Deep Reinforcement Learning for Board Games

CS 541: Deep Learning
Instructor: Prof. Jacob Whitehill
Worcester Polytechnic Institute

Team Members:
Madhan Suresh Babu
Nikhil Jonnavithula
Surya Murugavel Ravishankar

TABLE OF CONTENTS

LIST OF FIGURES

INTRODUCTION

Board games have been a way to keep our minds engaged. Most board games are tricky, hard, and involve playing against other players. Some of them factor in luck like dice or card games and are probabilistic in nature while others are deterministic. A general reinforcement learning algorithm that could solve any board game, without any prerequisite knowledge about the game is a step towards building a general AI, and it is the focus of our project. This approach combines Deep Reinforcement Learning with Monte-Carlo Tree Search to learn the best action to take in a given game state to potentially win the game. We have implemented this algorithm for two games namely, Chess and Connect-4. Both games are on the different end of the spectrum in terms of their complexity and size of the action space and provide a good comparison of the memory and compute requirements for this algorithm to be successful.

METHODOLOGY

Reinforcement Learning:

Reinforcement learning is an area of machine learning that focuses on how to act in an environment in order to maximize some given reward. Reinforcement learning algorithms study the behavior of subjects in such environments and learn to optimize that behavior. Game playing is a common domain that aptly illustrates its power and capability. In this domain, the state of the agent is defined using the game situation and for a board game, it encodes information on who's turn it is and the positions of pieces on the board.

A neural network is used to approximate the optimal policy and the Value for a given state. It takes as input the state of the board and has two outputs, a Value (in the range from [-1,1]) that represents the estimate of winning for the agent from that state and a Policy which is the probability vector to be followed over all possible valid actions in that state. A negative value estimate implies that the opponent has a higher chance of winning. The choice of network architecture used for the Policy and Value depends entirely on the complexity of the game. In our project, we have used a deep Convolutional Neural Network with residual connections for the game of Chess, and a shallow Feed Forward Neural Network for the game of Connect-4.

Monte-Carlo Tree Search(MCTS):

Monte-Carlo Tree Search is a search algorithm that relies on randomness to learn the optimal Value function and optimal Policy for a game over time. It uses the existing policy from the neural network to calculate a UCT (Upper Confidence Bound). This UCT is used to determine the best action to take in a state in the simulation (self-play) that would lead to the next state that gives the highest chance of winning for the agent. It is computed from the Q-value estimate for that (state, action) pair, Policy from the network, state visit count for that state and the state-action count for that state and action :

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

Fig. 1: Utility estimate (Reference[2])

Where P(s, a) is the Policy output from the neural network

N(s,a) is the State-action count for the (state,action) pair under consideration

Q(s,a) is the Q-value of the (state,action) pair

C_{puct} is the exploration parameter

In MCTS, a tree is built with the current state of the board as the root of the tree in each turn. The UCT is computed for each (state, action) pair for the current state, and the action with the highest UCT is taken. Additionally, the tree also keeps a count of the number of times it has visited a state, and the number of times a certain action was taken in a given state. At the end of the search, the policy estimate to be returned by the MCTS algorithm for a state S is obtained from the number of times each possible action was taken in that state :

MCTS Policy for a valid action in a state = (Number of times that action was taken in the given state in all MCTS simulations) / (Total number of visits for that state)

One MCTS simulation involves calculating the UCT recursively for each state starting at the root node, and performing the best action based on the UCT estimate. This is done for each next state if that next state is present in the tree until a new next-state is encountered. When a new next-state is encountered (leaf), it is added to the search tree and its state-count and state-action-count for this leaf are initialized to zero. Then the Value of this state (v) is determined by passing the state through the neural network, and this value is propagated upward till the root of the tree to update the Q value of each preceding (state, action) pair. Starting from the leaf node, each state returns the negative of the Value estimate to its parent node, as the Value estimate for a player in a state is the negative of that Value for his opponent. The Q-function update equation is as follows :

$$Q(state,action) = (v + state\text{-}action\text{-}count(state,action)*Q(state,action)) / (1+state\text{-}action\text{-}count(state,action))$$

As the number of simulations is increased, the Q values become more accurate leading to better estimates of the UCT, which in turn leads to better estimates for the MCTS Policy. The MCTS Policy could then be used by the agent to take action in the actual game.

Policy Iteration:

Policy iteration is a method that starts with an initial policy and iteratively improves it to learn the optimal policy for the environment. It involves two steps, namely Policy evaluation, and Policy improvement. In Policy evaluation, the existing policy is used to obtain its corresponding Value function. This Value function is improved upon in Policy improvement and is used to obtain a better estimate of the policy.

In our project, the calculation of UCT using the Policy from the neural network and the update of the Q-values through MCTS simulations: forms the Policy evaluation step, and the calculation of the MCTS Policy from the State-action count and state-count forms the Policy improvement step. This improved policy is used to take action proportional to the probabilities in the self-play games, and it is used to take the best action (action with the maximum probability) in the actual game when testing.

