Developing an AI Agent for Splendor using Reinforcement Learning

Felipe Garavano[B00804287], Olivier Halkin[B00802430], and Yasmina Hobeika[B00804284]

CentraleSupélec

Abstract. Splendor is a widely-known board game that requires strategic decision making and is typically played by 2 to 4 players. In this project, we created a virtual environment from scratch that incorporates all the rules,and board elements of Splendor, as well as the various moves that a player can make for a 1 player version. To enhance game-play, we utilized reinforcement learning and developed a DQN agent that can help players improve their strategy. Nevertheless, the complexity of our environment limits our agent, which we will discuss in detail, along with potential venues for further improvements.

Link to Github: https://github.com/GaravanoF/Splendor_RL

Keywords: Reinforcement Learning · Splendor · Turns · Environment · Games · Board · Cards · DQN Agent · Replay Memory · Training

1 Motivation & Problem Statement

Splendor is a strategy board game where players collect and manage gemstones to buy cards, which in turn provides them with more gemstones and victory points. The goal of the game is to reach a certain number of victory points (15) before your opponents do. It requires to consider a multitude of different possible actions, many interactions with the other players and some randomness in the cards drawn. It is therefore a hard game to play as optimal strategies are heterogeneous based on a one time situation. This, however, makes the game interesting to do a Reinforcement Learning project on it.

While there are already several existing environments and agents for Splendor, creating our own environment and agent offered a unique and challenging learning opportunity. By building these components from scratch, we were able to gain a more thorough understanding of the game mechanics, the challenges of designing a RL system for it, and the potential benefits of this approach for improving gameplay. Additionally, our implementation includes moves that where not possible with existing frameworks.

2 Environment

The Splendor environment is a game simulation where the player aims to gain the required number of victory points in a minimal number of turns. In the beginning, the environment initializes with a deck of cards, where each card has a unique cost, color, tier and victory point value and a deck of nobles with requirements and victory points. The player has a fixed set of gem tokens, consisting of 4 tokens from each color(Blue, Black, Green, Red and White) and 3 jokers, which can be utilized for purchasing cards from the deck. At the start of the game, the player has no cards or gem tokens.

The state space of the environment is defined by the state of the deck of card and nobles, the state of the player's gem tokens, the cards that the player has purchased, the cards that the player reserved, and the nobles the player has acquired. Specifically, the deck state includes the number of remaining cards, the cost and victory point value of each card, and the tier of each card. The player's gem token state includes the number of each type of gem the player has available to use for purchasing cards. Finally, the cards that a player has purchased represent a unique state for a player, as a player can have a different set of cards with different victory point values and costs.

The action space of the environment is defined by the actions that the player can take during their turn. These actions include taking gem tokens, purchasing cards from the deck, reserving cards for later purchase, and purchasing reserved cards. When a player takes gem tokens, he can choose to take three gem tokens of different types or two gem tokens of the same type if there are at least 4 gems of that color. When a player purchases a card, he must pay the cost of the card using the gem tokens he has collected and the cards he has in possession (one blue card counts as an extra blue gem). Finally, when a player reserves a card, he can take a card from the deck and place it in his hand for later purchase while also acquiring a joker that is a special token that can be used as any color.The max number of reserved cards is 3 and of gems in the player hand 10. Note that if the player has more than 7 gems he can only take up to 10 gems (so 2,1 or 0).

The rewards in the Splendor environment are based on the number of turns used by a player to win the game. Thus it is calculated based on the number of maximum turns, which is 60 minus the actual number of turns taken by the player. The reward is higher for a player who wins the game in fewer turns, so the smaller the number of turns, the higher the reward. Other rewards were tested, but demonstrated poorer results.

