Chair of Computer Science 6 (Machine Learning and Reasoning)

RWTH AACHEN UNIVERSITY

The present work was submitted to the Chair of Machine Learning and Reasoning.
Diese Arbeit wurde vorgelegt am Lehrstuhl für Maschinelles Lernen und Inferenz.

Investigating Policy Gradient Methods for Rubik's Cube

Untersuchung von Policy Gradient Methoden auf dem Rubik's Cube

Bachelor Thesis
Bachelorarbeit

Presented by / Vorgelegt von

Allegra Nagel
434209

Supervised by / Betreut von Martin Theisen, M.Sc.

1st Examiner / 1. Prüfer Prof. Hector Geffner, Ph.D.

2nd Examiner / 2. Prüfer Prof. Dr. rer. nat. Christopher Morris

Aachen, June 2, 2025

Abstract

The Rubik's Cube (RC) is a challenging environment due to its large state space and sparse rewards. While prior work has successfully solved it, this thesis investigates whether it can be solved using only Reinforcement Learning (RL) without relying on handcrafted rules, search algorithms, or supervised data. The focus is on policy gradient methods, particularly REINFORCE and an actor-critic approach. A custom reward function is introduced to support learning, and various neural network architectures are explored, including Multi Layer Perceptrons (MLPs), Recurrent Neural Networks (RNNs), Convolutional Neural Networks (CNNs), and Graph Neural Networks (GNNs), to evaluate how well they can help learning in this setup. Although the agent is not able to consistently solve the $3 \times 3 \times 3$ cube, the results highlight what works and what does not. They offer valuable insights into the strengths and limitations of applying pure RL to structured, complex problems like the RC, especially regarding reward shaping, network architecture, and training strategies.

Contents

1 Introduction

The Rubik's Cube (RC) is a well-known combinatorial problem that is characterized by its complexity, large state space, and sparse reward structure. Despite its relatively simple and deterministic rules, it remains a challenging benchmark for evaluating learning strategies. Traditional appraoches, like heuristic search methods, can solve the cube optimally, but they typically rely on a lot of domain knowledge, handcrafted rules, or large precomputed databases that limit how much they can be applied to other problems.

In recent years, Reinforcement Learning (RL) has shown promise in learning to solve problems through environment interaction. Policy gradient methods, in particular, perform well in high-dimensional action spaces and are well-suited for learning policies that generalize across many states. Although previous research has explored hybrid approaches combining RL with supervised learning or search algorithms to solve the RC, pure policy gradient methods have been mostly overlooked.

This thesis explores whether the RC can be effectively solved using pure RL, eliminating the need for supervised guidance, handcrafted heuristics, or search-based planning. The focus is on assessing the potential of policy gradient methods in this highly structured and sparsely rewarded environment. We apply the standard policy gradient method REINFORCE and investigate various strategies to achieve learning in this domain.

Our approach involves designing a dense reward function that enables learning signals even when the cube is not solved, as well as exploring different variations to improve performance. We experiment with different neural network architectures for policy learning, including Multi Layer Perceptrons (MLPs), Recurrent Neural Networks (RNNs), Convolutional Neural Networks (CNNs), and Graph Neural Networks (GNNs). Each architecture offers different advantages in handling spatial or temporal dependencies. We also assess the effectiveness of actor-critic methods as a more stable alternative to REINFORCE.

Our evaluation targets the main challenges the RC poses: a sparse reward signal, large state space, and long solution paths. We examine how different RL techniques and network architectures interact with these challenges, aiming to understand both the advantages and limitations of policy gradient methods in such settings. While our approach does not achieve high performance in fully solving the RC, the results offer valuable insights into how architecture and reward design affect learning in the RC environment.

2 Background

As we explore policy gradient methods applied to solving the RC, we rely on foundational ideas from RL and neural networks. This chapter introduces the core concepts required to understand the methods and experiments discussed in later chapters. It begins with an overview of RL, including fundamental concepts, policy gradient methods, and key challenges in the field. It then presents the neural network architectures relevant to this work, MLPs, RNNs, CNNs and GNNs.

2.1 Reinforcement Learning

In an experiment conducted by psychologist B.F Skinner, rats received food when they pressed a lever while a green light was on. They received an electric shock when they pressed the lever while a red light was on. After some time, they learned to press the lever only when the green light was on [4]. This is called operant conditioning, the phenomenon whereby behaviors individuals freely choose to perform become more or less frequent, depending on whether they are followed by a reward or punishment [4]. This same concept was then adopted in Computer Science under the name of Reinforcement Learning (RL) to teach an artificial agent to make decisions.

In RL, the agent interacts with an environment. The agent makes the decisions which actions to take and is the party that learns to make better decisions. The environment is the system in which the agent operates. It encodes all the information important for the decision and can be in a specific state. In Skinner's experiment, the rat is the agent, the light is the environment, and to press or not to press the lever are the actions. At each timestep, t, the agent observes the current state s_t, chooses an action a which transforms the environment from state s_t into state s_{t+1}, and then observes the new state, which it then uses to make another decision. Together with the state, the agent also receives a reward r_{t+1}. This enables the agent to learn in a way similar to the food (positive reward) or the electric shock (negative reward) the rat receives.

