Technion – Israel Institute of Technology

Project: Mini Alpha Go

Niv Goldberg & Yotam ishay

Supervised by Oron Anschel

April 2017

Abstract

In this work, we have implemented a simplified version of *AlphaGo* (by "Google DeepMind") – a computer program that uses the Monte-Carlo tree search algorithm combined with deep Neural Networks in order to play the game of "Go" at a master level.

Contents

1. Introduction

In recent years, AI programs have been beating human players in almost every "classic" board game (Connect Four, chess, etc.). The ancient Chinese game "Go", invented more than 2,500 years ago, was an exception. Due to its enormous search space and the difficulty of evaluating board positions and moves, the game has been viewed as the most challenging of classic games for AI.

On January 27th 2016, "Google DeepMind" published a paper called "Mastering the Game of Go with Deep Neural Networks and Tree Search". In this paper, a new approach to computer Go is introduced, followed by a new search algorithm. A computer program called "AlphaGo" used this search algorithm and achieved a 99.8% winning rate against other Go programs, and defeated the European Go champion and later the World champion, Lee Sedol.

In this paper, a simplified version of "AlphaGo", named "mini AlphaGo", will be presented.
Mini AlphaGo uses a new search algorithm, which will be described as follows, in order to play "Connect Four".

1.1. Monte Carlo tree Search algorithm

Both AlphaGo and mini AlphaGo are based on the same search algorithm – Monte-Carlo tree Search (MCTS). MCTS is based on many 'rollouts' – in each rollout, the game is played to its end. The final game result of each rollout is then used to weigh the nodes in the game tree so that better nodes are more likely to be chosen in future rollout. MCTS employs this principle by recursively repeating these four steps:

Selection: Start from the root (the current stage of the game) and select successive child nodes until you reach a node L with incomplete statistics (for example, a node with child nodes which were not yet revealed) or a terminal node (win/lose/tie).

Expansion: Unless L ends the game, create child nodes and choose node C from them.

Simulation: Play a rollout from node C.

Backpropagation: Use the result of the rollout to update information in the nodes of the path from C to R.

Sample steps from one round are shown in the figure below. Each tree node stores the number of won/played rollouts.

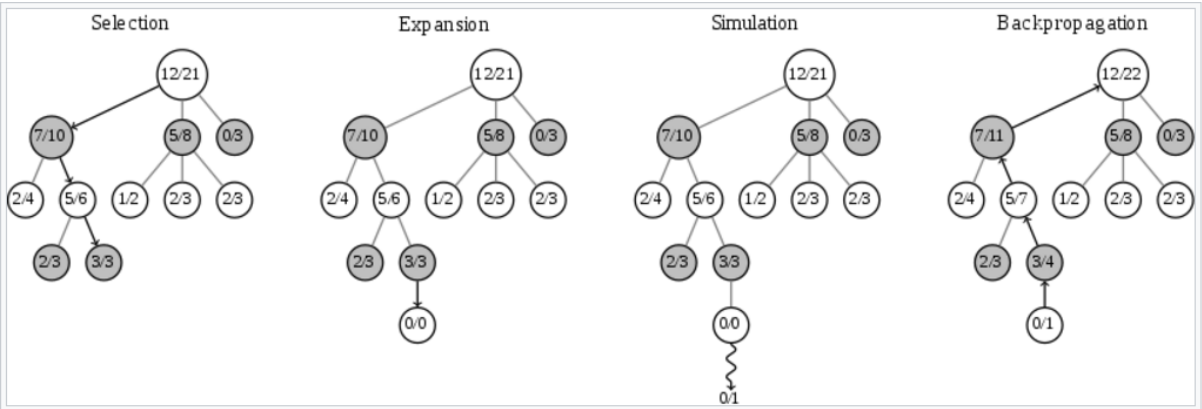

Figure 1: Steps of MCTS

Different versions of MCTS differ mainly by two things:

- The policy used to select actions during the search

  The way of selecting child nodes needs to maintain some balance between the *exploitation* of moves with high average win rate and the *exploration* of moves with few simulations.
  There are several formulas for balancing between the two. The most common is called *UCT* (Upper Confidence Bound 1 applied for trees), and is denoted by:

$$a = \text{Max}\left(\frac{w_i}{n_i} + c * \sqrt{\ln(t)\,/n_i}\right)$$

  Where

  a – action to be taken from the current node
  w_i – number of wins after the i-th move
  n_i – number of simulations after the i-th move
  c – exploration parameter (chosen empirically. Theoretically equal to $\sqrt{2}$)
  t – total number of simulations for the node considered

- The type of rollouts

  A distinction is made between *light rollouts* and *heavy rollouts*. Light rollouts consist of random moves, while heavy rollouts apply various heuristics in order to choose moves (for instance, employ expert knowledge of a given game).

