A Deep Reinforcement Learning Survey on Zero-Sum Games
Advanced Deep Learning Models and Methods Project

Andrea Menta
Politecnico di Milano
andrea.menta@mail.polimi.it

Giovanni Nisti
Politecnico di Milano
giovanni.nisti@mail.polimi.it

Abstract

There are many techniques and architectures used in the Deep Reinforcement Learning field. We analyze 2 of the main model-free approaches proposed in literature, namely Dueling Double Deep Q-Network (DDDQN) [17] and Soft Actor-Critic (SAC) [3] and we asses their performance in increasingly harder zero-sum game-environments with no prior knowledge. Then we explore the self-play learning framework [13], comparing its performance with the above cited classic learning approaches. While we observe comparable results for DDDQN and SAC, both succeeding in reaching the expected theoretical results in a reasonable amount of steps, we notice a significant higher requirement of samples when using the self-play learning architecture, making it really difficult to successfully deploy. Moreover, although being very promising, the self-play agent seems not easily applicable without adequate computational resources.

1. Introduction

In the last years Deep Learning has massively contributed in advancing performance and scalability in a variety of challenging domains. One of these fields is Reinforcement Learning, where the approximation power and flexibility of neural networks have made possible to solve complex sequential decision making problems. However, the price of handling such powerful tools lies in the optimization complexity, due to the brittleness of their hyperparameters, and in the amount of data required, resulting in massive number of steps even for simple tasks.

In this survey we firstly analyze 3 of the most significant algorithms, namely Dueling Double Deep Q-Network, Soft Actor-Critic, and AlphaGo Zero, highlighting their main characteristics and innovations. Then we contextualize these algorithms in the zero-sum game field, proposing 3 increasingly complex environments, specifically TicTacToe, Connect Four and Santorini. Afterwards in the experiments we assess the performance of the algorithms comparing them with both the expected theoretical results and their relative behaviour.

Finally, for reproducibility reasons, we provide our GitHub repository[1], containing the implementations of all the algorithms and the test infrastructure.

2. Context

Over the years multiple approaches have been proposed, allowing us to create a *taxonomy* of the major algorithms. The first possible distinction is between model-free and model-based agents, where the first ones learn without explicitly knowing the dynamics of the environment, while the latter exploit a model to describe the state transitions and rewards. Even if the model-based approach allows the agent to plan a strategy it is usually more complex to develop, due to the lack of a reliable representation of the environment and the difficulty of deriving it from pure experience.

Another distinction can be made based on the objective of the learning procedure. For what concerns the model-free approaches there exist 2 major families, namely policy optimization and value optimization. The first focuses on optimizing the parameters θ of a policy $\pi_\theta(s, a)$, usually in an on-policy fashion, while the latter learns an approximation $Q(s, a; \theta)$ of the optimal state-action value function following an off-policy approach, which leads the agent to a final greedy-policy based on the Q-function values. Representatives of the first category are for example Asynchronous Advantage Actor-Critic (A3C) [5] and Proximal Policy Optimization (PPO) [11], while for the second we can include all the derivatives of the Deep Q-Network (DQN) [6] algorithm; there exist also some hybrid approaches such as Soft Actor-Critic (SAC) [3] and Twin Delayed Deep Deterministic Policy Gradient (TD3) [2].

Regarding the model-based agents we can encounter much orthogonal paradigms that can not be easily classified, nevertheless one possible distinction can be made between the algorithms where the model is learnt, such

[1] https://github.com/Menta99/Advanced_Deep_Learning_Models_and_Methods-Menta-Nisti

as Imagination-Augmented Agents (I2A) [8], and the one where it is given, such as the derivatives of AlphaGo [12].

3. Agents

We will expose the problem formulation of the generic sequential decision making problem and its notation, then we will explore 3 solutions proposed in the literature.

3.1. Background

We can represent the reinforcement learning framework using a sequential decision making problem as follows [14]. There is an agent that interacts with an environment $\mathcal{E}$. At each time step t the agent observes a state $x_t \in \mathcal{S}$ produced by the environment and performs an action $a_t \in \mathcal{A} = \{1, ..., |\mathcal{A}|\}$ obtaining as result a reward $r_t \in \mathcal{R}$ and a new observation $x_{t+1} \in \mathcal{S}$. The goal of the agent is to find an optimal policy $\pi^\star(s, a)$ in order to maximize the expected sum of discounted rewards (known also as expected discounted return) $R_t = \sum_{\tau=t}^{\infty} \gamma^{\tau-t} r_\tau$, where $\gamma \in [0, 1]$ is a discount factor that trades-off the importance of immediate and future rewards.

