TIC TAC TOE GAME USING DEEP REINFORCEMENT LEARNING

Amro Abbas, AhmedElmogtaba A., Idriss Cabrel Tsewalo T., Lawrence Moruye N. & Simon Bassey *
African Master in Machine Intelligence (AMMI)
{afagiri, aali, itondji, lnyangoro, sbassey}@aimsammi.org

1 INTRODUCTION

In this project, we used deep reinforcement learning to train an agent to play tic tac toe game. First, we built an environment for the tic-tac-toe game. Then we implemented an MCTS agent. Finally we utilized the MCTS agent to train a DQN to play the tic tac toe game. Our work in this project can be divided into three main parts:

1. Implementation of Monte-Carlo Tree Search (MCTS) to play **tic tac toe game**.
2. Tic tac toe game environment.
3. Implementation of deep-Q learning algorithm to train RL agent to play tic tac toe game.

2 MONTE CARLO TREE SEARCH

Monte-Carlo Tree search is made up of four distinct operations:

1. Tree Traversal/Selection.
2. Node Expansion.
3. Rollout (random simulation).
4. Backpropagation

1. Tree Traversal

The idea is to keep selecting best child nodes until we reach the leaf node of the tree.We achieve this by using Upper Confidence Bound (UCB) represented by the following formula:

$$UCB = \frac{w_i}{n_i} + \frac{c * sqrt(t)}{n_i} \tag{1}$$

Where :

- w_i = number of wins after the i-th move.
- n_i = number of simulations after the i-th move(visit count).
- c = exploration hyperparameter that we choose (theoretically equal to 2).
- t = total number of simulations for the parent node(sum of the visit count of all reachable nodes).

If there is a tie between nodes, we can choose between them at random.

2. Node Expansion

When we can no longer apply UCB to find the successor node, we expand the game tree by appending all possible states from the leaf node.

3. Simulation

*

Here the algorithm picks a child node to arbitrarily simulate the game from our current node to the end and see who won. Actions along the simulation are chosen randomly.

4. BackPropagation

Once the algorithm reaches the end of the game, it evaluates the state to figure out which player has won. It traverses upwards to the root and increments visit score at each node for all nodes visited. It also updates the win score for each node if the player for that position has won the playout.

you can look at Monti-Carlo algorithm steps for tree traversal and node expansion phase at figure[2] below :

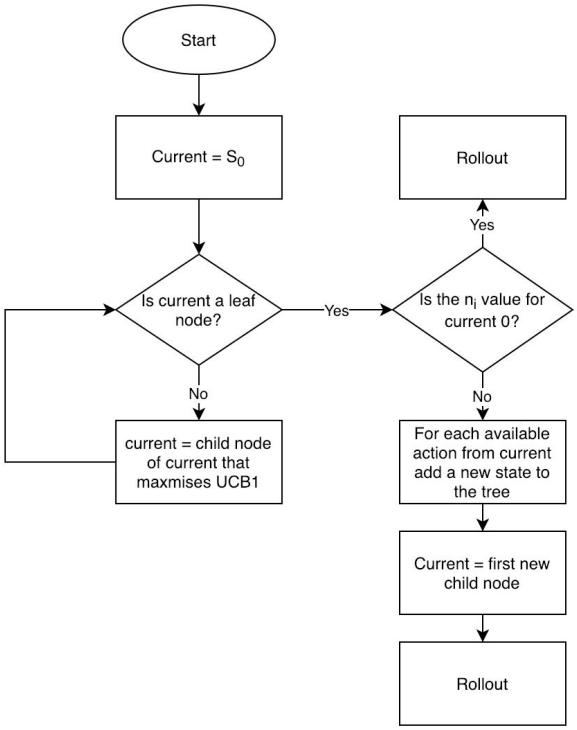

Figure 1: Flow chart for Monte-Carlo Tree Search.