The concepts and terminology used to describe RL in this section are based on Sutton and Barto's foundational work [32].

2.1.1 Markov Decision Process

An RL environment is commonly modeled using a Markov decision process (MDP). We begin by introducing the Markov process (MP), which is a memoryless process consisting of states and transition probabilities. The MP is always in a certain state. From this state, there are

probabilities that the MP will transition from this state into another. A Markov rewards process

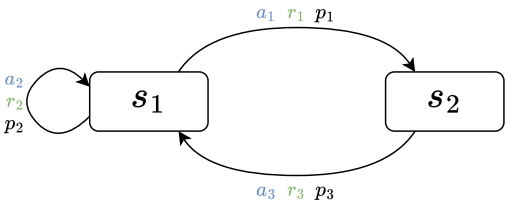

Figure 2.1: An MP (black) extended with rewards (green) and actions (blue) (deterministic)

(MRP) adds rewards to that. The purpose of the reward is to give the agent information about the environment. So, for example, if a green light is on and the rat can either press a button or not press it, the rat would receive a higher reward for pressing the button. An MDP adds actions to that (see figure 2.1), so instead of there only being transition probabilities, we have probabilities for the actions we can take. So now, if the green light is on, the rat might assign a high probability to the action of pressing the button and a low probability to not pressing it. In non-deterministic environments, we also account for transition probabilities. This means that even if the agent selects a certain action, there is still a chance that a different action is executed. For example, if the green light is on and the rat chooses to press the button, it might happen that, in a small number of cases, it gets distracted and does not press the button.
Formally an MDP is defined as $\langle \mathcal{S}, \mathcal{A}, \mathcal{R}, p(\cdot|\cdot) \rangle$ where:

- $\mathcal{S}$ is the set of all states the environment can be in,
- $\mathcal{A}$ is the set of all actions the agent can choose from in each state,
- $\mathcal{R}$ is the set of all possible rewards the agent can receive,
- $p(s', r|s, a)$ is the dynamics function

The dynamics function describes the probability of reaching the state s' and the reward r when being in state s and applying action a. In practice, these transition and reward dynamics are usually unknown to the agent, which must learn about the environment through interaction alone.
The Markov property states that the distribution of the current state depends only on the previous state and not on the history of states. This is true for MPs, MRPs, and MDPs since the state transition probability or the action probability depend only on the state we are currently in.

$$\mathbb{P}[S_t|S_{t-1}] = \mathbb{P}[S_t|S_{t-1}, S_{t-2}, S_{t-3}, ...] \tag{2.1}$$

This is often automatically guaranteed because the state we are in encodes all the information the agent needs to make a decision. The Markov property is helpful since it simplifies the modeling of the system by eliminating the need to track the entire history of states.
As mentioned earlier, MDPs provide a formal representation of the environment and the problem in RL, which aims to find a solution to these MDPs.

2.1.2 Fundamental Concepts

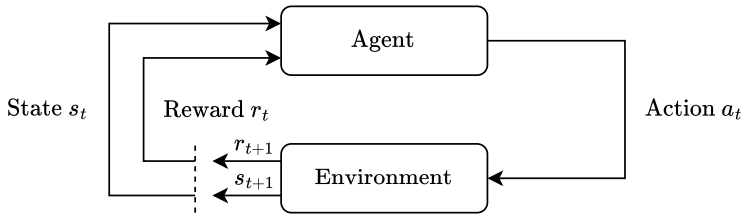

Figure 2.2: The agent-environment interactions, figure adapted from Sutton and Barto [32]

The learning process in RL can take two forms, depending on the type of task being solved: episodic or continuing. In an episodic task, the agent interacts with the environment over a finite sequence of steps, forming an episode. Each episode starts in an initial state and ends when a terminal state is reached. The set of terminal states is denoted by $\mathcal{S}_T$, and reaching any terminal state $s_{\text{term}} \in \mathcal{S}_T$ ends the episode. The number of steps in an episode, denoted by T, can vary between episodes. After each episode, the environment resets to the starting state, and episodes are independent of one another. A typical example of an episodic task is a single playthrough of a game. In contrast, continuing tasks have no terminal state and no natural end. The agent-environment interaction continues indefinitely and must be stopped manually if needed. A common use case of continuing tasks is balancing a pole on a moving cart, where the agent must act continuously to keep the pole upright.

To solve an MDP, the agent requires additional information to the observation it makes about the state of the environment. This is necessary because it does not know what it is supposed to do in the environment. This additional information is feedback the environment gives in the form of a *reward signal*.