3.2. Dueling Double Deep Q-Network

DDDQN is the state-of-the-art of the Deep Q-Learning architecture, thus it is a model-free value-based off-policy algorithm which uses neural networks as approximators for the state-action value function, namely $Q(s, a; \theta)$. The foundation of this approach lies in the Bellman Optimality Equation [14]:

$$Q^\star(s, a) = \mathbb{E}_{s' \in \mathcal{E}} \left[r + \gamma \max_{a'} Q^\star(s', a') \right] \tag{1}$$

which derives from the intuition that given the knowledge of the optimal Q-function for every action a' in the next state s', the optimal policy should select the action a' that maximizes the expected value of $r + \gamma \max_{a'} Q^\star(s', a')$. Due to the impracticality of such a tabular approach, neural networks have been introduced as approximators of the Q-function [6], with the objective of minimizing the following loss:

$$L_t(\theta_t) = \mathbb{E}_{s,a,r,s'} \left[\left(y_t^{DQN} - Q(s, a; \theta_t) \right)^2 \right] \tag{2}$$

with

$$y_t^{DQN} = r + \gamma \max_{a'} Q(s', a'; \theta^-) \tag{3}$$

where θ^- represents the parameters of a separate target network, which is updated to the parameters of the online network only after k learning-steps of the latter. A further improvement has been proposed in order to minimize the overestimation bias induced by the use of the same network to both select and evaluate an action, amplified also by the use of the $\max$ operator. The proposed solution, namely Double Deep Q-Network [15], proposes the following modification:

$$y_t^{DDQN} = r + \gamma Q(s', \arg\max_{a'} Q(s', a', \theta_t); \theta^-) \tag{4}$$

The 2 final upgrades have concerned more the architecture than the learning procedure, specifically the experience replay buffer and the neural network structure. The first proposes to substitute the uniform sampling of episodes with a prioritized one [10], where the sampling probability P_j of each transition

$$P_j = \frac{p_j^\alpha}{\sum_k p_k^\alpha} \tag{5}$$

is updated using the magnitude of the TD-error as a proxy for the novelty of the event

$$p_j = \left| y_t^{DDQN} - Q(s_j, a_j; \theta_t) \right| \tag{6}$$

and an importance sampling weight w_j

$$w_j = \left(\frac{1}{N P_j} \right)^\beta \tag{7}$$

is used to alleviate the induced bias in the optimization step. The latter improvement instead introduces explicitly the concept of advantage function

$$A(s, a) = Q(s, a) - V(s) \tag{8}$$

which models the relative importance of each action. This is directly translated in the neural network architecture [17] by creating 2 separate streams, after a common feature extractor part, which compute respectively the advantage and the value function, that are then combined via a special aggregating layer to produce a more stable estimate of the state-action value function

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + \left(A(s, a; \theta, \alpha) - \frac{1}{|A|} \sum_{a'} A(s, a'; \theta, \alpha) \right) \tag{9}$$

All these improvements have contributed to create the Dueling Double Deep Q-Network architecture which has achieved human level performance in the Atari 2600 testbed.

3.3. Soft Actor-Critic

SAC is a model-free policy-based off-policy algorithm built upon the maximum entropy reinforcement learning framework using an actor-critic architecture, applied to continuous action spaces. The first novelty proposed by the authors has been to enrich the classic reinforcement learning objective function, namely the expected discounted return

r, with a regularizing factor based on the expected *entropy* of the policy

$$J(\pi) = \sum_{t=0}^{T} \mathbb{E}_{(s_t,a_t)\sim\rho_\pi} \left[r(s_t, a_t) + \alpha \mathcal{H}\left(\pi(\cdot|s_t)\right) \right] \quad (10)$$

with

$$\mathcal{H}(\pi(\cdot|s_t)) = -\log \pi(\cdot|s_t) \quad (11)$$

This correction accounts for the exploration-exploitation trade-off, in fact with higher values of the temperature parameter α we push the policy towards greater exploration, preventing it from getting stuck in local optima. The core of the agent is the soft policy iteration method, which is an algorithm that learns the optimal maximum entropy policy by alternating between policy evaluation and policy improvement steps. The first step consists in the evaluation, performed by the critic networks, of a batch of transitions sampled from an experience replay buffer. Here, following the approach proposed in TD3 [2], is applied the clipped double-Q trick, in order to reduce the overestimation bias. This technique consists in using the minimum value of 2 decoupled critic target-networks to estimate the Q-value of a state-action pair.

$$y_t^{SAC} = r + \gamma \left(\min_{i=1,2} Q(s', \tilde{a}'; \phi_{i,t}^-) - \alpha \log \pi_{\theta_t}(\tilde{a}'|s') \right), \quad \tilde{a}' \sim \pi_{\theta_t}(\cdot|s') \quad (12)$$

