GUIDED REINFORMENT LEARNING WITH MONTE-CARLO TREE SEARCH ALGORITHM

Reinforcement learning –

In reinforcement learning the algorithm has to learn from experience. This type of learning is usually in a dynamic environment where the algorithm has to perform necessary actions that will maximize the reward. It is different from supervised learning as it does not the answer key to the right answer and instead the reinforcement agent performs the given task.

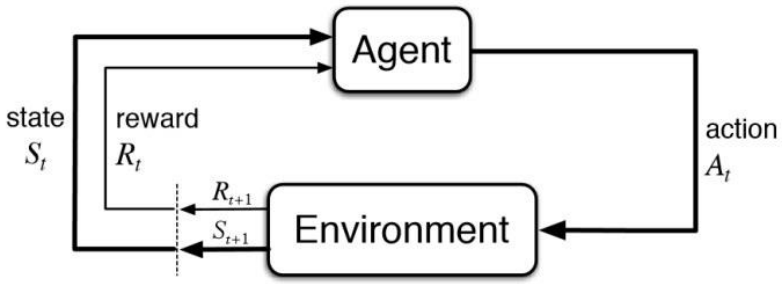

Fig 1.1: Reinforcement learning

Reinforcement learning algorithm follows the above diagram during execution. Every action A_t that the agent performs on the environment at a certain time t will result in a new state S_{t+1} and a subsequent reward R_{t+1} is determined.

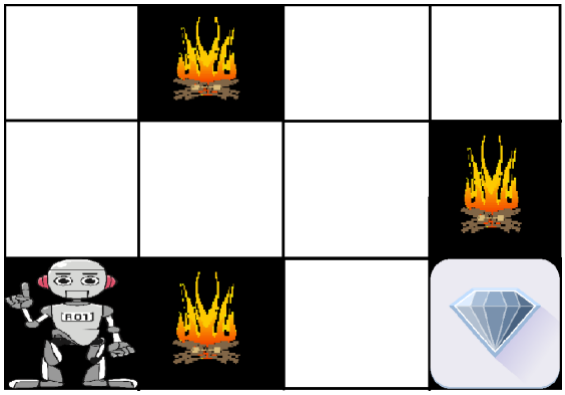

Fig 1.2: Example of Reinforcement Learning

The above image shows a simple game involving robot, diamond and fire. Suppose we need to implement reinforcement learning on this game to determine the best possible way to reach the diamond without coming across fire. Henceforth the robot explores all possible paths and thereby learns which route is the most optimistic one while avoiding fire.

Positive Reinforcement Learning

Positive reinforcement learning is a variant of RL in which we encourage certain behavior of the model. This variant is used to explore the best performance of a task as the model thrives to reach

the maximum reward possible. Also, positive reinforcement leads the model to make more sustainable changes, which can become persistent patterns and persist for long periods of time.

Negative Reinforcement Learning

Negative reinforcement learning is the inverse implementation of positive reinforcement learning. In this method the learning algorithm is encouraged to maintain a certain minimum rather than reaching the model's maximum reward.

Deep Reinforcement Learning

Deep reinforcement learning combines artificial neural networks with a reinforcement learning algorithm. It will enable software-designed agent to learn which actions will optimize the reward in a virtual environment. This is accomplished by deep reinforcement learning which unites function approximation with target optimization. When neural networks are combined with reinforcement learning the resultant artificial intelligent agents produced can showcase far more superior performance than the best of the humans. AlphaGo is a clear example for this. This is proven even further when reinforcement learning techniques were used for training models not just in 'GO' but in several Atari video games.

Using RL we can solve the problem of correlating immediate actions with the delayed returns they produce. Reinforcement learning algorithms sometimes have to wait until they get the desired reward. It happens because some actions will result in no immediate reward but will help in reaching the maximum reward possible as they exist a delayed return environment. We also have to consider the fact that the total possible cases in an environment will exponentially grow. This is why it takes a lot of time for reinforcement learning algorithms to train themselves to be the best.

From reinforcement learning, the concepts of Agent, Environment, Reward and Action is understood. In order to define how a deep reinforcement learning algorithm works we need to understand a few more concepts. They are –

Policy (π): Policy is the strategy employed by the deep reinforcement learning algorithm which determines the next action that is most suited in a certain state. Policy is used to map states to actions, actions which will optimize reward.