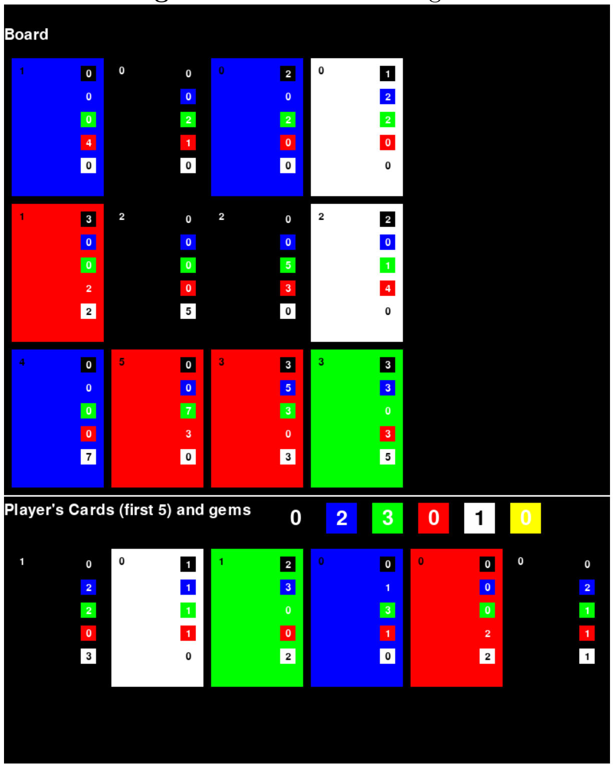
2.1 Environment visual representation

Based on the *pygame* python library, a visual representation was made. It includes critical information, and helps understanding the complexity of the game.

3 DQN Agent

For this project, we implemented a Deep Q-Network (DQN) agent to learn how to play the game of Splendor. The DQN algorithm learns to approximate the optimal Q-value function using a neural network.

Fig. 1. Visualisation of the game



The DQN agent consists of two main components: the Q-network and the replay memory. The Q-network is a deep neural network that takes in the game state as input and outputs a Q-value for each possible action. In our implementation, the Q-network has three fully connected layers with 32 hidden units each and uses the rectified linear unit (ReLU) activation function. The output layer of the network has the same number of units as the number of possible actions in the game (67 possible actions). The Q-network is trained using the mean-squared error (MSE) loss function and the Adam optimizer. We tried more complex architectures but didn't achieve good results.

The replay memory is a data structure that stores the agent's experiences in the form of tuples containing the current game state, the chosen action, the resulting reward, the next game state, and a flag indicating whether the game is over. The replay memory is used to break the correlations between consecutive experiences and to sample uniformly from past experiences during training.

The DQN agent also uses an epsilon-greedy exploration strategy, where it selects a random action with probability epsilon and selects the action with the highest Q-value with probability 1-epsilon. The value of epsilon is annealed over time to gradually decrease the exploration rate and encourage the agent to exploit its learned policy.

During training, the DQN agent interacts with the Splendor environment and stores its experiences in the replay memory. The agent samples a batch of experiences from the replay memory and updates the Q-network using the Bellman equation to approximate the optimal Q-value function. The agent also updates its exploration rate after each training episode to gradually decrease the amount of exploration.

4 Results

Overall, the DQN agent was able to learn a good policy for playing the game of Splendor, achieving an average win rate of 95.3% after 2,000 episodes of training, and an average number of turns equal to 36.6 which is almost 7 turns better than the random agent. Some important things to note, our agent seems quite unstable so we defined a loss in the game as the agent taking more than 60 turns to win since sometimes it gets stuck in a loop. Also the results varied greatly depending on the seed of the training and we did manage to achieve better average performances reaching as low as 32 but with a lower win rate. We decided in the end to prioritize the best agent we could get while being as stable as possible. Still our agent has a 5 % loss rate compared to 0.8 % of the random agent.

Performance Comparison			
Agent	Average turns	Losses	Average w/o losses
Random Agent	42.46	8	42.46
DQN Agent	37.75	47	36.6

Table 1. Performance by Agent

5 Discussion and Next Steps

The first and major faced challenge was the representation our complex actions and observation space. For our action space we decided to encode all 67 possible actions to specific index in a list of 67 values this allows us to at each step calculate the valid actions and let our agent choose only from the valid actions indexes. For the observation space however it was more tricky since the state is dependant on a multitude of variable, that could take different size and various value. As an example, there are 60 different cards, and the player can have from 0 to twenty cards. We decided to represent this by doing a dictionary which during training we flatten and the hash to a numpy array representation. This is adapted to our varying observation length size as we needed to have a representation of fixed size for the DQN agent to learn. It is important to note though that by hashing our flattened states we are losing a lot of information. For the size of the hased array we chose a large size to avoid collisions problems.

Regarding the hyperparameters' tuning, we employed *Optuna*, an open-source hyperparameter optimization framework, to fine-tune the hyperparameters of our Deep Q-Network (DQN) agent. Optuna facilitates an efficient and automated search for optimal hyperparameters by employing a tree-structured Parzen estimator (TPE) algorithm, which sequentially constructs a probabilistic model to predict promising hyperparameters. This approach offers superior exploration-exploitation balance compared to traditional grid search or random search techniques. Using Optuna, we conducted a series of trials wherein the DQN agent's hyperparameters, such as learning rate, discount,epsilon decay and exploration strategy, were optimized based on their impact on the agent's performance.

