Department of Mathematics and Computer Science

University of Southern Denmark

Reinforcement Learning

Deep Poker RL

Author
Casper Juul Majgaard Nielsen
caspn18@student.sdu.dk

Author
Christoffer Falk Bøgebjerg
chboe17@student.sdu.dk

December 24, 2021

Contents

1 Introduction

Poker is an old gambling game that has been played and studied for a long time. Many people have gone broke but many people have also gotten extremely rich from their winnings. This report covers our implementation and results of *Algorithm 1* from the paper *Deep Reinforcement Learning from Self-Play in Imperfect-Information Games*[2], from here referenced as the NFSP paper. This paper only covers a Limit Hold'em implementation for 1 versus 1, commonly referred to as a heads up situation, but we also want to extend this for multiplayer (2-9). Limit Hold'em is a very restricted version of poker and we also want to explore some of the problems that may arise from implementing this algorithm for No limit Hold'em as this variant is more common and more lucrative to professional poker players.

2 Poker recap

Poker, otherwise known as Hold'em comes in many variants. All these variants share a common structure. This structure consists of alternating between revealing cards to the players and a betting round, allowing the players to make bets of how good their hand is compared to the others' hands. A player has to match the previous bets to stay in the hand. The reveal of cards are called as follows: Pre-flop players are dealt their own cards, flop 3 shared cards are revealed, the turn reveals 1 shared card and the river also reveals 1 shared card. After the betting round after the river the strength of the remaining players' hands are revealed and the winner(s) collects the bets. There are always 2 forced bets, a small blind and a big blind, for the first 2 players to act which rotates to prevent stagnation. The difficulty in poker consists of a combination between estimating your hand's strength and also evaluating the strength of the other players' hands. Depending on these evaluations you can try to maximize your overall winnings.

2.1 Limit Hold'em

For Limit Hold'em, referred to mostly as LHE, each betting round is limited to a maximum of 4 raises, meaning a player can increase the bet relative to a previous bet 4 times. The bet size is also very restricted in LHE. Here a player can only raise by 1 big blind during pre-flop and flop. During turn and river raises are increased to 2 big blinds.

2.2 No Limit Hold'em

For No limit Hold'em, referred to mostly as NLHE, there is no upper limit to the amount you can bet and there is no limit on how many times you can raise the pot. There is however a lower bound called the minimum bet size, min bet, which is equal to the previous raise.

There are more rules, but with this knowledge of poker is sufficient to understand this paper.

3 NFSP

NFSP uses two networks. 90% of the time it uses a very stable and slowly changing "average policy network" and 10% of the time it uses a simple Q-network. This is to get a more stable data distribution.

The NFSP agent uses two replay memories, MSL and MRL. MRL is the replay memory for the (reinforcement learning) Q-network. All actions, state, reward, next_state tuples are stored in this memory. MSL is the replay memory for the (supervised learning) average policy network. 90% of episodes are played out using the average policy network and the remaining 10% of episodes are played out using the Q network. Only the actions taken by the Q-network are stored in the MSL memory.

In deep Q-learning the parameters of the neural network, denoted by θ, can be step-wise updated by:

$$\theta_{t+1} = \theta_t + \alpha[R_{t+1} + \gamma \max_{a_{t+1}} Q_\theta(s_{t+1}, a_{t+1}) - Q_\theta(s_t, a_t)]\nabla Q_\theta(s_t, a_t)$$

However, when s_{t+1} is terminal, the corresponding Q-value, $\max_{a_{t+1}} Q_\theta(s_{t+1}, a_{t+1})$, is 0 always. This is implemented in the Q-network update function. Furthermore, we also use a stable Q network to predict the Q-value of the next state and action. This stable Q network is updated at a fixed interval.

The NFSP paper mentions that it uses SGD as optimizer and while we initially used it we encountered an entire training session that was ruining by NaN weights. After some research we discovered that a high learning rate might result in SGD outputting NaN, a problem named exploding gradients, which would feed back in the network later to produce NaN weights. After a serious reduction in learning rate we still encountered the problem and decided to use the Adam optimizer instead.

The NFSP paper also claims this approach would work for multiplayer instead of only heads up scenario. Since professional players approach a heads up much different to a normal 6 player game we also wanted to prepare for training networks for up to 9-player poker to confirm that a Nash equilibrium would be achieved and how the resulting play-style would differ compared to a network, only trained in a heads up scenario, in a 1 versus 1 game.

4 LHE Design

This section will cover the reinforcement learning relevant game design for LHE.

4.1 State

There are a total of 52 cards. An agent can only see its own two cards along with the cards on the table. The player's card are one-hot encoded into a vector of size 52. The game reveals three cards during flop, one card during turn and one card during river, each of the cards revealed in these phases are one-hot encoded in vectors of size 52. This means the representation of cards is a vector of size 208.

Both the state space and action space in LHE is discrete. There's up to 9 players, 4 rounds of play (pre-flop, flop, turn, river), 4 raises per round and only 3 actions to choose from. This means that the betting state can be represented by a tensor of size 9*4*5*3=540. The actions are one-hot encoded.

4.1.1 2 player LHE

The state space is heavily reduced for 2 player LHE. The smaller the state space is, the faster the agents can learn. In 2-player LHE the state space can be reduced to 2*4*5*3 since there only are 2 players. Furthermore, since the game ends if a player folds, this action is not needed as part of the betting state. Thus the state space in 2-player LHE is reduced to size 2*4*5*2=80.

4.2 Rewards

We had 2 ideas of how to feed rewards to the Q-Network. The first idea was to give it a single reward at the very end of the hand. This single reward would simply be the player's change in stack size e.g if you win 200BB we reward the Q-Network with 200. The second idea was to penalize the network every time it would put money in the pot e.g calling a raise of a single BB would give a reward of -1. We went with the first option since the second option would require more communication between the game and the agent as the the game would have to check if the desired action, call or raise, was possible or not, withdraw the call amount from the player and only then tell the network what the reward was.

5 Code design

For the first step of this project we needed a working version of poker. Initially we searched for open source poker implementations but we quickly realized that all the results suffered for at least 1 of 2 problems:

- 1. They weren't compatible/flexible in their encoding of state.
- 2. They were too slow to train networks in a reasonable amount of time.

Therefore we decided to implement poker on our own. This provided us the exact flexibility we needed to adjust the game state, and even create small optimizations depending on the amount of players, but also a fast running time that allows us train our networks efficiently.

In our code there are 3 folders. Each folder contains an implementation of a poker variant, Limit Hold'em or No Limit Hold'em, and optimizations depending on the amount of players, Multiplayer vs duo. Each folder also contains the relevant trained models, evaluation scripts for these models and the outcome of these evaluations. Furthermore we have a couple of setting scripts which were used for the training with easily adjustable parameters.

5.1 The game

The game initializes with some parameters, small blind value and a list of players. The players are prompted to prepare for the hand. Then they are then dealt their hand and are given a state relative to their own seating at the table. That is a player sitting in seat 5 will be given a state where their own actions appear from seat 0. The game continues with alternating betting round and card reveals until the winner is eventually found according to the rules of the poker variation. Then the game reports each players' winnings back to them.

5.2 The agent

We made an abstract class, or as close as you can get in python, for agents requiring them to implement 3 methods.

