YANIV – a social card game from Nepal

Creating a AI players to play the game of YANIV

Participants:

David Shavin

David Ohayon

Chen Cnaani

Ilay Aharoni

Game Description

Yaniv is a social card game originated in Nepal, legends tell us the origin of the name is from 2 Israeli hikers named Yaniv and Assaf and they're the ones who brought the game to Israel and gave the game the name Yaniv.

Rules: *in bold are the important terms that are repeated throughout the project*

The games starts with a single card on the **pile** (which is open for people to see the top card) and a **stack** of all the other cards which is closed.

We'll define the **score** of a player as the sum of the numbers in the player's cards.

Each player will start with 5 random cards from the stack, each one makes a move in his turn.

The goal of the game is to have the minimal score between all contesters when a player calls Yaniv. The player with the lowest score wins.

Move: a move consists of dropping cards to the top of the **pile** and taking a single card either from the **stack** of cards (in which case the player won't know what card he's picking up) or from the top of the **pile**. The player can only put 1 card of his choosing on top of the pile, unless he drops a **serie**.

Serie: a **serie** can either be multiple cards with the same number or a sequence of consecutive cards (e.g. 7,8,9) of at least 3 cards with the same suit (clubs, hearts, diamonds or spades).

A move can also be calling "**Yaniv**" which will terminate the game immediately making the winner the one with the lowest number of points. Do note however that **Yaniv** can be declared by a player only in his turn and only if his **score** is not greater than 7

PILE

STACK

Proposed methods for solving the problem (build a YANIV agent)

The project proposes a few ways of solving the problem of creating the Yaniv player. The main focus is on developing a RL based player. We offer multiple heuristic-based players and a RL based player which is named Harel *(sounds like the letter R and L in English)*. Harel is trained used Q-learning against the heuristic players and itself, we'll eventually add Harel as a pkl file in our report for users to be able to enjoy playing against. I'll now present the details of each player and implementation details about the RL based player

Players:

Heuristic Players:

GreedyRandomPlayer(p=0.5): Greedy-random-player is a player whose tactic is to choose with probability **p** *(a parameter defined by the user)* between if to pick a card from the pile or from the stack. The player will prioritize laying down multiple cards if he has multiple cards with the same number. His second priority will be to drop to cards with the highest numbers (e.g. we'll prefer to put on the pile the cards (7_hearts, 7_hearts) over (6_spades, 6_diamonds). He'll also prefer putting down (3_hearts,3_clubs) over (13_hearts) and he'll prefer putting down (8_diamons) over (13_diamonds)).

The player will call Yaniv if he can, thus, if his score is not greater than 7 then in his turn he'll call Yaniv.

SemiRandomPlayer(): Semi-random-player is a player who's tactic is to randomly which card to put down to the pile. It will choose a random number from the numbers in his deck and will put the cards with that number to the pile. Due note, if he picks the number 4 for example and he has multiple cards with the number 4 than we'll put them all in the pile. The player choose to take a card from the pile if the card on the top of the pile has a smaller value (number) than the number of the card/cards his putting on the pile. If the number of the card on top of the pile is greater than the number that the player is putting down than he'll pick a card from the stack.

The player will call Yaniv if he can, thus, if his score is not greater than 7 then in his turn he'll call Yaniv.

Reinforcement Learning Player (using Q-Learning):

Harel (R-L): Harel is our RL player, he'll learn to play the game using Q-learning.

Implementation details:

State: The state will be represented as a vector of length 6.

The first 5 indices will represent the cards in the player's hand. If a player has less than 5 cards than the value at the index of the missing cards will be 0 (in terms of the game, having no card and having a card with a value of 0 (a joker) is equivalent).

The 6th index represents the number that is currently on top of the pile.

Action: The action will be represented as a vector of length 2.

The first element will be an integer from 1 to 3, where: 1=YANIV | 2=PILE | 3=STACK. This will represent the type of move we're doing, if the value is 1 then the action will be to call Yaniv (note that this isn't always possible) and if the value is 2 or 3 than we'll pick a card from either the pile or the stack.

