Reinforcement Learning for Tic-tac-toe in Arbitrary Dimensions

Irada Bunyatova Daniil Smoliakov Gaël Van der Lee Victor Videau

***Abstract*—**Tic-tac-toe is a well-known game in reinforcement learning (RL). It has simple set of rules and a reasonable state space. Previous works have solved the problem and provided agents capable of never losing a game. Researchers are also interested in 3D tic-tac-toe and has created agents for this game based on neural networks. In this paper, we first consider the 2D tic-tac-toe and propose an optimized algorithm derived from existing methods. Then, we extend the game to n dimensions and create an agent that could learn to play the game. Finally, we provide an environment to play tic-tac-toe on a grid whose side size and dimensions can be set by the players. Link to our code:
https://mega.nz/folder/Po0DVYjb#yLeAj5SFjvmTH2HUFqjrVA

I. INTRODUCTION

The purpose of the game tic-tac-toe is very simple. There are two players and a 3x3 grid. One player can fill the grid with noughts O and the other player with crosses X. They play in turns and their goal is to be the first to align their symbol. Starting from this basic set of rules, one can complexify this game by increasing the dimension and size of the grid. The goal is then to align symbols on a grid of side k and dimension n (k^n grid). Since aligning k symbols is often easy in n dimensions, it gives a point and the game continues until the grid is full. The score determines the winner. We call this version the n-tic-tac-toe. The strategies complexity depends mainly on the dimensions, so we pay less attention to the side size.

In this paper, we study both the original and extended tic-tac-toe and we propose different approaches to create an autonomous agent for this game. We chose this problem because it is quite broad. It can involve both classical Q-table methods and more advanced ones such as Policy Gradient. The challenge is to find a good policy to play the game in 3 dimensions and above. It is a challenge because the state space rockets for more than 2 dimensions.

There are tons of repositories for 2-tic-tac-toe on GitHub. They implement the SARSA and Q-learning algorithms to update a Q-table. There are also some research papers on 3-tic-tac-toe, including deep learning and temporal differences.

A major contribution of our work is the environment. We implemented a dynamic version of n-tic-tac-toe where the players can choose the dimensions and side size of the board. While this has already been done by others, our environment can do all the alignment checks and compute the current score for any choice of dimension and side size. This cannot be found anywhere else. This contribution sets up the conditions to train a RL agent for n-tic-tac-toe. Finally, it is well optimized and can perform the checks almost instantly in 7 dimensions.

Another contribution is the optimization of the classical approach for the 2D case. We slightly modified the SARSA algorithm to adapt it to our problem. By changing the reward attribution and backpropagation, we make the learning of the Q-values faster. Moreover, we handle the grid symmetries and we avoid storing useless states. All of this enables to converge to the optimal Q-values in a smaller number of episodes.

Our final contribution is the training of an agent for the 3D case and above. The number of states in higher dimensions is extremely large. Thus, it is impossible to use SARSA or Q-learning algorithms for more than 2 dimensions. Therefore, we have implemented a Policy Gradient algorithm for the case of higher dimensions. Finally, we give the possibility to play against our agent.

II. BACKGROUND AND RELATED WORK

In reinforcement learning, a model is called an *agent*. There is no dataset, but an *environment* instead. The environment is composed of *states*, with generally an initial state, intermediate ones and one or more final states. The agent interacts with the environment by taking *actions*. Actions lead from a state to a next one. We can view this process as a Markov Decision Process, which is a graph that links the different states and actions with the probabilities of transitions [1]. The agent has a goal, which is to get to certain final states. He needs to learn the sequence of actions to take from an initial state to get to a final state that achieves the goal. The agent is guided by *rewards*. A reward is given to the agent when he achieves the goal or a subgoal. The agent aims at learning the best states to actions mapping, which is called a *policy*. An *optimal policy* is a policy that maximizes future rewards. Our goal is to find a policy *P* that is as close as possible to an optimal policy *P**.

