CSE 326 Analysis and Design of Algorithms

Dr.Walid Gomaa

Name	**ID**
Mohamed Abdelmonem Makram	120220055
Abdelrahman Ahmed Shaheen	120220228
Abdelrhman Mohamed Eldenary	120220253
Anas Ihab Badr	120220360

E-JUST

Computer Science Engineering Department
Egypt-Japan University of Science and Technology

Contents

1 Introduction

This week we aimed to understand how the popular Alpha Go Deep Learning Model works, and try to implement a simplified version that utilizes a **Policy Function**, **Value Function**, and a **Monte Carlo Tree Search** (MCTS) algorithm. We tried to lean Reinforcement learning in 1 day which was not an ideal choice. We managed to get a working demo. We illustrate the thought process, the implementation, and the results in this report.

2 Our Thought Process

2.1 Alpha Go

Alpha Go is a computer program developed by DeepMind Technologies, a subsidiary of Alphabet Inc. It plays the board game Go and was the first program to defeat a professional human player, the reigning world champion, and several other top players without handicaps on a full-sized board. The program uses deep neural networks and reinforcement learning to improve its performance.

2.2 Adapting Alpha Go Architecture

In our coloring problem, we noticed that the problem actually had a massive state recursive tree (Link), and brute forcing while providing a correct solution for $S(4)$, could not be scaled to $S(5)$ or $S(6)$ due to memory constraints (Maybe if we had more memory...). So we thought "What also has a massive state tree?" and it was the GO game which has approximately 2.1×10^{170} legal board positions.
So if it uses RL and MCTS to accurately traverse the states tree, so could we. One challenge was how we would assign the rewards or what to use when estimating the value of a state.

3 Implementation

We adopted a simplified version of the AlphaGo architecture to solve the Schur Number coloring problem. Our architecture consists of three main components: a **Policy-Value Network**, a **Monte Carlo Tree Search (MCTS)** module, and a **training loop** that mimics self-play reinforcement learning.

3.1 Policy-Value Network

Our neural network takes as input the current coloring state represented as four binary bitmasks, one for each color class (Red, Blue, Green, Cyan), encoded into a 4×64 tensor. The network has the following architecture:

- **2 Convolutional Layers:** Each with 64 filters of kernel size 3, followed by Batch Normalization and ReLU activation. These layers capture local spatial features within each bitmask.
- **Flattening:** The output is flattened into a vector of size 64×64.
- **Fully Connected Layer:** A dense layer with 256 units and ReLU activation to create a shared latent representation.
- **Output Heads:**
  - **Policy Head:** Outputs a probability distribution over 4 actions (corresponding to the 4 colors), using a Softmax layer.
  - **Value Head:** Predicts a scalar value estimating the expected outcome from the current state, using a Tanh activation to bound the output between -1 and 1.

3.2 Monte Carlo Tree Search (MCTS)

We implemented a basic version of MCTS to guide action selection during self-play:

- **Selection:** Traverse the tree using the PUCT formula to balance exploration and exploitation.
- **Expansion:** For a selected node, we expand valid children using the policy output from the neural network.
- **Simulation:** A random rollout is performed to the end of the episode using valid actions.
- **Backpropagation:** The simulation result is backpropagated to update visit counts and node values.

The search outputs a policy distribution over actions, based on visit counts at the root.

3.3 Training Loop

We simulate self-play episodes using MCTS-guided decisions. For each episode:

- We begin from a hardcoded initial state (covering some known valid subsets).
- At each time step, we perform MCTS with the current state and collect the state, policy, and outcome.
- When the episode ends (either by reaching a maximum element or by failure), we assign a reward of $+1$ for success and -1 for failure.

These collected triplets (s, π, v) are used to train the network using a combination of cross-entropy loss for the policy head and mean squared error for the value head.

3.4 Key Hyperparameters

- MCTS Simulations per Move: 100–500
- Max Element (z): up to 30
- Training Epochs: 10
- Optimizer: Adam, with a learning rate of 0.001
- Batch Size: 32