- 1. pre_episode_setup(): Void, which is used to signal the start of an episode. An agent can use this to reset counters, memory etc. between episodes.
- 2. get_action(state): Action, which is used to signal the desired action of the player.
- 3. get_result(reward): Void, which is used to tell the player how much they won on the hand.

For our naive agents these methods are very simple but for our NSFP agent they are more complex.

During the pre episode setup our NSFP agent would decide whether to generate actions using our average policy network or the Q network.

During the get action it would first update the state and if learning was enabled it would do a batch update for both the Q-network and the average policy network based on sample tuples drawn from their respective memory data structures covered below. After this it would simply feed the state to the policy in use and return the appropriate action.

Finally for the get result the networks will simply learn but here the pot winnings are used as a reward for the Q-network.

5.3 Data structures

For our NFSP agent we use 2 very important data structures for storing values for MRL and MSL. Initially they were just python arrays but since we only perform very restricted actions on them we can reduce the time complexity of these actions and therefore our model can learn faster.

5.3.1 Exponential Reservoir

The exponential reservoir is a data structure containing the n, often 3,000,000, last tuples of state and action drawn from the Q-network used for batch updating our average policy network. The NFSP paper references another paper, *Exponential reservoir sampling for streaming language models*[3], which guarantees that newer elements have exponentially higher probability of appearing compared to older elements when new elements are sampled with a constant probability. This simply means, that more recent data points have more weight than older data points. For this technique we only need to replace elements after the reservoir is full which is O(1) time complexity. This is much faster than our initial naive approach of deleting the first element and appending a new one, which ran in O(n) time.

5.3.2 Circular buffer

Our MRL is a data structure containing the n, often 600,000, last tuples of state, action, next_state, reward and a boolean if the state is terminal or not which is used for batch updating the Q-Network. When this storage is full it initially had to delete the first element which for python arrays is O(n) time complexity and then insert an element which is O(1) time complexity. By creating a circular buffer of size n we can simply keep track of "the first" element and replace it. This gives a time complexity of O(1).

6 Experiments

Agents approach Nash equilibrium after millions of iterations, meaning only one experiment can be carried out every 12-16 hours for very small networks and multiple days for larger networks.

For reference, figure 2 in the NFSP paper shows that the agent is still learning after 20 million iterations and stops learning at around 35 million. Their agent used 4 fully connected dense layers with 1024, 512, 1024, 512 neurons in each. We will train with smaller networks.

Note that we switch between two strategies of updating the networks. In the NFSP paper they update the networks of the NFSP agent twice every 256 steps in the game. For some experiments we also do this, but for others we update the networks on every step. What strategy we choose to use for a model will be explicitly stated.

6.1 LHE Duo

The first attempt at training an agent used a very large network with the following amount of neurons in the four layers: [1024, 512, 1024, 512]. The training was very slow, even on a GPU it would take approximately 24 hours to get just one million data points, so we optimized the code a lot and also trained with smaller network sizes.

6.2 LHE Multi

For LHE Multiplayer (2-9 players) we did perform any experiments. The first reason for this is a slight time restriction. Since we would have 4 agents to train on average (uniform distribution between 2-9 players) it would take at least double the time of our 2 agents to achieve valuable results. Another big restriction was our hardware. We were running our training on a computer with 32GB of RAM if we were using the CPU and 12GB if we were using the GPU. Our LHE Duo were already filling up 19 GB of RAM on the CPU training and the GPU training crashed because it ran out of memory. This means we would have to considerably reduce the MSL and MRL sizes for our agents to the point where we believe it would impact our results.

7 Results

When our models finish trianing we run three tests always. We test the model against two very simple agents. One that always raises and one that always calls. Furthermore the model plays against earlier versions of itself. Finally, we run these three tests on both networks of the model - the average policy network and the Q network.

Note that on the y-axis of all graphs below mbb/hand stands for milli big blinds per hand.

7.1 model_id=2000 (NFSP small network update every step)

We wanted to see if a model with a very small network could reach a stable point of play.

```
/* Parameters: */
ANTICIPATORY_PARAMETER = 1
NETWORK_SIZE = [128, 64, 128, 64] # Neurons in each layer
RL_LR = 0.1    # Reinforcement learning rate
SL_LR = 0.01   # Supervised learning rate
MRL_SIZE = 500_000 # Reinforcement learning memory
MSL_SIZE = 1_000_000  # Supervised learning memory
BATCH_SIZE = 256
TARGET_POLICY_UPDATE_INTERVAL = 1000 # Q_stable <- Q
START_EPS = 0.12
EPS_DECAY = 1_000_000 # Steps before eps=0
```

This model updated the networks once for every step in the game.

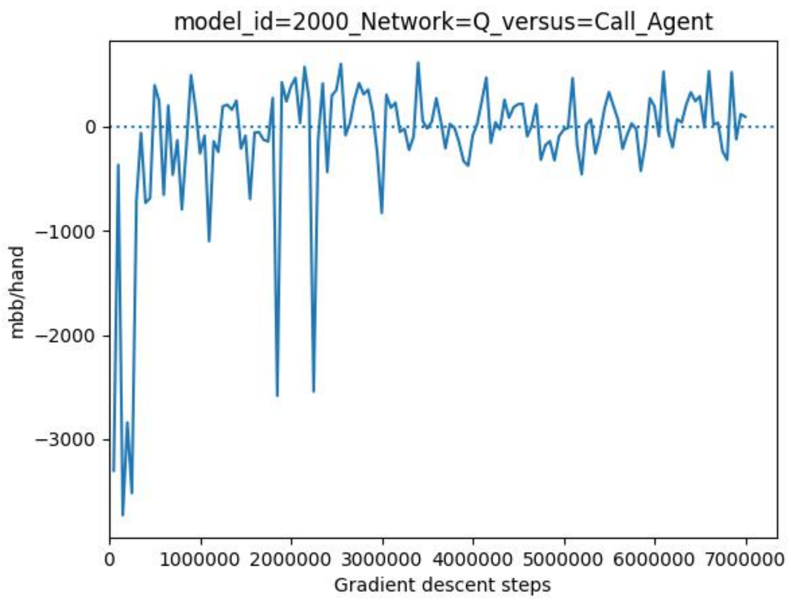

Figure 1: Q network versus Call_Agent.

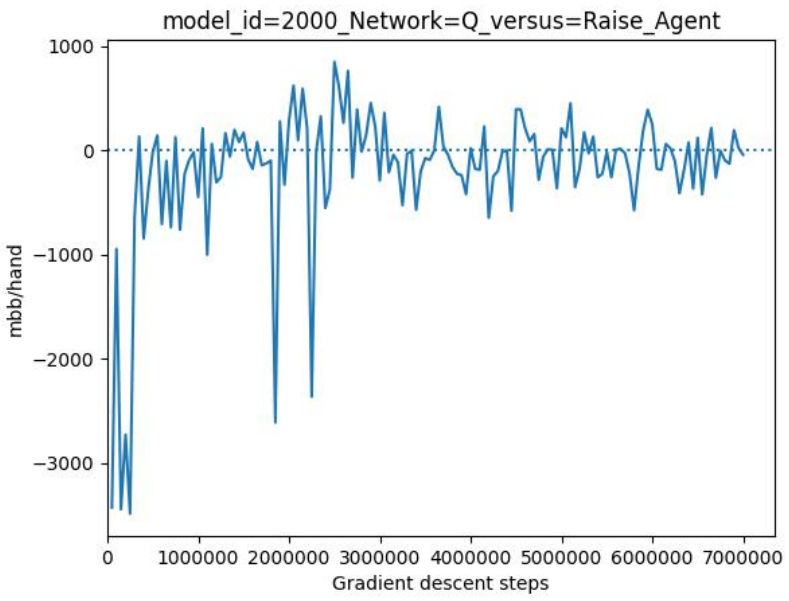

Figure 2: Q network versus Raise_Agent.

