Quinto: Solving Quarto with Deep Reinforcement Learning

Computational Intelligence
Winter Semester, 2022-2023

Politecnico di Torino
Data Science and Engineering

Francesco Capuano, Matteo Matteotti
@fracapuano, @mttmtt31

Abstract

In this work, we present *Quinto*, a novel approach towards the resolution of the game of Quarto. Differently from other approaches that have been tried out in the past, we here focus on exclusively applying the advancements made by the research community on On-Policy Policy-Gradient Deep Reinforcement Learning, using different algorithms and experimental settings. In particular, we experimented with game-rules agnostic, action-masked and symmetry-aware agents.

As it was possible to expect, we here show that symmetries-aware action-masking agents outperform other agents, as they are clearly able to make a better usage of their much cleaner reward signal.

While in our experiments we did not witness a 100% win ratio, we here present a blazing-fast algorithm, able to learn how to win 90%+ games out the 1000 it is able to play in less than 1 minute on commercially available hardware. Our code is freely available at: github.com/fracapuano/Quinto.

1 Introduction

1.1 The game of Quarto

Quarto is a two players board game consisting in positioning 16 pieces on a 4x4 board. Each of the available pieces is characterized by four (non-mutually exclusive) binary attributes: size

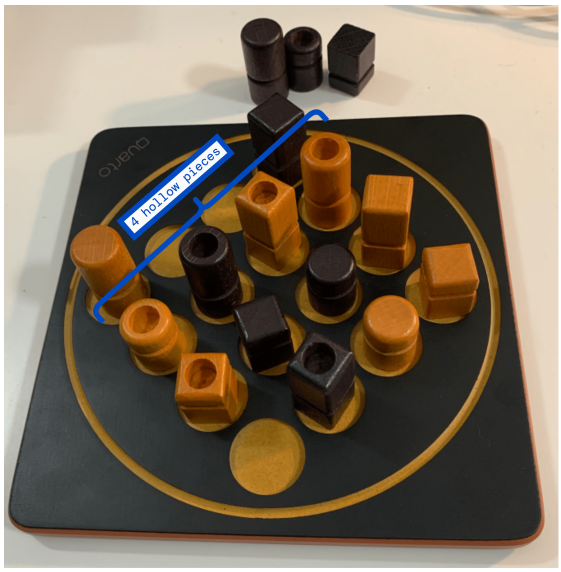


Figure 1: A Quarto board with a winning configuration (4 hollow pieces forming a line)

(large or small), color (white or black), shape (round or square), and type (hollow or solid). Players take turns, choosing a piece for their opponent to place on the board, while aiming at being the first player to get four pieces in a row that have at least one attribute in common.

In the context of game theory, Quarto is defined as a two-player zero-sum impartial game with perfect information where *zero-sum* means that one player's win results in the other player's loss, *impartial* means that each player shares the same set of common pieces and with *perfect information* means that both players are aware of the game's state and all possible moves at all times.

A winning configuration for the game of Quarto is presented in Figure 1.

1.2 Why Deep Reinforcement Learning

In the very vast set of board games, the game of Quarto clearly stands out for its relative simplicity (as opposed to the game of Chess, for instance). The presence of a large number of symmetries [3], for instance, would possibly allow reducing the number of board configurations agents based on min-max strategies would have to explore. In principle, one could even store the rewards associated with each action and then choose the perform the best one based on the current state, using decision-rule algorithms like Q-Learning.

However, the vast number of states [3] (16 pieces and 16 board positions amount to $16!^2 \simeq 10^{26}$ possible states when choosing the first move) makes unfeasible any approach as such, both in terms of inference time for min-max agents and storage of the actual Q-table. Our claim is that albeit symmetries do indeed reduce the total number of possible states, such a reduction can be considered negligible.

In a previous work [4], the authors implemented a min-max algorithm with alpha-beta pruning which scans the game tree up to a certain depth. Though their method obtained satisfactory results against human players, a mid-game move can take up to 30 seconds, thus seriously hindering the application of this approach in real-world matches.

For what concerns traditional RL, Q-tables are reportedly only a proxy of policies and they tend to introduce one more element between the Sequential Decision Making (SDM) process and the agent's interaction with the environment, [8].

Our choice of using Deep RL is therefore motivated by the aforementioned drawbacks on min-max and Q-learning-based approaches, but it is also inspired by our understanding of how *humans* do actually learn to play games. By leveraging increasingly-good intermediate rep-

resentations of the states they encounter and by letting these very representations drive their decision-making process, humans are able to learn how to perform very complex tasks (such as playing Chess or Quarto), without having to memorize para-exhaustive portions of the configurations tree spanning from the present state.

While this intuition about human learning is certainly not a novelty in the context of SDM problems, to the best of our knowledge it has never been applied to board games, and most certainly it has not been applied to Quarto.

To model the aforementioned intuition-action mechanism, we did make use of On-Policy Policy-Gradient algorithms such as *Actor-Critic* (mainly as an extension of [9]) and *PPO* ([7], mainly an extension of [6]).

2 Quarto as RL problem

Reinforcement Learning (RL) is the branch of Machine Learning devoted to investigating the prolonged-in-time interaction between programs and the environment in which these are executed.

In the framework of RL, programs are represented by means of an *agent* which perceives the environment in terms of its *state*. The interactions between the program and the environment are represented in terms of *actions* the agent performs on the environment, virtually as a consequence of having observed a particular state, and *rewards* the agent obtains as a consequence of both its condition and the state of the environment after an action has been performed.

When dealing with games, the definition of the action space becomes crucial - not only does the agent need to perform the best action, but the said action should also be legal according to game rules.