We learn a close-to-optimal policy for 2-tic-tac-toe with SARSA. Like Q-learning, it provides “capability of learning to act optimally in Markovian domains by experiencing the consequences of actions, without requiring them to build maps of the domains” [2]. The principle is to associate a value with all possible (state, action) couples. These values are an estimation of the quality of the move in term of expected future reward. They are stored in an array called Q-table and computed during the training, according to some algorithms. The general idea is the following : the Q-values are initialized either at random or with a fixed value, and they are updated each time the agent meets their associated (state, action) during the training. There are different methods for updating the Q-values, and we will

present here SARSA. **Eq. (1)** is the SARSA formula for updating the Q-value noted by $Q^{\pi}(S_t, A_t)$ under the policy π, with S_t and A_t the current state and action, R_t the reward for taking action A_t in state S_t and S_{t+1} the resulting state. A_{t+1} is the next action chosen by the agent in S_{t+1}. Finally, $\alpha \in (0, 1]$ is the learning rate and $\gamma \in [0, 1]$ the discount factor.

$$Q^{\pi}(S_t, A_t) = Q^{\pi}(S_t, A_t) + \alpha\,[(R_t + \gamma Q^{\pi}(S_{t+1}, A_{t+1}) - Q^{\pi}(S_t, A_t)] \quad \textbf{Eq. (1)}$$

During the training, the agent chooses actions according to some policy that may depend on the current Q-table. The most common one is the epsilon-greedy policy, which selects a random action with probability epsilon, otherwise the action with the highest expected reward in the Q-table. SARSA is an on-policy, meaning that it learns the Q-values with respect to the policy it follows. This is enough to understand our work for the 2D case. Let us now introduce the n-tic-tac-toe.

In 2D, the horizontal and vertical axis are just names given to the 2 dimensions. The diagonal and antidiagonal axis are just a combination of both dimensions. In n dimensions, we rather enumerate the dimensions $d_1 \dots d_n$. We define *straight* axes the axes along one dimension, like the vertical and horizontal ones in 2D. We call *diagonal* axes the axes combining more than one dimension, like the diagonal and antidiagonal ones in 2D. It is not easy for the human brain to visualize spaces in higher than 3D, so we give an example in 4 dimensions. The mathematical conditions for the alignments are given in Appendix A, since it is not directly related to RL.

Fig. (1 a) : straight axes

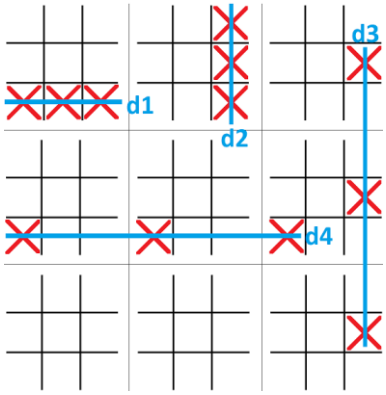


Fig. (1 b) : diagonal axes

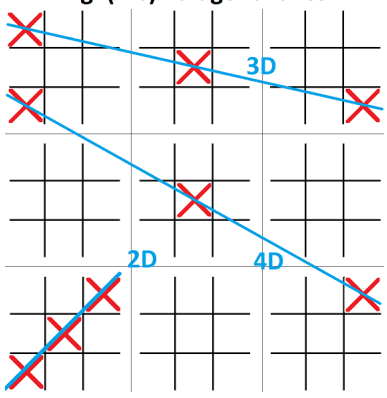


Fig. (1 a) shows alignments on straight axes (along dimension d1, d2, d3 and d4). **Fig. (1 b)** illustrates diagonal axes. The **2D** diagonal combines dimensions d1 and d2. The **3D** diagonal uses d1, d2 and d4. The **4D** diagonal uses all the 4 dimensions.

We use Policy Gradient for n-tic-tac-toe. Policy Gradient consists of directly optimizing a policy with respect to the expectation of its return G_t (long-term cumulative reward). It requires a parametric stochastic policy π_θ with parameters θ. A stochastic policy gives a probability distribution over actions so $\pi_\theta(a|s)$ gives the probability of taking action a in state s with parameters θ. The policy also needs to be differentiable [3]. The idea of Policy Gradient is to improve the policy by finding the best values for its parameters θ. These values maximize the expected return J(θ), so we would like to differentiate J(θ) in order to do a gradient ascent. Without going into details, using the Policy Gradient Theorem and the Log-Derivative trick, we get in **Eq. (2)** an expression for the gradient of the expected return with parameters θ.

$$\nabla_\theta J(\theta) = \mathbb{E}_\pi[G_t \nabla_\theta \ln \pi_\theta(a|s)] \quad \textbf{Eq. (2)}$$

We will present in section IV how we estimate $\mathbb{E}_\pi$ using the reinforce algorithm.