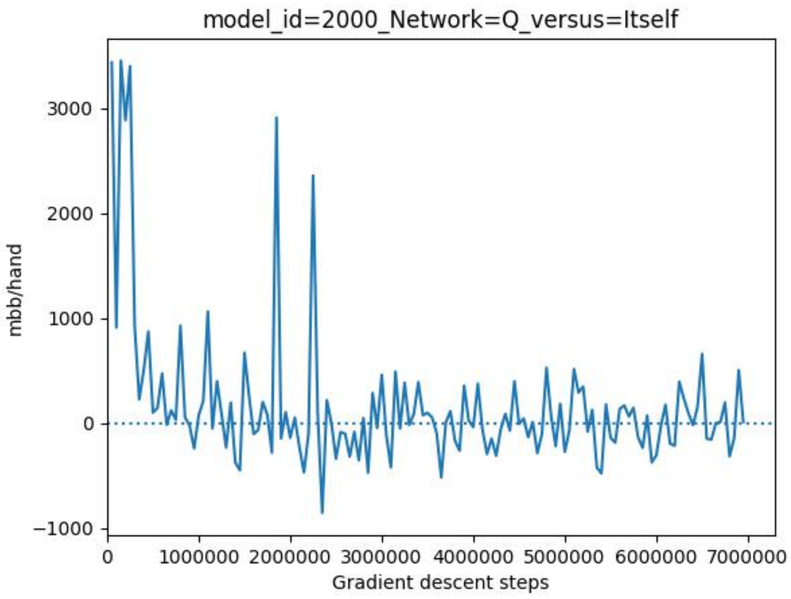

Figure 3: Q network at iteration=7,000,000 versus earlier versions of itself.

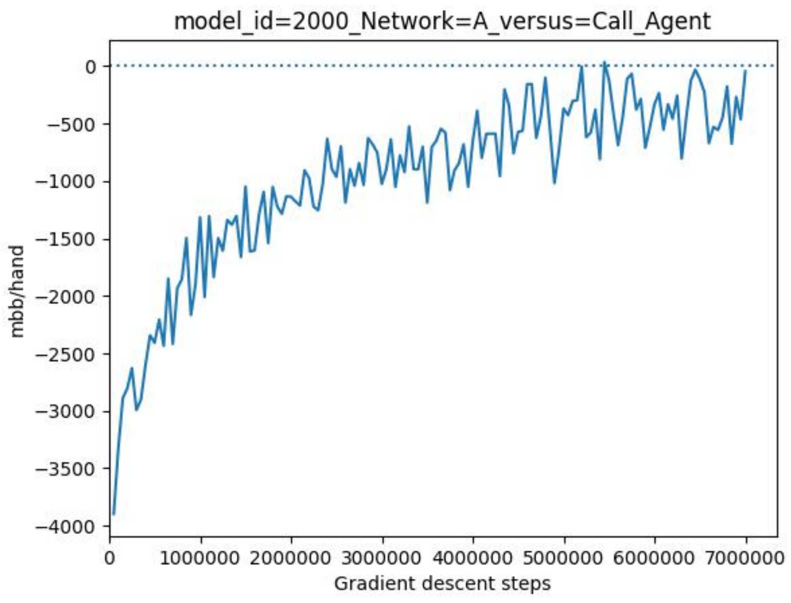

Figure 4: Average policy network versus Call_Agent.

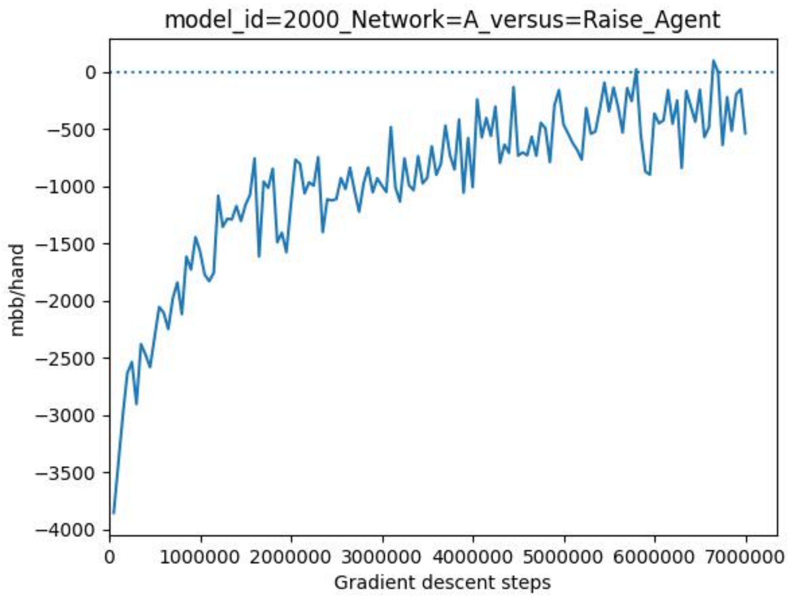

Figure 5: Average policy network versus Raise_Agent.

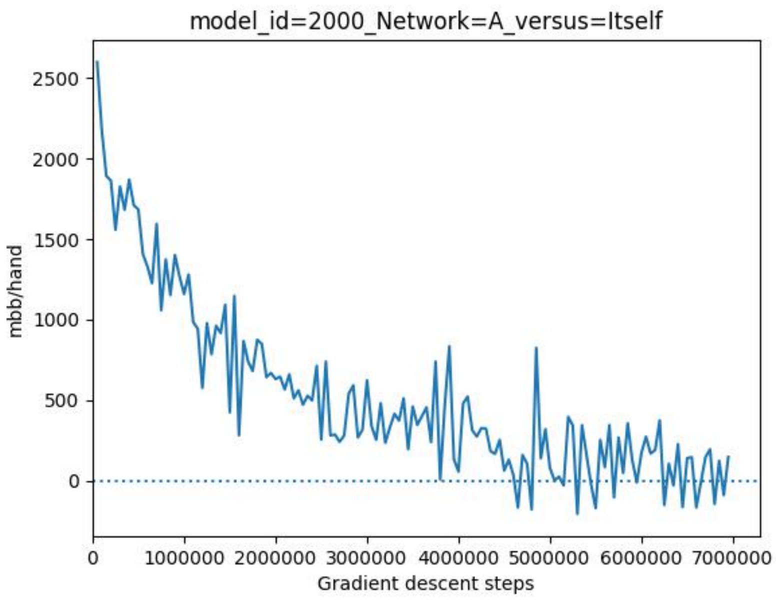

Figure 6: Average policy network at iteration=7,000,000 versus earlier versions of itself.