The reward $r_{t+1} = r(s_t, a_t) \in \mathbb{R}$ provides the agent with feedback about the value of the action it has just taken. Every time the agent takes an action a and transitions from state s_t to state s_{t+1}, it receives an observation about s_{t+1} and the reward r_{t+1} (See figure 2.2). The reward is used for the agent to learn what the goal of the problem is and needs to be defined task-specifically. The agent learns to solve a task by maximizing the return G it receives. The return is the accumulated reward r:

$$G_t = \sum_{k=0}^{N} \gamma^k r_{t+k+1} \tag{2.2}$$

where $\gamma \in [0, 1)$ is the discount factor, which determines how important future rewards are. The larger γ, the more emphasis is put on future rewards. The discount factor is necessary because in continuous tasks $T = \infty$, potentially causing the return to become infinite.

The return is used to shape the *policy* π the agent learns, which describes the behavior of the agent. The policy determines how the agent selects actions based on the current state. For small state and action spaces, the policy can be represented as a simple lookup table, as each state can be directly mapped to an action, i.e., $\pi(s) = a$. In more complex settings, the policy must

be learned, for example, using a function approximator such as a neural network or through a search process. The policy can be either deterministic or stochastic. In a deterministic policy, the agent always selects the same action for a given state. In contrast, in a stochastic policy, actions are chosen according to a probability distribution. This is denoted as $\pi(a \mid s)$. In most applications, stochastic policies are used. For finite MDPs, there is always at least one policy that is better than or equal to all other policies. This is the optimal policy π_*
Instead of directly learning a policy, an agent can also learn a *value function*. A value function assigns a numerical value to each state of the environment, representing how good it is for the agent to be in that state in terms of the expected future return. Formally, the *state-value function* for a policy π is defined as:

$$v_\pi(s) = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \,\middle|\, S_t = s\right] \tag{2.3}$$

This expected return reflects not only the immediate reward but also the rewards of future states the agent is likely to visit if it follows policy π from state s. So, the reward for a single state can be quite low, if the reward for all the states that usually follow it is high, then the value of that previous state can still be high. Once the agent has a value function, it can use it to define a policy. A simple example is always choosing the action that leads to the state with the highest estimated value.
Using value functions to define a policy implicitly guides the agent's decisions based on the long-term desirability of states. However, estimating value functions can be challenging since the agent typically has limited knowledge of the environment and must learn from its own observations. In addition to state values, the agent can also learn the value of taking a specific action in a given state. This is captured by the *action-value function*:

$$q_\pi(s, a) = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \,\middle|\, S_t = s, A_t = a\right] \tag{2.4}$$

With these foundational elements introduced, the central goal of RL can now be stated as follows: the goal of RL is to find or approximate an optimal policy π_* that maximizes the expected return:

$$\pi_* = \arg\max_\pi \mathbb{E}_\pi\left[\sum_{t=0}^{\infty} \gamma^t r_{t+1}\right] \tag{2.5}$$

2.1.3 Learning a Policy

There are different types of policies and approaches used to learn policies, which can be broadly categorized into on-policy and off-policy methods, as well as value-based and policy-based learning strategies. The difference between on-policy and off-policy methods is whether the same policy is used for both decision making and learning. The target policy is the policy that the agent tries to learn and the one that is actually used in the end. It should become

the optimal policy or an approximation of it. The behavior policy is the policy used to explore the environment and generate data, i.e., sequences of states, actions, and rewards. *On-policy methods* are methods where the target and the behavior policy are the same. This means that the agent learns from the actions it takes based on its current policy. Because of this, some exploration is necessary to improve the policy. This makes purely optimal behavior during training unlikely, as exploration requires deviating from the current best-known action. This may result in on-policy methods taking longer to learn the optimal policy or being less sample efficient compared to off-policy methods. In *off-policy methods*, the target function and behavior function are two separate functions. The agent uses one policy to generate experience (typically more exploratory or stochastic) and a different, often deterministic, policy to learn from that experience. One advantage of off-policy methods is greater flexibility, such as using previously collected data from other agents or policies. Algorithms like Q-learning and Deep Q-Networks (DQNs) are classic examples of off-policy methods.
Another important distinction is between value-based and policy-based approaches. *Value-based approaches* focus on learning the value function $v(s)$ or the action-value function $q(s, a)$. The policy is only learned indirectly by choosing the action that maximizes that function. *Policy-based methods*, on the other hand, optimize the policy directly. These methods typically define the policy as a parameterized function $\pi_\theta(a|s)$ and adjust the parameters θ to maximize the expected return. This class of methods includes REINFORCE and Actor-Critic algorithms. Policy-based methods often perform better in environments with high-dimensional or continuous action spaces, where defining or computing a value function can be more challenging. Additionally, they can naturally represent stochastic policies, which is useful for exploration.

