Can Deep Reinforcement Learning Solve Erdos-Selfridge-Spenser Games?

Ahmad CHAMMA & Hadi ABDINE

29 February 2020

1 Introduction

Deep Reinforcement Learning[1] - which is the combination of deep learning and reinforcement learning - has been the solution to many complex decision-making tasks that were previously not solvable by traditional machine learning algorithms.
Now, using DRL (Deep Reinforcement Learning), machines can teach themselves based upon the results of their own actions that will be rewarded or penalized while learning also data representation. It is one of the areas of AI that achieved human-level prowess in many games.
In the paper, the authors aimed to study the strengths and limitations of the different reinforcement learning algorithms throughout experiences on DQN (Deep Q-Network), PPO (Proximal Policy Optimization) and A2C (Advantage Actor Critic). To perform this study, a family of environments is needed and it must be characterized by:

- Parameters (used to increase or decrease the difficulty of the environment)
- Optimal behavior characterization
- Interaction and multiagent play

The chosen environment to run these algorithms is a game inspired by Erdos-Selfridge-Spencer games that we will talk about it later in the report.

2 Erdos-Selfridge-Spencer Games

2.1 Motivation

Erdos-Selfridge-Spencer (ESS) games are two players games in which each player selects objects from a combinatorial structure in his turn. These games are very interesting for the deep reinforcement learning because, based on the possible random future play, we can find the optimal play strategies using potential functions derived from their conditional expectation.
These games have the following properties:

- Low dimensional and simple parametric environment.
- Rich enough to support interaction and multi-agent play.
- The existing of a linear closed-form solution for an optimal behavior.
- The difficulty of the game can be changed by altering the environment parameters.

2.2 Tenure Game

Among the ESS games, the tenure game which also known as Spencer's attacker-defender game, is used in this article. In this game, and as its name implies, we have two players: attacker and defender, and also a board with K different levels filled with N pieces and it proceeds like this:

1. The attacker proposes a partition of 2 parts: A and B of the current game.
2. The defender chooses one of them to destroy.

3. The pieces in the remaining part move up a level.
4. If all pieces are destroyed, the defender wins.
5. If one piece at least reaches level K, the attacker wins.

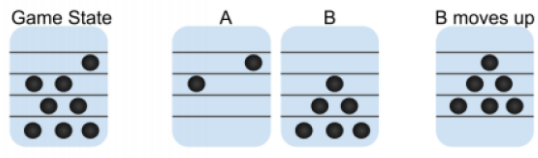

Figure 1: one turn of the tenure game

As we mentioned before, one property of these kind of games is the relation between the environment parameters and the difficulty of the game. In the tenure game, if K (the number of levels) is rather big, then the game will last longer and the possibility to make a mistake will be also bigger. To introduce the notion of potential function later, let's suppose for simplicity that all pieces start at level zero.
Now, we attend to prove theorem 1 in the article which states that if $N < 2^K \implies$ the defender can always wins.
In this case, if we assume that the defender will destroy, the set with higher number of pieces, the number of pieces will be reduced at least by factor of 2 each turn, so if $N < 2^K$, the defender will always win because no piece will reach level K.
In general, each piece in this case has the probability of $\frac{1}{2}$ to advance to the next level (or to be destroyed by the defender), which means that a piece will reach level K with probability of 2^{-K}. So assuming that T_i is an indicator that piece i reaches level K and let T be a random variable for the number of pieces that reach level K, then we will have $E[T_i] = 2^{-K}$ and $T = \sum_{i=1}^{N} T_i$, thus:

$$E[T] = \sum_{i=1}^{N} E[T_i] = \sum_{i=1}^{N} 2^{-K} = N2^{-K}$$

So, if $N < 2^k$ we will have that: $\sum_{l=1}^{K} P(T = l) < \sum_{l=0}^{K} l.P(T = l) = E[T] < 1$
Thus : $P(T = 0) = 1 - \sum_{l=1}^{K} P(T = l) > 0$