7.2 model_id=2100 (NFSP large network update twice every 256 steps)

We wanted to see if NFSP approaches Nash equilibrium for a big network also. We did however also use the update strategy presented in the NFSP paper which is two gradient descent steps every 256 in-game steps, instead of just performing one gradient descent step for every step in the game.

```
/* Parameters: */
ANTICIPATORY_PARAMETER = 1
NETWORK_SIZE = [512, 256, 512, 256] # Neurons in each layer
RL_LR = 0.1    # Reinforcement learning rate
SL_LR = 0.01   # Supervised learning rate
MRL_SIZE = 500_000 # Reinforcement learning memory
MSL_SIZE = 1_000_000  # Supervised learning memory
BATCH_SIZE = 256
TARGET_POLICY_UPDATE_INTERVAL = 1000 # Q_stable <- Q
START_EPS = 0.12
EPS_DECAY = 1_000_000 # Steps before eps=0
```

This model updated the networks once for every step in the game.

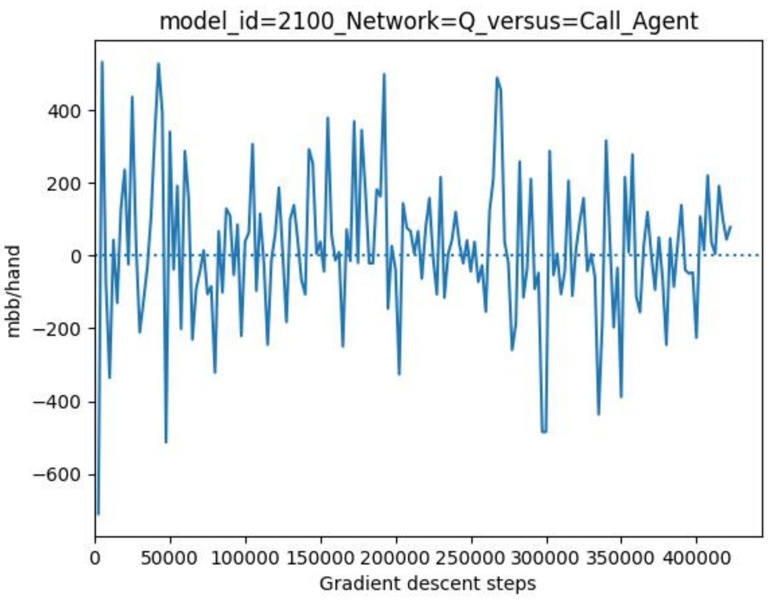

Figure 7: Q network versus Call_Agent.

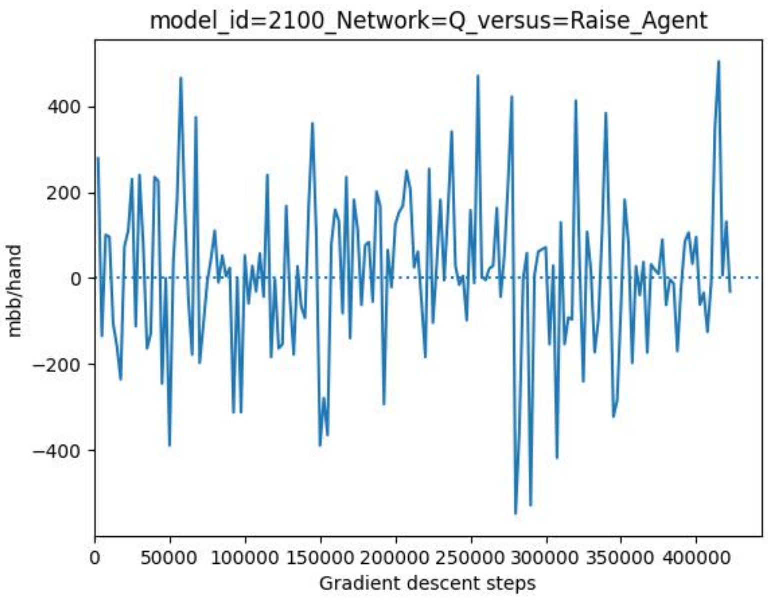

Figure 8: Q network versus Raise_Agent.

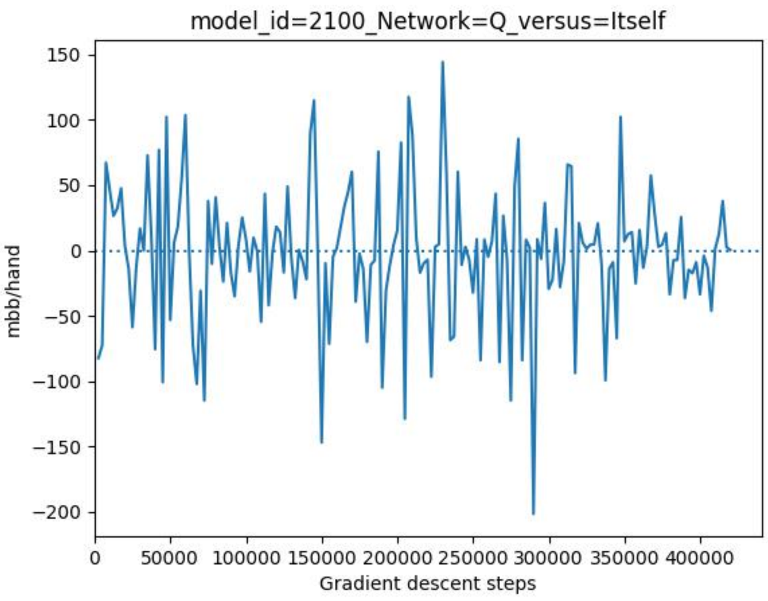

Figure 9: Q network at iteration=442,500 versus earlier versions of itself.

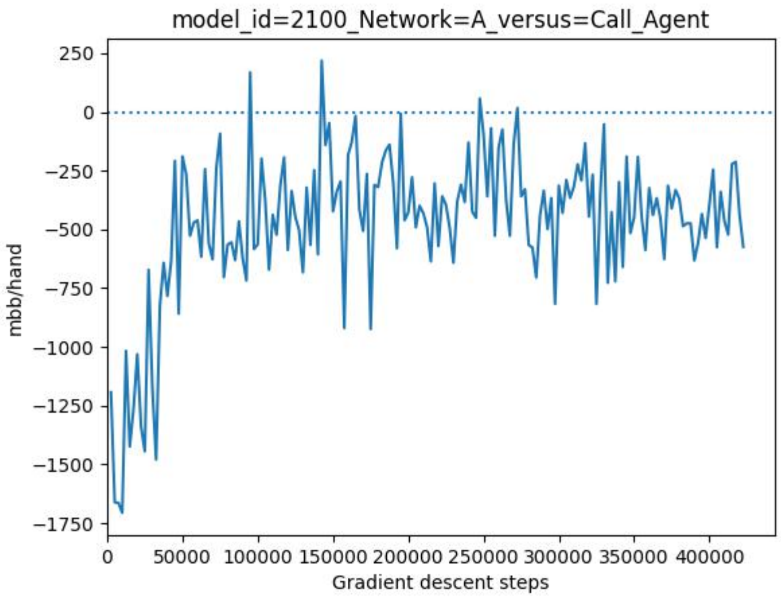

Figure 10: Average policy network versus Call_Agent.