The obtained Q-target is then used to compute the loss

$$L_t(\phi_{i,t}) = \mathbb{E}_{s,a,r,s'} \left[\left(y_t^{SAC} - Q(s, a; \phi_{i,t}) \right)^2 \right] \quad (13)$$

and perform a gradient descent step to update the 2 critic online-networks. Then the policy optimization step is performed, minimizing the expected KL-divergence

$$L_t(\theta_t) = \mathbb{E}_s \left[\min_{i=1,2} Q(s, \tilde{a}, \phi_{i,t}) - \alpha \log \pi_{\theta_t}(\tilde{a}|s) \right], \quad \tilde{a} = f_{\theta_t}(\epsilon_t, s) \quad \epsilon_t \sim \mathcal{N}(\vec{0}, I) \quad (14)$$

of the policy represented by the actor network. Due to the nature of the latter, which outputs the mean and the standard deviation vectors of the action distributions, this step involves the use of the *reparameterization trick* [4] in order to allow the gradient computation. Finally a polyak average update [7] of the critic target-networks is performed

$$w_{\phi_{i,t}^-} = \rho w_{\phi_{i,t}^-} + (1 - \rho) w_{\phi_{i,t}} \quad (15)$$

to guarantee more stability to the learning procedure. This algorithm has achieved state-of-the-art performance with an impressive sample efficiency in challenging continuous-control domains, such as the OpenAI Gym environments. There exist also a discrete-action variant of SAC [1] which we will use for the rest of our survey. This version is based on the same soft policy iteration method as the continuous one, with only some small modifications to account for the different action space. The most significant lies in the ability of computing directly the expectations of the actions, without involving the reparameterization trick, due to the actor network output, that now is the exact action distribution.

3.4. AlphaGo Zero

AlphaGo Zero is a model-based reinforcement learning algorithm based on the self-play learning framework. It is one of the latest iterations of the AlphaGo series, sharing with them some of their main features and proposing at the same time a simpler approach to master the game of Go without human knowledge. As its predecessors it joins the approximation power of neural networks with the classic Monte Carlo Tree Search (MCTS) heuristic. The main improvements of this approach can be found both in the network architecture and in the learning algorithm. For what concerns the architecture the 2 separate networks, devoted respectively to the computation of the move probabilities p (policy network) and the position evaluation v (value network), have been substituted by a single network f_θ, with the same outputs as before. This new implementation adds residual blocks to the basic convolutional layers, increasing the overall accuracy. Another novelty is in the input which has been pruned by the handcrafted features and is now composed simply by the raw board history, created by stacking 17 19×19 binary feature maps representing respectively the presence/absence of a stone in the last 8 time steps (one for each player) acting as a sort of *attention* mechanism [16] and a final layer indicating the color to play. Regarding instead the self-play learning pipeline, it is composed by 3 key components, which are executed asynchronously in parallel:

1. Continuous optimization of the neural network parameters θ by sampling from an experience replay buffer containing the most recent transitions
2. Evaluation of the actual player against the current best $\alpha_{\theta_\star}$ with the eventual swap in case of solid improvement
3. Generation of new self-play data using the actual best player $\alpha_{\theta_\star}$ and the MCTS heuristic

The neural network optimization is carried out following a policy iteration approach, where the MCTS is used both in the policy evaluation and in the policy improvement step. In fact MCTS has shown to select much stronger moves than the network, acting as a powerful policy improvement operator on one hand and on the other granting good estimate

of the position value based on the performed simulations. These factors have lead to the following network loss

$$L_t(\theta_t) = (z_t - v_t)^2 - \pi^T \log p_t + c||\theta||^2 \tag{16}$$

that given a sampled transition (s_t, π_t, z_t) and the corresponding network output $f_{\theta_t}(s_t) = (p_t, v_t)$, combines the mean squared error between the predicted value and the transition outcome, the cross-entropy loss between the predicted action distribution and the MCTS policy and a L2 weight regularization. Lastly the experience replay buffer is continuously updated by performing multiple self-play games in parallel and storing new transitions. The generic self-play game is carried out as follows:

1. *Simulation*, given the current board position a fixed number of MCTS simulations is performed following these steps:
   (a) *Selection*, starting from the actual root of the search tree till a leaf is reached, an action is selected in each node based on its statistics, following a modified version of the PUCT algorithm [9]
   (b) *Expansion and Evaluation*, the leaf node is added to the evaluation queue of the neural network; once evaluated it is expanded, the corresponding node statistics in the tree are initialized and the transition edges are set with the action probabilities produced by the network
   (c) *Backup*, the edges statistics are updated along the path from the leaf to the root; a virtual loss is also involved to create diversity in the various simulations
2. *Play*, the agent after performing a fixed number of simulations, actually selects a move a in the current root s_0, proportionally to its exponentiated visit count

$$\pi(a|s_0) = \frac{N(s_0, a)^{\frac{1}{\tau}}}{\sum_{a'} N(s_0, a')^{\frac{1}{\tau}}} \tag{17}$$

   where τ is a temperature parameter that controls the exploration; then the child node corresponding to the played action becomes the new root node and the procedure restarts.

Once the game ends a score is assigned and a corresponding final reward $r_t \in \{-1, 1\}$ is propagated in all the encountered transitions (s_t, π_t, z_t) setting z_t based on the game winner from the perspective of the current player at step t; finnally these transitions are added to the experience replay buffer. This algorithm has been able to achieve superhuman performance in the game of Go, exceeding the results of its predecessors with a simpler and more scalable architecture, without any human knowledge.

4. Environments

We will describe the concept of zero-sum game and some useful metrics to assess a game complexity. Then we will analyze 3 well-known implementations of it.

