Solving the Rubik's Cube Using Attention-Based Neural Networks

Marc Felix Brinner

August 3, 2021

Abstract

The Rubik's Cube is a combinatorial puzzle invented in the late 1970s. It's huge state-space makes it impossible to find the single solved state with uninformed search methods and even the use of simple heuristics to select paths in the search tree in an informed way will lead to solutions that are far from optimal. This study explores the possibility of using neural networks as state evaluation functions to predict the number of moves required to solve the given state, which can then be used to find near-optimal solutions using an informed search algorithm. In contrast to previous studies, the neural network architectures will employ attention mechanisms that allow it to learn more general concepts while using less parameters. This, in combination with a new, more direct, training method ultimately leads to less training time while preserving good solving performance.

1 Introduction

The Rubik's Cube (Figure 1), a combinatorial puzzle invented in the late 1970s, is one of the most successful toys of all times. It's simplicity in design lets even kids grasp the basic concept of turning the individual faces of the cube (Figure 1, right) to reach the desired goal state. This strongly contrasts the inherent complexity of the puzzle, which has so many different states that reaching the solved state with random moves is highly unlikely. More precisely, there are about 4.3×10^{19} possible states that the Rubik's Cube can be in [16], which is why it took more than 30 years to prove that every possible state can be solved in 20 moves if measured in half-turn metric [16], compared to 26 moves if measured in quarter-turn metric [7]. For more details on the difference between half-turn metric (htm) and quarter-turn metric (qtm), see Appendix A.

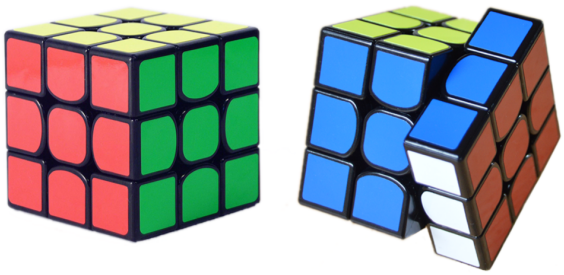

Figure 1: The Rubik's Cube

Over the years, many algorithms were proposed to solve the Rubik's Cube, differing in their computational complexity and their ability to find optimal or near-optimal solutions. One of the algorithms that is used the most is the two-phase algorithm, which solves two sub-problems to find near optimal solutions (at most 29 moves (htm), [5]). Additionally, there are optimal solvers that always find optimal solutions to any given state. One of them is Korf's Algorithm [10], which starts with a sub-optimal solution and iteratively finds more optimal solutions while making use of extensive pruning to speed up the search.

These algorithms were invented more than 20 years ago, and the more recent advances, like finding God's Number (which is the name for the minimum number of moves needed to solve every possible state), were mainly achieved due to having access to more powerful computers and not through inventing new ways to actually solve the cube. In contrast to this stagnation, machine learning models, and most notably neural networks, improved a lot and achieved outstanding results in many areas of research like image recognition [9] and natural language processing [13]. One other area in which neural networks perform very well is playing games, including computer games [19], but also many

board games like Chess [17] and Go [18]. This success generalizes to all tasks that can be described by a Markov Decision Process (MDP), especially if it fulfills some simplifying requirements like having a finite state-space, deterministic state transitions and rewards and a finite action space.
The problem of solving a Rubik's Cube can actually be stated as an MDP that fulfills all of these requirements, which suggests that neural networks are well suited to succeed at this task. The main issues are these of representing the cube in a way that makes it easy for the neural network to find all the information it needs, to select a neural network structure that fits this specific problem and to use a training procedure that trains the neural network in an efficient way and leads to the desired performance. For the first aspect, this study proposes to use a representation that connects color information with position information for each of the 54 colored patches of the Rubik's Cube. Regarding the second and third aspects, this study explores the possibilities of using attention mechanisms as a basis for the neural network structure and to train the resulting networks with a combination of Bellman updates and the use of training sets.

The following section will give an overview over some related work in the area of using algorithms to solve the Rubik's Cube. Section 3 will explain the way the Rubik's Cube is represented for the neural network to process it, describe the actual neural network structures used in this study and discuss the basic learning procedure employed to train the networks. Section 4 then displays the results that the models were able to achieve, compare them to the state-of-the-art results and explain the reason for the differences between the models. Section 5 concludes this study with some final thoughts and remarks.

2 Related Work

After the invention of the Rubik's Cube, a lot of work has been dedicated to finding efficient cube solvers with optimal or close-to-optimal performance as well as to finding lower and lower bounds on God's Number.
One of the first algorithms developed to solve the Rubik's Cube efficiently was Thistlethwaite's algorithm, which is able to always find a solution of length 52 or less measured in htm [11]. Kociemba's two-phase algorithm, developed in 1991, then improved the approach used in Thistlethwaite's algorithm to guarantee a solution of length 30 or less [5], which eventually was lowered to 29 moves in 1995 [6]. Two years later, Korf's algorithm was developed [10], which is the first optimal solver and remains one of the most popular optimal solvers until today. Determining God's Number to be 20 took another 15 years though, since finding it involves solving a large set of states optimally, which can then be shown to generalize to all possible states. This was, for a long time, computationally infeasible though, even though the set of states that need to be solved is significantly smaller than the set of all possible states. This led to a slow progression, in which the upper bound for God's Number was first lowered to 27 in 2007 [15], to 23 in 2008 [3] and finally it was determined to be 20 in 2010 [16] (all measured in htm). For qtm, it took four more years to determine God's Number to be 26 [7].

