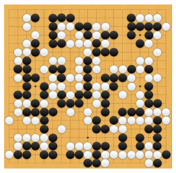
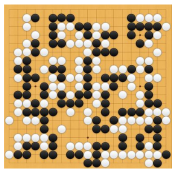



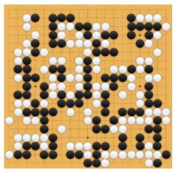
TrojanGo

Software Requirement Specification

Version 1.0 approved

Prepared by @trojanGo-team

June 13th 2020



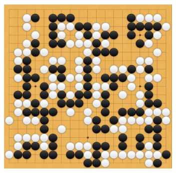
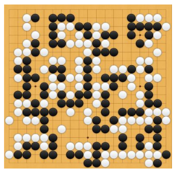


Table of Contents



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?**
5. **Training in a distributed environment.**

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.

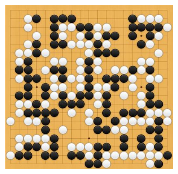
```
    def is_over(self):
        “””
                Input: GameState(self)
                Return: bool
        “””
```

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)
```

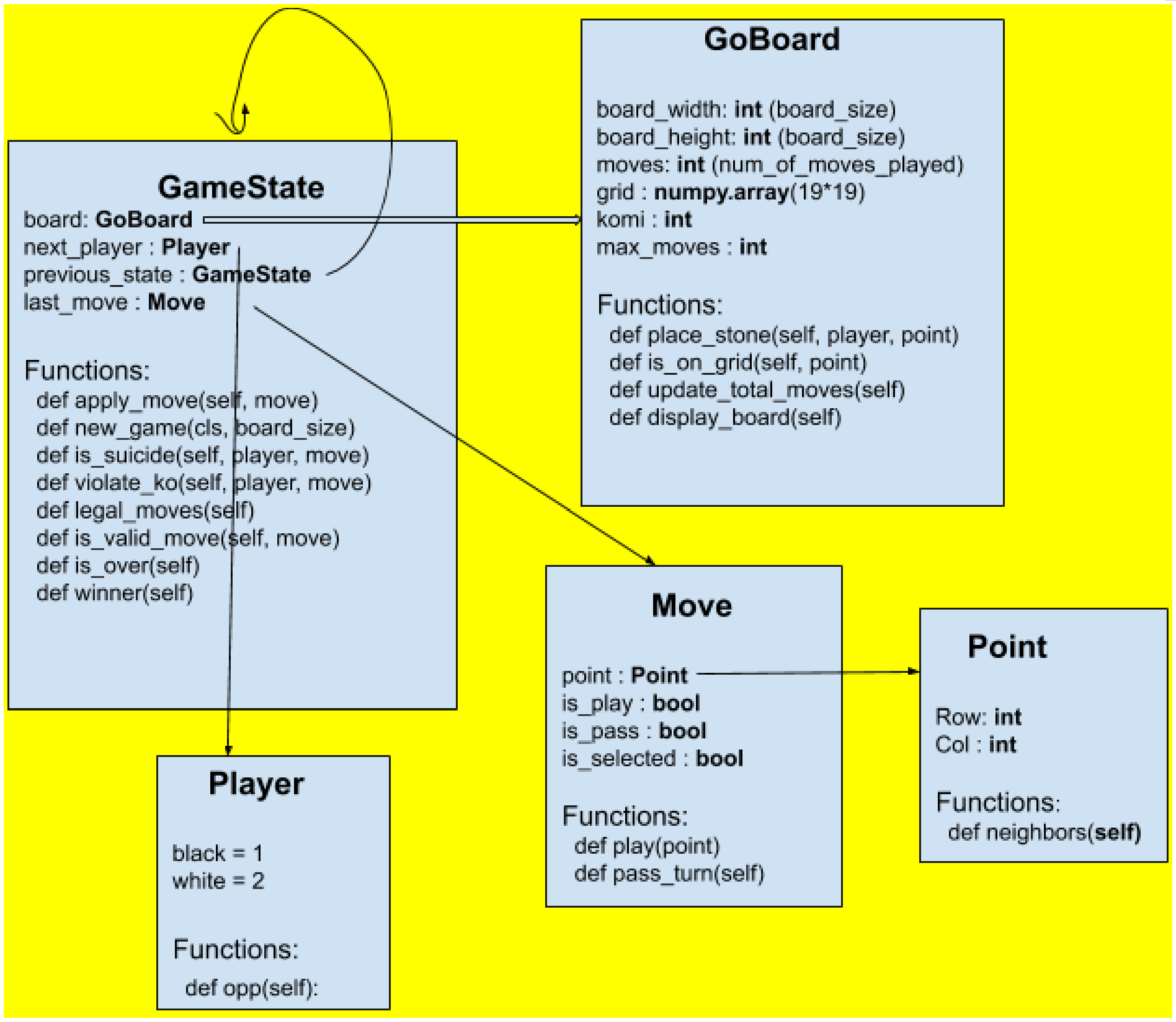


Fig 2.1 showing the TrojanGo defined Go specific classes and functions.

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.

4.2 Training at Scale

Training the network at scale presents a challenge since the hardware required could be expensive using a cloud provider. The best way to do distributed training with Tensorflow/Keras on a machine with multiple GPUs or multiple machines is via the tensorflow.distribute.strategy API. From what I've read, there are multiple strategies to distributed training, but the MirroredStrategy seems to be the best fit. The way this works is by keeping a copy of the current network on each of the devices (GPUs or separate machines) that you wish to train on and then after training, you combine the gradients from all of them to update the network using an algorithm called Allreduce. The Allreduce algorithm can use any reduction function (sum, multiplication, average etc) and Tensorflow has implementations for various allreduce functions already. The default is Nvidia NCCL allreduce. Clusters of computers can then be defined/set using the TF_CONFIG variable.

Google cloud seems to have the best integration options for this since they have the AI Platform which can help take care of a lot of the details in setting up the training pipeline. All we would really have to do is specify how many instances and what kind of hardware we would need. There is a pricing calculator for Google Cloud services, but AI Platform running for an entire week non-stop would roughly come out to be a little over $300.

Resources:
https://www.tensorflow.org/guide/distributed_training#mirroredstrategy
https://cloud.google.com/ai-platform/training/docs/distributed-training-containers
https://cloud.google.com/products/calculator#id=c26dadc5-e17e-4251-83dc-2983a0d18602
https://keras.io/guides/distributed_training/#multiworker-distributed-synchronous-training

5 . MCTS

5.1 Big Idea

Self-play training pipeline. AlphaGo Zero's self play training pipeline consists of **three main components, all executed asynchronously in parallel.**

- Neural network parameters θ_i are continually optimized from recent self play data;
- AlphaGo Zero players $\alpha_{\theta i}$ are continually evaluated; and
- the best performing player so far, $\alpha_{\theta *}$, is used to generate new self play data.

NOTE: Currently, we are going to evaluate for 5*5 board size, so the given hyper-parameters are applied for 5*5 boards. We will evaluate the hyper-parameters for other board sizes once we get something concrete for 5*5.