The problem of finding an optimal strategy to play Quarto can be formalised as the search of the best possible move given the currently available pieces and slots on the board.

Specifically, this problem can be turned in a Markov Decision Process (MDP) represented by:

- A state space $\mathcal{S}$, in which each element s_t is a vector in $\{0, 1, 2, \ldots, 16\}^{17}$ where the first 16 elements are a flat representation of the 4×4 board and the 17^{th} element is the piece that the player needs to put down. Integers in $[0, 15]$ represent unique pieces, whereas 16 represents an empty space on the board.

- An action space $\mathcal{A}$, in which each element a_t is a tuple where the first element is the position where the player wishes to place the piece and the second element is the piece chosen for the opponent's next move. At each step, the action must be legal. In other words, the piece should be placed in an *empty* slot and the piece for the opponent has to be chosen from the set of *available* pieces.

- A reward function $r : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto \mathbb{R}$. We designed the reward function to encourage our agent to win and, later on, to develop strategies to win as quickly as possible.

An illustration of how this paradigm is applied to the game of Quarto is presented in Figure 2.

3 Method

In this section, we introduce three possible solutions to Quarto, which can be divided into two different categories:

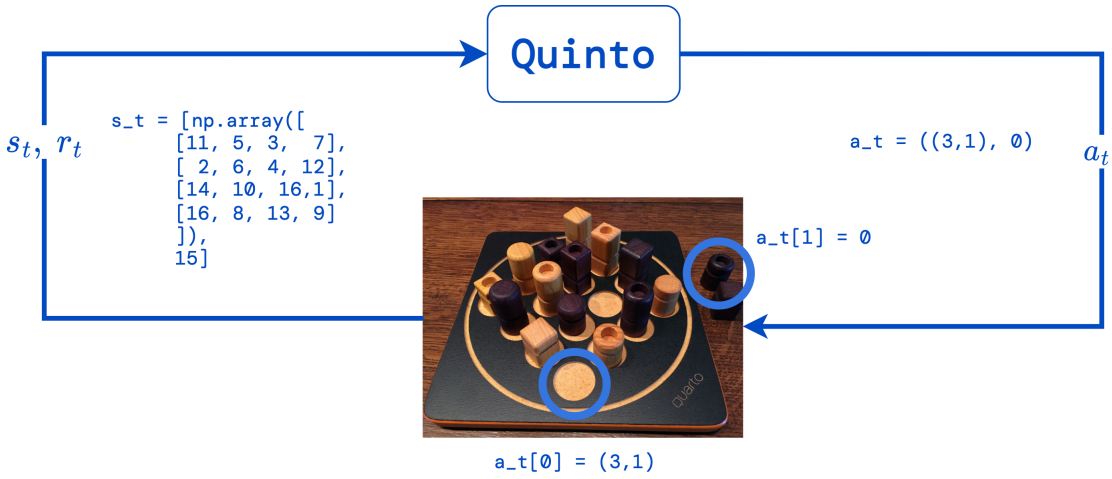


Figure 2: Quarto as an RL problem tackled by our agent, *Quinto*.

- *non-masked strategies*: algorithms where the agent does not explicitly know the concept of *legal* moves but has to learn it by playing.
- *masked strategies*: algorithms where the agent is aware of game rules and can therefore harness in full its experience to learn how to play *well* rather than simply playing *fair*.

All the techniques we did employ to solve this problem derive from the very same mathematical foundation on *Markov Decision Processes* (MDPs), which we here briefly recall.

3.1 MDPs

In principle, *Quinto* should be able to select actions to play based on the current state of the game, *i.e.* to learn a function $\mu : \mathcal{S} \mapsto \mathcal{A}$ such that:

$$a_t = \mu(s_t) \tag{1}$$

However, as argued in [8], it is rarely the case that one can use fully deterministic functions mapping states to actions. What is done in practice is to replace μ with a probability distribution over actions *conditioned* on the present state s_t.

Conceptually, this would allow one to be able to treat non-fully deterministic scenarios in which well-performing policies can only be defined up to a certain extent, as well as permitting a more deterministic scenario, in which simply enough $\pi(a_t|s_t) \simeq \delta^{(a_t)}$.

If one then defines $\pi : \mathcal{S} \times \mathcal{A} \mapsto [0, 1]$ as the policy adopted by an agent (for the sake of simplicity, from here on we will refer to the term agent and *Quinto* interchangeably), then one can define a *trajectory* in the State-Action space as the (possibly infinite) sequence:

$$\tau = s_0, a_0, s_1, a_1, \dots \tag{2}$$

Under the assumption that a_t is drawn from the policy $\pi(a_t|s_t)$, one can express the probability mass of said trajectory as:

$$\mathbb{P}_\pi(\tau) = \mathbb{P}_\pi(S(0) = s_0, A(0) = a_0, S(1) = s_1, A(1) = a_1, \dots) \tag{3}$$

Without loss of generality, let us assume that these trajectories consist of an arbitrarily large (but finite) number of transitions T. In light of the Markov property underlying the whole RL framework and simple conditioning arguments the probability mass of a given trajectory given a policy can, conveniently enough, be simplified to:

$$\mathbb{P}_\pi(\tau) = \mathbb{P}(s_0) \cdot \prod_{t=0}^{T-1} \mathbb{P}(s_{t+1}|s_t, a_t) \cdot \pi(a_t|s_t) \tag{4}$$

This equation simply describes that the probability mass of the whole trajectory can be reduced to a Markov chain in which actions are chosen with probability $\pi(a_t|s_t)$ and do modify the state s_t into s_{t+1} with probability $\mathbb{P}(s_{t+1}|s_t, a_t)$.