Several machine learning approaches have been proposed to tackle the problem of solving the Rubik's Cube, most of which rely on neural networks. In [8] though, multiple classical machine learning models were tested as *Learned Guidance Functions*, which basically suggest moves that might be sensible in the current state. Several models were tested, like k-NN, LDA, Linear SVM, RBF SVM and others, with random forests achieving the best results, still falling far behind the results achieved by neural networks.
In [12], the authors used a neural network called DeepCube to predict a value-policy pair for a given state s, with the value being the expected reward for the corresponding MDP and the policy being probabilities for each possible move to be taken in the current state. To solve the cube with these information, a search tree is built where the predicted Q-values are used to eliminate useless branches at a certain depth and the predicted policy is used to restrict the number of branches that the search tree adds. This approach achieves a median solve length of 30 moves in qtm.
The approach proposed in [1] uses a neural network called DeepCubeA, which works similar to Deep-

Cube. Instead of predicting a value-policy pair for a given state, here only the value of the current state is predicted and then used to build the search tree. The main difference to DeepCube lies in the slightly larger number of neural network parameters (about 17% increase) and a deeper structure that uses residual connections. DeepCubeA achieves very good results, being able to find the optimal solution 60.3% of the times and needing only 2-4 moves more than the optimal solution in most other cases. In both studies, reinforcement learning with value iteration was used to train the network.

3 Methods

We develop a new approach to solve the Rubik's Cube, which relies on attention based neural networks in combination with a mixture of classical value iteration and a training set of given state-value-pairs to optimize and speed up the training process. While the general use of the neural network as well as the training procedure stay the same, we will look at two different neural network models that take a completely different approach at processing the information given to them to predict the corresponding state value.

3.1 The Cube Representation

Before looking at the actual neural network structures, we need to take a look at how to represent the Rubik's Cube in a way that can easily be processed by the networks. The Rubik's Cube has six sides with nine colored patches on each side, which results in 54 patches in total. On the other hand, the cube consists of 26 individual building blocks called cubelets, namely 8 corner cubelets with 3 colored patches on each one, 12 edge cubelets with two colored patches and 6 center cubelets with one corresponding colored patch. An algorithm does not need to rotate the cube while solving it, which means that the center cubelets will not move in the process of solving the cube. This makes them theoretically irrelevant since a representation that omits these patches conveys the same information about the current state of the cube. Explicitly including the center cubelets in the representation could lead to a representation that makes it easier for the neural network to extract useful information though, such that we end up with three sensible ways of representing the cube: Representing the locations of the 20 cubelets with their corresponding orientation, only representing the 48 colored patches that can actually be moved or representing all 54 colored patches of the cube. In the cubelet representation, it is not needed to specify the color of each patch individually since, for example, the position and orientation of a single colored patch on a corner fully specify the positions of the two remaining colored patches. In this way, the dimensionality of the representation can be reduced, as was done in [12]. This comes at the cost of making it harder for the neural network to access all information since it needs to learn to deduce the positions of the other patches to correctly evaluate a certain state. Additionally, if the patch representation is used, it is easier for the neural network to learn the concept of equally colored blocks, which is an important concept for predicting the value for a given state. This representation difference might be one of the reasons for the worse performance of DeepCube compared to DeepCubeA since the general approach was quite similar, except DeepCube chose a cubelet representation while DeepCubeA chose a patch representation.

For these reasons, a patch representation was chosen as input for both neural networks that are tested in this study. Further, the extended patch representation, which covers the six immovable patches as well, was chosen such that each patch has a corresponding index, ranging from 0 to 53, and each color has and index as well, ranging from 0 to 5. The final representation is then a list of length 54 that contains at position i the corresponding color value for the patch that is indexed with position i.
This list representation is then converted into a representation that is more suitable to be processed by a neural network. This actually happens in the first layers of the neural networks, which are the same for both networks. The first layer is an embedding layer that maps each color index to a six-dimensional vector that one-hot encodes the given color values, resulting in a cube representation of shape $(54, 6)$. In addition to this, the position of each patch is one-hot encoded, which results in a 54-dimensional

vector for each input patch. This position encoding is appended to the color encoding such that the cube representation now has the shape $(54, 60)$, with only two values being 1 and all other values being 0. This then is the final representation of the cube which can easily be processed by subsequent neural network layers. Explicitly encoding the position in this way is necessary because attention mechanisms are used to process this representation, and these generally discard any information about order in the input.