In conclusion, there is always an optimal strategy that assures the win of the defender in this case.

```python
import numpy as np
#The number of levels
K = 3
print('The number of levels: ', K)
#The number of pieces
N = 2 ** K - 2
print('The number of pieces: {} < 8 (2 ^ 3)\n'.format(N))
lev_state = np.zeros((K+1, N)).astype(int)
#All the pieces are at level 0
lev_state[K] = np.ones(N)
print(lev_state)
done = False
current_level = K

#1: Present A, 2: Present B
while not done:
    print('\n')
    #Naive attacker/ every piece has the probability of 1/2 to belong to A/B
    count = np.where(lev_state[current_level] == 1)[0]
    A = []
    B = []
    for i in list(count):
        indx = np.random.choice(2)
        if indx == 0:
            A.append(i)
        else:
            B.append(i)
    for i in A:
        lev_state[current_level, i] = 1
    for i in B:
        lev_state[current_level, i] = 2
    print('A: ', A)
    print('B:  {}\n'.format(B))
    print(lev_state)

    #Choosing which set to keep (the set with the low length)
    if(len(A) < len(B)):
        lev_state[current_level - 1][:len(A)] = 1
    else:
        lev_state[current_level - 1][:len(B)] = 1
    lev_state[current_level] = np.zeros(N)
    print('\n {}'.format(lev_state))
    current_level -= 1
    if(np.sum(lev_state) == 0):
        print('\n Defender wins')
        done = True
    elif np.sum(lev_state[0]) >= 1:
        print('\n Attacker wins')
        done = True
```

Figure 2: The implementation of Theorem 1

In this implementation, we supposed the following:

- Each piece has a probability of $\frac{1}{2}$ to belong either to the set A or the set B.
- The defender is destroying the set with the higher number of pieces.

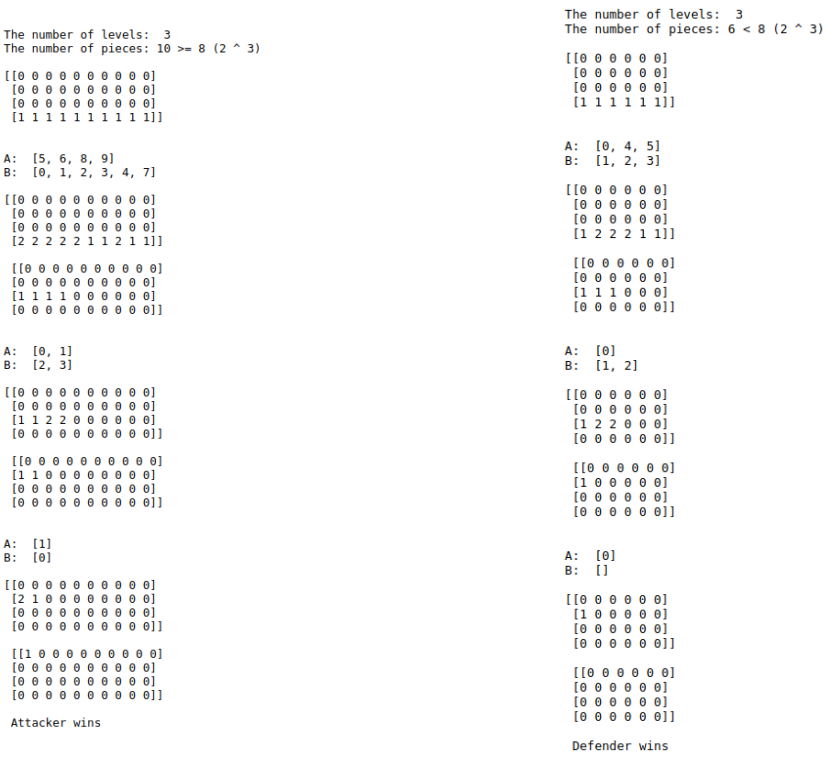

Figure 3: The presentation of two cases for the game with a number of levels = 3. In the first one, the attacker can win based on the initialisation. In the second, we are sure the defender will always win.

By Applying the introduced implementation for 100 iterations, we had the following results:

In the first case with a number of pieces below the threshold in the theorem (2^N), the defender is always winning. However, with a number of pieces higher than the threshold, we can see some iterations or games where the attacker can win.