III. THE ENVIRONMENT

The state space is the grid whose cells are either empty or filled with a O or a X. An additional constraint is that there must be a difference of at most one between the number of O and X. Let us consider a grid of side k and dimension n. It has N different cells with N = k^n. Assuming the same symbol always plays first and the game ends when the grid is full, one can compute the total number of states *S*, given by **Eq. (3)** :

$$S = \sum_{i=0}^{N} \binom{N}{\lceil \frac{i}{2} \rceil} \times \binom{N - \lceil \frac{i}{2} \rceil}{\lfloor \frac{i}{2} \rfloor}$$

Eq. (3) : Number S of different states with respect to the number N of grid cells

To understand how we derived this formula, let us take a 3x3 grid as an example. The order the symbols are put does not matter, so it is about binomial coefficient.

The initial state is the empty grid, and $\binom{9}{0}\binom{9}{0} = 1$. Then there are 9 choices for the first symbol, and $\binom{9}{1}\binom{8}{0} = 9$. The second player plays next and has 8 choices after the 9 first possible moves, which makes $\binom{9}{1}\binom{8}{1} = 72$, and so on.

To give an order of values, it makes 6038 different states for a 3x3 grid, and 1 414 819 440 989 for a 3x3x3 grid.

The action space is the set of all different coordinates in the grid. As far as the game goes, some actions become impossible because some cells get filled. The size of the action space is $size^{dim}$ at the beginning and is reduced by one at each turn, to only one at the end (for the last available position).

The reward function *R(S, A, S')* defines the reward the agent gets for performing action *A* in state *S*. The reward depends on the new state *S'* which is also impacted by the opponent's action. This is why in SARSA the same (state, action) couple needs to be explored several times by the agent. Its value needs time to converge to the expectation of the reward. We define the reward function such that the agent gets a positive reward when he scores or wins in *S'*, or if he gets a draw.

Tic-tac-toe is a fully observed environment, which means that the agent's observations are the states. The action's result is deterministic. However, there are 2 players and the agent's opponent is not always deterministic. So, we consider that the opponent is part of the environment, which is then stochastic. Therefore, the result of an action done by the agent is not the state after he plays, but the state after his opponent has played. The only exception is when the game

ends after the agent plays. We have created two different strategies of interest for the opponent. The first one is the random strategy, the opponent picks randomly an action between those available. The second one is more advanced. If it can score, it scores. Secondly, if it can block the agent, it does. Otherwise it plays randomly.

The main challenges of this environment are the following. Firstly, the reward signal is delayed (need to align k symbols before getting a reward) and sparse (only few states among all give a feedback). Secondly, the state and action space is huge for 3D and above. Thirdly, there is an opponent which is considered part of the environment. The agent needs to be trained against one opponent that plays according to a fixed policy. The opponent's policy can be non-deterministic, however it needs somehow to be the same over time. This entails two challenges. Firstly, the agent will perform poorly if his opponent can also learn. Secondly, if the agent is trained against one policy, he might be bad against a different policy because the environment will have changed (different transition function because of different probabilities).

This work is mainly for theorical interest, but there could be some real-world applications later on. For now, people mostly play the 2D very basic game and few people give a try to the 3D version. Maybe in the future people will look for more complex games and consider n-tic-tac-toe. Then, our agent could be used as an opponent to real players. Finally, n-tic-tac-toe is close to games like Chess and Go on a reinforcement learning point of view, because all three are by turns, with deterministic actions but a non-deterministic opponent and a huge state and action space. So, making progress in n-tic-tac-toe could help research on other similar games.

IV. THE AGENT

2 dimensions : SARSA

We have designed an agent that relies on SARSA for the 2-tic-tac-toe. The use of a Q-table is easy and efficient for the 2D state and action space. The main advantage is that the agent can learn quickly good Q-values close to the optimal ones. The disadvantage of this approach is that it cannot be applied in a higher dimension.

The general rule for the game's end is when the grid is full. We adapt this rule in 2D such that the game ends when a player scores. This makes some situations impossible and reduce the 6038 different states mentioned earlier to 5478. This is still quite huge, considering there are also up to 9 available actions. By eliminating the symmetries, we can reduce the state space size to 765. Finally, we realized that the final states do not need to be stored in the Q-table. At the end, there are only 627 different states to store. For each of them, all actions are not available, so we filter the valid actions before choosing one.