The main advantage of using this representation in this study is that a representation which clearly divides the final embedding vector into a part that encodes the color of the patch and a part that encodes it's position is especially well applicable for the self-attention mechanisms that will be used in both neural networks. Self attention layers work by using query vectors which are multiplied by each row of the current cube representation using the dot product to get similarities scores. These scores are then used as weights to create a linear combination of these rows from the cube representation such that, for each query vector, we end up with a single response vector that encodes the information we queried for. This cube representation now allows the network, for example, to create a query that returns information about all positions of elements with a specific color, which can be done by using a query that only has a single non-zero element that corresponds to the specific color. The converse works in the same way: The network can chose a query that specifically selects the patches from one side of the cube and receive a vector containing information about all the colors that exist on the specific side.

3.2 The Neural Networks

Similar to the approaches taken in [12] and [1], the approach used in this study relies on a neural network that predicts the corresponding value for any given state. DeepCube was used to predict the actual value function of the corresponding MDP, while DeepCubeA instead acts as a cost function that predicts the expected number of moves needed to reach the goal state, so that the goal state has the lowest predicted value of 0 (see Section 3.3 for more details). In this study, we use the same approach of predicting the cost of each state. To do this, two different architectures are tested, one of which relies on stacking multiple attention blocks to enable the network to query for more and more complex patterns with each new block, while the other one relies on a recurrent architecture that allows the network to query for new information on the basis of previously gathered knowledge.

The main reasoning behind the proposed approaches is the following: We, as humans, do not just look at the cube once and assemble all information we gained by calculating complex interactions between individual patches. Additionally, we do not process every state we see in the same way. Both of these aspects are disregarded when processing the cube representation by using just multiple dense layers, though. To more closely match the process of a human reasoning about the current state of the Rubik's Cube, the proposed method is to use attention mechanisms with learned queries to query for specific patterns on the cube and to process the information gained in this way by subsequent dense layers.
To justify the neural network architecture choices, let us imagine what a skilled Rubik's Cube solver would do if he sees a cube in a scrambled state. On the one hand, he might be looking for small patterns like blocks of the same color or many patches that are colored the same and that are already on the correct side. The individual patterns alone will not be enough to fully reason about the state of the cube, though. Instead, this will require reasoning about the interplay and the consequences of co-occurrence of the patterns that were detected previously, such that we will need to query for more complex patterns and combinations of the patterns we just detected. This process is resembled by the first neural network architecture, the stacked attention network, which will be described in Section 3.2.1.
The second thing a skilled Rubik's Cube solver would do is looking for specific information on the basis of previously gathered information. For example, it might be possible to move a certain block of patches to the correct spot very soon, but doing so will require the solver to not mess up other structures that are already solved. This means, that after gathering information about an interesting

pattern, further investigations on the basis of the current knowledge are needed to be able to fully assess the current state of the cube. This behavior is resembled by the recurrent attention network, which will be described in Section 3.2.2.

3.2.1 The Stacked Attention Network

The first neural network uses stacked attention blocks to be able to query for more and more complex patterns with each new block. One individual attention block is depicted on the left side of Figure 2, the right side displays the complete neural network architecture. The input to each attention block is of shape $(n_{query}, 60)$, with the number of queries being chosen to be 216. The shape of the input to the attention block needs to match the number of queries since residual connections are used to add the output of the attention layer to its input, which is the sole reason for the repeating of the initial cube representation. The 216 queries are used to extract information from the input to the attention block by means of a multi head attention layer using self-attention. In this architecture, the queries are constant and thus do not change for different states that are processed by the network. The assumption is, that 216 "looks" at the cube, performed by using the attention mechanism with 216 queries, are enough to extract useful information from any state the cube can be in (state-dependent queries that are created by means of dense layers did not lead to better results). The output of the attention layer is then fed into an addition to create a residual connection, followed by a subsequent dense layer which is applied to each 60-dimensional vector individually. A final dense layer with linear activation, again applied individually to each vector, and subsequent layer normalization [2] layer conclude the double-attention-block.

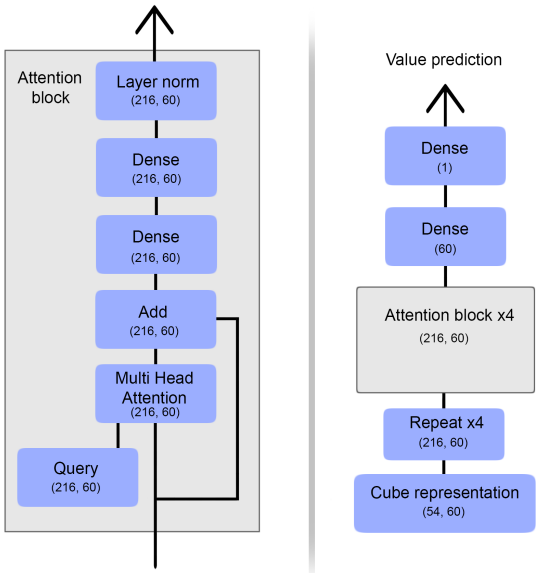

Figure 2: The structure of the stacked attention network. The numbers below the layer types denote the output shape of the respective layer.