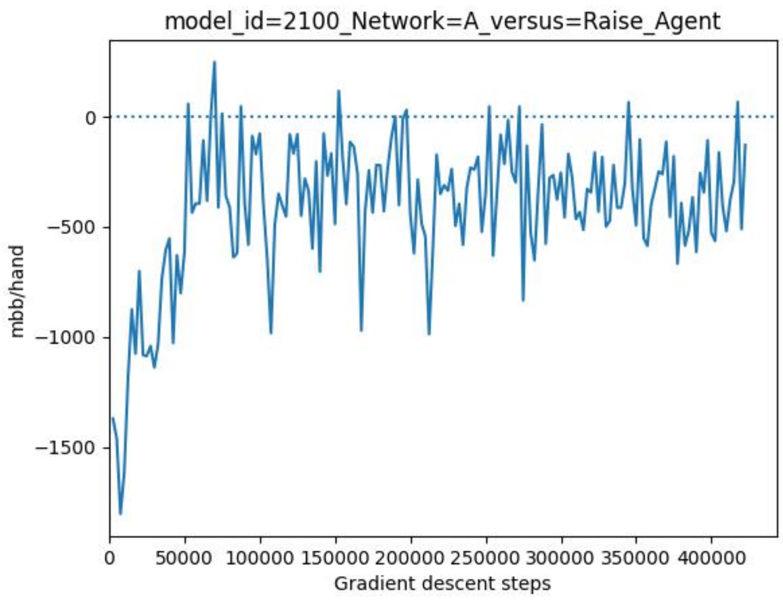

Figure 11: Average policy network versus Raise_Agent.

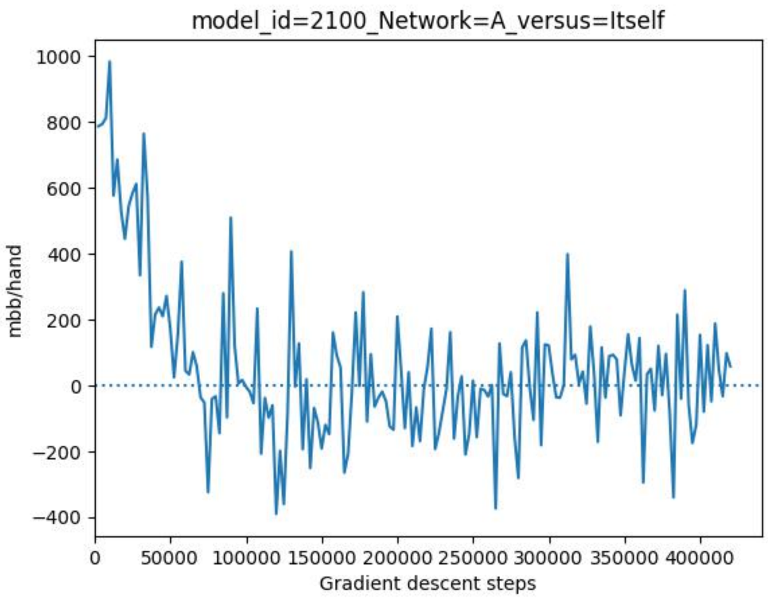

Figure 12: Average policy network at iteration=442,500 versus earlier versions of itself.

Either the update method (two gradient descent steps per 256 in-game steps) or the increased network sizes caused the model to fail. The Q-network is super unstable, and the average policy network stops improving already after 1 million steps.

7.3 model_id=2200 (DQN agent update twice every 256 steps)

This model was trained with anticipatory parameter set to 1, meaning that it only chose actions following its Q-policy network only.

```
/* Parameters: */
```

```
ANTICIPATORY_PARAMETER = 1
NETWORK_SIZE = [512, 256, 512, 256] # Neurons in each layer
RL_LR = 0.1    # Reinforcement learning rate
SL_LR = 0.01   # Supervised learning rate
MRL_SIZE = 500_000 # Reinforcement learning memory
MSL_SIZE = 1_500_000  # Supervised learning memory
BATCH_SIZE = 256
TARGET_POLICY_UPDATE_INTERVAL = 1000 # Q_stable <- Q
EPS_START = 0.15
EPS_DECAY = 3_000_000                # Steps before eps=0
```

This model updated the networks twice every 256 steps in the game. Only updating the networks twice every 256 steps in the game leads to very slow data collection, and these 120,000 points were collected over a period of 16 hours.
Notice, that even though this agent only chooses actions from its Q-policy, it still updates the average policy network (the network that predicts its best past response behaviour).

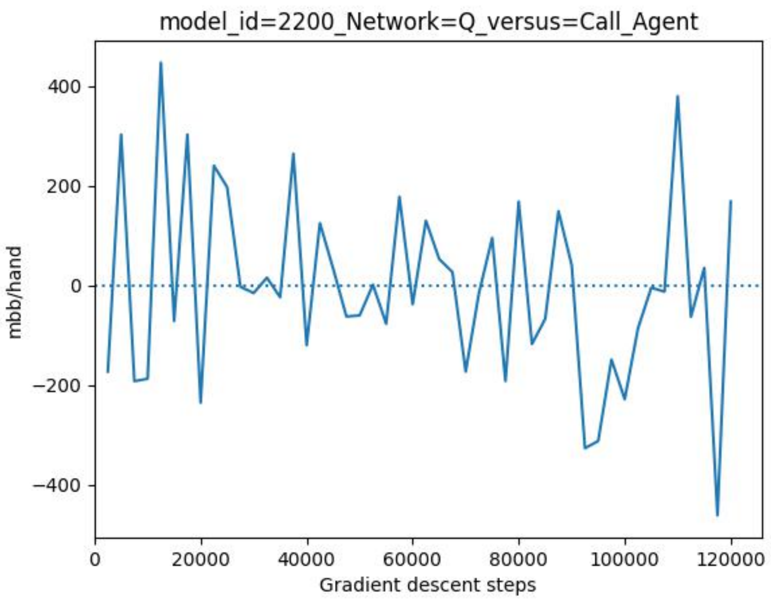

Figure 13: Q network versus Call_Agent.

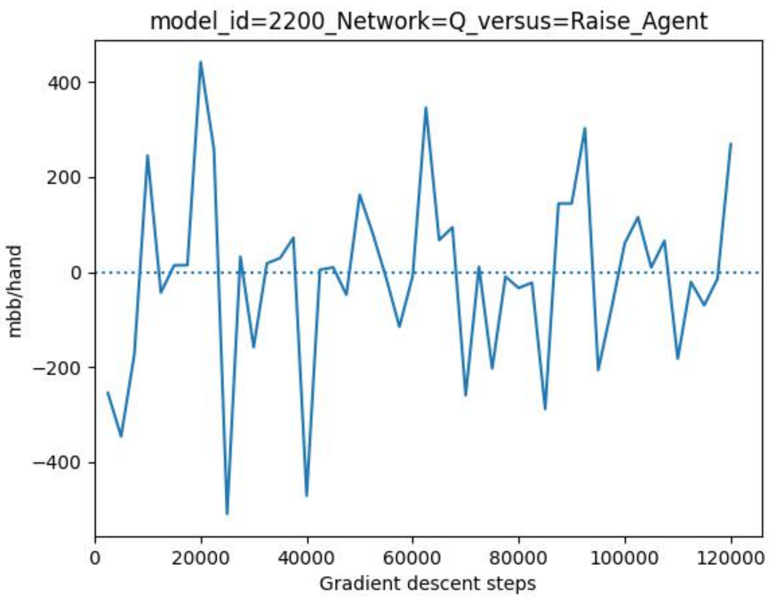

Figure 14: Q network versus Raise_Agent.