4.1. Zero-Sum Games

In game theory, zero-sum games are a mathematical representation of an environment where generally 2 players are involved p_1, p_2 and their respective outcome is linked by the following relation

$$r_{p_1} + r_{p_2} = 0 \tag{18}$$

involving that one's player reward is the opposite of the other. These games are a specific example of constant-sum games and due to the satisfaction of the Pareto Optimality property, namely the impossibility to improve the outcome of one player without worsening that of the other, they fall also into the category of *conflict games*.

4.2. Metrics

There exist different metrics to assess the complexity of a game based on its structure. Some of the most used are the following:

- *Branching factor* b, defined as the average number of child nodes of each game state
- *Average game length* d, defined as the average number of plies, namely a turn taken by one of the players
- *State-space complexity* (SSC), defined as the number of legal game positions reachable from the initial state of a game, expressed as logarithm to base 10
- *Game-tree complexity* (GTC), defined as the number of leaf nodes in the smallest full-width decision tree, namely a tree that includes all nodes at each depth, that establishes the value of the initial position, expressed as logarithm to base 10; this can be viewed as a proxy of the number of positions evaluated by a classic Min-Max search and due to its estimation complexity it is usually adopted the following inequality

$$GTC \geq b^d \tag{19}$$

The game complexities of the games covered below are exposed in Table 1.

4.3. TicTacToe

TicTacToe is a 2 player zero-sum turn game belonging to the family of m, n, k-games, namely the games with a $m \times n$ board with the winner being the first player to place k stones in a row horizontally, vertically or diagonally, thus it is a $3, 3, 3$-game. It is known to be solved.

Game	Branching Factor	Game Average Length	State-Space Complexity ($\log_{10}$)	Game-Tree Complexity ($\log_{10}$)
TicTacToe	4	9	3	5
Connect Four	4	36	13	21
Santorini	60	25	21	44

Table 1: Game complexity metrics

4.4. Connect Four

Connect Four is a 2 player zero-sum turn game. As TicTacToe it belongs to the family of m, n, k-games, specifically is a $6, 7, 4$-game, with the main difference that the stones are not positioned in a cell but are dropped in a column, forming a pile. It is also known to be solved.

4.5. Santorini

Santorini is a 2 player zero-sum turn game set up in a 5×5 board, where each player controls 2 pawns, named *workers*, and each ply is composed by 2 mandatory ordered phases:

1. Movement, the player selects a worker and moves it in a free adjacent cell (free from workers or domes), eventually climbing up of at most 1 level, or falling down of an arbitrary number of levels

2. Build, the player with the same worker selected before builds a new piece in a free adjacent cell (same as above), following the building order (level-1 $\rightarrow$ level-2 $\rightarrow$ level-3 $\rightarrow$ dome)

A player wins if one of his workers climbs up to a level-3 or if the opponent is not able to complete the turn (movement or build). This game as an estimated complexity that lies between checkers and chess and it is not known to be solved.

5. Experiments

We will describe the test setup for each game, including the main features of the different configurations, and comment the performance of the models in each context.

5.1. General Setup

We have created an environment for each of the game tested following the class interface of the Gym API[2], with a dedicated wrapper to handle the network opponent behaviour. Specifically each game features 2 possible *representations*

- *Tabular*, a synthetic representation including the minimum data to represent the game state
- *Graphic*, a simple gray-scale image to represent the board and the pawns

and 2 *opponents*:

- *Random*, an agent that follows a stochastic policy based on game rules
- *Smart*, a classic search algorithm that chooses the best action to play building a game tree

Based on the game representation DDDQN and SAC implement 2 different network structures for what concerns their feature extractor, specifically fully connected for the Tabular and convolutional for the Graphic; for AlphaGo Zero instead we have stuck to the original implementation based on convolutions and residual blocks. For what concerns the reward function, each environment assigns, at the end of each game, a value of 1 for the victory, 0 for the tie, -1 for the loss and -2 in case of invalid actions. Further implementation details can be found in the Appendix A.

We have then performed several tests to assess the models performance with respect to the different configurations.