Asymptotically, it has been shown that the MCTS algorithm converges to optimal play. The need for using different policies and rollouts is the result of the fact that the convergence time can be very slow (extremely slow in the case of playing 'Go').

1.2. AlphaGo

The strongest Go programs before AlphaGo were based on MCTS, enhanced by policies that were trained to predict human expert moves. These policies were used to narrow the search to a beam of high probability actions, and to sample actions during rollouts. This approach has only lead to strong amateur play.
In Google DeepMind's paper, it is suggested that the reason for the relatively poor performance was that this approach has been limited to shallow policies or to value functions based on a linear combination of input features.

AlphaGo uses a combination two types of networks:

1. Deep *value networks* – used to evaluate board positions.
2. Deep *policy networks* – used to sample actions during rollouts.

An ingenious idea used in AlphaGo was to pass the board position as a 19X19 image and use convolutional layers to construct a representation of the position. This approach – using deep convolutional neural networks (CNN) for training – has achieved great results, aided by recent developments of CNN in visual domains such as image classification and face recognition.

1.3. AlphaGo's policy

AlphaGo combines the policy and value networks in a MCTS algorithm that selects actions by lookahead search. Each edge (s,a) of the game search tree stores three values: an *action value* $Q(s,a)$, a *visit count* $N(s,a)$ and a *prior probability* $P(s,a)$.

The final policy used by AlphaGo to select actions is:

$$a_t = argmax_a \left(Q(s_t,a) + u(s_t,a)\right)$$

- $Q(s,a) = \frac{1}{N(s,a)} * \sum_{i=1}^{n} 1(s,a,i) * V(s_L^i)$.
  - n is the *budget* – the number of repeats of the above four steps of the MCTS algorithm (Selection, Expansion, Simulation and Backpropagation).
  - $1(s,a,i)$ is an indicator function and equals 1 if the edge (s,a) was traversed during the *i*-th simulation.
  - Followed by the definition of the indicator function, $N(s,a) = \sum_{i=1}^{n} 1(s,a,i)$.
  - s_L^i is the leaf node of the *i*-th simulation. A node L is evaluated in two different ways:
    1. A value network $v_\theta(s_L)$
    2. The outcome z_L of a random rollout using a fast rollout policy network p_π.

    These evaluations are combined, using a mixing parameter λ, into the leaf evaluation function - $V(s_L) = (1-\lambda) * v_\theta(s_L) + \lambda * z_L$.

- $u(s_t,a)$ is a *bonus value* and is proportional to $\frac{P(s,a)}{1+N(s,a)}$. $P(s,a)$ (the prior probability value) equals $p_\sigma(s|a)$, where $p_\sigma$is a policy network.

2. Mini AlphaGo

Mini AlphaGo (mAlphaGo) is a simplified version of AlphaGo.
It uses a new search algorithm which draws inspiration from several ideas and techniques used in the original search algorithm of AlphaGo.
As will be shown below, the basic concept of our search algorithm is to generalize the MCTS algorithm with the use of a deep policy CNN, in order to achieve optimal play in a shorter time period than the original MCTS (the UCT version).

2.1. Mini AlphaGo vs. AlphaGo

There are several basic differences between mAlphaGo and AlphaGo. The main ones are:

- AlphaGo plays Go. MAlphaGo plays a version of Connect Four - a 4X3 board of 'Connect Three' (The goal is to connect 3 checkers in a row/column/diagonal before the opponent).

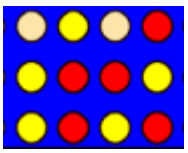

Figure 2: Connect3 board

The main reason for choosing to play 'Connect Three' over Go was that AlphaGo required tremendous computing power (its final version used 48 CPUs and 8 GPUs, and the distributed version exploited 1202 CPUs and 176GPUs). MAlphaGo uses only 1 CPU.
It is, however, quite easy to have mAlphaGo play any other board game (including Go) – all that is needed is to write a compatible class in python for the selected board game, plus a few minor adjustments to the CNN.