Considering that the reward function r is indeed defined on triplets of the form (s_t, a_t, s_{t+1}) it is natural to extend the notion of reward associated with a transition to the more general concept of a trajectory's return.

In particular, one can even define a discount factor $\gamma \in (0, 1]$ (typically $\gamma = 0.99$ or $\gamma = 0.999$) that allows defining the discounted return of a trajectory as:

$$R_t = \sum_{j=t}^{T} \gamma^t r_t \tag{5}$$

Clearly enough, $R_t = R_t(\tau)$ (since rewards functionally depend on transitions). The discount factor allows trajectories to be evaluated (in terms of the reward they yield) only up to a certain distance in time (that is, $\frac{1}{1-\gamma}$) from the timestep t considered. However, oftentimes in practice (and especially in the context of sparse rewards) $\gamma = 1$.

Clearly enough, a natural metric of performance for a given policy π is the expected return it generates, that is:

$$J(\pi) = \mathbb{E}_{\tau\sim\pi}\big[R(\tau)\big] \tag{6}$$

Once this has been done, it is not difficult to prove that a well-performing agent should aim at finding:

$$\pi^* = \arg\max_\pi J(\pi) \tag{7}$$

3.2 Policy Gradients

Considering that π is a probability distribution, it is not immediate to obtain a closed-form solution to the problem of maximizing J.

Nevertheless, one can easily solve this issue by parametrizing π with θ (for instance, one can have that the policy $\pi_\theta \sim \mathcal{N}(\theta, I)$ is a random variable distributed according to a multivariate normal distribution with mean vector θ).

The major possibility offered by policy's parametrization is, however, that one can make use of first (and, possibly, second) order information to iteratively optimize J. In particular, one may compute the gradient of J with respect to the parameter set θ.

In particular, it is possible to prove that:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \cdot R(\tau)\right] \tag{8}$$

Which, in principle, can be used to optimize the parameters of the policy via gradient ascent via the following relation:

$$\theta_k \leftarrow \theta_k + \alpha \nabla_\theta J(\pi_\theta)|_{\theta=\theta_k} \tag{9}$$

This approach would therefore optimize directly the policy parametrization in order to increase the probability of sampling actions that yielded a positive return and decrease the probability of sampling actions that yielded a negative return.

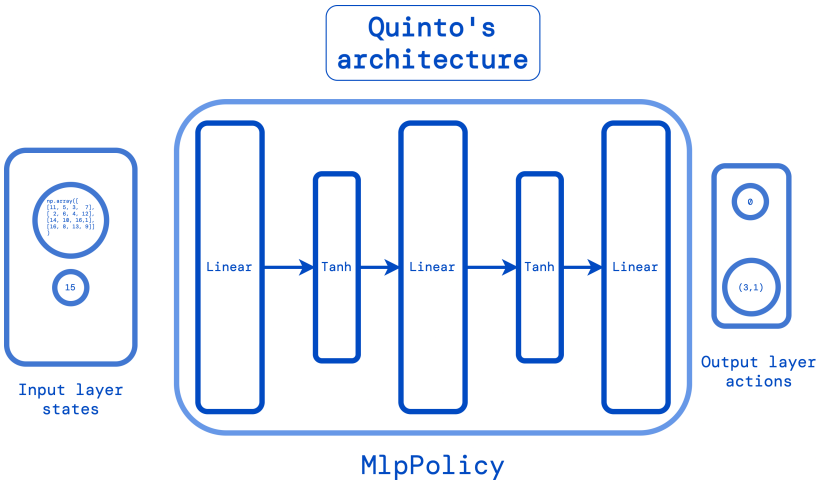


Figure 3: Quinto's architecture. Intermediate layers present 64 units each. The activation function used is Tanh. On the right, is the final output of the network.

In practice, policy parametrization is often performed using Deep Neural Networks (hence the *deep* nature of RL), whose weights are therefore tuned according to 9.

An example of the architecture used in Quinto can be found in Figure 3.

3.3 A2C

Various theoretical results allow using different formulations of the policy gradient presented in equation 8 that, while mathematically equivalent, drastically reduce the variance of the policy updates during training, which can be interpreted as how much the agent actually struggles in training.

One of these major improvements, theoretically justified by the *Expected Grad-Log-Prob Lemma*, states that an alternative (yet, once more, mathematically equivalent) formulation for equation 8 is:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \cdot A_{\pi_\theta}(s_t, a_t)\right] \tag{10}$$

In which $R(\tau)$ has been replaced by the Advantage function, whose expression follows by simply applying the definition of the State-Action Value function and State Value function:

$$A_{\pi_\theta}(s_t, a_t) = Q_{\pi_\theta}(s_t, a_t) - V_{\pi_\theta}(s_t) = \mathbb{E}_{\tau\sim\pi_\theta}\big[R(\tau)|s_0 = s_t, a_0 = a_t\big] - \mathbb{E}_{\tau\sim\pi_\theta}\big[R(\tau)|s_0 = s_t\big] \tag{11}$$

This quantity represents the *advantage* associated with choosing the action a_t when in state s_t and, conceptually, represents how better off the option of choosing a_t is with respect to any other viable action in s_t.

The fact that one considers this term instead of the return over the trajectory $R(\tau)$ clearly triggers a process in which actions-probabilities are increased only when they are really better with respect to others.

Here, $\mathbb{E}_{\tau\sim\pi_\theta}[\cdot]$ is the expected value of a given quantity over trajectories drawn from the same policy. Usually, one refers to this type of expectation as expectations over *roll-outs*. In practice, these expectations are often approximated by Monte Carlo averages.