Parallel: Determine number of CPU cores and split workload among them
Average game length for 5*5 board = **15** and 19*19 board is **200** [5]

Neural Network: We will be using 4 layers CNN architecture for this (small_nn.py).

We have an initial model with randomized weights which outputs policy (***P***) and value (***V***) head. Hence we are good to go with self-play.

MCTS will be using the initial model for simulating the self-play games. The games should be played **parallely**. **In each iteration**, we will play 2,500 games (400 simulations in MCTS to select each move) before starting the training.

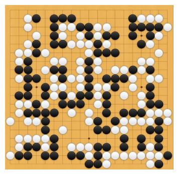
Make sure, self-play is still going on (means next iteration of 2,500 games and **so on** until we have the next strong model available) as we may need more self-play games to find the next strong model. We may not use all the CPUs available as we need it for evaluation purposes too.

Training

We will be **training** the model (on GPU) with **batch_size** = 128 (for 19*19 board size, AGZ uses 2048) and **mini-batches** of X (X = number of examples from 2,500 self-play games / 128).

Once training is done for ith iteration of self-play games (with 2,500 self-play games in each iteration), checkpoint this model (say **checkpoint_model**$_i$) as it will be used for evaluation and ***continue training*** (with another 2,500 self-play games that it generated in (i+1)th iteration).

[AGZ = **700,000 mini-batches of 2048 positions]**



Evaluator: evaluate the current best model (first time, it will be an initial random model) with saved/checkpointed model (**checkpoint_model$_i$**) by playing 400 games (**play the games parallely, 5 seconds time per move in the game**) and update the new trained model as the strongest/current model only if it wins with a margin > 55 %.

If it doesn't win (with margin > 55 %) then use the next checkpointed model (this will be trained on 2,500 + additional 2,500 = 5,000 self-play games and so on) for the 400 games competition and see if it's winning or not and this process will continue until we get the next better model.

Now that we have our new model, use this model to generate next sets of self-play games and then train the model with the new set of data available.

Evaluator. To ensure we always generate the best quality data, we evaluate each new neural network checkpoint against the current best network $f_{\theta *}$ before using it for data generation. The neural network $f_{\theta i}$ is evaluated by the performance of an MCTS search $\alpha_{\theta i}$ that uses $f_{\theta i}$ to evaluate leaf positions and prior probabilities (see Search algorithm). Each evaluation consists of 400 games, using an MCTS with 1,600 simulations to select each move, using an infinitesimal temperature $\tau \to 0$ (that is, we deterministically select the move with maximum visit count, to give the strongest possible play). If the new player wins by a margin of >55% (to avoid selecting on noise alone) then it becomes the best player $\alpha_{\theta *}$, and is subsequently used for self play generation, and also becomes the baseline for subsequent comparisons.

5.2 MCTS Algorithms and Self-play

<Puranjay and Raveena to update this section in details>

