Final Project - Reinforcement Learning

Minhah Saleem - 2023120344

Summary of algorithm:

The overview of the reinforcement learning algorithm proposed in the paper "Solving the Rubik's Cube Without Human Knowledge" (S McAleer, et al., 2018) is as follows:

Autodidactic Iteration (ADI):

ADI is an iterative supervised learning procedure that trains a deep neural network, denoted as fθ(s), with parameters θ, to output value and policy pairs (v, p) given the input state s. After network training, the policy is used to reduce breadth, and the value is utilized to decrease depth in the Monte Carlo Tree Search (MCTS). In each ADI iteration, training samples are generated, and through this, new neural network parameters are obtained. Training samples start from a solved cube and are generated by mixing it k times, with each sample using the result of the corresponding depth-1 Breadth-First Search (BFS) as the training target.

Training Sample Generation:

Starting from a solved cube and mixing it k times, this process is repeated l times to generate N = k * l training samples. Each sample is associated with the number of shuffles D(xi). For each training sample xi, perform depth-1 BFS to obtain the set of all child states, then evaluate the optimal value and policy for each child using the current neural network.

ADI Algorithm:

The ADI algorithm is an iterative process where training samples X are initially obtained, and the network is trained for each sample. RMSProp optimizer, mean squared error loss (value), and softmax cross-entropy loss (policy) are used for training. Specifically, depth-1 BFS is employed for training, and loss weights are used to address convergence issues by assigning higher weights to samples closer to the solved cube.

Solver:

The solver utilizes the trained neural network fθ to implement an asynchronous extension of the Monte Carlo Tree Search (MCTS). It iteratively builds a tree T from the initial state s0, performing simulations. Each node s in the tree has memory to store statistics and information for actions in a specific state. Simulation continues until a maximum computation time is reached or a specific state is solved.

Upon completion of the simulation, the tree T is extracted, and Breadth-First Search (BFS) is performed to find the shortest predicted path from the initial state to the solution. The optimal path can be obtained using edges with a weight of 1 in the tree T.

Summary of the implementation of the algorithm in code:

Reference: https://github.com/azaharyan/DeepCube/tree/main
This code implements a reinforcement learning model to solve the Rubik's Cube problem. The code can be broadly divided into the following parts:

CubeEnv Class: This class represents the Rubik's Cube environment, managing the state of the cube and handling various actions. The get_one_hot_state() function returns the cube's state in One-Hot encoding, and the get_direct_children_if_not_solved() function obtains all possible next steps from the current state.

MCTS (Monte Carlo Tree Search) Class: This class implements the MCTS algorithm. The train() function explores and expands the tree, while the backpropagate() function backpropagates rewards. It performs reinforcement learning through MCTS.

Model Creation and Training: The buildModel() and compile_model() functions create and compile a Keras model. The adi() function trains the model for a given number of iterations. Training data includes information about various states of the cube and the corresponding possible actions.

Other Files: The remaining files mainly provide auxiliary functions necessary for running the environment, model, and training process.

This implementation involves converting the cube's state into One-Hot encoding, performing reinforcement learning using MCTS, and training the model from training data. It uses MCTS to solve the cube, finding the optimal actions as it progresses.

Results in the paper:

Model Architecture and Training:

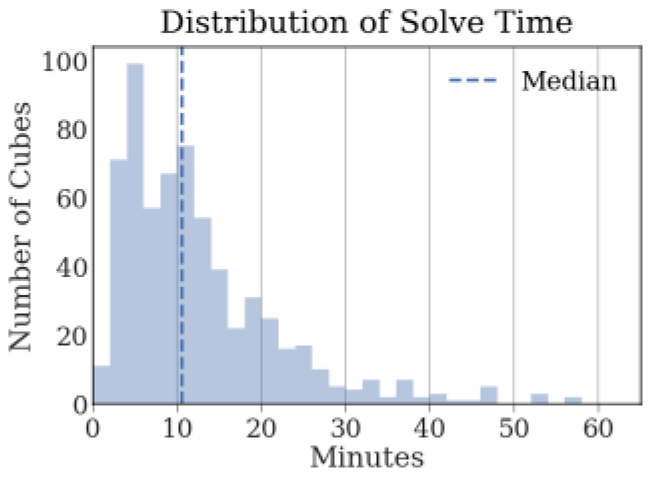
- The neural network used has a feed-forward structure, with outputs comprising a 1-dimensional scalar (v) representing the value and a 12-dimensional vector (p) representing the probability for each possible move.
- The model was trained using the Async Deep Iterative (ADI) algorithm for 2,000,000 iterations.
- Approximately 8 billion cubes were trained over 44 hours, utilizing a training server with a 32-core Intel Xeon E5-2620 and three NVIDIA Titan XP GPUs.