2.3 Potential Function

The potential function is defined by $\Phi(S) = \sum_{i=0}^{K} n_i 2^{-(K-i)}$ where $S = (n_0, n_1, ..., n_K)$ is (K+1)-dimensional vector that describes the state of the game with n_i the number of pieces at level i and supposing that the game has (K+1) levels.
The value of the potential function is based on the fact that, a piece at level i has probability of $2^{-(K-i)}$ to survive in random play, and that we have n_i pieces at this level.

The potential function Φ is very important because it gives an important information about which player is more favorable to win, and also how much error in the optimal strategies.
Using the potential function of the initial states S_0, we can generalize the theorem 1. proved earlier, and this generalization is stated in Theorem 2. in the paper which states that:

- If $\Phi(S_0) < 1$, the defender can always win. (1)
- If $\Phi(S_0) \geq 1$, the attacker can always win. (2)

The proof of (1), is the same as the proof we did before for Theorem 1. . We can deduce in the same way that $P(T = 0) > 0$ and this will give us that under this condition there is always an optimal play that allow the defender to win. In fact, at each turn, if the defender destroys the part with the higher potential, the second part will surely have a potential lower than 0.5, and by advancing one level the new potential will surely be lower than 1 and so on.

The second part of the theorem is proved by the fact that, if the attacker can always choose the two parts A and B such that $\Phi(A) \geq 0.5$ and $\Phi(B) \geq 0.5$, then the remaining part will always have $\Phi \geq 1$ because the potential of the remained part will be multiplied by 2 after advancing one level, and this is proved using greedy algorithm in [2].

The environment in this game allow us to characterize step by step optimal strategy and this allow us to find the exact place of the errors in addition to the measure of the reward based on the win/loss. Also, it allows us to train the defender and attacker, separately or together in multi-agent and in self-play.

3 Deep Reinforcement Learning on the Attacker-Defender Game/Implementation

In order to test the different ideas presented in this paper, we need to build the corresponding environment described. Usually, when working with Reinforcement Learning Environments, OpenAI Gym is the python package the most used for this purpose. However, the ESS game environment is not included in the package. Thus, the need to implement it. Fortunately, this python package has the ability to include new environments (some tutorial by the author). The explanation will include: Trained Defender, Trained Attacker, and Trained Defender vs Trained Attacker.

Some common properties for the environments are:

- The two parameters used will be:
  - K $\rightarrow$ The number of levels.
  - $\phi(S_0) \rightarrow$ The initial potential.
- The game state is represented by a K + 1 dimensional vector for levels 0 to K.
- For the defender agent, the input is the concatenation of the partition A, B, giving a 2(K + 1) dimensional vector.
- The start state S_0 is initialized randomly from a distribution over start states of a certain potential.

3.1 Training a Defender Agent on Varying Environment Difficulties

As mentioned above, the defender role is to destroy the set having the highest potential function. The action state in this environment will have 2 values: Destroy A or Destroy B.

The implementation of the defender environment requires multiple functions:

- The initialisation of the environment with the corresponding number of levels and the initial potential (chosen lower than 1 to verify Theorem 2).
- In order to build the environment with the pieces placed arbitrary in the different levels, a random start function is used to always measure the possible difference between the potential reached so far (by adding a piece at a random level) and the initial potential with the ability to reduce the size of the interval of levels.
- The potential function used to compute the sum of the inner product between the given state and the weights computed in the initialisation.

- In order to train the defender, we need an attacker characterized by the fact of randomly choosing between (mostly) playing optimally, and (occasionally) playing suboptimally for exploration (with the Disjoint Support Strategy):
  - With optimal strategy, the attacker tries to find the partition that results in sets A and B as equal as possible.
  - With Disjoint Support Strategy, the attacker greedily picks A, B so that there is a potential difference between the two sets, with the fraction of total potential for the smaller set being uniformly sampled from [0.1, 0.2, 0.3, 0.4] at each turn (As described in the paper). In our implementation, we took a multinomial sampling over $[\frac{1}{3}, \frac{5}{16}, \frac{14}{32}]$ at each turn.
- Another 2 functions for splitting between the two sets A, B and for erasing the chosen set to destroy.

