Mastering the game of Corso without human knowledge

Francesco Mistri[a;*]

Abstract. Reinforcement learning (RL), specifically deep reinforcement learning (DRL) has been successfully applied to solve a variety of problems, even setting world records in complex competitive games like Go. Similarly to Go, the game of Corso is a multiplayer zero-sum perfect information game. As such, it provides a challenging environment for deep reinforcement learning due to the delayed rewards and the intrinsic non-stationarity of multiplayer games. This work shows how DRL aided by tree search (AlphaZero) can be used in a self-play setup to obtain an agent effectively playing the game of Corso, given minimal human knowledge. Experiments are carried out on a 5x5 grid, which is the standard format for the game, and show that a superhuman policy can be obtained with reasonable computational time.

1 Introduction

Reinforcement learning has recently seen an explosion in popularity thanks to its promising results in many fields including combinatorial optimization [6] [7], chatbots [5] and complex comptetitive board [8] [10] [9] [1] and video [3] [13] games. Such problems can be intuitively seen as optimization ones (e.g. maximizing the win rate/probability in a game) and can naturally benefit from the self-improving nature of reinforcement learning, as in most cases the optimal solution is not known and computing it directly is unfeasible due to the lack of resources. Focusing on board games, a common implied complication is the search space, as a larger search space will naturally require more computational time in order to be explored and consequently to obtain good results. Additional challenges of these environments are delayed rewards and non-stationarity.

While the goal of reinforcement learning is always maximization of long term utility, extremely delayed rewards will provide a noisy signal, possibly making the learning process considerably harder. This is typical in board games, where the natural reward signal is the win/loss of a game. Intuitively, a win or loss is not the consequence of absolute winning/losing moves but, depending on the skill of the players, of some winning and some losing moves. In fact, a player could be mathematically winning for the entire game and lose in only one last move.

Moreover, they are usually multiplayer. In multiplayer games, the learning agent may face opponents of different strengths i.e. having different behaviours (policies). This usually implies some form of non-stationarity, based on the design of the algorithm employed for opponent selection.

This work focuses on the development of a game playing agent for the game of Corso, a multiplayer zero-sum perfect information game.

* Email: francesco.mistri2@studio.unibo.it.

The game (better described in Preliminaries) can be played by an arbitrary number of players and on an arbitrary rectangular grid. The standard format for the game is two-player with a 5x5 grid.

Recent successful works on board games inject some form of explicit planning into the algorithm, usually represented by some variation of Monte Carlo Tree Search (MCTS) [4] [8] [10] [9] [1]. This work is inspired particularly by [10] and [9], replicaiting the AlphaZero design to solve the game of Corso. All the relevant code can be found on GitHub[1].

2 Preliminaries

The problem of playing a game can be defined as a (finite) Markov Decision Process (MDP). Notation is borrowed from [11], but it is here partially quoted. Given a set of states $\mathcal{S}$, a set of actions $\mathcal{A}$ and a set of rewards $\mathcal{R} \subset \mathbb{R}$, and agent interacting with the environment will, at each timestep t, receive an state representation $S_t \in \mathcal{S}$ and consequently perform and action $A_t \in \mathcal{A}$ to obtain a reward $R_{t+1} \in \mathcal{R}$ and transitioning to a new state $S_{t+1} \in \mathcal{S}$ with a certain probability. The probability of transitioning to a certain state and receiving a certain reward is defiend as:

$$Pr\{S_t = s', R_t = r | S_{t-1} = s, A_{t-1} = a\} \doteq p(s', r|s, a) \quad (1)$$

The cumulative reward for a trajectory is defined as:

$$G_t \doteq \sum_{k}^{\infty} \gamma^k R_{t+k+1} \quad (2)$$

An agent is usually depicted by its policy π, that is, its mapping between input states and probabilities to take each possible action, i.e. $\pi(a|s)$ is the probability of taking action a if the current state is s. Given a policy, its state value function is defined as:

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t | S_t = s] \quad (3)$$

The goal is usually to find an optimal policy π^*, that is, one which acts maximizing the expected reward for each state:

$$v^*(s) \doteq v_{\pi^*}(s) \doteq \max_\pi v_\pi(s), \quad \forall s \in \mathcal{S} \quad (4)$$

v^* is the optimal (state) value function.