3 DEEP Q-LEARNING

The basic idea behind reinforcement learning algorithms is to estimate the action-value function, by using the Bellman equation as an iterative update, $Q_{i+1}(s,a) = E[r_{t+1} + \gamma * max_{a'} Q_i(s',a')|s,a]$. This approach is totally impractical, because the action-value function is estimated separately for each sequence, without any generalisation. Instead, it is common to use a function approximator to estimate the action-value function $Q(s,a,\theta) = Q^*(s,a)$, which can be a non-linear function approximator such as a neural network. However, a neural network function approximator with weights is referred as a Q-network. A Q-network can be trained by minimising a sequence of loss functions $L_i(i)$ that changes at each iteration i,

$$L_i(\theta_i) = E[(y_i - Q(s,a;\theta_i))^2] \tag{2}$$

Where $y_i = E[r + \gamma * maxQ(s',a';\theta_{i-1})|s,a]$ is the target for iteration i.

We'll make use of a deep neural network to estimate the Q-values for each state-action pair in a given environment, and in turn, the network will approximate the optimal Q-function. The act of combining Q-learning with a deep neural network is called deep Q-learning, and a deep neural

network that approximates a Q-function is called a deep Q-Network, or DQN. Here, The network is trained with a variant of the Q-learning algorithm, with stochastic gradient descent to update the weights. We use an experience replay mechanism which randomly samples previous transitions to solve the problems of correlated data, and thereby smooths the training distribution over many past behaviors.

Let's present in a few lines the detailed steps of the Deep Q-Learning algorithm.

Deep Q Learning Step:

1. - Initialize replay memory capacity
2. - Initialize the network with random weights
3. - Clone the policy network, and call it the target network
4. - For each episode:
   (a) Initialize the starting state
   (b) For each time step:
      i. Select an action: via exploration or exploitation
      ii. Execute selected action in a emulator
      iii. Observe reward and next state
      iv. Store experience in replay memory Sample random batch from replay memory
      v. Preprocess states from batch
      vi. Pass batch of preprocessed states to policy network
      vii. Calculate loss between output Q values and target Q values:
         • Require a pass to the target network for the next state
      viii. Gradient descent updates weights in the policy network to minimize loss:
         • After x time steps, weights in the target network are updated to the weights in the policy network.

We're able to calculate the target Q-values using this formula:

$$Q^*(s,a) = E[R_{t+1} + \gamma * maxQ^*(s',a')|s,a] \tag{3}$$

The figure 2 below illustrates the overall resulting data flow into the Deep Q Network architecture.

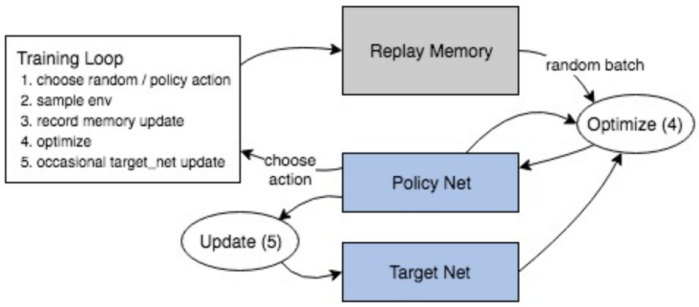

Figure 2: Deep Q-Network data flow.

4 ENVIRONMENT

For the implementation of the DQN we built a tic-tac-toe environment. The environment represents the game board as follows:

1- For a 3x3 board we have 9 cells, each cell is filled with one of three values:

Table 1: state representation

action	numerical value
'x'	1
'o'	-1
'-'	0

'x' : represents the RL agent, which means that the RL agent always plays with 'x'.
'y' : represents the opponent, which means that the opponent always plays with 'o'.
'-' : represents the empty cell.

The board is a 3x3 grid and each cell in the grid contains on of the values above. An example of a board is figure[1](a).

[a]
```
x - x
- - o
o x o
```
[b]
```
0 1 2
3 4 5
6 7 8
```
[c]
```
x x x
- - o
o x o
```

Figure 3: tic-tac-toe board (a)Board state example. (b)Board indexing. (c)Board state after the agent played at index 1.