To train the defender, we use a fully connected deep neural network with 2 hidden layers of width 300 to represent our policy. Here, we used another new modified python package called **stable baselines**. This package has the necessary deep reinforcement learning algorithms implemented that the authors used in their paper:

- **DQN** (**Deep Q − Learning**):

  When talking about the reinforcement learning environment, we should specify two keywords:
  - State s
  - Action a

  A reinforcement learning agent is required to define the action to take when facing a specific state. So, facing the state-space (the set of all possible states) and the action-space (the set of all possible actions to take) we could build the Q matrix (Quality measuring matrix). Therefore, for each pair (state, action), we assign a scalar expressing the quality of the action toward the state called **Q − Value**.

  The Q-values can be approximated iteratively by applying **Bellman equation**:

  $$Q(s, a) = r + \gamma max_{a'} Q(s', a')$$

  where $\gamma \in [0, 1]$ s called the discount factor and s' is the future state.

  So at each step in a given episode, facing the state s, the agent will choose the action having the higher Q-value. Then, the update will take effect when computing the sum of the reward and the Q-value of the new state maximized over the actions.

  Now storing the Q(s, a) values is not always possible as the number of states in a lot of applications is much higher than what a memory can handle. This is why **DeepMind** introduced their Deep Q-learning algorithm that will train a neural network instead of fitting the Q table in memory.

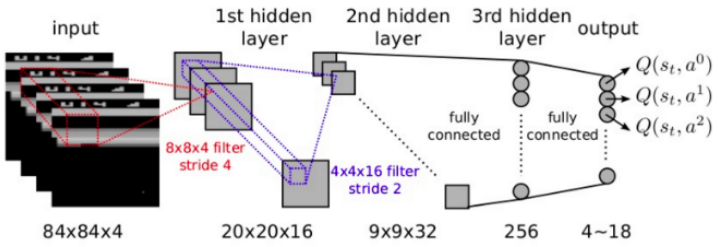

Figure 4: DQN trained with raw pixels from Atari 2600 games using convolutional networks

The network will have the current state as input and all possible Q-values (actions) as output. The model will try to optimize the loss function defined as:

$$L = (Q(s,a) - r - \gamma max_{a'} Q(s',a'))^2$$

On a higher level, Deep Q learning works as such:

- First, it is characterized by two neural networks: policy net and target net.
- We introduce the state as an input for the policy net and we perform the optimization of the Loss function given earlier.
- However, the loss function needs $max_{a'}$ Q(s', a') which will be computed using a cloned net (by loading the weights of the policy net after a defined number of steps).
- The training of the policy net is performed using batch size of examples called experiments stored in a replay memory.

- **PPO** (**Proximal PolicyOptimization**):

  The main difference between Q-learning and policy methods is that while Q-learning tries to approximate a Q function that maps each state and action to a scalar, proximal methods try to find directly the policy by minimizing a loss function that looks like

  $$L(\theta) = log(\pi_\theta(a_t|s_t)A_t)$$

  where π_θ is a stochastic policy and A_t is an estimator of the advantage function at time-step t. Optimizing this loss proved to be not always sufficient so the PPO replaces this loss with another one defined as a combination of CLIP loss (cf. reference [3]) and the squared error loss.

  Explaining the theoretical properties of this optimization problem requires a separate article to be explored in details.

- **A2C** (**Advantage ActorCritic**):

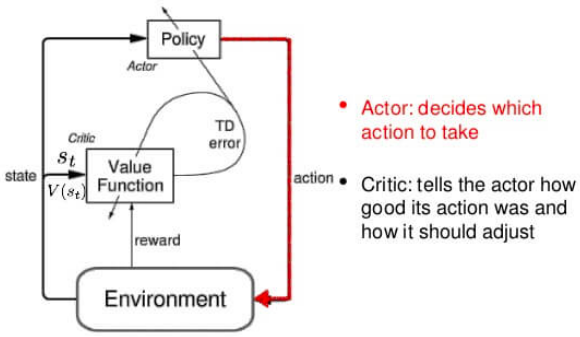

Figure 5: Actor-Critic

What is Advantage? Q values can, in fact, be decomposed into two pieces: the state Value function V(s) and the advantage value A(s, a)

$$Q(s,a) = V(s) + A(s,a) \Longrightarrow A(s,a) = Q(s,a) - V(s) \Longrightarrow A(s,a) = r + \gamma V(\hat{s}) - V(s)$$

Advantage function captures how better an action is compared to the others at a given state, while as we know the value function captures how good it is to be at this state.