Value (V): Value of a certain reward is the expected long-term return. In deep reinforcement learning the algorithm takes into consideration the value rather than the reward before making a decision of which action to perform. The example in discount factor showcases the difference between reward and value. It is represented as **Vπ(s).**

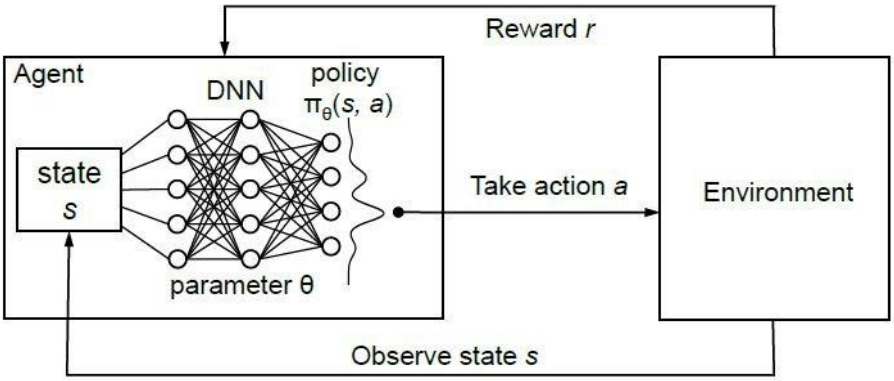

Fig 1.3: Deep Reinforcement Learning

Q-value or action-value (Q): Q-value is similar to Value except that it is specific to a certain state. **Qπ(s, a)** is the value of the long-term return that can be obtained when we perform an action a when in state s under the π policy. Q maps states to actions. Consider a scenario where we are in a state S_0. We need to select what our next action is going to be, and we have four different choices to choose from. Let us name them as a_1, a_2, a_3 and a_4. In order to find out which action is preferred we calculate the Q-value for each of these actions at the state S_0. We then implement the action which corresponds to the highest Q-value.

Policy Gradient in Reinforcement Learning

The goal of reinforcement learning is to achieve the maximum reward possible. The policy gradient methods target at modeling and optimizing the policy directly. We usually model the policy with respect t to a parameterized function θ, such that policy is obtained as $\boldsymbol{\pi_\theta(a|s)}$.

Algorithms usually depend on this policy to find the value of the reward and various algorithms can be applied to optimize θ to gain maximum rewards.

The reward function is defined by -

$$J(\theta) = \sum_{s \in \mathcal{S}} d^{\pi}(s) V^{\pi}(s) = \sum_{s \in \mathcal{S}} d^{\pi}(s) \sum_{a \in \mathcal{A}} \pi_{\theta}(a|s) Q^{\pi}(s, a)$$

Fig 1.4: Reward function of Policy Gradient in RL

Where dπ(s) is the stationary distribution of Markov chain for πθ

Asynchronous DQN

The goal of these algorithms is to develop deep neural networks using reinforcement algorithms such that the resultant neural networks are reliable and do not require large resources. This is accomplished by creating multiple threads on a single CPU machine. Through this we will utilize the computational resources to the maximum capacity. Also keeping them on a single machine instead of dividing them helps us in reducing the communication costs which occur while sending gradients and parameters. Another advantage of running multiple learners in parallel is that each of them will be interacting with different parts of the environment. The diversity can be maximized by employing unique policies in each of the learners.

The four variants in which asynchronous DQN can be applied are –

1. Asynchronous one-step Q-learning
2. Asynchronous one-step Sarsa
3. Asynchronous n-step Q-learning
4. Asynchronous advantage actor-critic

AlphaGo

AlphaGo is a computer program that was designed to play the board game Go. AlphaGo was designed based on reinforcement learning and it has arguably the strongest Go player in the world. Though later versions of AlphaGo were even powerful, it was the first computer program to win in a full 19 x 19 game against a 9-dan professional.

All previous computer programs could only play against the level of amateur players. Standard players which use only tree-search methods to test all possible moves could not cope up with the

complexity of Go. The complexity of Go is such that there are 10^{170} moves that are possible. This makes it several times more complex than chess. The "policy network" layer decides which is the next best possible move. AlphaGo was initially trained against amateur level humans to introduce the way humans play. AlphaGo has data collected from 6000 different opening which it has gained by analyzing 230,000 human games. Each of these games were analyzed with 10 million simulations by AlphaGo Master. AlphaGo Master is the online version implemented to collect and analyze more data.

TREE SEARCH ALGORITHMS:

MCTS:

It is a heuristic tree search algorithm which uses randomness for problems which are deterministic, which are really hard or really impossible to crack with other approaches. It is a search algorithm which uses min-max criteria with expected-outcome based model on episodes of game randomly till the end. It is one such algorithm which does problem-solving with reduction in search times exponentially when compared to BFS, DFS, iterative deepening etc., According to the creator of this algorithm, he told that this expected-outcome based algorithm is "**is shown to be precise, accurate, easily estimable, efficiently calculable, and domain-independent."** It was experimented in depth with games like Orthello, Chess which provided with really spectacular results when used in games by RL algorithms like AlphaGo.

