Solving Rubik's Cube with Deep Reinforcement Learning and Search

Mohammadreza Moravej

Abstract

The Rubrik's cube is an interesting puzzle which due to large number of states and a single goal state is challenging to solve using RL techniques. DeepCubeA is a state of art algorithm which combines deep RL and search to solve the Rubik's cube. Though the algorithm is fairly successful at solving the puzzle, it is very computationally expensive to solve and cannot be applied to problems with even larger number of states. In this work we introduce Multi-goal DeepCubeA, a modified version of DeepCubeA which is less computationally expensive and is (hopefully) applicable to problems with larger state spaces.

1 Introduction

There has recently been a lot of interest in solving combinatorial games using reinforcement learning. One such problem is the Rubik's cube, where the goal is to rotate the joints of the cube so that all the stickers on each face have the same color. Clearly, one cannot hope to solve the cube by randomly scrambling it. The can be said when trying to solve the cube with a naive Rl algorithm. There are 4.3×10^19 different configurations of the cube (which would be the states in the RL setting) and only one single goal. Hence, the reward function is $r(s, s') = 0$ for any $s, s' \neq s_{goal}$. A sequence of random moves (rollouts for policy learning or value estimation for value based RL) will not work since the algorithm is very unlikely to end in the goal state.

There have been two recent works that address solving the cube using RL. McAleer et al. [1] trains a deep neural net f(s) which outputs a policy π (probability distribution of the actions) and a value estimate $v(s)$ for a given state. The algorithm then uses a the trained model along with a Monte Carlo Tree Search. The work by Agostinelli et al. [2] trains a feed forward model to estimate the distance of a given state to the goal state. The estimate of the neural network is then used as the heuristic in A* search to get to the goal from any state. The two works follow very similar approaches to train the respective models, which will be explained later . Here, the focus will be on the latter work since our algorithm is built on it.

2 Related Work - DeepCubeA

DeepCubeA, presented by Agostinelli et al. [2] is the algorithm which addresses the challenge imposed by the large state space of the problem. DeepCubeA is an approximate value iteration algorithm which uses a feed forward model $J(s)$ to approximate the value of a state s. The model is trained to minimize the mean squared error between its estimation of the cost-to-go of state s, $J(s)$, and the updated cost-to-go estimation $J^{/}(s)$.

$$J'(s) = min_a(c_a(s, A(s, a)) + J(A(s, a))) = min_a(1 + J(A(s, a)))$$

Here, A(s,a) is the state the agent ends up in after taking action a from state s. $c_a(s, A(s, a))$ is the cost of going from s to A(s,a) which we set to one for any two states. This is just the equation we use to update values in DP, except there is not a probability matrix since the moves are deterministic. There is also no discount factor since the horizon only the next state. The authors claim a one step look ahead gives the same results as a multi-step look ahead while requiring less training time.

In order for the model to be able to learn, it must be trained on a state distribution that allows information to propagate from the goal state to all the other states during training. To achieve this, training data is generated by exploring in reverse from the goal state. The cube is randomly scrambled from the goal state k_i times, where k_i is uniformly distributed between 1 and K. Each time the resulting state s_i is added to the training set. During training, the cost-to-go function first improves for states that close to the goal state. The cost-to-go function then improves for states further away as the reward signal is propagated from the goal state to other states.

A more through description of the training procedure is outlined below:

Algorithm 1 *DeepCubeA Training*

```
Inputs: B: Batch size, M: Training iterations, K: Max number of scrambles
l ← $nn - parameters$
for $m$ from 1 to $M$ do
    S = get-scrambled-states(B,K)
    for $s_i \in S$ do
        $y_i = min_a(c_a(s, A(s,a)) + J_l(A(s,a)))$
    end
    end
    l, loss ← $train(J_\theta, S, Y)$
```

Finally, once the model is trained,it is used as a heuristic to do A* search. The cost of each state s is g(s) + h(s), where g(s) is the path cost (distance between goal and s) and h(s) is the heuristic which is estimated by the feed-forward network parameterized by θ.

3 The Problem with DeepCubeA

Below we will describe three important downsides of DeepCubeA. The ifirst two issues are general problems that arise in similar problem settings. We will then discuss the mislabeled data problem, which describes the fundamental issue with algorithm 1 above.