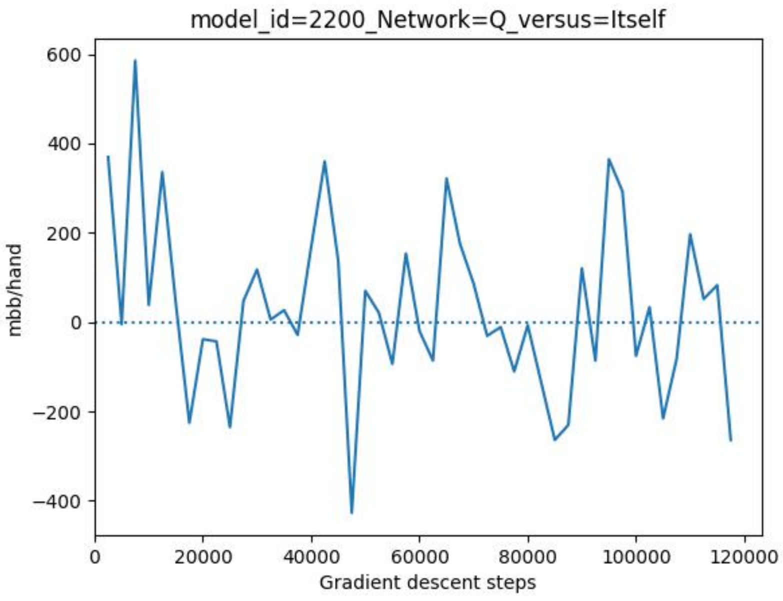

Figure 15: Q network at iteration=120,000 versus earlier versions of itself.

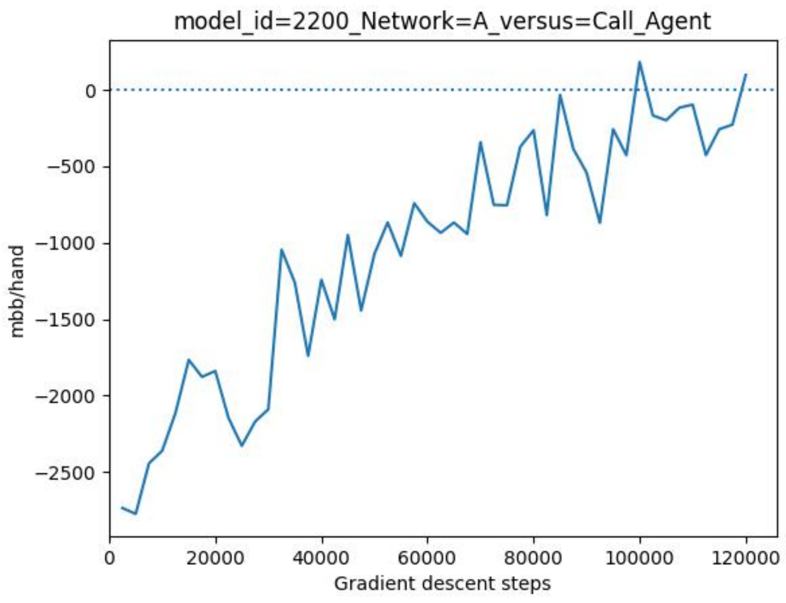

Figure 16: Average policy network versus Call_Agent.

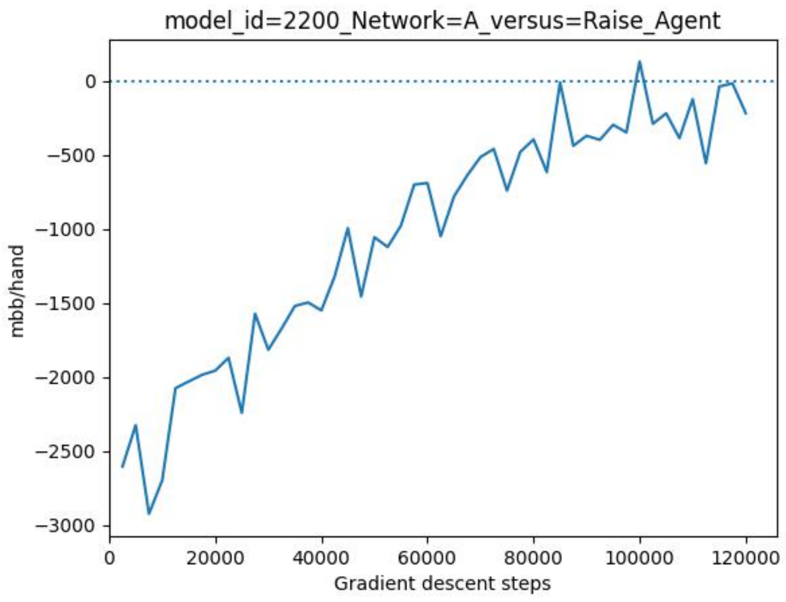

Figure 17: Average policy network versus Raise_Agent.

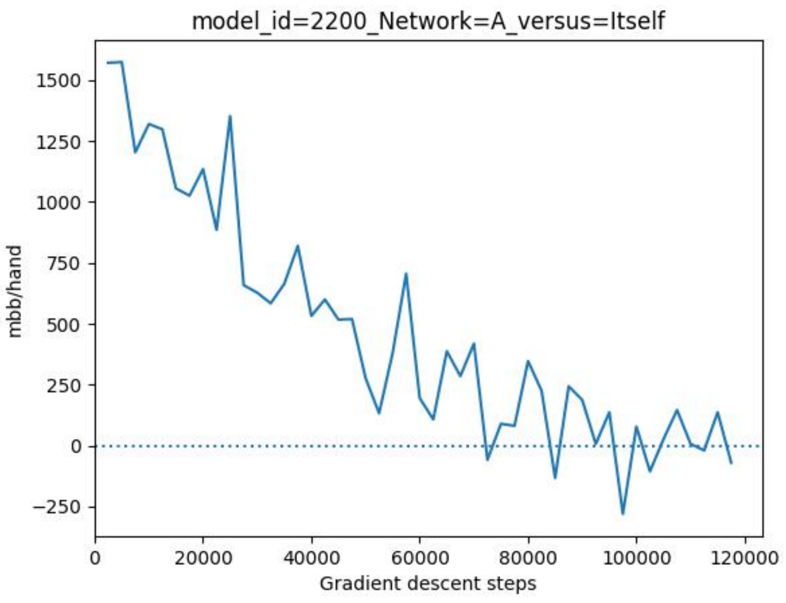

Figure 18: Average policy network at iteration=120,000 versus earlier versions of itself.

It is obvious from the results of the DQN agent (model_id=2200), that its Q-network is not improving, when playing against itself, nor is it improving when playing against the simple rule-based agents. However, the average policy did seem to improve in a stable manner, but this network is only mimicking what the Q-network has been doing in the past, so if the Q-network is performing poorly in the long run, the average network policy should also be performing poorly in the long run. If the DQN agent was trained for longer, this might have become obvious, but we cannot say for certainty without more data.

7.4 model_id=2300 (NFSP large network update every step)

This model is a copy of model_id=2000 except that its network is larger, hopefully making it able to learn more.