The second element in the vector will represent the numbers we're putting down. We put down a number and not a card since we ignore the suits of the cards, in the case of multiple cards in the player's hand with the same number, we'll drop them **all** into the pile. Note: if the first element in the vector is 1 (we're calling Yaniv) then we'll make the second element 0.

Notes about implementation of the action and the state:

- Notice that Harel completely ignores the suit of the cards, we decided to do that because it reduces drastically the number of states and actions in the algorithm. Also the suit of the cards has a very minor effect in the game (having a serie is a rare occurrence in the game)

- We also don't allow the player to put down one card if he has multiple cards with that number. The idea behind this is that it reduces the number of possible actions which will make the learning more feasible and it doesn't really makes sense in the game to put down a seven for example when the player can put down 2 sevens (though we rather Harel learns himself that it isn't a good decision but for our program to work properly we must reduce the number of possible of states and actions so we don't encounter memory problems)

- The numbers in the first five elements in the state is represented in an ascending order which reduces even more the number of possible states (the order of the cards in the players hand has no meaning in the game)

- Harel doesn't take advantage of the fact that in the game of Yaniv we can put down a sequence of consecutive cards (e.g. 7,8,9) of at least 3 cards with the same suit, this is because we ignore the suits of the cards to reduce the action and the state spaces. We're ok with this since having a serie of at least 3 consecutive cards with the same suit is a rare occurrence in the game and we'll rather our agent ignoring that case so it can learn faster and better (increasing the state and action spaces will make it harder to learn and so we don't expect good results and we've encountered in previous implementations of the states and the actions memory issues).

Q-Learning:

The algorithm we used to build our agent is Q-learning, the algorithm is straightforward however there were 2 issues worth mentioning:

1. **nextState:** The next state includes the card on top of the pile at the next state, however, the card on top of the pile right before Harel's turn (its next state) isn't the card on top of the pile right after his turn. This is because other players have played and changed the card on top of the pile. Therefore, we defined the card on top of the pile at the next state to be the card that Harel will see on top of the pile when it's his turn again. This means that now we'll have to finish the entire round in order to know what the next state. Do note, during that round a player can call Yaniv and the game will end immediately and the next state will be a terminal state.
2. **Reward:** The most obvious reward to choose is +1 or -1 if Harel wins or Losses, else, the reward would be 0 (the game isn't over yet). The problem with this method is that we'll

update the qvalues only once per game, this takes a long time to train. Our solution was to try different rewards and see the difference we get when we test our player.
Alternative reward-per-round:
Gives the same reward but a part of checking the winner at the end of the game, we'll check for each round in the game who's the leader.
The leader will be defined as the player with the lowest score (the lower the score is the closer the player is to winning).
We'll give Harel a reward of +1 or -1 if he's the leader or not by the end of each round.
Problem with this alternative: We're teaching Harel that he should be the leader at each round of the game and while this sounds like a good idea, the goal of the game isn't to be the leader at each round, the goal is to win the game, and sometimes the better strategy might be to get in purpose a higher score (which is bad) in order to get a smaller score later. For example, if Harel has a king (13) and a 4 in his hand and on the top of the pile there's also a king, Harel should learn that taking the king while giving up the 4 is a good strategy (even though his score will get higher) because now he has 2 kings and in the next turn he'll be able to drop 2 cards to the pile *(basically it's a good strategy to try and get a serie in your hand so you'll be able to put more than a single card on top of the pile which will result in fewer cards in your hand, since in Yaniv the player always takes only a single card from either the pile or the stack).*
However, this method of reward might be worth considering. Our Q-Learning algorithm is implemented in a way that tries to minimize the number of states and actions, therefore, the state takes into account only the player's card and the card on the top of the pile, resulting in our player not learning that the chance of a Yaniv being declared is dependent on what the other players have picked from the pile, the number of cards they have in their hand, the number of rounds that were played in the game (it's unlikely that a Yaniv will be declared in the first few rounds) etc. This means that Harel won't be able to learn if he should take a risky move (like the one described above), and he'll make this decision based only on the state defined in our Q-learning algorithm and on if it was worth taking that risk in the past. This means that our agent sees the changes of a Yaniv being declared as totally random and so it does make some sense to use the alternative reward and we'll later see in the graphs that it does yield better results (probably because it's easier to train with that reward since the agent gets feedback every round and not only once per game).