Board games and stationarity this definition of a MDP is inherently stationary, meaning that if the agent is going to learn by playing on a predefined MDP environment, its dynamics are going to be

[1] https://github.com/Ball-Man/corso

unchanged from episode to episode, so that the algorithm will eventually “figure it out”. In a board game scenario it seems reasonable to consider the opponent as being completely part of the environment, effectively morphing the multiagent RL problem into a single agent one. However, to retain generality and obtain an agent that can face any player it is necessary to define a general enough pool of opponents and/or to employ not only a form of improvement over time of the agent, but of its opponents as well. A popular technique in this case is self-play [8] [10] [9] [1], which comes in a variety of forms and approaches. Results obtained by related works show how effective these techiniques can be, even achieving super-human performance in complex games. Nonetheless, it provides an additional challenge for RL algorithms as stationarity of the MDP is a common theoretical assumption.

2.1 *AlphaZero and AlphaGo Zero*

Notation is adapted from [9]. AlphaZero[9] and AlphaGo Zero[10] have very similar designs. Here, the most general AlphaZero algorithm is described, highlighting differences with its predecessor AlphaGo Zero.

The AlphaZero algorithm is a hybrid approach between DRL and MCTS[4], with the goal to learn a (near) optimal policy for the given game. The core of the algorithm is a function approximation:

$$(\tilde{v}, \tilde{\pi}) = f_\theta(s) \tag{5}$$

Where s is a game state, $\tilde{v}$ is an approximation of the value function, $\tilde{\pi}$ an approximation of the policy and f_θ is the approximator, parameterized on θ. Such approximator is assumed to be a neural network. In AlphaZero, the network is used to generate train data, and simultaneously being trained on it, while AlphaGo Zero divides the process into iterations: each iteration consists in an initial generation of train data and a subsequent train session.

Data generation makes use of the network, aided by a variation of MCTS. The MCTS is used to perform some lookahead upon the current state s, obaining an enhanced policy:

$$\pi' = \text{MCTS}_{f_\theta}(s) \tag{6}$$

In particular, the variations consist in:

- No simulation phase: the simulation phase, generally a random traversal of the game tree from a leaf node to a terminal state, is replaced with network estimation $\tilde{v}(s)$.
- Selection algorithm: MCTS commonly uses some adaptation of the Upper Confidence Bound formula[2] (UCB) to seek balance between exploration and exploitation. That is, at each step t, the selected move in the tree search is:

  $$a_t = \underset{a}{\text{argmax}}(Q(s_t, a) + U(s_t, a)) \tag{7}$$

  Where $Q(s, a)$ is the Q-value (average state-action score) computed by the MCTS throughout multiple playouts, and $U(s, a)$ is the UCB variation.
  AlphaZero employs a variant which takes into account the predictions of the network, called PUCT:

  $$U(s, a) = c_{\text{puct}} P(s, a) \frac{\sqrt{\sum_b N(s, b)}}{1 + N(s, a)} \tag{8}$$

  Where c_{puct} is an arbitrary constant (encourages/discourages exploration), $P(s, a)$ is the estimated probability of winning for the state-action pair s, a, $N(s, a)$ is the total number of visits to the state-action pair s, a (total number of playouts which selected that action so far).
  $P(s, a)$ can be obtained through the network ($\tilde{\pi}(s)$).
- Policy construction: after a certain amount of playouts, typically the best scoring move is selected to be played. However, AlphaZero requires a probability distribution over the possible actions. This can be obtained for example by normalizing the Q values of the actions. In Alpha(Go) Zero, the policy is obtained by normalizing *node visits* instead:

  $$\pi'(s) = \frac{N(s)^{1/\tau}}{\sum_a N(s, a)^{1/\tau}} \tag{9}$$

  τ is an exploration parameter which alters the entropy of the distribution. For $\tau \approx 0$ the distribution is almost deterministic i.e. only the most promising move(s) are played. For $\tau = 1$ each action's probability is proportional to its visit count in the MonteCarlo playouts. This is mostly used during data generation to promote exploration.
  As of why using visit counts instead of action values directly, the original AlphaGo paper[8] briefly justifies this as being less sensitive to outliers.

These variations shall ensure that applying the search improves the plain policy given by the network, so that the improved policies can be used to train the network, enhancing its predictions. By repeating this process, the network can get better at each iteration.

In practice, data is generated through several games of self-play, where the MCTS enchanced agent plays against itself. At each ply the agent runs a fixed amount of tree search playouts, obtaining the improved policy π'. The played move is sampled from this policy.