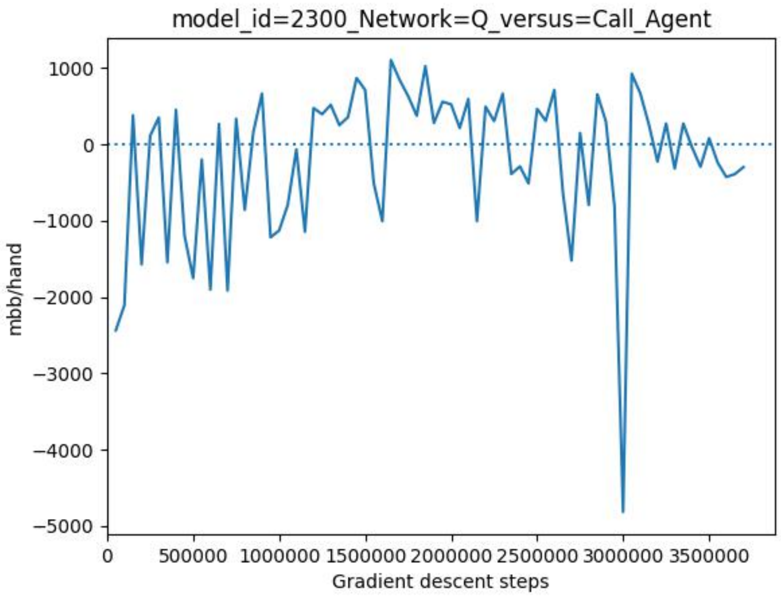

Figure 19: Q network versus Call_Agent.

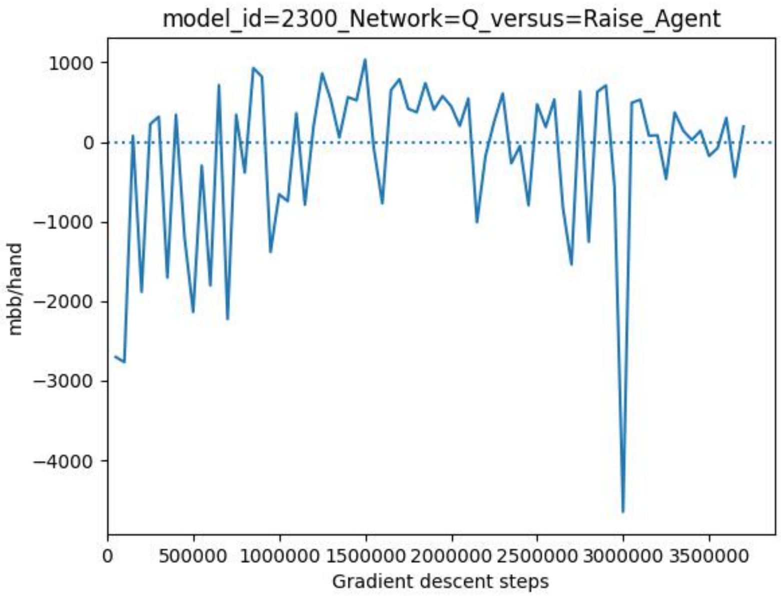

Figure 20: Q network versus Raise_Agent.

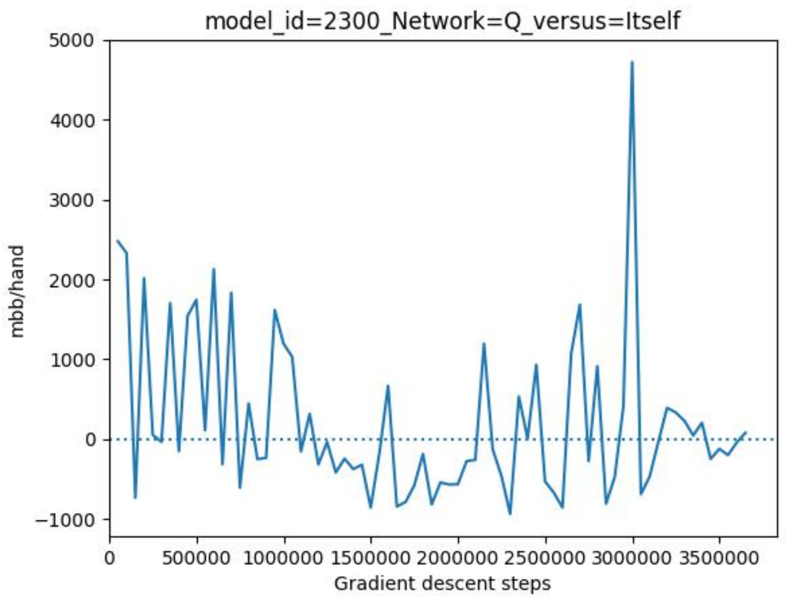

Figure 21: Q network at iteration=3,700,000 versus earlier versions of itself.

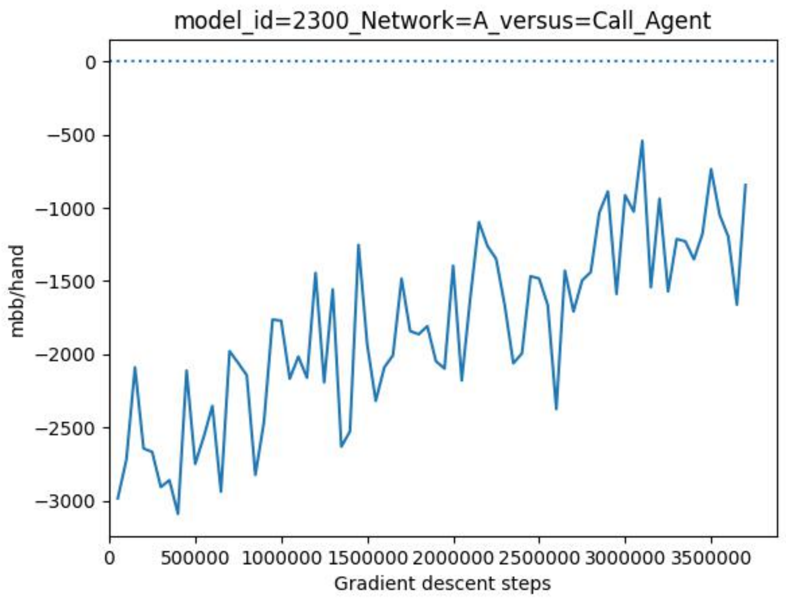

Figure 22: Average policy network versus Call_Agent.

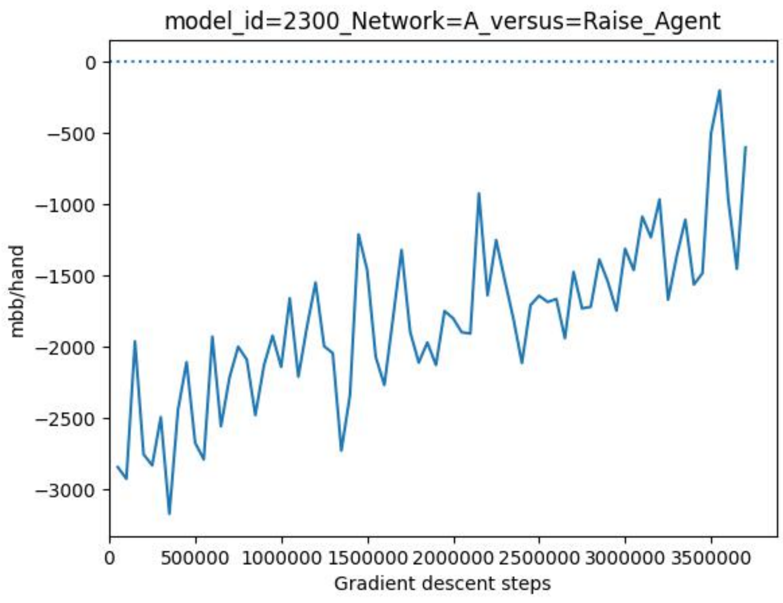

Figure 23: Average policy network versus Raise_Agent.