Results and conclusions (Graphs, reality vs expectations etc.):

We'll represent multiple graphs showing the result of the RL agent against the heuristic players. To understand the graphs, we'll need to understand the training process involved in training Harel.

Harel was trained on 20 intervals, each interval consists of 20,000 games.

Harel was trained playing against SemiRandomPlayer for intervals 1->5 and 11->15.

Harel was trained playing against GreedyRandomPlaer for intervals 6->10 and 16->20.

Harel was trained in intervals 1->10 with an exploration probability of 0.5.

Harel was trained in intervals 11->20 with an exploration probability of 0.25.

We've repeated this entire process with different learning rates and discount (presented on the graphs' legends).

All This process was implemented twice:

1. With a reward of +1 or -1 if Harel wins or Losses, else, the reward would be 0.
2. With a reward of +1 or -1 if Harel is currently leading or not (leading means that he has the lowest score in his hand)

Harel was **tested** after each interval, the test consists of 1,000 games playing against a GreedyRandomPlayer and another 1,000 games playing against a SemiRandomPlayer.

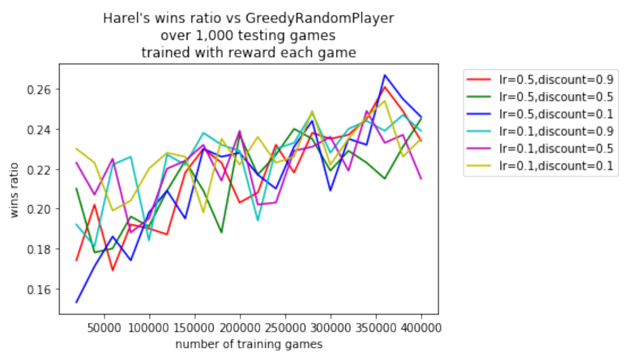
Note, the test doesn't consider who was Harel trained against, so we'll expect the slope in the graphs to indicate on a better performance rate for SemiRandomPlayer in the tests after intervals 1->5 and 11->15 (games 1->100,000 and 200,001->300,000) and a better performance rate for GreedyRandomPlayer in the tests after intervals 6->10 and 16->20 (games 100,001->200,000 and 300,001->400,000). We can in fact see this in the graphs of the performance when using the method of reward-per-round however it's not trivial to see this and the effect is minor, a reason for that might be because of the nature of the game of Yaniv. Yaniv is a game where the type opponent doesn't have a big effect on the player's decisions and we'll care mostly about his own cards and how to minimize his score. This is especially true in our representation of the state and action in the game since Harel can't even know anything about his opponent and so he can't learn how to predict his move and act accordingly.

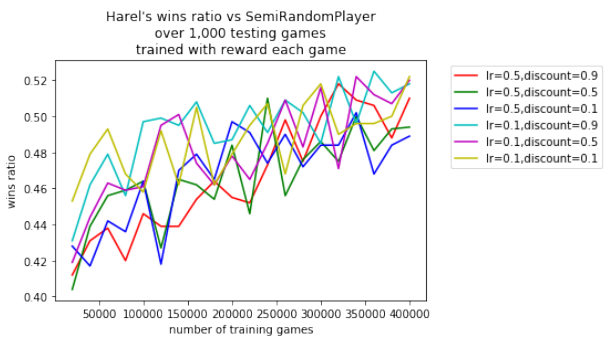
Graphs meaning:

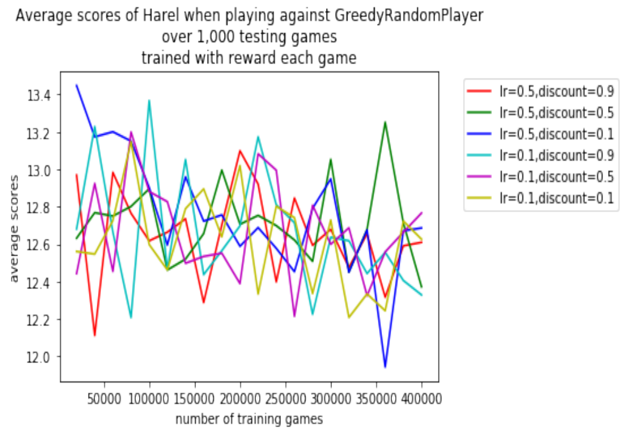
We'll display 2 types of graphs,

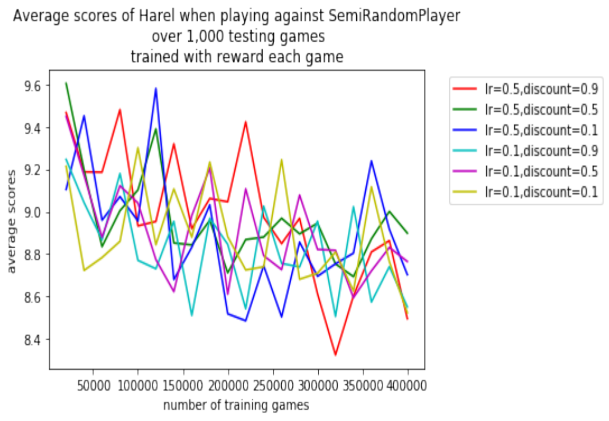
1. **Harel's average score at the end of the game**: shows how Harel's average score at the end of each game changes according to the amount of training he has. Note that this doesn't show if Harel won or loss, but instead shows "how close" he is to winning, eventually Harel's goal is to have the minimal score at the end of the game, and this is what the graph shows.
2. **Harel win-loss ratio at the end of the game**: shows the win-loss ratio vs his opponent over 1,000 games.

Graphs With a reward of +1 or -1 if Harel wins or Losses, else, the reward would be 0:

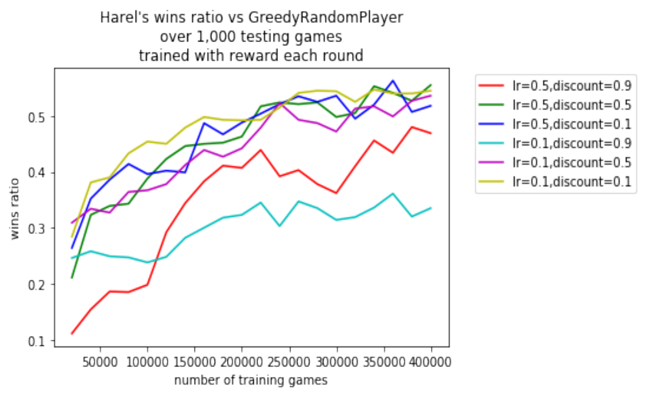


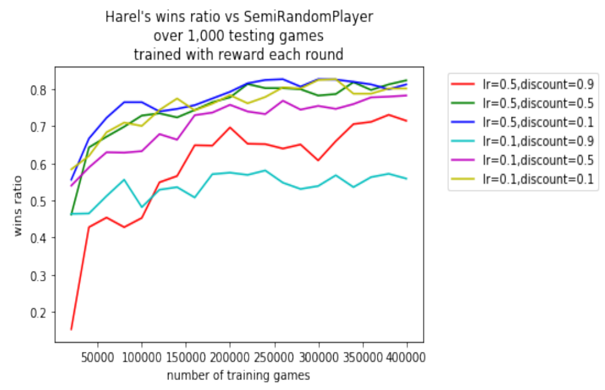


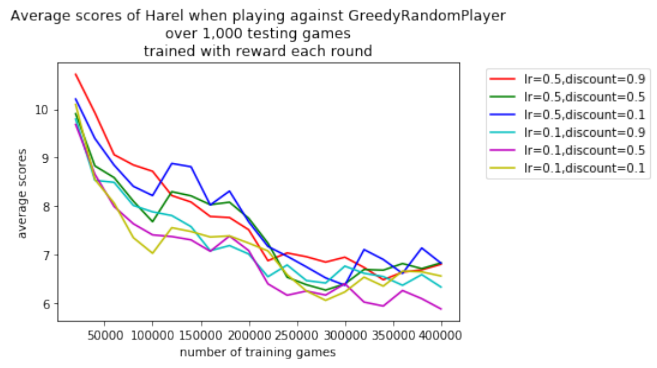


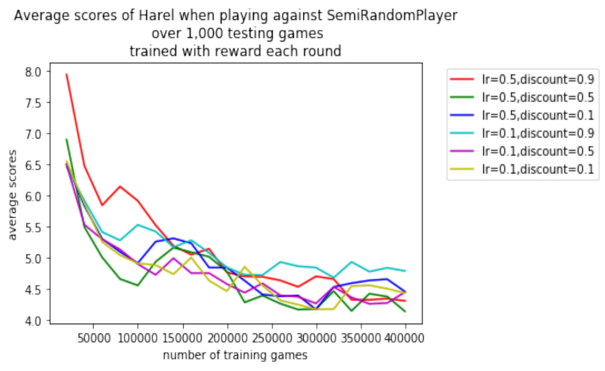


Graphs with a reward of +1 or -1 if Harel leads the game of not (reward-per-round):









Discussion about the graphs:

Graphs with reward each only at the end of the game:

We can see the graphs are messy and drawing conclusions about the preferred learning rate and discount isn't simple. Our understanding is that we should expect to see better results with a higher discount because non-zero reward is given only at the end of the game and so states should care more about future state when updating, however there's also a big random element in the game which will make it harder for the states to learn from the future states since the card on top of the pile is essentially random which might explain why it not obvious to draw from the graphs that a higher discount is better. In addition, the method of reward every game doesn't yield good results as the method of reward every round, this makes sense since this method of reward gets far less feedback than the method of reward-per-round and so training might take much longer.

Graphs with reward-per-round:

Yields better result than reward only at the end of the game, probably since it gets more feedback which means it learns faster. When looking at the graphs with the ratio of wins we see that Harel is preforming better with a lower discount, that makes sense since with that type of reward Harel will be encouraged to look solely in its current state and action in order to maximize the immediate reward it can get. We don't learn much from looking at the learning rates.

Additional notes:

- For the first 10 intervals we'll use an exploration probability of 0.5 and for the next 10 intervals we'll use an exploration probability of 0.25, this is because we'll want Harel to initially train with a higher exploration rate so it will explore more it's options.
- Notice how Harel is preforming better against SemiRandomPlayer compared to GreedyRandomPlayer, the reason for that is obvious, GreedyRandomPlayer is a better player who makes smarter decisions. The fact that Harel wins over 50% of the games vs GreedyRandomPlayer and over 80% of the games vs SemiRandomPlayer, when training with reward-per-round, is not bad. In fact, if we would have trained him even more Harel yields even better results, however, we didn't show it in the graph since it takes a very long time, and Harel trained already with 400,000 for each type of hyperparameters (6 types) and for 2 different reward method, thus, 400,000x6x2=4,800,000 games.

Attachment – state and action space complexity , code summary, how to play the game and learning what Harel learned:

State space & Action space complexity:

State space complexity if 13^6 since we have a vector with 6 dimensions, where each value can take a value from 1 to 13.
Action space complexity is 3*13 since we have a vector with 2 dimensions where the first dimension can take a value from 1 to 3 and the second dimension can take a value from 1 to 13.

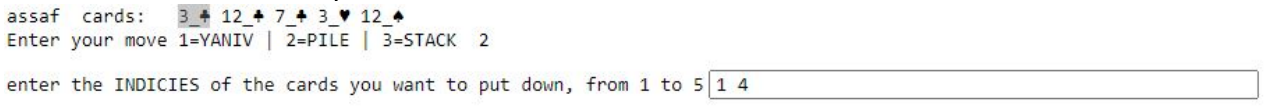
Code (description of the code):

game_manager.py:
This file contains the class GameManager. A game_manager object will manage the game, it will take as input the players and will start initializing the pile and the stack by dividing up the cards between the players. A game_manager will also contain a step method, this step is called when making a turn in the game and takes as an input the player which turn we'll make.