Of course, the advantage has no clear functional form and must therefore be estimated in order for the policy parameters to be changed accordingly to this new paradigm. This is the main characteristic of Actor-Critic algorithms. As a so-called *actor network* outputs actions, a parallel (partially weight-sharing) network is trained at the task of approximating this advantage function

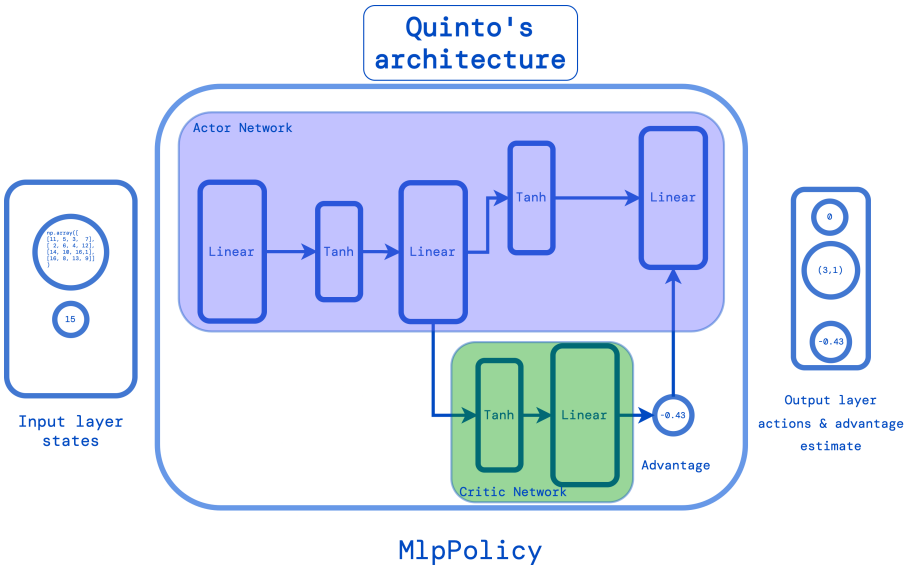


Figure 4: An Actor-Critic Policy Network. As it is possible to see, Actor and Critic share a significant portion of their weights. This is done for the sake of enforcing the exploitation of common features of the state during the process of choosing what action to perform and evaluating the given configuration.

to bootstrap the action prediction with the knowledge that derives from being able to precisely estimate the expected reward from a certain configuration when choosing certain actions.

An illustration of this type of Actor-Critic Policy Network is presented in Figure 4.

3.4 PPO

Proximal Policy Optimization ([7]) follows Shulman's ground-breaking work in designing and applying trust regions to policy optimization ([6]).

The main intuition underlying the very wide adoption (and success) in the RL-community of Trust-Region-based methods (such as *TRPO* and, consequently, *PPO*) is that the configuration space $\mathcal{S} \times \mathcal{A}$ can be so large (in continuous problem, may even be infinite), complex that $J(\pi)$ is very seldom Lipschitz continuous, thus implying that incautious policy updates can seriously hinder the possibility of the agent learning well-performing behaviors.

In this sense, Trust-Region-based approaches tackle the problem of optimizing π in a more conservative way. For the sake of brevity, let us only give a conceptual overview of said approaches.

Considering that updates of the policy are based on equation 10, it is clear that the optimization route on the policy parameters clearly depends on the (possibly very noisy) estimate of the advantage function. Since the advantage function clearly is approximated correctly only up to a certain extent, it is clear that one has to embed a mechanism of conservative updates to avoid imprecise and incorrect estimates of this (crucially important) quantity to drive completely off the policy parametrization, both at the beginning and during the training process.

Interestingly, this intuition has proven to be correct to the extent that algorithms such as TRPO (and its "simpler" derivation, PPO) are now the state of the art for what concerns On Policy Policy-Gradient based algorithms.

3.5 Masked-PPO

Despite the fact that both A2C and PPO were initially designed for the more general case of continuous state and action spaces problems, these algorithms have been proven to be very

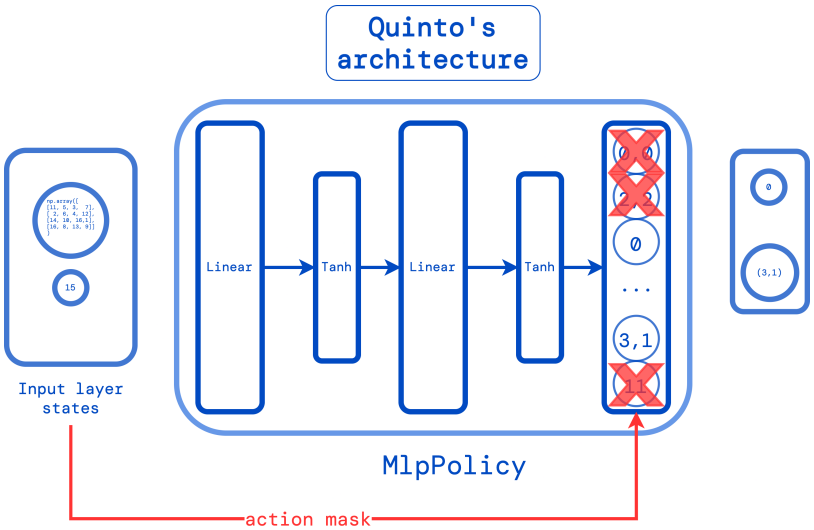


Figure 5: Quinto's architecture when performing action masking. The knowledge of game rules is used to mask out the probability of selecting unfeasible actions, thus allowing the agent to focus on the task of learning how to play good moves rather than feasible ones. Action masking is performed by zeroing infeasible actions' probabilities and then re-normalizing the output's layer probability.

effective also in solving discrete problems.