Benchmarks and Comparisons:

- Benchmarks were performed by comparing against the Kociemba two-stage solver and Korf Iterative Deepening A* (IDA*) as benchmarks.
- DeepCube was compared with self-modifications like Naive DeepCube and Greedy.

Performance Comparison Results:

- When applying DeepCube and Kociemba to 640 randomly shuffled cubes, both algorithms solved all cubes within an hour.
- While Kociemba solved each cube within 1 second, DeepCube had an intermediate solving time of 10 minutes.
- DeepCube outperformed Kociemba in 55% of cases and effectively solved the problem even with a larger variance in solution lengths.



(b) DeepCube's Distribution of solve times for the 640 fully scrambled cubes. All cubes were solved within the 60 minute limit.

Optimal Path Finding and Performance Evaluation:

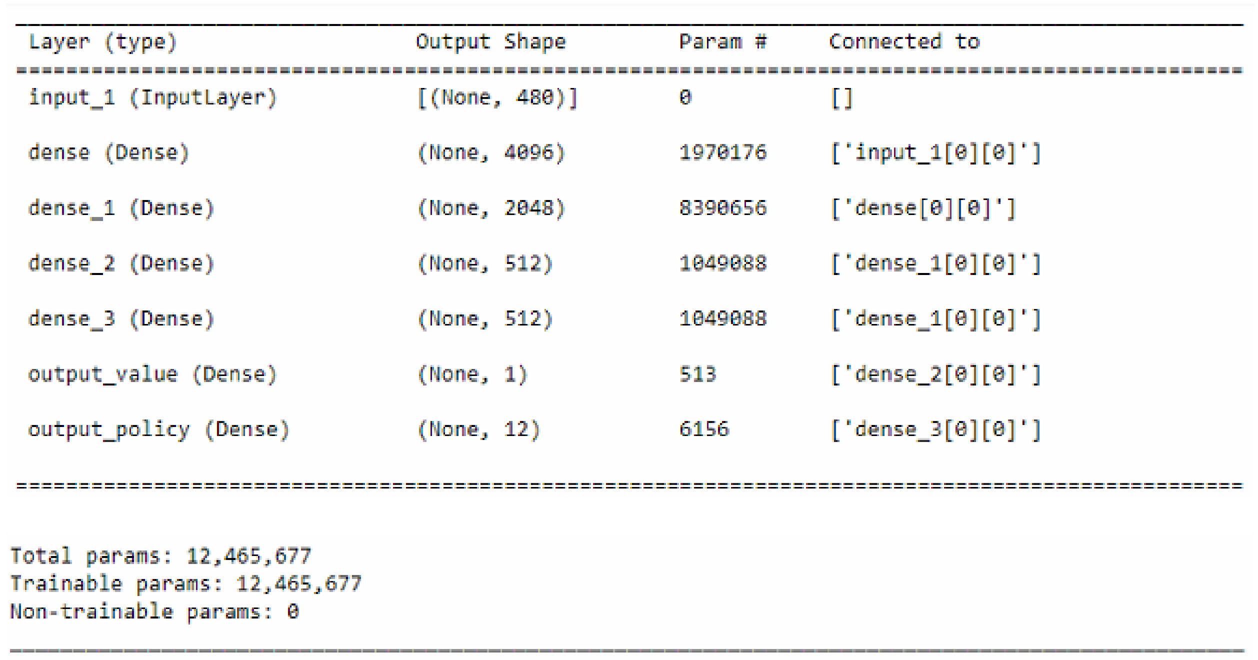
- DeepCube's performance was consistently high, especially for nearby cubes, when compared with the Korf optimal solver.
- DeepCube exhibited a median solving length nearly identical to Korf, with Korf matching the optimal path found by DeepCube in 74% of cases.

Algorithm Performance and Efficiency:

- DeepCube's MCTS algorithm demonstrated effective problem-solving while exploring fewer tree nodes compared to heuristic-based search methods.
- The DeepCube MCTS averaged 1,136 nodes explored, significantly less than the Korf optimal solver, which required an average of 1,220 billion nodes for fully scrambled cubes.