We use the SARSA algorithm to update the Q-table. However, the general form of this algorithm is thought for environments with undetermined number of steps before the end and with potential intermediate rewards. In 2-tic-tac-toe, we know for sure that the game ends after a maximum of 5 actions done by the agent. Also, the only way to assess if an agent is doing good or bad moves is to look at the score when the game ends. So a reward is given only at the end of the game. This leads to a very slow learning because the rewards does not propagate well. **Algo. (1)** is the SARSA algorithm with the changes we made to it. We changed the random initialization of the Q-values by a deterministic one. We put a default value of 0.5 so that the agent is quite optimistic at the beginning. With a reward slightly greater than 0.5 for a draw, greater for a win and a reward of 0 for a defeat, the agent will explore many paths. We also change the reward propagation. Instead of updating the Q-value at each sequence State Action Reward State' Action' (that is where the name S.A.R.S.'A.' comes from), we store these information and keep going until the end. The hint is that there is no reward before the end, so using **Eq. (1)** forward in time will not be efficient. Once the game ends, we attribute the reward corresponding to the final state (win, draw or defeat) to the (state, action) couple that led to it. We just set the Q-value at the reward's value because it is directly its optimal value. Then, we backpropagate the reward to the sequence of (state, action) that led to it. We update the values using **Eq. (1)** and this time the value of the future state $Q^{\pi}(S_{t+1}, A_{t+1})$ will be a better estimate.

Algo (1) : Modified SARSA

Input:
none
Algorithm parameter:
discount factor γ
step size $\alpha \in (0, 1]$
small $\epsilon > 0$

Initialize with the value 0.5 $Q(s, a) \quad \forall s \in S, a \in A(s)$

FOR EACH episode
$S \leftarrow$ env.reset()
$A \leftarrow \epsilon$-greedy(S, Q)
state_history $\leftarrow$ [S]
action_history $\leftarrow$ [A]
WHILE TRUE
$R, S \leftarrow$ env.step(A)
IF S is final
BREAK
$A \leftarrow \epsilon$-greedy(S, Q)
state_history.append(S)
action_history.append(A)
S_last $\leftarrow$ state_history.get_last_element()
A_last $\leftarrow$ action_history.get_last_element()
Q(S_last, A_last) $\leftarrow$ R
FOR EACH two consecutive (state, action), (S_next, A_next)
in (state_history, action_history) starting from the end
Q(state, action) $\leftarrow$ (1 – α) Q(state, action)
+ α γ Q(S_next, A_next)

The optimal parameters depend on the opponent's policy and on the number of episodes. To assess the quality of the agent after his training, we make him play many games against the opponent of his training. A good agent should not

lose a single game, had he played in first or second, and should win some games if the opponent's policy is not optimal. We use grid search to find the best parameters. It turns out that the agent is learning better with an initial epsilon set to 1 with decay over time. So, the agent performs pure exploration at the beginning, and relies more and more on his Q-table along the training. It is also better to have an alpha factor decaying over time and which tends to zero at the end of the training.

Arbitrary dimensions : Policy Gradient

In Policy Gradient, instead of looking for the Q-values, we directly optimize the policy itself without need for Q-table. In our implementation, we have decided to use the Reinforce variant (Monte Carlo version) of Policy Gradient. In this algorithm, we firstly sample trajectories (as in Monte Carlo method) to get an estimation of the gradient's value. In our case, the trajectories are states of the game. This step is done because calculating the gradient over all the games is computationally expensive. The second step is estimating the return, which will then be used for calculating the gradient. Finally, the weights of the model are computed with gradient ascent on the obtained value of gradient. **Fig. (2)** shows the architecture of the neural network that we used for the Policy Gradient. It is done for a grid of side k and dimension n. Our neural network consists of input layer, one hidden layer and an output layer of the following sizes:

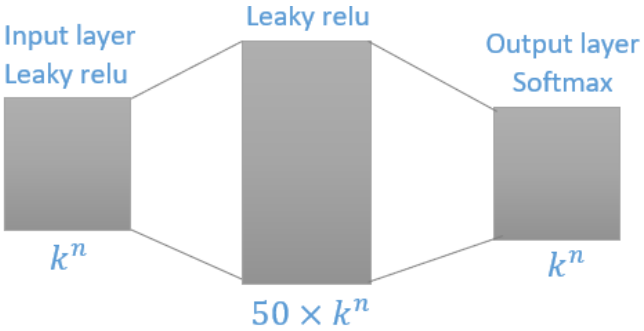


Fig. (2) : Model architecture of the neural network

Leaky relu was used as activation function for the first two layers and softmax for last layer since the output is the probability of each step. There might be some impossible states with positive probability in the output. Thus, after computing the probability of all different steps, it finds the allowed step with highest probability. This will be the final step made by the agent. For the model, the RMSprop optimization was used and for the loss, the binary cross entropy loss function. The architecture of the model, the loss and optimization functions were chosen such that they fit into our problem, as well as they show good final results after the training of the model.