A peculiarity of discrete problems (a particular instance of these are board games such as Quarto) is that the whole actions space $\mathcal{A}$ is seldom available to the agent. Way more often, to obey game rules, agents can only play actions in a smaller subset of $\mathcal{A}$, say $\mathcal{A}_{\text{legal}}$.

Clearly enough, this $\mathcal{A}_{\text{legal}}$ depends on the state the agent encounters at a given timestep, i.e.:

$$\mathcal{A}_{\text{legal}} = \mathcal{A}_{\text{legal}}(s_t) \subseteq \mathcal{A} \quad \forall s_t \in \mathcal{S}$$

In this case, we say that $a \in \mathcal{A} \setminus \mathcal{A}_{\text{legal}}$ are *masked* out, in the sense that a fairly playing agent clearly follows a policy π_θ such that:

$$\pi_\theta(\tilde{a}_t|s_t) = 0 \quad \forall \tilde{a}_t \in \mathcal{A} \setminus \mathcal{A}_{\text{legal}(s_t)} \tag{12}$$

In all those cases in which $\mathcal{A}_{\text{legal}(s_t)}$ cannot be expressed *a priori* as a function of s_t, π_θ can be iteratively influenced to eventually satisfy equation 12 with largely negative rewards associated to all actions in $\mathcal{A} \setminus A_{\text{legal}}$.

Despite being very practical (as one would simply have to be able to conclude whether or not an action is legal) this approach is however very much sub-optimal. Agents learn best (especially in sparse-reward scenarios) when the reward signal is as clean as possible. Using a reward function that jointly discourages both bad *and* unfeasible actions can make the learning process more complicated, as the agent would have to learn, from the same reward signal, how to play fair and well, at the same time.

Instead, using Action Masking is a more efficient and effective approach than using large negative rewards mechanisms. Since action masking limits the actions that an agent can take, this allows the agent to focus on the most important actions that lead to positive rewards. This reduces the complexity of the policy-optimization problem and allows the agent to learn more quickly and effectively from the experience it collects. Overall, action masking provides a more intuitive and controlled approach to shaping an agent's behavior, making it a better option than negative reward mechanisms.

An example of action-masked Policy Networks is presented in Figure 5.

By zeroing the probability of the unfeasible action, the policy network is updated only for what concerns the actions that are not masked out, i.e.:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \cdot A_{\pi_\theta}(s_t, a_t)\right] = 0 \iff \forall a_t \notin \mathcal{A}_{\text{legal}} \tag{13}$$

Notice how $a_t \notin A_{\text{legal}}$ is only a sufficient condition for $\nabla_\theta J(\pi_\theta)$ to be zero and the necessity of this condition does not generally hold.

4 Experimental setting

To carry out our experiments, we mainly used Stable-Baselines3 [5], the state-of-the-art open-source Pytorch-backed RL library. To monitor our experiments we instead made great usage of Weights & Biases [1].

We now give an overview of our experimental setting.

4.1 Environments

We experimented with four different environments, with different configurations of reward functions, adoption of symmetries-aware decision making, and opponent challenged during training.

Reward function r

The reward function r represents the reward obtained for winning, drawing, and losing a training game. In addition to that, each environment sets the reward associated with an invalid move to -200. This penalty only applies to non-masked algorithms and, as aforementioned, is the only practical way to teach these agents the game rules.

The sole difference between the *v0* and *v1* environments is the penalty associated with a loss, respectively set to 0 and -1 (*i.e.*, the opposite of the reward for a victory).

In *v0* and *v1* environments, we associated the same reward to *every* victory, as winning in two moves or winning in ten moves was rewarded exactly the same. However, as the game evolves, the number of available moves decreases for a fair randomly-playing opponent. As a consequence, the chances of said opponent picking a winning move increase.

This brought us to come up with the intuition that an agent should be incentivized not only to win but also to do it in the minimum number of moves - in Quarto, a victory can be declared after four total moves (two for each player), that is, four pieces on the board.

This kind of behavior is incentivized in the *v2* and *v3* environment - a two-move victory is rewarded 5, whereas every additional move leads to a decrease of 0.75 in the reward. An eight-move victory is then rewarded 0.5 (exactly like a draw).

Symmetries

Quarto comes with several symmetries which can considerably improve the exploration capacity of our agent whilst training. As a matter of fact, playing one game is like simultaneously playing many different games, each of them on a symmetric version of the board, when said symmetries are exploited by the agent. As presented in [3], the possible symmetries are the following:

- **Reflection Symmetry**: The game board is symmetrical with respect to a vertical axis that runs through the center of the board. This means that the positions of the pieces on the left side of the board are reflected on the right side, and vice versa.

- **Rotation Symmetry**: The game board is symmetrical with respect to rotations of 90, 180, and 270 degrees. This means that the positions of the pieces on the board can be rotated to obtain different but equivalent positions.
- **Piece Symmetry**: The pieces in Quarto are symmetrical, meaning that one can consider four equivalent games in which one *negates* a different property every time (*e.g.*, you turn every light piece into dark, and vice-versa).

Our *v3* environment makes use of rotational symmetries only during training, resulting in better performances. The choice of only implementing rotational symmetries was made to reduce the duration of the board-sensing phase.

