Learning to play a 2-player adversarial game : Cathedral

Final report

Simon Defradas Adrien Goldszal Gabriel Mercier Mathias Perez

***Abstract*—In this project, we teach agents to play the Cathedral 2-player board game. We first do a review of DQN and PPO algorithms, study and optimize their performance in this setting. This also implies adapting the environment to our needs. In addition, we then test the Representation Learning method and analyze its impact on the training objective.**

I. Introduction

A. Learning 2-Player Adversarial Games

With this project, we wanted to study how an agent could efficiently learn to play a two-player game. What were the best methods in terms of training in this adversarial setting, as well as the best algorithms. Going further, we wanted to see how Representation Learning techniques could help to scale and accelerate this learning.

Learning to play board games was one of the first elements of the advent of RL, notably through the game of chess, which was Demis Hassabis's main motivation behind founding Google DeepMind in 2014, which culminated with AlphaZero in 2017 [1]. These 2-player adversarial games bring a unique challenge to them because of their non-stationarity, the high branching factor, and the sparse rewards.

Training such algorithms can be complicated. Should supervision be involved? How should an opponent be crafted? Or should it be purely self-play, as is increasingly the case today. The question of structuring this adversarial self-play setting is also challenging. Should there be two policies, or just one? Choosing and tuning the optimal algorithm is also crucial.

While algorithms like **Proximal Policy Optimization** (PPO) demonstrate state-of-the-art performance on high action spaces and multi-agent learning, works like those by DeepMind and AlphaZero [2] show that having **Monte-Carlo Tree Search** (MCTS) in addition to deep RL clearly improves performance in adversarial 2 player games like Chess, for example.

Deep Q-Networks (DQN) [3] is not a state-of-the-art model for adversarial games. However, it remains a viable choice with certain modifications. Its simplicity and accessibility make it particularly attractive for experimentation, as it is relatively easy to implement and requires fewer computational resources compared to policy gradient methods.

An idea we also wanted to explore was **Representation Learning**, or learning meaningful representations of the large observation space, which can help reduce dimensionality, improve generalization, and facilitate style separation in our diffusion model. Various approaches exist for learning such representations, including auto-encoders, contrastive learning, and self-supervised learning methods [4].

We therefore decided to study a 2-player adversarial board game called **Cathedral** [5], where dark and light factions battle for terrain on a grid (or fortified village), by placing pieces in turn-based fashion. This strategy game, for which we found an open source, but **very much unused**, PettingZoo environment [6], allowed us to implement and test our algorithms.

Our code implementation can be found in https://github.com/gabriel-mercier/Cathedral-RL-CL

II. Background

Fig. 1. Illustration of the cathedral board game [6]

A. The Cathedral Game

The Cathedral game [5] involves two players trying to place as many of their pieces as possible on a fixed 10×10 square grid. The first player starts by placing a special piece, the cathedral, shaped like a cross. This piece does not contribute to the player's score. Then, in a turn-based manner, each player places one of their pieces on the board. The **value of each piece corresponds to its size** in grid squares. The objective is to get the highest score (i.e., to have put most large pieces on the board) once there is no more space left on the board.

An additional challenge in this game is the ability to encircle an area using one's pieces, **creating 'territory'**. This territory

becomes a protected zone: any opponent's piece already within it is immediately removed, and the opponent is prevented from placing any new pieces inside. Successfully creating territory is difficult and serves as a key strategic goal, in addition to efficiently placing high value pieces.

B. Deep Q-Learning

Deep Q-Learning (DQN) leverages deep neural networks to approximate the action value function $Q(s, a)$. DQN is capable of handling large state spaces by using a neural network as a function approximator [3].

1) Replay Buffer and Target Network: As described in the course, we used a **Replay Buffer** to break the correlations between consecutive updates and a **Target Network** to reduce instability.

In an adversarial setup, we use the following equation:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(r \text{ - } \gamma \max_{a'} Q_{\text{t}}(s',a') - Q(s,a) \right) \quad (1)$$

We use a minus sign $(-)$ instead of a plus sign $(+)$ due to the adversarial nature of the game. Unlike standard Q-learning, where the agent maximizes its future reward, here the next action a' represents the opponent's move, which we aim to minimize.