Policy Gradient Methods

Policy Gradient Methods are policy-based methods, so the policy is learned directly. The policy is parametrized by parameters θ. The goal is to optimize θ so that for $\pi_\theta(a \mid s)$, the weights for the good actions increase, and the weights for the bad actions decrease. Formally, we aim to maximize the expected return, meaning we choose θ so that $J(\theta) = \mathbb{E}[R(\tau)]$ is maximized where $\tau = (s_0, a_0, s_1, a_1, \dots)$ is a trajectory generated under the current policy π_θ. The update for θ is $\theta_{t+1} = \theta_t + \alpha \nabla J(\theta_t)$. The gradient can then be defined by:

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a q_\pi(s, a) \nabla \pi(a \mid s, \theta) \tag{2.6}$$

$$= \mathbb{E}_\pi \left[\sum_a q_\pi(s, a) \nabla \pi(a \mid s, \theta) \right] \tag{2.7}$$

Here, the sum over states can be rewritten as an expectation by interpreting $\mu(s)$ as the stationary distribution over states under the policy. This allows the gradient to be expressed more compactly as an expectation with respect to the policy, as shown in (2.7).
REINFORCE is a policy gradient algorithm. In theory, it is based on the gradient given in Equa-

tion (2.6). However, computing this gradient exactly would require knowing $q_\pi(s, a)$ for all s, a and the full model of the environment, which is usually unavailable. Therefore, an alternative gradient estimator based on sampling is used in practice. We move from the theoretical gradient to a sample-based gradient by approximating the expectation with actual sampled trajectories. The action a becomes the sampled action A_t at time t. Furthermore, the action-value $q_\pi(s, a)$ is approximated by the sampled return G_t from time t onward. Applying the logarithmic derivative trick allows us to write the gradient in terms of $\nabla_\theta \pi(a \mid s) = \pi(a \mid s) \nabla_\theta \log \pi(a \mid s)$ leading to the final REINFORCE update rule:

$$\theta_{t+1} = \theta_t + \alpha \gamma^t G \nabla \ln \pi(A_t | S_t, \theta_t) \tag{2.8}$$

This update rule is unbiased but can suffer from high variance, which negatively affects learning stability. A common technique to reduce this variance without introducing bias is to subtract a baseline $b(s_t)$ from the return:

$$\theta_{t+1} = \theta_t + \alpha \gamma^t (G_t - b(s_t)) \nabla_\theta \log \pi_\theta(A_t \mid S_t). \tag{2.9}$$

If the baseline is chosen as the state-value function $b(s_t) = v_\pi(s_t)$, the term $(G_t - v_\pi(s_t))$ becomes the advantage estimate, which measures how much better an action was compared to the average at that state.

Actor-Critic Methods

Actor-Critic Methods combine elements of both policy-based and value-based RL. They learn the policy and the value function at the same time. The actor is responsible for learning the policy, and the critic is responsible for learning the value function. The policy is learned with the parameters θ, and the value function is learned with parameters ω. It is used to assess the action taken and to suggest the update direction of policy parameters. The advantage of this is that now we do not need to wait until the end of the episode to perform an update step because we do not need the whole return, since the update uses the Temporal Difference (TD) error defined as:

$$\delta_t = (R_{t+1} + \gamma \hat{v}(S_{t+1}, \omega_t) - \hat{v}(S_t, \omega_t)) \tag{2.10}$$

The TD error measures how much better or worse the outcome was than expected and approximates the advantage function. The actor updates the policy parameters θ in the direction of the estimated advantage:

$$\theta_{t+1} = \theta_t + \alpha \, \delta_t \nabla_\theta \log \pi_\theta(A_t \mid S_t), \tag{2.11}$$

while the critic updates the value function parameters ω by minimizing the mean squared TD error:

$$\omega_{t+1} = \omega_t + \beta\, \delta_t \nabla_\omega \hat{v}_\omega(S_t), \tag{2.12}$$

where α and β are separate learning rates for the actor and the critic, respectively.
An important advantage of Actor-Critic methods is that they allow for online and incremental updates: the agent does not need to wait until the end of an episode to update its parameters, enabling faster learning and better scalability to continuing tasks. Additionally, by learning the policy and value function at the same time, these methods typically reduce the variance in gradient estimates compared to pure policy gradient approaches[6], which improves learning stability and efficiency. This combination also allows for more flexibility in approximating a function, which makes Actor-Critic methods especially effective in complex environments with many possible states and actions, including continuous ones [34].

2.1.4 Common Challenges

Two challenges that occur often in RL are the trade-off between exploration and exploitation and sparse rewards.
The *exploration vs. exploitation trade-off* describes whether an agent should exploit its current policy to maximize immediate returns or explore new actions to discover a better policy. Once a policy is found that performs relatively well, continuing to exploit it may yield consistently good results. However, doing so might prevent the agent from discovering an even better strategy. Exploring unfamiliar actions can potentially lead to higher long-term rewards but may also result in poor performance in the short term. One method that addresses the trade-off is the use of off-policy methods. As mentioned, the target policy is the one we aim to optimize and typically follows the currently learned strategy. The behavior policy, in contrast, is used to explore by taking actions that may deviate from the current policy. Another simpler method to work against the trade-off is the ε-greedy policy. In this method, the agent selects the action according to:

$$A = \begin{cases} \arg\max_a \pi(a, s) & \text{with probability } 1 - \varepsilon \\ \text{a random action} & \text{with probability } \varepsilon \end{cases} \tag{2.13}$$