Opponent

Training an agent against a random opponent (which only performs valid actions) yielded satisfactory results. Ideally, as our agent gets better, it needs to face opponents which are roughly at its ability level. This is a key feature in basically every video game, where pretty much all matchmaking systems are ranking-based [2].

To mimic such a dynamic, and inspired by the results presented in AlphaGo and AlphaZero, we introduced *self-play* inside the *v3* environment: our agent is periodically saved so that its opponent can be periodically updated to be its most recent version.

Our best-tested configuration was the one where we initially let the agent play against a random opponent (to avoid *cold start*), and then incrementally trained using self-play for the remaining timesteps.

A comprehensive view of our experimental setting is presented in Table 1.

Version	Reward function	Use symmetries	Opponent
v0	$r_T = (+1, +0.2, 0)$	No	Random opponent
v1	$r_T = (+1, +0.2, -1)$	No	Random opponent
v2	$r_T = (5 - 0.75 \cdot (T/2 - 2), +0.5, -1)$	No	Random opponent
v3	$r_T = (5 - 0.75 \cdot (T/2 - 2), +0.5, -1)$	Rotations only	Self-play

Table 1: The tested environments. Each version has its own specification of the reward function, of the symmetries, and of the opponent faced during training.

4.2 Agents

We used different implementations of the different algorithms previously introduced to train our agent. Each algorithm was trained in different environments.

Each agent is based on a specific algorithm (according to Section 3). Each agent was trained through a different number of timesteps. The training details are summarised in Table 2.

Our best-performing model is initially trained against a random opponent for 100e6 timesteps, and then incrementally trained against itself (exploiting symmetries) for further 20e6 timesteps.

5 Results

To guarantee a fair comparison, we here compare only the performances of the agents trained in the same environment.

Algorithm	Environment	Timesteps	Training time
PPO	*v0*	5e6	6h 29m
PPO	*v1*	5e6	4h 34m
A2C	*v0*	5e6	5h 43m
A2C	*v1*	5e6	4h 52m
MaskedPPO	*v0*	5e6	6h 27m
MaskedPPO	*v1*	5e6	7h 28m
MaskedPPO	*v2*	100e6	∼1 week
MaskedPPO	***v2* + *v3***	**100e6 + 20e6**	**∼1 week + 27h**

Table 2: Recap of the agents' training. In boldface is our best-performing model, which was first trained against a random opponent for 100e6 timesteps (inside the v2 environment) and then incrementally trained against itself and symmetries for further 20e6 timesteps (inside the v3 environment).

Environment *v0*

Agents PPO- or A2C-based go through an initial burn-in period where they need to learn which moves are valid and which are not.

This can be clearly seen in Figure 6, where pretty much every game in the first part of the training process ends due to an illegal move. As time goes by, the portion of games ending due to illegal moves decreases, and the win ratio increases (as an effect of the positive reward associated with it). Though the decrease is significant, the proportion of games ending due to an invalid move is still remarkably above zero in the 5e6 training timesteps considered. Figure **??** also shows how PPO generally outperforms A2C in learning the game rules, and as a consequence, the win ratio is usually higher.

On the other hand, agents based on MaskedPPO have no burn-in period, as they always make a legal move. As a consequence, this type of agent significantly outperforms the non-masked counterparts throughout the entire training process, since it can focus on learning how to play instead of learning how to play *fairly*.

Environment *v1*

In the *v0* environment, the only penalty our agent received was in the case of an illegal move. No reward was attributed in the case of a *legal* loss.

In the *v1* environment, instead, we decided to penalize *legal* losses, giving them the opposite value of a win (*i.e.*, -1).

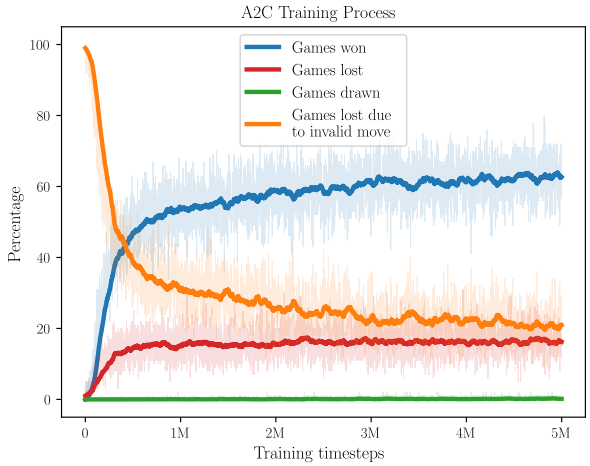
The results are reported in Figure 7 and clearly show that this modification did not bring a significant improvement in the 5e6 timesteps considered.

Environment *v2*

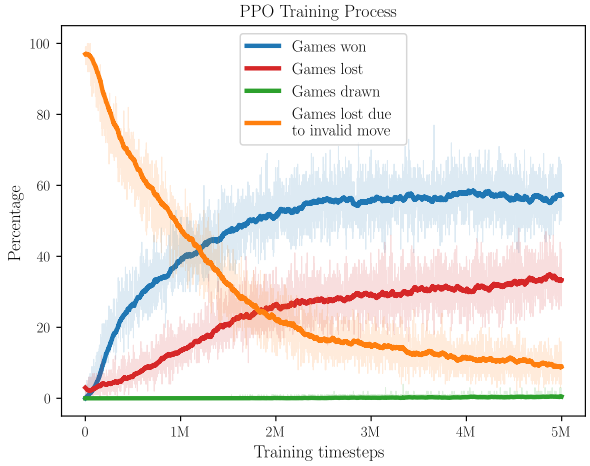
This environment (tested only using MaskedPPO, in light of the previously introduced arguments on the effects of action masking in the training process) incentivized agents to win in the minimum number of moves, using the reward function presented in Table 1.