- AlphaGo trains the neural networks using a pipeline consisting of several stages of Supervised Learning (SL) networks and Reinforcement Learning (RL) networks, combines them (as was previously shown) to a policy and value networks and efficiently combines those networks with MCTS. MAlphaGo trains one SL policy network and combines it with MCTS.

2.2. Difficulties and Considerations

During the work on mAlphaGo we have encountered several difficulties:

- Knowledge gap in different fields of study and in technical abilities, which were relevant to mAlphaGo and to the understanding of Google DeepMind's paper (mainly SL theory, RL theory, programming in python and using Tensorflow).
- Choosing a suitable board game to be played by mAlphaGo – as was previously mentioned, AlphaGo used tremendous computing power. Using only 1 CPU, it was necessary to find a suitable game that was not too complicated, but also complicated enough to have a significant number of possible board states. Eventually, the game of Connect Four was chosen.

- Choosing the specific dimensions of Connect Four – Standard Connect Four board has the dimensions of 7X6. In these dimensions, there are over $4 * 10^{12}$ possible legal states of the board game. In order to train the NN, an order of magnitude of 10^5 states was chosen. Finally, the dimensions of 4X3 were selected. In this size, there are a bit more than $2.7 * 10^5$ possible legal board states.
- Training the NN – after selecting the dimensions of the game, the type and architecture of the neural network needed to be selected. Inspired from the work on AlphaGo, we decided to use a CNN. The specific architecture and implementation will be discussed in the following section.

2.3. Policy CNN

The policy network is a SL policy CNN. It is built and trained in python using Tensorflow (an open source software library for machine learning developed by Google).
The input training data is about 300,000 states (all possible legal board positions for the 4X3 connect 3, extracted by a script that traverses the entire search tree).
Each state (*s)* is tagged by a 5 –dimension *one hot vector* (y) that denotes the optimal action to be taken from (*s)* (for instance – if the best action to be taken is the second column of the board game, then y=(0,1,0,0,0). The tag (0,0,0,0,1) is for terminal states, where no action shall be taken).
As has been done by AlphaGo, we pass the board position as a 4X3 image and use convolutional layers to construct a representation of the position.
The CNN has 2 hidden layers and has the following architecture (this is Tensorflow syntax for building the network graph):

```
# Store layers weight & bias
weights = {
    # 3x4 conv, 1 input, 32 outputs
    'wc1': tf.Variable(tf.random_normal([3, 4, 1, 32]), ),
    # 3x4 conv, 32 inputs, 64 outputs
    'wc2': tf.Variable(tf.random_normal([3, 4, 32, 64])),
    # fully connected, 3*4*64 inputs, 1024 outputs
    'wd1': tf.Variable(tf.random_normal([3*4*64, 1024])),
    # 1024 inputs, 5 outputs (class prediction)
    'out': tf.Variable(tf.random_normal([1024, n_classes]))
}

biases = {
    'bc1': tf.Variable(tf.random_normal([32])),
    'bc2': tf.Variable(tf.random_normal([64])),
    'bd1': tf.Variable(tf.random_normal([1024])),
    'out': tf.Variable(tf.random_normal([n_classes]))
}
```

We trained the network using the Adam (Adaptive Moment Estimation) optimizer to reduce the cross entropy metric, and used the softmax function after the last layer.
The network was then trained to predict moves with an accuracy of 90% (to account for the more interesting cases where the search tree cannot be fully expanded (as in the case of Go and the policy networks used by AlphaGo)).

