Introduction to Artificial Intelligence Final Project - Durak with Reinforcement Learning

Vitaly Rajev, Ziv Mahluf, Idan Yamin, Eyal Diskin

Hebrew University of Jerusalem
August 2020

Table of contents

Introduction

Durak (Russian: дурак, meaning: 'fool') is a popular russian card game. The objective of the game is to get rid of all one's cards when the deck has no cards remaining in it. The last player which has cards in their hand loses and is the 'durak' (fool).
Durak is a flexible game with regards to the rules with which it can be played, and there are many variations of the game as a result, including ones which allow cheating (and it's the players' resposibility to notice in time), attacks of multiple cards, addition of cards after the round is over (in case of failure to defend), and more.
In this project, we've implemented AI agents for a simple variation of the game using reinforcement learning approaches.

Rules of the Game

Game Setup The game is played with a deck of 36 cards, which is a standard 52-card deck from which the cards with values 2-5 are removed. Before the start of the game, the deck is shuffled and a card is drawn and placed face-up under the deck, so it's visible to all players, and the suit of the card is the trump suit for the game. Then, each player is dealt 6 cards, and if all starting hands are legal, the game begins, otherwise, the cards are returned to the deck, the deck is reshuffled, and the cards are dealt until all opening hands are legal. An opening hand is considered legal if there are no 5 or more cards of the same suit in it.
After the cards are dealt, the first attacking player is the player with the lowest valued trump card (if no player has a trump card, a random player is chosen to attack first).
An attacking player attacks the player to its left, and after each round the turn to attack goes clockwise.

In our implementation of the game, the trump suit is constant for all games, and there is no card placed under the deck face-up.

Round The game consists of consecutive rounds, each consisting of multiple steps of attack and defence.
Each step involves at most one attacking card, and one defending card.
A round begins with a mandatory attacking card from the attacking player.
The defending player responds to an attacking card with a defending card - which must be a card with the same suit and a higher value, or, if the attacking card is not a trump card, any trump card.
If the defending player has no legal cards to respond with, or chooses not to respond, they take all cards currently on the table.
If the defending player responds with a legal card, another step begins.
After the first step, the attacking player can choose not to attack, or might not be able to, and pass the chance to the next player in clockwise order.
Any attack after the first step must be with a card with a value that is already on the table (for example, if the cards on the table are 6 and 8 of hearts, attacking cards can be any 6 and any 8).
A round is over when there are no more cards in the defending playr's hand, when the defender successfully defended from 6 attacking cards, when no player attacked, or when the defending player has taken the cards on the table (unable or unwilling to defend).
If the defender did not take the cards, they are discarded face-down and are inaccessible for the players for the rest of the game.
At the end of the round, all players draw from the deck until they have 6 cards in hand (if needed), or until the deck is empty, starting from the attacker, then the other non-defending players in clockwise order (the same order of attack), and lastly the defender.
After drawing the cards, all players with no cards in their hand leave the game,

and another round begins with the next attacker.
The next attacker is the next player clockwise if the defender did not take the cards. Otherwise, the defender's turn to attack is skipped, and the next attacker is the next player clockwise after the defender.

End of Game The game is over when there are no cards left in the deck, and there is at most one player with a non-empty hand. If all players' hands are empty, then the game is considered to end in a tie. If there is one player with a non-empty hand, then they're considered the loser.

Approach and Methods

For this game, we've decided to use reinforcement learning techniques to train the AI agents with two different algorithms. NFSP learns using the Q value, while PPO learns using the Advantage. The use of a neural network to approximate the Q-values of a state different actions was chosen since the number of possible states in Durak is too large to store in memory, even without taking into account memory regarding the cards in other players' hands, the cards in the current player's hands, and discarded cards.