3.1 Computational Issue

Training the model is very expensive; the authors mention that the network was train on 10 billion examples. Also, training was carried out for 1 million iterations on two NVIDIA Titan V GPUs, with six other GPUs used in parallel for data generation. It too 36 hours for the model to be trained. The huge number of iterations is due to the fact that it takes a lot of time for the reward function to propagate back from the goal state to other states. At each iteration, we have to compute $J'(s)$ for all of 10 billion examples. Consider that there are 12 possible actions that can be applied to every state, computing $J'(s)$ is very expensive. Clearly, computing $J'(s)$ for states that are further away is harder.

3.2 Non-linear Approximate Value Function

DeepCubeA trains on states that are generated by starting from the solved cube and taking random actions, allowing the reward signal to propagate from the solved state to states that are farther away. However, ensuring the reward signal is propagated is not the only concern for the Rubik's Cube. Training a nonlinear approximate value function often leads to unstable or divergent behavior. Although the optimal value function is usually smooth and easy to approximate, the intermediate value functions generated by the network are rarely smooth, resulting in divergent behaviour.

3.3 Mislabeled Training Data

Let s_i be a random state which we have picked to train the model on. In Algorithm 1, the pair (s_i, y_i) is inputted into the neural network M times. Majority of these M times, the label y_i is incorrect. This is due to the fact that y_i is itself dependent on J_θ which is the output of an untrained model. Hence, the model will be trained on mislabeled data for a long period of time for at least the first thousands of iterations. Hence, why DeepCubeA took 1 million iterations to train. Experimentally, we trained the model for 500 iterations and all the model predicted was values very close to 1 for the majority of the states.

4 Formal Description of the Algorithm: Multi-Objective DeepCubeA

All three problems above can be partially handled by reducing the distance between training examples and the goal. If such distance is reduced:

- We would required less number of iterations to propagate the reward.
- The algorithm trains on data that it is more confident of the true value since the training states are close to the goal.
- Training will be less computationally expensive. The further away s_i is from the goal, the longer it takes for the reward function to propagate to it, and the longer the model will take to learn the correct label for s_i.

However, limiting the training set in such a way will result in the algorithm not learning the state space properly and not being able to solve every configuration. In this work, we present Multi-goal DeepCubeA which aims to resolve this issue. Multi-goal DeepCubeA is more computationally efficient and can potentially be used to solve problems with even larger state spaces.

4.1 Training

The general idea is to train multiple models to learn to approximate the value state with respect to multiple configurations instead of having a single model learn the heuristic for the final goal state. More formally, let f^i be the model that learns to approximate a value function for an intermediate goal g^i. We start by training the DeepCubeA model to learn to approximate the value function for the goal state. However, we only train the model on states that are closer to the final goal. We call this training set T_k^goal, where k is the max number of scrambles we perform to generate each training example (notice k is small here compare to the original DeepCubeA algorithm). Reducing the Training set in such a way will result in the reward signal being propagated faster to the states nearby. However, the model is not able to estimate a value for every state as it has only been trained on the states that were close to the goal. To circumvent this issue, we pick n training states s^1, s^2, ... s^n from T_k^goal and use f^goal to find a path to get from each s^i to the goal. We save each path and the distance from s_i to the goal for later use. We also initialize n models f^1, f^2, ..., f^n. Each model f^i is trained on a small data set T_k^i to learn to estimate the value function to get to s^i. The purpose of doing this is to make each s^i and intermediate goal. We can continue doing the same procedure m times until we create a tree of intermediate goals (there will be n children for each intermediate goal and the tree will have height m).

