DAT410 Module 6 Assignment 6 – Group 26

Yahui Wu (MPMOB) (15 hrs)
yahuiw@chalmers.se
Personal number: 000617-3918
Tianshuo Xiao (MPMOB) (15 hrs)
tianshuo@chalmers.se
Personal number: 000922-7950

February 28, 2023

We hereby declare that we have both actively participated in solving every exercise. All solutions are entirely our own work, without having taken part of other solutions

Reading and reflection

Yahui Wu

As a exhaustively searching method, the traditional tree search used to play games automatically could have some computational issues when it is dealing with some complex games like chess and checkers not to mention the game of go which has been viewed as the most challenging of classic games with large depth and breadth of searching. AlphaGo combined Monte-Carlo simulation algorithm with value and policy networks based on neural networks and it achieved high winning rate against other Go programs and beat the top human player.
Generally, there are two ways to reduce the search space. First, the depth of search could be reduced by position evaluation. Second, the breadth could be reduced by sampling actions from a policy $p(a|s)$ referring a probability distribution over possible moves a in state s. The first stage of the training pipeline of AlphaGo is using a fast rollout policy and a supervised learning (SL) policy whcih alternates between convolutional layers with weights σ, and rectifier nonlinearities. And a final softmax layer outputs a probability distribution over all legal moves a. In the second stage, the reinforcement learning (RL) is used to improve the policy networks by playing games between the current policy and the randomly selected previous one and improve the current policy by stochastic gradient ascent. Basically, it's a RL algorithm to make AlphaGo learn from doing self-play. The last stage focuses on the position evaluation. The position value networks here output a single prediction of the current state s. The weights of value networks are trained by regression on state-outcome pairs (s, z), using stochastic gradient descent to minimize the mean squared error (MSE) between the predicted value $v_\theta(s)$, and the corresponding outcome z.

Tianshuo Xiao

This article focuses on a challenging problem in Go confrontation because of its huge search space and the difficulty of evaluating board positions and moves. The article introduces a new computer approach to Go that uses a value network to evaluate board positions and a strategy network to select moves, combining supervised learning from human expert games and reinforcement learning from self-games. A search algorithm that combines Monte Carlo simulation with value and strategy networks is also presented.
The widely known AlphaGo combines strategy and value networks in the MCTS algorithm by traversing the tree from the root state through the simulation. At each time step t of each simulation, an action is selected from the state to maximize the value of the action plus a reward. At the end of the simulation, the action values and the number of visits are updated for all traversed edges. Each edge accumulates the visit counts and average evaluation for all simulations that pass through that edge.
The article also describes the use of Monte Carlo tree search, a technique for exploring the game tree and identifying promising moves. By combining the evaluation capabilities of neural networks with Monte Carlo tree search, AlphaGo is able to make more strategic decisions and outperform its human opponents.