Neural Fictitious Self-Play (NFSP): brief introduction to NFSP: NFSP uses two independent neural networks and two memory buffers, one for each network.
The first network is $Q(s, a|\theta(Q))$ with M_{rl} memory buffer.
The second network is $\Pi(s, a|\theta(\Pi))$ with M_{sl} memory buffer.
Mrl is a circular buffer that stores agent experience in the form of: $(s_t, a_t, r_{t+1}, s_{t+1})$
s_t = state at time t
a = action at time t
r_{t+1} = reward at time t+1
s_{t+1} = state at time t+1
$Q\left(s, a|\theta\left(Q\right)\right)$ is a neural network that uses off-policy reinforcement learning to predict action values from data in M_{rl}. It approximates best response strategy, $\beta = \varepsilon - greedy(Q)$. meaning it selects random action with probability ε, and the action that maximizes the Q-value with probability of $1 - \varepsilon$. M_{sl} is a reservoir that stores best behaviors in the form of (s_t, a_t)
s_t = state at time t
a_t = action at time t
$\Pi\left(s, a|\theta\left(\Pi\right)\right)$, is a neural network that maps states to action probabilities. It is trying to learn best behavior using supervised learning from its history of previous responses in M_{sl} and thus, it defines the agent's average strategy, $\pi = \Pi$.
Now we have two neural networks which predicts actions. Each turn that the agent makes we must choose an action, so with probability η we are choosing an action from $\varepsilon - greedy\left(Q\right)$ and with probability $1 - \eta$ we are choosing an action from Π. In each turn we store $(s_t, a_t, r_{t+1}, s_{t+1})$ in M_{rl}, and for each turn that uses $\varepsilon - greedy\left(Q\right)$ to get an action we also store (s_t, a_t) in M_{sl} for future use in the supervised learning of Π.
The algorithm as presented in the paper:

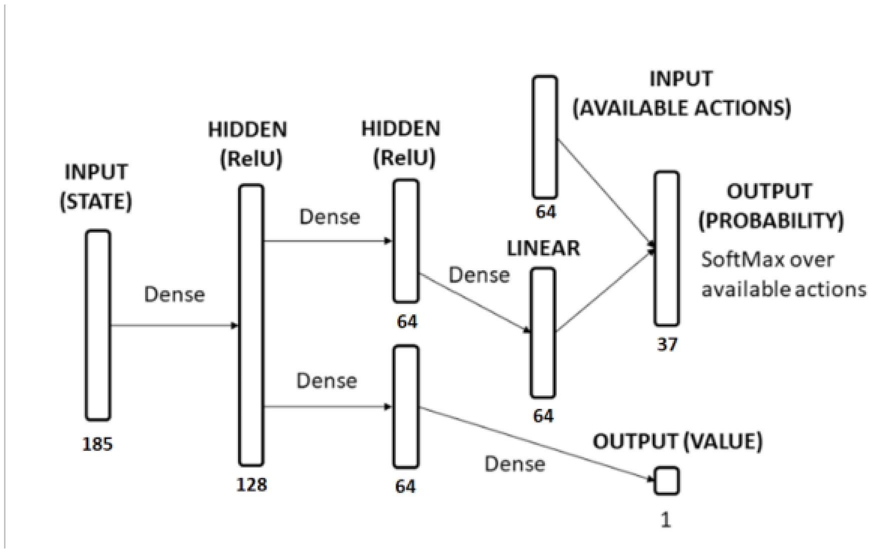
Algorithm 1 Neural Fictitious Self-Play (NFSP) with fitted Q-learning

```
Initialize game $\Gamma$ and execute an agent via RUNAGENT for each player in the game
**function** RUNAGENT($\Gamma$)
    Initialize replay memories $\mathcal{M}_{RL}$ (circular buffer) and $\mathcal{M}_{SL}$ (reservoir)
    Initialize average-policy network $\Pi(s, a \mid \theta^{\Pi})$ with random parameters $\theta^{\Pi}$
    Initialize action-value network $Q(s, a \mid \theta^{Q})$ with random parameters $\theta^{Q}$
    Initialize target network parameters $\theta^{Q'} \leftarrow \theta^{Q}$
    Initialize anticipatory parameter $\eta$
    **for each** episode **do**
        Set policy $\sigma \leftarrow \begin{cases} \epsilon\text{-greedy}(Q), & \text{with probability } \eta \\ \Pi, & \text{with probability } 1-\eta \end{cases}$
        Observe initial information state $s_1$ and reward $r_1$
        **for** $t = 1, T$ **do**
            Sample action $a_t$ from policy $\sigma$
            Execute action $a_t$ in game and observe reward $r_{t+1}$ and next information state $s_{t+1}$
            Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in reinforcement learning memory $\mathcal{M}_{RL}$
            **if** agent follows best response policy $\sigma = \epsilon$-greedy $(Q)$ **then**
                Store behaviour tuple $(s_t, a_t)$ in supervised learning memory $\mathcal{M}_{SL}$
            **end if**
            Update $\theta^{\Pi}$ with stochastic gradient descent on loss
                $\mathcal{L}(\theta^{\Pi}) = \mathbb{E}_{(s,a)\sim\mathcal{M}_{SL}}\left[-\log \Pi(s, a \mid \theta^{\Pi})\right]$
            Update $\theta^{Q}$ with stochastic gradient descent on loss
                $\mathcal{L}\left(\theta^{Q}\right) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{M}_{RL}}\left[\left(r + \max_{a'} Q(s', a' \mid \theta^{Q'}) - Q(s, a \mid \theta^{Q})\right)^2\right]$
            Periodically update target network parameters $\theta^{Q'} \leftarrow \theta^{Q}$
        **end for**
    **end for**
**end function**
```