The complete network then looks as depicted in Figure 2 (right): Starting with the embedding procedure, which works as described in Section 3.1, the resulting cube representation is repeated four times to make the dimensions match the input requirements for the attention block. This does not change the output of the network due to the nature of the attention layer processing this cube representation, since each of the four equivalent vectors will be added to the output for a specific query with one fourth of the weight assigned to it compared to the case of no repetition. Four consecutive attention blocks are used to extract useful information from the initial cube representation: The first attention block is supposed to extract some basic information from the cube, which is processed by the dense layers. With each new attention block, the network is supposed to create higher-level information by combining and processing the pieces of information created by the previous block. The output of the final block is then processed by two dense layers, the output of the second one acts as the final output of the network.

3.2.2 The Recurrent Attention Network

The recurrent attention network relies on a recurrent structure to repeatedly query for new information based on previously gathered knowledge about the current state of the cube. We start by looking at the operations that are done within each recurrent step. The design of the recurrent layers is displayed in Figure 3, in which two recurrent block are displayed to show the passing of information between them. We begin by using our current query vector q to query for information using an attention layer with self-attention. The query vector is determined by the previous recurrent step, the initial vector is a constant (but learned) initial query.

This results in a 60-dimensional vector containing new information about the current state of the cube. This vector is then fed into an LSTM-cell which works with 600-dimensional hidden state and output vectors. The output of the LSTM-cell is then processed by two dense layers to get our current estimate of the number of moves needed to solve the current cube state. During prediction, only the output from the last recurrent block is used as the final estimate of our current cost. In parallel, the output of the LSTM-cell is processed by another dense layer which generates the new query vector for the next recurrent step. The complete network then consists of input and embedding layers as discussed in Section 3.1 followed by ten recurrent blocks. The cost estimate of the final recurrent block is then used as the output of the neural network.

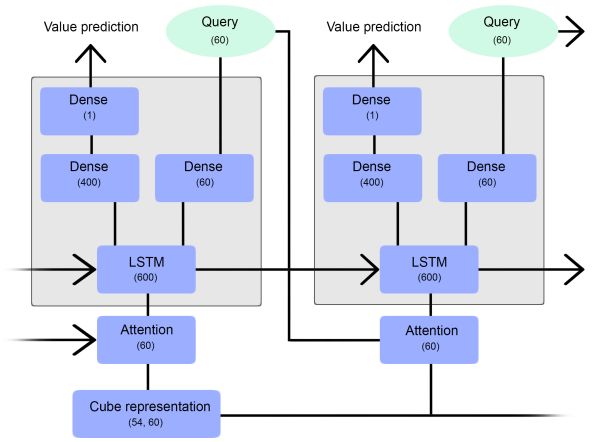

Figure 3: The structure of the recurrent attention network. The numbers below the layer types denote the output shape of the respective layer.

3.3 The Training Procedure

Solving the Rubik's Cube can be formulated as a Markov Decision Process (MDP). The neural networks then work as cost predictors for this MDP. A MDP consists of a state space, an action space, a state transition model and a reward function. For the given case of solving the Rubik's Cube, the state space is the space of all possible cube states and the action space is the set of possible moves, which is taken to be the set of moves available in qtm (all results and move counts will be stated with respect to qtm as well, see Appendix A for a list of all possible moves in qtm). The state transition model is a deterministic function specifying the resulting state for each state-action-pair. We chose the reward function to assign a negative reward of -1 to every action, which penalizes longer solutions due to more negative rewards being accumulated if more moves than necessary are taken. The MDP ends if the goal state is reached such that the absolute value of the cost accumulated while acting in this MDP is equivalent to the number of moves taken to solve the cube. For each state-action pair, the corresponding value function specifies the expected reward we would gain under our current policy (the model specifying which move to take in every state) starting from the current state. The goal of the neural networks is to learn the value function that correspond to the optimal policy, which is equivalent to learning the minimum number of moves required to solve every state. Having access to this function allows to chose perfect moves in each state by just computing the value function for each neighboring state and select the move that results in the state with the highest value function.

There are two main approaches for making neural networks learn the correct values for each state: The first one is having a training set of state-value pairs that can be used to train the networks. The second one is using the Bellman-Equation [14], which allows to use the current estimate of the value function to iteratively update it and converge (under specific conditions) to the optimal function. The update rule is the following:

$$V_{t+1}(s) = \max_a \{r(s,a) + \gamma V_t(s')\} \tag{1}$$

Here, V_{t+1} and V_t denote our estimates of the value function for the current state s at iteration t and $t+1$ respectively and $r(s,a)$ is the reward gained by applying action a in state s resulting in a transition to the state s'. γ is a discount factor that makes future rewards contribute less to the value of a state. Since the MDP is supposed to end within at most 20 moves (for the optimal policy) and since we want the cost function to specify the number of moves needed at the current state we choose γ to be 1. If we additionally recognize that the reward will be -1, independent from our current state or the action