2.4. Mini AlphaGo's policy

The policy used by mAlphaGo to select actions is:

$$a_t = argmax_a \left(Q(s,a) + c_1 * \sqrt{\frac{\ln\left(N_{parents}(s,a)\right)}{N(s,a)}} + c_2 * \frac{1(s,a,p_m(s|a))}{1+N(s,a)}, \right)$$

- $Q(s,a)$is the action value (the exploitation part – similar to the UCT version of MCTS).
- $c_1 * \sqrt{\frac{\ln\left(N_{parents}(s,a)\right)}{N(s,a)}}$ is the first exploration term. $N_{parents}(s,a)$ is the visit count of the parents of node *d*. c_1 – the first exploration parameter, is chosen empirically (the issue will be addressed in the following section).
- $c_2 * \frac{1(s,a,p_m(s|a))}{1+N(s,a)}$ is the second exploration parameter. $1(s,a,p_m(s|a))$ is an indicator function that equals 1 if the action (a) from state (s) was taken by the policy CNN (hence has the highest probability of winning the game). c_2 – the second exploration parameter, is also chosen empirically (the issue will be addressed in the following section).

As shown above, mAlphaGo does the same exploitation as the UCT version of MCTS, but explores the game tree differently (better, as will be shown in the following section)

2.5. Implementation

MalphaGo was implemented in python. The chosen game to be played – Connect Four – was modeled as a tree search problem. Several key components of the implementation will be discussed:

- In order to traverse the game search tree and maintain and update several numerical values necessary for both MCTS and the CNN, a class 'Node' was created. This is the initialization function of each new node in the search tree:

```
def __init__(self, parent, action, state, player, game=None):
    # Game
    self.game = game or parent.game
    # Structure
    self.parent = parent
    self.children = dict.fromkeys(self.game.actions(state))
    # Tree data
    self.action = action
    self.state = state
    # Search meta data
    self.player = player
    self.visits = 0
     self.value = 0.0
```

- As can be seen in the code above, an additional class - 'Game' - is needed, both for specifying node data (allowed actions, state structure, etc.) and for game simulations to evaluate performance (discussed in the following chapter). Class 'Game' was built as a base class for multi-player adversarial games:

```
class Game(object):
    """
    Base class for multi-player adversarial games.
    """

    def actions(self, state):
    def result(self, state, action, player):
    def terminal(self, state):
    def next_player(self, player):
    def outcome(self, state, player):
```

Then, the class 'Connect Four' was written on top of 'Game'. The following constants were defined for the particular game of 4X3 'Connect Three':

```
PLAYERS = (1, 2)
HEIGHT = 3
WIDTH = 4

TARGET = 3

VALUE_WIN = 1 #positive reward for winning
VALUE_LOSE = -1 #negative reward for losing
VALUE_DRAW = 0 #no reward for draw0
```

- The search algorithm of MAlphaGo was implemented as follows:

```
def mcts_mAlphaGo(game,state,player,budget,CNN_action):
    """
        Implementation of the mAlphaGo's search algorithm
        """

    root = Node(None, None, state, player, game)
    while budget:
        budget -= 1
        # Tree Policy
        child = root
        while not child.terminal():
            if not child.fully_expanded():
                child = child.expand()
                break
            else:
                child = child.best_child()
        # Default Policy
        delta = child.simulation(player)

        # Backup
        while not child is None:
            child.visits += 1
            child.value += delta
            child = child.parent

    return root.best_action_by_mAlphaGo(CNN_action=CNN_action,c=1)
```

3. Performance evaluation and results

The performance of mAlphaGo was evaluated by two different metrics:

- Metric I: game simulations with different budgets and exploration parameters - the outcome of 1000 game simulations of 'connect 3' played by mAlphaGo against a MCTS (UCT version) player.
- Metric II: Monte-Carlo convergence plots of the root's weight value.

3.1. Game simulations

The following results were obtained by playing 1000 head to head games between mAlphaGo and a MCTS player, given identical conditions (each player started 500 times and both were given the same budget (number of iterations, as was previously explained) and the same exploration parameters).

Using this metric, two aspects of mAlphaGo have been studied:

1. The effect of changing the budget (of both mAlphaGo and the MCTS player).

Budget(Iterations)	mAlphaGo Wins (%)	MCTS wins (%)	Tie (%)
25	68.7	26.4	4.9
50	69.8	24.6	5.6
100	69.5	22.5	8.0
300	72.2	21.4	6.4
500	77.5	16.5	6.0
1000	80.8	12.4	6.8
3000	81.4	12.1	6.5
5000	82.3	11.6	6.1

Table 1: head to head games between mAlphaGo and the MCTS player with different budgets