Main steps of the algorithm:

1. Sample policy σ such that with probability η it is $\varepsilon - greedy\,(Q)$ and with probability $1 - \eta$ It is Π
2. Sample an action a from σ and execute it. Insert $(s_t, a_t, r_{t+1}, s_{t+1})$ to M_{rl}
3. if σ is $\varepsilon - greedy(Q)$, then store (st, at) in M_{sl}.
4. use Msl to train $\Pi\,(s, a|\theta\,(\Pi))$ + use M_{rl} to train $Q\,(s, a|\theta\,(Q))$
5. periodically update target networks parameters.

PPO Agent: The agent is trained using the Advantage value of the neural network, which is calculated after the game is finished. The calculated loss of the network is a linear combination of 3 different losses - the entropy loss, the value function loss, and the "conservative policy iteration" loss. Further information about the loss evaluation can be found at (Henry Charlesworth, 2018). The reason we use these types of losses is to prevent changes that are too big in the gradients, while encouraging exploration.



We used the same architecture as described in (Henry Charlesworth, 2018) but with smaller layer shapes, since we trained the models on our personal laptops for smaller number of iterations.

Training

Neural Fictitious Self-Play (NFSP)

Network parameters: we chose to represent the input of the network (and as an extention, a state in the game) as a vector containing the following information:

1. legal actions that the player can make
2. cards that are currently part of the attack
3. cards that are currently part of the defence
4. cards in the hand of the player
5. cards that are in the discard pile of the game

each of those is represented by a vector with 36 values (there are 36 cards in the game) except for the legal actions vector which has an additional value for the "no card" action, and each value is either 0 or 1 (bit vector).

therefore, the size of the input is $36 \cdot 5 + 1 = 181$

that information does not represent the full knowledge that a player may have at a certain point (for example, a player may be able to know that another player has certain cards, and there is no way to represent that information in our state vector), but because we wanted to create an agent that can play against any number of opponents, we didn't want to encode into the state any information that depends on the number of player.

the output for each vector is a vector with 37 values, one for each action in the game.

each network is comprised of fully connected layers, each followed by a *ReLU* layer.

Training parameters:

Training set 1: 3 hidden layers, starting from 256 and descending to 64
Π learning rate $= 0.00075$
Q learning rate $= 0.05$
$\eta = 0.1$
the M_{sl} and M_{rl} sizes are $15,000$
ε starts at 0.9 and decays to 0.0001
we traind the networks for 100 epochs, each consisting of 50 games

Training set 2: 6 hidden layers, starting from 256 and descending to 64
Π learning rate $= 0.0005$
Q learning rate $= 0.001$
$\eta = 0.1$

the M_{sl} and M_{rl} sizes are $15,000$
ε starts at 0.9 and decays to 0.01
we traind the networks for 100 epochs, each consisting of 50 games

PPO:

Network parameters: The network input is a binary vector of size 185 (37 * 5) representing the legal actions, cards played as attack, cards played as defense, cards in the hand of the player, and cards in the discarded pile. For each of these options we added a NO_CARD option, therefore getting a binary vector of size (36 + 1) * 5 = 185. The output vector is of size 1, which contains the index of the action we chose.

Training parameters: We trained the model for 140k games, using mini-batches of 5 games each, and training steps of 25.