chosen, we end up with the following update formula that can be used to train the networks:

$$V_{t+1}(s) = -1 + \max_a V_t(s') \tag{2}$$

Instead of learning this value function, the neural networks will actually learn the corresponding cost function, which just specifies the expected cost starting from our current state instead of the specifying the reward, which means that it is just the negative of the value function. This keeps the values positive and lets the networks predict the actual number of moves needed to solve the cube starting from the current state.
Both approaches to training the neural network will be used alternately, usually in a ratio of 100:1 in favour of the training set method. One advantage of using the Bellman update is that it does not require knowledge of the ground-truth value function. We can just randomly sample states and use our current value function estimate to find all information needed for the update. The creation of a training set, on the other hand, requires knowledge of the value function to create correct state-value pairs.
Fortunately, there is a way to quickly generate a high-quality training set of state-value pairs: We can just generate a scramble consisting of random moves and just take the length of the scramble and the resulting state as the state-value pair for our training set. This naive approach will not work if the moves are sampled completely randomly. The reason for this is that consecutive moves might cancel each other out, which will make the scramble longer but will keep the state close to the solved state. This can be prevented by creating a specialized sampling procedure that does not sample moves that cancel out previous moves. Nevertheless, not all sampled state-value pairs will be correct since many scrambles of a certain length might lead to states that can be solved in fewer moves. This becomes more likely the longer the scramble is, as can be easily seen by considering that every scramble of length 27 or more will for sure be solvable in 26 moves or less. Luckily, the number of scrambles for which this is the case stays very small even up to scrambles with length 18, for which, according to empirical tests, only about 2% of scrambles lead to states which are solvable in fewer moves (typically 16). This now allows us to create large training sets with values of up to 18 within seconds.
The reasons for using both update rules and the main advantages and disadvantages of both methods are discussed in the following Section.

4 Evaluation

4.1 Prediction Performance

To evaluate the prediction performance of the different models, we use an evaluation set consisting of 50 scrambles with corresponding optimal solutions found by the Cube Explorer program [5]. This allows to compare the length of the solution found by the different neural networks to the optimal solution length. To test the performance of the neural networks, we at first need a way to use the estimates of the value function to generate complete solutions. The most prominent way of doing this is to build a search tree and to iteratively expand the most promising nodes based on their value estimate and the moves needed to reach this node in the first place. A variant of this procedure was used in [1] to find optimal or near-optimal solutions to every given scramble. One problem with this approach is that the number of nodes that are expanded and evaluated is not determined ahead of time. Even if a solution is found, it is possible to expand more nodes in the hope of finding a shorter solution. This makes a comparison of different models hard, especially if we consider that even a neural network that returns completely random numbers as value estimates would eventually find the optimal solution because every path in the search tree will eventually be tried out (even though this is computationally infeasible). Thus, this approach of searching solutions does not allow to properly compare different models and to judge their performance.

Algorithm 1

```
1: procedure FindSolution(n_states, n_split, S)
2:     currentStates ← {S}
3:     visitedStates ← {S}
4:     while no solution found do
5:         newStates ← {}
6:         for state in currentStates do
7:             candidates ← ExpandNode(state)
8:             costs      ← CalculateCosts(candidates)
9:             candidates ← TakeBestStates(candidates, costs, n_split)
10:            newStates  ← newStates ∪ candidates
11:        newStates     ← newStates \ visitedStates
12:        visitedStates ← visitedStates ∪ newStates
13:        currentStates ← newStates
14:    return solution
```

The method of finding solutions to a given scramble used in this paper is displayed by Algorithm 1: We again build a search tree, but this time it has a fixed number of nodes and thus improves comparability between models. In every step, we have a number of nodes that are all at the same depth in the search tree. We expand all of these nodes, which means that we compute all 12 following states and their corresponding value estimates. At most n_{split} of these following 12 states are kept, this prevents a single path with a (maybe incorrect) good value to dominate the search tree and thus enforces diversity. n_{split} was chosen to be 9 since this empirically led to the best results. After expanding all nodes in our current set, we only keep the n_{states} best resulting states and expand these in the next iteration. This will lead to comparable results since all neural networks will have expanded the same number of nodes at each depth and every network will produce a single shortest solution without uncertainty about finding a better one later on. Additionally, it is possible to test how robust the networks are to using a smaller value for n_{states}.

Figure 4 displays the prediction results of four neural networks for different values of n_{states} by showing how much longer the average solution found by the neural networks is compared to the optimal solution. The networks compared here are the two networks proposed in this paper as well as DeepCubeA, which is tested once as it was trained by the authors of the original paper (denoted DeepCubeA Official) and once trained with the same training procedure used in this paper (denoted DeepCubeA). The effects of using different training procedures are further examined in Section 4.2. We see that the official DeepCubeA version performs best for every n_{states}. The stacked attention model is the better performing model among the two models proposed in this paper, being almost level with DeepCubeA Official in all but the smallest and the highest values for n_{states}. The recurrent attention model does not perform as well as the stacked attention model, falling far behind for every value of n_{states} except the highest one. The DeepCubeA implementation that was trained in the same way performs reasonably well for the higher values of n_{states} but it is the least robust one with respect to small values for n_{states}. Additionally, it performed worse than the stacked attention model for every value of n_{states}.