During training, the agent plays several games with itself (self-play) performing Monte-Carlo Simulation in each turn to obtain the improved policy. The state of the game, MCTS Policies in each state and the result of the self-play games are used to train the neural network using the loss function :

$$l = \sum_t (v_\theta(s_t) - z_t)^2 - \vec{\pi}_t \cdot \log(\vec{p}_\theta(s_t))$$

Figure 2: Loss function (Reference[2])

Where $v(s_t)$ is the value estimate from the neural network,
z_t is the result of the self-play game
pi_t is the improved policy estimate from MCTS simulations,
and $p(s_t)$ is the policy estimate from the neural network.

Input Representation:

The input to the neural network implicitly tries to convey the state of the game to the network. The performance of the algorithm solely depends on how we represent the board state to the network. Though we could make the network learn the game with just the images of the board, we could easily speed up the process of learning by having a better representation of the board.

Chess:

Chess is a two-player strategy game that is played on 64 squares 8x8 checkerboard. Each player begins with 16 pieces, one king, one queen, two rooks, two knights, two bishops, and eight pawns. Each type of the aforementioned chess piece has a unique way of moving across the board. A player's pieces are used to attack and capture the opponent's pieces, whilst supporting one another. The goal is to checkmate the opponent's king by placing it under an inescapable position such that there is no other move to save the king. In such a case the player wins. The player can win if the opponent resigns or, in a timed game, runs out of time.

Neural Network Architecture utilized for Chess:

Input to the network is an 8*8*111 { 8*8*((6+6+1)*8+7) } matrix, where 8*8*104 { 8*8*((*6*+*6*+1)*8) } represents the position of each unique chess piece ('R', 'N', 'B', 'Q', 'K', 'P' - *6* planes belonging to each player) and one additional layer indicating the number of times that state has repeated in the game. These (6+6+1) layers represent the pieces for one time step. The state also encodes the information of the game in the last seven-time steps hence consisting of (6+6+1)*8 channels. The last 7 channels of the state implicitly represent the status of the current board with a layer for each of the following features: color(1), full-move count(1), castling right(4), half-move count(1).

The output of the network is an 8*8*73 tensor, where each position in the 8*8 grid of any channel is the position from which the piece should be picked. There are 73 possible moves for each picked piece 56(8 direction * 7 steps) - queen moves, 8 (2 possible * 4 directions) - knight moves, 9(3 directions* 3 pieces) - under promotions for pawn as Knight, Bishop, or Rook. The underpromotions for Queen are included by default in the queen moves for that pawn when it reaches the farthest side of the board. This denotes the policy of the given state. Another output is a Value estimate from [-1,1] that denotes the value of the given state for the current player.

The neural network outputs policies for several invalid moves as well. The valid policy is obtained by masking the invalid moves and normalizing the masked policy.

Model summary of the neural network :

```
Network_arch(
  (conv1): Conv2d(111, 111, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(111, 111, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv3): Conv2d(111, 111, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv4): Conv2d(111, 111, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv5): Conv2d(111, 111, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv6): Conv2d(111, 111, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv7): Conv2d(111, 111, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv8): Conv2d(111, 111, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv9): Conv2d(111, 111, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv10): Conv2d(111, 111, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv11): Conv2d(111, 111, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv12): Conv2d(111, 111, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv13): Conv2d(111, 111, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv14): Conv2d(111, 111, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv15): Conv2d(111, 111, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv16): Conv2d(111, 111, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv17): Conv2d(111, 111, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv18): Conv2d(111, 111, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv19): Conv2d(111, 111, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv20): Conv2d(111, 111, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (bnorm1): BatchNorm2d(111, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (bnorm2): BatchNorm2d(111, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (bnorm3): BatchNorm2d(111, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (bnorm4): BatchNorm2d(111, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (bnorm5): BatchNorm2d(111, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (bnorm6): BatchNorm2d(111, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (bnorm7): BatchNorm2d(111, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (bnorm8): BatchNorm2d(111, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (bnorm9): BatchNorm2d(111, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (bnorm10): BatchNorm2d(111, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (fc1): Linear(in_features=7104, out_features=1024, bias=True)
  (bnorm11): BatchNorm1d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (fc2): Linear(in_features=1024, out_features=1024, bias=True)
  (bnorm12): BatchNorm1d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (fc3): Linear(in_features=1024, out_features=4672, bias=True)
  (fc4): Linear(in_features=1024, out_features=1, bias=True)
)
```

Figure 3: Network Architecture for Chess

The neural network for Chess consists of 10 residual blocks each with 2 Convolutional Neural Networks present in them. The output of the residual blocks is fed to a 2 layer feed-forward neural network, whose output is fed to a final softmax layer for obtaining the Policy vector. And the value of the input state is obtained from a final layer with a single neuron whose activation function is the hyperbolic tan function (tanh). All other hidden layers used the Rectified Linear Unit as the activation function. Residual blocks were used for the Policy output as they deal with the Vanishing gradient problem better.