We used a clip range of 0.2 to prevent the network from doing big changes every iteration, and a small learning rate of 0.00025 which decays linearly to 0.000001. We also used maximum gradient norm of 0.5 to prevent big changes.

The coefficients we used (value function and entropy) were used to reward/punish the player's loss function according to the given parameters, as described in the loss calculation in the PPO paper.

Results

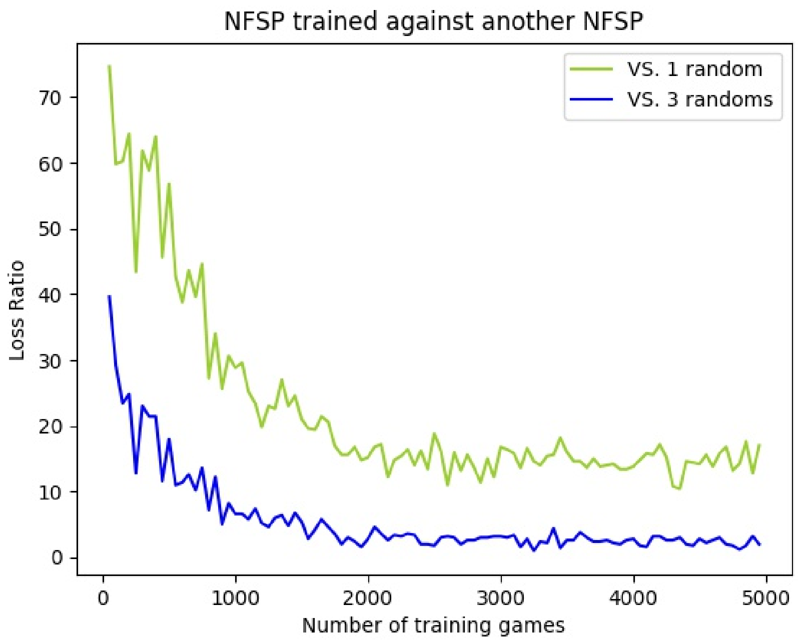
Types of Players Against Which the Agents Are Tested

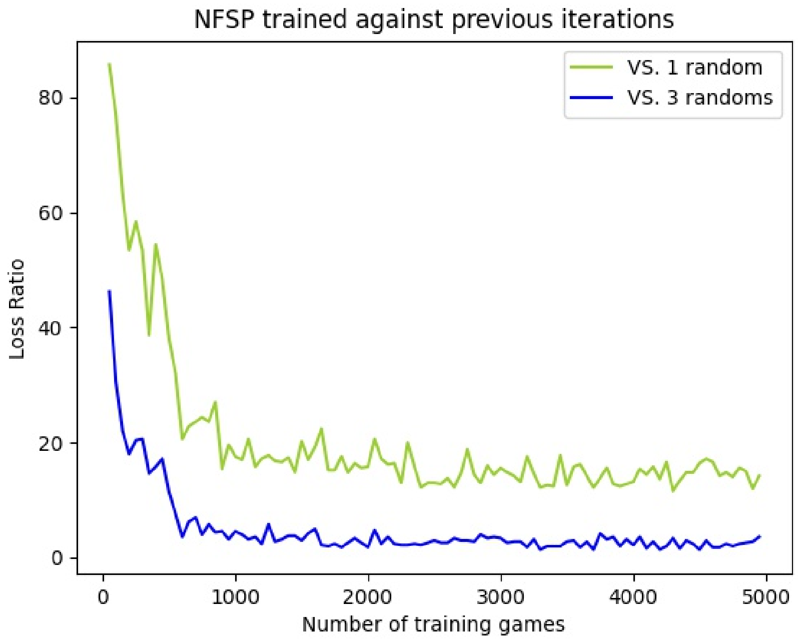
Aggressive Player An aggressive player attacks with the strongest non-trump card available (the one with the highest value). If no non-trump cards can be played, the player will choose the weakest trump card available. The player will defend with the weakest card possible. The player will always prefer to play a card, if possible, and only pass if it's the only legal action available.