Instead of having the critic to learn the Q values, learning the advantage values will make the evaluation of an action based not only on how good the action is, but also how much better it can be.

The advantage of the advantage function is that it reduces the high variance of policy networks and stabilize the model.

In the paper, there is a comparison between two approaches:

- **Training the Defender using LINEAR model**:

  As the optimal policy characterized by the potential function is linear, the choice of the linear model is thought to be theoretically expressive to learn the optimal behavior of the defender. The results with this model weren't good enough with both PPO and A2C but with DQN, there was a surprising performance. When trying deeper models, the algorithms showed a great improvement and then a better performance since the reward scores reached the optimal score around the value 1 relative to the optimal behavior. In a nutshell, the linear models were theoretically thought to work well but ended up by being empirically outperformed by deeper models in terms of performance improvements and variance reduction.

- **Training the Defender using DEEP model**: All algorithms show variation in performance across different settings of potentials and K, and show noticeable drops in performance with harder difficulty settings. A2C shows the greatest variation and worst performance out of all three methods.

Now, we tried to reproduce some results given in the paper. We implemented the defender environment as explained and the deep model with the 3 mentioned algorithms:

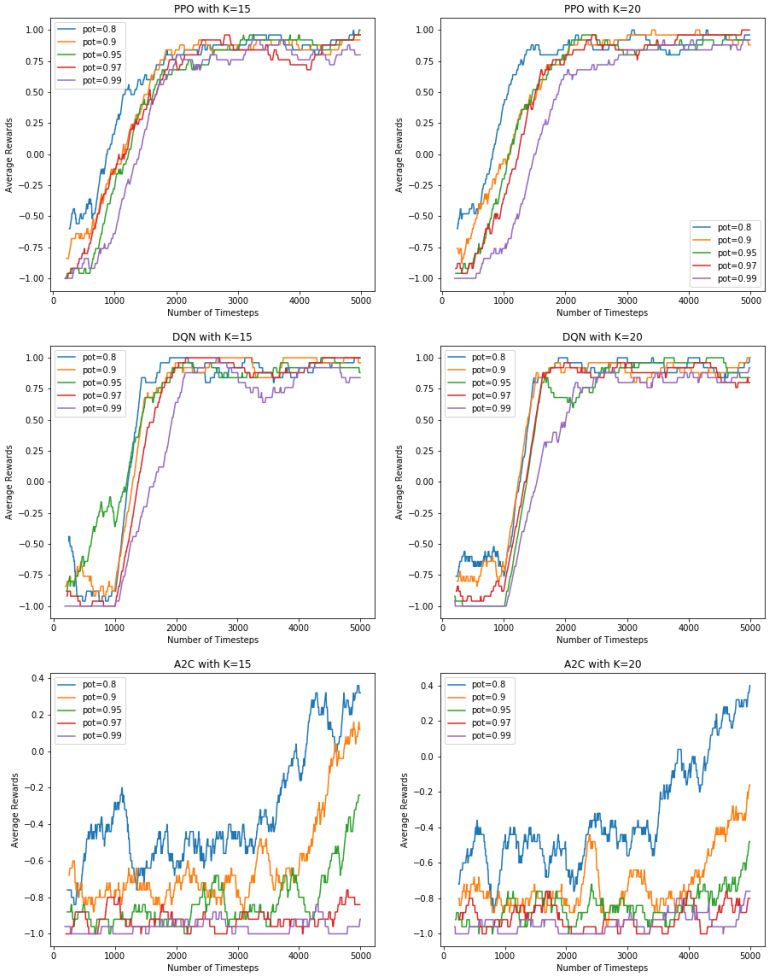

Figure 6: The Defender agent trained using a deep model (2 layers, 300 each). PPO, DQN and A2C used with two values for K (15, 20) and 5 values for ϕ_0 (0.8, 0.9, 0.95, 0.97, 0.99)

We can see clearly the fact that A2C is the algorithm having the worst performance (given 5000 as number of episodes). Also, both DQN and PPO are presenting remarkable performance with a faster learner for a lower value for the initial potential.