Here ε typically is a very small value. This approach ensures that most of the time, the agent chooses what it currently believes to be the best action, but with a small probability, it explores other actions. This random exploration helps avoid local optima and allows the agent to gather information about alternative strategies.
Another frequent challenge are *sparse rewards*. They occur when the agent only occasionally receives a reward, for example, when the entire problem has been solved completely or when reaching certain subgoals. An example is a task set in a maze, where the agent only receives a

positive reward upon finding the exit and gets no feedback otherwise. In sparse reward settings, the agent might perform many episodes without receiving any reward signal at all. This could become a problem as the agent may never reach a state where it will get a reward, or it does not reach enough states to learn from. Since the gradient depends on the reward, we cannot perform parameter updates without it, and the agent cannot learn. The opposite of sparse rewards are dense rewards, where the agent gets a non-zero reward every step. To avoid sparse rewards, one can handcraft dense reward functions or introduce additional subgoals to receive rewards more frequently.

2.1.5 Rubik's Cube

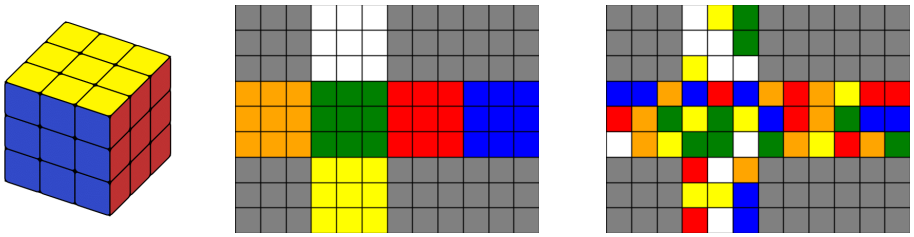

Figure 2.3: (left) 3D Rubik's cube, (center) unfolded cube grid, (right) scrambled cube grid.

The Rubik's Cube (RC) is a combinatorial puzzle that consists of a $3 \times 3 \times 3$ cube, where each of the six faces is covered by nine stickers of a single color: orange, green, red, blue, white, and yellow. The cube's mechanism allows each face to rotate independently, which scrambles and rearranges them. Each face has a fixed center piece surrounded by edge and corner pieces. While the latter can change positions during manipulation, the center pieces remain fixed and define the color of their respective sides in the solved state. There are various moves to manipulate the cube [30]. In the Quarter-Turn Metric (QTM), there are twelve basic moves: each of the six faces can be rotated 90 degrees clockwise or counterclockwise. For example, the action `F` rotates the front face clockwise, while `F'` performs a counterclockwise rotation. Another metric is the Half-Turn Metric (HTM), which includes 180 degree turns of faces, which increases the total number of actions to 18. The RC is scrambled by applying a sequence of these moves. However, not every arrangement with exactly nine stickers of each color is actually valid. Because of the cube's internal mechanics, one can not just randomly assign colors to the facelets, as many of those configurations would be impossible to reach with real moves [17]. There are more than $4.3 \cdot 10^{19}$ different configurations of the cube [30]. Despite this large search space, it has been proven that any configuration of the RC can be solved in 26 moves or fewer using the quarter-turn metric. This upper bound is often called God's Number, which denotes the maximum number of moves required to solve the cube from the position farthest away from the goal [23]. While the standard $3 \times 3 \times 3$ cube is the most common one, there are many other versions, like the $2 \times 2 \times 2$, the $4 \times 4 \times 4$, and larger ones such as the $5 \times 5 \times 5$ or even the $17 \times 17 \times 17$. Each comes with its own unique structure, complexity, and

solving challenges.
The RC is well-suited for RL because it offers a clearly defined set of states, deterministic moves, and a well-defined goal. The agent interacts with the cube by choosing actions (rotations of the cube's faces) and observing how the cube's configuration changes as a result. The RC typically uses a sparse reward structure, where the agent receives a reward only when the cube is fully solved.
Although the clear structure of the RC makes it a suitable environment for RL, it also presents several challenges. First, the large state space of the RC makes it extremely unlikely for an agent to reach a solved state through random exploration alone. Second, the environment typically provides a sparse reward signal, offering feedback only when the cube is solved. Third, the long solution paths required to reach the goal make it hard for the agent to evaluate which actions were the most beneficial. Finally, many cube configurations look similar or are symmetric but still require entirely different solutions, increasing the complexity of the decision-making process [2].