2- We use a list representation to store the state of the game. The list also represents the input to the neural network. For the list we use numerical values to store the game state. Each cell in the board is represented with 0 for '-', 1 for 'x' or -1 'o' as illustrated in table[1].

Each value in the state list represents the numerical value of the cell in the corresponding index. The cells indexes are showed in figure[1](b)

For example, the state in the figure[1](a) above can be represented using list representation as [1,0,1,0,0,-1,-1,1,-1].

3-We represent the actions in our environment by the index of the cell we want to put the 'x' or the 'o' on it. For example if the RL agent returns the action [1], we put 'x' at index 1 on the board. For example, if the RL agent plays action [1] on the board in figure[1](a) the result will be the state in figure[1](3).

The environment contains all the necessary function for playing the game like:

- Environment initialization.
- Playing actions.
- Checking the winner/draw after each turn.
- Representing the game state.
- Representing the agent opponent.
- Agent evaluation.

For the opponent we tried different options:

- Opponent that plays randomly.
- eps-random MCTS opponent.

eps-random MCTS opponent refers to choosing random action eps*100% of the time and using MCTS (1-eps)*100% of the time.

5 DEEP Q LEARNING IMPLEMENTATION FOR TIC TAC TOE GAME

For the DQN we choose our architecture to be a MLP with 9 neurons in the input layer to receive the list representation of the game and 9 neurons at the output layer to predict the Q-value of each action at the given state. In our experiments we used MLPs with 2 and 3 layers. The target network and the online network share the same architecture.
Since during the game some actions becomes unavailable and the number of available action reduces during the game, we tried two approaches to handle this problem:

- **Option 1:**
  To allow the network to predict from all the actions space and ignore the action if its unavailable. Here we escape from the loop of predicting the unavailable action by adding randomness to the action prediction process. In this case we have two options for the reward:
  - Penalizing the network for predicting unavailable actions by giving it negative reward with high magnitude.
  - Ignoring the negative reward and tolerating the network.
    In both cases we play the selected action and add the sample to the data memory.
- **Option 2: ignore the action if its unavailable**
  To prevent the online network from predicting unavailable actions by masking out all the unavailable actions from the network output and selecting the highest action that is available.

5.1 RESULTS FOR OPTION 1: PREDICTING FROM ALL THE ACTIONS SPACE (WITHOUT MASKING):

To allow the network to predict from all the actions space and ignore the action if it is unavailable. Here we escape from the loop of predicting the invalid action by adding randomness to the agent decisions. We applied this by allowing the agent to choose random actions during both training and validation. For training we used exponential decaying eps-greedy policy. For evaluation we fixed epsilion to 0.1 (10% random).

5.1.1 RESULTS WITH NEGATIVE REWARD (RANDOM OPPONENT):

Here we set the opponent to be random agent that selects the valid action from a uniform distribution. We refer to this experiment in figure[6] as "Rand opp w/ -ve R".

Results For Experiment 5.1.1						
Training opp	Eval opp	Max Q value	eval games	win	lose	draw
random	random	-10.008	3000	1176	1255	569

5.1.2 RESULTS WITH NEGATIVE REWARD (90 PERCENT RANDOM OPPONENT 10 PERCENT MCTS):

Here we set the opponent to be random agent 90 percent of the time and Monti-Carlo agent 10 percent of the time. We refer to this experiment in figure[6] as "90%rand MCTS w/ -ve R".

Results For Experiment 5.1.2						
Training opp	Eval opp	Max Q value	eval games	win	lose	draw
rand mcts	random	-9.41	3000	1644	1005	351

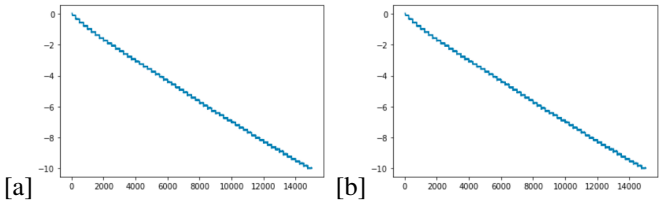

Figure 4: Q value after 15000 games without masking the invalid actions , applying the negative reward (a)playing against random opponent (b)playing against eps-random MCTS (eps=0.9). you can see from the Figure that the q is decreasing which is a very bad thing.