4.2 Training Procedures

As described in Section 3.3, a combination of using a training set of state-value pairs and using the Bellman update was chosen to train the models proposed in this paper. In contrast, DeepCubeA was originally trained using only the Bellman update. This way of training the network might be superior to using a training set with respect to prediction performance since the models trained with a training set suffer from the following problem: The states used in the training set were sampled randomly, if one state is present in the training set it might be the case that none of its neighboring states are in the training set. This leads to a behavior where the trained model predicts vastly different values for states

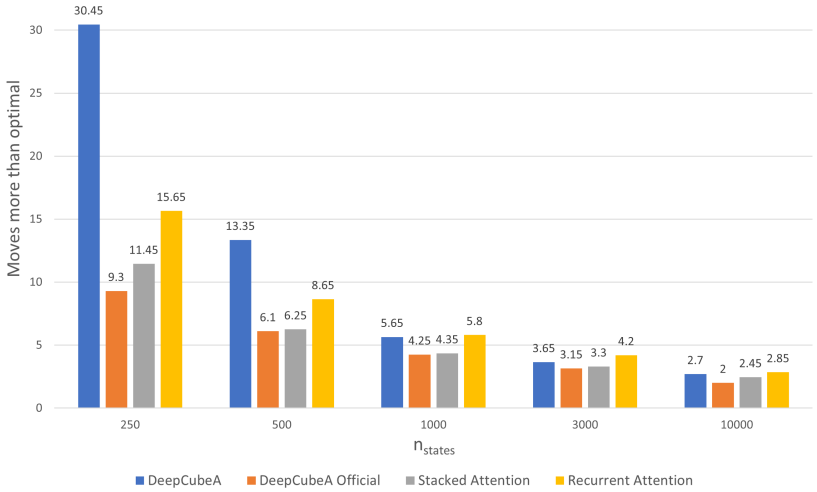

Figure 4:
Comparison of the prediction performance of the two neural networks proposed in this paper as well as DeepCubeA, once trained like the other two models and once trained by the authors of the original paper. The numbers indicate how many moves more than optimal were needed on average.

that are only a single move apart. If we check the predicted values for an exemplary optimal solution (displayed in Figure 5), we see that the predicted values show no progress for the first nine moves, predicting that there are around 16 moves left for each of the states. Later, there are some sudden drops, which means that the network predicts neighboring states to be more than one move apart. Additionally, sometimes the network predicts a sudden rise in moves left, which might be enough to sort out the optimal solution and only keep other paths that were incorrectly predicted to have less moves left. DeepCubeA on the other hand is way closer to the ground truth for all states close to the solved state, but it has problems to properly distinguish the states with higher move counts. In comparison to the stacked attention network though, it predicts around the same value for each one of them, whilst the stacked attention model predicts a sudden rise already on the first move, which might be enough to sort this path out if this rise is not present for lots of the other moves. So overall, this behavior makes it very hard to select the best moves, especially in the first nine moves of the solve. Training with the Bellman update automatically enforces the network to learn that neighboring states have corresponding values that are exactly one move apart, which seems to be able to at least prevent the sudden drops. Using the Bellman update in combination with the training set already led to a substantial improvement in prediction performance, so even incorporating it partly into the training procedure had this positive effect. The main drawback of this training procedure is the time it takes to train a model using only this update rule, which is further examined in Section 4.3.

From Figure 5, we additionally see that the general reason why none of the networks is able to reliably find the optimal solution to every given scramble is that they are not able to reliably distinguish all of the states that are more than 15 moves away from the solved state. This makes it impossible to select the best moves at the lower levels of the search tree and thus often prevents the network from finding the optimal solution.

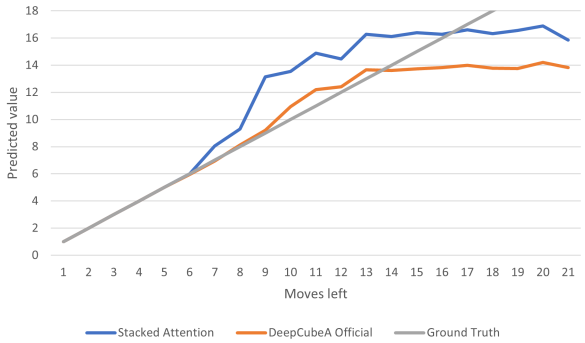

Figure 5: The predicted costs for all states within one optimal solve created by an optimal solver.

One other aspect regarding the training procedure is the use of curriculum learning [4]. Using this technique means that we start by presenting only states that are, for example, within five moves from the optimal solution in the first stage of training the network. After the network leaned to distinguish this smaller set of states, we extend the set of states that the network is trained on and thus gradually work towards training the network on the complete training set. This methods lead to a substantial improvement in prediction performance (about 19%) for the stacked attention model, for the recurrent attention model using curriculum learning had no effect.