2.2 Neural Networks

Neural Networks (NNs) are a class of function approximators inspired by biological neural systems. They consist of interconnected layers of units called neurons. Each neuron receives a set of inputs and applies a computation, producing an output that is passed to neurons in the next layer [7]. The computation is a non-linear activation layer applied to a weighted sum. Formally, a single layer applies the transformation:

$$y = \sigma(Wx + b), \tag{2.14}$$

where $x \in \mathbb{R}^D$ is the input vector, W is the weight matrix, b the bias vector, and σ a non-linear activation function (e.g., Rectified Linear Unit (ReLU), sigmoid, tanh). The entire network is composed of multiple such layers stacked sequentially: an input layer, one or more hidden layers, and an output layer. The weights and biases are denoted as the parameter θ. The activation functions must be non-linear. Otherwise, stacking multiple layers would still result in a linear transformation. For instance, the composition of two linear layers: $f(x) = a_2(a_1x + b_1) + b_2$ can be simplified into a single linear transformation $g(x) = cx + d$ where $c = a_1 \cdot a_2$ and $d = a_2 \cdot b_1 + b_2$. Thus, non-linear activation functions enable networks to approximate complex, non-linear functions [13]. A common architecture is the Multi Layer Perceptron (MLP), which consists of fully connected layers where each neuron in one layer is connected to every neuron in the next [7]. An MLP learns a function $f : \mathbb{R}^D \to \mathbb{R}^K$, where D is the input dimension and K is the output dimension (e.g., number of classes in a classification task). The model learns a mapping from inputs $x \in \mathbb{R}^D$ to outputs $y \in \mathbb{R}^K$ by optimizing its parameters. For example, a basic two-layer MLP with one hidden layer of μ units can be

written as:

$$y = g^{(2)}\big(W^{(2)}g^{(1)}\big(W^{(1)}x + b^{(1)}\big) + b^{(2)}\big) \tag{2.15}$$

where $g^{(1)}$ and $g^{(2)}$ are activation functions, $W^{(1)} \in \mathbb{R}^{\mu \times D}$ and $W^{(2)} \in \mathbb{R}^{K \times \mu}$ are weight matrices for the first and second layer respectively and $b^{(1)} \in \mathbb{R}^{\mu}$ and $b^{(2)} \in \mathbb{R}^{K}$ are bias terms for the hidden and output layers.
Training an NN involves finding parameters θ that minimize a loss function $L(\theta)$ [13], which in supervised learning measures the error between the network's predictions and the ground truth. A common optimization method is Stochastic Gradient Descent (SGD). At each iteration, the parameters are updated in the direction opposite to the gradient of the loss:

$$\theta_{t+1} = \theta_t - \eta \nabla_{\theta_t} L(\theta_t), \tag{2.16}$$

where $\eta > 0$ is the learning rate. SGD iteratively reduces the loss by adjusting the parameters in a way that decreases the prediction error. To compute gradients efficiently in deep networks, the backpropagation algorithm is used. It applies the chain rule to propagate errors backward through the layers and compute the gradient of the loss with respect to each parameter in the network.
The form of the loss function depends on the learning paradigm. In supervised learning, where training data consists of input-output pairs (x_n, t_n), a common loss function for classification tasks is the cross-entropy loss:

$$L(\theta) = -\sum_{n=1}^{N}\sum_{k=1}^{K} \mathbb{I}(t_n = k) \ln p(C_k \mid x_n; \theta), \tag{2.17}$$

where $p(C_k \mid x_n; \theta)$ is the model's predicted probability for class C_k given input x_n. Intuitively, cross-entropy penalizes incorrect predictions more strongly the more confident the model is in making them. Other learning paradigms include unsupervised learning and RL, which use different loss formulations. In RL, the model is trained to maximize expected cumulative rewards (see Section 2.1.2).

2.2.1 Recurrent Neural Networks

A Recurrent Neural Network (RNN) is a type of NN designed to handle sequential data. What distinguishes RNNs from standard feedforward networks is that, in addition to the regular input, they also take a hidden state h_t as input. This hidden state enables the model to keep information across time steps. So RNNs handle the input in sequential timesteps. At each time step t, the RNN receives the input x_t and the hidden state from the previous step h_{t-1}. It then

computes the new hidden state:

$$h_t = \sigma(Wx_t + Uh_{t-1} + b), \tag{2.18}$$

where W, U, b are parameters, and σ is a non-linear activation function. The hidden state h_t serves both as the output for the current step and as input for the next. This recurrence allows RNNs to process sequences of arbitrary length. So, the advantages of RNNs are that they can model temporal dependencies in data. They are suitable for variable-length inputs (e.g., text, time series, speech) and support parameter sharing across time steps, making them memory-efficient [7].
Training RNNs is more difficult than training feedforward networks, especially for long sequences. Two common problems are vanishing gradients and exploding gradients [28], which arise during Backpropagation Through Time (BPTT), a version of backpropagation used to train RNNs by unfolding them across time steps. These can be formalized as follows:

$$\text{Vanishing:} \quad \left|\frac{\partial h_t}{\partial h_{t-k}}\right| \ll 1 \qquad\qquad \text{Exploding:} \quad \left|\frac{\partial h_t}{\partial h_{t-k}}\right| \gg 1 \tag{2.19}$$