Facts:

- This algorithm which we are using is AlphaZero which is actually invented by DeepMind, now owned by google but is different by addition of a comparison.
- AlphaGo Zero is better than AlphaGo and AlphaZero is better than AlphaGo Zero
- AlphaGo is the algorithm used to beat the world champion of the game called Go.
- Need a lot of computational power to process and generate the training data.
- AlphaGo learns by the opposite real time player's gameplay but AlphaZero is self-taught.

EMPERICAL ANALYSIS:

Why this project?

We wanted a project where we could have our hands on some actual AI implementation. This project was inspired from the movie Ex-Machina where the robot learns and behaves human. Thinking about the complexity of such level AI, this project is a minuscule but a worthwhile step in a right direction. We wanted the project to be different and unique from the ML (prediction, comparing accuracy) schema, meaning that it should have something different, apart from only tuning the project based on ML parameters and better datasets for better results.
It might be a bit on large on theoretical side with some results, but we feel it's important to explain our project in-detail. Thanks!!!

Project:

Here, we would like to deal with the concept which helps in reducing the exploration space, reducing the complexity of which policy to go for in Reinforcement learning. For this before getting into the actual topic, Let's know a bit about the thinking process of humans first. Generally, when playing a game for the first time we may lose but upon playing we will be able to think and plan about our next move based on the opponents upcoming move apart from learning from mistakes. This ability to think and plan ahead is our intuition. So, our mind is something like a combination of intuition and learning.

Now with the advancement in human like intelligence, machines are able to learn and imitate humans. They use the concept of machine learning like reinforcement learning to learn about the environment through agents and actions performed or to get to a successful state where they can beat humans given the conditions of winning.

So, now think about the possible exploration space needed for the RL algorithm to learn, to come to a human level. In real world, the possible exploration space is expected to be filled with infinite points for the agent to choose from. So, given such limitless space consider the number actions they need to perform on a single object to learn about the handling of such one object.

For example, consider a game, given a learning capability to the machine, let's say the machine learnt how to play the game. Even now the knowledge of the machine is limited to learning from its possible previous reinforcement learning network of the game. We believe the system still lacks the intelligence to plan the next move i.e., intuition from the option of moves available because it always chooses the best possible move (Min-Max) every time to win the game. Though it wins the game there's still a pattern in it.

Now let's induce the intuition i.e., planning capability for the machine to make it a wholesome human. For this, we considered inducing a tree-search algorithm (Monte-Carlo tree-search algorithm in this case) into the DRL/DQL (Deep Reinforcement Learning)

Why Tree search algorithm + DQL?

In general, we would implement a board games like tic-tac-toe as in homework with DFS on a tree (with possible states) where we would actually choose the move which results in opponent getting the lowest score every time. Does that imply we actually solved the problems of all 2-player games?

That's not quite the case because we have to go through all the possible states that might be the possible next moves to compute the next state's value. So, even though this looks easy to there might be cases for games like chess, large connect, Go type of games where we have to go through their titanic game trees which would be impossible to find them back.

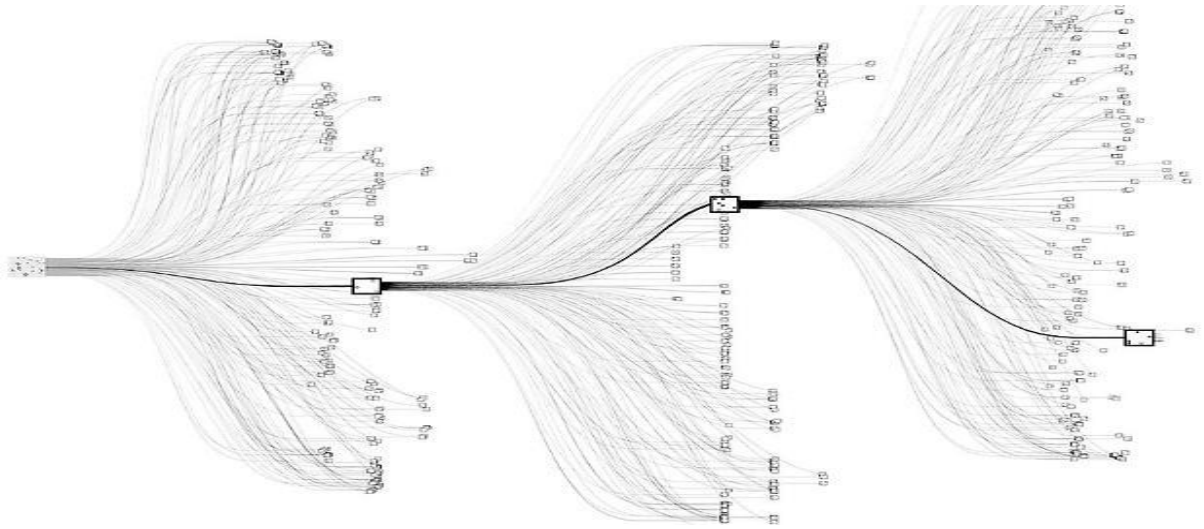

Fig 2.1: Complexity of game trees

In current DQL algorithms like DQN and Policy gradient we won't have any lookaheads so we won't be having the intuition like feature which we were talking about in humans before, which will actually help the algorithm by guiding them in making stronger policies.