2) Double DQN: Standard DQN suffers from an overestimation bias when computing the target Q-values. **Double DQN** (DDQN) mitigates this by decoupling action selection and value estimation [7] (see the Appendix).

a) Prioritized Experience Replay (PER): In standard experience replay, transitions are sampled uniformly from the buffer. However, in **Prioritized Experience Replay**, important transitions, those with high temporal difference (TD) errors, are sampled more frequently [8] (see the Appendix).

b) Exploration Strategies: ϵ-Greedy vs. Boltzmann: ϵ-Greedy Exploration is described in the course. Boltzmann Exploration: Instead of selecting the best action outright, the Boltzmann exploration sample actions are based on a softmax probability distribution over TD-errors (see the Appendix).

C. Proximal Policy Optimization (PPO)

PPO is often considered one of the state-of-the-art RL algorithms and is often used in multi-agent settings. It also has an advantage as it works with discrete action spaces, unlike other algorithms such as Deep Deterministic Policy Gradient (DDPG) and Twin Delayed DDPG (TD3).

1) Algorithm structure: PPO works in an actor-critic way (policy θ and value function ϕ) but is on-policy. One or multiple episodes $\mathcal{D}_k = \{\tau_i\}$ are added to a buffer by running the policy and then the PPO is updated at the end and goes through the trajectories, computing advantages $\hat{A}_t$ and discounted rewards $\hat{R}_t$ before updating θ (actor) and ϕ (critic). This step is done multiple times per update. [9]

The policy is updated via stochastic gradient ascent by maximizing the PPO-Clip objective, the value function is fitted by regression on mean-squared error and updated by gradient descent. (See the Appendix for detailed equations)

2) Generalized Advantage Estimation (GAE): GAE [10] is a technique used in PPO to reduce variance while maintaining a low bias in advantage estimation. Instead of relying on a single-step or Monte Carlo estimate of the advantage function, GAE introduces a trade-off between bias and variance by using a weighted sum of temporal-difference (TD) residuals. The advantage function is estimated as:

$$\hat{A}_t = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l} \quad (2)$$

where the TD residual is defined as:

$$\delta_t = r_t + \gamma V_\phi(s_{t+1}) - V_\phi(s_t) \quad (3)$$

γ is the discount factor and λ is a smoothing parameter that balances bias and variance.

D. Representation Learning with a Variational Autoencoder

Variational Autoencoders (VAEs) [11] are a class of generative models that learn structured latent representations of data by optimizing a probabilistic lower bound on the data likelihood.

A VAE consists of an encoder network $q_\phi(z|x)$ that approximates the true posterior distribution of latent variables given input data, and a decoder network $p_\theta(x|z)$ that reconstructs the input from the latent representation. The model is trained by maximizing the Evidence Lower Bound (ELBO):

$$\log p_\theta(x) \geq \mathbb{E}_{q_\phi(z|x)} \left[\log p_\theta(x|z)\right] - D_{KL}\left(q_\phi(z|x) \| p(z)\right) \quad (4)$$

where D_{KL} is the Kullback-Leibler divergence that regularizes the approximate posterior $q_\phi(z|x)$ to be close to the prior $p(z)$, typically chosen as a standard Gaussian $\mathcal{N}(0, I)$.

In the context of reinforcement learning for board games, VAEs can be used to learn low-dimensional representations of the state space. Given an observation tensor, the encoder extracts a compact latent representation that captures meaningful game dynamics, which can help with policy learning.

III. Methodology/Approach

A. The Cathedral PettingZoo environment

To be able to train agents in the cathedral game, we use and **adapt an open source PettingZoo environment** implementation by Elliot Towers [6]. This multi-agent RL environment allows agents to train in an adversarial setting, placing pieces turn by turn.

Observations In the environment, the agent observes the board through five one-hot encoded layers providing different information (pieces and territory for both the player and the cathedral). The observation is therefore of size 5 * (board size, board size). The default board size is 10, but we chose to reduce it to 8 for computational efficiency, resulting in a vector of size (5, 8, 8) = **320 observations**.

Actions The agent can place any of its pieces at each turn, choosing its position and orientation. The action space is therefore (pieces * positions * orientations) = **1753 actions** for a board size of 8. By default, there are 15 pieces, each having their own size and shape on the board. There is an action mask to force only valid actions by the agent.

