Advanced Reinforcement Learning: Skyjo Project Report

Sebastian Bieber, Florian Chen, Henry Forman

February 2025

1 Introduction

This report presents our Advanced Reinforcement Learning course project, where we train agents to play the card game Skyjo, a turn-based game that involves strategic decision-making under uncertainty. The primary objective of the game is to minimize one's total card sum while making informed choices about drawing, replacing, and discarding cards. This introduces several challenges for an AI agent, such as optimizing long-term rewards, managing hidden information, and adapting to the actions of opponents.

The focus of this report is to detail the methodology used to train and evaluate our Skyjo-playing agents. We provide a comprehensive overview of the implementation, including the design of the game environment, the training process, and the evaluation of different agent strategies. We explore various reinforcement learning techniques, including self-play, to improve the agent's ability to make optimal decisions. Additionally, we examine different reward structures and determine their impact on training efficiency and performance.

In related work, Michael Feil developed a multi-agent environment for playing Skyjo[1]. However, this implementation contains mistakes in the game's mechanics, and, due to simplistic rewards and a suboptimal observation structure, training fails and cannot produce competent policies. Guillaume Barthe created a simpler single-agent environment using `stable-baselines`[2], where a single agent trains with fixed rewards while playing against a static policy. This approach enables training strong agents that achieve human-level play, although only in a single-agent setting. In contrast, our implementation[3] supports up to eight players (Skyjo's maximum), offers multiple observation schemes in varying expressivity, allows dynamic reward adjustments during training, and outperforms the previous state-of-the-art in direct comparisons.

The structure of this report is as follows. In Section 2, we describe the rules and objectives of Skyjo. Next, Section 3 covers our methods for representing actions and observations, reward assignment during training, the algorithms used to train agents, and the training setup. In Section 4, we evaluate the training process and discuss how different parameter settings influence agent performance. Section 5 outlines our GitHub repository structure and serves as a user guide for training, evaluating, and playing against trained agents. Finally, Section 6 concludes the report.

2 Background

Skyjo[4] is a turn-based card game for 2-8 players. Each card shows a number from -2 to 12. For the set-up, each player starts with 12 random cards laid face-down before them. Before the game starts each player reveals two of their cards. The rest of the shuffled cards are placed in a draw pile, from which the topmost card is revealed and forms, face-up, the discard pile.

[1] https://github.com/michaelfeil/skyjo_rl
[2] https://github.com/Guillaume-Barthe/Skyjo
[3] https://github.com/faguodev/SkyjoAI
[4] Rules can be found here https://www.officialgamerules.org/card-games/skyjo [EN] (visited 5. Feb. 2025) or here https://kleopas.de/skyjo/ [DE] (visited 5. Feb. 2025)

In each turn, the active player can choose to either draw the topmost card from the discard or draw pile. The placer can then choose between two actions: Either replace one of their cards with the drawn card, placing the drawn card face-up in its place and discarding the replaced card. Or discard the drawn card onto the discard pile and reveal one of the face-down cards. The game concludes with one final round of turns once the first player has revealed all their cards. Columns with same card values are eliminated. The player with the fewest cumulative points wins the game. An example game state is displayed in Figure 1.

In this game, an agent has to learn to properly draw new cards and replace its own cards to minimize its card sum. Additional difficulties include trying to eliminate columns, not discarding cards that might help an opponent, and being wary of the number of face-down cards of opponents such that the agent does not find itself with many face-down cards at the end of the game as these cards could potentially be very detrimental to the overall score.

Figure 1: An example game state for a Skyjo player.

3 Methodology

This section details our implementation of the Skyjo environment, encompassing the action space, observation space, reward schemes, and training approaches. It is based on Michael Feil's environment.

3.1 Actions

Our Skyjo environment implements a discrete action space comprising 26 possible actions, indexed from 0 to 25. Each agent's turn is divided into two sequential steps that require fundamentally different types of actions, with valid choices indicated by an action mask.

During the first step, the agent must acquire a card either by drawing randomly from the main pile (action index 24) or by taking the top-most face-up card from the discard pile (action index 25). In the second step, the agent faces a choice between two possibilities: they may either place the drawn card by exchanging it with one of their twelve cards (corresponding to action indices 0-11) or discard the drawn card and reveal one of their hidden cards (corresponding to action indices 12-23).

3.2 Observations

The agent's observations are structured to provide both indirect and direct information about the game state. Indirect information is incorporated to help the agent anticipate the game's progression and assess potential risks. Specifically, the agent is given the lowest card sum among all players and the lowest number of unrevealed cards held by any opponent. These indicators serve as proxies for estimating the likelihood of the game ending soon. Additionally, to track the distribution of remaining cards in play, the observation space includes a count of how often each card value has already appeared.

The nature of the agent's observation depends on whether it operates under direct or indirect observation settings. In the direct observation setting, the agent has full visibility of all opponents' cards, allowing it to make precise strategic decisions based on complete information. Under indirect observation, the agent does not have access to individual opponents' card values and instead only observes their own cards.

To encode card information, three different representation schemes are considered. The simplest (in our environment called "simple") approach provides the agent with one numeric input for each card, given as the card value, with unrevealed cards represented by a default average value of 5. A more informative approach employs a fully one-hot encoded representation (called "one-hot"), requiring 17 input units per card — 15 to represent specific card values, plus additional indicators for hidden and eliminated statuses. Lastly, an effective intermediate representation (called "efficient_one_hot") balances informativeness and efficiency by using one-hot encoding only to indicate whether a card is hidden, while otherwise maintaining the simple observation scheme of one input unit for the card value.

We furthermore adapted the simple and efficient one-hot encoded observation schemes to fit a trained model in the different single-agent environment from Guillaume Barthe[5]. In our environment, these observation schemes are called by their normal name + `_port_to_other`. This allows us to let our agents play against agents trained in the other environment directly.

3.3 Rewards

The rewards give the trained agent "feedback" on how well it is doing. Since the agent tries to maximize the reward, it is substantial to choose the rewards in such a way, that the agent gets rewards for completing its objective. At the same time, one has to be careful not to choose rewards, such that the agent can learn to exploit the environment without achieving its objective.

In our case, the objective of the agent is winning the game. Therefore the first scheme only gives a positive reward when the agent wins the game, otherwise, a negative reward is given. In our environment, the reward is by default set to 100, which is also the value we used for training. While this simple reward scheme introduces no bias by making assumptions on what a good strategy might look like, rewards are only given very sparsely. Thus, the agent has to complete an entire game before getting any rewards. Hence, it may be difficult to identify actions that most contribute to winning and the model would train very slowly.

To address the slow training observed from the first reward scheme, the second scheme assumes we know an "optimal" strategy to win the game, and using this gives rewards in every step. This scheme builds on the assumption, that a good turn is one where the agent lowers the total card sum of its cards. At the end of a game, the agents receive the same "end-of-game" reward as before, and additional rewards are given for every step using the following formula:

$$R = (\texttt{card\_sum}_{\texttt{before}} - \texttt{card\_sum}_{\texttt{after}} + \texttt{curiosity\_reward}) \times \texttt{action\_reward\_reduction} \tag{1}$$

R is the reward given to the agent. The reward given is the difference of the card sums before and after the agent's action. Furthermore, a reward for "curious" actions (`curiosity_reward`) can be given, which will be added only if the agent discards its taken card and reveals a new card. This should simulate the value gained by the agent having more knowledge about its own cards. `action_reward_reduction` denotes a scaling parameter, which can be used to give more or less emphasis on the every-step rewards.

These short-term rewards should lead to an initial faster improvement of the policy. However, it could hinder the policy's training later on, as the agent learns to exploit the round-by-round rewards instead of winning games. To mitigate this, we implemented a callback function that dynamically decreases the `action_reward_reduction` parameter over time, which we call `action_reward_decay`. This gives the agent guidance early on in the training and then lets it develop its own strategy as the training advances.

3.4 Algorithms

In this subsection, we present details of the reinforcement learning algorithm(s) and architectures we used to train our agents. We mainly focused on Proximal Policy Optimization (PPO) as it is a well-established

[5] https://github.com/Guillaume-Barthe/Skyjo

method and the de-facto standard in `rllib`, the Python library used in our project.

3.4.1 Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO) is an on-policy learning algorithm that alternates between sampling epochs of minibatches from the environment and optimizing a surrogate objective function using stochastic gradient ascent. To understand the PPO objective function, it is useful to first consider the history and motivation behind its development. Although Deep Q Networks (DQNs) achieved great early successes, such as learning to play Atari games [Mnih et al., 2013], they were prone to instability and large, irrecoverable changes to the policy function. To address this issue, Trust Region Policy Optimization (TRPO) [Schulman et al., 2015] was introduced. TRPO optimizes the relative probability of choosing advantageous actions while constraining overall policy changes. By enforcing this constraint, large policy shifts are prevented, improving learning stability. However, TRPO's implementation, tuning, and computation are complex, making real-world applications challenging.

PPO [Schulman et al., 2017] is an approximation of TRPO with more favorable computational properties. Like TRPO, it optimizes the relative likelihood of choosing advantageous actions while enforcing constraints to prevent drastic policy updates. However, instead of a KL-divergence-based constraint, PPO introduces a clipped surrogate objective that simplifies training.

To formally define the objective function, we introduce some notation. Let

$$r_t(\theta) := \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}$$

denote the ratio of action probabilities under the new and old policies, and let

$$A_t(s, a) := Q_t(s, a) - V_t(s)$$

represent the advantage function, or the relative benefit of choosing an action a. For brevity, we will also just write A_t. The PPO objective is then given by

$$J_{\text{CLIP}}(\theta) = \hat{\mathbb{E}}_t \left[\min \left(r_t(\theta)\hat{A}_t, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}_t \right) \right], \tag{2}$$

where ϵ is a hyperparameter, typically set to 0.1 or 0.2. To maximize this objective, the agent iteratively improves its policy by increasing the probability of actions with high advantage values and decreasing the probability of less advantageous actions. However, the potential improvement is capped using the clipping function, and drastic changes to the relative likelihood will not increase the objective beyond a certain point. Yet, the opposite does not hold, and, as the minimum of the clipped and unclipped objectives is taken, drastic changes can lead to unbounded decreases in the objective. As a result, agents trained using PPO make small and safe updates to their policy functions, thus preventing instability in training.

As an implementational detail, it is worth mentioning that PPO is an actor-critic method in that it uses two neural networks: one for learning the policy and another for estimating the value function. In practice, it is often assumed that these two networks make use of similar features, thus a shared network with different prediction heads is commonly used, an approach we also adopt in our project.

Furthermore, not all actions in Skyjo are allowed at all times. To prevent illegal actions, we follow the standard approach by using an action mask and modifying the model logits to ensure that invalid actions receive a probability of zero.

3.4.2 Deep Q-learning

Deep Q-learning was proposed by Mnih et al. [2013] where they trained a CNN to play seven Atari 2600 games. It is a model-free method, i.e. it does not estimate the environment and instead just learns from observations, as well as off-policy, which means it learns from a behavior policy.

One of the key improvements to standard Q-learning is the loss function $L_i(\theta_i)$ they define to train a neural network:

$$L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot)}\left[(y_i - Q(s,a;\theta_i))^2\right] \tag{3}$$

where

$$y_i = \mathbb{E}_{s'\sim\varepsilon}\left[r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) \mid s,a\right] \tag{4}$$

Instead of using $Q*(s,a)$, they use an approximation $Q*(s,a;\theta)$, which enabled them to use a neural network to use a bigger state space. The loss function can then be differentiated and then be used for stochastic gradient descent. A neural network can therefore now be trained.

Another improvement they made is to use an experience replay buffer $\mathcal{D}$. This buffer saves N transitions $e_t = (s_t, a_t, r_t, s_{t+1})$. It addresses the issue of the network potentially forgetting old information when only considering the most recent transition in its update. When performing gradient descent on the Deep Q-Network (DQN), a random minibatch will be sampled from this buffer, thereby replaying old experience.

We tried to train an agent using Deep Q-learning. However, during implementation, we encountered a critical issue, possibly a bug, that prevented us from training a DQN. When using `Torch` as a backend for neural nets and `DQNConfig` the policy `DQNTorchPolicy` is being created, which is an instance of `TorchPolicy`. This class is being passed an `action_distribution_fn`, which for some reason introduces noise into the logits of the last layer of the neural net. Since we mask the logits of actions that are not supposed be picked with `FLOAT_MIN`, the noise introduced afterwards will lead to underflow errors on some logits. When `EpsilonGreedy` wants to now pick an exploratory action, it will use the fact that disallowed actions are supposed to have a logit value of `FLOAT_MIN`. Due to the aforementioned perturbations introduced by `action_distribution_fn`, this will lead to a random action mask and therefore a high probability of a disallowed action being chosen. Unfortunately, we were not able to determine what function is being passed as `action_distribution_fn` to `TorchPolicy` exactly.

3.5 Training

Training for all agents is done in a multi-agent environment. As is usual for PPO, each training iteration proceeds by generating a number of episodes from the environment and then updating the policy and value networks. Our environment is set up in a way such that the training can easily be adapted to any number of players, allowing for flexibility in the training process. As described in Sections 3.2 and 3.3, various observation and reward settings can be chosen to adapt training.

In our initial training attempts, we trained multiple agents at the same time by allowing them to play against each other directly. However, such a training scheme proved highly unstable, as over time the agents learned to exploit the weaknesses in each other's strategies. This resulted in agents improving during some stretches of the training, and then becoming objectively worse again at other stages. Overall, only a very slow learning trend was observed. To combat this issue, we developed two training approaches. In the first approach, we implemented static predetermined policies to train against, whereas, in the second approach, we used self-play to train against previous versions of the same policy.

3.5.1 Predetermined Policies

The most simple policy, the random admissible policy, randomly chooses one of all allowed actions. A preprogrammed clever policy works by always taking cards with low value (< 4) and replacing revealed cards with higher value, or replacing unrevealed cards. If no low-value card is available, a new card is revealed. This policy ends up being competitive even with human players. Lastly, we managed to port a model trained in the online available single-player environment[6] to our environment and allow playing against that model directly.

[6] https://github.com/Guillaume-Barthe/Skyjo

3.5.2 Hyperparameter Tuning

With various options for observation and reward schemes, along with other training hyperparameters like model architecture, a large search space of possible model configurations is defined. A more detailed overview of all tunable parameters is given in Section 5.4. To systematically assess different combinations while keeping computation feasible, we devised a modular staged parameter grid search approach.

First, a set of default parameters is defined. Then, in each stage, possible combinations of parameter settings are explored. For example, one stage might evaluate all possible observation scheme configurations, testing both direct and indirect opponent observation alongside different card representation schemes such as simple numerical, one-hot, and efficient one-hot encoding.