Implementation

```
# Create a calss of node
class Node:
    def __init__(self, parent, board, marker):
        self.parent = parent
        self.board = board
        self.children = []
        self.marker = marker
        self.n = 0
        self.q = 0
        self.ucb = 10000

def monte_carlo_tree_search(root,t):
    inter = 0
    while inter<=t:
        leaf = traverse2(root)
        simulation_result = rollout(leaf)
        #print(leaf.board)
        backpropagation(leaf,terminal(simulation_result))
        inter+=1
    return best_child(root)

# Create an empty board given a size
def create_board(n):
    board = [' ']*n**2
    return board

# Find the index of the empty sapce on the board
def node_select(node):
    node_index = []
    for i in range(len(node.board)):
        if node.board[i] == ' ':
            node_index.append(i)
    return node_index

def play(node):
    choice = node_select(node)
    if choice != []:
        choice = random.choice(choice)
        if node.board.count('X') == node.board.count('O'):
            node.board[choice] = 'O'
        else:
            node.board[choice] = 'X'
    return node

# judge whether the state is terminal or not
def terminal(node):
    n = int(math.sqrt(len(node.board)))

    for i in range(n):
        if node.board[n*i:n*i+n] == ['O']*n:
            if node.marker == 'O':
                return 1
            else:
                return -1
        elif node.board[n*i:n*i+n] == ['X']*n:
            if node.marker == 'O':
                return -1
            else:
                return 0
        elif node.board[i:(n-1)*n+i+1:n] == ['O']*n:
            if node.marker == 'O':
```

```
                return 1
            else:
                return -1
        elif node.board[i:(n-1)*n+i+1:n] == ['X']*n:
            if node.marker == 'O':
                return -1
            else:
                return 1
    if node.board[0::n+1] == ['O']*n:
        if node.marker == 'O':
            return 1
        else:
            return -1
    elif node.board[0::n+1] == ['X']*n:
        if node.marker == 'O':
            return -1
        else:
            return 1
    elif ' ' in node.board:
        return 10
    else:
        return 0

# expand the selected node
def node_expand(node):
    index = node_select(node)
    if index != [] and terminal(node) == 10:
        for i in index:
            board = copy.deepcopy(node.board)
            node1 = Node(parent = node,board = board,marker = node.marker)
            if node1.board.count('X') == node1.board.count('O'):
                node1.board[i] = 'O'
            else:
                node1.board[i] = 'X'
            node.children.append(node1)
    return node

# find the node with best uct
def best_ucb(node): # the chosen node must has children, children != []
    list_ucb = []
    list_child = node.children
    for node_child in list_child:
        list_ucb.append(node_child.ucb)
    max_index = list_ucb.index(max(list_ucb))
    selected_node = list_child[max_index]
    return selected_node

# judge whether a node is fully expanded
def fully_expanded(node):
    list_n = []
    for node_child in node.children:
        list_n.append(node_child.n)
    #print(np.prod(list_n))
    if np.prod(list_n) == 0 or list_n == []:
        return False
    else:
        return True

# select unvisited child, if the node is fully expanded, select child with highest uct
     as the new selected node
def traverse2(node):
    if node.parent == None and node.n == 0:
        node = node_expand(node)
    while fully_expanded(node):
```

```
        node = best_ucb(node)
        if node.children == []:
            node = node_expand(node)
    if terminal(node) == 10:
        return best_ucb(node)
    else:
        return node

# rollout the node
def rollout(node):
    while terminal(node) == 10:
        node = rollout_policy(node)
    return node

# rollout policy --> random sampling
def rollout_policy(node):
    node1 = copy.deepcopy(node)
    node1 = play(node1)
    return node1

# backpropagate
def backpropagation(node, value):

    done = False

    #Update all parent nodes up to root node
    while done == False:
        node.n += 1
        node.q += value


        if node.parent == None:
            node.ucb = (node.q/node.n)
            done = True

        else:
            #if node.parent.n == 0:
            node.parent.n += 1
            node.ucb = (node.q/node.n) + 2 * np.sqrt(np.log(node.parent.n)/node.n)
            node = node.parent

# find the best child which is most visited
def best_child(node):
    list_n = []
    list_child = node.children
    for node_child in list_child:
        list_n.append(node_child.n)
    index = list_n.index(max(list_n))
    return list_child[index]
```

Listing 1: Monte-Carlo tree search

Description of the system

We implemented the tree search using Monte-Carlo Tree Search (MCTS). The whole model consists of several parts was built based on the four steps of MCTS - selection, expansion, simulation and backpropagation. In the model, we defined a class "Node" for nodes in the tree. The defined node has some attributes such as "parent" refer to its parent node (the root node has parent = None), "board" represents the current state of the board in this node and "children" as a list left to store its children during the expansion of the node. Also, the node has attributes "q" and "n" for the calculation of "uct" using UCT method.
At the very beginning of MCTS, the current state is considered as the root node of the tree. The node is then expanded by the function "node_expanded" which gives all the possible future states as the children stored in the attribute of the node. When the current node is expanded, it has children to be visited. The child to go through the simulation is selected sequentially.
The roll-out/simulation adopts random sequence of moves that starts from the selected and unvisited child node until the terminal state where a victory, defeat or tie occurs. The random sequence of moves achieved by using function random.choice() which randomly chooses a blank position on the board to play each time.
After each time of roll-out, the evaluation will be carried out from the visited child node back to the root node. The evaluation is based on the upper confidence tree (UCT) method which is defined by the following equation: $UCT(V_i, V) = \frac{Q(v_i)}{N(v_i)} + c\sqrt{\frac{logN(v)}{N(v_i)}}$. The first part $\frac{Q(v_i)}{N(v_i)}$ is a winning/losing rate and nodes with higher $\frac{Q(v_i)}{N(v_i)}$ are more likely to reach winning states. The second part $c\sqrt{\frac{logN(v)}{N(v_i)}}$ favor nodes that have not been explored. During each time of backpropagation or evaluation, the number of visits $N(v_i)$ and the reward of simulation $Q(v_i)$ of a node will be updated so will its UCT value. The evaluation will go up to the root node. Once the current node is fully expanded, the node selected for the next iteration will be the one with highest UCT value. By selecting nodes based on UCT method, we trade off the reward of making a certain action and the exploration of other possibilities.
To play tic-tac-toe using MCTS in a simple situation at the beginning, we set our opponent policy to be randomly play. Then, considering a more competitive situation, we let AI play with itself.