4 Generalization in RL and Multi agent Learning

The generalization concept can be seen as the idea of considering different configurations of the game including the ways with which the agents interact. The defender was trained with an optimal attacker. After that, the deep model was tested with an optimal defender(the result of the training) against a sub-optimal attacker applying the disjoint support strategy.

As a consequence, compared to the optimal configuration where both the agents are optimal in training and evaluation, the disjoint support configuration described before has lead to a lower performance score. In other words, performance is higher with optimal attacker. Since the generalization consisted in using a non-optimal attacker, then we come to the conclusion that the RL agent fails to generalize on new environments or configurations.

Before Talking about the approaches used to generalize, we will talk about the difference between single and multi-agent settings. In the single-agent scenario the environment changes solely as a result of the actions of an agent. However, in multi-agent context, the environment is subjected to the actions of all agents. The complexity of multi-agent Reinforcement Learning scenarios increases with the number of agents in the environment.

With the environment of the Attacker-Defender game, the single-agent setting is when the state of the game changes as a result of an action A_t of one of the agents: the Attacker or the Defender.

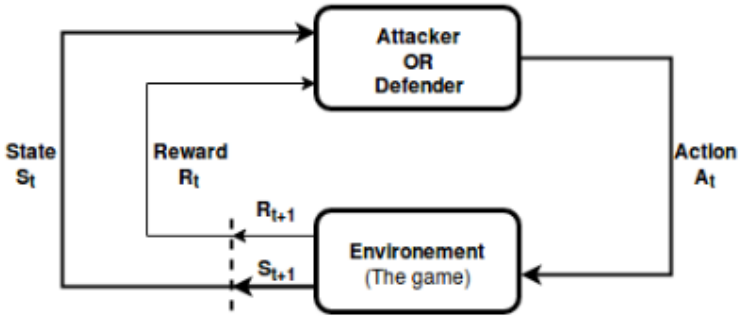

Figure 7: Single-Agent setting

On the other hand, the Multi-Agent setting is when the game's state changes after taking into consideration both the Attacker"s action and the Defender's action A_t.

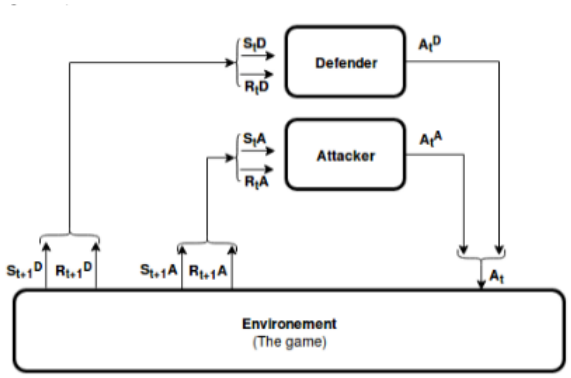

Figure 8: Multi-Agent setting

So, The RL agent failed to generalize. The generalization approach is discussed within:

- **Training an Attacker Agent**:

  The main action, done by the attacker during the game, consists essentially in knowing how to divide the pieces into two sets in such a way to guarantee its winning. One first naive approach proposed by Spencer was given by the policy outputting for each level the number of pieces that should be allocated to the set A and thus the rest would correspond to the set B. In this context, the action space would grow exponentially with the number of levels K since the more levels there are, the more allocations will be done. Obviously, this is an expensive implementation to go for.

