TrojanGo

Software Requirement Specification

Version 1.0 approved

Prepared by @trojanGo-team

June 13th 2020

Table of Contents

1 . Introduction

We are trying to build a Game of Go[1] similar to AlphaGo Zero[2]. Our algorithm should be working for all board sizes but we would be testing for 5*5, 9*9, 13*13 and 19*19.

1.1 TrojanGo Goal

What would be different from AlphaGoZero?

1. **We will be trying with different input features and see how it is behaving.**
2. **Evaluating different neural network design: We will try CNN, ResNet and Inception network for the experiment purpose and record the performance for the comparison.**
3. **Monte Carlo Tree Search : Rather than taking Reward/Penalty as +1/-1 for all game state, can we take the average of value (which MCTS simulation is giving) for the Z_t or Z_t to be a function of final reward(+1/-1) and average value which we got during the MCTS simulation.**
4. **Evaluating TD(λ)[3] algorithm instead of MCTS for the game simulation?**

1.2 Additional Features

Extra Additional Features:

1. **Our project will have an in-built tool for visualizing the neural network.**
2. **Our bot will be supporting Go-Text-Protocol (GTP) for playing against any open source Go player like GNUGO,Pachi, etc.**

2 . TrojanGo Domain Knowledge

2.1 Big Idea

Define all the base classes and functions needed for playing a game of Go. This is like an infra for the entire project to follow and build modules on top of it.

Class Agent:

“”” Agent could be any bot, RandomBot or NeuralNetwork which will be selecting a move and returning it.
“””

```
bot = RandomBot(Agent)
gamestate = GameState.new_game(BOARD_SIZE)
gamestate.next_player = Player(Player.black.value) # set player as black

move = bot.select_move(gamestate) # move is a Move object and gamestate is a GameState object.
gamestate = gamestate.apply_move(move) # apply_move is a GameState function
```

class GameState:

```
def __init__(self, board, next_player, previous, last_move):
    self.board = board                  # <1> Board
    self.next_player = next_player      # <2>  Player
    self.previous_state = previous      # <3>  GameState
    self.last_move = last_move          # <4>  Move


# <1> board          : What is the current board
# <2> next_player    : Who's going to make the next move.
# <3> previous_state : What was the previous GameState. Or can be referred to as Parent.
# <4> last_move      : Last move played (Move.point)
```

apply_move: Now that you have move (say G5 or Point(rows=2, cols=2)) selected (selected by the Agent), you want to apply on the board. So, it will call GoBoard.place_stone who will take care of placing the stone on the current board.

```
def apply_move(self, move):
        “””
        Input Params
          move : Move class.
```

```
            Return Params
              return the new GameState object after applying the move.
              Internally it will call Board.place_stone.
            """
```

new_game : To start a new game of Go, call this function.

```
def new_game(cls, board_size):
    """
        Input : GameState(self), board_size
        Return: new GameState object
    """
```

is_suicide : Check whether the player's move is a suicide or not.

A player may not place a stone such that it or its group immediately has no liberties, unless doing so immediately deprives an enemy group of its final liberty. In the latter case, the enemy group is captured, leaving the new stone with at least one liberty

```
def is_suicide(self, player, move):
        """
                Input : GameState(self), Player, Move
                Return: bool
        """
```

violate_ko : Check whether the player's move is violating KO rule or not. We are checking KO for the last 8 history GameState. If we see the current player's move is leading to one of the previous board then return 'True' meaning it is violating KO rule.

```
def violate_ko(self, player, move):
        """
                Input : GameState(self), Player, Move
                Return: bool
        """
```

legal_moves: Given the GameState, Return all the legal moves possible. It will internally call whether a given move is a valid move or not (is_valid_move) and add all such moves in a list and return it.

```
def legal_moves(self):
        """
                Input : GameState(self)
                Return : a list of Move objects.
        """
```