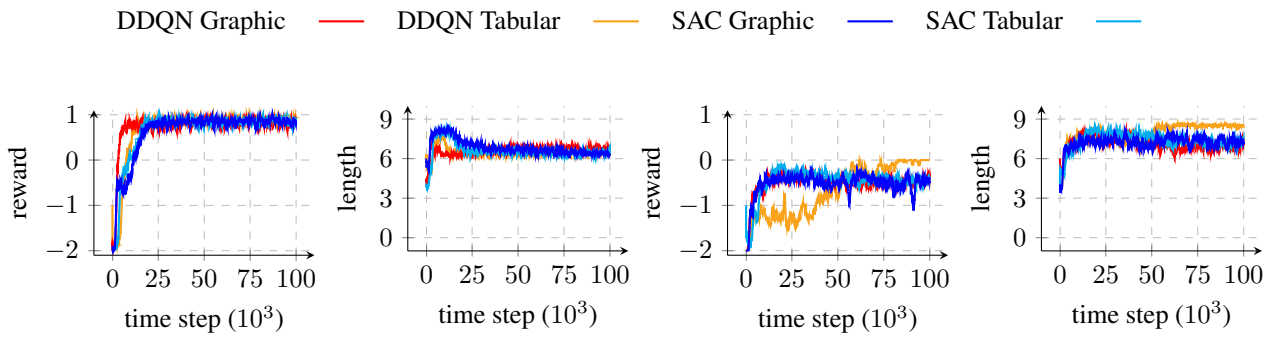
5.2. TicTacToe

Due to the models structures we have adopted a discrete action space with values from 0 to 9, representing the cell to tick, numbered from the top-left to the bottom-right of the board. Then we have performed 100k training steps, with a 10-game evaluation every 100 steps for DDDQN and SAC; for AlphaGo Zero instead have used 25 iterations of the learning procedure, each composed by 64 self-play games, 1 optimization step on 128 mini-batches of size 32, 11 evaluation games and 10 test games.

5.2.1 vs Random

The theoretical result of this configuration, keeping into consideration the small branching factor of TicTacToe, is about 0.9 for the reward and 7 for the game length. As expected both the model-free agents have succeeded in matching these expectations, reaching, as shown in Figure 1a, a stable performance in about 25% of the total training steps.

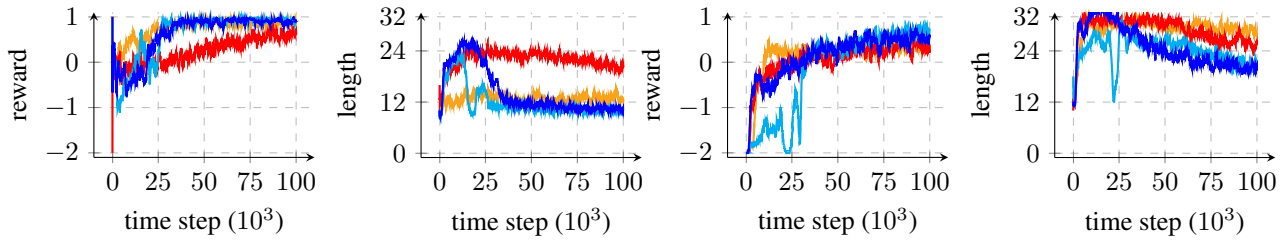
[2] https://www.gymlibrary.dev



(a) TicTacToe Random

(b) TicTacToe MinMax

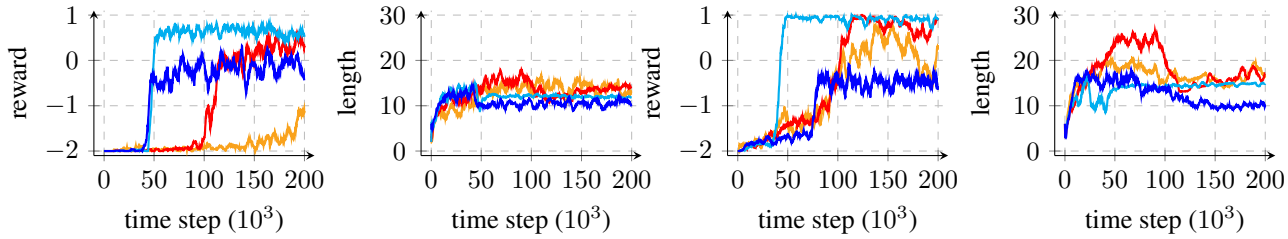
Figure 1: TicTacToe



(a) Connect Four Random

(b) Connect Four MinMax

Figure 2: Connect Four



(a) Santorini Random

(b) Santorini MonteCarlo

Figure 3: Santorini

5.2.2 vs MinMax with Alpha-Beta Pruning

This opponent leverages a complete MinMax tree-search enriched with some randomness based on board symmetries to expand the range of moves, thus the maximum achievable result for the models is a tie. As can be seen in Figure 1b, all the agents have succeeded in crossing the reward threshold of -0.5, implying a tie rate greater than 50%.

5.3. Connect Four

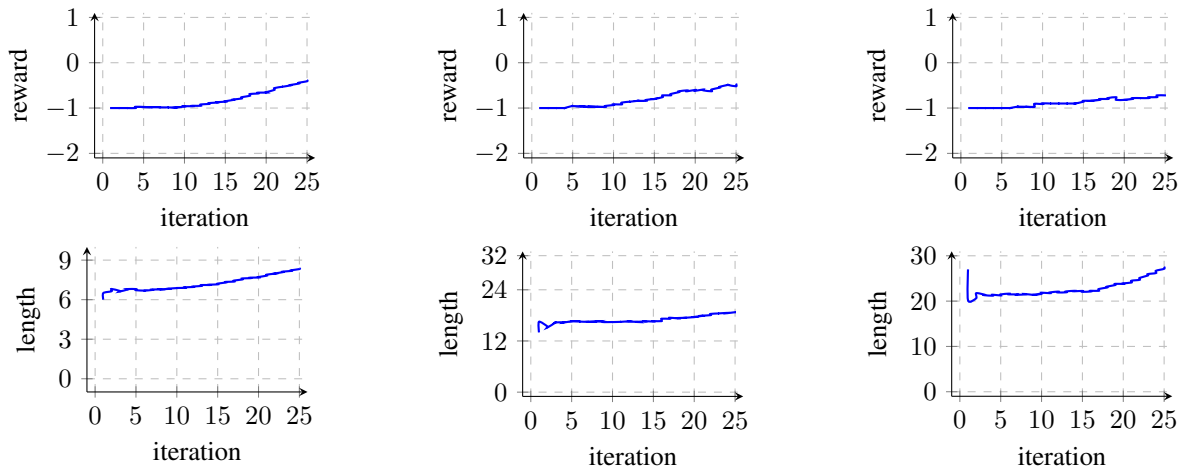
We have opted for a 7-valued discrete action space, representing the columns where to place the stone, numbered from left to right. The tests have been setup the same way as TicTacToe.