Enhanced policies π', along with true game outcomes[2] v' for all episodes in the iteration are collected and finally used to update the network for a given number of steps. The network is updated through Stochastic Gradient Descent (SGD) in a supervised learning fashion, with the loss function:

$$l = (v' - \tilde{v})^2 - (\pi')^{\mathrm{T}} \log \tilde{\pi} + c||\theta||^2 \tag{10}$$

Where $c||\theta||^2$ is an L2 regularization term (hence c an arbitrary regularization constant). In pratice, the value head of the network uses the squared error as loss while the policy head uses crossentropy.

The algorithm can also be seen as a form of iterated imitation learning. This point of view was formalized in an algorithm called Expert Iteration[1] (ExIt). ExIt is a parallel, independent work from AlphaZero, although of simiar design.

2.2 *The game of Corso*

Corso is a multiplayer zero-sum perfect information game created by Francesco Mistri (also the main author of this work). The game can be played on a rectangular grid of arbitrary size and by an arbitrary amount of players. Natural setups include two players and a square grid, generally of size 3x3 (game tree similar to tic-tac-toe) or 5x5 (standard board). See Figure 4 in Appendix for an example game between a human and the neural agent. Here a brief explanation of the rules for a two player game.

[2] The value for the game outcome changes based on the game taken in consideration. In AlphaGo Zero, it is 1 if the winner is player one, -1 if it is player two, as in Go it is not possible to draw.

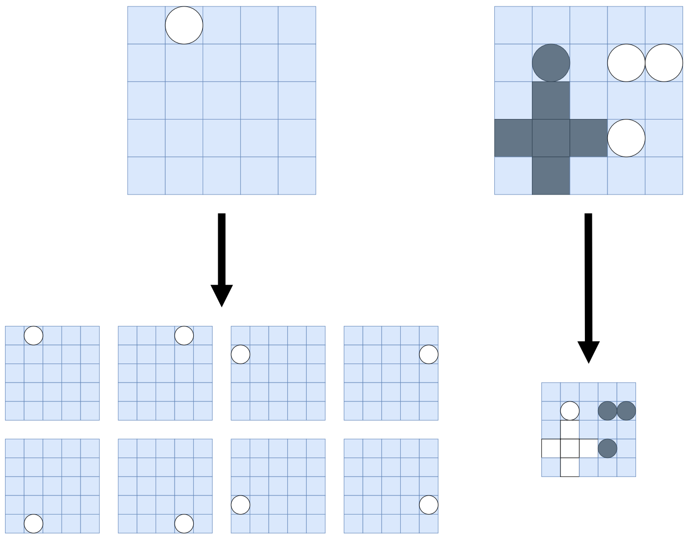


Figure 1: Symmetries exploited for data augmentation: rotations, reflections and color inversion.

Rules Each player has one action per turn. The action can: be placing a marble on an open cell or expanding an existing marble. It is not possible to expand the opponent's marbles. The goal of the game is controlling the highest number of cells.

Expansion of a marble results in the selected marble being lost. The cell containing the marble and adjacent cells (not diagonal) are dyed by the player's color. It is not possible to place marbles on dyed cells and they cannot be expanded anymore. Eventually they can be overridden by the opponent with nearby expansions.

Expansion leads to chain reactions with nearby marbles (not diagonal). A chain reaction will convert the entire result of the expansion into the color of the player causing it.

When all cells are occupied (either dyed or occupied by unexpanded marbles) the game ends immediately. A player's score is given by their number of controlled cells i.e. the total number of dyed cells and unexpanded marbles of their color currently on the board. The player with the highest score wins the game.

Theoretical outcome 3x3 is a solved version of the game, which is a guaranteed win for the first player. 2x2 is trivially a win for the second player. 5x5 theoretical outcome is still unknown, nonetheless, all experiments to this day suggest that the first player has the advantage. However, this may be due to the Monte Carlo nature of such experiments i.e. there is a larger number of winning trajectories for the first player in the game tree, but second player could be the theoretical winner through a limited, more subtle, set of trajectories.

Draws Since the winner is decided by counting occupied cells, in a two-player game it is only possible to draw when the total number of cells is even i.e. $\forall w, h \in \mathbb{N}: \; w \cdot h = 2n, \; n \in \mathbb{N}$.