Table 2: Game Rewards (Masking)

state	**reward**
'win'	1
'lose'	-1

5.1.3 Results without Negative Reward (Random Opponent):

Here we set the opponent to be random agent that selects the valid action from a uniform distribution. We refer to this experiment in figure[6] as "Rand opp w/o -ve R".

Results For Experiment 5.1.3						
Training opp	Eval opp	Max Q value	eval games	win	lose	draw
random	random	0.34	3000	2020	689	290

5.1.4 Results without Negative Reward (90 percent Random Opponent 10 percent MCTS):

Here we set the opponent to be random agent 90 percent of the time and Monti-Carlo agent 10 percent of the time. We refer to this experiment in figure[6] as "90%rand MCTS w/o -ve R".

Results For Experiment 5.1.4						
Training opp	Eval opp	Max Q value	eval games	win	lose	draw
rand mcts	random	0.33	3000	1892	754	354

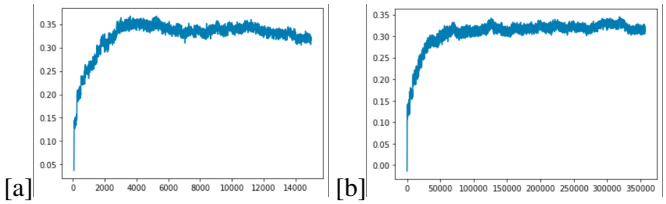

Figure 5: Q value after 15000 games without masking the invalid actions , ignoring the negative reward (a)playing against random opponent (b)playing against eps-random MCTS (eps=0.9).

5.2 Results for Option 2: Masking the invalid actions

For this settings, we always give the reward only at the end of the game. The reward we give to the agent during training is in table[2].

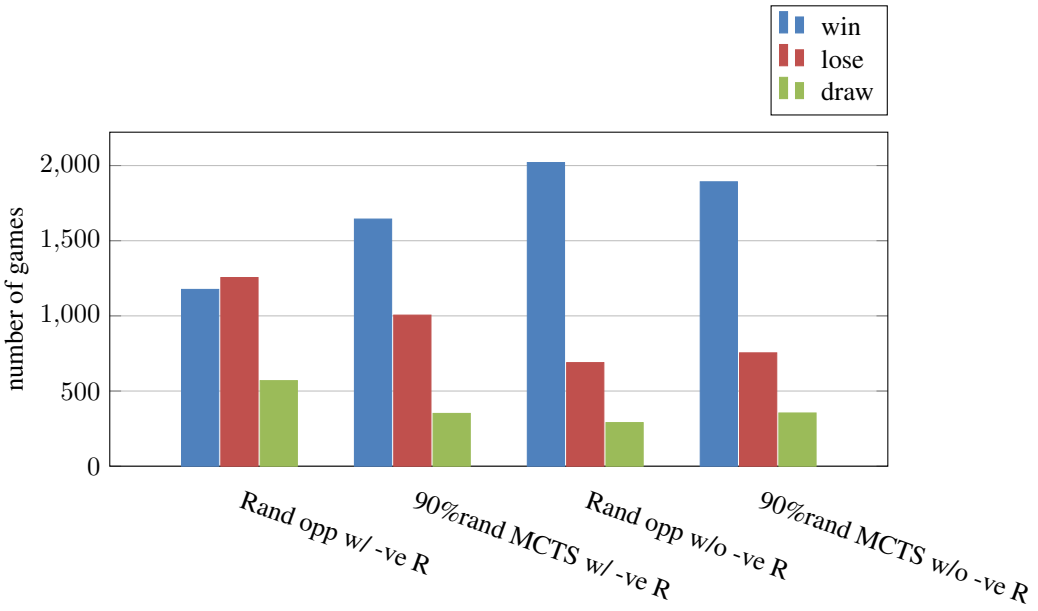

Figure 6: From left to right you can see the results for evaluation against random opponent and 90%random+10% MCTS opponent. using the to options mentioned above (penalizing and ignoring the negative reward penalization, you can see that the second option is far better)