Evaluation and improvement of the system

At first, we implement our AI player against the weakest opponent whose policy is randomly playing which could imitate player who doesn't know the game rules at all. We let them play for 100 games. When AI moves first, it has an almost 100% winning rate and the average computational time for each game is 1.5s with 100 iterations. When AI moves after, its winning rate drops to 77% and it lost 13 of 100 games. Then we let the system play with itself with the iteration number of 50, the outcome is that the AI who plays first will have a 64% winning rate while the AI who plays after will have a 34% winning rate and the computational time for each game is 2.4s on average. If we increase the number of iterations, the winning rate of AI playing first will increase to 77%. It seems that the algorithm is powerful enough as long as we let it move first thus even a experienced player could be defeated. However, if we take an insight of each game when AI plays as a defender, we could find that it prefer to attack even its opponent is going to win with only one move to make. The reason that AI likes to bet on victory is UCT algorithm which we use in selecting the node as the best node to explore tends to let the player win rather than avoid losing the game. Therefore, in tic-tac-toe, the one moves after is actually to avoid losing instead of seeking for victory straightforward.
At the very start of the game, the computational time that the system consumes would be much longer than when the game is close to the end state because if the number of iteration is set to be fixed, the initial state of the board has lots of spaces left for the next move with much more possible states to be

explored. In other words, if we set the number of iteration as a constant number and make sure our AI system is powerful to defeat an ordinary opponent (the number of iteration can't be too small), the most of the iterations would be wasted when the game is near the end. If we would like to trade off the computing power and the computational time, we should let the the number of iteration decrease as the game goes on. We defined a function using polynomial fit to create a list of the numbers of iteration which decrease nonlinearly.
When we apply the system to a 4x4 grid and let AI play against randomly playing opponent for 100 times, AI player as the defender still has about 70% winning rate which is almost as before. However, it only lost 6 of 100 games and 28 are tie. The AI game playing system is more adapted to tic-tac-toe with larger grid.

References

[1] Kamil, Czarnogorski. Monte Carlo Tree Search – beginners guide. Retrieved MAR, 24, 2018, from https://int8.io/monte-carlo-tree-search-beginners-guide

[2] Silver, D., Huang, A., Maddison, C. et al. Mastering the game of Go with deep neural networks and tree search. Nature 529, 484–489 (2016). https://doi.org/10.1038/nature16961

Reflection on the previous module

Yahui Wu

In the last module, we discussed the interpretability of AI systems. We argued for and against the use of black-box AI surgeons in the case of human surgeon with lower cure rate than a black box AI surgeon. Sometimes, a brain of human could be regarded as a black-box that a decision can not be fully explained since the working mechanism of our brain is very complicated. Some current neural networks imitate how a human's brain works and it has higher cure rate which could be reliable. However, we don't know where the cure rate comes from which leads to the lack of trust. In Zachary Lipton's paper, causality was considered as an important reason of why we pursue the interpretability of AI systems. Besides, the other desiderata of interpretability are trust, transferability, informativeness and fairness. Additionally, there are two important properties of interpretability, the first is transparency of a system and the second is called post hoc interpretability.
In our assignment, we defined the interpretability that the system should be transparent and could be understood by users. Thus we chose the decision tree as our method since its process of making the decision can be visualized and each component and node represents the selected feature and its value could be observed. Therefore, we can explain our diagnoses by analyzing the features and take the features as the causality.

Tianshuo Xiao

In the last module, we studied diagnostic systems and focused on the application of diagnostic systems in healthcare. However, the application of AI in healthcare can be more controversial. AI diagnoses are less interpretable but more accurate, while manual diagnoses are the opposite.
In Zachary Lipton's article, we get the following points: trust, causality, transferability, informativeness and fairness.The advantages of diagnosis through AI are a higher cure rate, time savings and greater sensitivity. But there are also some disadvantages, such as the analysis of the causes of the patient's disease is more difficult to explain.
An important effort in machine learning is to investigate post hoc explanations. These explanations may explain the predictions without clarifying the mechanisms by which the model works. For example, verbal explanations generated by people or saliency maps used to analyze deep neural networks. Thus, despite the black box nature of the human brain, human decision making may recognize post hoc interpretability, revealing a contradiction between two popular notions of interpretability.This is one of the reasons why people do not trust AI diagnostic systems enough.
In model-based testing, the purpose of statistical testing is to estimate the probability that the assigned label is the correct label, but is not applicable to multiple tests.We have reflected on our last assignment. We plotted the correlation matrix for different data sets, but did not elaborate on the selection of thresholds. Also we were missing a discussion of the association of the assessment system with healthcare. The selection of thresholds mainly relied on plotting a scatter plot of the two features with the strongest correlation and then partitioning the different aggregation regions to find the corresponding thresholds.