A model is then trained for each possible parameter combination in that stage. Next, the combination with the best performance—measured by the lowest sum of final card values—is selected as the new default for those parameters, and the search proceeds to the next stage. Again, all possible combinations in that stage are evaluated. However, if no parameters are explicitly defined in that stage, the best-performing parameters from previous stages are used.

Although this approach introduces some bias due to the initial default parameter choices, it allows for efficient exploration of a large search space within reasonable time.

3.5.3 Self play

A major issue in multi-agent training is that if one policy is significantly weaker than the others, it contributes far less to the learning process. To address this, we adopted a self-play approach.

In this setup, policies are initialized as before, but only one is actively trained. Once it reaches a predetermined win rate against the others, a snapshot of the policy is saved and added to a bucket of past versions. Training then continues as usual, but now the agent competes not just against the original policies but also against its own previous versions. At the start of each game, opponent policies are randomly selected from the bucket. This approach also has the advantage that the trained policy gains experience by playing versus many different policies.

This process repeats, with new snapshots being saved whenever the trained policy meets the win rate threshold. If the bucket reaches a maximum size, or the training hits a predefined limit in terms of steps or sampled games, the process stops. Choosing the right win rate is crucial. If set too high, the training takes too long between snapshots, forcing the policy to play against much weaker opponents for extended periods. If set too low, policies may be saved simply by luck, leading to weaker strategies being retained.

In our environment, the win rate must be set relatively low due to the game's randomness. Even the strongest players occasionally lose to weaker opponents, meaning a high win rate requirement would result in long gaps between saved policies. In a three-player setting, for example, requiring an 80% win rate would make training impractically slow.

4 Evaluation

In this Section, we explain the methods and metrics used for model evaluation. Afterward, we provide the results of a thorough empirical evaluation of the different models.

4.1 Metrics

To evaluate the quality of a trained model, we measure three main metrics:

1. **Average Final Score:** The average final card sum of a player at the end of the game. This includes both revealed as well as unrevealed cards. This metric serves as an indicator of how well the model was able to reduce its overall card sum by exchanging and eliminating cards.

2. **Win Rate:** The percentage of games the model wins against a certain policy. This is ultimately the most important metric, as the sole objective of the game is to win. The win rate can be calculated

against any policy, but we usually evaluate it against (i) a random policy, (ii) the pre-programmed clever policy, or (iii) the model obtained from the previous work with a single-player environment.

3. **Undesirable Actions:** The number of actions that increase the player's score. This helps assess how "human-like" a policy plays, assuming humans avoid swapping low-value cards for high-value ones. Since the score is based on revealed cards, uncovering new cards has a 50% chance of increasing it. Hence, while no model will achieve a score of 0, this metric still serves as an indicator of human-likeness.

To measure all these metrics during the training process, we implemented a callback function that extracts the information from the environment and saves it into logging files. The plots that will be shown, depict the metric and the moving average. This was done so that the curves can be differentiated more easily.

4.2 Results

Here, we describe the results of our evaluation. First, we describe hyperparameter grid search and the impact of various parameters. Then, we examine the impact of training a specific model for a much larger number of iterations. Lastly, we examine the effect of using self-play during training. The full list of all experiments can be seen in Appendix A. For all of the experiments described below, we trained agents in a two-player environment. This allows us to directly compare two policies. However, the training approach can easily be extended to any number of agents, and those models trained with indirect observations can also play against any different number of players without requiring retraining.

4.2.1 Grid Search

In the first stage, many different models were trained to get a sense of which hyperparameters are important, to narrow down our grid search. Initially, we tried to train large models with 3-5 layers, each with multiple thousands of neurons. While the size would make sense when using one-hot encoding, since the input layer would then contain a few hundred neurons per player, it seems unnecessarily large for the efficient one-hot or simple encoding. It soon became clear that one-hot encoding contains too much information and training would take too long. To our surprise, very small models with only a few dozen neurons and one layer performed best. With this information, we started a more systematic grid search. The grid search parameters are shown in Table 1 for training against the pre-programmed policy and Table 2 for training against the policy from Guillaume Barthe. Since a different action reward reduction was better against each policy, this parameter differs, since the better value was used for later stages. All plots for these experiments can be found in Appendix B. Against the pre-programmed policy, our best model achieved an average win rate of 55.8%, whereas against Guillaume Barthe's single-player policy, our best model achieved an average win rate of 60%, implying both models surpassed their respective opponents. Both models are also competitive against the respective other policies, achieving win rates of 50% or more.

Grid search, Training against Pre-Programmed Policy
Figure 2 shows the effect of the action reward reduction and decay on the mean reward. Using a larger value for the reduction, the agent will receive more reward for its actions. The decay dictates how long it takes for the reduction to become 0. However, we need to be careful to not give too much reward for the actions for too long. In Figure 3, we see that the setup that gives the most reward for the longest time has the worst win rate. While it does catch up eventually, it seems to be on the edge of producing a usable model. Figure 4 gives us an insight into how different model sizes affect training speed. The smallest models take the longest to achieve a similar win rate. In this case, the simple encoding seems to be preferable, probably due to the limited size of the neural net. Otherwise, it would seem that both encodings work similarly well.

Grid search, Training against Single-Agent Policy
A high decay seems to be a bigger problem when training against the single-player policy. Figure 5 shows an increase in the final score, which means the agent is losing more frequently when the decay is 0.995. This is also reflected in Figure 7, where both models make an increased number of undesirable actions. The training

Obs. Mode	AR Reduction	AR Decay	NN Size	WR	FS	UA
Simple	1	0.95	[32]	0.558	20.059	17.240
Simple	1	0.98	[32, 32]	0.558	21.181	17.274
Simple	1	0.98	[32]	0.551	20.403	17.473
Simple	1	0.98	[32]	0.551	20.403	17.473
EOH	1	0.98	[64]	0.541	20.687	17.274
EOH	1	0.98	[32]	0.540	20.510	17.174
Simple	5	0.95	[32]	0.530	20.574	17.171
EOH	1	0.98	[32, 32]	0.526	23.172	17.494
Simple	1	0.995	[32]	0.526	21.304	17.159
EOH	1	0.98	[64, 64]	0.517	23.421	17.897
Simple	1	0.98	[64]	0.516	21.558	16.927
Simple	5	0.98	[32]	0.515	21.798	17.266
Simple	1	0.98	[64, 64]	0.508	24.642	17.851
Simple	1	0.98	[16]	0.498	21.331	17.590
Simple	5	0.995	[32]	0.485	21.611	17.532
EOH	1	0.98	[16]	0.483	21.869	17.156

Table 1: Direct observations, a shared NN for the value function, 0 curiosity reward, and an entropy coefficient of 0.01 were used for this series of experiments. The policy played against the pre-programmed policy. WR stands for win rate, FS for final score and UA for undesirable actions.

seems to be more unstable in general, which can be observed in Figure 6. For some reason, efficient one hot encoding and 32 neurons struggled in the beginning, before recovering and getting better.

4.2.2 Lengthy Training

Additionally, we trained one model for a large number of training iterations. For this, we decided to train the model on the "efficient_one_hot" observation space with indirect observations (cards of opponents are not directly observed), since grid search showed only slightly slower training for this observation space than for the "simple" observation space, while this space potentially offers more information for the model. We furthermore trained a rather large model with a neural network with two hidden layers containing 64 neurons each and shared value and policy functions, to not overly limit the expressivity.

The entropy coefficient was set to 0.03. The model was trained against the pre-programmed policy and was run in several stages, differing by their reward scheme, without using any self-play. We started training this model simultaneously with the grid search. Therefore at the first stage, we started to pre-train the model and after receiving results from the grid search we continued the training with stage two. The first training stage ran for 9769 iterations and the model was trained with the following parameter values:

- curiosity_reward = 5
- action_reward_reduction = 1
- action_reward_decay = 1

The second stage was only run for 1006 training iterations due to its performance being poor. The parameters were changed to the following:

- curiosity_reward = 0
- action_reward_reduction = 0.6
- action_reward_decay = 0.995

Obs. Mode	AR Reduction	AR Decay	NN Size	WR	FS	UA
Simple	5	0.98	[64, 64]	0.6	23.198	17.161
Simple	5	0.98	[32, 32]	0.588	25.221	17.592
Simple	1	0.95	[32]	0.565	24.559	17.287
Simple	5	0.95	[32]	0.564	23.571	17.1
Simple	5	0.98	[64]	0.564	22.998	17.953
EOH	5	0.98	[64]	0.56	23.088	17.293
Simple	5	0.98	[32]	0.557	22.454	17.608
Simple	1	0.98	[32]	0.557	22.951	17.9
Simple	5	0.98	[32]	0.557	22.454	17.608
EOH	5	0.98	[32, 32]	0.55	25.01	17.738
EOH	5	0.98	[16]	0.525	24.302	18.129
Simple	5	0.98	[16]	0.523	24.732	17.629
EOH	5	0.98	[32]	0.449	27.997	18.11
Simple	5	0.995	[32]	0.322	29.256	19.549
EOH	5	0.98	[64, 64]	0.255	34.642	19.309
Simple	1	0.995	[32]	0.231	32.091	19.709

Table 2: Direct observations, a shared NN for the value function, 0 curiosity reward and an entropy coefficient of 0.01 were used for this series of experiments. The policy played against the single-player policy of Guillaume Barthe. WR stands for winrate, FS for final score and UA for undesirable actions.

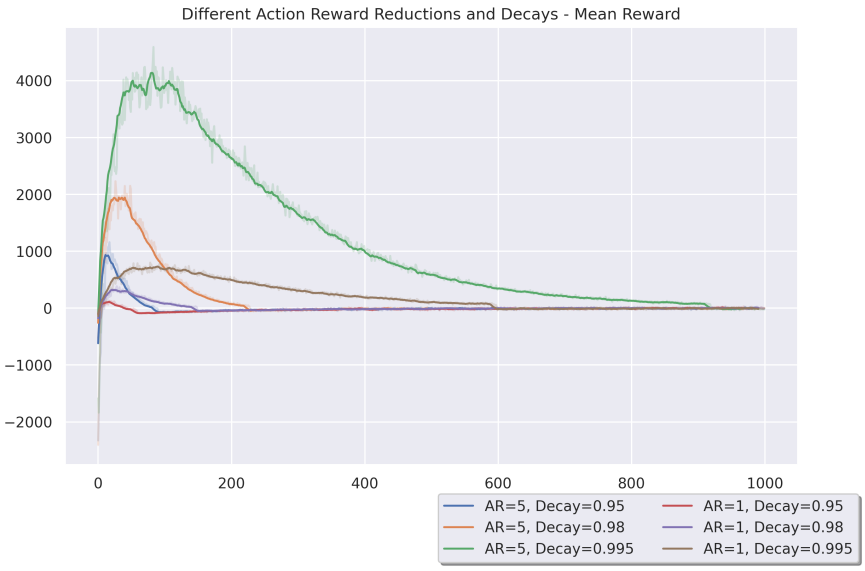

Figure 2: The mean reward over time for different action reward reduction (importance) and decay values is compared in this plot. Simple, direct observations, a shared NN for the value function, 0 curiosity reward, an entropy coefficient of 0.01, and a neural network of size [32] were used. Training is done against the pre-programmed policy.

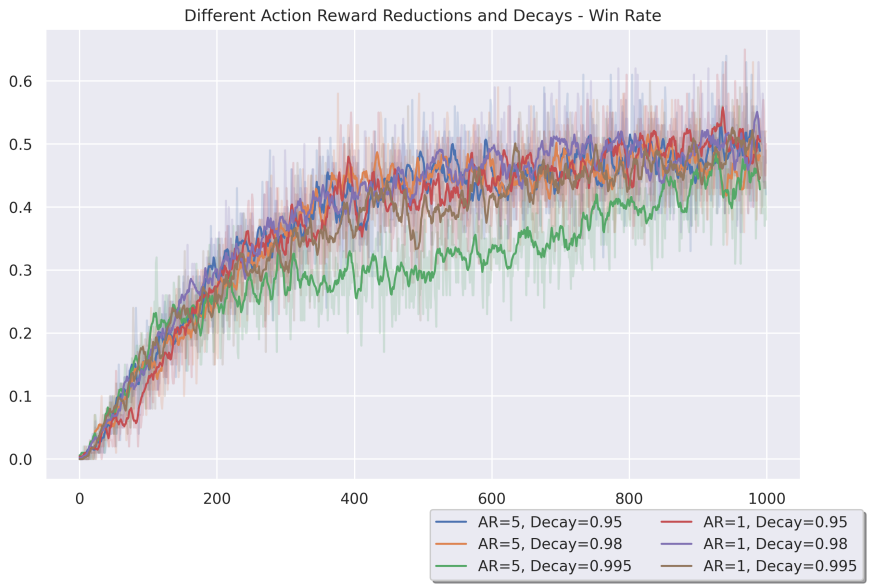

Figure 3: The average win rate over time for different action reward reduction (importance) and decay values is compared in this plot. Simple, direct observations, a shared NN for the value function, 0 curiosity reward, an entropy coefficient of 0.01, and a neural network of size [32] were used. Training is done against the pre-programmed policy.

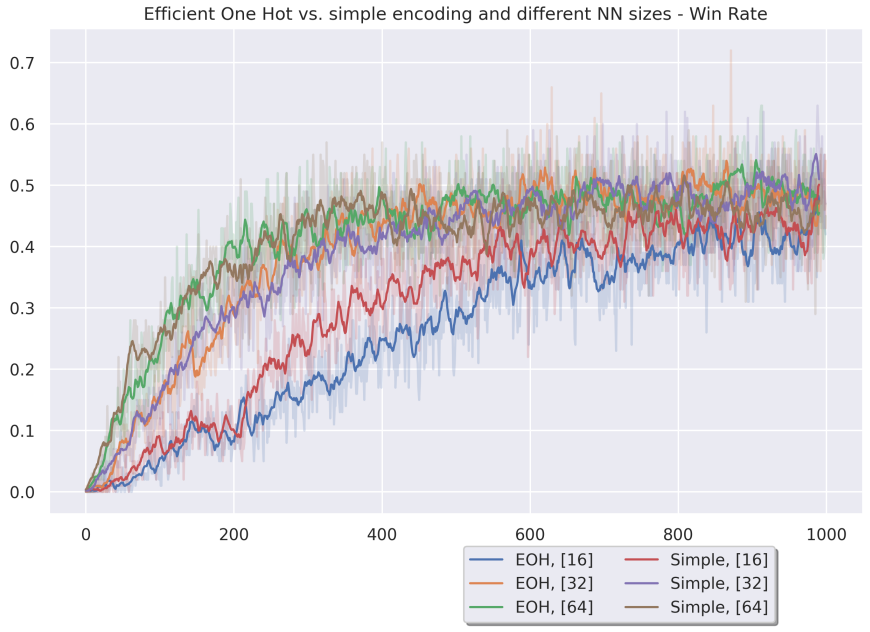

Figure 4: The average win rate over time for different observation encodings and neural net sizes is compared in this plot. Direct observations, a shared NN for the value function, 0 curiosity reward, an action reward reduction of 1, a decay of 0.98, and an entropy coefficient of 0.01 were used. Training is done against the pre-programmed policy.