It is worth noting that curriculum learning is automatically performed when using the Bellman update even though we show states for all possible move counts to the network. The reason is that the only fixed information we give to the network is that states that are one move off the solved state have a value of 1. Now, the network gradually learns, that states that are one move away from these states will have a value of 2, and in this way the value predictions unfold until states far away from the solved state are correctly assigned a high value. Thus, the network decides on its own how fast to progress with curriculum learning and it progresses in a soft manner, which eliminates the need of manually selecting the curriculum learning steps. This most likely is another reason for the good prediction quality of the original DeepCubeA model.

4.3 Training Speed

The networks proposed in this paper were trained using a single RTX 2060 Super graphics card, in contrast to the usage of two Ttitan V GPUs used to train DeepCubeA originally, with six further GPUs being used for data generation. The training of DeepCubeA took about 36 hours, while the training of the models proposed in this paper took about 48h with less than one tenth of the computing power available. The main reason for this is that the networks were trained using a training set consisting of state-value pairs. This makes the unfolding procedure, that was described in the previous Section, completely obsolete and thus leads to convergence orders of magnitude faster than just using the Bellman update. If no curriculum learning is used, the networks will be able to solve every given scramble after less than two hours of training, even though the solutions will be far longer than optimal. This shows that this training procedure is a more direct way to lead the network to a good state, even if the completely trained network might perform slightly worse.
The second reason for the longer training time with the Bellman update is that we need to use the neural network itself to generate the data set since the Bellman update relies on the current value function estimate. Additionally, we need to predict the values for all 12 consecutive states to compute the value for a single training sample. This slows the training procedure down enormously and thus is another factor for the superior training speed achieved by the training set method.

4.4 Number of Parameters

One of the main advantages of using attention mechanisms instead of just using dense layers is that the network is able to query for exactly the information it needs and thus does not need as many neurons to process the information it extracted. This leads to a much smaller parameter footprint: The stacked attention model and the recurrent attention model have about 1.2×10^6 and 1.87×10^6 trainable parameters respectively, which is around 10 times less than the 1.46×10^7 trainable parameters used in DeepCubeA. If we take into account that the stacked attention model performs almost as well as DeepCubeA, and even better than it if DeepCubeA is trained in the same way as it, it seems like using attention layers is a good way to extract useful information from the cube representation and to reduce the parameter footprint.

5 Discussion

This paper proposed two new, attention based, neural network architectures that can be used to solve the Rubiks Cube. These models achieve results comparable to current state-of-the-art approaches like DeepCubeA while reducing the parameter footprint and the training time by an order of magnitude.
One main shortcoming in this study is that it was not possible to train the proposed models using just the Bellman update to be able to better compare them to the original DeepCubeA network. The reason for this is the limited availability of computational resources for this study. This also means that it was not possible to test a broad range of hyperparameters, so different numbers of neurons, different learning procedures and in general slightly altered neural network architectures might be able to improve the results reported in Section 4.1 a lot.

One interesting possibility of having the trained neural networks is to investigate whether the neural networks solve the Rubik's Cube similar to human solvers. In usual speedsolving, reducing the actual number of moves is not as important as finding moves as quickly as possible. The most skilled solvers can find and execute a solution of around 50 moves in less than seven seconds, the resulting solves will not be comparable to the solutions found by the neural networks that just want to finde the shortest solution. On the other hand, there is a competition category that requires the participants to find the shortest solution to a given scramble within one hour, and in consequence many complex techniques were developed to find efficient solutions to any given scramble. Interestingly, a skilled solver in this category had a look at some solves produced by both neural networks and was not able to recognize any of these techniques in the solutions. Thus, we end up with another example of a neural network beating human performance with methods not comprehensible to humans.

In conclusion, this study showed that it can be worth investigating possible improvements with regards to the neural network structures or the training procedures. Just using dense neural networks and a general update formula like the Bellman update can lead to good results but suffer from a larger parameter footprint and longer training times. This result can be applicable to many other tasks, for which more specialized structures can lead to improvements in many different aspects.
The code for this project is available at `https://github.com/MarcBrinner/Cube`.

References

[1] Forest Agostinelli et al. "Solving the Rubik's cube with deep reinforcement learning and search". In: *Nature Machine Intelligence* 1 (Aug. 2019). DOI: 10.1038/s42256-019-0070-z.

[2] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. *Layer Normalization*. 2016. arXiv: 1607.06450 [stat.ML].

[3] *Barrycades and Septoku: Papers in Honor of Martin Gardner and Tom Rodgers.*

[4] Y. Bengio et al. "Curriculum learning". In: vol. 60. Jan. 2009, p. 6. DOI: 10.1145/1553374.1553380.