Summary of lectures

Yahui Wu

Game Playing Systems
February 20, 2023
In the last class, we discussed about game playing systems. Game playing systems could be traced back to the 1700s when a automation chess player was invented but was revealed to be a fraud that the so called "automation" was achieved by a player hiding in the box. The real earliest AI game playing system is the chess computers designed in the 1900s and AI game playing systems are widely used in many computer games today. To apply AI to game playing systems, we need to formalize games we want to play. Zero-sum is a common form of games where one player winning implies one player losing and Tic-Tac-Toe is a typical zero-sum game. To design a good game playing system, we should try to find out the optimal action that improves our chances of winning given different situations. Search trees could be helpful since it traverses all the possible states of a game. In the search tree, each node represents a certain state of the game as an action leading to the state was made. The state sometimes leads to victory, defeat or tie and the value of the state can be evaluated by this. Basically, a search tree can be used to enumerate possible futures and to find the best action leading to the good end can be found by backtracking the tree. However, the probability of winning given an action in the current state varies depends on the opponent. If our opponent used to be optimal, we need to minimize the maximum success of our opponent. We use μ to represent our opponent's policies and π to represent our policies. Let $\hat{R}(\mu, \pi)$ denote the success rate of μ versus π. Minimax optimization could be written as $\min_{\pi} \max_{\mu} \hat{R}(\mu, \pi)$. However, when playing more complex games, exhaustive search like minimax method is not feasible. Monte-Carlo tree search (MCTS) can be used to deal with complex games by selecting actions randomly rather than searching exhaustively. MCTS consists of four steps - selection, expansion, simulation and backpropagation. At the very beginning of each time's search, we have a certain state which is the root node of the tree and we traverse down until an unexplored child node is reached, Then we expand the selected node and do the roll-out according to a certain policy. After the roll-out, we backpropagate the outcome through the tree until the root node is reached. The expanded node is marked as visited and once a selected node is fully visited, the next selected node is the node with higher statistics. The statistics of the node could be calculated according to upper confidence trees(UCT) whose function is $UCT(V_i, V) = \frac{Q(v_i)}{N(v_i)} + c\sqrt{\frac{logN(v)}{N(v_i)}}$ where $Q(v_i)$ is the reward of this selected node, $N(v_i)$ is the number of visits of the selected node and $N(v)$ is the number of visits of the root node. The equation trades off the reward and exploration of unvisited nodes.

Tianshuo Xiao

Game Playing Systems
February 20, 2023
In the last lecture, we learned about the game system. We mainly learned the history and algorithms of the development of game systems, such as Monte Carlo tree search, and finally the development of machine learning aspects.
The most classic zero-sum game is Tic-Tac-Toe. This game is essentially about actions, so our goal is to choose actions that improve our chances of winning, and in order to win, we consider the outcome of the next step when choosing our actions. First, we listed the branching paths and choose the most appropriate one. We set up terminal nodes through a search tree formalism, where a final state is reached at a certain point in the tree, and then calculate the value in that state. By identifying good futures (wins), we can backtrack and compare different nodes.
In a confrontation, in order to ensure success against the best opponent one needs to minimize the probability of the opponent's maximum success. Therefore, we need to calculate the future success

rate of each action and act accordingly. This require minimax optimization $\min_\pi \max_\mu \bar{R}(\mu, \pi)$. In our design, the decision process describes an agent. We need to take action to enter a certain state by a rule that is optimized by setting a reward function.But the exhaustive approach cannot always enumerate all possible states in some cases.

The MCTS algorithm is divided into four main steps, which are selection, expansion, simulation, and backtracking. The selection, which starts from the root node, recursively selects the optimal child node and finally reaches a leaf node, which we assign according to the following formula $\mathrm{UCB1}\left(S_i\right) = \overline{V_i} + c\sqrt{\frac{\log N}{n_i}}$. The application of UCB to search trees is called UCT. $\mathrm{UCT}\left(v_i, v\right) = \frac{Q(v_i)}{N(v_i)} + c\sqrt{\frac{\log N(v)}{N(v_i)}}$. Extension, if the current leaf node is not a terminating node, creates one or more child nodes and selects one of them for extension.Simulations, starting from an extension node, run the output of a simulation until the end of the gaming game. For example, if ten simulations are run from that extension node and the final victory is nine times, then the extension node will have a higher score and the other way around will be lower. Backtracking, using the results of the third simulation step, reverberates the propagation to update the current action sequence.

In early AI for Backgammon and Chess, points were assigned manually to non-terminal states, now we use machine learning to learn the value of states like Non-Markov long games.