150 games each. The x-axis is performance ga

Chapter 7

evaluation & Discussion

It can be shown, that the most superior algorithm which trained, is Q-learning. Another interesting fact, is that the target neural network had little to no effect on the performance of the agent. It is observed that the winning strategy of the game, is to recognize the value of the standard cards (Copper, silver, gold, estate, duchy, province and curse!). This is especially the case with high reward cards, as gold and provinces. But by the nature of the game, it is very rare, that the AI experiences buying gold and provinces, so each event in which the agent gains the given cards, must be learned from as much as possible. Since Q-learning bootstraps based on the maximal action possible in the next state, then the value of buying gold and province is better learned, than with the other agents.

Chapter 8

Conclusion

For this assignment, 3 different methods have been compared up against each other. The methods are Q-learning, SARSA and expected SARSA. Through evaluation of the different methods, it has been shown, that the Q-learning agent learns faster than the other agents. Furthermore, based on the evaluation difference between Q-learning with and without a target policy, it has been shown that the target neural network for, this application, shows no significant improvement in the performance of the agent.