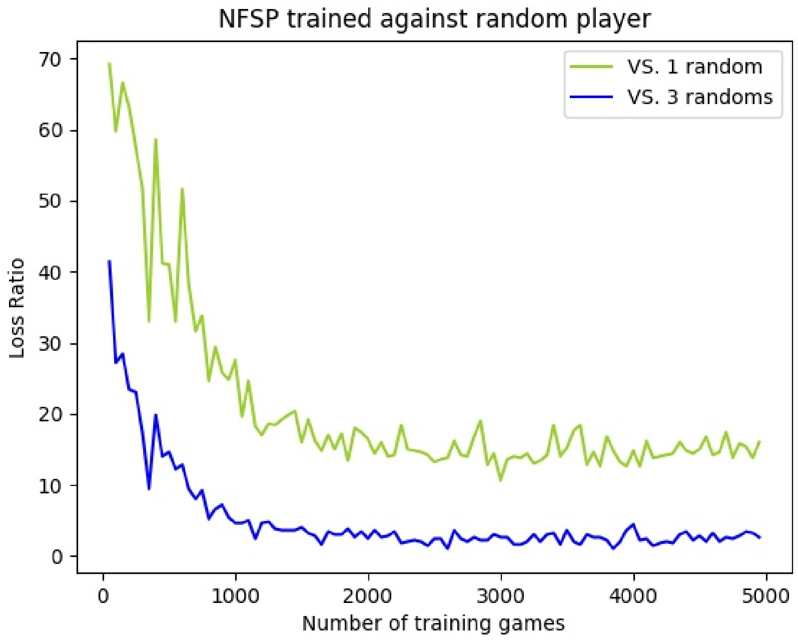
Defensive Player A defensive player will always attack and defend with the weakest card available. The player will always prefer to play a card, and will pass only when it's the only legal action available.

Random Player A random player will attack and defend with an action chosen at random from the legal actions available.

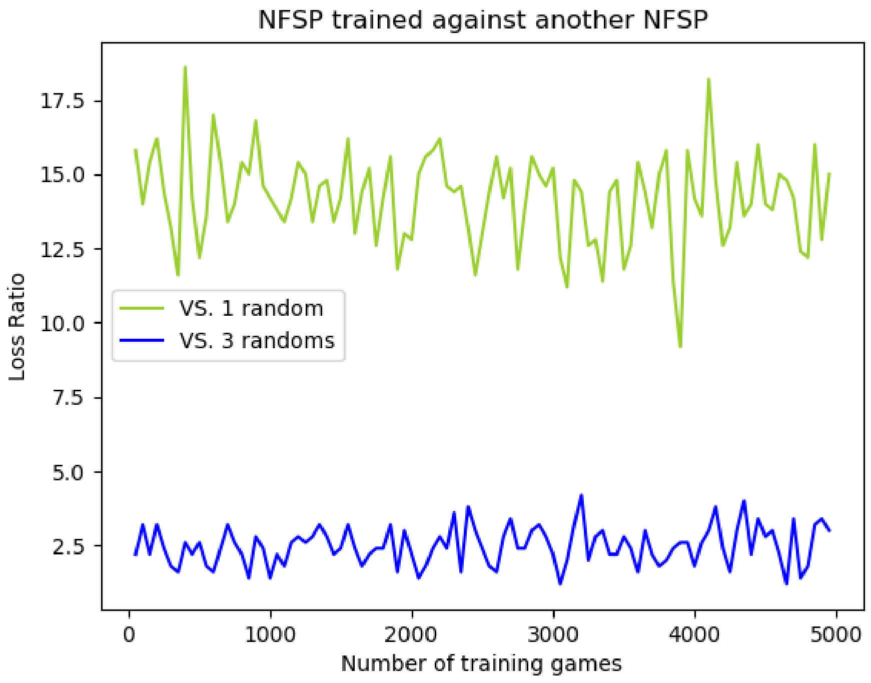
NFSP training set 1:

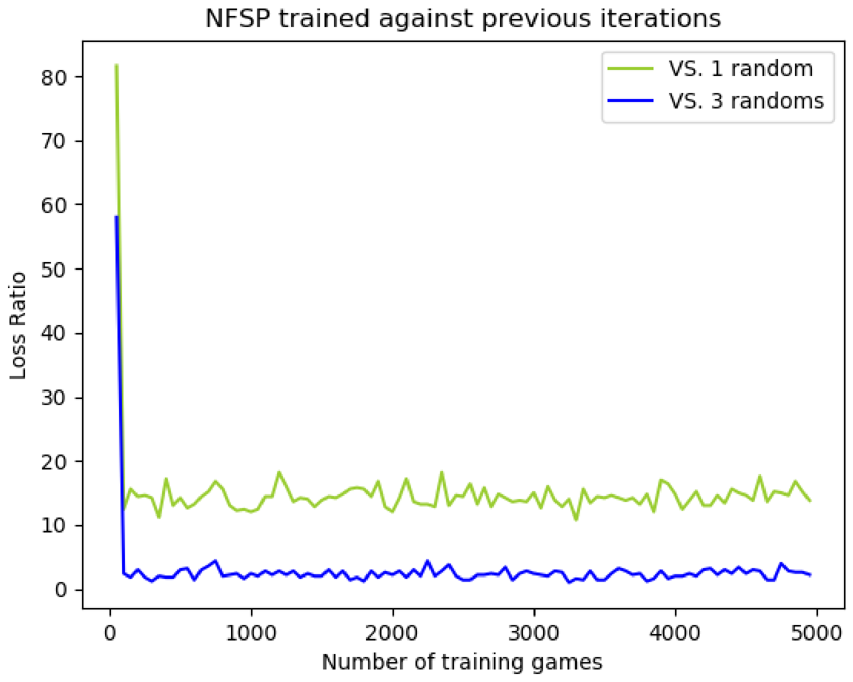


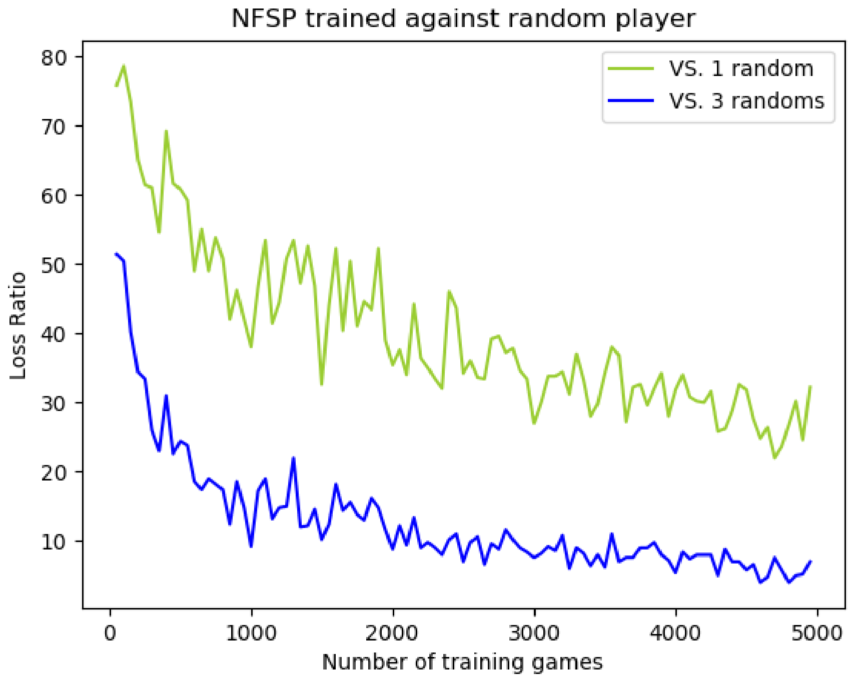

NFSP trained against previous iterations
VS. 1 random
VS. 3 randoms
Loss Ratio
0
20
40
60
80
0
1000
2000
3000
4000
5000
Number of training games


NFSP trained against random player
VS. 1 random
VS. 3 randoms
Loss Ratio
0
10
20
30
40
50
60
70
0
1000
2000
3000
4000
5000
Number of training games