Let' say we want to create a heuristics-based learning system, think about the number of players and the number of games it has to play with them, it is the impossible part as it can't learn the thoughts of every human to win every time. (Might be a bad decision !!!)

So, for the intuition feature consider adding the Monte Carlo tree-search algorithm, this will actually help us in the planning phase of the algorithm before taking decisions. Also, the usage of action guidance helps to improve the sample efficiency in conditions where we have to deal with sparse, delayed rewards. This kind of policy will help us to converge faster to making choices, thus making faster decisions and dealing with less exploration space in the network altogether.

Implementation of MCTS + DQL:

So, we are building and implementing a 3x3 tic-tac-toe board game. It itself needs a lot of time to generate the data we need in the future.

Here for this algorithm we are generating training data by writing a script called self-play, where we have 2 files. The first file generates training data and plays the game with the generated data we have, with the second file just like re-evaluating the mechanism. Now, if the performance of the file we have is greater than 55% (threshold) we would continue with the policy and data generated else we would replace it with the new policy and probability values of the second file (which we assume would be best as it's better than the current best).

Here the main goals of our algorithm are as follows:

- To decide which move to make at every point of gameplay i.e., the learning policy.
- To decide on what the learning value should be at every point i.e., the immediate reward in general terms.

Self-play:

We have self-play designed because usually we had to give favorable options for the algorithm to choose from, thus making the learning heuristics-based. But, now with self-play the algorithm will be learning without any previous knowledge other than just the rules of game and we explore the data randomly for the estimation of state's value.

Let’s start with how the training data is generated i.e., how self-play works:

- We are using a Deep Q Neural network to generate this data.
- Move selection is decided, based on the MCTS.
- Then we are going to have a variable to save all the following information at every point of decision making:
  - The state of the game
  - The probabilities while making the search, which is further used to take the best of the possible nodes depending on the range of probabilities to make the correct decision.
  - The information about the winner.

Working of MCTS in Tic-Tac-Toe:

The MCTS during the self-play creates a tree of nodes each node creating a the same 3x3 replica of the game board with possible chances of playing.

For easy explanation, please look the image below.

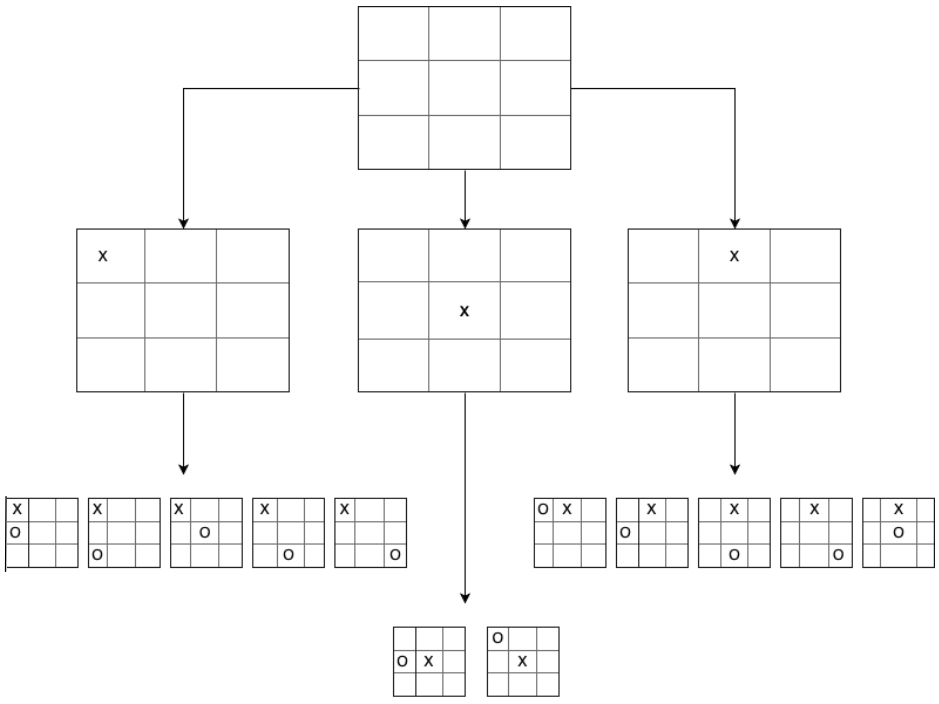

Fig 2.2: Visualization of MCTS

*Couldn’t show the all the possible states due to lack of space, so we skipped the repetitive patterns

Here, the root of the tree indicates a tic-tac-toe node which is empty, the first level shows the possibilities of making a move by player 1. Similarly, level-2 shows the possible states that can be achieved based on the player in respective level-1.
Generally, we could explore all the possible combinations of the available exploration space, but it will be computationally very expensive, so we could solve this problem using the clever technique of MCTS and focus our choices to a small number of best possible nodes to minimize the exploration.