We used the following reward system : +1 for the agent doing an alignment and -1 for its opponent doing an alignment. Finally, 0.8 for the final state if the game ends by a draw. The reward is sparse and delayed because it is received only after an alignment is done. To handle this problem, we have implemented a reward propagation. So, to get the final reward R_S of a state *S*, we add the reward R_{S+1} of the next state *S+1*, multiplied by a scaling factor of 0.8, to the reward $R_{S\_current}$ that *S* has before doing the reward propagation.

$$R_s = R_{s\_current} + 0.8 \times R_{s+1} \quad \textbf{Eq. (4)}$$

Eq. (4) helps to decreases drastically the sparsity of rewards. It leads to a better learning, since the agent learns to know which states lead to the winning game.

The learning process of the model was done by playing games between the model and a random policy. At the beginning, the model is performing similarly to the random policy, but throughout more games it improves its accuracy and beats the random policy in most games. For the learning of the agent, we have a batch of size 1000 (which are taken from 5000 states of the previous games).

V. RESULTS AND DISCUSSION

To assess our agent in 2D, we train him against a policy. He learns the best Q-values to beat that policy, both when he plays in first and in second. Then we make him play against another policy (possibly the same). We show the results with a varying number of training episodes. In **Fig. (3)** and **(4)**, we consider only the random policy. We see that the agent may reach an optimal policy in only 4000 training episodes, and he will reach it almost surely in 20 000 episodes.

Fig.(3) : Training and testing with the random policy (agent plays in first)

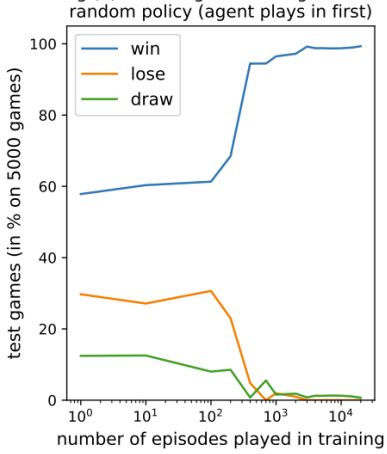
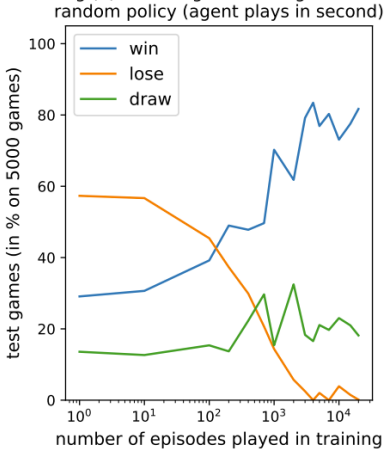


Fig.(4) : Training and testing with the random policy (agent plays in second)



Now we make our agent face the advanced policy described before. We remind that this policy firstly scores if it can score, secondly blocks the agent if it can, and plays randomly otherwise. **Fig. (5)** and **(6)** show the results of this matchup. The agent starts to get his best results at about 1000

Fig.(5) : Training and testing with the advanced policy (agent plays in first)

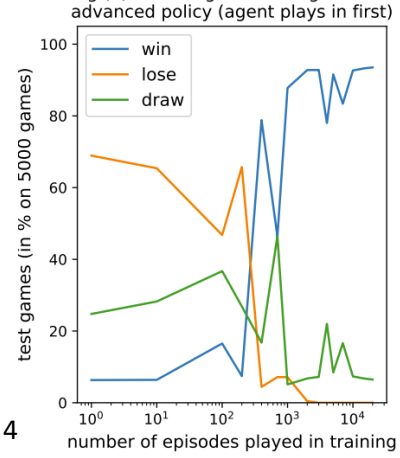
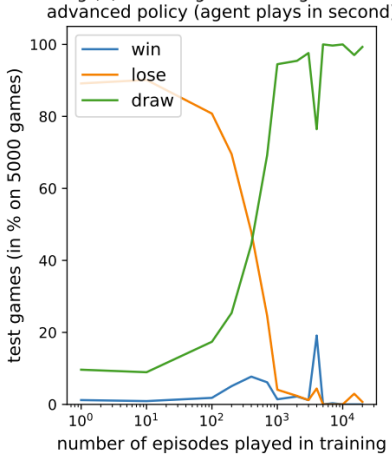


Fig.(6) : Training and testing with the advanced policy (agent plays in second)



episodes. It is less than the threshold of 4000 for the random policy. This can be explained by the fact that the advanced policy follows some pattern that the agent can exploit. Because the advanced policy is less random, the episode games are more similar, so the agent often meets the same states and gets used to them.