Results achieved through implementation:

Model:



```
______________________________________________________________________________________________
 Layer (type)                    Output Shape         Param #     Connected to
==============================================================================================
 input_1 (InputLayer)            [(None, 480)]        0           []

 dense (Dense)                   (None, 4096)         1970176     ['input_1[0][0]']

 dense_1 (Dense)                 (None, 2048)         8390656     ['dense[0][0]']

 dense_2 (Dense)                 (None, 512)          1049088     ['dense_1[0][0]']

 dense_3 (Dense)                 (None, 512)          1049088     ['dense_1[0][0]']

 output_value (Dense)            (None, 1)            513         ['dense_2[0][0]']

 output_policy (Dense)           (None, 12)           6156        ['dense_3[0][0]']

==============================================================================================

Total params: 12,465,677
Trainable params: 12,465,677
Non-trainable params: 0
______________________________________________________________________________________________
```

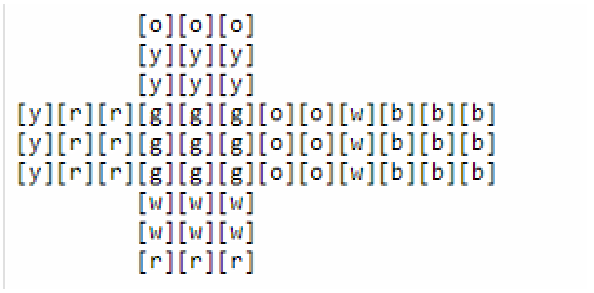
Training:

In this experiment, the model was trained using an iterative approach over a fixed number of iterations. Each iteration involved the generation of N scrambled cubes, where N was set to 100. For each cube, 15 batches of actions were performed, forming the training dataset. The

neural network model was updated using a custom training algorithm, and the training process was repeated for a total of 50 iterations. This iterative training approach allowed the model to learn and adapt to the complex Rubik's Cube-solving strategy. The convergence and performance of the model were assessed over the course of these 50 iterations, ensuring that the model achieved stability and effectiveness in solving the Rubik's Cube.
Training was done using NVIDIA A100-PCIE-40GB and took a total time of 16 hours 22 minutes and 06 seconds.

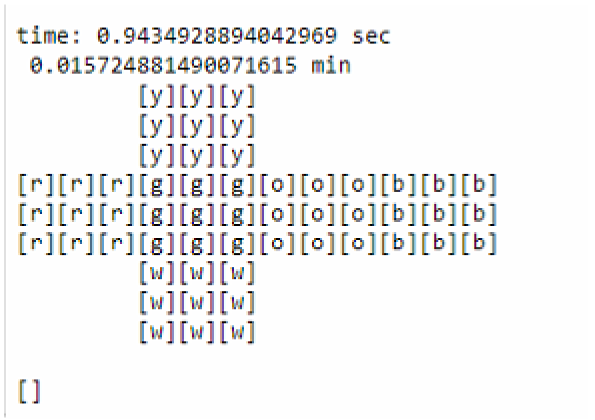
Results:

A randomly scrambled cube with number of turns = 1 is rendered as:



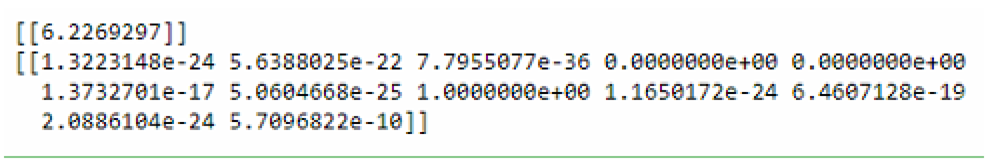
```
         [o][o][o]
         [y][y][y]
         [y][y][y]
[y][r][r][g][g][g][o][o][w][b][b][b]
[y][r][r][g][g][g][o][o][w][b][b][b]
[y][r][r][g][g][g][o][o][w][b][b][b]
         [w][w][w]
         [w][w][w]
         [r][r][r]
```

Printed actions and time to solve in both seconds and minutes for this cube:



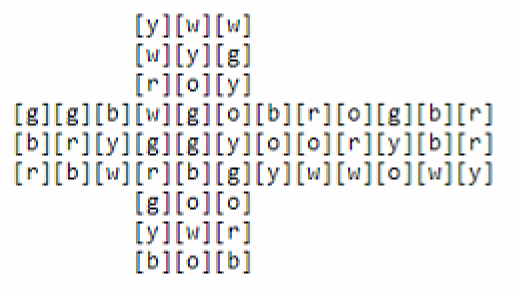
```
time: 0.9434928894042969 sec
 0.015724881490071615 min
         [y][y][y]
         [y][y][y]
         [y][y][y]
[r][r][r][g][g][g][o][o][o][b][b][b]
[r][r][r][g][g][g][o][o][o][b][b][b]
[r][r][r][g][g][g][o][o][o][b][b][b]
         [w][w][w]
         [w][w][w]
         [w][w][w]

[]
```