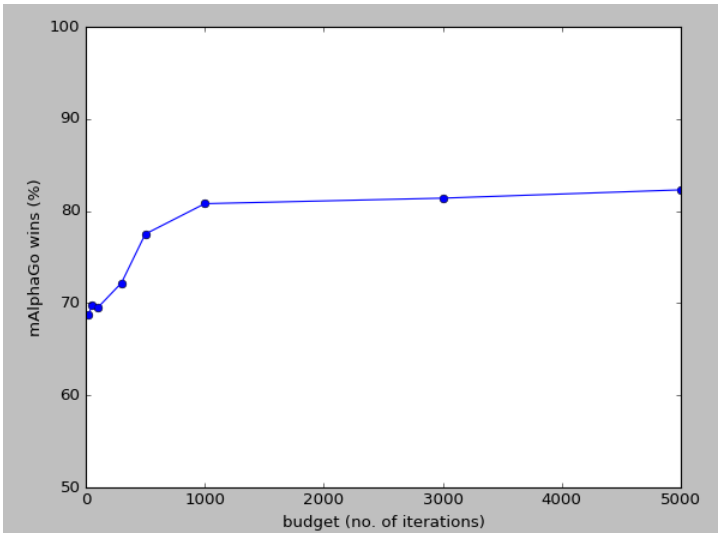

Figure 3: mAlphaGo's wins with different budgets ($c_1 = c_2 = \sqrt{2}$)

As can be seen from the graph, more iterations lead to better percentage of winning by mAlphaGo. As the budget increases, the winning rate converges to low 80s.

2. The effect of changing the exploration parameters

Budget(Iterations)	c_1	c_2	mAlphaGo Wins (%)	MCTS wins (%)	Tie (%)
100	0	$\sqrt{2}$	94.6	4.9	0.5
100	$\sqrt{2}$	$\sqrt{2}$	69.5	22.5	8.0
100	$4\sqrt{2}$	$\sqrt{2}$	64.1	26.3	9.6
100	$10\sqrt{2}$	$\sqrt{2}$	66.5	23.5	10.0
100	$50\sqrt{2}$	$\sqrt{2}$	64.9	29.3	5.8

Table 2: head to head games between mAlphaGo and the MCTS player with different values of c1

Budget(Iterations)	c_1	c_2	mAlphaGo Wins (%)	MCTS wins (%)	Tie (%)
100	0	1	94.0	5.1	0.9
100	0	$\sqrt{2}$	94.6	4.9	0.5
100	0	$4\sqrt{2}$	93.3	5.6	1.1
100	0	$10\sqrt{2}$	92.8	5.9	1.3
100	0	$50\sqrt{2}$	92.8	4.5	2.7

Table 2: head to head games between mAlphaGo and the MCTS player with different values of c2