is_valid_move : Check whether the given move is valid or not. Internally it will check whether the point is valid or not in terms of board.grid size, whether the move(point) is it's own eye or is a suicide, whether it is violating KO rule, etc.

```
is_valid_move(self, move):
        """
                Input : GameState(Self), Move
                Return: bool
        """
```

is_over : Check whether the game is over or not. Check for pass, resign or max number of moves played, etc.

```
def is_over(self):
        """
                Input: GameState(self)
                Return: bool
        """
```

winner : Declare the winner of the game. Use chinese rules to declare the winner.
AlphaGo Zero uses **Tromp-Taylor scoring**[4] during MCTS simulations and self play training. This is because human scores (Chinese, Japanese or Korean rules) are not well defined if the game terminates before territorial boundaries are resolved. However, all tournament and evaluation games were scored using Chinese rules.

```
def winner(self):
        """
                Input: GameState(self)
                Return: Player.black or Player.white or draw
        """
```

class GoBoard:

```
    def __init__(self, board_width, board_height, moves = 0):
      self.board_width = board_width
      self.board_height = board_width
      self.moves = moves                    # <1>
      self.grid = np.zeros((board_width, board_width))
      self.komi = 0                         # <2>
      self.verbose = True                   # <3>
      self.max_move = board_width * board_height * 2 # <4>

# <1> Number of moves played till now.
# <2> placeholder of komi, will see it later on how to use
```

<3> Keeping for debugging purposes, just a knob for enabling/disabling verbose logs.
<4> The max moves allowed for a Go game, Games terminate when both players pass or after 19 × 19 × 2 = 722 moves.

place_stone : Given Board, Player and Point, place the stone on the current board if the point is valid. **After keeping the stone, remove the dead stones** (take care of how you want to keep liberty for each stone or group of stones, do you want to calculate **liberty** for all on the fly or want to store it after each move and update at later point when it is affected by new move/stone)
If any opposite stones now have zero liberties, remove them.

```
def place_stone(self, player, point):
    “””
        Input : GoBoard(self), Player, Point
        Return: None
    “””
```

is_on_grid : Given Point, check if it is within the board sizes.

```
def is_on_grid(self, point):
    “””
        Input: Point
        Return: bool
    “””
```

update_total_moves : Update total number of moves played till now for the current game.

```
def update_total_moves(self):
    “””
        Input: GoBoard(self)
        Return: None
    “””
```

display_board : print the board

```
def display_board(self):
    “””
        Input: GoBoard(self)
        Return: None
    “””
```

Class Point:

Point(rows, cols)

def neighbors(self): Return all 4 neighbors of the point. These neighbors can be Invalid too (means out of grid), so it is the responsibility of the caller to take care of it.

Class Player:

```
class Player(enum.Enum):
    black = 1
    white = 2

def opp(self):   return opposition stone color
```

Class Move:

```
def __init__(self, point=None, is_pass=False):
    self.point = point
    self.is_play = (self.point is not None)
    self.is_pass = is_pass
    self.is_selected = False
```

play: Given Point object, this function will return Move object.

```
@classmethod
def play(cls, point):
    return Move(point=point)
```

pass_turn : It is return Move object with “is_pass” boolean set to True

```
@classmethod
def pass_turn(cls):
    return Move(is_pass=True)

@classmethod
def ply_selected(cls):
    return Move(is_selected = True)
```