For choosing the action during the MCTS self-play we use the Upper Confidence Bound (UCB). For every state-action pair we encounter the UCB is calculated and the state-action pair with highest values is selected.

$$U(s,a) = Q(s,a) + c_{puct} \cdot P(s,a) \cdot \frac{\sqrt{\Sigma_b N(s,b)}}{1 + N(s,a)}$$

Fig 2.3: Formula for UCB

Where,

- N(s, a) = number of times an action (a) has been taken from state (s)
- $\sum N(s, b)$ = Summation of rewards gained after taking action (a) from state (s)
- Q(s, a) = The quality of taking action (a) from state (s).
- Computed by dividing W(s, a) by N(s, a).
- P(s, a) = Initial probabilities predicted by the neural net (will explain more in depth)
- cpuct = A tunable hyperparameter. The larger this number, the more the model explores

It's like visiting your favorite Cookie shop, every time you will be choosing only your favorite variety cookie but for a change you would like to try a different one. So, now you have different options to choose from.
Let's see how the formula works. Referring to the expressions above (Fig 2.3), your favorite action here will be $Q(s,a)$ that the self-play takes. Now the $\sum N(s, b)/1 + N(s, a)$ helps in making a different choice. This randomization in choosing nodes happens in a condition when the self-play hasn't

chosen new nodes in a while. At this point the numerator $\sum N(s, b)$ increases by value and the $N(s, a)$ still being a small number in the denominator, which means a large value will always be added to the specific action of $Q(s,a)$ making it an have a large upper bound.

The term $P(s,a)$ is the initial probabilities of the confidence of choosing an action **a** by state **s**. This term helps in choosing the decision, specially at times when the self-play has been through the state only a couple of times. If the node is visited multiple times then the variables Q and N help in overcoming this problem with $P(s,a)$.

In this way the algorithm goes down until it reaches the leaf node which means the node travels through the nodes which it has never visited before. The NN now calculates and records the policy of the state which is used in UCB calculation at every state; the value which is actually propagated again back through all the states which actually led to reaching this leaf node.

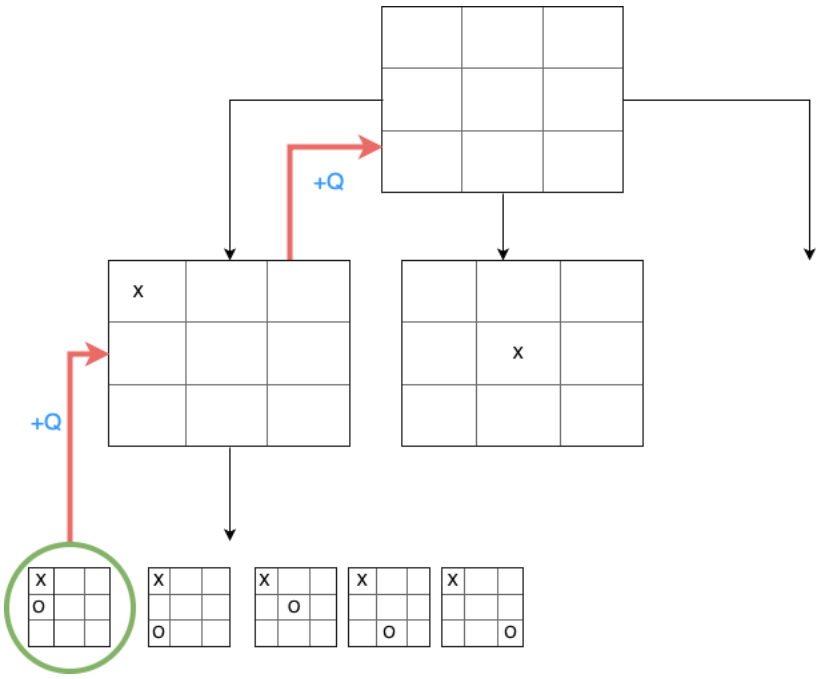

Fig 2.4: Visual view of how the Q value propagates to the above nodes.

This algorithm runs from start to end i.e., if a player wins a game it reaches leaf node. Such one cycle of gameplay is called an episode.