Vanishing gradients are problematic because the network cannot learn long-range dependencies: gradients diminish as they propagate backward, and earlier time steps receive almost no signal. Exploding gradients, on the other hand, cause excessively large updates, which can make the optimization unstable or diverge. To avoid this, one can use, e.g., gradient clipping, which limits the magnitude of gradients to avoid instability [28], or ReLU activations instead of saturated functions like sigmoid. This can help reduce vanishing gradients, but does not fully solve the problem. More robust solutions involve gated architectures, such as Long Short-Term Memorys (LSTMs) and Gated Reccurent Units (GRUs).
The LSTM is a specialized RNN architecture introduced to overcome vanishing/exploding gradients. It is a more complex memory unit that consists of input, output, and forget gates. The input gate decides how much of the new input to incorporate. The forget gate determines what information from the previous cell state to discard. The output gate decides what to output. In addition to the hidden state h_t, LSTMs also use a separate cell state c_t, that allows gradients to flow more easily and helps the learning of long-term dependencies [16].
A simpler alternative to LSTMs are GRUs, which reduce the number of gates while preserving the advantages of the LSTMs. They combine the cell and hidden state into a single vector and merge the input and forget gates into an update gate. The GRU updates are defined as:

$$z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z) \qquad \text{update gate} \tag{2.20}$$

$$r_t = \sigma(W_r x_t + U_r h_{t-1} + b_r) \qquad \text{reset gate} \tag{2.21}$$

$$\hat{h_t} = \Sigma(W_h x_t + U_h(r_t \odot h_{t-1}) + b_h) \qquad \text{candidate hidden state} \tag{2.22}$$

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \hat{h_t} \qquad \text{hidden state} \tag{2.23}$$

The candidate hidden state $\hat{h}_t$ represents the new information to be stored. The reset gate determines how much of the past information should be included. So if r_t is close to zero, then very little of h_{t-1} is included, but if r_t is close to one, then most information from the previous hidden state is preserved. The update gate works similarly just that it determines how much of the old hidden state is kept and how much is new information. So GRUs in contrast to LSTMs have fewer parameters, so they are easier to train and tune and often have comparable performance, expecially with smaller datasets. This gating mechanism lets GRUs flexibly choose when to update their memory and when to keep it unchanged, which helps them capture patterns over both short and long time spans [9].

2.2.2 Convolutional Neural Networks

While RNNs handle sequential data, Convolutional Neural Networks (CNNs) are designed to process grid-like data, such as images. Images can be interpreted as 2D grids of pixel values. Using a standard MLP for images would require one input neuron per pixel, leading to a large number of parameters. CNNs however, use the same parameters across different locations, this is called parameter sharing [24]. Instead of assigning a unique weight to each pixel, they apply the same filter (or kernel) across the entire image. This reduces the number of trainable parameters and enables the network to generalize features across spatial locations [7].
A convolution operation, denoted by $f * g$, involves sliding a kernel g over an input image f, computing the element-wise multiplication, and summing the results:

$$(f * g)(i, j) = \sum_m \sum_n f(i + m, j + n) \cdot g(m, n) \tag{2.24}$$

This allows the network to detect local patterns such as edges, corners, or textures, regardless of their position in the image. This is known as shift invariance. Small input changes (like slightly translating an object) do not drastically change the convolutional output, unlike in MLPs, which treat each pixel position independently. A typical CNN consists of multiple convolutional layers followed by activation functions, pooling layers, and eventually fully connected layers for tasks such as classification or regression. Convolutional layers learn spatial features from local regions. Each layer extracts increasingly complex and abstract features as the network goes deeper [13].
Pooling layers (e.g., max pooling or average pooling) downsample the feature maps, which makes the model more robust to small shifts in the input image. Additionally, pooling expands the area of the input that later layers can consider, helping the network to understand more global patterns in the image. In convolutional layers, two common hyperparameters are stride and padding. Stride defines how far the kernel moves across the image at each step. Padding controls how the borders of the image are handled [13].

2.2.3 Graph Neural Networks

While CNNs are well-suited for data with an underlying grid structure, such as images, Graph Neural Networks (GNNs) are a class of NNs specifically designed to operate on graph-structured data. Many real-world domains, such as social networks, molecular chemistry, and recommender systems, are naturally represented as graphs. Graphs consist of a set of vertices and edges connecting them, formally defined as $G = (V, E)$, where V is the set of nodes and $E \subseteq V \times V$ is the set of edges. Each node $v \in V$ is typically associated with a feature vector $x_v \in \mathbb{R}^d$ with d channels [8].

Other architectures like MLPs and CNNs assume fixed-size and ordered input data, making them ill-suited for graphs that are variable in size and unordered. GNNs address this by learning directly from graphs through repeated message passing between connected nodes. In each layer of a GNN, node representations are updated by aggregating information from their neighbors, which allows it to reflect both how it is connected in the graph and the features of nearby nodes.