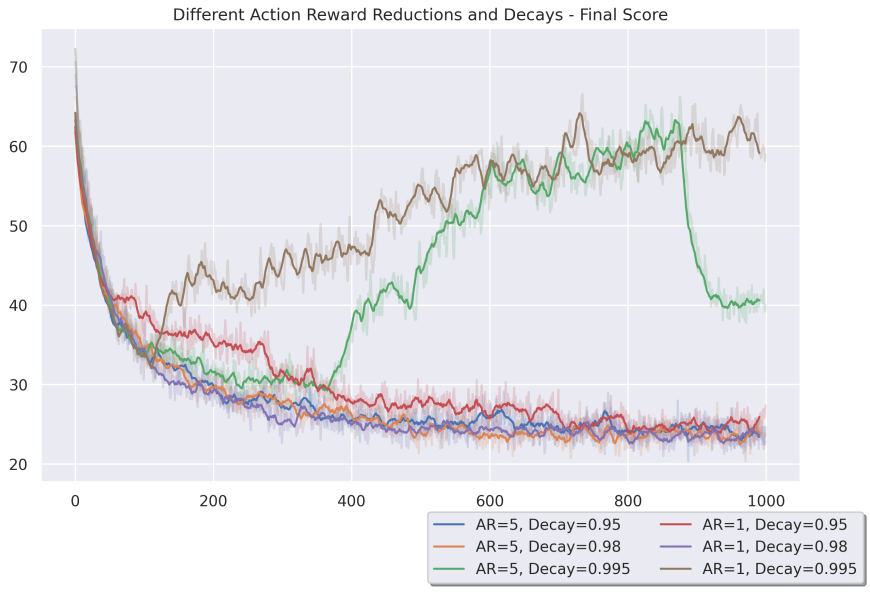

Figure 5: The average final score over time for different action reward reduction (importance) and decay values is compared in this plot. Simple, direct observations, a shared NN for the value function, 0 curiosity reward, an entropy coefficient of 0.01, and a neural network of size [32] were used. Training is done against Guillaume Barthe's single-agent policy.

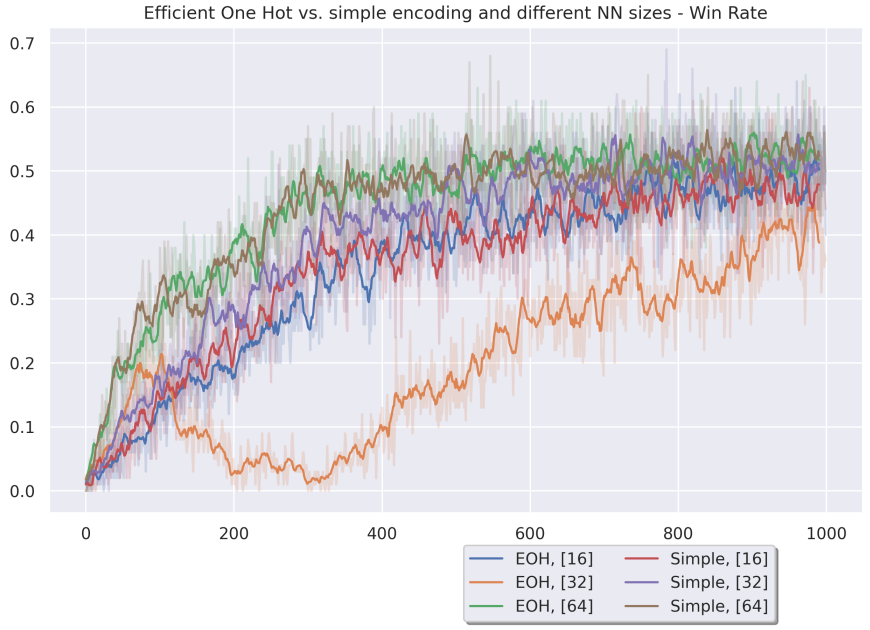

Figure 6: The average win rate over time for different observation encodings and neural net sizes is compared in this plot. Direct observations, a shared NN for the value function, 0 curiosity reward, an action reward reduction of 5, a decay of 0.98, and an entropy coefficient of 0.01 were used. Training is done against Guillaume Barthe's single-agent policy.

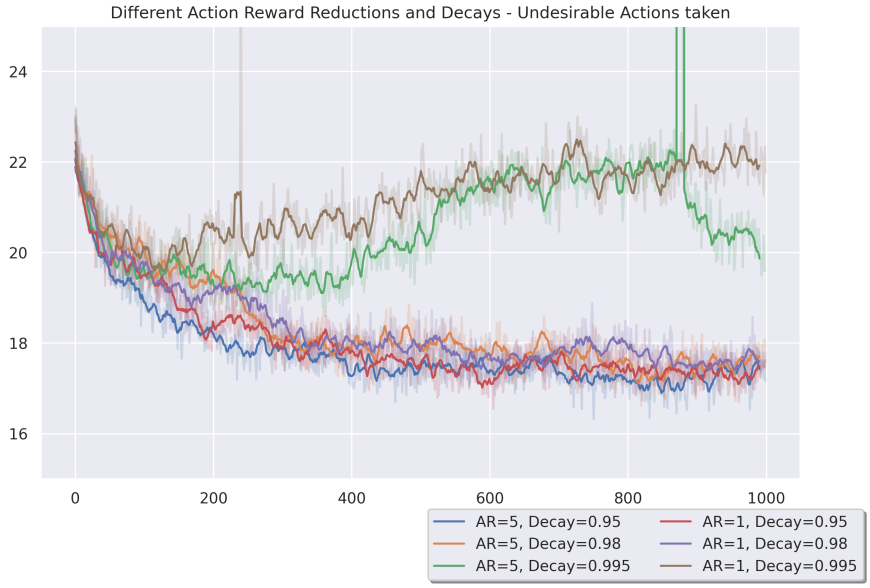

Figure 7: The average number of undesirable actions over time for different action reward reduction (importance) and decay values is compared in this plot. Simple, direct observations, a shared NN for the value function, 0 curiosity reward, an entropy coefficient of 0.01, and a neural network of size [32] were used. Training is done against Guillaume Barthe's single-agent policy.

Using the `action_reward_decay` did not seem to work well (see Figure 8, the huge dip happened due to training using action_reward_decay = 0.995) and the model improved when `action_reward_reduction` reached zero. For this, in the last training stage, these parameters were all set to zero and the model was only trained on the "end-of-game"-reward. The training with this reward scheme continued for the largest amount of iterations. Instantly switching the reward scheme appears ineffective, whereas gradually modifying it in small increments (using `action_reward_decay` as in grid search) leads to better performance.

The resulting win rate, mean of the final game score, and mean of undesirable actions for every iteration can be seen in Figures 8 to 10. When pausing the training and reloading the model, the model's weights need one iteration to be successfully reloaded. This is probably due to the `rllib`-library splitting the training into different modules, with some modules gathering samples and one being used for training. The sampling modules seemingly get the updated weights of the model after the first run. For this, all training pauses lead to a spike of a very low win rate and very high final score (visible as a dip/peek in the blue line in the Figures 8 to 10), as well as a high number of undesirable actions. This model was trained up to a win rate of around 54% against the pre-programmed policy and can be found in our repository in the folder `trained_models` folder.

In general, we decided not to further train the models obtained in grid search due to our results from training this model for nearly $40,000$ iterations. The model showed only marginal improvement over the last $30,000$ steps and failed to outperform our best models from the grid search.

4.2.3 Self-Play

We noticed early on in the project that training new agents from the ground up solely using a self-play scheme resulted in either very slow or very unstable learning. More specifically, not using any every-step action rewards would lead to slow learning as the agents receive only very sparse feedback. Conversely, using action rewards could lead to very unstable learning as agents could learn to collaborate to maximize such rewards by never finishing the game and obtaining virtually infinite every-step action rewards.

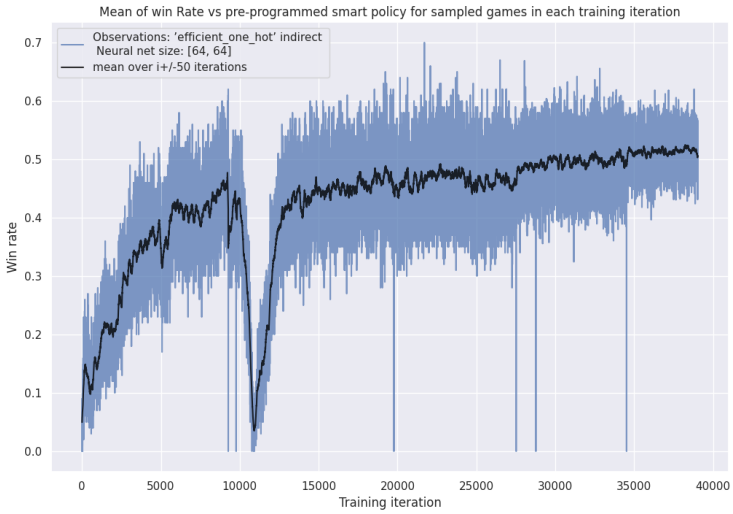

Figure 8: This plot shows the relative won games by the trained model for each training iteration. Since Skyjo is a rather random game, the win rate shows great variance. Therefore the black line shows the average win rate of the last and next 50 iterations at the corresponding iteration.

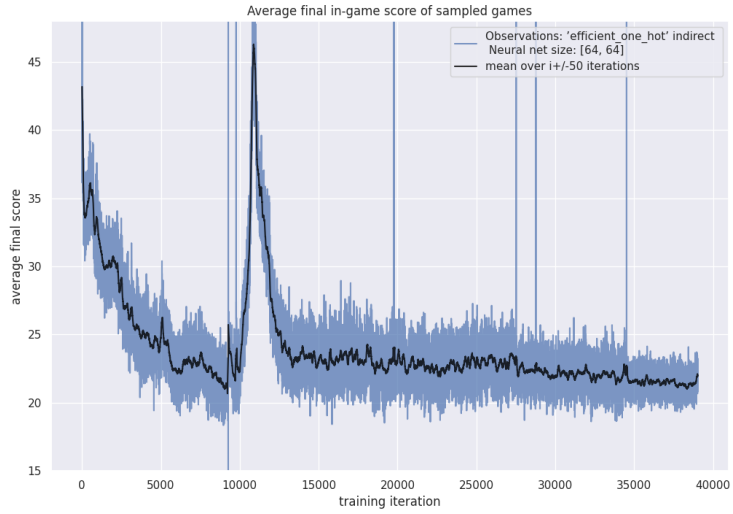

Figure 9: This plot shows the average final game score by the trained model for each training iteration. Since Skyjo is a rather random game, the final score - as the win rate - shows great variance. Therefore the black line shows the average final game score of the last and next 50 iterations at the corresponding iteration.

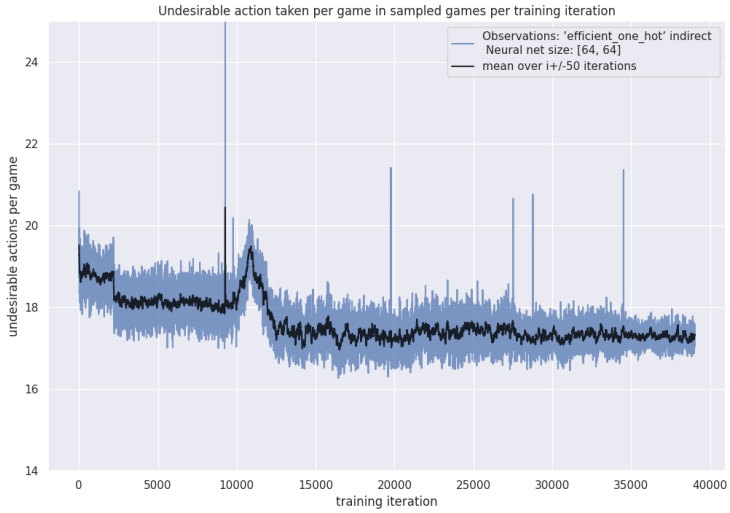

Figure 10: This plot shows the average number of undesirable actions (actions where after the action the score of the agent is higher than before) per game by the trained model for each training iteration. Since Skyjo is a rather random game, the number of undesirable actions - as the two metrics before - shows great variance. Therefore the black line shows the average number of undesirable actions per game of the last and next 50 iterations at the corresponding iteration.

As a result, we decided to use self-play not to train new agents from zero but rather to initialize the training with policies trained against fixed policies, i.e., the models resulting from the previous step detailed in Section 4.2.1. In Figures 11 to 13, we show the results of training the [64, 64] network, simple observation model (the top performing configuration in Table 2) for 10000 iterations using self-play. Note that all metrics are worse in the very first iteration, as the library `rllib` only fully loads the model in the second iteration.

These figures show that none of the three relevant metrics improved noticeably. In a self-play learning scheme, the win rate is always supposed to stay consistent, as for a higher win rate stronger opponents are introduced. However, further evaluation also showed that the win rate against other policies, like the Pre-Programmed policy and Guillaume Barthe's single-agent policy, stayed consistent throughout the self-play training process and did not improve.

While the slow training of new agents from scratch using self-play could likely be solved with sufficient computing power, it seems unlikely that self-play can be used to significantly improve strong pre-policies further. We hypothesize that in this highly random game, even an optimal strategy would lose a significant portion of games against a simple yet clever approach such as the pre-programmed policy. Hence, the inherent stochasticity in Skyjo could prohibit any sizeable improvements, and consistent win rates in the realm of 70 or 80 percent might be unobtainable.

5 The repository

This section gives an overview of the GitHub repository `https://github.com/faguodev/SkyjoAI` of our implementation. Furthermore, we will explain how to play against trained agents (Section 5.1), how to train your own agents (Section 5.2), and how to evaluate the different agents (Section 5.3).