Rewards We modified the environment to provide two reward structures:

- **Sparse Reward**: A binary reward system in which the agent receives +1 for victory and -1 for defeat (0 for draw).
- **Continuous Reward**: A heuristic-based reward function that considers both the size of the placed piece and the amount of territory claimed. The reward for each move is given by:

  Reward = claimed territory + piece score

  where the piece score is defined as:

  $$\text{piece score} = \text{piece size} - \max(\text{legal piece sizes})$$

  This encourages putting down larger pieces and claiming territory. A bonus and penalty are applied (to the player and its opponent, respectively) if an opponent's piece has been pushed out of the board, equal to this piece's value. Then the victory gives +10 and the defeat a -10 (0 for draw).

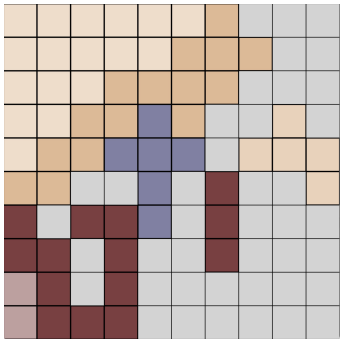


Fig. 2. A rendered frame of the Cathedral environment [6]

Remarks The provided environment, as it does not appear to be widely used (we could not find any existing implementation of RL algorithms), presents several issues:

- **Biased starting conditions**: Due to the placing of the cathedral which was always done by player 1, the first player has only a 37.5% win rate when two random players compete against each other. Importantly, **we modified the environment so that there was a random cathedral initialization between the two players**, restoring the balance in the game. This is key for a self-play setup to learn how to play the game.
- **Human player vs algorithm**: Numerous bugs occur in the human player interface.
- **Wrapper-related issues**: Certain wrappers cause inconsistencies that require us to manually handle the game logic.

B. DQN

We tested the simple sparse rewards on victory and losing (+1/-1) without much success. We therefore decided to try the **move-based approach** to encourage good behavior. It should be noted that, in standard training, when the agent produces a probability distribution over all possible actions, **a mask of legal actions is applied** only.

An implementation of Deep Q-Networks (DQN) was developed (further detail is provided in the appendix), incorporating a standard experience replay buffer as well as a prioritized replay buffer (using a Sum-Tree).

The latter still requires parameter tuning to optimize performance. We arbitrarily choose **fixed parameters**:

- Batch size of 64 for training.
- Learning rate of 10^{-3}.
- The discount factor γ is set to 0.95, as each episode is relatively short (approximately 18 moves), making $\gamma^{18} \approx 0.4$ a reasonable choice.
- The target network update frequency set to 30, based on literature recommendations.

Key hyperparameters requiring further tuning:

- Number of episodes.
- Replay Buffer properties (size and whether prioritization is used).
- Exploration strategy (ϵ-greedy vs Boltzmann).

The DQN agent was tested in **three different training setups**:

- Self-play training: The DQN is trained by continuously playing against itself.
- Training against a random player.
- Training with high penalization of illegal actions.

C. PPO

1) Architecture: A PPO with shared convolutional layers for an actor and critic head was implemented. Convolutional layers perform well for feature extraction in observation spaces such as ours that represent five different boards.

- Shared CNN : 3 layers : Following [3], we increase the number of channels from 3 to 64 to learn rich features
- Actor & Critic : 2 linear layers of size 512 for the actor and critic.

2) Training setup: We train the PPO in a **self-play** manner where the primary agent plays against an older version of itself. This seemed to be standard in adversarial training regimes. We also regularly test it against random players during training to log progress.

PPO can be pretty unstable in an adversarial setting and with longer episodes of 20 steps. For **stability** and better learning, we implement the following elements which showcased better results :

- Low learning rates of $l_{critic} = 0.0002$ and $l_{actor} = 0.0005$ which are slightly on the lower end of the default baseline.

- We wait **5** episodes before updating the policy to stabilize the learning.
- Policy updates are done with $K_{epochs} = 20$ as default
- We use GAE with $\lambda = 0.95$ which is a default usual value
- We do soft updates on the target where we only update 30% of the weights every 10 episodes, and a complete update after 100 episodes

Other parameters were also fixed at standard values : $\epsilon_{clip} = 0.2$, $\gamma = 0.97$, and an entropy coefficient of 0.01.