Until now, the agent has played the test games against the opponent of his training. He could adapt to his policy and find a specific behaviour to beat it. How would the agent perform against an unknown policy ? We tried to train our agent with the random policy and make him play against the advanced one. The results are shown in **Fig. (7)** and **(8)** and we can see that the agent is actually doing fine. It is not as great as before, but simply by playing against the random policy, the agent has learnt to not be defeated and even to set a trap (see Appendix B for concrete game examples). An explanation is that the random policy has behaved like the advanced one in a few episodes, enabling the agent to learn decent moves.

Fig.(7) : Training with the random policy and testing against the advanced one (agent plays in first)

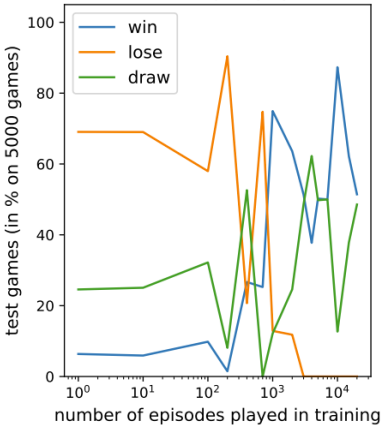
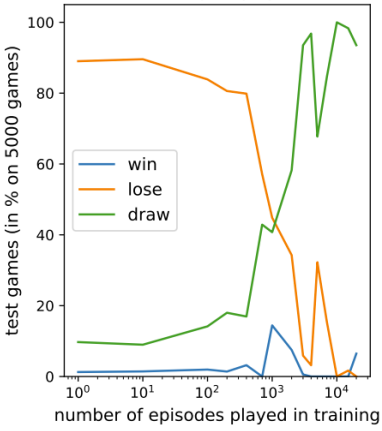


Fig.(8) : Training with the random policy and testing against the advanced one (agent plays in second)



For SARSA, all training were made with the following parameters

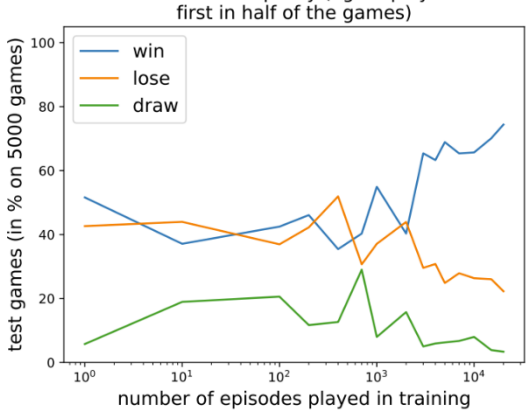
$$\alpha = 0.45, \alpha_{rate} = 0.9995^{\frac{10000}{n_{ep}}}, \varepsilon = 1, \varepsilon_{rate} = 0.9997^{\frac{10000}{n_{ep}}},$$

$$\gamma = 0.7, R_{win} = 11, R_{lose} = 0, R_{even1} = 1, R_{even2} = 1.25$$

with n_{ep} the number of episodes, R_{even1} the reward for a draw if playing first and R_{even2} when playing in second position.

Now we present the results of our model implementing Policy Gradient. The **2D** case is shown in **Fig. (9)**. We can see that the policy improves with more episodes in training, and that it has not converged yet to an optimal policy after 20 000 iterations. Slow convergence is typical from Monte Carlo methods. In 2D, the Q-table approach converges much faster, but 2D is not our main interest for Policy Gradient.

Fig.(9) : Training and testing with the random policy (agent plays first in half of the games)



We show in **Fig. (10)** and **(11)** the results for the **3D** case. When comparing **Fig. (9)** and **Fig. (10)** we can see that the agent overall does better against the random policy in 3D. This is due to the fact that the model architecture that we exploit is quite large for the case of 2D. The goal of implementing the Policy Gradient was to create an agent for n-tic-tac-toe with n > 2, so the model architecture was chosen to improve the winning rate of agent in higher dimensions. The agent struggles a bit more against the advanced policy in **Fig. (11)**, however the global trend is to the improvement, and it suggests better results if we trained the agent more.