```
class MCTSNode:
    """ A node in the game tree. Note wins is always from the viewpoint of
playerJustMoved.
    """
    def __init__(self, state, move = None, parent = None):

        self.move = move # the move that got us to this node - "None" for the
root node
        self.state = state # GameState object that the node represents
        self.parentNode = parent # "None" for the root node
        self.childNodes = []
        self.wins = 0
        self.visits = 0
        self.untriedMoves = state.GetMoves() # future child nodes
        self.playerJustMoved = state.playerJustMoved # the only part of the
state that the Node needs later
```

```
        self.q = q # q value of the node

Add function for storing input feature stack on disk

    def SelectChild(self):
        """ Use Q+U to select action
        """
        # returns s which is a sorted list of child nodes according to win
formula

    def expand(self, m, s):
        """ Remove m from untriedMoves and add a new child node for this move.
            Return the added child node
        """

    def update(self, result):
        """ Update this node - one additional visit and result additional wins.
result must be from the viewpoint of playerJustmoved.
        """

    def UCT(self):
      """ Returns UCT Score for node
        """

def uct_select_move(simulations, verbose = False):
    """ Conduct a tree search for itermax iterations starting from rootstate.
        Return the best move from the rootstate.
        Assumes 2 alternating players (player 1 starts), with game results in
the range [-1,1]."""
        # Select
# node is fully expanded and non-terminal

        # Expand
        # if we can expand (i.e. state/node is non-terminal)
        # add child and descend tree

        # Rollout - this can often be made orders of magnitude quicker using a
state.GetRandomMove() function
        # while state is non-terminal

        # Backpropagate
# backpropagate from the expanded node and work back to the root node
# state is terminal. Update node with result from POV of node.playerJustMoved


    # Output some information about the tree - can be omitted

# return the move that was most visited
```

```
class MCTSPlayer :

      def __init__(self,player,encoder,save_file):

            self.player = player
            self.encoder = encoder
            self.save_file = save_file

      def save_move(self, move,search_prob,winner=None) :
      “”” Save input feature stack, search probabilities,and game winner to
disk “””
            Data is saved using a namedtuple and saved to an hdf5 file:

            mctsMove = namedtuple(“MCTSMove”,[‘state’,’search_prob’,’winner’])
            # search prob is a list of tuples of the form (Move,prob)
            # state is the encoded game state
            # winner is an int (-1 or 1) representing the game winner
      def select_move(self,state,saveToDisk=False) :
             """ Takes in a GameState object representing the current game
state and selects the best move using MCTS. Returns a Move object. Provides
optional parameter to save the move to disk (for use in self-play for training
neural network)
        """



class MCTSSelfPlay :

      def __init__(self,encoder):