3 Method

Experiments are carried out on a small custom framework, implementing a simple version of AlphaZero for the game of Corso. Neural networks are used as function approximators. In practice, an automatic differentiation framework is used to train the networks via gradient descent, minimizing the loss function. Other components of the pipeline are entirely single threaded for simplicity (MCTS, self-play data generation). Since most of the algorithm follows the design described in [10] and [9], only notable differences are reported.

3.1 Iterations

The pipeline is similar to the one described in AlphaGo Zero[10], that is, training is divided in iterations. During each iteration, a certain amount of self-play games (episodes) is played to generate train data and then used to fit the network. The network is fit on the generated data for multiple epochs. After each iteration, a round of evaluation against different agents is carried out in order to monitor the generalization of the network.

3.2 State, policy and value representation

The state is represented as a $width \cdot height$ grid with an additional dimension (channel dimension) $2 \cdot players + 1$, which encodes the cell state as a binary vector for each spatial location. As in [10] the last channel encodes the current turn: 0 if it is first player's turn, 1 if it is second player's turn. This means that a state tensor for the interested variant, two-player 5x5 Corso, has a total size of $5 \cdot 5 \cdot (2 \cdot 2 + 1) = 125$.

Policies are represented as neural networks with a softmax output of size $width \cdot height$. The normalized output serves as probability distribution over the possible actions, even though in practice it is never directly sampled. In fact, the actual distribution used during game playing is obtained from the MCTS playouts as described in Preliminaries. Invalid moves are masked before softmax normalization.

The state value function is also represented as a neural network, which output is constrained in the range $(-1, 1)$. Game outcome is 1 if player one wins, -1 if player two wins.

3.3 Symmetries

The game of Corso, similarly to Go, is invariant to board rotations and reflections. In particular, it follows the dihedral group of the square: four rotations and four reflections, where the first rotation

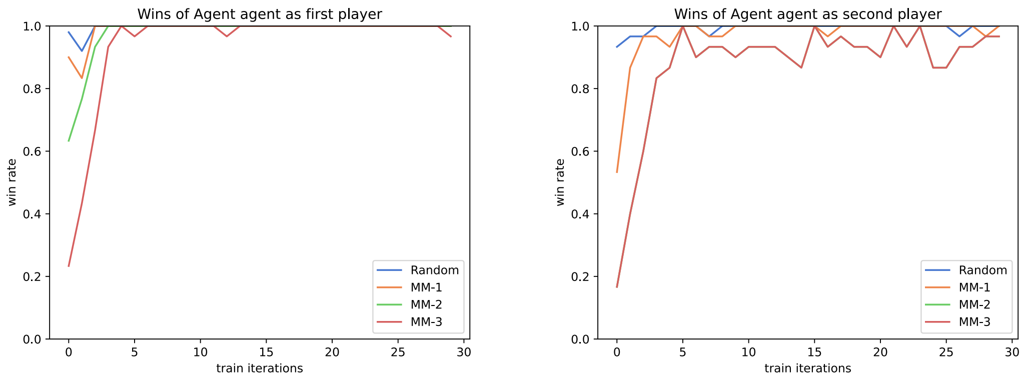


Figure 2: Train-time evaluation of neural agent against classical ones.

($0°$) is the identity function. At train time, these are exploited for data augmentation: each sample is randomly transformed following one of the 8 symmetries. Targets (improved policies) are transformed accordingly.

The game is also invariant to color inversion. This is also exploited for data augmentation by randomly (50%) inverting samples at train time. Targets (game outcomes) are inverted accordingly. This is independent from the dihedral group, so that a sample could end up both rotated and color-inverted.

Symmetries are not used in any other circumstance in the pipeline, conversely to [10] where they are used during the MCTS.

3.4 Network architecture

As in the original work a convolutional neural network is used. The network has a "convolutional tower" to extract lower level features, and two separate output heads. One for the policy ($width \cdot height$) and one for the value function (a single output node). The network is not residual. For the exact hyperparameters see Setup and evaluation.

3.5 Non-neural agents

For evaluation, comparison and Elo score calculations a family of non-neural classical agents is used. In particular, a random agent and a family of MinMax-k (MM-k) agents. k is the maximum tree depth reached by the agent (plies), which is an intuitive way of tuning its strength.