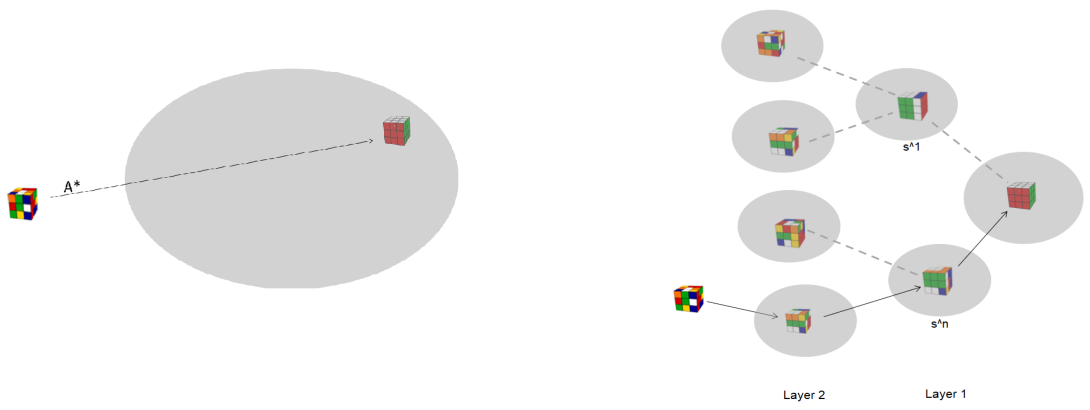
Algorithm 2 Multi-objective DeepCubeA Training

```
Inputs: Goal state, n
Train DeepCubeA on the final goal on states close to the goal
for $i$ from 1 to $n$ do
    s$^i \leftarrow scramblethecubem\ U(t1, t2)times$
    $trainf^i ons^i$
    $path^i \leftarrow pathfroms^i togoal$
    $d^i \leftarrow distance(s^i, goal)$
    end
```

To construct a tree of intermediate goals, we run the above block n^m times (where m is the height and n is the number of children in the tree). At each time the goal state is set to some s^i.

DeepCubeA was trained on 10 billion (10^{10}) examples. Since there are m models, we can train each model on $10^{10}/n^m$ examples. Setting m = 3 and n = 5, this will make $8 * 10^7$ examples per neural net and 125 models in total. We should decrease the number of parameters for each model to make it not overfit the data. We also reduce k (number of scrambles to generate a training configuration) since we want the examples to be near each intermediate goal state. Even though the number of total training examples have been the same, the total amount of computation needed to train the models will decease. This is due to the fact that we will need many less iterations for each model to approximate the value function of the states nearby as the reward propagates back faster.

4.2 Testing



The gray space represents the states the model is trained on. Left: DeepCubeA, a single model learns to estimate the distance to the goal state. Right: Multi-objective DeepCubeA, multiple models are learned, one for each intermediate goal. A tree of intermediate goals is contracted. The path to the goal state is depicted by the back arrows.

Let s be the state we start at. We do an A* search with $h(s) = min_i(\frac{1}{m*d_i} * f_i)$. This simply means that the algorithm will start by solving for the closets intermediate goal. The coefficient $\frac{1}{m*d_i}$ is to encourage picking the intermediate goal that is closer to the final goal. Here m is a positive hyper parameter. The agent then uses $h(s)$ to get to the intermediate goal s^j. From there, we use $path^j$ (which we have already found during training) from s^j to the parent of s^j, etc. We add the paths to get to the final goal.

5 Limitations

Despite its faster training time, Multi-goal DeepCube has the disadvantage of almost always finding the sub-optimal path when it comes to the number of moves. This is due to the fact that the solution found must always be thorough the intermediate goals.

6 Conclusion

DeepCubeA is capable of solving planning problems with large state spaces and few goal states such as the Rubik's cube. The algorithm achieves this by learning a cost-to-go function via a neural network which is used as a heuristic function for A* search. The cost-to-go function is learned by using approximate value iteration on states generated by starting from the goal state and taking moves in reverse. While DeepCubeA has been successful at solving the puzzle with a few moves, it is very expensive to train. It is also practically impossible to solve problems with larger states using DeepCubeA, such as the 4 by 4 Rubik's cube. We suggest that the main cause behind the computational bottleneck is mislabelled data during a large chunk of training. We then propose Multi-objective DeepCubeA, which we assume is much less expensive to train and can be applied to problems with larger state space.

References

[1] Solving the Rubik's Cube with Approximate Policy Iteration

[2] Solving the Rubik's cube with deep reinforcement learning and search.

Code: https://github.com/forestagostinelli/DeepCubeA