Fig.(10) : Training with the advanced policy and testing against the random one (agent plays first in half of the games)

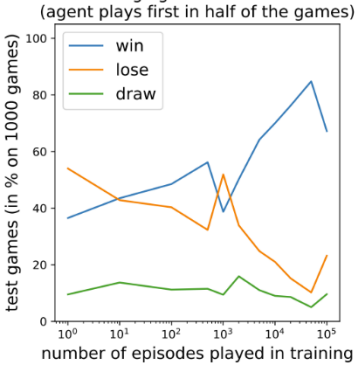


Fig.(11) : Training with the advanced policy and testing against the advanced one (agent plays first in half of the games)

Finally, we trained our agent in 4D with the advanced policy with 100 000 iterations (21 hours of training). The results in **Fig. (12)** and **(13)** show that it is a hard task, especially against the advanced policy. 21 hours of training is still not enough.

Fig.(12) : Agent against random policy in 4D

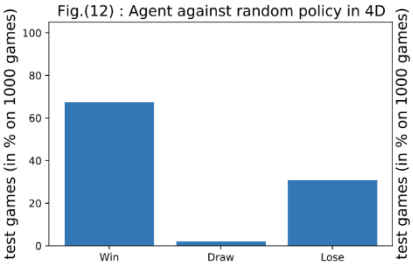
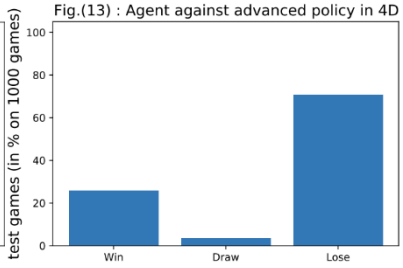


Fig.(13) : Agent against advanced policy in 4D



VI. CONCLUSION AND FUTURE WORK

We have created an environment to play n-tic-tac-toe with all the materials needed to train an agent. It turned out that adapted SARSA with basic Q-tables provides the best agent for the original tic-tac-toe in 2D. The symmetries removal and SARSA modified with reward backpropagation proved to be essential to the success. In 3D, the reward system and the model architecture appeared to be the key. We have seen agent with acceptable win rates with Policy Gradient, however best results requires a huge training time.

The best thing we one can learn is that reinforcement learning proposes algorithms to train an agent, but the most important thing is to understand the environment and adapt the algorithms to it. An interesting outcome is that the agent can be trained with one opponent and performs decently against a different one. In our hypotheses, the opponent is part of the environment. So, having two different opponents correspond to two different environments (same rules but different behaviours). The results we obtained give the hint that an agent trained in a specific environment could perform well in similar-looking environments.

REFERENCES

[1] Reinforcement Learning II - INF581 Advanced Machine Learning and Autonomous Agents

[2] Watkins, Christopher JCH, and Peter Dayan. "Q-learning"

[3] Reinforcement Learning III - INF581 Advanced Machine Learning and Autonomous Agents

APPENDIX A – mathematical details for symbol alignment in arbitrary dimensions

Here, we give the materials for those who wish to implement our dynamic approach to compute the score for n-tic-tac-toe. We have a grid of size k and dimension n, with cells empty or filled by O or X symbols, and we want to compute the current score. The first trick is to replace the X by 1, the O by -1 and empty cells by 0 in the grid. Then, one can sum the values over the axes. For one axis, if the sum is equal to k then X scored one point, if -k then O scored, otherwise no one scored on that axis.

Axes are identified by a tuple of k coordinates of dimension n. For example in a 3x3 grid, ([0, 0], [0, 1], [0, 2]) is an axis. However, ([0, 0], [1, 1], [0, 2]) is not. The challenge is to identify dynamically all tuples of coordinates that make an axis in the grid.

Straight axes are characterized by coordinates similar in n-1 dimensions and different in the remaining dimension. We can look again at Fig. (1 a) and notice that. The cell coordinates are given in the order [d4, d3, d2, d1].

Fig. (1 a) : straight axes

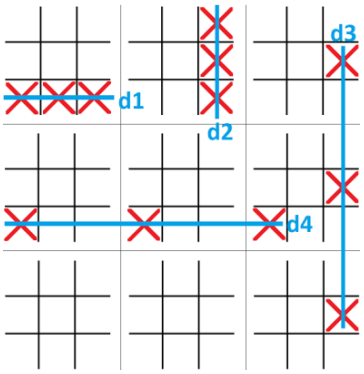


Table (1 a) : Axes identification for Fig. (1 a)

Alignment	Axes
d1	([0, 0, 2, 0], [0, 0, 2, 1], [0, 0, 2, 2])
d2	([1, 0, 0, 2], [1, 0, 1, 2], [1, 0, 2, 2])
d3	([2, 0, 1, 2], [2, 1, 1, 2], [2, 2, 1, 2])
d4	([0, 1, 2, 0], [1, 1, 2, 0], [2, 1, 2, 0])

Indeed for the 4 alignments, the tuples are composed of coordinates which have the same 3 fixed components, and a last one (in red) browsing all the possible values.