MM-k agents use a simple but effective count based heuristic:

$$h(s) = \begin{cases} M & \text{if } s \text{ is winning for the first player} \\ -M & \text{if } s \text{ is winning for the second player} \\ m_1 + w \cdot d_1 - (m_2 + w \cdot d_2) & \text{otherwise} \end{cases} \tag{11}$$

Where m_i is the number of marbles of the i-th player currently on the board. Similarly, d_i is the number of dyed cells. Dyed cells are weighted by $w \in [0, 1]$. Intuitively, marbles have the potential to be expanded and generate more dyed cells, hence they are weighted more. M is a very large number.

The top node of the search is modified so that instead of choosing the maximum (minimum) of the scores, they are normalized to form a probability distribution, where the most promising action has the highest probability to be selected. Normalization happens via a softmax with temperature $softmax(scores/T)$. The large value M ensures that certainly winning trajectories are always selected (or most likely). $T \approx 0$ ensures selection of the best score, with the additional potential benefit of uniform random selection between equally promising moves. $T \to +\infty$ collapses to uniform random selection. Non-determinism is essential in this case as MM-k agents are used to evaluate the neural agent, which however will always act greedily during evaluation. Without stochasticity in at least one of the two players all games would have the same outcome, defeating the purpose, for example, of an Elo calculation.

4 Setup and evaluation

All experiments were carried out on UniBo HPC machines mounting a Intel(R) Xeon(R) W-2123 CPU @ 3.60GHzm, a NVIDIA GeForce RTX 2080 Ti and 38GB of RAM. Training of the agent here described requires $\sim$ 6 hours with said hardware.

The network is a non residual, convolutional neural network, with two output heads, one for the policy and one for the value function. All convolutions, except the output one, are followed by batch normalization and ReLU non linearity. All dense layers, except the output one, are followed by ReLU non linearity and a dropout with probability 0.3. In more detail, the "convolutional tower" of shared parameters is:

- 4 identical Conv layers: 3x3 kernel, 64 channels, unit stride and padding.

The policy head:

- Conv layer: 3x3 kernel, 32 channels, unit stride and padding.
- Conv layer: 3x3 kernel, 1 channel, unit stride and padding.

The entire policy network, also accounting for shared layers, is fully convolutional without downsampling and an output of size $w \cdot h$. In practice, output is flattened for convenience.

The value head:

- Conv layer: 1x1 kernel, 2 channels, unit stride and no padding.
- Dense layer: 512 units
- Dense layer: 1 unit
- $\tanh$ to constrain the output between $(-1, 1)$

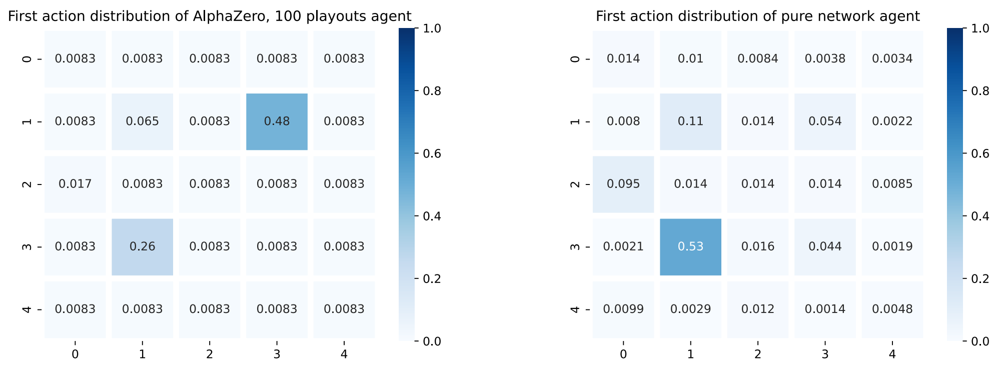


Figure 3: Evaluation of the first state (empty game board) from AlphaZero with 100 playouts, compared to the direct distribution output by the network (without any MCTS playout).

The 1x1 convolution in the value head is used to squeeze the number of channels, reducing the number of parameters of the following linear layer.

The entire network has around $\sim$ 150k parameters. Network parameters are optimized via stochastic gradient descent, in particular using the Adam optimizer with a learning rate of 10^{-3}. Each iteration consists in 100 episodes of self-play, for a total of 30 iterations. During self-play, each move is played by sampling a policy obtained through 100 playouts of MCTS, with an exploration temperature τ of 1. At evaluation time, a τ of 0 is used, so that only the most promising moves are played. Generated data is used to fit the network for 10 epochs with batches of 64. The L2 regularization coefficient c is set to 10^{-4}. The exploration factor of the PUCT algorithm c_{puct} is 1.