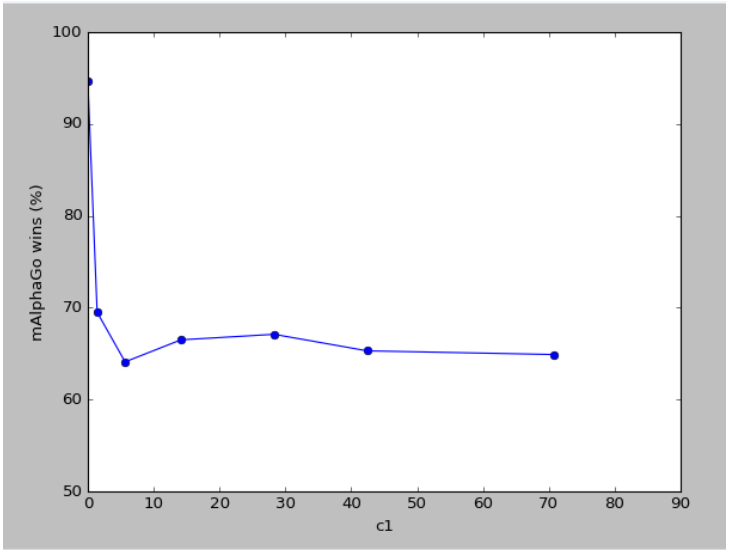

Figure 4: mAlphaGo's wins with different values of c1

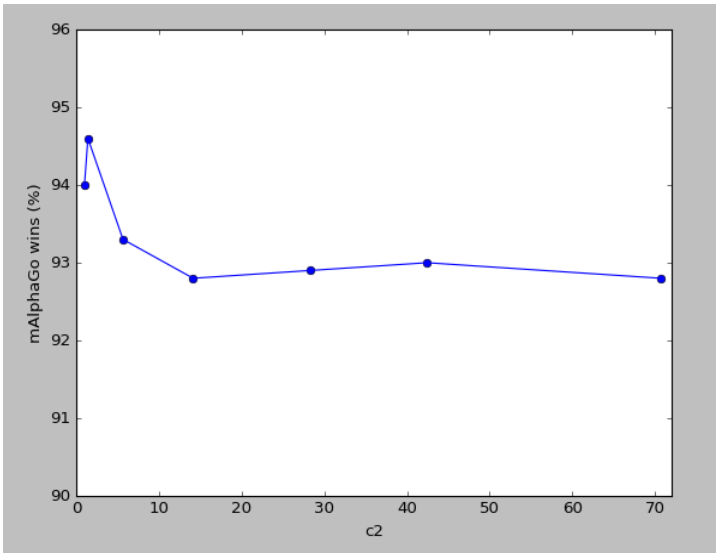

Figure 5: mAlphaGo's wins with different values of c2

- Figure 4 suggests that the CNN ($\frac{1(s,a,p_m(s|a))}{1+N(s,a)}$ – the second exploration term) explores the game search tree better than the first exploration term ($\sqrt{\frac{\ln(N_{parents}(s,a))}{N(s,a)}}$) originated from the UCT formula .
- Figure 5 states that empirically, $\sqrt{2}$ is the preferred value for c_2.

Thus, in the final version of mAlphaGo, the exploration parameters were chosen to be $c_1 = 0, c_2 = \sqrt{2}$.

3.2. Monte-Carlo convergence plots

As was previously mentioned, one of the main problems of the standard MCTS is its slow convergence toward the optimal play. The following graphs show the convergence rate of the weight of the root node (the weight is the node's value divided by its visit count). As can be shown in the graphs, mAlphaGo converges to the optimal weight value faster than the standard MCTS.

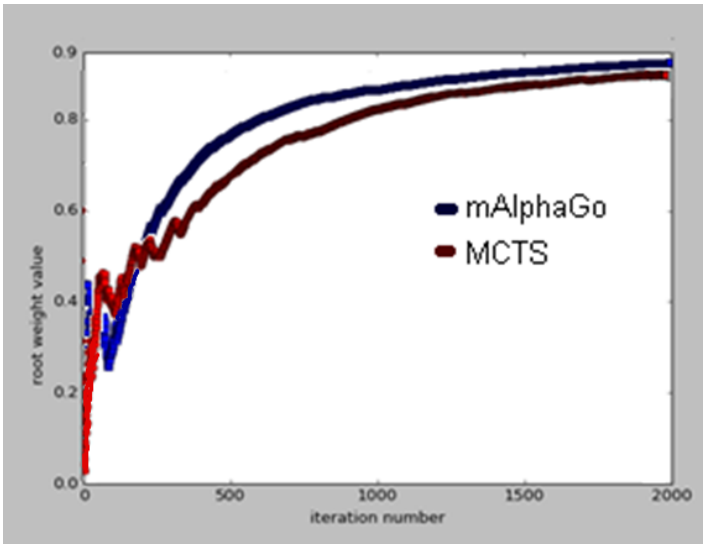

Figure 6: Monte-Carlo convergence plots of mAlphaGo and standard MCTS player ($c_1 = c_2 = \sqrt{2}$)

4. Summary and conclusions

In this work we have developed a computer program – mini AlphaGo – that plays a version of the game "Connect Four" based on a search algorithm that combines the Monte-Carlo tree search algorithm and a policy convolutional neural network.
Evaluation of mAlphaGo's performance against a standard MCTS player points to a significant advantage in favor of mAlphaGo. According to both metrics of evaluation, mAlphaGo converges faster than standard MCTS and performs better in several different conditions (different budgets which are equivalent to different time period allowed for each player's move, and different exploration parameters).
After experimenting with several parameters, the final policy used by mAlphaGo to select actions was chosen to be:

$$a_t = argmax_a \left(Q(s,a) + \sqrt{2} * \frac{1\big(s,a,p_m(s|a)\big)}{1+N(s,a)} \right)$$