Differently from what we originally thought, this (relatively important) modification did not result in *significantly* better performances, as shown in Figure 8.

Though this reward choice is the most reasonable out of the ones we tried, the reason behind these results was due to the agent's opponent: playing against a random opponent, the learning phase saturated relatively early, and our agent needed a better opponent to improve.



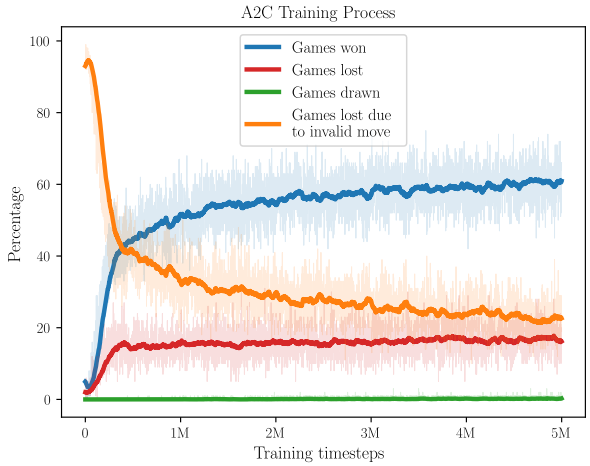
(a) Evolution of the Percentage of won, lost and drawn games. A2C in *v0* environment training for 5e6 timesteps.



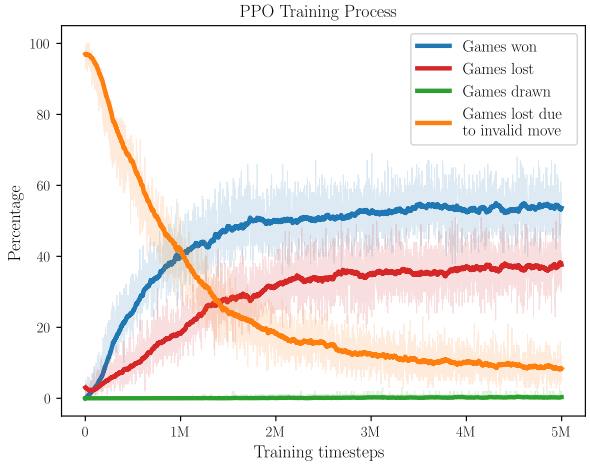
(b) Evolution of the Percentage of won, lost and drawn games. PPO in *v0* environment training for 5e6 timesteps.

(c) Evolution of the Percentage of won, lost, and drawn games. MaskedPPO in *v0* environment training for 5e6 timesteps.

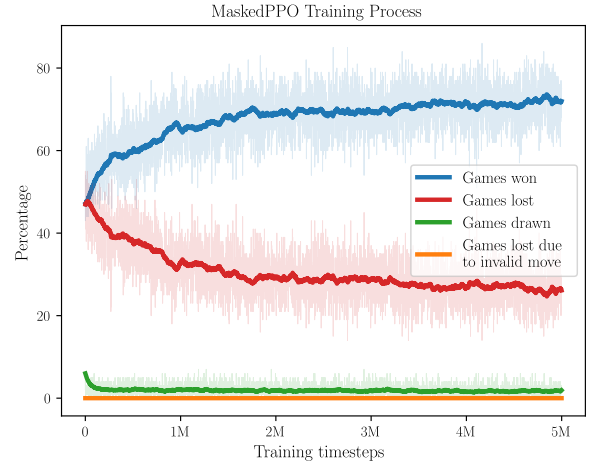
Figure 6: Evolution of games won against opponents over 100 games.



(a) Evolution of the Percentage of won, lost and drawn games. A2C in *v1* environment training for 5e6 timesteps.



(b) Evolution of the Percentage of won, lost and drawn games. PPO in *v1* environment training for 5e6 timesteps.



(c) Evolution of the Percentage of won, lost and drawn games. MaskedPPO in *v1* environment training for 5e6 timesteps.

Figure 7: Evolution of games won against opponents over 100 games.