5.3.1 vs Random

As seen for TicTacToe, the stochastic policy has been easily outperformed in a short time span (Figure 2a) by both agents even if with 2 different learning curves, in fact on one hand SAC has reached better optima, while on the other DDDQN has exhibited a smoother convergence even if less performing. Given the game structure of Connect Four the agents have been also less prone to invalid actions, resulting in high rewards even in the early stages of training.

5.3.2 vs MinMax with Alpha-Beta Pruning

For performance reasons we have limited the maximum search depth of this MinMax to 6, thus even if strong it was



(a) TicTacToe MinMax (b) Connect Four MinMax (c) Santorini MonteCarlo

Figure 4: AlphaGo Zero

not foolproof. All the agents have successfully reached a reward between 0 and 1, with the tabular version of SAC as the leading performer of the batch by far (Figure 2b).

5.4. Santorini

For what concerns Santorini the turn steps have been condensed in a single action, ranging from 0 to 128, representing all the possible combinations of pawn selection (2), movement (8) and build (8). Due to the higher game complexity we have doubled the total training steps for SAC and DDDQN, while for AlphaGo Zero we have kept the same parameters.

5.4.1 vs Random

The extremely high game-tree complexity paired with the unpredictability of a random opponent have made the learning procedure really challenging for the agents. DDDQN resulted as the worst of the 2, struggling in reaching the expected performance, while SAC (specially the tabular one) has shown a significant better learning trend (Figure 3a).

5.4.2 vs MonteCarlo Tree Search

As for the ConnectFour MinMax we have been obliged to limit the exploration of the MonteCarlo Tree Search, perfoming 100k initial simulations and 10 simulations for each action, resulting in a narrower game strategy. This limit has paved the way for the agents, that given the fewer game differences, have successfully learned to defeat their opponent (even if with a brittle strategy) as shown in Figure 3b.

5.5. Self-Play

The self-play agent, due to the structure of its algorithm, ensures a more stable convergence, apparently solving the problem of catastrophic forget (experienced with few of the above agents). This stability comes at the price of a really demanding and slow training procedure that, even if highly parallelizable, cannot be easily deployed to achieve good results in a reasonable time span with limited resources. In our tests, as shown in Figure 4, we have spotted, even if very subtle, a positive trend that could imply an effective learning procedure. For the future we reserve the opportunity to further investigate the performance of this model, maybe adopting a more optimized implementation and more computational resources.

5.6. Tournament Test

Finally, to assess the relative performance of each model and the solidity of their policies, we have performed a tournament for each game, where each trained agent has been tested against all the others (including itself) in 100 games, playing in both turns. The obtained results are shown in Appendix B.

6. Conclusion

We have successfully re-implemented from scratch all the 3 proposed approaches and we have applied them to the zero-sum games field, reaching the expected theoretical results for both the model-free algorithms (DDDQN and SAC), with the only regret concerning the lack of computational resources to fully train the AlphaGo Zero model.

For the future we reserve the opportunity to further deepen the above mentioned model, trying to reach a more

solid performance in the presented games; furthermore we plan to extend the repository to include other state-of-the-art algorithms and games.

References

[1] P. Christodoulou. Soft actor-critic for discrete action settings. *CoRR*, abs/1910.07207, 2019. 3

[2] S. Fujimoto, H. van Hoof, and D. Meger. Addressing function approximation error in actor-critic methods. *CoRR*, abs/1802.09477, 2018. 1, 3

[3] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In J. G. Dy and A. Krause, editors, *ICML*, volume 80 of *Proceedings of Machine Learning Research*, pages 1856–1865. PMLR, 2018. 1

[4] D. P. Kingma, T. Salimans, and M. Welling. Variational dropout and the local reparameterization trick. *CoRR*, abs/1506.02557, 2015. 3

[5] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. 2016. cite arxiv:1602.01783. 1

[6] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning, 2013. cite arxiv:1312.5602Comment: NIPS Deep Learning Workshop 2013. 1, 2

[7] B. Polyak and A. Juditsky. Acceleration of stochastic approximation by averaging. *SIAM Journal on Control and Optimization*, 30:838–855, 07 1992. 3