To use the repository to its fullest, install all dependencies listed in "environment.yml". The repository is split into different folders. The folders contain the following:

- **./custom_models** the custom action mask model for PPO training, the `fixed_policies.py` file,

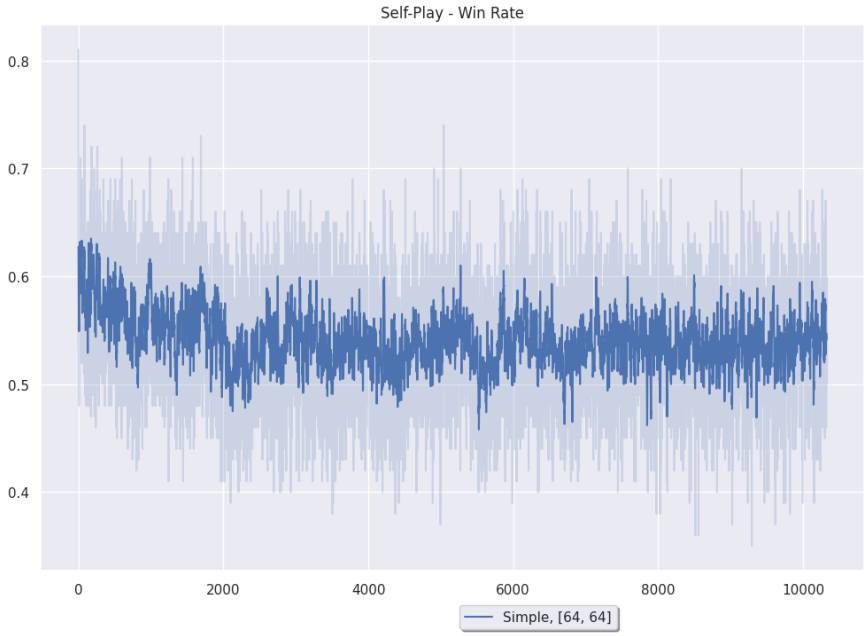

Figure 11: The average win rate of the main, trained policy in the self-play learning scheme.

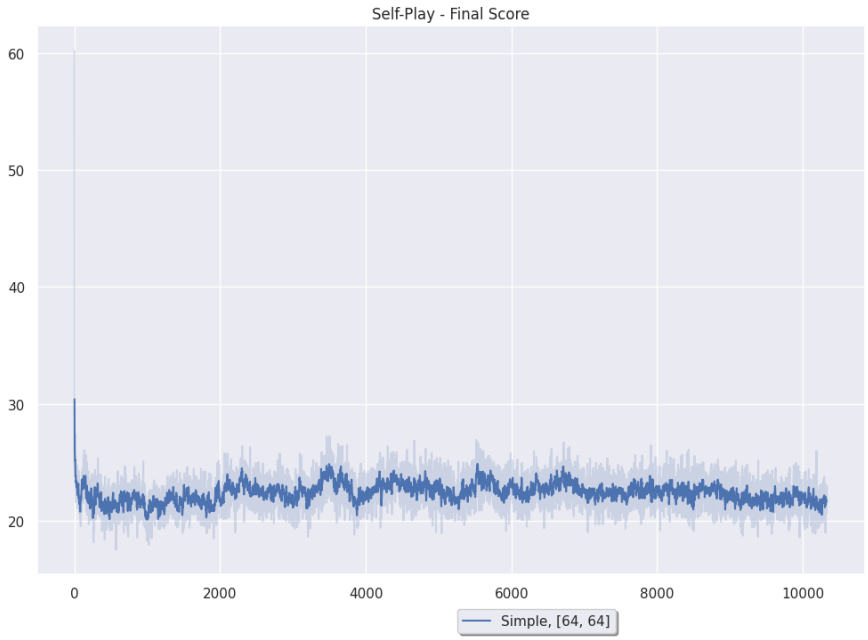

Figure 12: The average final score of the main, trained policy in the self-play learning scheme.

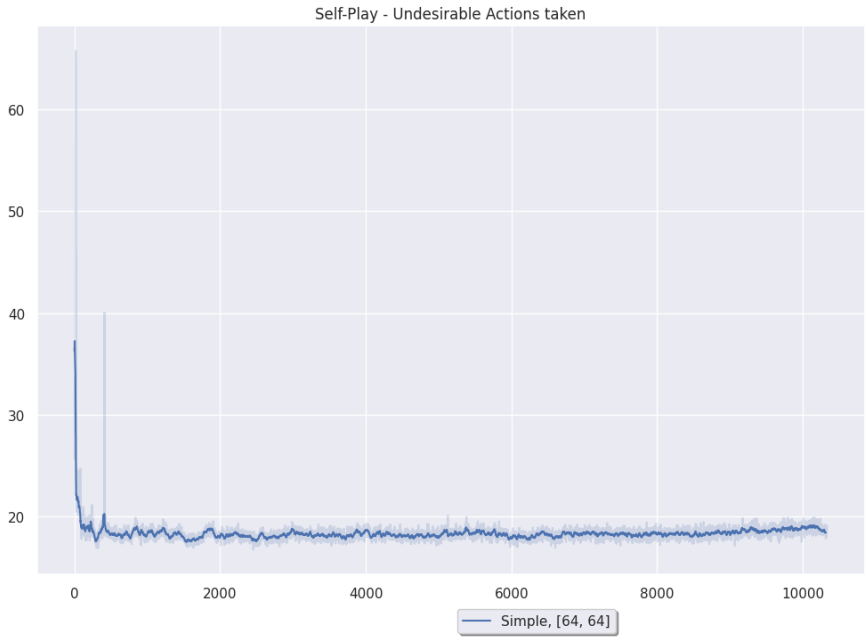

Figure 13: The average number of undesirable actions of the main, trained policy in the self-play learning scheme.

which contains the predetermined policies described in Section 3.5.1, and the model trained in Guillaume Barthe's single-player environment. This model can be used to play against it, for training agents, and for evaluating them.

- **./environment** contains the files for the environment the agents are trained in (therefore the implementation of the game "Skyjo".
- **./logs** is the folder where training log files are saved (if training is done using `rllib_self_play.py` or `rllib_grid_search.py`).
- **./trained_models** is the folder where trained models are saved (for training with `rllib_self_play.py` or `rllib_grid_search.py`). Additionally, some of our trained models are provided in this folder.

Further important files are in the main directory, namely:

- **callback_functions.py** contains a callback function for logging and training.
- **eval.py**, **evaluation.ipynb** and **eval_compare_with_others.py** can be used to evaluate (trained) models. More info can be found in Section 5.3.
- **rllib_self_play.py** and **rllib_grid_search.py** can be used to train models. For more information consult Section 5.2.
- **skyjo_gui.py** and **skyjo_gui_compare_with_others.py** can be run to play a game of Skyjo against one or more trained models (more in Section 5.1).

5.1 How to play

To play against trained agents in a graphical user interface, run `skyjo_gui.py`. This will launch a game against one or multiple trained models. To select different models to play against change following variables:

1. **model_path**: change to the name of the folder containing the model
2. **checkpoint**: change to the checkpoint from which the trained model should be loaded

If no corresponding `experiment_config.json` file exists to load the model's configuration, manually set the following parameters to fit the model:

1. **observation_mode**: string. Describes the observation space. The names of possible schemes are described in Section 5.4 and their workings in Section 3.2.
2. **observe_other_player_indirect**: boolean value. Defines observation mode (for more info again consult Section 3.2)
3. **vf_share_layers**: boolean value. If true, a shared network is used for the value and policy functions.
4. **neural_network_size**: array of integers, like so: `[32, 32]`. The entries define a neural network with layer consisting of as many neurons as the entries value.

5.2 How to train

To train a model, `rllib_self_play.py` and `rllib_grid_search.py` can be used. In `rllib_grid_search.py`, a grid search is implemented to find optimal training parameters. Simply run the Python file to start the search. We explored various observation schemes, model-specific parameters, and an importance and decay parameter for the "every-step" reward scheme (Section 3.3). To reduce computational effort, the grid search is divided into several stages, with each stage usually focusing only on a subset of the parameters. After a specified number of training iterations, the models are compared, and the parameter values of the best-performing model are saved. The search then continues to the next stage using the best parameter values from all previous stages. The searched parameters, parameter values, and default configuration can be adjusted at the beginning of the file. The `tuning_stages` list defines the search stages and can be modified accordingly. This list contains dictionaries specifying the parameters and search values for each stage.

To train a model with a fixed parameter setting use `rllib_self_play.py`. This file can be used to train the model with the "self-play" (Section 3.5.3) routine or with a "standard" training (for this set the `win_rate` in the configuration for the Callback function to 1 i.e. 100%). To continue training an already pre-trained model it can be loaded with its configuration from either a successful grid search or a previous self-play training. The training configuration can also be manually adapted to suit your needs. The most important training parameter can be set at the start of the file. Otherwise, the environment is configured at `skyjo_config`, the model with the `model_config` dictionary, and the `rllib` algorithm and training can be configured in the `config` dictionary.

5.3 How to evaluate

For evaluation, we have three different files, with different kinds of evaluation. First, the `evaluation.ipynb` notebook can be used to compare and plot the training logs. Secondly, the `eval.py` file can be used to let policies play against each other and track their win rate. Thirdly the `skyjo_gui.py` file can be run to personally play against policies and qualitatively assess their performance, as described in Section 5.1. Similarly, the files `eval_compare_with_others.py` and `skyjo_gui_compare_with_others.py` can be used to let policies play against agents trained in the single-agent environment from Guillaume Barthe[7], and to personally play against such agents.

[7] https://github.com/Guillaume-Barthe/Skyjo

5.4 The environment

The environment is set up in two classes. The `SkyjoGame` class (in the file `skyjo_game.py`) defines the game of Skyjo. The `SimpleSkyjoEnv` class (in the file `skyjo_env.py`) creates an, for the agents playable, environment using the `SkyjoGame` class following the AEC API from PettingZoo.
Skyjo Environment: The Skyjo environment, defined in the class `SimpleSkyjoEnv`, creates the environment for agents to learn to play the game Skyjo. The environment can be configured with different reward (Section 3.3) and observation (Section 3.2) schemes. The `step()` and `observe()` functions are used by agents to interact with the environment. The following parameters can be set for configuration:

- **"num_players":** [int] The number of players, which can be set to any integer between 2 and 8 (technically, up to 12 players can play, though this is not recommended).
- **"observe_other_player_indirect":** [boolean] Determines whether agents observe opponents' player cards ("False") or only game statistics ("True").
- **"render_mode":** ["human" or None] A legacy variable previously used for rendering the game; not relevant for training.
- **"observation_mode":** Defines the observation scheme. Options include: "simple", "onehot", "efficient_one_hot", "efficient_one_hot_port_to_other", or "simple_port_to_other".
- **"reward_config":** Configures the reward scheme. An object of the "RewardConfig" class—a dictionary with the entries listed below—must be provided. The dictionary must contain the following entries:
  - **"reward_refunded":** [float] Reward given to the agent for discarding a complete column of cards (i.e., when all three cards in a column have the same value).
  - **"final_reward":** [float] Reward given to the winner at the end of the game, with the negative equivalent given to all losers.
  - **"score_per_unknown":** [float] Used in the "every-step" reward scheme. Defines the value assigned to unknown cards when calculating the score before and after an action.
  - **"action_reward_reduction":** [float] Multiplies the "each-turn reward" to weight it.
  - **"curiosity_reward":** [float] Reward given to the agent for revealing an unknown card of their own.
  - **"old_reward":** [bool] Set to "True" for a simple "end-of-game" reward or "False" for an "each-turn" reward.

An exemplary configuration setup is given below:

Listing 1: Example Configuration

```
skyjo_config = {
    "num_players": 2,
    "reward_config": {
        "reward_refunded": 10,
        "final_reward": 100,
        "score_per_unknown": 5.0,
        "action_reward_reduction": 1,
        "old_reward": False,
        "curiosity_reward": 5,
    },
    "observe_other_player_indirect": True,
```

```
    "render_mode": "human",
    "observation_mode": "simple",
}
```

6 Conclusion

In this project, we began by exploring existing implementations for inspiration. After conducting research, we chose to build upon Michael Feil's Skyjo environment[8]. We ported the environment to the, at the time, latest version of PettingZoo, fixed various gameplay bugs, and implemented different state encodings and reward structures. To train agents, we used `rllib`, experimenting with PPO and Deep Q-learning. However, due to an issue in the setup or a potential bug, we were unable to successfully train an agent using DQN. Additionally, we implemented self-play, though it did not lead to improved policies beyond those obtained through grid search.

Our approach consistently achieved a win rate exceeding 50% against the pre-programmed policy and Guillaume Barthe's single-agent policy, which represented the prior state-of-the-art. However, significantly surpassing this threshold remains a challenge. Given the stochastic nature of Skyjo, it is uncertain whether substantial further improvements are feasible. Future work could involve more extensive grid searches, particularly for PPO-specific parameters, to enhance training efficiency or achieve marginal performance gains. Additionally, one-hot encoding was only briefly explored due to its computational expense, but with extended training time, optimized pipelines, or better hardware, it might yield superior results.

Ultimately, we developed a competitive reinforcement learning agent capable of playing Skyjo at a strong level. Winning against the agent is a challenge, as users can experience firsthand by following the instructions in our repository and playing against the trained model.

References

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *ArXiv*, abs/1312.5602, 2013.

John Schulman, Sergey Levine, P. Abbeel, Michael I. Jordan, and Philipp Moritz. Trust region policy optimization. *ArXiv*, abs/1502.05477, 2015.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *ArXiv*, abs/1707.06347, 2017.