These hyperparameters are inspired by the original works[10][9] and by the related work[12]. In particular, the network design was inspired by [12], even though this work's network is much smaller. A larger network, such as the one they propose, proved to bring little benefits.

4.1 Evaluation methods

Win rate of the learning agent versus different agents is constantly monitored during training. In particular, each evaluation round consists in tens of games against: random playing agent, MM-1, MM-2, MM-3. Deeper agents are not considered during training due to their inference cost.

After training, 10 round-robin tournaments are played between: random player, MM-1, MM-2, MM-3, MM-4, MM-6, and the trained agent. Both the pure network and an AlphaZero with 100 playouts per move are evaluated, in order to measure the generalization of the network itself and the improvements given by the playouts. Tournaments contribute to the computation of an Elo score for each agent.

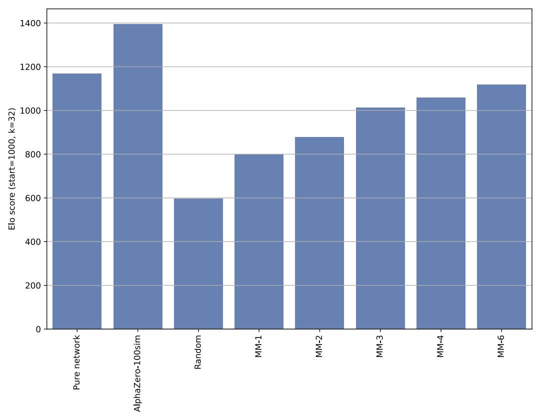
4.2 Win rates and Elo scores

Train time evaluation rounds are reported by Figure 2, clearly showing that model reaches a good extent of generalization within the first 10 train iterations.

Performance as second player seem to slightly degrade against deeper agents. This can be noticed in Figure 2 looking at the rounds against MM-3.

An example game against a human (this work's and Corso's author, Francesco Mistri) can be found in the Appendix (Figure 4).

Ultimately, Elo scores computed from round-robin tournaments show that the neural agent is consistently more performant than classical ones, even more than deeper agents whose inference time can be higher than a CPU-only query of AlphaZero . This makes AlphaZero a valuable replacement for classic tree search approaches.



Please note that the MM-k agents here have a temperature of 1, to ensure some level of diversity in the games. However, a lower temperature ($T \approx 0$) would improve their performance.

5 Conclusion

This work demonstrates how DL aided by tree search can be used to tackle competitive multiplayer games, in particular the two players game of 5x5 Corso. The agent, trained with minimal human knowledge, mostly in a single threaded environment and for few hours, is able to outperform most classical algorithmical game playing approaches as well as humans.

Possible improvements to the project include: deeper study on network capacity (e.g., smaller networks), fully fledged multithreading, use of augmentations in the tree search, tackling larger state spaces (grid size > 5), handling "even" state spaces (where draws are possible), multiplayer (> 2) support.

Acknowledgements

This work originates as a project for the Deep Learning course, held at University of Bologna. I would like to thank the professor of said course, Andrea Asperti, for introducing me to Deep Learning, as well as Mirco Musolesi, whose Autonomous and Adaptive Systems course helped me refined this knowledge. I would also like to thank my colleague Michele Faedi for the great conversations on related topics.

The game of Corso was developed by me (Francesco Mistri) to be featured in the upcoming videogame (2023-2024) Lone Planet.

Experiments were carried out on HPC machines provided by the University of Bologna for project works and research purposes.

References

[1] Thomas Anthony, Zheng Tian, and David Barber, 'Thinking fast and slow with deep learning and tree search', in *Advances in Neural Information Processing Systems*, eds., I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, volume 30. Curran Associates, Inc., (2017).

[2] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer, 'Finite-time analysis of the multiarmed bandit problem', *Machine Learning*, **47**, 235–256, (2002).

[3] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, and Susan Zhang, 'Dota 2 with large scale deep reinforcement learning', (12 2019).

[4] Rémi Coulom, 'The monte-carlo revolution in go', in *Japanese-French Frontiers of Science Symposium*, (2008).