            self.encoder = encoder

      def play(self,num_game=25000) :
            “”” Play num_games of self play against two MCTS players and save
move information to disk. “””
```

<Take care of Dirichlet noise, and *τ value*>

#### Below stuffs are directly copied from AlphaZero Paper. Use it as per need.

Self-play. The best current player *aθ*∗, as selected by the evaluator, is used to generate data. In each iteration, *aθ*∗ plays **25,000 games of self play, using 1,600 simulations** of MCTS to select each move (this requires approximately 0.4 s per search). For the first 30 moves of each game, the temperature is set to *τ* = 1; this selects moves proportionally to their visit count in MCTS, and ensures a diverse set of

positions are encountered. For the remainder of the game, an infinitesimal temperature is used, $\boldsymbol{\tau \to 0}$. Additional exploration is achieved by adding Dirichlet noise to the prior probabilities in the root node $s0$, specifically $P(s, \text{a}) = (1 - \varepsilon)pa + \varepsilon\eta a$, where $\eta \sim \text{Dir}(0.03)$ and $\varepsilon = 0.25$; this noise ensures that all moves may be tried, but the search may still overrule bad moves. In order to save computation, clearly lost games are resigned. The resignation threshold $v\{\text{resign}\}$ is selected automatically to keep the fraction of false positives (games that could have been won if AlphaGo had not resigned) below 5%. To measure false positives, we disable resignation in 10% of self play games and play until termination.

Search algorithm. AlphaGo Zero uses a much simpler variant of the asynchronous policy and value MCTS algorithm (APVMCTS) used in AlphaGo Fan and AlphaGo Lee.

Each node s in the search tree contains edges (s, a) for all legal actions $a \in A(s)$. Each edge stores a set of statistics,

$$\{N(s, a), W(s, a), Q(s, a), P(s, a)\}$$

where $N(s, a)$ is the visit count, $W(s, a)$ is the total action value, $Q(s, a)$ is the mean action value and $P(s, a)$ is the prior probability of selecting that edge. Multiple simulations are executed in parallel on separate search threads. The algorithm proceeds by iterating over three phases (Fig. 2a-c), and then selects a move to play (Fig. 2d).

Select (Fig. 2a). The selection phase is almost identical to AlphaGo Fan12; we recapitulate here for completeness. The first in-tree phase of each simulation begins at the root node of the search tree, $s0$, and finishes when the simulation reaches a leaf node sL at time step L. At each of these timesteps, $t < L$, an action is selected according to the statistics in the search tree, $at = \text{argmax}(Q(st, a) + U(st, a))$, using a variant of the PUCT algorithm24, $a\ \sum b\ N(s, b)$

$$U(s,a)=cpuctP(s,a)\ 1+N(s,a)$$

where $cpuct$ is a constant determining the level of exploration; this search control strategy initially prefers actions with high prior probability and low visit count, but asymptotically prefers actions with high action value.

Expand and evaluate (Fig. 2b). The leaf node sL is added to a queue for neural network evaluation, $(di(\mathbf{p}), v) = f\theta(di(sL))$, where di is a dihedral reflection or rotation selected uniformly at random from i in [1..8]. Positions in the queue are evaluated by the neural network using a minibatch size of 8; the search thread is locked until evaluation completes. The leaf node is expanded and each edge (sL, a) is initialized to $\{N(sL, a)=0, W(sL, a)=0, Q(sL, a)=0, P(sL, a)=pa\}$; the value v is then backed up.

Backup (Fig. 2c). The edge statistics are updated in a backward pass through each step $t \le L$. The visit counts are incremented, $N(st, at) = N(st, at) + 1$, and the action value is updated to the mean value, $W(st, at) = W(st, at) + v$, $Q(st, at) = W(st, at)$

We use virtual loss to ensure each thread evaluates different nodes.[check the references given in the paper]

Play (Fig. 2d). At the end of the search AlphaGo Zero selects a move *a* to play in the root position *s{0}* , proportional to its exponentiated visit count,

$\pi(a|s0)=N(s0,a)1/\tau/\sum 0N(s0,b)1/\tau$,where τ is temperature parameter that controls the level of exploration.

The search tree is reused at subsequent time steps: the child node corresponding to the played action becomes the new root node; the subtree below this child is retained along with all its statistics, while the remainder of the tree is discarded. AlphaGo Zero resigns if its root value and best child value are lower than a threshold value v{resign}.

Compared to the MCTS in AlphaGo Fan and AlphaGo Lee, the principal differences are that AlphaGo Zero does not use any rollouts; it uses a single neural network instead of separate policy and value networks; leaf nodes are always expanded, rather than using dynamic expansion; each search thread simply waits for the neural network evaluation, rather than performing evaluation and backup asynchronously; and there is no tree policy. A transposition table was also used in the large (40 blocks, 40 days) instance of AlphaGo Zero.

5.3 Training and Evaluation

Assuming the experience buffered data is stored on disk in .h5 file, extract model_input, action_target, and value target from the file as a numpy array.

train : Read the examples from file on disk(assuming input as a numpy array of dimension 7*5*5, action_target as a numpy array of dimension (26,1) and value target as a numpy 1D array). Now train the model with different hyper-parameters.

This method will be checkpointing the model after every iteration. The caller of this function will take care of calling it (“train”) after each iteration (as the data is available from self-play games) and using the checkointed model for the evaluation.

If the experience data is big then we will use the “**generator**” concept of python so that it will load the game data in memory as per our need.