D. VAE

We train a simple VAE with the objective of **learning meaningful representations of the observation space**, and using it as a replacement for the convolutional layers of our DQN and PPO.

The VAE has a latent space representation of 32, smaller than the observation space, with the aim of learning denser representations and potentially accelerating the learning of our RL algorithms.

We train on 4000 games, half of which being with random players, and the other half with a trained PPO agent, to have a more diverse set of boards. Training is done over 300 epochs.

E. Tree methods

We also experimented with **tree-based methods like minimax and MCTS**, which yielded less significant results. Further developments on these methods are detailed in the appendix.

IV. Results and Discussion

To perform a quantitative analysis of our tested methods, we evaluated them using three different metrics over 10000 episodes:

- **Win rate vs a random player.**
- **Win rate vs two competitive models.**
- **Average reward per episode.**

A. DQN Results

Since hyperparameters are not independent and training time ranges from 15 minutes to 3 hours, fine-tuning them perfectly is a challenging task. However, we aimed to make our tuning process as objective as possible.

1) Number of episodes: To evaluate the number of episodes required for convergence, we trained a DQN model over 10,000 episodes using Boltzmann exploration, a prioritized replay buffer, and a buffer size of 40,000 in a self-play setting. However, the results remained consistent across different scenarios.

As shown in Figure 3, the average reward **stabilized between 2,000 and 4,000 episodes** (but the reward itself is very unstable). This observation is further supported by Figure 4, where we see that the win rate against a random player rapidly increases to between 95% and 98% in just about $1,000$ episodes.

Fig. 3. Average reward per episode during training (additional training metrics available in the Appendix).

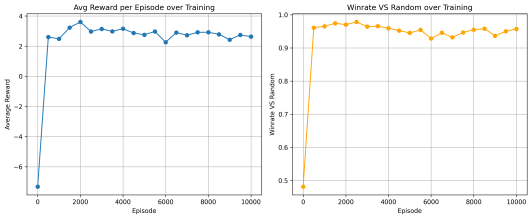


Fig. 4. From left to right vs. random player over 10,000 episodes: (a) Average Reward (b) Win rate

This model reached an average reward of *2.638* and a win-rate VS random player of *0.958%*. Based on these findings, we decided to conduct further experiments with 4,000 episodes.

2) Exploration: We compared both exploration strategies described above, with an annealed rate (see Appendix) with a decay rate of 500 to obtain comparable results (Table IV).

Strategy	Avg reward	Win rate
Playing VS Random		
Epsilon	2.731	0.939
Boltzmann	**2.851**	**0.963**
Playing against each other		
Epsilon	-0.0239	0.488
Boltzmann	**-0.0236**	**0.512**

TABLE I
Comparison of Epsilon-Greedy and Boltzmann strategies.

We observe that the **Boltzmann strategy outperforms** Epsilon-Greedy in both cases, achieving a higher win rate and average reward against a random player, as well as an advantage in head-to-head matches.

3) Replay Buffer: Classic Replay Buffer VS Prioritized Replay Buffer (Table III):

Similarly, a **Prioritized Replay Buffer gives better results** over a 4000 episodes training.

4) Ideas: Here, we explored different training strategies inspired by the literature.

- **VS Illegal**: Since over 90% of initial actions were illegal, we first trained the model to make only legal moves (for 1500 episodes) before standard self-play training (for 2500 episodes).

- **VS Random**: Similarly, pure self-play can lead to local optima, so we tested initial training against a random player.

We compare these methods to a classic self-play training baseline of 2500 episodes (Table IV).

Despite training on more episodes, **the proposed methods doesn't give better results** than classic training, illustrating that enforcing legality early or pre-training against a weak opponent does not provide a strategic advantage over standard training.

B. PPO Results

Adversarial self-play training with the PPO over 10 000 episodes presented **similar results to the DQN**. Presenting convergence of the rewards, value function and policy updates at around 3000 episodes.

Fig. 5. PPO Self Play Training results with Soft Updates

The study of the policy ratio stability and the value function were integral to the adaptation of the algorithm and the changes made to stabilize training such as smoother policy updates and the larger replay buffer. (See appendix 10 for value functions, policy ratios and losses). The figure 9 show the problems with too sudden policy changes on training, which lead to less stability.