[5] Yiheng Liu, Tianle Han, Siyuan Ma, Jiayue Zhang, Yuanyuan Yang, Jiaming Tian, Hao He, Antong Li, Mengshen He, Zhengliang Liu, Zihao Wu, Dajiang Zhu, Xiang Li, Ning Qiang, Dingang Shen, Tianming Liu, and Bao Ge. Summary of chatgpt/gpt-4 research and perspective towards the future of large language models, 2023.

[6] Nina Mazyavkina, Sergey Sviridov, Sergei Ivanov, and Evgeny Burnaev, 'Reinforcement learning for combinatorial optimization: A survey', *Computers & Operations Research*, **134**, 105400, (2021).

[7] Syed Mohib Raza, Mohammad Sajid, and Jagendra Singh, 'Vehicle routing problem using reinforcement learning: Recent advancements', in *Advanced Machine Intelligence and Signal Processing*, eds., Deepak Gupta, Koj Sambyo, Mukesh Prasad, and Sonali Agarwal, pp. 269–280, Singapore, (2022). Springer Nature Singapore.

[8] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis, 'Mastering the game of go with deep neural networks and tree search', *Nature*, **529**(7587), 484–489, (Jan 2016).

[9] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm, 2017.

[10] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis, 'Mastering the game of go without human knowledge', *Nature*, **550**(7676), 354–359, (Oct 2017).

[11] Richard S. Sutton and Andrew G. Barto, *Reinforcement Learning: An Introduction*, The MIT Press, second edn., 2018.

[12] Shantanu Thakoor, Surag Nair, and Megha Jhunjhunwala. Learning to play othello without human knowledge, 2016.

[13] Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H. Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, Laurent Sifre, Trevor Cai, John P. Agapiou, Max Jaderberg, Alexander S. Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom L. Paine, Caglar Gulcehre, Ziyu Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps, and David Silver, 'Grandmaster level in starcraft ii using multi-agent reinforcement learning', *Nature*, **575**(7782), 350–354, (Nov 2019).

Appendix

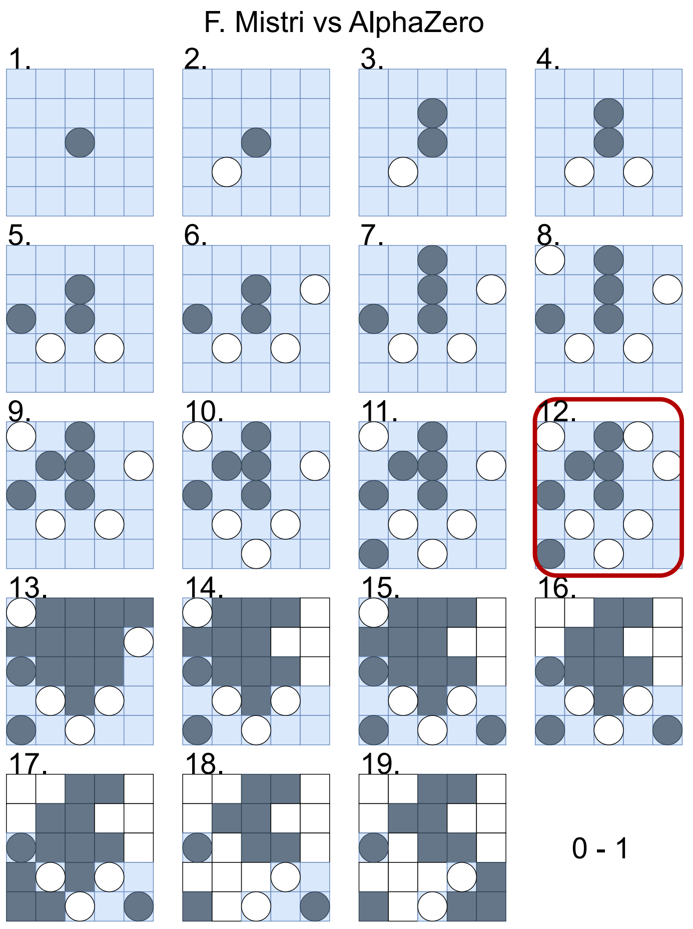


Figure 4: A game featuring the game creator Francesco Mistri against the trained AlphaZero-like agent (100 playouts per move). Notable move 12 from AlphaZero is highlighted: AlphaZero puts pressure on first player's structure, forcing him to expand.

Note that first player unusually has gray color here. The color order does not make any difference, but in Corso games first player usually has white color.