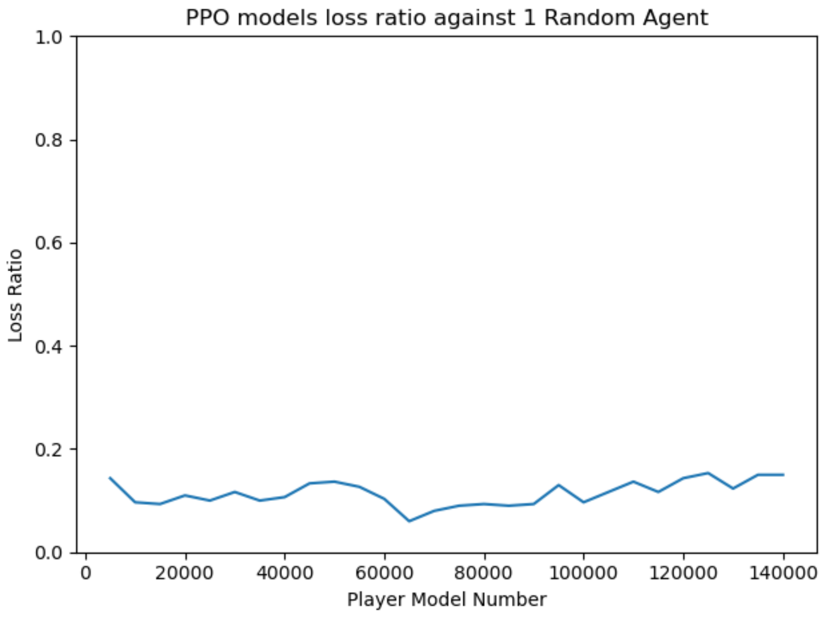
NFSP training set 2:

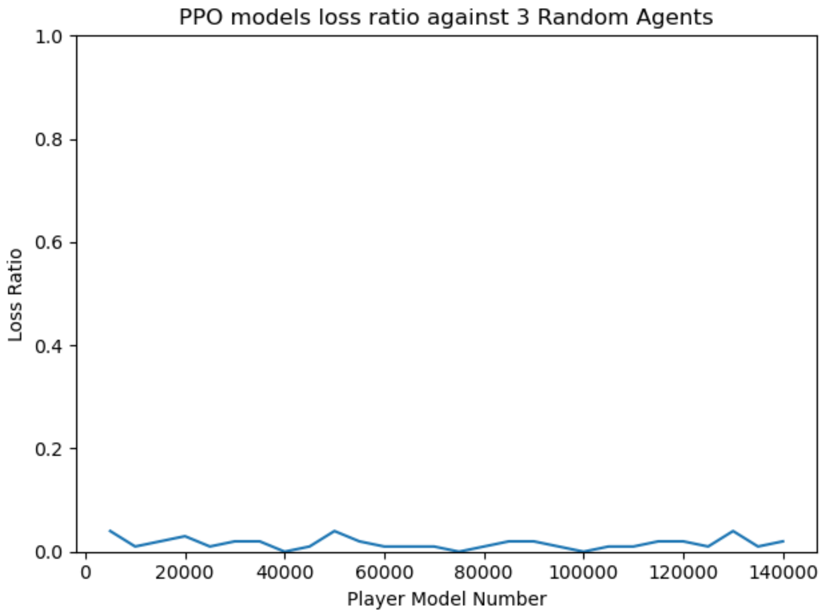


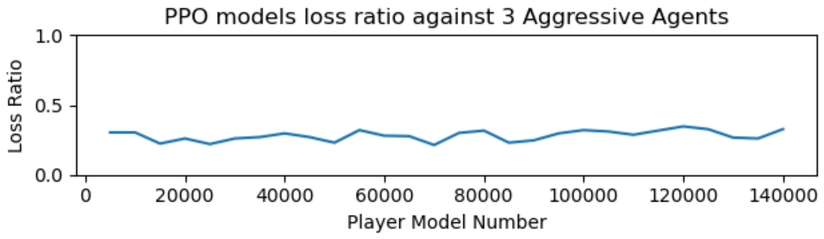

NFSP trained against previous iterations
VS. 1 random
VS. 3 randoms
Loss Ratio
Number of training games
0
10
20
30
40
50
60
70
80
0
1000
2000
3000
4000
5000