Winrates against random agents plateau rapidly to *98%* on average after around 900 episodes which show that the agents have learned meaningful policies (See appendix 10). Examining and printing out games also corroborates this claim, where the larger pieces are put down first and territory is conquered.

C. Representation Learning with a VAE

Training of the VAE with the 4000 games seemed to **present positive training results**, with the stabilization of the loss and a reconstruction rate of *94,3%* (see Appendix).

The VAE was then used to replace the convolutional layers of the PPO and tested as a way to encode representations for the actor and critic heads. The VAE weights are frozen to see if the learned representations could impact the algorithm training.

Interestingly, training the PPO with the VAE **yielded multiple difficulties**. While the learning starts at around the same rate and with the same evolution as with the convolutional architecture, the results become unstable in scenarios with richer rewards. This could be due to the encoder not being trained on sufficiently different samples, especially samples where two good players play against each other, i.e more advanced winning policies. It's also worth noting that there are no updates at each stage, and even with these frequent weight updates of the target, the behaviour and results of the primary and target players diverge. Detailed analysis in the appendix 12 show that losses still fluctuate importantly, as well as rewards and values even after 10 000 episodes.

Concerning the discrepancy in behaviour between the target and primary agents, it is still unclear why this happens. Still discrepancies in initialization of the VAE weights or optimizer momentum could potentially have an impact on such small latent spaces of 32 like we have here despite frequent weight updates.

D. PPO vs DQN

To compare our two best models against each other, we ran 1,000 episodes with a temperature of 1 (Table II).

Strategy	Avg Reward	Win Rate
DQN	**-0.775**	**0.784**
PPO	-1.481	0.172
Draw	-	0.044

TABLE II
COMPARISON OF DQN AND PPO STRATEGIES.

The results indicate that **DQN significantly outperforms PPO** in win rate.

V. CONCLUSIONS

In this work, we explore reinforcement learning techniques for playing the 2-player adversarial board game Cathedral. We implemented and compared various algorithms, including Deep Q-Network (DQN) and Proximal Policy Optimization (PPO), within a customized PettingZoo environment.

DQN is extensively tested: our experiments demonstrate that careful tuning of hyperparameters, exploration strategies, and replay buffer techniques significantly impacts performance. However, pre-training on legal moves or against a weaker opponent did not yield significant improvements. PPO presents positive results similarly to DQN after additional modifications to stabilize learning such as soft updates, larger buffer updates and GAE are implemented. Representation learning is implemented through a VAE and incoporated into the Reinforcement Learning training frameworks. While enabling learning with denser, smaller representations and less parameters, it presented instabilities that need to be further studied.

Overall, this work underscores the complexities and the different methods adapted for adversarial learning

REFERENCES

[1] Wikipedia contributors, "Demis Hassabis." https://en.wikipedia.org/wiki/Demis_Hassabis.

[2] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," 2017.

[3] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," 2013.

[4] Y. Bengio, A. Courville, and P. Vincent, "Representation learning: A review and new perspectives," 2014.

[5] R. B. Moore, "A little history." https://www.cathedral-game.co.nz/about-history.html.

[6] E. Tower, "cathedral-rl." https://github.com/elliottower/cathedral-rl, 2023.

[7] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," 2015.

[8] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," 2016.

[9] OpenAI, "Proximal policy optimization, spinning up documentation." https://spinningup.openai.com/en/latest/algorithms/ppo.html.

[10] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation," 2018.

[11] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," 2022.

[12] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *Machine Learning: ECML 2006* (J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, eds.), (Berlin, Heidelberg), pp. 282–293, Springer Berlin Heidelberg, 2006.

[13] D. E. Knuth and R. W. Moore, "An analysis of alpha-beta pruning," *Artificial Intelligence*, vol. 6, no. 4, pp. 293–326, 1975.

VI. APPENDIX

A. DQN

1) DoubleDQN Equation:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(r + \gamma Q_{\text{target}}(s', \arg\max_{a'} Q(s',a')) - Q(s,a) \right) \tag{5}$$

2) Prioritized Replay Buffer Equation:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \tag{6}$$

where:

- $P(i)$ is the probability of sampling transition i,
- p_i is the priority of transition i (e.g., $|\delta_i| + \epsilon$, with δ_i being the TD error),
- α controls the degree of prioritization (typically $\alpha = 0.6$).