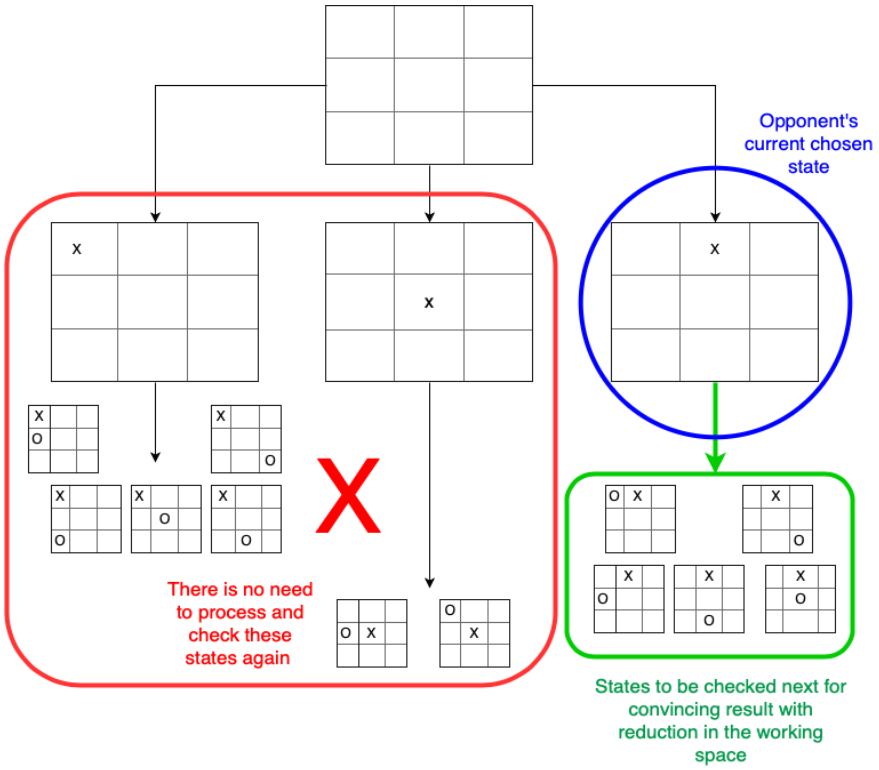

Fig 2.5: Representing how the MCTS works on planning it's next step

All this is actually stored in a simple format in the code where the values of tic-tac-toe are stored as 10 digits where the first 9 digits are binary numbers and the last digit indicates the user who is actually player 1 or 2.

For example: 1000000001 -> box 1 is marked by player 1 (we set it as A for player 1)

0000000012 -> box 9 is marked by player 2 (since we already assigned A to player1 now we assign B to player-2)

We have actually run this on our system with 100 episodes and our system took almost 7-8 hours to process. We used TensorFlow, keras to import the convolutional-2D matrices, layers and NumPy for most of the part.

- We have 1 convolution layer, 40 residual layers with value and policy heads.
- For our understanding and faster convergence, we divided model into 2 separate models, one for value and one for policy.
- Steps for the victorious player will be +1 and for the losing player it's the inverse -1.
- Value head has values in range [-1,1] which is used to predict moves.
- Batch size = 256, Sample size=512

- Loss function is SoftMax for policy and MSE for value.
- Learning rate = 0.01 (for getting really good results)
- Momentum = 0.9
- Optimizer = SGD

```
Episode :  3
Start State

|    |    |
 ---  ---  ---
|    |    |
 ---  ---  ---
|    |    |
 ---  ---  ---

Monte Carlo 2's Move
free total :  [3]
p_val :  [[0.1, 0.11, 0.09, 0.15, 0.09, 0.13, 0.1, 0.14, 0.09]]
p_child :  0
len :  (1, 9)
After Monte Carlo 2's Move
|    |    |
 ---  ---  ---
| B  |    |
```

Fig 2.6: Output during training at the beginning on episode 3.

As you can see the output for the code at the starting of episode 3 of our training.
It starts with no move then from the probability values (already set based on training 2 episodes before) the value is assigned by player 2 as B in box 4 as it has the highest probability value. We generated the training data for these files stored in .h5 files but are around 365MB each for 2 files of training and re-evaluating. For testing try running Main file, retrain and then compVsHuman for testing. Be patient to see results and that's not been a positive side as it takes at least 6 hours to run 100 episodes.
We tried running the game with different episode ranges and found the following statistics:

1 episode	-	0 - 5	(win vs loss)
10 episodes	-	1 - 6	
100 episodes	-	2 - 10	

We feel it to be a significant difference in gameplay of model, but it may not be true in every case. But think of what it can do after 3000 episodes. We might be in a 30-0 level. That's what we predict the power of such combination will be and think of the reduction in exploration space needed. It's truly a remarkably spectacular work by DeepMind. Thanks to them.

Challenges:

- Could be done by programming the code to run on GPU's but we didn't have one, making it hard to see some truly worthy results.
- Needs a lot of understanding of the RL before implementation, it is not something like the general ML algorithms. So, had to find suitable sources and sample codes to understand the application of RL and AlphaZero concepts, used them to implement the code.
- Generally, we found that the AlphaZero algorithm has to go through a minimum of 2000-3000 episodes to reach to a superhuman level which might take at least 5 days on our computer. So, it's actually impossible to run locally for visible results.