[8] S. Racanière, T. Weber, D. P. Reichert, L. Buesing, A. Guez, D. J. Rezende, A. P. Badia, O. Vinyals, N. Heess, Y. Li, R. Pascanu, P. W. Battaglia, D. Hassabis, D. Silver, and D. Wierstra. Imagination-augmented agents for deep reinforcement learning. In I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, editors, *NIPS*, pages 5690–5701, 2017. 2

[9] C. D. Rosin. Multi-armed bandits with episode context. In *ISAIM*, 2010. 4

[10] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay, 2015. cite arxiv:1511.05952Comment: Published at ICLR 2016. 2

[11] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017. 1

[12] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–, Jan. 2016. 2

[13] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354–, Oct. 2017. 1

[14] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, March 1998. 2

[15] H. van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning, 2015. cite arxiv:1509.06461Comment: AAAI 2016. 2

[16] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, page 5998–6008. Curran Associates, Inc., 2017. 3

[17] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas. Dueling network architectures for deep reinforcement learning, 2015. cite arxiv:1511.06581Comment: 15 pages, 5 figures, and 5 tables. 1, 2

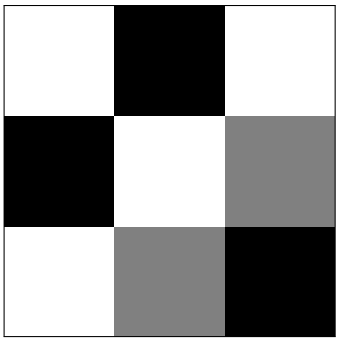
Game	Tabular Representation	Graphic Representation
TicTacToe	$3 \times 3 \times 1$	96×96
Connect Four	$6 \times 7 \times 1$	192×224
Santorini	$5 \times 5 \times 6$	160×160

Table 2: Environment State Representation

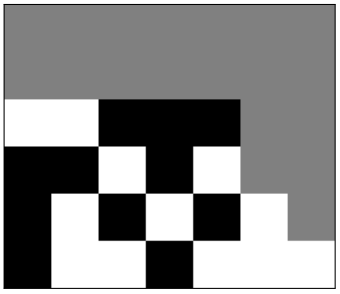
A. Implementation Details

Each game has a specific representation of the state as shown in Table 2. In the following is reported the meaning of each cell value[3] based on the game and its representation.

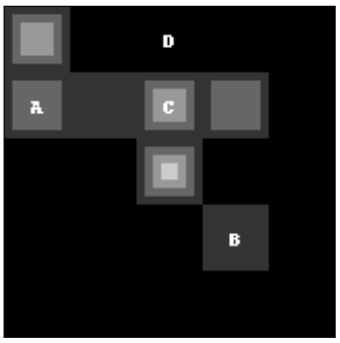
- *TicTacToe Tabular*
  - first player (1)
  - empty (0)
  - second player (-1)
- *TicTacToe Graphic*
  - first player (255)
  - empty (128)
  - second player (0)
- *Connect Four Tabular*, same as TicTacToe Tabular
- *Connect Four Graphic*, same as TicTacToe Graphic
- *Santorini Tabular*, each cell represents the presence/absence (1, 0) of the specific pawn (based on the layer, specified between brackets)
  - level 1 building (0)
  - level 2 building (1)
  - level 3 building (2)
  - dome (3)
  - first player workers positions (4)
  - second player workers positions (5)
- *Santorini Graphic*
  - empty (0)
  - level 1 building (51)
  - level 2 building (102)
  - level 3 building (153)
  - dome (204)
  - players (A,B,C,D, with value 255)



(a) TicTacToe



(b) Connect Four



(c) Santorini

Figure 5: Game Frames

In Figure 5 we report an example of the *Graphic* representation of each game state for better visualization.

Another important implementation detail consists in the training game structure, in fact the agent is trained alternatively playing as the first and the second player; this approach has shown to improve the model performance, making the learned policy more robust to eventual strategy changes.

For what concerns the main *hyperparameters* used in the tests, they are reported in Table 3.

B. Extended Test Results

The tournament test described in Section 5.6 has been performed, for what concerns the model-free agents, using

[3] Values are range normalized before feeding them to the agent

Parameter	DDDQN	SAC	AlphGo Zero
Network Update	4	1	1
Target Update	10k	-	-
Discount Factor	0.99	0.99	-
Learning Rate	$1e^{-4}$	$3e^{-4}$	$1e^{-2}$
Loss	Huber	Huber	Custom
Optimizer	Adam	Adam	Adam
Episode Number	100k/200k	100k/200K	25
Learning Start	1k	1k	0
Memory Size	32768	32768	20k
Memory Alpha	0.7	0.4	0
Memory Beta	0.7	0.4	0
Batch Size	32	32	32
Mini Batches	1	1	128
Gradient Norm	10	5	-
Tau	1	0.005	-
Initial Entropy	-	50	-
Initial Epsilon	1.0	1.0	-
Final Epsilon	0.05	0.05	-
Win Percentage	-	-	55%

Table 3: Algorithms Hyperparameters

Result	Score
First Player Normal Win	1
First Player Invalid Win	2
First Player Normal Loss	-1
First Player Invalid Loss	-2
Draw	0

Table 4: Tournament Scoring

an epsilon-greedy policy, with epsilon set to 0.01, in order not to worsen too much the performance but at the same time allowing variety in the test games. For what concerns instead the self-play agent we have kept the same parameters involved in the training procedure.