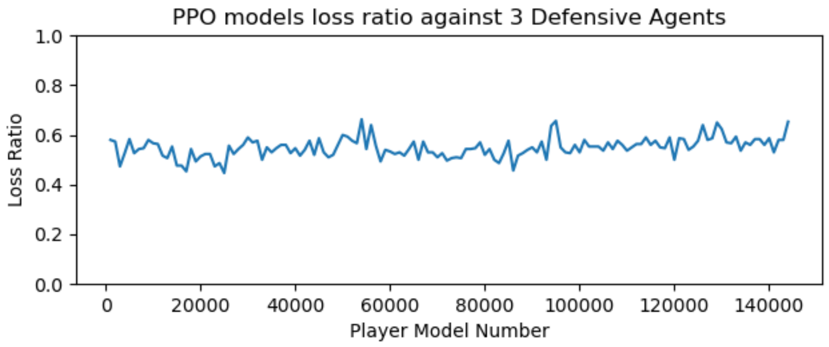

NFSP trained against random player
VS. 1 random
VS. 3 randoms
Loss Ratio
Number of training games
80
70
60
50
40
30
20
10
0
1000
2000
3000
4000
5000

PPO:


PPO models loss ratio against 1 Random Agent
Loss Ratio
Player Model Number
1.0
0.8
0.6
0.4
0.2
0.0
0
20000
40000
60000
80000
100000
120000
140000

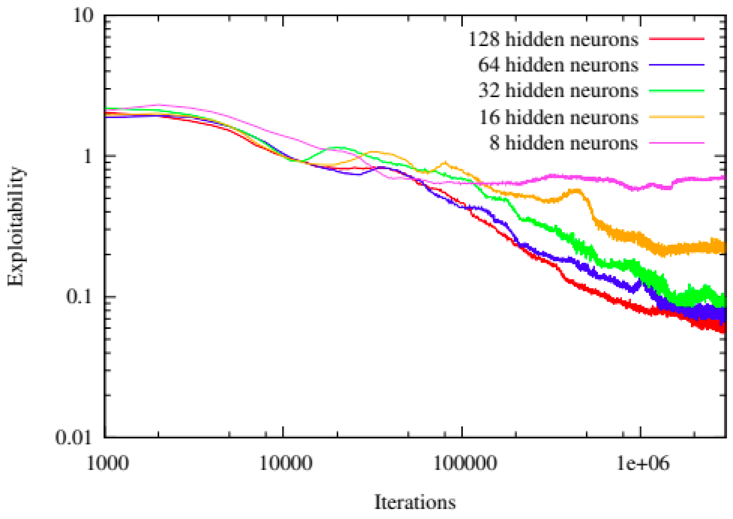

PPO models loss ratio against 3 Random Agents
Loss Ratio
1.0
0.8
0.6
0.4
0.2
0.0
0
20000
40000
60000
80000
100000
120000
140000
Player Model Number


PPO models loss ratio against 3 Aggressive Agents
Loss Ratio
1.0
0.5
0.0
0
20000
40000
60000
80000
100000
120000
140000
Player Model Number


PPO models loss ratio against 3 Defensive Agents
Loss Ratio
1.0
0.8
0.6
0.4
0.2
0.0
0
20000
40000
60000
80000
100000
120000
140000
Player Model Number

Conclusions

NFSP: according to the article which we based our work on, the deeper the network, the better the agent preforms. our experiments reinforce that claim as it can be seen in the first two graphs respectively, and if we had more computational power we could have reached better results. in the article we based our project on, most of the differences between the architectures manifested between 100000 and 1000000 iterations, which is several orders of magnitude above the amount of training we could reach during our limited time.



added here for reference is the comparison between different layer architectures from the original article.

also, to note, in the original article, NFSP was tested on the Texas hold'em variant of poker, which has much simpler board states and actions available to the player.

PPO: As can be seen from the graphs, after short training the PPO model already achieves excellent results against 3 random agents (~99% win rate).

Against 1 random agent the PPO model varies between 83-92% win rate.

These results show the robustness of PPO. After being trained on relatively small amount of games (140k compared to 100+ mil of Henry Charlesworth, 2018) the agent performed well against random agents, achieving an impressive win rate consistently.

References

Johannes Heinrich and David Silver (2016): Deep Reinforcement Learning from Self-Play in Imperfect-Information Games

Henry Charlesworth (2018): Application of Self-Play Reinforcement Learning to a Four-Player Game of Imperfect Information

Ziad SALLOUM (2019): Neural Fictitious Self-Play - *https://towardsdatascience.com/*