Reinforcement learning in chess (Part I)

Ganesh Arvindh Ramanan (SC21B150)
Harshit Prashant Dhanwalkar (SC21B164)

March 9, 2024

1 Introduction

The goal of this project is to create a chess engine using the reinforcement learning algorithm TD(λ). This project is divided into two parts, the first being the implementation of the TD(λ) algorithm to get the *evaluation function*. In the second part, this evaluation function is used to generate the reward for all possible legal moves that the engine can make. Once the reward is generated, the engine simply plays the move which maximizes the reward.

In this report, we will focus on the code to enable the engine to play with an opponent provided that the *evaluation function* was already learned using the TD(λ) algorithm.

2 The problem statement

Let us denote with S the set of all possible states (chess positions) and with $x_t \in S$ the state at time t. The index t also means that x_t was obtained after t played *actions* or moves. Let us assume that the game lasted for N moves. For each state x_t we have a set A_{x_t}, which is the set of all possible legal moves given the chess position x_t. The agent chooses to play a move $a \in A_{x_t}$ which leads to the next state x_{t+1}. Note that x_{t+1} comes after both the players have played their moves. Thus, only those positions will be consided in which the agent has the ability to take an action.

The agent will receive after each finished game a reward for the final move $r(x_N)$, which in chess takes the value 0 for draw, -1 for loss and +1 for victory. We will now define a function $J(x)$ which is defined as

$$J(x) = E_{x_N|x} r(x_N)$$

That is, given a state x, $J(x)$ gives us the expected value of reward. We can think of this as a function which returns the evaluation of a chess position and gives us the probability of success. TD(λ) algortihm is applied to get an approximation to $J(x)$.

3 Code

Let us assume that the TD(λ) algorithm was employed and a suitable value of the evaluation function is obtained. Let's say that this evaluation function is given by

$$J(x) = \sin(x)$$

The goal now is to the exploit this function to generate intermediate rewards. To achieve this we first create a function which returns a list containing all possible legal moves given a state. We then iterate through this list and play every move. The corresponding states that are obtained due to each move is stored and not shown. The computer then iterates through all possible moves of the opponent and

the states obtained are judged using the evaluation function. The computer assumes that the opponent will play the move which reduces the computer's chance of victory as much as possible. Thus it takes a minimum of the set of evaluations of all possible final states and assigns it as a *reward* for choosing to play that move. It finally plays the move that maximizes reward. Note that the computer only thinks one step deep as it assumes that the evaluation function given to it is ideal.

```
import chess
import chess.svg
import math
import base64
from PIL import Image
import cv2
from cairosvg import svg2png


def eval_func(x):
    return math.exp(-(x**2)) # We are taking J(x) = sin(x)

def encode_board(board):
    # Encode the board using base64
    encoded_board = base64.b64encode(board.board_fen().encode('utf-8')).decode('utf-8')
    return encoded_board

def decode_board(encoded_board):
    # Decode the encoded board using base64
    decoded_board_fen = base64.b64decode(encoded_board.encode('utf-8')).decode('utf-8')
    board = chess.Board(decoded_board_fen)
    return board

def base64_to_int(encoded_board):
    bytes_repr = base64.b64decode(encoded_board)
    int_repr = int.from_bytes(bytes_repr, 'big')
    return int_repr


def dispBoard(board):
    f = open("pic.svg","w")
    a = chess.svg.board(board, flipped=True)
    f.write(a)
    f.close()
    svg2png(url="./pic.svg", write_to="./pic.png")
    image = cv2.imread('./pic.png')
    cv2.imshow('image window', image)
    cv2.waitKey(1)
    #cv2.destroyAllWindows()


def simple_terminal_engine():
    # Create a chess board
    board = chess.Board()

    while not (board.is_checkmate() and board.is_stalemate() and board.
    is_insufficient_material()):
        final_eval_arr = []
        A = [i for i in board.legal_moves] #Creating action space
        for WhiteMove in A:
            board.push_san(str(WhiteMove)) # This function plays the move and modifies
    board
            temp = []
            for BlackMove in board.legal_moves:
                board.push_san(str(BlackMove))
                intermediate_eval = eval_func(base64_to_int(encode_board(board)))
                temp.append(intermediate_eval)
```

```
56                  board.pop() #Undo the last move
57              final_eval_arr.append(min(temp))
58              board.pop()
59          final_eval = max(final_eval_arr)
60          action = final_eval_arr.index(final_eval)
61          board.push_san(str(A[action]))
62          print(board)
63          dispBoard(board)
64          PlayerMove = str(input("Enter move : "))
65          board.push_san(PlayerMove)
66          print(board)
67          dispBoard(board)
68  simple_terminal_engine()
```

Note : This code is taken from the file 'Chess.py'.

Let us go through this code line by line. In line number 1, we have imported a module called *chess*. This module provides a number of useful features such as storing he chess position as a string using FEN (Forsyth–Edwards Notation), detecting checks, checkmates and stalemates etc... We now want a way to convert this string (the FEN string) into a number. This is done by converting the string into base64 format which is then converted to base10 (which is our required number). The function for doing this is given in lines 12 and 23.

Thus we have now found a way to represent every state with a unique number which can be fed to the evaluation function to obtain the evaluation. In line 28, we have defined a function which converts the board in FEN to a picture. Line 37 contains the function which has our required algorithm. We start by creating an object named 'board' of the class 'Board'. The while loop iterates the block of code underneath if the game didn't end.

We now create an empty array which stores the evaluations of the states obtained after the computer has played for the opponent. The block of code under the for loop populates this array. We then take the index of the maximum element in this array and play the move in the action space corresponding to this index.