  To perform the training of the attacker, a new theorem was introduced which states that for any Attacker-Defender game with $\phi(S_0) \geq 1$, there exists a partition A,B such that $\phi(A) \geq 0.5$, $\phi(B) \geq 0.5$, and for some level l, A contains pieces of level i ¿ l, and B contains all pieces of level i ¡ l.

  This theorem is important in building a learned Attacker. In fact, using this theorem, the environment assigns all pieces before level l to A, all pieces after level l to B, and splits level l among A and B to keep the potentials of A and B as close as possible. So, this theorem guarantees the optimal policy is representable, and the action space is linear instead of exponential in K.

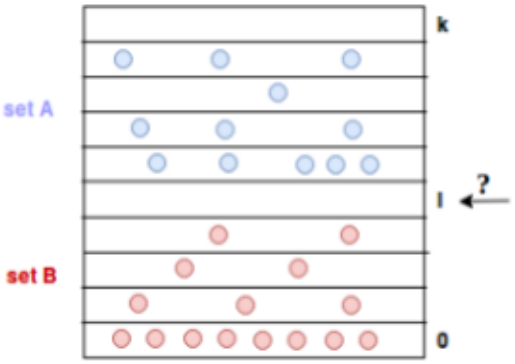

Figure 9: Training the attacker (new Theorem)

- For the implementation of the training of the attacker agent:
  - The initialisation of the environment of the attacker with the fact that the action space will have K levels (the last level is not supposed to be integrated in the partitioning).
  - Again, we need a random start function to initialize the environment with the pieces arbitrary placed in the different levels to obtain the initial potential.
  - A defense play function for the opponent of the attacker willing to destroy the set with the highest potential (optimal defender).
  - A split function that applies the theorem described above to find the level l needed to perform the partitioning.

We tried to reproduce some of the results in the paper by performing the training of the attacker using the 3 algorithms (PPO, DQN and A2C) with different levels (5, 10, 15, 20 ,25) and 2 values for the initial potential (1.1, 1.4)

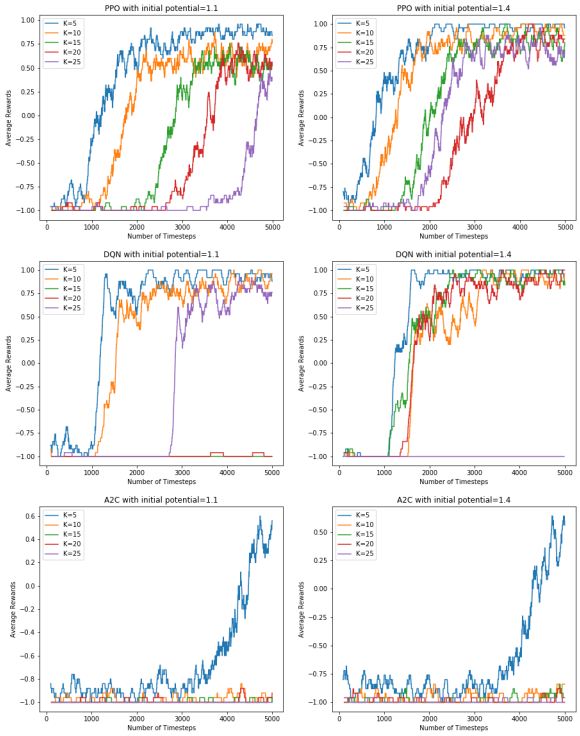

Figure 10: The Attacker agent trained using a deep model (2 layers, 300 each). PPO, DQN and A2C used with 5 values for K (5, 10, 15, 20, 25) and 2 values for $\phi(S_0)$ (1.1, 1.4)

Here, we can see the fact that A2C again is presenting the worst performance among the 3 algorithms. Also, when the number of levels is increasing, the attacker is struggling more in order to win (the difficulty of the game is increasing).

Finally, After training the attacker and the defender, we integrated both agents in one game for different values of the initial potential and the result was always the same as the Theorem 2.

5 Self-Play

Rather than using two different neural network to train the attacker and the defender, we can use only one. In fact, we note that both the attacker and the defender rely their strategy over the potential of sets. Therefore, we can implement a neural network which determines which set has a higher potential than the other. It will allow the defender to destroy the set with the highest potential. Moreover, by binary search, it will allow the attacker to find two balanced set in term of potential.