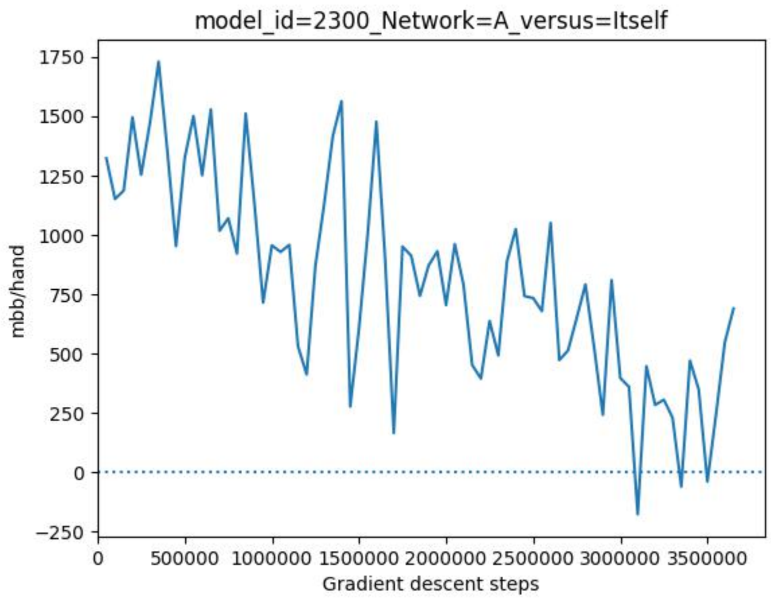

Figure 24: Average policy network at iteration=3,700,000 versus earlier versions of itself.

This model's average policy network definitely still was improving at the time training was stopped.

7.5 Final remarks on results

A possible flaw with doing one gradient descent step per step in the game, is that not a lot of memory is generated to store in MSL. The MSL exponential reservoir never manages to fill up during the entire training of this model, and some of the earliest (and very bad moves) from the Q-network are stored in this reservoir and never replaced before ending training.

It seems puzzling, that a trained NFSP agent can lose after millions of iterations to such simple rule based agents as the call and raise agents, but after reading another recent study on adversarial AI training, it seems that there is a simple and perhaps obvious reason to it. This problem is in fact general to adversarial agent training.

In the paper *Adversarial Policies: Attacking Deep Reinforcement Learning*[1] adversarial agents are trained against each other in various tasks, and it shows that a trained agent has a much lower win-rate against an "unintelligent" agent compared to an "intelligent" one.

For a nice visual example of this phenomenom check the paper's github frontpage: `https://adversarialpolicies.github.io/`

Or watch this youtube video explaining it from 03:05 to 04:33: `https://youtu.be/u5wtoH0_KuA?t=185`

This is due to the fact, that the "unintelligent" AI induces off-distribution data (data that is never or rarely seen during training). It might be the case, that the raise and call agents also induce off-distribution data also, because NFSP agents risk the most when they are going all-in on a hand, and this does probably not happen a lot during training.

A potential way of getting more all-in situations is to lower the epsilon to a fixed value of let's say 5% instead of 0. This would ensure more exploration and less exploitation.

8 Conclusion

We have implemented a functioning version of LHE which we believe runs very fast. We have also implemented the NFSP paper *Algorithm 1* with a few minor adjustments for LHE multiplayer and for heads up with slight performance increase. Using our poker game and our NFSP agent we have trained networks that we found to improve dramatically over time only being restricted by training time and network size. This is demonstrated by comparing the results for model 2000 to the results of model 2300 which did not show signs of plateauing after 3.7 million steps. Furthermore we have trained models that suggests that 2 DQN agents will not reach a Nash equilibrium just using their Q-networks, because their actions become highly correlated over time.

9 NLHE Extension

We have left this section after the conclusion of the paper as this is still a work in progress. The original goal, although not mentioned in our project proposal, was to create a strong agent for NLHE. We believed that implementing LHE was a necessary step on the way to gain more knowledge about the NFSP algorithm with its current constraints. It is important to note that we did not get a working version of the NLHE agent yet but we still wanted to explain the theoretical solutions to some of the issues that arose during our current development of NLHE. For NLHE we had to make 3 adjustments:

- Obviously the poker rules must be adjusted
- The state needs to be changed as it can no longer be contained in a tensor of size 80 (for LHE Duo)
- The agents must now not only decide on an action but also on a value, action value

9.1 The State

Since the betting state could be theoretically infinite, only limited in practicality by the amount a player decides to buy in for, it is not possible to feed this information to a neural network. Our solution to this was fairly simple. We feed this state to a recurrent neural network, RNN, and use its latest latent state as a representation of the current betting history. This RNN outputs 100 variables which we believe will adequately describe the state. The amount of variables was arbitrarily chosen and would likely be prone to change after testing with different amounts. The state we feed to the RNN is list of tuples containing the following information:

- ti, the relative table index to the current player/agent
- round, the index of the current betting round
- action, the action that player took, 0, 1 or 2
- action_value, the value that player used
- start_stack_size, how big of a stack this player started the hand with
- current_stack_pct, how big a percentage of the initial stack this player has

9.2 Deciding action value

When raising in NLHE any number between minimum bet size and your current stack size can be the action value. Thus the action space in NLHE is continuous. This means that not only do we have to output probabilities of choosing actions, but we also need to output an action value. This can be achieved by using a multi-head output in the both the Q-Network and the average policy network. Since this action value is continuous we need to use policy gradient methods to update the network parameters, and we originally proposed to use the REINFORCE algorithm, but we had one issue.

In an episode with more than one step, an action and an action value for each state is generated, and the gradients are stored. However, after using the gradients from the first action and action-value to perform an update on the Q-network's parameters, the gradients for all the other action and action values change, and can no longer be used. This can be solved by just feeding the state to the network again and getting a new action value and corresponding gradients, but this new action value might have changed the reward trajectory of the episode, and cannot be used for learning.

References

[1] Adam Gleave et al. “Adversarial Policies: Attacking Deep Reinforcement Learning”. In: *CoRR* abs/1905.10615 (2019). arXiv: `1905.10615`. URL: `http://arxiv.org/abs/1905.10615`.

[2] Johannes Heinrich and David Silver. “Deep Reinforcement Learning from Self-Play in Imperfect-Information Games”. In: *CoRR* abs/1603.01121 (2016). arXiv: `1603.01121`. URL: `http://arxiv.org/abs/1603.01121`.

[3] “Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics, ACL 2014, June 22-27, 2014, Baltimore, MD, USA, Volume 2: Short Papers”. In: The Association for Computer Linguistics, 2014. ISBN: 978-1-937284-73-2. URL: `https://aclanthology.org/volumes/P14-2/`.