A Tables

[8] https://github.com/michaelfeil/skyjo_rl

Obs. Mode	Indirect	VF Share	CR	AR Reduction	AR Decay	Entropy Coeff.	NN Size	WR	FS	UA
Simple	False	True	0	5	0.98	0.01	[64, 64]	0.6	23.2	17.2
Simple	False	True	0	5	0.98	0.01	[32, 32]	0.588	25.2	17.6
Simple	False	True	0	1	0.95	0.01	[32]	0.565	24.6	17.3
Simple	False	True	0	5	0.95	0.01	[32]	0.564	23.6	17.1
Simple	False	True	0	5	0.98	0.01	[64]	0.564	23	18
EOH	False	True	0	5	0.98	0.01	[64]	0.56	23.1	17.3
Simple	False	True	0	5	0.98	0.01	[32]	0.557	22.5	17.6
Simple	False	True	0	1	0.98	0.01	[32]	0.557	23	17.9
EOH	False	True	0	5	0.98	0.01	[32, 32]	0.55	25	17.7
EOH	False	True	0	5	0.98	0.01	[16]	0.525	24.3	18.1
Simple	False	True	0	5	0.98	0.01	[16]	0.523	24.7	17.6
EOH	False	True	0	5	0.98	0.01	[32]	0.449	28	18.1
EOH	False	True	3	1	0.995	0.01	[64, 64]	0.42	31.6	19.3
Simple	False	True	0	1	0.98	0.01	[32]	0.411	27.4	18.4
Simple	False	True	0	1	1	0.01	[64]	0.381	27.9	18.7
Simple	False	True	0	1	1	0.01	[128]	0.38	29.9	18.8
EOH	False	True	0	1	1	0.01	[32]	0.363	27.6	20.1
EOH	False	True	2	1	0.995	0.01	[64, 64]	0.36	29.3	18.9
Simple	False	True	0	1	1	0.01	[32]	0.341	29	19.1
Simple	False	True	0	1	1	0.01	[32]	0.334	27.2	19.3
EOH	False	True	0	1	1	0.01	[64]	0.331	28.6	19.2
Simple	False	True	0	5	0.995	0.01	[32]	0.322	29.3	19.5
EOH	False	True	2	1	0.995	0.01	[16]	0.271	31.1	33.3
EOH	False	True	2	1	0.995	0.01	[64, 64]	0.263	33.5	20.4
EOH	False	True	2	1	0.995	0.01	[128, 128]	0.26	35.8	19.6
EOH	False	True	2	1	0.995	0.01	[32, 32]	0.257	32.6	20.3
EOH	False	True	0	5	0.98	0.01	[64, 64]	0.255	34.6	19.3
EOH	False	True	2	1	0.995	0.01	[32, 32]	0.25	35.7	31.9
Simple	False	True	0	1	0.995	0.01	[32]	0.231	32.1	19.7
EOH	False	True	2	1	0.995	0.01	[128, 128]	0.224	36.2	20.3
Simple	False	True	3	1	0.98	0.01	[32]	0.093	46.8	20.7
Simple	False	True	5	1	1	0.01	[32]	0.065	47.8	21.3
Simple	False	True	3	1	0.98	0.01	[16]	0.028	58.6	21
EOH	False	True	3	1	0.98	0.01	[16]	0.0161	65.9	22

Table 3: All trainings where our policy was trained against Guillaume Barthe's policy. The table is sorted by win rate. 'EOH' stands for efficient one hot encoding, 'Obs. Mode' for observation mode, 'Indirect' indicates if the agent only saw metadata of the game or each player's cards, 'VF Share' describes if the value function was integrated in the same neural net as the action value function, 'CR' is the curiosity reward, 'AR' stands for action reward, 'WR' is the win rate, 'FS' is the final score and 'UA' is undesirable actions. The last three metrics are averages of the last 10 epochs.

Obs. Mode	Indirect	VF Share	CR	AR Reduction	AR Decay	Entropy Coeff.	NN Size	WR	FS	UA
Simple	False	True	0	1	0.95	0.01	[32]	0.558	20.1	17.2
Simple	False	True	0	1	0.98	0.01	[32, 32]	0.558	21.2	17.3
Simple	False	True	0	1	0.98	0.01	[32]	0.551	20.4	17.5
EOH	False	True	0	1	0.98	0.01	[64]	0.541	20.7	17.3
EOH	False	True	0	1	0.98	0.01	[32]	0.54	20.5	17.2
Simple	False	True	0	5	0.95	0.01	[32]	0.53	20.6	17.2
EOH	False	True	0	1	0.98	0.01	[32, 32]	0.526	23.2	17.5
Simple	False	True	0	1	0.995	0.01	[32]	0.526	21.3	17.2
EOH	False	True	0	1	0.98	0.01	[64, 64]	0.517	23.4	17.9
Simple	False	True	0	1	0.98	0.01	[64]	0.516	21.6	16.9
Simple	False	True	0	5	0.98	0.01	[32]	0.515	21.8	17.3
Simple	False	True	0	1	0.98	0.01	[64, 64]	0.508	24.6	17.9
Simple	False	True	0	1	0.98	0.01	[16]	0.498	21.3	17.6
Simple	True	True	0	1	1	0.03	[16]	0.493	25	16.1
Simple	False	True	0	5	0.995	0.01	[32]	0.485	21.6	17.5
EOH	False	True	0	1	0.98	0.01	[16]	0.483	21.9	17.2
Simple	True	True	0	1	1	0.03	[32]	0.482	25.3	16.3
Simple	True	True	0	1	1	0.03	[64]	0.473	25.2	16
Simple	True	True	0	1	1	0.03	[128]	0.389	27.2	16.1
Simple	True	True	0	1	1	0.03	[32, 16]	0.377	30	16.3
Simple	True	True	0	1	1	0.03	[128, 64]	0.373	28.3	16.2
Simple	True	True	0	1	1	0.03	[64, 32]	0.366	28.5	16.2
Simple	True	True	0	1	1	0.03	[128, 32]	0.365	29.1	16.4
Simple	True	True	0	1	1	0.03	[16, 16]	0.342	29.3	16.3
Simple	True	True	0	1	1	0.03	[8]	0.334	30	16.4
Simple	True	True	0	1	1	0.03	[32, 32]	0.327	30.2	16.3
Simple	True	True	0	1	1	0.03	[16, 8]	0.311	31.1	16.3
Simple	True	True	0	1	1	0.03	[256]	0.284	32.7	16.6
Simple	True	True	5	1	1	0.03	[32]	0.263	33.8	17
Simple	True	True	5	1	1	0.03	[128]	0.256	33.3	17
Simple	True	True	0	1	1	0.03	[128, 64, 128]	0.253	33.7	16.6

Table 4: First table of all trainings where our policy was trained against the pre-programmed policy. The table is sorted by win rate. 'EOH' stands for efficient one hot encoding, 'OH' for one hot encoding, 'Obs. Mode' is the observation mode, 'Indirect' indicates if the agent only saw metadata of the game or each player's cards, 'VF Share' describes if the value function was integrated in the same neural net as the action value function, 'CR' is the curiosity reward, 'AR' stands for action reward, 'WR' is the win rate, 'FS' is the final score and 'UA' is undesirable actions. The last three metrics are averages of the last 10 epochs.

Obs. Mode	Indirect	VF Share	CR	AR Reduction	AR Decay	Entropy Coeff.	NN Size	WR	FS	UA
Simple	True	True	0	1	1	0.03	[128, 64, 32]	0.24	33.1	16.4
Simple	True	True	0	1	1	0.03	[8, 8]	0.233	34.1	16.4
Simple	True	True	5	1	0.995	0.005	[128, 64]	0.232	35.6	17.1
Simple	True	True	0	1	0.5	0.03	[16]	0.231	29.3	18.3
Simple	True	True	0	1	1	0.03	[256, 256]	0.23	33.8	16.7
Simple	True	True	0	1	1	0.03	[64, 128, 256]	0.23	33.9	16.6
Simple	True	True	5	1	1	0.03	[16]	0.224	34.6	17
Simple	False	True	0	1	1	0.03	0	0.214	35.6	17
Simple	False	True	0	1	1	0.03	[0]	0.208	37.3	16.9
Simple	False	True	0	1	1	0.03	[256, 256]	0.208	36.3	16.7
Simple	True	True	5	1	1	0.03	[64]	0.208	35.7	17.1
Simple	False	True	0	1	1	0.03	[256, 256]	0.202	37.1	16.9
Simple	True	True	5	1	0.995	0.01	[128, 64]	0.171	38	17.2
Simple	True	True	5	1	0.995	0.01	[128]	0.171	37.3	17.2
Simple	True	True	5	1	0.995	0.005	[128]	0.168	36.6	17.3
Simple	True	True	5	1	1	0.03	[256]	0.155	38.8	17.1
Simple	True	True	5	1	1	0.01	[2048, 2048, 1024, 512]	0.153	40.9	17.3
Simple	True	True	5	5	0.98	0.01	[2048, 2048, 1024, 512]	0.152	40.4	17.1
Simple	True	True	5	1	0.995	0.03	[128]	0.151	37.8	17.3
Simple	False	True	0	1	1	0.03	[2048, 2048, 1024, 512]	0.149	43.4	17.4
Simple	True	True	5	1	0.995	0.03	[256]	0.141	39.7	17.4
Simple	True	True	5	1	0.995	0.005	[256]	0.14	40.2	17.4
Simple	True	True	5	1	0.995	0.03	[128, 64]	0.138	40.3	17.4
Simple	False	True	5	1	0.995	0.01	[8192, 2048, 512]	0.137	42.3	17.3
Simple	True	True	0	5	0.98	0.01	[2048, 2048, 1024, 512]	0.136	45	17.2
Simple	True	True	5	1	0.995	0.01	[2048, 2048, 1024, 512]	0.131	43.4	17.5
Simple	True	True	5	1	0.995	0.01	[2048, 4096, 2048, 512]	0.121	42.2	17.4
Simple	True	True	5	5	1	0.01	[1024, 1024, 512, 256]	0.119	41.9	17.4
Simple	True	True	5	1	1	0.03	[8]	0.119	42.4	17.5
Simple	True	True	5	1	0.995	0.01	[2048, 2048, 1024, 512]	0.117	81.6	17.4
Simple	True	True	5	5	0.98	0.01	[2048, 2048, 1024, 512]	0.117	42.2	17.3

Table 5: Second table of all trainings where our policy was trained against the pre-programmed policy. The table is sorted by win rate. 'EOH' stands for efficient one hot encoding, 'OH' for one hot encoding, 'Obs. Mode' is the observation mode, 'Indirect' indicates if the agent only saw metadata of the game or each player's cards, 'VF Share' describes if the value function was integrated in the same neural net as the action value function, 'CR' is the curiosity reward, 'AR' stands for action reward, 'WR' is the win rate, 'FS' is the final score and 'UA' is undesirable actions. The last three metrics are averages of the last 10 epochs.

Obs. Mode	Indirect	VF Share	CR	AR Reduction	AR Decay	Entropy Coeff.	NN Size	WR	FS	UA
Simple	True	True	5	1	0.995	0.01	[256]	0.114	39.9	17.3
Simple	True	True	5	1	0.95	0.01	[2048, 2048, 1024, 512]	0.114	44.9	17.6
Simple	True	True	5	1	0.995	0.03	[512, 256]	0.11	41.8	17.5
Simple	True	True	5	5	1	0.01	[512, 256, 128]	0.106	43.1	17.5
Simple	False	True	5	5	0.98	0.03	[2048, 2048, 1024, 512]	0.105	46.2	17.4
Simple	True	True	5	5	1	0.01	[1024, 512, 256]	0.103	44.4	17.8
Simple	True	True	5	5	0.98	0.03	[2048, 2048, 1024, 512]	0.101	45.4	17.5
Simple	True	True	5	1	0.995	0.01	[4096, 2048, 512]	0.091	45.9	17.6
Simple	True	True	5	1	0.995	0.01	[8192, 2048, 512]	0.091	46.2	17.7
Simple	True	True	5	1	0.995	0.01	[512, 256]	0.087	43.7	17.7
Simple	True	True	5	1	0.995	0.01	[2048, 3072, 2048, 512]	0.087	46.4	17.6
Simple	True	True	5	1	0.995	0.03	[8192, 2048, 512]	0.084	46.4	17.5
Simple	True	True	5	1	0.995	0.01	[2048, 2048, 1024, 512]	0.084	46.6	17.7
Simple	True	True	5	1	0.995	0.01	[2048, 2048, 2048, 2048, 512]	0.06	49.1	17.8
Simple	True	True	5	1	0.98	0.03	[256, 256]	0.052	45.8	17.6
Simple	False	True	0	1	1	0.03	[2048, 2048, 1024, 512]	0.034	51.1	18
OH	True	True	5	5	0.98	0.03	[2048, 2048, 1024, 512]	0.022	52.4	19.4
OH	True	True	5	1	0.995	0.01	[8192, 2048, 512]	0.02	56.4	19.7
OH	False	True	5	1	0.995	0.01	[8192, 2048, 512]	0.016	57.5	19.5
OH	True	True	5	5	0.98	0.03	[2048, 2048, 1024, 512]	0.015	60.5	19.8

Table 6: Third table of all trainings where our policy was trained against the pre-programmed policy. The table is sorted by win rate. 'EOH' stands for efficient one hot encoding, 'OH' for one hot encoding, 'Obs. Mode' is the observation mode, 'Indirect' indicates if the agent only saw metadata of the game or each player's cards, 'VF Share' describes if the value function was integrated in the same neural net as the action value function, 'CR' is the curiosity reward, 'AR' stands for action reward, 'WR' is the win rate, 'FS' is the final score and 'UA' is undesirable actions. The last three metrics are averages of the last 10 epochs.

B Figures

B.1 Grid search, Training against Pre-Programmed Policy

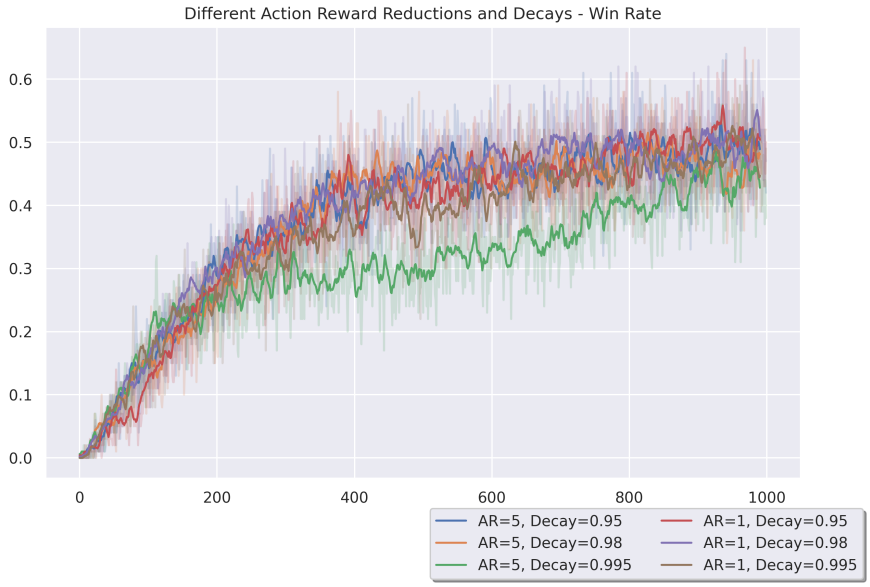

Figure 14: The average win rate over time for different action reward reduction (importance) and decay values is compared in this plot. Simple, direct observations, a shared NN for the value function, 0 curiosity reward, an entropy coefficient of 0.01, and a neural network of size [32] were used. Training is done against the pre-programmed policy.