The key insight is the following: both the defender's optimal strategy and the construction of the optimal attacker in Theorem 3 depend on a primitive operation that takes a partition of the pieces into sets A, B and determines which of A or B is "larger" (in the sense that it has higher potential). For the defender, this leads directly to a strategy that destroys the set with the higher potential. For the attacker, this primitive can be used in a binary search procedure to find the desired partition in Theorem 3: given an initial partition A, B into a prefix and a suffix of the pieces sorted by level, we determine which set has higher potential, and then recursively find a more balanced split point inside the larger of the two sets.

N.B : In the Notebook, we performed a sort of visualisation for the different scenarios concern-

ing the alternating roles, the partitioning, the destroying of the chosen set and the announcing of the winner. (We will present here some screenshots).

6 Extra Implementation

After performing all the above work, we thought of comparing the 3 mentioned algorithms together with the same number of pieces and the same initial potential and for a number of episodes = 10000. It is clear that PPO and DQN are learning very fast to master the game (in a less number of episodes) where A2C needs 10000 episodes and above to master the game. This schema may change based on the desired difficulty of the game.

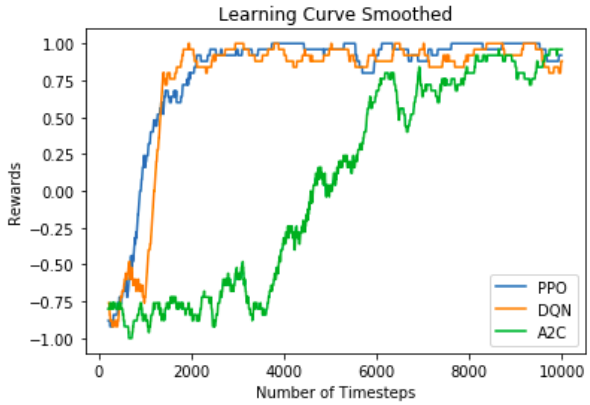

Figure 11: Comparison between the three algorithms PPO, DQN and A2C used to train the defender agent

Also, here we are showing a small scenario of the training of the defender with an implemented visualization to follow the game and the choices of the two agents (partitioning and destroying).

7 Conclusion

In this paper multiple Deep Reinforcement Learning algorithms such as DQN, PPO ans A2C, were used to train the attacker and the defender in most interesting and suitable game to test the strengths and limitations of the deep algorithms with handling reinforcement problems. The Erdos-Selfridge-Spencer game was a good example for an environment for this study.

The paper showed what a Deep Reinforcement Learning algorithm can do where an optimal policy can be visible (where in other environments, it is complex). With an environment characterized by a varying difficulty (depending on the parameters) or a favorable agent to win (depending on the initialisation), each agent was struggling more or less to master it's role and get the higher possible reward (single-agent or multi-agent).

As the paper, and mention that was not easy to re-implement the algorithms since no details on their implementation where provided, we tested as it is shown in the notebook multiple scenario and algorithms on the mentioned environment, from training the defender to the training of the attacker with a visualisation of the scenario of random games (displaying the moves of the two agents). In addition to testing the trained attacker/defender with a number of episodes to get the reward at each episode (to display their performance across the different episodes). Last, we included both the defender and the attacker (both trained) in one game to verify the theorem 2 already mentioned.

Finally, we can say that yes the Deep Reinforcement Learning algorithms can solve ESS game but it's important to pay attention to the architecture of the neural network and tuning it's hyperparamters.

References

[1] Riashat Islam Marc G. Bellemare Vincent François-Lavet, Peter Henderson and Joelle Pineau. An introduction to deep reinforcement learning. 2018.

[2] Joel Spencer. Randomization, derandomization and antirandomization:three games. In *Theoretical Computer Science*, 1994.

[3] Prafulla Dhariwal Alec Radford John Schulman, Filip Wolski and Oleg Klimov. Randomization, proximal policy optimization algorithms. In *https://arxiv.org/pdf/1707.06347.pdf*, 2017.