The value and policy are also displayed respectively:



```
[[6.2269297]]
[[1.3223148e-24 5.6388025e-22 7.7955077e-36 0.0000000e+00 0.0000000e+00
  1.3732701e-17 5.0604668e-25 1.0000000e+00 1.1650172e-24 6.4607128e-19
  2.0886104e-24 5.7096822e-10]]
```

For a scrambled cube with 100 no. of turns, the rendered cube can be seen as:



```
         [y][w][w]
         [w][y][g]
         [r][o][y]
[g][g][b][w][g][o][b][r][o][g][b][r]
[b][r][y][g][g][y][o][o][r][y][b][r]
[r][b][w][r][b][g][y][w][w][o][w][y]
         [g][o][o]
         [y][w][r]
         [b][o][b]
```

The model took more than 3 hours to solve the rubik's cube.

Note: In the given time, the validation of results i.e. median time of solving fully scrambled cubes couldnot be completed. According to paper the median time is 10 min which implies 10*640 = 6400 mins = 106.67 hrs = 4.4 days. It can be seen that for fully scrambled cubes it takes longer. Considering the max time reported in paper 60 mins implies 60*640 = 38400 mins = 640 hrs =26.67 days. The inference for 640 fully scrambled cubes will take anywhere between 5 to 27 days to complete.
In practice, it was found that for a fully scrambled cube (no. of turns = 100), it took more than 3 hours and thus the inference was interrupted without completion.
Therefore, the reported results are for only 1 cube i.e. least scrambled (no. of turns=1).

Areas for improvement or limitations and ideas for addressing them:

Algorithm Limitations and Improvement Ideas:

Model Performance and Generalization:

Limitation: The current trained model relies on specific cube states, limiting its ability to generalize to the broader Rubik's Cube problem.
Improvement Idea: Introducing techniques such as transfer learning or domain adaptation to enhance the generalization ability to diverse cube initial states.

Efficiency of MCTS Algorithm Exploration:

Limitation: While MCTS is effective in large search spaces, it struggles to find optimal actions due to the unique nature of the Rubik's Cube problem.
Improvement Idea: Enhancing exploration efficiency by utilizing domain-specific MCTS variations or incorporating additional heuristic information.

Diverse Cube States and Action Scenarios:

Limitation: The current learning is limited to a constrained set of initial states and action scenarios, lacking diversity.

Improvement Idea: Generating more diverse and complex cube initial states and action scenarios, enabling the model to adapt to a wider range of situations.

Lack of Dynamic Learning Strategies:

Limitation: The current training strategy relies on a fixed number of iterations, lacking dynamic adaptability.
Improvement Idea: Adjusting the learning rate based on the model's performance during training or introducing reinforcement learning strategies to enhance effective learning.

Limitations of Environment Modeling:

Limitation: The current cube environment modeling only considers the present state, potentially overlooking temporal dynamics or various cube manipulation techniques.
Improvement Idea: Introducing environment modeling considering temporal dynamics or techniques for modeling various cube manipulations, creating a more flexible environment model.

Exploration-Exploitation Trade-off:

Limitation: Finding the right balance between exploration and exploitation in MCTS can be challenging.
Improvement Idea: Introducing various UCB (Upper Confidence Bound) variations or parameters adjusting the exploration-exploitation trade-off to find a more effective exploration strategy.
By incorporating these improvement ideas, the algorithm's performance and effectiveness in solving the Rubik's Cube problem can be enhanced.

Conclusion:

This project began with a careful analysis of the details of the "Solving the Rubik’s Cube Without Human Knowledge" paper (S McAleer et al., 2018) and providing an overview and summary of the ADI algorithm.

The next steps of the project involved the actual implementation of the algorithm. The implementation specifically focused on replicating the results presented in the original paper, utilizing open resources such as GitHub.

While the implementation results were documented in the report, the high inference time prevented the confirmation of the paper's results. Although, it was confirmed that for our system specification, a fully scrambled cube CAN take more than 60 mins to be solved, negating the paper’s claims.

Additionally, the report identifies improvements and limitations of the presented reinforcement learning algorithm. This critical analysis lays the groundwork for future work, encouraging consideration of potential enhancements and new approaches to overcome identified limitations. Recognizing areas in need of improvement underscores dedication to the advancement of the field of reinforcement learning.