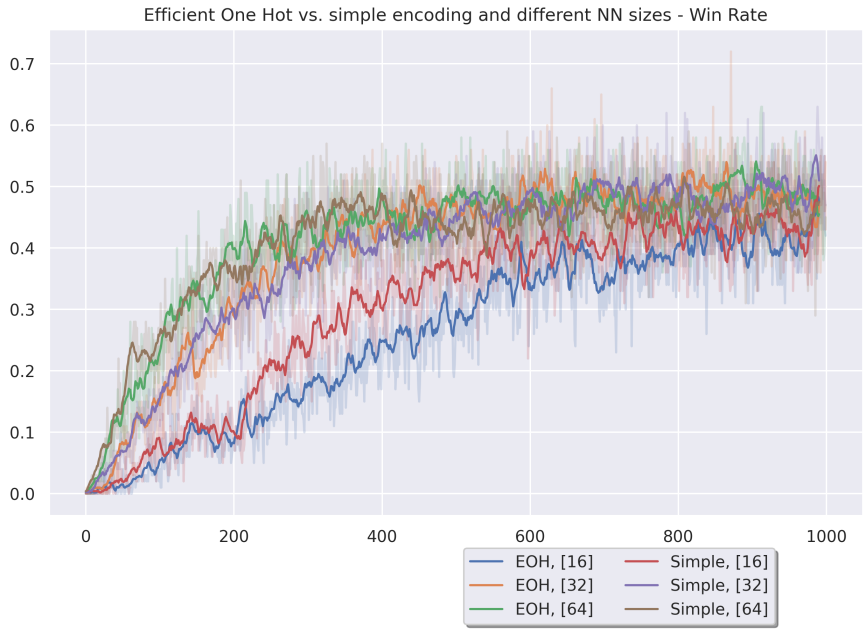

Figure 15: The average win rate over time for different observation encodings and neural net sizes is compared in this plot. Direct observations, a shared NN for the value function, 0 curiosity reward, an action reward reduction of 1, a decay of 0.98, and an entropy coefficient of 0.01 were used. Training is done against the pre-programmed policy.

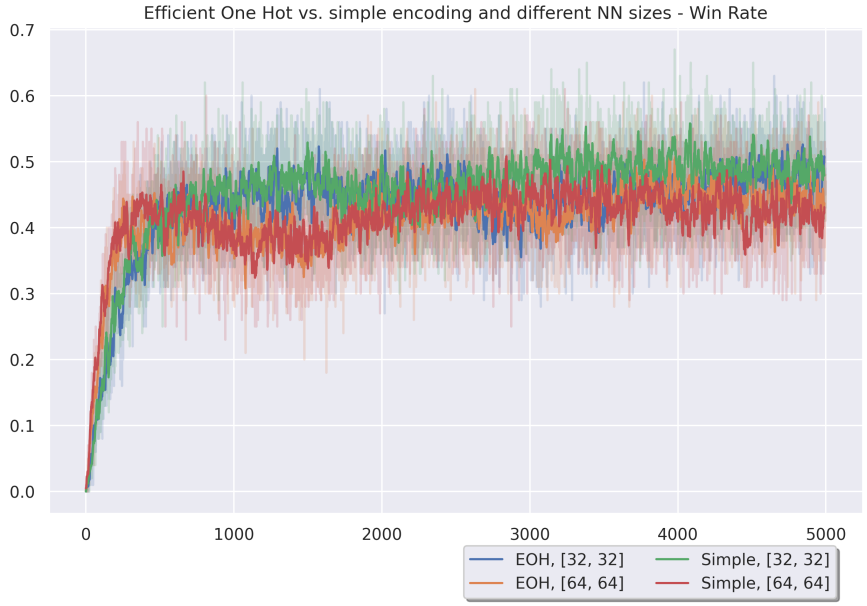

Figure 16: The average win rate over time for different observation encodings and neural net sizes is compared in this plot. Direct observations, a shared NN for the value function, 0 curiosity reward, an action reward reduction of 1, a decay of 0.98, and an entropy coefficient of 0.01 were used. Training is done against the pre-programmed policy.

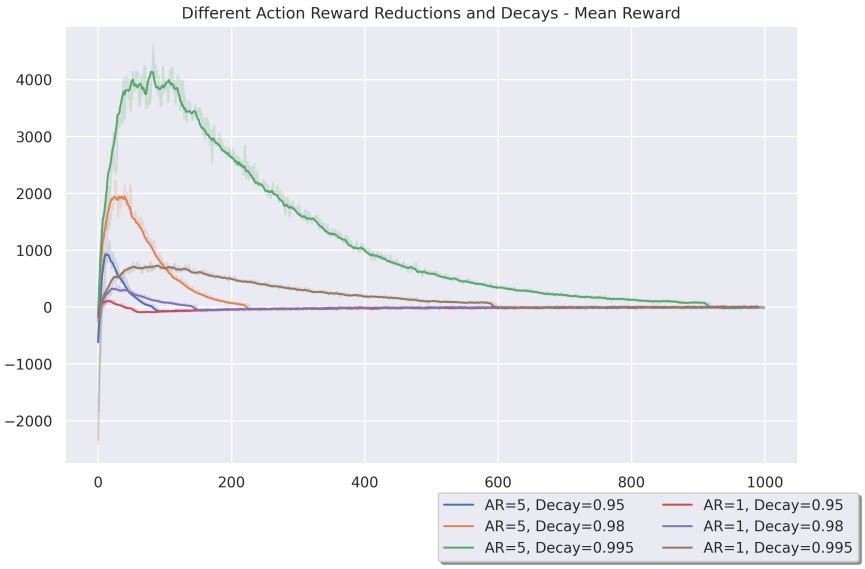

Figure 17: The mean reward over time for different action reward reduction (importance) and decay values is compared in this plot. Simple, direct observations, a shared NN for the value function, 0 curiosity reward, an entropy coefficient of 0.01, and a neural network of size [32] were used. Training is done against the pre-programmed policy.

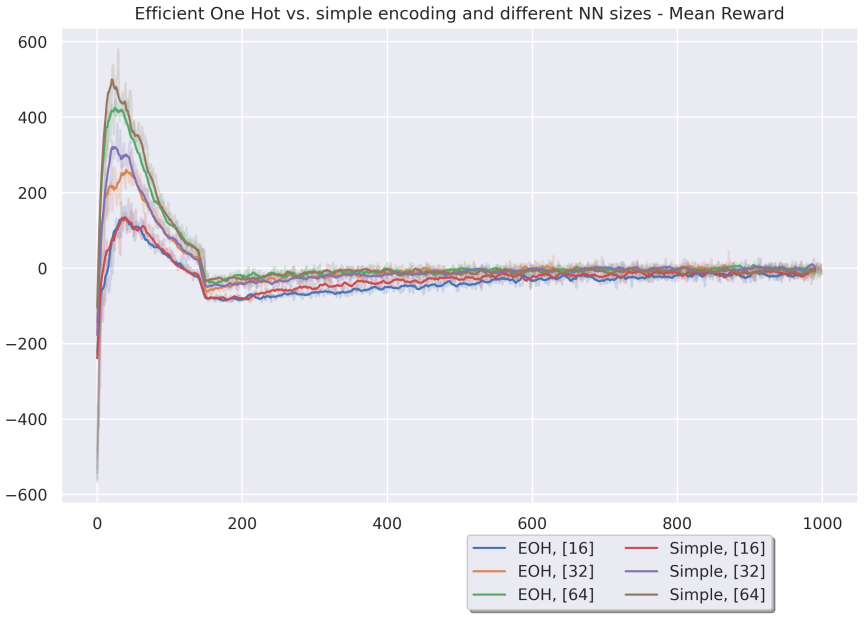

Figure 18: The mean reward over time for different observation encodings and neural net sizes is compared in this plot. Direct observations, a shared NN for the value function, 0 curiosity reward, an action reward reduction of 1, a decay of 0.98, and an entropy coefficient of 0.01 were used. Training is done against the pre-programmed policy.

Figure 19: The mean reward over time for different observation encodings and neural net sizes is compared in this plot. Direct observations, a shared NN for the value function, 0 curiosity reward, an action reward reduction of 1, a decay of 0.98, and an entropy coefficient of 0.01 were used. Training is done against the pre-programmed policy.

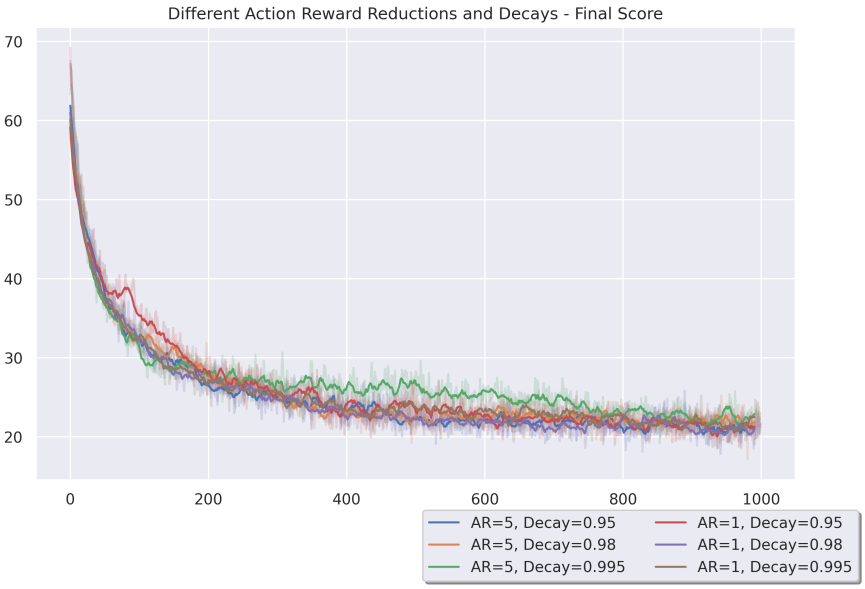

Figure 20: The average final score over time for different action reward reduction (importance) and decay values is compared in this plot. Simple, direct observations, a shared NN for the value function, 0 curiosity reward, an entropy coefficient of 0.01, and a neural network of size [32] were used. Training is done against the pre-programmed policy.

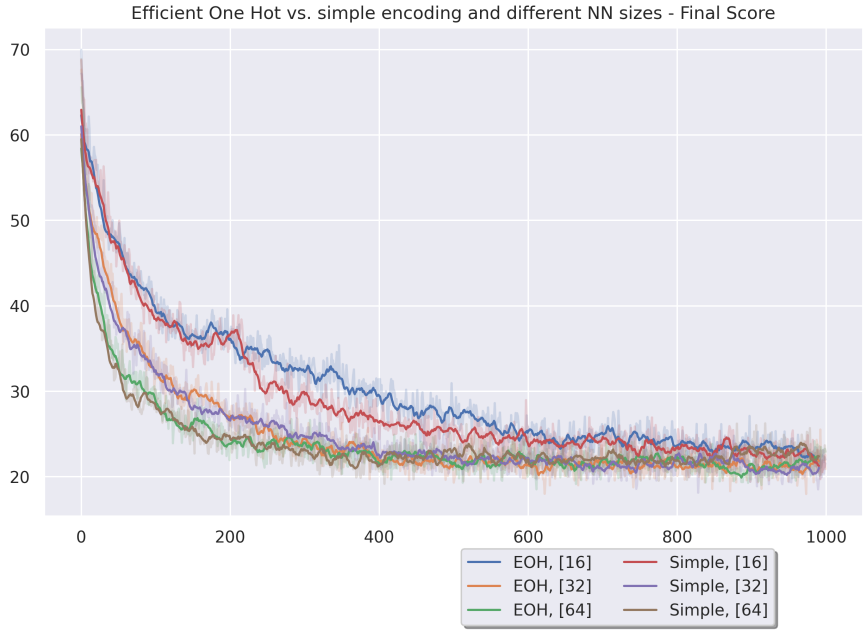

Figure 21: The average final score over time for different observation encodings and neural net sizes is compared in this plot. Direct observations, a shared NN for the value function, 0 curiosity reward, an action reward reduction of 1, a decay of 0.98, and an entropy coefficient of 0.01 were used. Training is done against the pre-programmed policy.

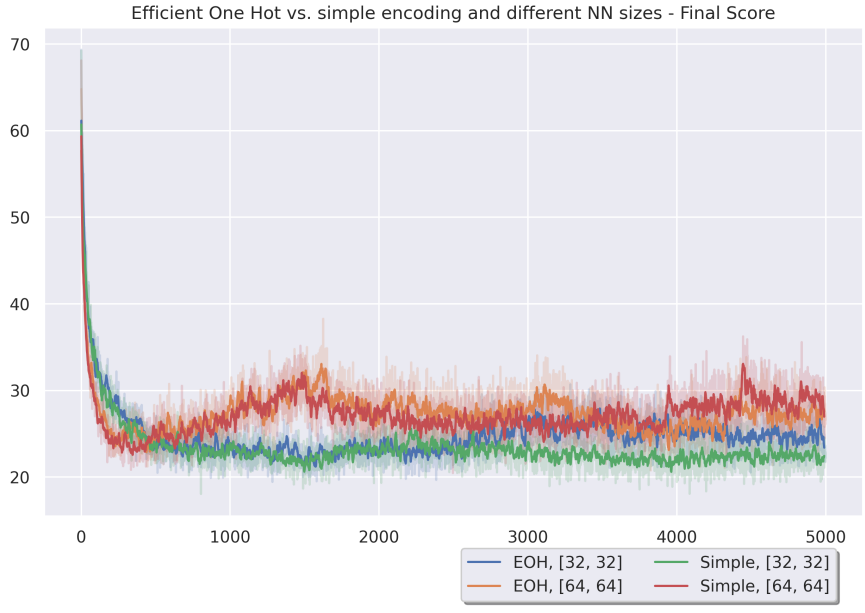

Figure 22: The average final score over time for different observation encodings and neural net sizes is compared in this plot. Direct observations, a shared NN for the value function, 0 curiosity reward, an action reward reduction of 1, a decay of 0.98, and an entropy coefficient of 0.01 were used. Training is done against the pre-programmed policy.