3) Boltzmann Equation:

$$P(a) = \frac{\exp(Q(s,a)/T)}{\sum_{a'} \exp(Q(s,a')/T)} \tag{7}$$

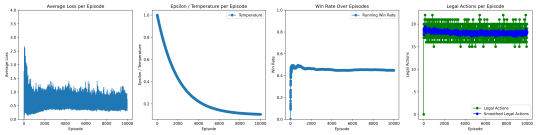
where T is the temperature parameter controlling exploration.

4) Network Architecture: The Deep Q-Network (DQN) used in our experiments is a **convolutional neural network (CNN)** designed to process a $(10, 10, 5)$ input observation space and output Q-values for n possible actions. It consists of:

- **Convolutional Feature Extractor**:
  - 4 convolutional layers with **ReLU activations**:
    - **Conv1**: $5 \rightarrow 32$ filters, 3×3 kernel, stride 1, padding 1
    - **Conv2**: $32 \rightarrow 64$ filters, 3×3 kernel, stride 1, padding 1
    - **Conv3**: $64 \rightarrow 64$ filters, 3×3 kernel, stride 1, padding 1
    - **Conv4**: $64 \rightarrow 128$ filters, 3×3 kernel, stride 1, padding 1
  - The output feature maps are **flattened** into a 1D vector.
- **Fully Connected Decision Module**:
  - **FC1**: input size $\rightarrow 1024$, ReLU
  - **FC2**: $1024 \rightarrow 2048$, ReLU
  - **FC3**: $2048 \rightarrow n$ (final output layer for Q-values)

5) Training metrics example: See Figure 6.

Fig. 6. From left to right: (a) Loss (b) Exploration rate decay (c) Winrate (d) Steps per episode



6) Decay Equations:

$$\epsilon(\text{episode}) = \epsilon_{\text{final}} + (\epsilon_{\text{start}} - \epsilon_{\text{final}}) \cdot e^{-\frac{\text{episode}}{\epsilon_{\text{decay}}}} \tag{8}$$

$$T(\text{episode}) = T_{\text{final}} + (T_{\text{start}} - T_{\text{final}}) \cdot e^{-\frac{\text{episode}}{T_{\text{decay}}}} \tag{9}$$

Strategy	Avg reward	Win rate
Playing VS Random		
Classic Replay Buffer	2.699	0.961%
Prioritized Experience Replay	**2.851**	**0.963%**
Playing against each other		
Classic Replay Buffer	-0.008	0.465%
Prioritized Experience Replay	**0.008**	**0.534%**

TABLE III
COMPARISON OF CLASSIC AND PRIORITIZED EXPERIENCE REPLAY.

7) Results for Buffers: See TABLE III.

8) Results for Training Setup Tests: See TABLE IV and Fig 11

Strategy	Avg reward	Win rate
Playing VS Random		
Illegal	1.654	0.927
Random	2.468	0.949
Baseline	**2.912**	**0.962%**
Playing VS Baseline		
Illegal	-0.222	0.415
Random	-0.052	0.450

TABLE IV
COMPARISON OF ILLEGAL PLAY, RANDOM PLAY VS BASELINE.