Diagonal axes are more complicated. They are characterized by 2 or more dimensions in which the values are browsed forward or backward, and the remaining dimensions have fixed values. It will be easier to look at Fig. (1 b) and notice that phenomenon in the axes identification.

Fig. (1 b) : diagonal axes

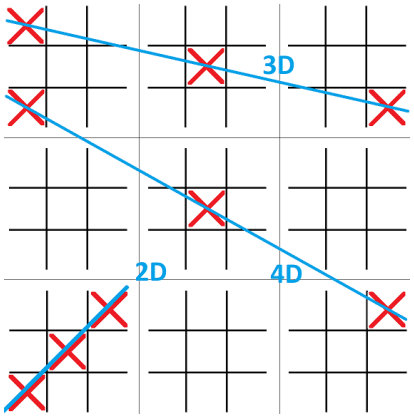


Table (1 b) : Axes identification for Fig. (1 b)

Alignment	Axes
2D	([0, 2, 2, 0], [0, 2, 1, 1], [0, 2, 0, 2])
3D	([0, 0, 0, 0], [1, 0, 1, 1], [2, 0, 2, 2])
4D	([0, 0, 2, 0], [1, 1, 1, 1], [2, 2, 0, 2])

In Table (1 b), we have represented in red the values browsed forward in the tuple components, in green the values browsed backward and in blue the fixed values. To put it simply, just see axes as matrices, like in the tables above. Then, an axis is a diagonal axis if it has at least 2 red columns, or 2 green columns, or one red and one green column, and if the remaining columns are blue (same values).

With this matrix representation, we can easily see that if we have a diagonal axis, all the permutations of his columns will give a diagonal axis. This goes with the huge state space of n-tic-tac-toe.

APPENDIX B – example of games played by our agent

Here we show some games to illustrate the strategies our agent trained with modified SARSA has learnt. Both players are called "Agent", but the X is our agent and the O is the advanced policy.

In **Game (1)**, our agent learnt a trap by playing against the random policy. The trap is to start in the middle and set a double alignment possibility (fork) if the opponent does not play in a corner. With that strategy, our agent gets 50% of wins and 50% of draws when playing first.

In **Game (2)** and **(3)**, our agent was trained also with the advanced policy, and he has learnt an even better strategy for it, with that first move in one corner. The agent can then set up many different traps based on that first move.

Game (1) : Agent X trained with the random policy and playing a game against the advanced policy O

```
Agent plays : (1, 1)

. . .
. X .
. . .

Agent plays : (0, 1)

. O .
. X .
. . .

Agent plays : (0, 2)

. O X
. X .
. . .

Agent plays : (2, 0)

. O X
. X .
O . .

Agent plays : (2, 2)

. O X
. X .
O . X

Agent plays : (1, 2)

. O X
. X O
O . X

Agent plays : (0, 0)

X O X
. X O
O . X

Game over. Score : (1, 0)
```

Game (2) and **(3)** : Agent X trained with the advanced policy O and playing two games against it

```
Agent plays : (2, 2)

. . .
. . .
. . X

Agent plays : (0, 0)

O . .
. . .
. . X

Agent plays : (0, 2)

O . X
. . .
. . X

Agent plays : (1, 2)

O . X
. . O
. . X

Agent plays : (2, 0)

O . X
. . O
X . X

Agent plays : (2, 1)

O . X
. . O
X O X

Agent plays : (1, 1)

O . X
. X O
X O X

Game over. Score : (1, 0)
```

```
Agent plays : (2, 2)

. . .
. . .
. . X

Agent plays : (1, 1)

. . .
. O .
. . X

Agent plays : (1, 0)

. . .
X O .
. . X

Agent plays : (1, 2)

. . .
X O O
. . X

Agent plays : (2, 0)

. . .
X O O
X . X

Agent plays : (2, 1)

. . .
X O O
X O X

Agent plays : (0, 0)

X . .
X O O
X O X

Game over. Score : (1, 0)
```