The results can be observed in Tables 5,6, 7, specifically we report the average reward from the perspective of the first player, keeping into consideration the scoring metrics reported in Table 4.

First Player	DDDQN Graphic Random	DDDQN Tabular Random	DDDQN Graphic MinMax	DDDQN Tabular MinMax	SAC Graphic Random	SAC Tabular Random	SAC Graphic MinMax	SAC Tabular MinMax	SelfPlay
DDDQN Graphic Random	1.000	0.940	0.890	0.980	1.880	0.980	1.000	0.870	0.100
DDDQN Tabular Random	1.000	0.000	0.940	0.940	0.970	0.970	0.870	1.000	-0.090
DDDQN Graphic MinMax	-1.980	-0.930	-0.040	0.940	-0.980	-0.970	0.080	0.010	0.010
DDDQN Tabular MinMax	0.930	0.900	0.010	0.980	1.820	0.900	-0.040	0.020	-0.040
SAC Graphic Random	0.920	0.050	1.870	1.000	0.970	0.090	0.980	0.970	0.010
SAC Tabular Random	0.950	0.980	0.970	0.980	0.940	0.980	1.020	0.960	-0.080
SAC Graphic MinMax	-0.960	-0.009	-1.830	-1.910	-0.005	0.920	0.080	0.050	-0.060
SAC Tabular MinMax	-0.970	-0.010	0.020	0.900	-0.870	-1.930	-0.020	-1.980	0.020
SelfPlay	0.040	0.040	0.060	-0.020	0.090	-0.010	0.050	0.016	0.070

Table 5: TicTacToe Tournament

First Player	DDDQN Graphic Random	DDDQN Tabular Random	DDDQN Graphic MinMax	DDDQN Tabular MinMax	SAC Graphic Random	SAC Tabular Random	SAC Graphic MinMax	SAC Tabular MinMax	SelfPlay
DDDQN Graphic Random	0.900	-0.900	-0.870	0.900	-0.920	0.880	0.940	0.940	0.600
DDDQN Tabular Random	0.980	1.000	1.000	0.900	0.920	0.940	0.980	0.980	0.670
DDDQN Graphic MinMax	0.800	-0.900	0.940	0.920	-0.920	-0.940	-0.880	0.960	0.520
DDDQN Tabular MinMax	-0.740	0.980	-0.800	0.930	-0.960	-0.900	0.940	-0.780	0.670
SAC Graphic Random	0.920	1.000	0.980	0.920	0.960	0.980	1.000	-0.940	0.680
SAC Tabular Random	0.960	1.000	0.960	-0.820	-0.920	0.960	0.980	-0.760	0.500
SAC Graphic MinMax	0.920	-0.980	-0.840	0.800	-1.000	-0.920	-0.860	-0.880	0.640
SAC Tabular MinMax	-0.880	-0.960	0.940	0.880	0.970	-0.880	0.980	-0.890	0.620
SelfPlay	-0.340	-0.320	-0.470	-0.450	-0.340	-0.170	-0.330	-0.360	0.130

Table 6: Connect Four Tournament

First Player	DDDQN Graphic Random	DDDQN Tabular Random	DDDQN Graphic MonteCarlo	DDDQN Tabular MonteCarlo	SAC Graphic Random	SAC Tabular Random	SAC Graphic MonteCarlo	SAC Tabular MonteCarlo	SelfPlay
DDDQN Graphic Random	1.880	0.980	-0.870	1.880	0.960	0.970	0.930	0.970	0.810
DDDQN Tabular Random	2.000	-1.950	-0.850	-1.810	-1.900	-1.810	-1.960	-2.000	-1.810
DDDQN Graphic MonteCarlo	-0.990	-1.960	-1.920	-2.000	-1.940	1.920	-1.970	-1.960	-1.960
DDDQN Tabular MonteCarlo	-0.910	0.820	-1.910	-1.820	-1.730	-1.920	-1.900	-1.860	-1.680
SAC Graphic Random	-2.000	-0.840	-1.550	-1.960	0.750	-0.900	1.640	0.870	0.680
SAC Tabular Random	0.970	0.900	0.880	1.910	0.880	0.800	0.990	0.760	0.690
SAC Graphic MonteCarlo	-1.960	-1.000	0.930	-0.910	-1.960	-0.980	0.850	0.960	0.820
SAC Tabular MonteCarlo	0.770	0.860	0.870	-1.520	-0.760	1.880	0.860	-0.830	0.780
SelfPlay	-0.590	-0.580	-0.600	-0.690	-0.550	-0.660	-0.720	-0.490	0.240

Table 7: Santorini Tournament