Connect-4:

Connect-4 is a two-player connection board game played on a vertical grid of six rows and seven columns. The objective of this game is for each player to be the first to connect-4 of their pieces in a horizontal, vertical, or diagonal direction. Each piece falls straight down in the respective column in which it is dropped, occupying the lowest available row position in that column.

Neural Network Architecture utilized for Connect-4:

The neural network architecture for Connect-4 is a simple 3 layered feed-forward neural network, which uses the softmax activation function in the final layer to calculate the policy and the tanh activation function in a different final layer for the Value estimate. The input to the network is an 84 (2*6*7) neuron layer representing the current player's pieces in the first 42 (7*6) neurons and the opponent's pieces in the next 42 neurons.

The output of the network is a Policy vector of size 7 representing the probability to be followed for choosing one of seven columns to drop the pieces and a Value estimate for each state. As the game progresses, when certain columns become completely occupied the policy vector is masked and normalized to obtain the valid policy that excludes the invalid moves (columns).

Model summary of the neural network:

```
Network_arch(
  (fc10): Linear(in_features=84, out_features=42, bias=True)
  (fc11): Linear(in_features=42, out_features=21, bias=True)
  (bnorm11): BatchNorm1d(42, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (bnorm12): BatchNorm1d(21, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (fc3): Linear(in_features=21, out_features=7, bias=True)
  (fc4): Linear(in_features=21, out_features=1, bias=True)
)
```

Figure 4: Network Architecture for Connect-4

RESULTS

Chess:

By observation, we can identify that the RL agent of our project is able to beat its previous version after each iteration of self-play and training, proving that it learns the game and is getting better at it. But since Chess is a complicated game with a huge search space, the Policy estimate returned by the MCTS in our implementation was non-optimal. A higher number of MCTS simulations would certainly lead to near-optimal MCTS Policy from which the neural network could learn well.

Connect-4:

Due to the lack of powerful computing resources (several thousands of TPU as used by AlphaZero) required for the algorithm to search and learn to play Chess, we have benchmarked our project using the game of Connect-4. We pitted the trained RL agent of our project with a computer program that uses the Minimax game tree search algorithm. This program for the Minimax search algorithm was found online and is used as a baseline for our project.[3]

A series of 100 games were conducted between our RL agent and the Minimax algorithm. Of these games, the RL agent won 98 games, losing only 2 games to the Minimax algorithm. From this, we can infer that the neural network does a better job in learning the Value function and Policies for the states than the hardcoded evaluation function in the Minimax algorithm.

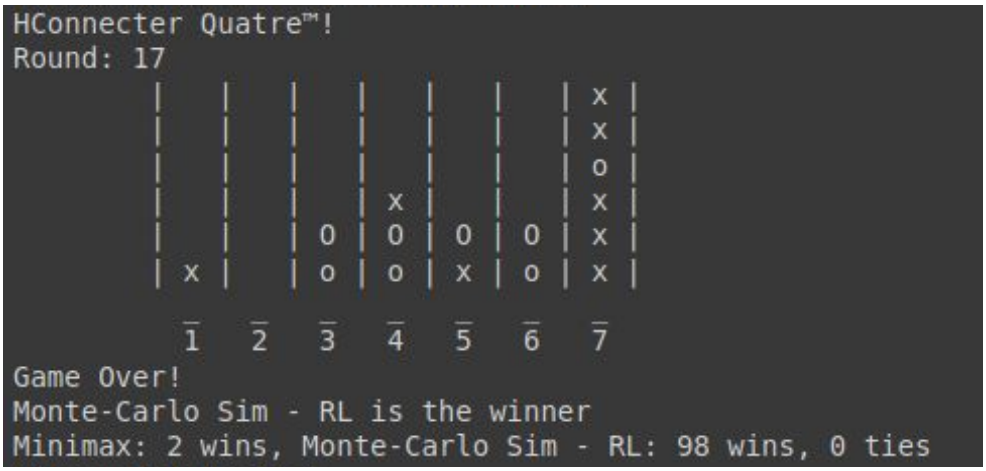

Figure 5: Result for MCTS-RL vs Minimax for Connect-4

REFERENCES

AlphaZero:
[1]:https://kstatic.googleusercontent.com/files/2f51b2a749a284c2e2dfa13911da965f4855092a179469aedd15fbe4efe8f8cbf9c515ef83ac03a6515fa990e6f85fd827dcd477845e806f23a17845072dc7bd

[2]:https://web.stanford.edu/~surag/posts/alphazero.html

[3]:https://github.com/erikackermann/Connect-Four