Results from references:

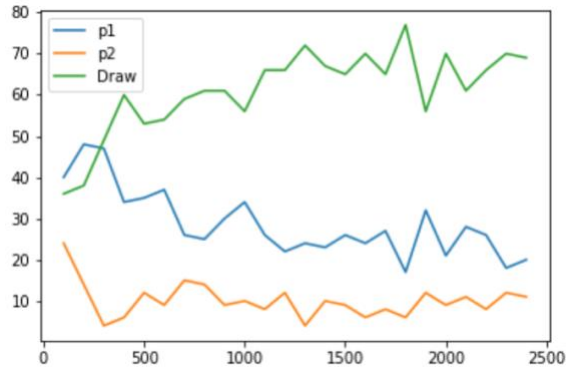

Fig 2.7: AlphaGo Zero player self-learning TTT.

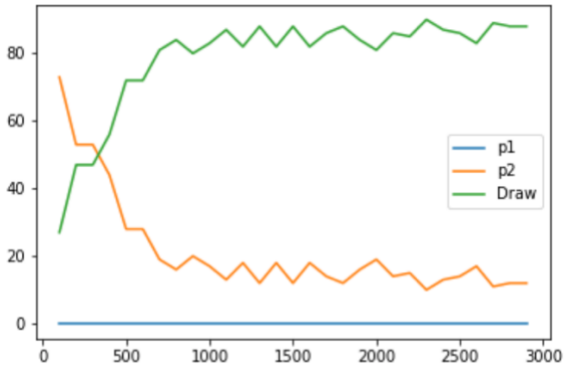

Fig 2.8: Async DQN player vs Min-max on TTT

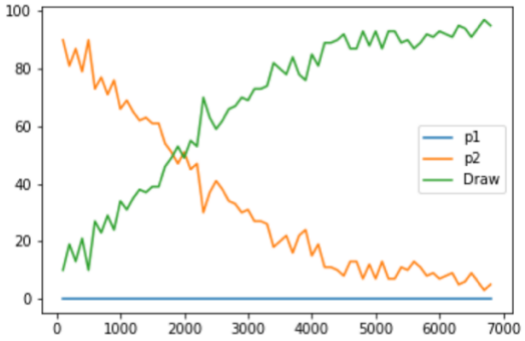

Fig 2.9: Double DQN Player vs Minimax on TTT

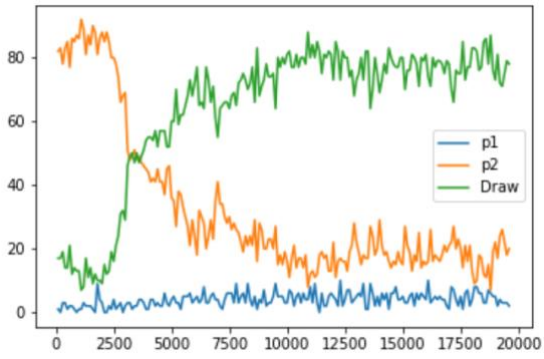

Fig 2.10: Policy Gradient Player vs Minimax on TTT

All the above figures are the outputs for running RL different algorithms on Tic-Tac-Toe (TTT). They are referred from one of our references [11] where they used a different strategy to implement the same AlphaGo Zero with multi-threading.

They are the plotted with the number of games on the X-axis and the number of wins, tie's among player-1 and player-2.

From the figures, it is clear that AlphaGo Zero is superior of the bunch. And the number of games it actually needs to reach a super-human level is a lot less than other models, making it a choice to always consider for best results for board games at least for now. But AlphaZero is much powerful than this.

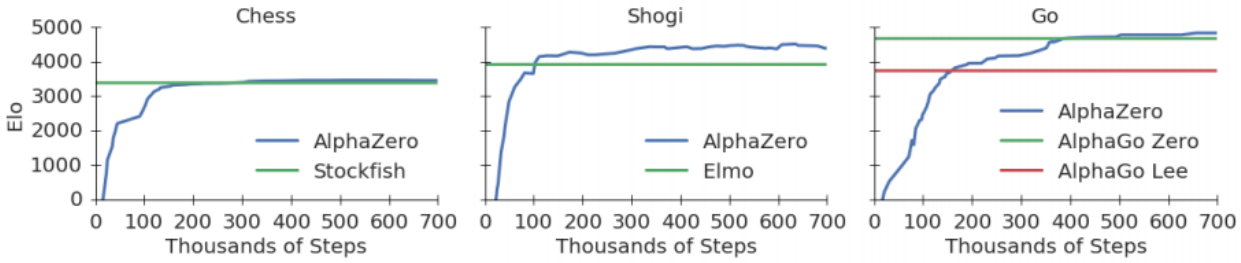

Fig 2.11: Elo ratings and performance of AlphaZero over other algorithms

The graphs described above in the fig 2.10 indicate the Elo ratings of various AlphaGo series algorithms. These ratings are calculated between players under the condition that each player is given a second to make a move.