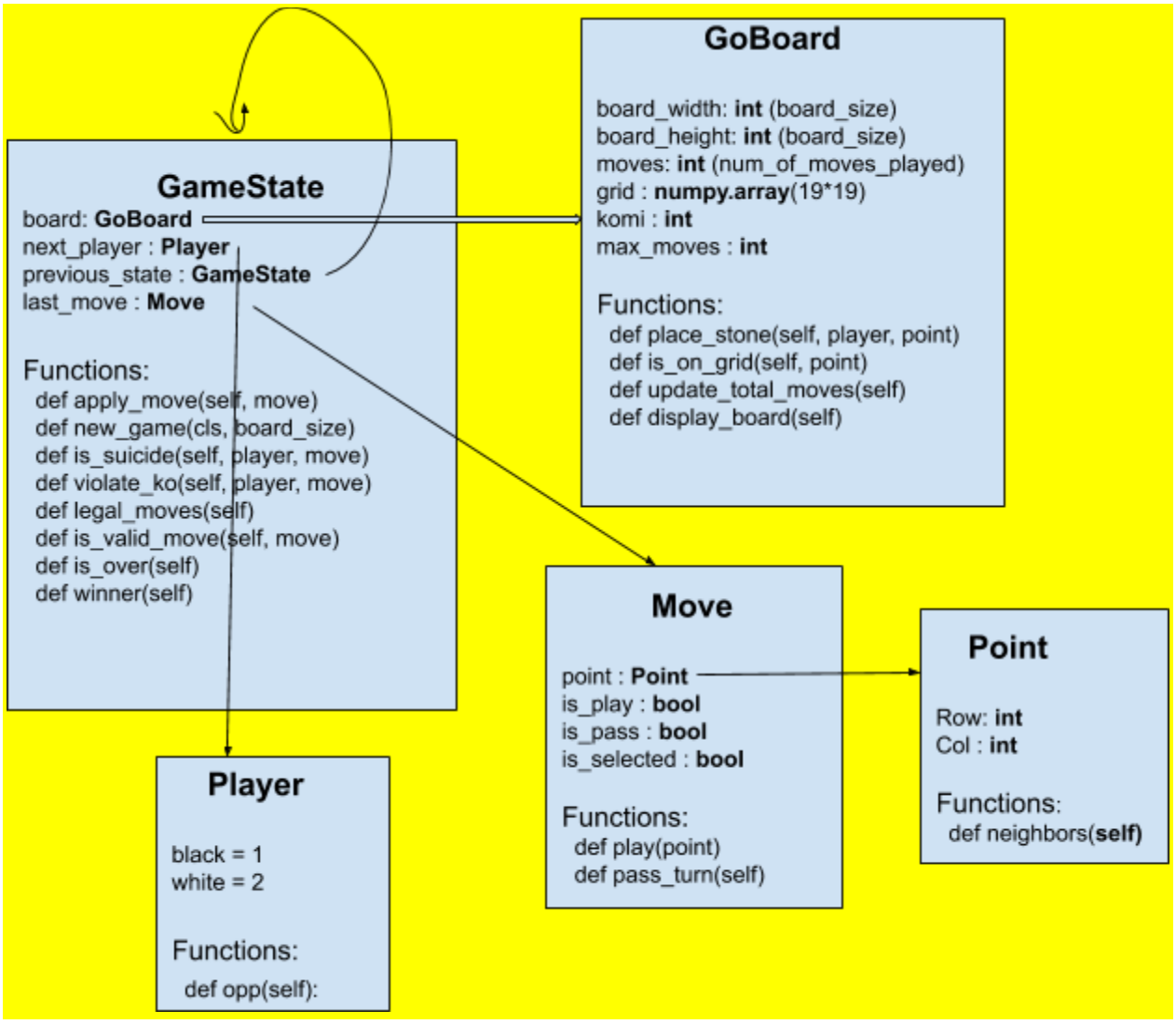

Fig 2.1 showing the TrojanGo defined Go specific classes and functions.

3 . Input Features

3.1 Big Idea

Build an input feature stack of dimension 17*19*19 for the neural network. The input feature stacks can be concatenated in different ways. The two common ways will be …

s{t} = [C, X{t=2}, X{t=1}, X{t=0}, Y{t=2}, Y{t=1}, Y{t=0}]
s{t} = [C, X{t=2}, Y{t=2}, X{t=1}, Y{t=1}, X{t=}, Y{t=0}]

We will be evaluating both the input feature stacks for the performance.