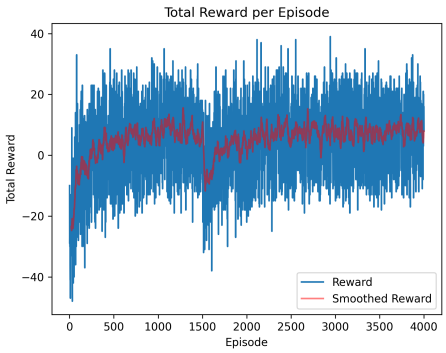


Fig. 7. Average Rewards Training VS Random then classic self-play

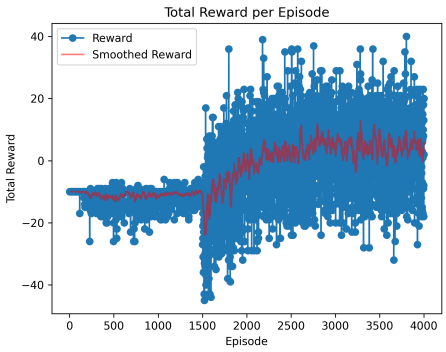


Fig. 8. Average Rewards Training VS Illegal Moves then classic self-play

B. PPO

PPO Update Equations

PPO policy θ and value function ϕ. One or multiple episodes $\mathcal{D}_k = \{\tau_i\}$ are added to a buffer by running the policy and then the PPO is updated at the end and goes through the trajectories, computing advantages $\hat{A}_t$ and discounted rewards $\hat{R}_t$ before updating θ (actor) and ϕ

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t), \; g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right) \tag{10}$$