- Here the first image titled chess indicates the performance of AlphaZero with the world-champion program called StockFish.
- The second image titled Shogi indicates the performance of AlphaZero with the world-champion program in Shogi called Elmo.
- The third image titled Go indicates the performance of AlphaZero with the learning algorithms like AlphaGo Lee and AlphaGo Zero. It's clear from the analysis as cited in [10] but it's after 20/blocks for 3 days.

AlphaZero which we tried to implement is different from AlphaGo Zero which is the generalized version of AlphaGo Zero and hard-coded configurations for search hyperparameters. Now, in this algorithm we update NN continually. AlphaGo Zero is made to work with games based on their symmetries and rotations whereas that's not the case with AlphaZero. There are games like tic-tac-toe and chess where there is still the probability of tie's in game and AlphaZero uses this possibility to predict the output of game.

Conclusion:

We feel this project to be a notable step for us towards our future study in field of RL and Deep learning. Alpha Zero masters any board game without human knowledge. Though using DFS with complex recursion for the same game where both players play their moves optimally(which is possibly the hardest thing to train the system based on heuristics as we always can't do that)we learnt that our learning and inclination towards the creation and application of a self-play schema to the game actually helps at times when we lack precise expert training to the game or training data or have sparse training data, it reduces the developer's burden in playing different possible combinations of the same game. We compare the trained data with another file taking it to be an input to play the self-play again making better decisions policy which might be a bit on the stricter side. It's relatively a new and challenging approach to implement but we believe it to have fruitful results. Tree search does the job of planning. Subsequently, increasing the strength of new plans with the guided tree search with NN. Thus, solving a possibly large exploration space of the learning mechanism. We have seen Alphazero to be applied on games like Go, Chess, Connect before etc., but we thought of implementing it to tic-tac-toe in a straightforward and easy way from the sources we had. We believe it to be an experimental hardship but a working successful attempt. Thus, narrowing the RL algorithms to have a guided way to solve problems on its own.

For future references, there are tree search techniques like Divide and conquer MCTS in this category which is the optimal version of MCTS and Mu-zero newer version of Alpha-Zero. Though this paper shows about how the implementation of AlphaZero actually reduces the exploration space of the learning algorithm, we still have high degree of uncertainty in the real world. Because it's just a board game with lot of restrictions the AI can see the opponent's movements as it's clearly easy to predict the opponent's next move from expecting the possible moves. There are cases like implementation such algorithm for auto-driving cars, where the car might actually learn to be driven to its best skill by AI but what if it suddenly rains or there's some unexpected thing happening in the surrounding to the car. There might be influence of such thing on the behavior of AI. Here we believe that our model is still like a guess-based model with a bias to win or tie most of the time and it needs a lot of experience in building such mechanisms to build one such model for real-time applications.

References:

[1] https://arxiv.org/pdf/1809.01843.pdf

[2] https://jaromiru.com

[3] http://tim.hibal.org/blog/alpha-zero-how-and-why-it-works/

[4] https://arxiv.org/pdf/1912.10648.pdf

[5] http://tim.hibal.org/blog/alpha-zero-how-and-why-it-works/

[6] https://arxiv.org/pdf/2004.06244.pdf

[7] https://medium.com/@arnavparuthi/playing-ultimate-tic-tac-toe-using-reinforcement-learning-892f084f7def

[8] https://papers.nips.cc/paper/7120-thinking-fast-and-slow-with-deep-learning-and-tree-search.pdf

[9] https://nikcheerla.github.io/deeplearningschool/2018/01/01/AlphaZero-Explained/

[10] https://www.unz.com/akarlin/alphazero-surpasses-best-chess-engines-in-4-hours/

[11] https://github.com/kirarpit/connect4

[12] https://medium.com/@arnavparuthi/playing-ultimate-tic-tac-toe-using-reinforcement-learning-892f084f7def

[13] https://nestedsoftware.com/2019/08/07/tic-tac-toe-with-mcts-2h5k.152104.html

[14] https://theoddblog.in/solving-connect4-with-Reinforcement-Learning/

[15] https://towardsdatascience.com/monte-carlo-tree-search-in-reinforcement-learning-b97d3e743d0f

[16] https://github.com/thanif/Alpha-Zero-on-Tic-Tac-Toe-using-Keras

[17] https://medium.com/applied-data-science/how-to-build-your-own-alphazero-ai-using-python-and-keras-7f664945c188

[18] https://github.com/nestedsoftware/tictac