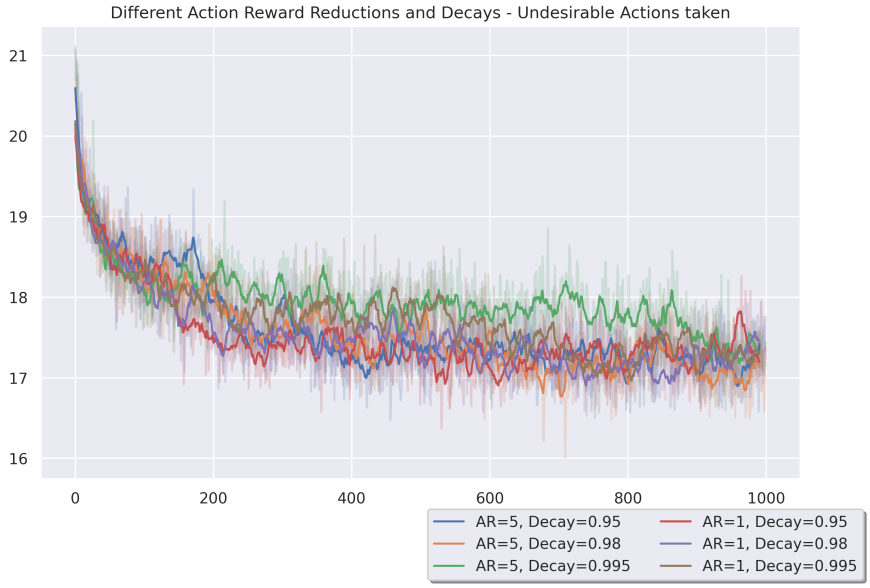

Figure 23: The average number of undesirable actions over time for different action reward reduction (importance) and decay values is compared in this plot. Simple, direct observations, a shared NN for the value function, 0 curiosity reward, an entropy coefficient of 0.01, and a neural network of size [32] were used. Training is done against the pre-programmed policy.

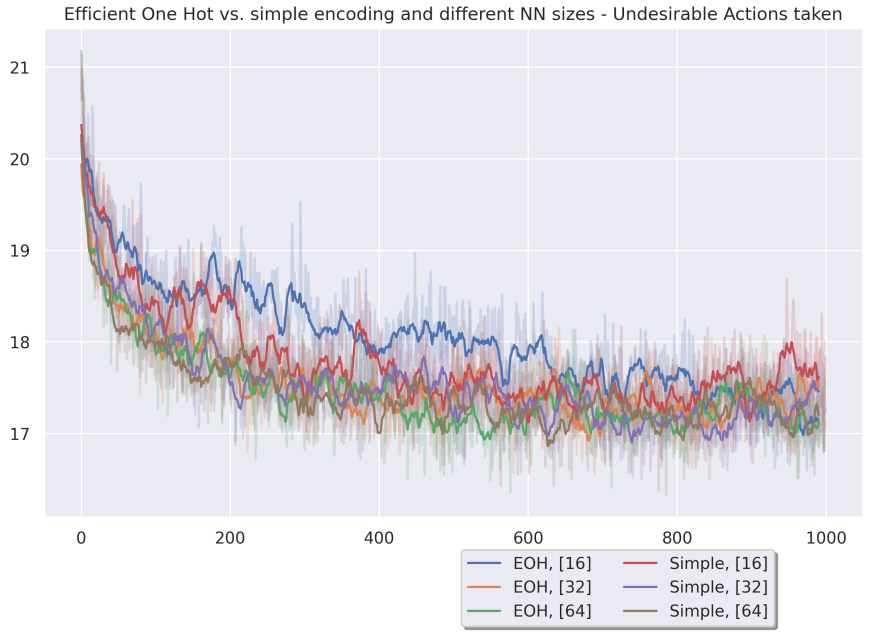

Figure 24: The average number of undesirable actions over time for different observation encodings and neural net sizes is compared in this plot. Direct observations, a shared NN for the value function, 0 curiosity reward, an action reward reduction of 1, a decay of 0.98, and an entropy coefficient of 0.01 were used. Training is done against the pre-programmed policy.

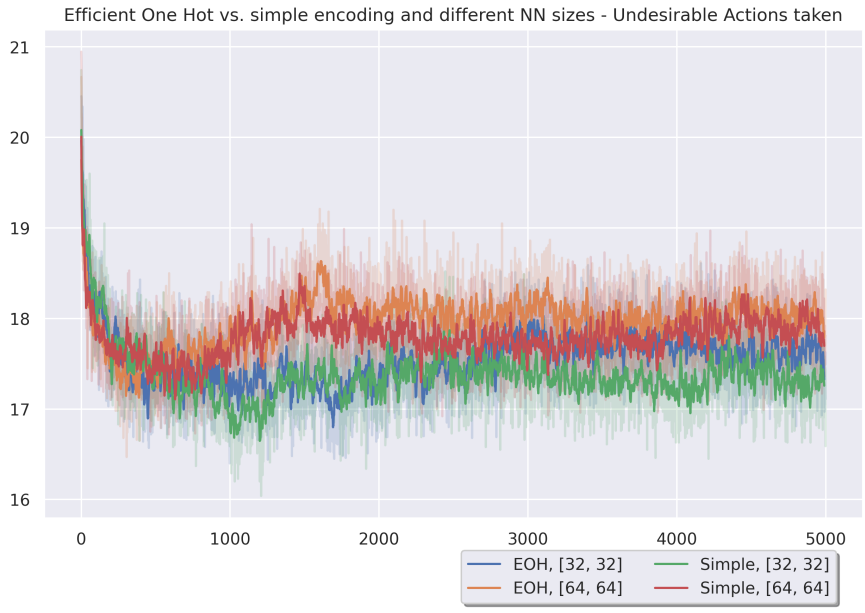

Figure 25: The average number of undesirable actions over time for different observation encodings and neural net sizes is compared in this plot. Direct observations, a shared NN for the value function, 0 curiosity reward, an action reward reduction of 1, a decay of 0.98, and an entropy coefficient of 0.01 were used. Training is done against the pre-programmed policy.

B.2 Grid search, Training against Single-Player Policy

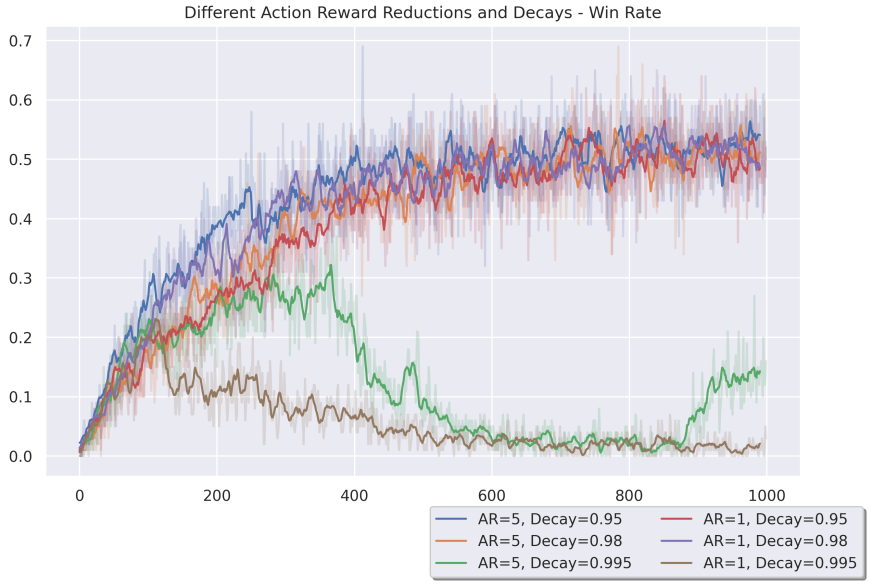

Figure 26: The average win rate over time for different action reward reduction (importance) and decay values is compared in this plot. Simple, direct observations, a shared NN for the value function, 0 curiosity reward, an entropy coefficient of 0.01, and a neural network of size [32] were used. Training is done against Guillaume Barthe's single-agent policy.

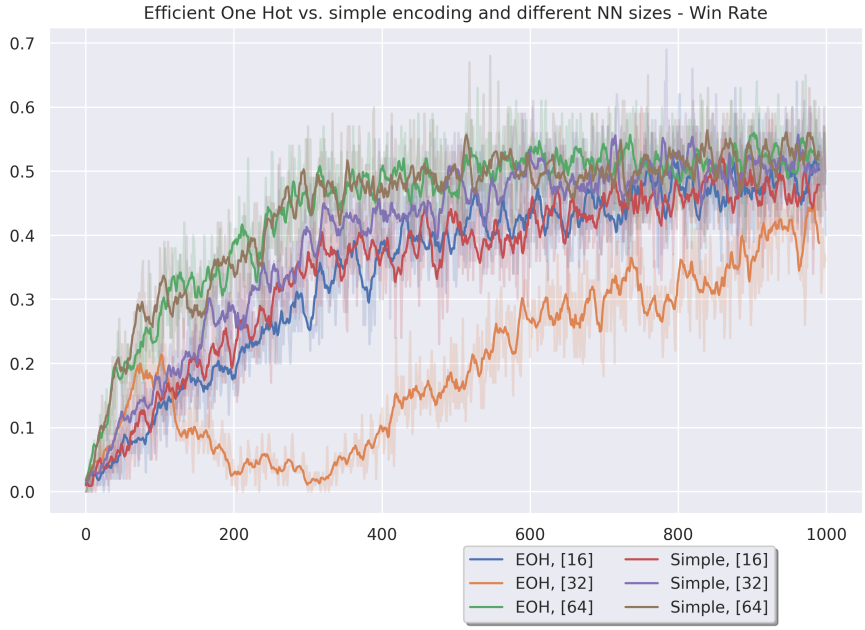

Figure 27: The average win rate over time for different observation encodings and neural net sizes is compared in this plot. Direct observations, a shared NN for the value function, 0 curiosity reward, an action reward reduction of 5, a decay of 0.98, and an entropy coefficient of 0.01 were used. Training is done against Guillaume Barthe's single-agent policy.

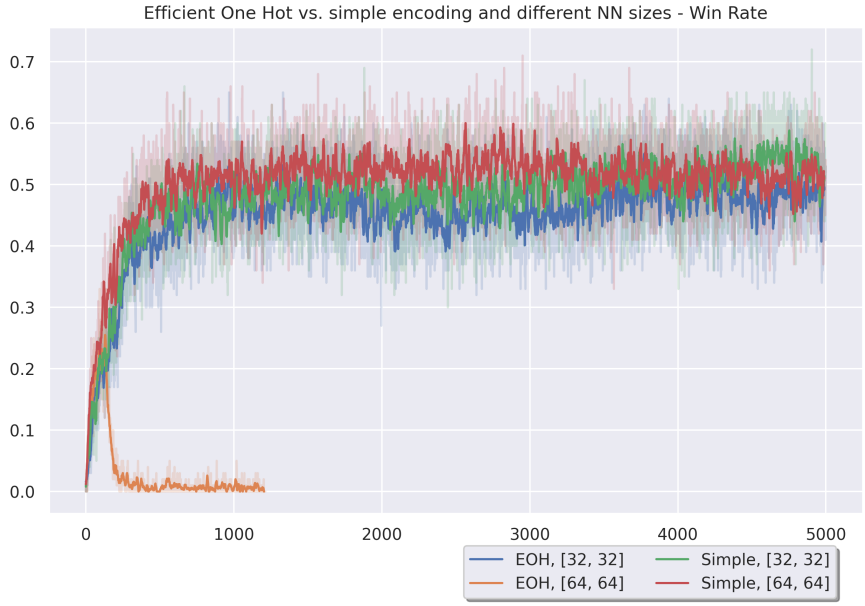

Figure 28: The average win rate over time for different observation encodings and neural net sizes is compared in this plot. Direct observations, a shared NN for the value function, 0 curiosity reward, an action reward reduction of 5, a decay of 0.98, and an entropy coefficient of 0.01 were used. Training is done against Guillaume Barthe's single-agent policy.

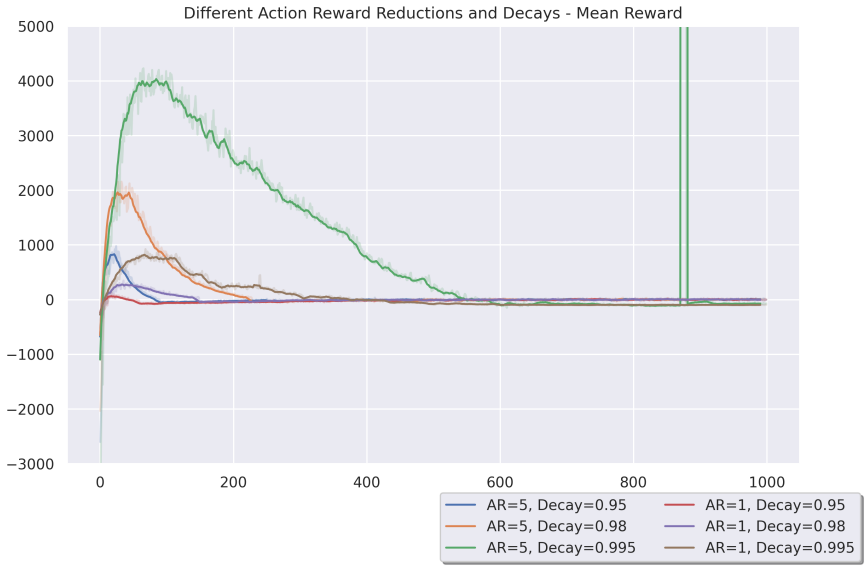

Figure 29: The mean reward over time for different action reward reduction (importance) and decay values is compared in this plot. Simple, direct observations, a shared NN for the value function, 0 curiosity reward, an entropy coefficient of 0.01, and a neural network of size [32] were used. Training is done against Guillaume Barthe's single-agent policy.

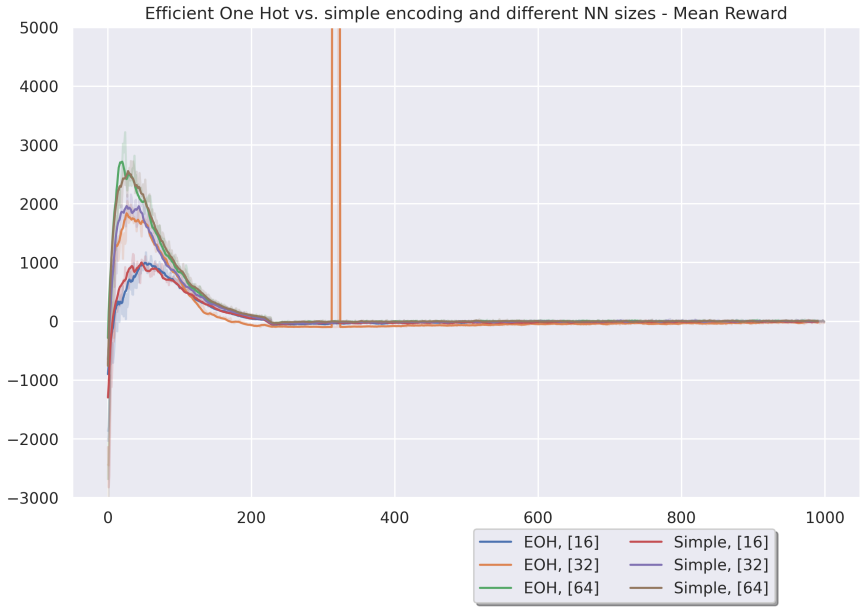

Figure 30: The mean reward over time for different observation encodings and neural net sizes is compared in this plot. Direct observations, a shared NN for the value function, 0 curiosity reward, an action reward reduction of 5, a decay of 0.98, and an entropy coefficient of 0.01 were used. Training is done against Guillaume Barthe's single-agent policy.