```
def train(self, experience, batchSize=128, lr_rate=0.01):
    “”” experience : ith iteration self-play games. Total for 5*5 board it is 2,500 games
    “””
    model_input = []

    num_of_examples = examples in the experience data

    # dummy code, just for understanding
    for _ in range(10000):
```

```
    board_tensor = np.random.randint(0, 3, size=(7, 5, 5))
    model_input.append(board_tensor)

model_input = np.array(model_input)


action_target = []
for _ in range (10000):
    search_prob = np.random.randn(5,5)
    search_prob_flat = search_prob.reshape(25,)
    action_target.append(search_prob_flat)

action_target = np.array(action_target)


value_target = np.random.rand(10000)
value_target = np.array(value_target)

from keras.optimizers import SGD
model.compile(SGD(lr=lr_rate), loss=['categorical_crossentropy', 'mse'])

model.fit(model_input, [action_target, value_target], batch_size=batchSize, epochs=1)

# Checkpoint the model here ...
```

evaluate: This function is responsible for selecting who’s best (bot2 if win margin > 55%). Caller of this function is responsible for calling it multiple times (means after every iteration to evaluate checkpoint_model$_i$) until we get our new strongest bot.

```
def evaluate(self, bot1, bot2, num_games=400):
        “”” bot1 : Current best model
             bot2 : checkpointed model
            Num_games: Number of games to be played b/w them
        “””
```

5.4 Optimization

Big Idea: To utilize available CPUs and GPUs to the fullest. Don’t keep them Idle.

We will explore the idea to optimize **self-play & MCTS simulation**, **Training the model**, and **evaluating the bots/models**.

We will talk about multiprocessing, multithreading, distributed systems, etc for speeding up the self-play training pipelines.

MCTS and Self-play : We are exploring the use of '**virtual loss’** to simulate MCTS search and GPUs for model prediction.

Looks like we can’t use python multithreading for this problem because python has the concept of GIL which is shared across threads and at one time only one thread can use and others would be waiting for it. So, even if you have multiple cores, you can’t get the real benefit of multithreading.

So, if we want to get the benefits of multithreading then we need to use C, C++ language to code it.

Training: tensorflow API (tf.distribute.Strategy) will be used for parallel training.

TrojanGo Pipeline

Self-play and MCTS is generating examples (encoded GameStates, MCTS search probability, Reward) by playing games in parallel and using current best Model (M_C). MCTS is performed in 4 phases (Selecting Branch, Expanding, Backup, Play), in which the **'Expanding'** phase makes use of a model to predict prior probability and value of the given GameState. This prediction can be time taking if the neural network is deep as it has to perform forward-feed algorithms which need calculation with all the parameters of the neural network. ***The Idea here is to have a separate thread to perform this task on the GPU and not the CPU (GPU will be faster in performing these calculations).***

In each Iteration, self-play is playing 2500 games (for 5*5 board size) and saving the examples to a folder called "./data".

# alert_flag is just used to tell self-play that you keep generating the data with the current best Model (M_C).