[5] *http://kociemba.org/cube.htm.*

[6] *http://www.math.rwth-aachen.de/~Martin.Schoenert/Cube-Lovers/michael_reid__new_upper_bounds.html.*

[7] *https://cube20.org/qtm/#:~:text=God's%20Number%20is%2026%20in%20the%20Quarter%20Turn%20Metric.*

[8] Colin Johnson. "Solving the Rubik's Cube with Learned Guidance Functions". In: Nov. 2018, pp. 2082–2089. DOI: 10.1109/SSCI.2018.8628626.

[9] Asifullah Khan et al. "A Survey of the Recent Architectures of Deep Convolutional Neural Networks". In: *CoRR* abs/1901.06032 (2019). arXiv: 1901.06032. URL: http://arxiv.org/abs/1901.06032.

[10] Richard E. Korf. "Finding Optimal Solutions to Rubik's Cube Using Pattern Databases". In: *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Conference on Innovative Applications of Artificial Intelligence*. AAAI'97/IAAI'97. Providence, Rhode Island: AAAI Press, 1997, pp. 700–705. ISBN: 0262510952.

[11] *M. Thistlethwaite, The 45-52 Move Strategy, 1981.*

[12] Stephen McAleer et al. *Solving the Rubik's Cube Without Human Knowledge*. 2018. arXiv: 1805.07470 [cs.AI].

[13] Daniel W. Otter, Julian R. Medina, and Jugal K. Kalita. "A Survey of the Usages of Deep Learning in Natural Language Processing". In: *CoRR* abs/1807.10854 (2018). arXiv: 1807.10854. URL: http://arxiv.org/abs/1807.10854.

[14] *R. E. Bellman. Dynamic Programming. Princeton University Press, 1957.*

[15] Silviu Radu. "A New Upper Bound on Rubik's Cube Group". In: (Jan. 2006).

[16] Tom Rokicki et al. "The Diameter of the Rubik's Cube Group Is Twenty". In: *SIAM J. Discrete Math.* 27 (Apr. 2013), pp. 1082–1105. DOI: 10.1137/120867366.

[17] David Silver et al. "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm". In: *CoRR* abs/1712.01815 (2017). arXiv: 1712.01815. URL: http://arxiv.org/abs/1712.01815.

[18] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529 (Jan. 2016), pp. 484–489. DOI: 10.1038/nature16961.

[19] Oriol Vinyals et al. "Grandmaster level in StarCraft II using multi-agent reinforcement learning". In: *Nature* (2019), pp. 1–5.

Appendices

A Rubiks's Cube Metrics

There are two main ways of measuring distances between two states a Rubik's Cube can be in: The *quarter turn metric (qtm)* and the *half trun metric (htm)*. Both metrics have a corresponding notation in which it is possible to express which moves are needed to get from one state to another. Figure 6 displays the set of moves available in quarter turn metric, which are all moves that can be performed by turning one face of the cube 90 degrees. Two identical moves (for example the move sequence *R, R*) are considered two moves in qtm, while these moves would be considered a single *half turn* in htm, denoted by the single move *R2*. This leads to certain states being closer together if measures in htm, and it reduces the minimum number of moves required to solve every possible state the Rubik's Cube can be in (named God's Number) from 26 to 20.

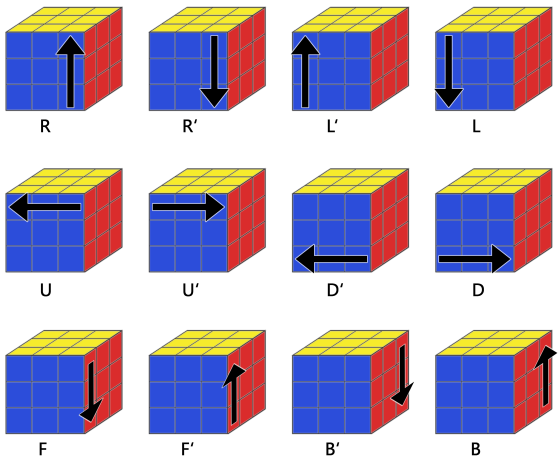

Figure 6: All moves available if measured in quarter turn metric.

B Implementational Details

Exact layer shapes and neuron counts of the neural networks are given by Figure 2 and Figure 3. The exponential linear unit activation function was used for all dense layers, except the final dense layer of the attention block as well as all output layers, which both just use the a linear activation function.
The networks were trained using the mean squared error loss function in combination with the Adam optimizer. For the recurrent attention network, just the cost prediction from the last recurrent step was used as the predicted output to compute the loss. The exception to this are the first iterations of training, where this loss was computed with respect to all outputs of every recurrent step (downweighted by a factor of 0.001 for all but the last output), which initially speeds up the optimization.
The batch size was chosen to be 128, the networks were trained until the performance on a separate test set reached a maximum, the best parameters with respect to this test set were chosen for evaluation.
The two training methods (training set and Bellman update) were used alternately, the number of samples in each training set was 2×10^6, the number of samples for the Bellman update was 2×10^4, leading to a ratio of 100:1. A new training set was generated for each iteration.