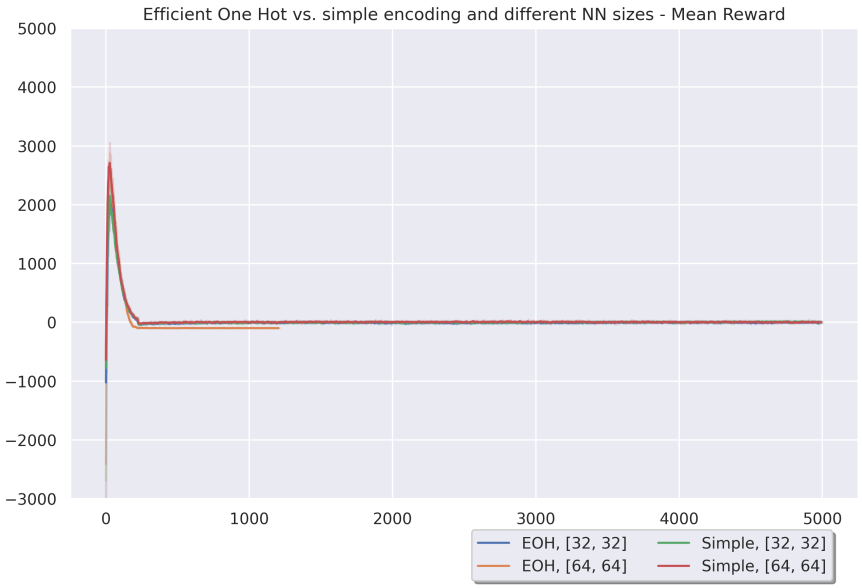

Figure 31: The mean reward over time for different observation encodings and neural net sizes is compared in this plot. Direct observations, a shared NN for the value function, 0 curiosity reward, an action reward reduction of 5, a decay of 0.98, and an entropy coefficient of 0.01 were used. Training is done against Guillaume Barthe's single-agent policy.

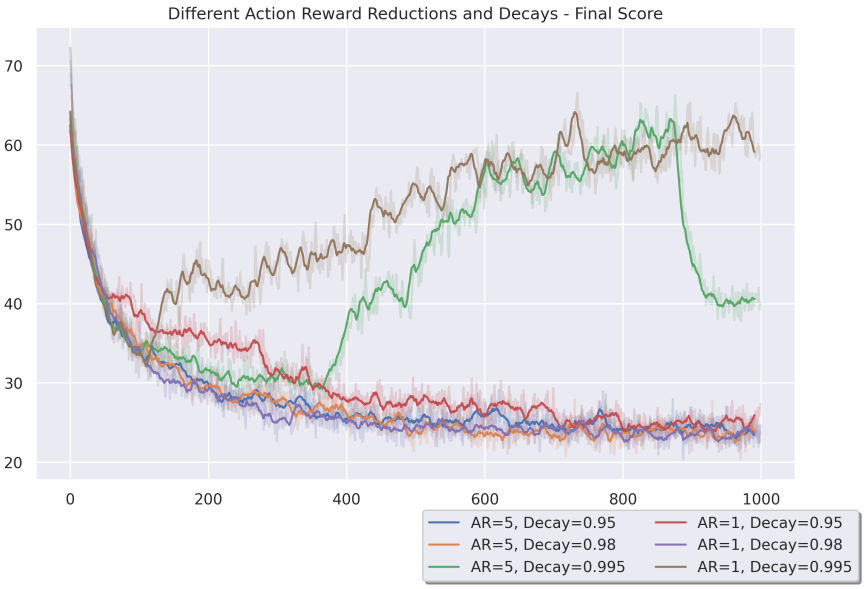

Figure 32: The average final score over time for different action reward reduction (importance) and decay values is compared in this plot. Simple, direct observations, a shared NN for the value function, 0 curiosity reward, an entropy coefficient of 0.01, and a neural network of size [32] were used. Training is done against Guillaume Barthe's single-agent policy.

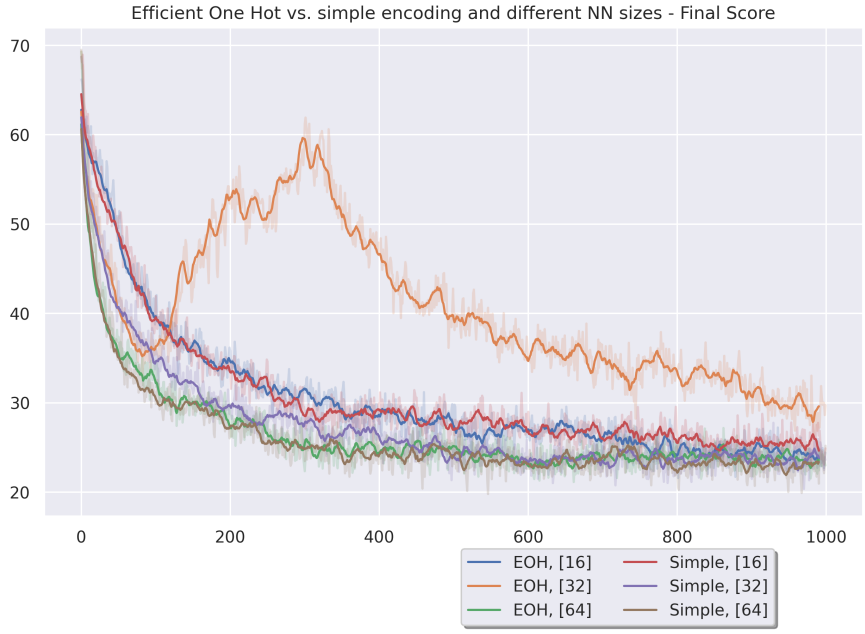

Figure 33: The average final score over time for different observation encodings and neural net sizes is compared in this plot. Direct observations, a shared NN for the value function, 0 curiosity reward, an action reward reduction of 5, a decay of 0.98, and an entropy coefficient of 0.01 were used. Training is done against Guillaume Barthe's single-agent policy.

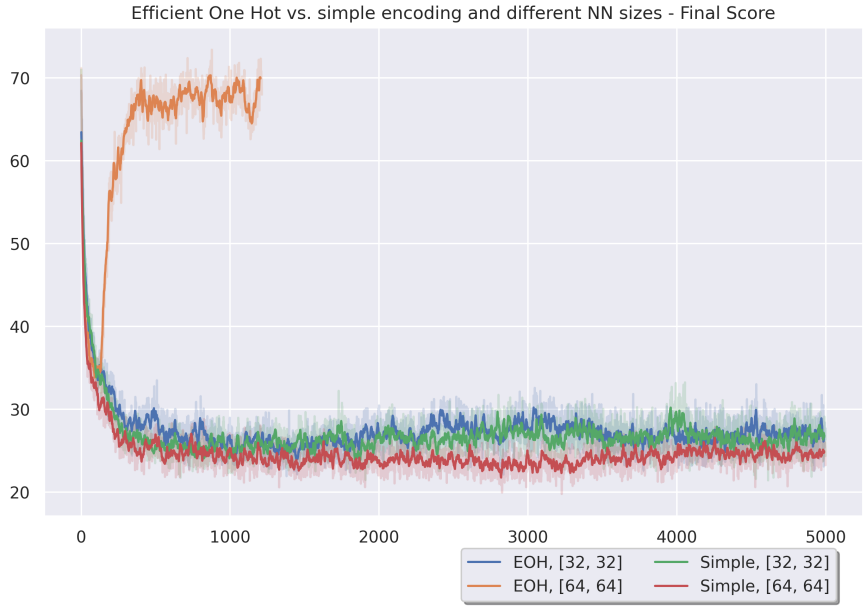

Figure 34: The average final score over time for different observation encodings and neural net sizes is compared in this plot. Direct observations, a shared NN for the value function, 0 curiosity reward, an action reward reduction of 5, a decay of 0.98, and an entropy coefficient of 0.01 were used. Training is done against Guillaume Barthe's single-agent policy.

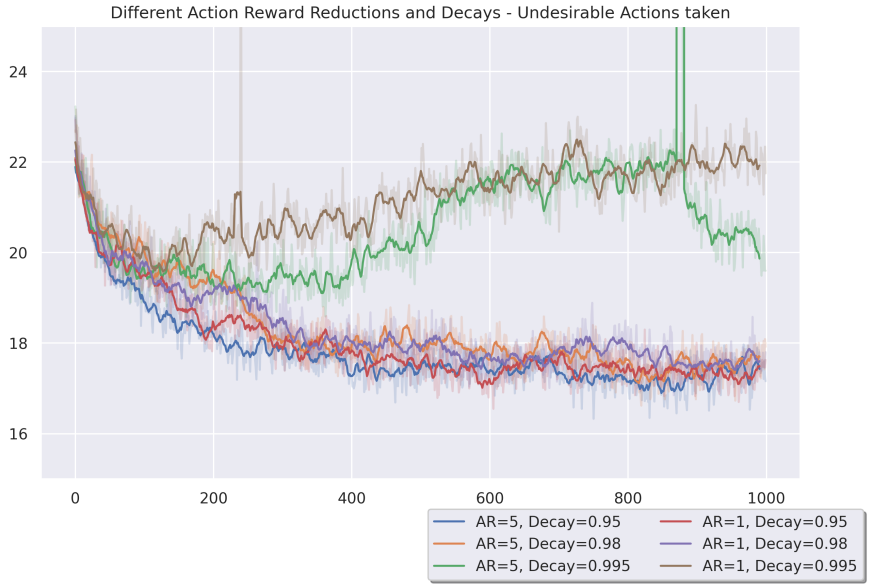

Figure 35: The average number of undesirable actions over time for different action reward reduction (importance) and decay values is compared in this plot. Simple, direct observations, a shared NN for the value function, 0 curiosity reward, an entropy coefficient of 0.01, and a neural network of size [32] were used. Training is done against Guillaume Barthe's single-agent policy.

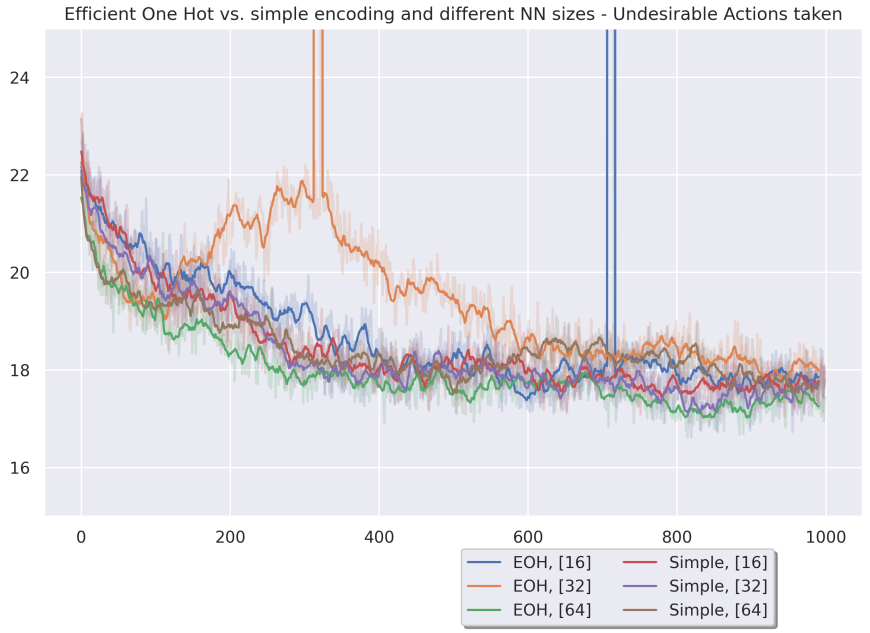

Figure 36: The average number of undesirable actions over time for different observation encodings and neural net sizes is compared in this plot. Direct observations, a shared NN for the value function, 0 curiosity reward, an action reward reduction of 5, a decay of 0.98, and an entropy coefficient of 0.01 were used. Training is done against Guillaume Barthe's single-agent policy.

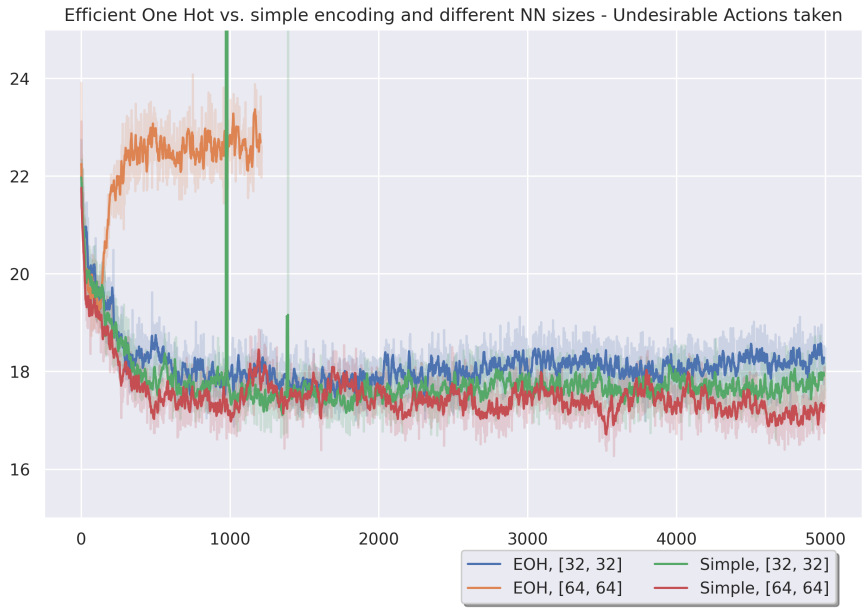

Figure 37: The average number of undesirable actions over time for different observation encodings and neural net sizes is compared in this plot. Direct observations, a shared NN for the value function, 0 curiosity reward, an action reward reduction of 5, a decay of 0.98, and an entropy coefficient of 0.01 were used. Training is done against Guillaume Barthe's single-agent policy.