```
While alert_flag:
    save examples to ./data/experience_i        # ith iteration experience examples
```

Training **module task is to train the** current best Model (M_C) with the help of experience data.
This module will be running on GPUs in parallel and checkpointing models to the checkpoint folder.

Training current best Model (M_C) on data/experience$_1$ -> ./checkpoint/checkpoint_model$_1$ (say M_{C1})
Training current best Model (M_C) on data/experience$_2$ -> ./checkpoint/checkpoint_model$_2$ (say M_{C2})
and so on …

Evaluate: loop through the checkpointed models and try to find a model which can win over the current best model (M_C) with 55 % win margin. Once we get a new best model, we can discard the previous stored data and ask **self-play** to generate a new set of data with the new model.
Also, we should inform training module to stop training old model (as we have got our new best model now) and also stop training on old data and ask them to train on newly generated sets of data with new model.

NOTE: These all 3 modules should be working asynchronously and not sequentially.

<David>

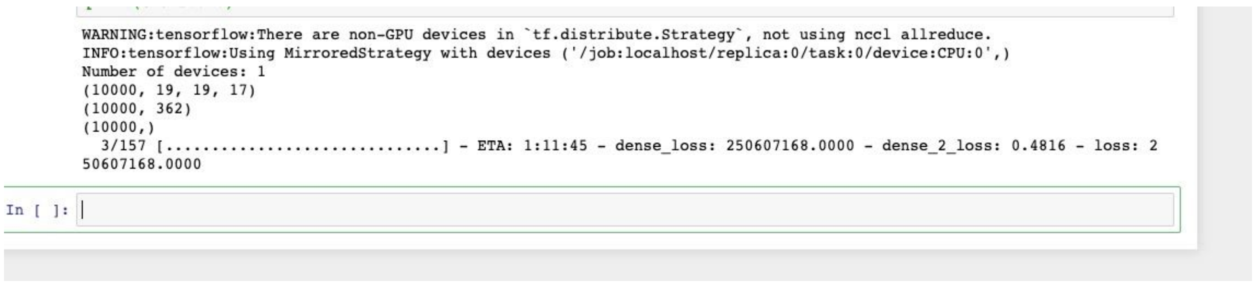
Model used : 20 layers Resnet model (40 layers CNN)

Distributed Training Results on AWS
Machine Instance (AMI)
Used the p3.8xlarge instance type. Specs can be found here:
https://aws.amazon.com/ec2/instance-types/p3/

In short, it has 4 GPUs (Tesla V100-SXM2)
P3 machines are not available in the Northern California region, so I had to use the Oregon region (west-2b).
Training was done with **100,000 instances of 17,19,19 random board states.**
The results are in separate files, but in summary:
Distributed Training over 4 GPUs: ~1min per epoch
Regular (without Mirrored Strategy): ~3min per epoch
The regular training without MirroredStrategy only used 1 GPU out of 4 available, as expected.

Log Files : uploaded on Slack FIles (dist_gup_usage, dist_results.txt, reg_reults.txt)

1 epoch time showing as 1 hour plus on 4 cores. (on Rupesh local laptop)



```
WARNING:tensorflow:There are non-GPU devices in `tf.distribute.Strategy`, not using nccl allreduce.
INFO:tensorflow:Using MirroredStrategy with devices ('/job:localhost/replica:0/task:0/device:CPU:0',)
Number of devices: 1
(10000, 19, 19, 17)
(10000, 362)
(10000,)
  3/157 [..............................] - ETA: 1:11:45 - dense_loss: 250607168.0000 - dense_2_loss: 0.4816 - loss: 2
50607168.0000

In [ ]: |
```

David’s local laptop: 4 hours for one epoch (6 cores, 2.6 GHz)

Puranjay Server’s report: you will keep trying on parallel training.

6 . GTP (Go Text Protocol)

6.1 Big Idea

Our agent should be able to play a match or tournament with other online go bot players like GNUGO and PACHI.
The thing we need is to use GTP for the communication and then simulate the game or tournament.

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.

GTP links :

http://www.lysator.liu.se/~gunnar/gtp/
https://web.archive.org/web/20191028082412/http://www.lysator.liu.se/~gunnar/gtp/

Evaluating the Playing Strength of TrojanGo

To evaluate TrojaGo, we ran an internal tournament among variants of TrojanGo and several other Go programs, including the strongest commercial programs Crazy Stone and zen, and the strongest open source programs **Pachi** and Fuego . All of these programs are based on high-performance MCTS algorithms. In addition, we included the open source program **GnuGo**,a Go program using state-of-the-art search methods that preceded MCTS. *All programs were allowed 5 seconds of computation time per move.*

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
5. https://en.wikipedia.org/wiki/Go_and_mathematics