Our agent right now has serious limitations mainly due to the pre-processing of the observation, given its complexity we are losing a lot of information that could be useful for better learning. In particular a better approach could have been to use padding for the state representation rather than hashing to deal with the varying observation length issue.

For next steps, the agent still has a long way to go to reach the performance of a human player though it is quite better than random right now. Future steps would include improving our representation of the observation, we recommend implementing a Proximal Policy Optimization (PPO) agent, as it often outperforms DQN by addressing its limitations, such as high sensitivity to hyperparameters and overestimation bias. PPO offers more stable and robust learning by constraining policy updates, resulting in improved sample efficiency and faster convergence to optimal solutions. And if an agent that solves the one player game is achieved the next step would be to transform the game into a 2-4 player game which should be fairly easy given our current set up but would mean learning a harder policy which takes into account the other players game play.

6 Conclusion

In conclusion, we successfully developed a virtual environment replicating the Splendor board game and implemented a Deep Q-Network (DQN) agent to enhance the gameplay. Despite achieving a win rate of 95.6 after 2,000 training episodes and beating a random agent on average by 7 turns, the agent still faces limitations due to the state representation's information loss. To overcome these limitations and improve the agent's performance, we suggest exploring alternative state representation techniques and adopting advanced reinforcement learning algorithms, such as Proximal Policy Optimization (PPO). Future work could also include transforming the game into a 2-4 player setup, which would require learning a more complex policy that accounts for other players' actions.

7 Appendix

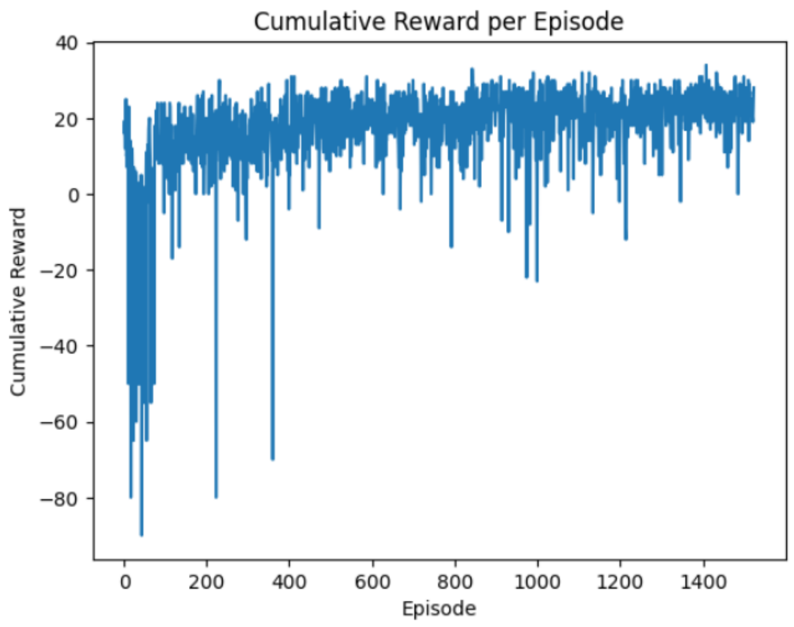
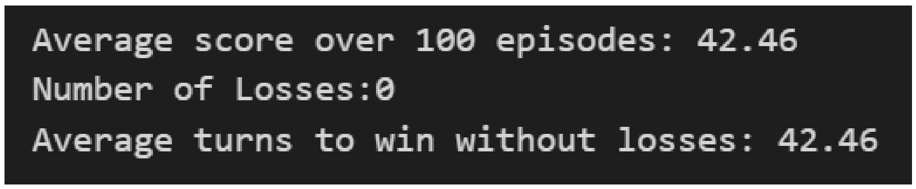
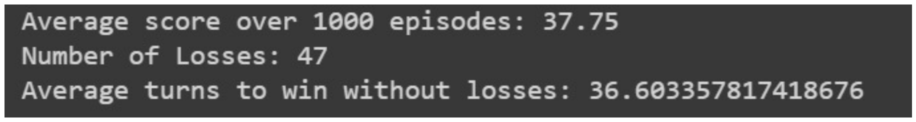


Fig. 2. Rewards per episode



```
Average score over 100 episodes: 42.46
Number of Losses:0
Average turns to win without losses: 42.46
```

Fig. 3. Random Agent Results



```
Average score over 1000 episodes: 37.75
Number of Losses: 47
Average turns to win without losses: 36.603357817418676
```

Fig. 4. DQN Agent Results