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

Detailed PPO Results

Fig. 9. PPO Self Play Training with complete updates every 10 episodes

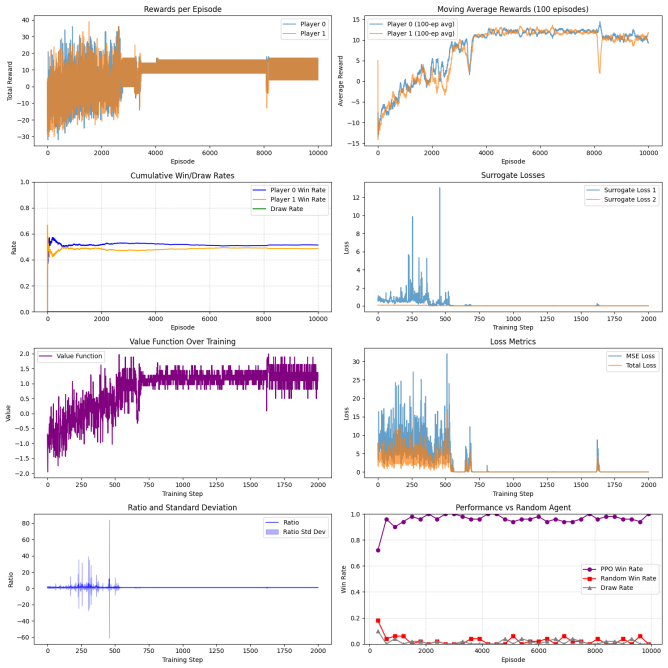


Fig. 10. PPO training results over 10 000 episodes of self-play and soft updates

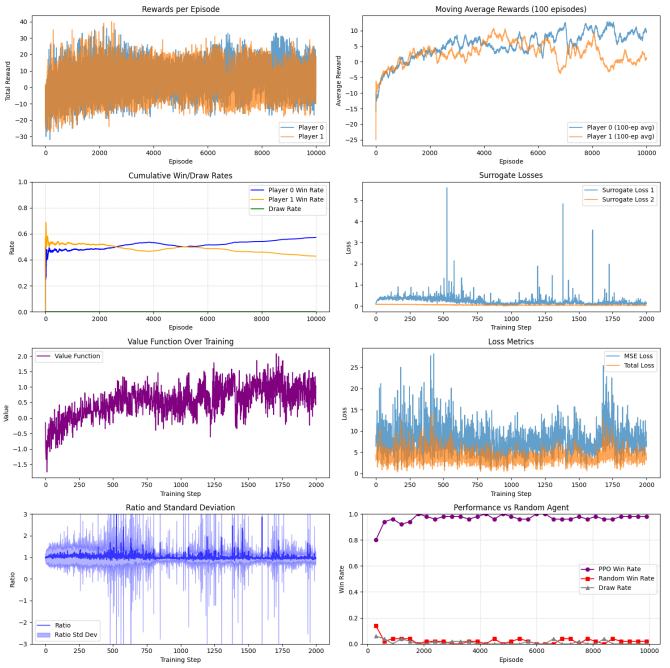


Fig. 12. PPO VAE Training over 10 000 episodes : updates every step

VAE

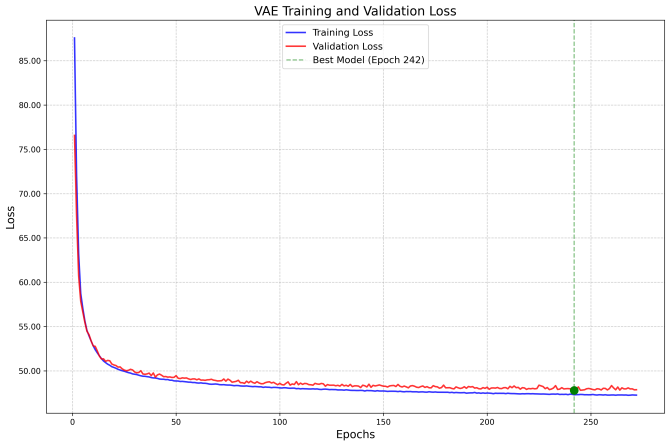


Fig. 11. VAE training and evaluation loss

Fig. 13. PPO VAE Training over 10 000 episodes : updates every step

C. Tree Methods

1) Monte-Carlo Tree Search: Monte-Carlo Tree Search (MCTS) is a decision tree exploration method widely used in game-playing applications, notably in Go. It constructs a search tree iteratively through random simulations (playouts), estimating move quality and guiding exploration toward promising areas.

MCTS consists of four main steps:

- **Selection**: A path is selected from the root node based on a tree policy.
- **Expansion**: If the leaf node is not terminal, new child nodes are added.
- **Simulation (Rollout)**: A random playout estimates the outcome from the new node.
- **Backpropagation**: The simulation results update the statistics of visited nodes.

A key variant, Upper Confidence bounds applied to Trees (UCT) [12], balances exploration and exploitation.

A deep convolutional network typically processes the current state s and outputs the **Policy Output** $P(s)$ (probability distribution over actions) and **Value Output** $V(s)$. Training involves optimizing the policy output via cross-entropy loss and the value output via mean squared error loss (see Appendix).

$$\mathcal{L}_{\text{policy}} = -\sum_{a} \pi_a \log P_a, \quad \mathcal{L}_{\text{value}} = (z - V(s))^2. \tag{11}$$

The final loss function is a sum of both components.

We were **unable to implement an effective training process** for the AlphaZero MCTS algorithm due to computational constraints. Despite using the 16-core computers available at École Polytechnique (Intel® Xeon®

W-1270P CPU @ 3.80GHz), and attempting parallelization, the action space of 1,753 proved to be excessively large (even compared to chess), making training infeasible. The program struggled to complete a single full simulation within an hour. Additionally, the lack of control over the computing resources at École Polytechnique—frequent shutdowns and shared usage with other students further hindered our ability to train the model efficiently.

2) Minimax with Alpha-Beta Pruning: The Minimax algorithm with Alpha-Beta Pruning is a baseline tree search method used to determine the best move in zero-sum games such as chess or go. It is based on a recursive exploration of the game tree, simulating the moves of both opponents: the maximizing player, who seeks to optimize the evaluation of the position, and the minimizing player, who attempts to reduce it.

The Alpha-Beta extension [13] optimizes Minimax by reducing the number of nodes evaluated. This process maintains two bounds: α, the best value guaranteed for the maximizing player, and β, the best value guaranteed for the minimizing player. If, during the exploration of a branch, it is determined that the potential value cannot improve the result for the current player (i.e., when $\alpha \geq \beta$), the branch is pruned, thus avoiding unnecessary computation.

The main steps of the algorithm are as follows:

1) **Recursive exploration:** The game tree is explored in depth by alternating between maximization and minimization phases.
2) **Updating bounds:** At each node, the values of α and β are updated based on the evaluations of the subtrees.
3) **Pruning:** If, at any node, α exceeds or equals β ($\alpha \geq \beta$), the exploration of that subtree is halted, as it cannot affect the final decision.
4) **Leaf evaluation:** Upon reaching a leaf node or a maximum depth, a heuristic evaluation function estimates the quality of the position.

In practice, the high dimensionality of the action space (approximately 1700 moves) makes exploring the move tree particularly slow even for shallow depths (2 or 3 moves ahead). To achieve reasonable computation speeds, the search depth is dynamically adjusted based on the number of legal moves available before beginning the search. In terms of performance, although it is challenging to run many simulations (e.g., simulating a game lasting 5 minutes), in a set of about a dozen simulations the Alpha-Beta algorithm consistently won against a player making random moves.