<trojangoPlane.py>

Assuming 5*5 go board size.

Given: given a board state, generate the input feature stacks/planes (in this case 7*5*5)
C = Plane[0] = current player (if Black turn then all ones, if white turn then all zeros)

X{t=2} = Plane[1] = current player all moves made till now, say (i)th move with all ones & others as zeros (self 1st history)
X{t=1} = Plane[2] = current player all moves made till (i-1)th move with all ones & others as zeros (self 2nd history)
X{t=0} = Plane[3] = current player all moves made till (i-2)th move with all ones & others as zeros (self 3rd history)

Y{t=2} = Plane[4] = opposition player all moves made till now (say jth move) with all ones & others as zeros (opp 1st history)
Y{t=1} = Plane[5] = opposition player all moves made till (j-1)th move with all ones & others as zeros (opp 2nd history)
Y{t=0} = Plane[6] = opposition player all moves made till (j-2)th move with all ones & others as zeros (opp 3rd history)

NOTE: if (i-n)th or (j-n)th move doesn't exist then the plane will have all zeros.
AlphaZero: These planes are concatenated together to give input features s{t} = [X{t}, Y{t}, X{t−1}, Y{t−1},..., X{t−7}, Y{t−7}, C].
board_tensor, s{t} = [C, X{t=2}, X{t=1}, X{t=0}, Y{t=2}, Y{t=1}, Y{t=0}]
</>

4 . Neural Network

4.1 Big Idea

Evaluating different neural network design: We will try CNN, ResNet and Inception network for the experiment purpose and record the performance for the comparison.

5 . MCTS

5.1 Big Idea

6 . GTP (Go Text Protocol)

6.1 Big Idea

Initialize a bot from an agent. We play until the game is stopped by one of the players. At the end we write the game to the provided file in SGF format. Our opponent will either be GNU Go or Pachi. We read and write GTP commands from the command line.

7 . Visualization Tool

7.1 Big Idea

8. Web Development

Codes:

Historical Go Game Record: https://u-go.net/gamerecords/
https://u-go.net/gamerecords-4d/

Our Code https://github.com/Go-Trojans/trojan-go

Go ranks and ratings

https://en.wikipedia.org/wiki/Go_ranks_and_ratings

Rank Type	Range	Stage
Double-digit *kyu* (級,급) (*geup* in Korean)	30–20k	Beginner
Double-digit *kyu* (abbreviated: DDK)	19–10k	Casual player
Single-digit *kyu* (abbreviated: SDK)	9–1k	Intermediate amateur
Amateur *dan* (段,단)	1–7d	Advanced amateur
Professional *dan* (段,단)	1–9p	Professional Player

Go Online Servers

We can submit our bot (uses GTP protocol mainly) and they can play against other online bots/human-players or can play tournaments.

OGS , IGS, KGS
https://en.wikipedia.org/wiki/Internet_Go_server

Hardware Requirements

Refer Appendix D
AWS: Deep Learning Base AMI (Ubuntu)
Deploying and Hosting : t2.small Instance
Training and Learning : p2.xlarge

P2 Instance Details : https://aws.amazon.com/ec2/instance-types/p2/
1 GPU, 4 vCPUs, 61 GiB RAM, $0.9 / hour

We can explore Google Cloud Platform …

References

1. https://en.wikipedia.org/wiki/Go_(game)
2. https://www.nature.com/articles/nature24270.epdf?author_access_token=VJXbVjaSHxFoctQQ4p2k4tRgN0jAjWel9jnR3ZoTv0PVW4gB86EEpGqTRDtpIz-2rmo8-KG06gqVobU5NSCFeHILHcVFUeMsbvwS-lxjqQGg98faovwjxeTUgZAUMnRQ
3. https://towardsdatascience.com/reinforcement-learning-td-%CE%BB-introduction-686a5e4f4e60
4. Tromp-Taylor rules, http://tromp.github.io/go.html