5.2.1 Training Against Random Opponent:

Here we set the opponent to be random agent that selects a valid action from a uniform distribution. The average Q value during training for this experiments is in figure[7](a). We refer to this experiment in figure[8] as "Rand opp w/ Mask"

Results For Experiment 5.2.1						
Training opp	Eval opp	Max Q value	eval games	win	lose	draw
random	random	0.38	3000	1554	1049	379

5.2.2 Training Against eps-random MCTS:

We changed the opponent to be 90% random and 10% MCTS and trained in the same way which means that we set eps to 0.9. The average Q value during training for this experiments is in figure[7](b). We refer to this experiment in figure[8] as "90%rand MCTS w/ Mask"

Results For Experiment 5.2.2						
Training opp	Eval opp	Max Q value	eval games	win	lose	draw
0.9-random mcts	random	0.20	3000	1697	956	339

6 Conclusion

We can see from the results in section[5] that the best results can be achieved if we ignore masking the actions and train against random opponent only. Additional experiments will be done in the future and additional evaluation methods will be incorporated like human evaluation and evaluation against MCTS only. Also it seems that training against stronger opponent gives better result, so additional experiment will we be done on this.

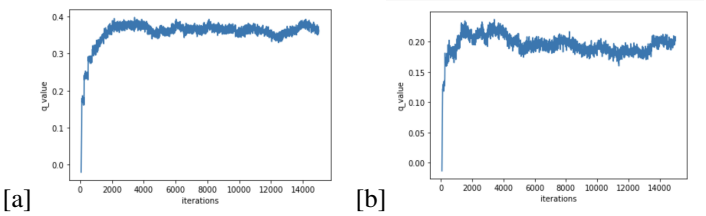

Figure 7: Q value after 15000 games when masking the invalid actions (a)playing against random opponent (b)playing against eps-random MCTS (eps=0.9).

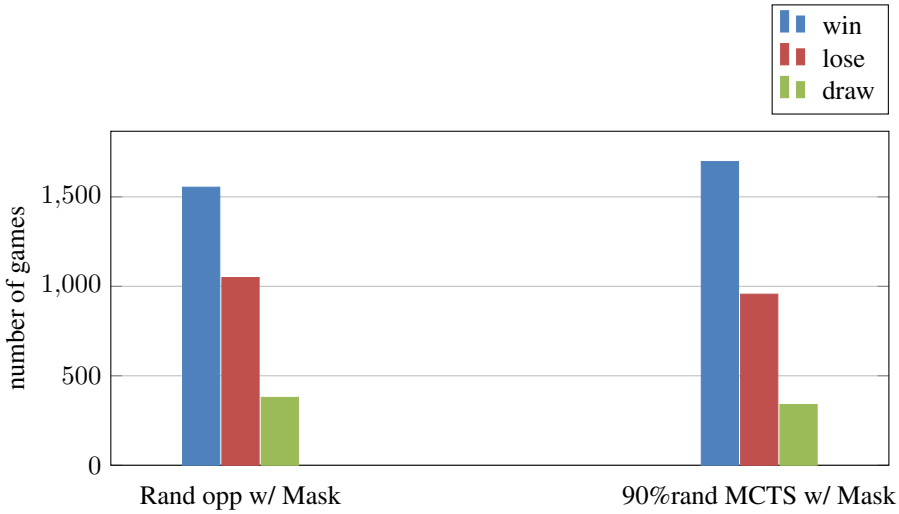

Figure 8: Masking results for evaluation against random opponent. left)Training against random opponent. right)Training against 90%random+10% MCTS opponent. We can see that training against stronger opponent gives better results.

another conclusion that you can easily notice is that the MCTS agent is far better than DQN agent in tackling this game and reason for that is the search space of this game is not very large , so the MCTS can easily expand the whole search space and know which is the best action to take that can maximise his reward and minimise the opponent reward, so that why you can think of it as an ideal agent.