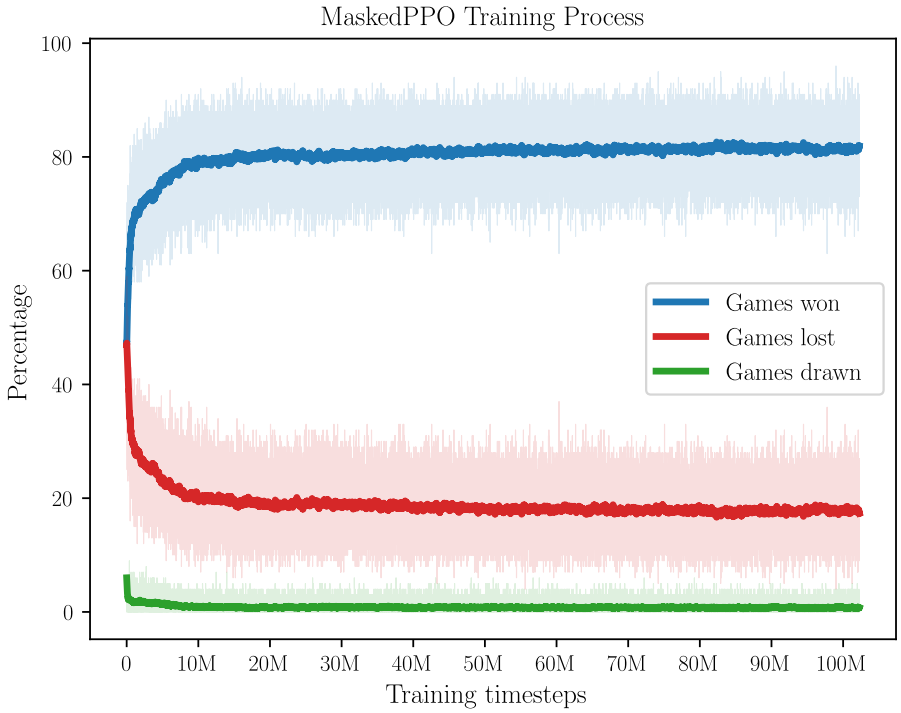


Figure 8: MaskedPPO training process over 100M timesteps

As a matter of fact, all agents trained against a random opponent reach a point where they hardly improve. As a consequence, the win ratio levels off, usually between 50% (in the case of non-masked agents) and 80% (in the case of masked agents).

This is what motivated us in creating a new environment, endowed with self-play and *v2*'s reward.

Environment *v3*

Similarly to *v2*, this environment was tested using MaskedPPO only. Specifically, instead of retraining an agent from scratch, we decided to fine-train the agent previously trained through 100e6 timesteps in *v2* using self-play. As a future extension, we plan to train an agent completely using self-play.

As shown in Figure 9, whenever the opponent is refreshed with a more recent checkpoint of the agent, the win ratio drastically drops, before (more rapidly as the agent is trained more and more) going up again.

This demonstrates the validity of our point: a random opponent does offer a good warm-up for our agent, but after a certain point effective learning is possible only when the agent is let play against itself.

With this combination of reward and opponent, our agent is able to achieve a win ratio above 90% against a random opponent and we believe that, also in light of the findings of AlphaGo and AlphaZero, self-play might be key for super-human performance.

6 Conclusions and Future Extension

To the best of our knowledge, this is the first time Deep Reinforcement Learning is applied to the game of Quarto.

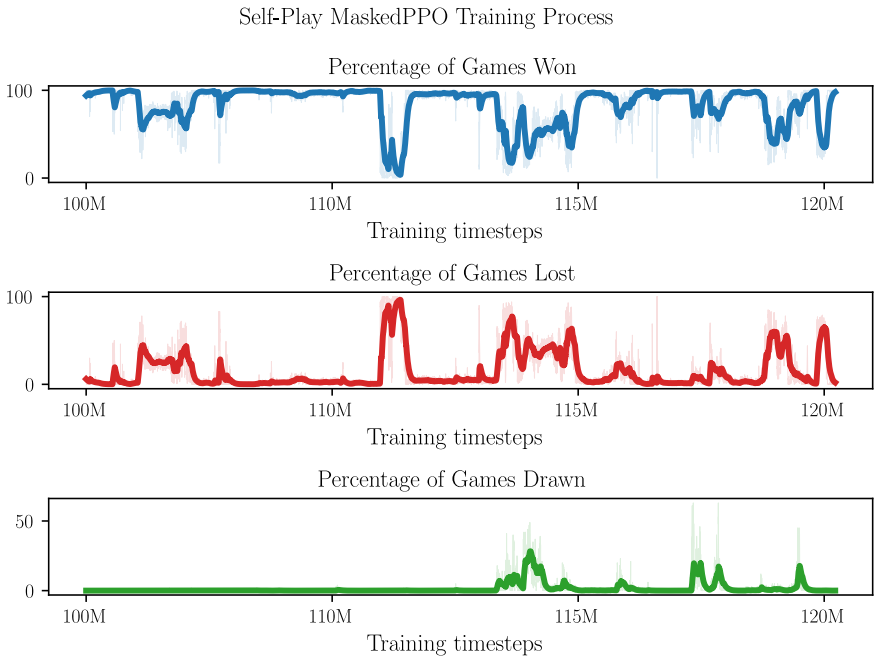


Figure 9: Fine-training of an agent trained with MaskedPPO over 100M timesteps for 20M episodes more. On the y-axis is reported the percentage of games won, lost, and drawn over 100 test games. In keeping with our expectations, it is possible to see how in presence of a change in opponent, the win-ratio drastically drops, before starting to grow once again

Though a real assessment of our agent is not possible as it never interacted with top-notch Quarto players, we believe that a peak of 90%+ victories over 100 games is an extremely satisfactory and promising result.

What is truly groundbreaking, however, is the execution time. All our agents can play more than a thousand games in less than one minute.

In future work, we plan to investigate the implementation of a masking strategy to A2C and the impact of different clipping range values on the performance of both PPO and MaskablePPO. The clipping range is a crucial hyper-parameter for PPO-based algorithms and it essentially controls the magnitude of the gradient update. Its purpose is to balance the trade-off between exploitation and exploration in the policy optimization process, as well as trading off conservative updates of the policy with very promising advantages estimates. We expect that a fine-tuned clipping range will lead to improved convergence and stability of the PPO algorithm and, ultimately, to better overall performance.

References

[1] Lukas Biewald. Experiment tracking with weights and biases, 2020. Software available from wandb.com.

[2] Thore Graepel and Ralf Herbrich. Ranking and matchmaking-how to rate players' skills for fun and competitive gaming. *Game Developer Magazine*, 2010.

[3] Wouter M. Koolen. Quarto!

[4] Jochen Mohrmann, Michael Neumann, and David Suendermann. An artificial intelligence for the board game 'quarto!' in java. pages 141–146, 09 2013.

[5] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021.

[6] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897. PMLR, 2015.

[7] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[8] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[9] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Reinforcement learning*, pages 5–32, 1992.