players.py:
This file contains an implementation of the players in the game (the heuristic players, the Q-Learning based player and the real player which is the player we'll define if a real player wants to play the game). The general structure for each player is to have a move method which will decide on a move and update its card according to the move, then it will return a **move** object (see utils.py) so the game_manager will update the game. In addition, each player will have a get_score method which will return the current score of the player.
play_yaniv.py:
This file will run a game of Yaniv, you'll need to define in it the players you'll want to play and then this file will initialize a game_manager and will run the game (basically calling game_manager.step with the player which turn it is) until a player calls Yaniv, and then we'll finish the game and declare a winner.
train.ipynb:
This notebook contains the training process we used to output the data for our graphs and the code used to build the graphs. This file is used mostly for viewing how exactly we trained our models. Note that in the notebook there's data about the Q-learning agent, this data was put there manually by taking the output of the training process and copying it over there (see the last cell for better understanding).
utils.py:
This file contains many useful classes and functions:
Move: a class to keep a player's move (the decision between calling Yaniv, taking a card from the pile or taking a card from the stack. It also holds cards the player is putting on top of the pile
Card: a class representing a card (a number and a suit)
Deck: a class that holds a list of cards, it also contains many useful functions (like sorting the cards) for the players to use (mostly useful when implementing the heuristic players, but not just).
CustomDict: The class from ex4, in ex4 it was called Counter, however, we're using already a python object named Counter, so I renamed it. We'll use CustomDict for holding the q-values.
RL_utils.py:
This file contains a class representing an Action and a class representing a state for the Q-Learning based player.

How to run the code:

Run the file play_yaniv.py.
In order to be able to play with different notice how this short script works:
We'll build objects of the players we want to play against, for the heuristic players we can simply call:
grplayer = GreedyRandomPlayer('grplayer_name')
smplayer = SemiRandomPlayer('srplayer_name')
For the real player we'll build an object of RealPlayer which takes as a parameter the player's name and for the Reinforcement Learning player we can load (using pickle) the player (the player will be added to the project as harel.pickle)
if we want to display cards of any of these players we can see the 'display' variable of the player as true (e.g. grplayer.display=True).
We build a GameManager object, passing a list of all the players as an argument and we'll set display=True so we'll view the cards on top of the pile.
😉You can now run the program
Notice the syntax of the program, you should enter only numbers (if not, the program will print error msg and ask you again for legal input)
The program we'll ask of you 2 types of input, the first is the type of move, you should enter a number from 1 to 3 (see program) or "q"/"Q" to quit the game, don't forget to press enter. The second input is the **indices** of the cards you want to put down. Enter the indices with a space between them and press enter. In the example below we can see that to put 3_spades and a 3_hearts which are at indices 1, 4 you should enter: "1 4"



```
assaf  cards:   3_♣ 12_♣ 7_♣ 3_♥ 12_♠
Enter your move 1=YANIV | 2=PILE | 3=STACK  2

enter the INDICIES of the cards you want to put down, from 1 to 5 1 4
```

By default, I left the code with a game of 4 players: the real player, Harel, GreedyRandomPlayer and SemiRandomPlayer.

Harel: The player stored in harel.pickle is trained a lot with the alternative reward method reward-per-round. He was trained against different versions of reinforcement learning agents and also against the heuristic players. The learning rate and discount we choose for him is 0.1 and 0.1 accordingly (the are the hyperparameters which yielded the best result when training with reward-per-round).

Analysis of Harel's action:

When analyzing Harel's action we see that he managed learning that getting rid of higher card is good. In addition, he learns that it is better to drop multiple cards. Harel seems to keep multiple cards in his hand if the value of these cards is small. Harel managed to learn that taking a high card from the pile is a bad choice and will prefer to take a card from the stack. Harel even occasionally preferred gathering pairs of cards (which will later allow him to drop multiple cards to the pile) even if that meant picking a high card from the pile (though this is quite rare and might just be chance). Harel is a good player but his tactic is no match for an experienced human player (though with a little bit of luck he manages to win human players). This isn't a surprise since human tactics are very strong and aren't simple at all for a Q-learning player to learn, especially since our state represents only part of all that is happening in the game (doesn't look at other players decisions and what they picked, doesn't look at the stage of the game we're in etc.)