Formally, a typical GNN layer updates these features as follows:

$$m_v^{(k)} = \text{AGGREGATE}^{(k)}\Big(\Big\{x_u^{(k-1)} : u \in \mathcal{N}(v) \cup \{v\}\Big\}\Big) \tag{2.25}$$

$$x_v^{(k)} = \sigma\Big(W^{(k)} \cdot m_v^{(k)}\Big), \tag{2.26}$$

where $x_v^{(k)}$ denotes the feature vector of node v at layer k and $\mathcal{N}(v)$ is the set of neighbors of v. The function AGGREGATE combines the features of v and its neighbors (often mean, sum, or max) to produce the message $m_v^{(k)}$. This message is then transformed by a a learnable weight matrix $W^{(k)}$ and activation σ to update the node's features [12].

This way, GNNs can create node features that do not depend on the order of the nodes and can handle graphs of different sizes and shapes. Like deeper layers in CNNs learn to recognize bigger patterns in images, deeper layers in GNNs capture information from farther away in the graph. For tasks focused on individual nodes (like classification or regression), the node features from the final layer can be used directly. For tasks that need a single representation of the whole graph (like predicting the properties of a molecule), the node features are combined into one vector using a readout function [12].

Despite their effectiveness, GNNs face challenges such as over-smoothing, where after many layers, node features start to look very similar and lose their uniqueness. Methods like residual connections, attention mechanisms, and graph sampling were introduced to prevent this. Overall, GNNs provide a strong and flexible way to learn from complex data by capturing the interactions and dependencies between nodes in a graph [25].

3 Related Work

This chapter examines existing approaches for learning to solve the RC. We begin with classical search methods, which are the basis for many of these approaches. We then describe DeepCubeA and its extension Q*, followed by a self-supervised approach, and conclude with a method that almost uses pure RL.

3.1 Classical Search

Many approaches that can optimally solve the RC are rooted in classical search techniques, particularly the foundational work by Korf [20]. These methods treat the cube-solving process as a single-agent deterministic planning problem and adapt strategies originally developed for two-player games. Two popular search algorithms that are often used to solve deterministic planning problems are A*[14] and Iterative Deepening A* (IDA*)[19]. These algorithms work on problems that can be represented as a graph. The RC can be modeled as a graph, where each node represents a cube state and each edge an action that transforms one state into another. Both algorithms rely on a heuristic function $h(n)$ to estimate the cost from a given node n to the goal, guiding the search towards promising regions of the state space. In A*, the evaluation function $f(n)$ is defined as:

$$f(n) = g(n) + h(n), \tag{3.1}$$

where $g(n)$ is the exact cost from the initial state to node n, and $h(n)$ is the heuristic function. A* is guaranteed to always find the optimal solution if the heuristic function never overestimates the actual solution cost. However, a major drawback is that A* requires exponential space and time in practice, which makes it impractical for problems with large state spaces like the RC. IDA* is a modification of A* that reduces the space complexity to linear by performing a series of depth-first searches in which a branch is cut off if the cost of a frontier node exceeds a certain threshold. Despite this improvement, it still requires exponential time and, like A*, must search to the end of a path before confirming even the first move. This is a consequence of their guarantee of optimality.

A direct application of IDA* to solve the RC was presented by Korf in [21], which introduces Pattern Databasess (PDBs). Instead of manually crafting heuristic functions, PDBs precompute and store the exact cost of solving certain subproblems of the puzzle. For the RC, this involves isolating subsets of cube pieces (e.g., corners or edges) and storing the number of moves required to solve them from every possible configuration. These databases are queried

during search to provide highly accurate heuristic estimates. This significantly improves the efficiency of IDA* by reducing the number of nodes that must be expanded, making it feasible to optimally solve the $3 \times 3 \times 3$ cube. However, PDBs come with a trade-off: their creation is computationally expensive, and they require large amounts of memory, often several gigabytes for complete databases. Moreover, they are task-specific and lack generalization capabilities.
While optimality is, of course, the ideal scenario, it is not always necessary. Real Time A* (RTA*) and its variants were designed to operate under strict time constraints, inspired by decision-making in two-player games. These algorithms must commit to actions quickly, often before all the consequences of these are known. So the goal for two-player games is to make the best decision possible regarding the available compute (e.g., time). RTA* is separated into a planning phase and an execution phase. In the planning phase, the moves are simulated as though they were executed. The agent does not get any additional information. If a state appears worse than a previously visited one, the agent may backtrack, although the cost of doing so is added to the total path cost. In the execution phase, the agent performs the first move chosen during planning and then the process is repeated from the resulting state.
While classical search algorithms like A* and IDA* are powerful and guarantee optimal solutions, they are computationally expensive and rely heavily on hand-crafted heuristics or precomputed databases. Variants like RTA* trade off optimality for efficiency, but still depend on the quality of the heuristic function. These limitations motivate the use of learning-based methods that can learn heuristics or policies more efficiently.

3.2 DeepCubeA

One learning-based approach to solving the RC is DeepCubeA introduced in the paper "Solving the Rubik's cube with deep RL and search" by Agostinelli et. al. [2]. It combines deep RL with heuristic search, using a technique called Deep Approximate Value Iteration (DAVI) to train an NN to approximate a function $J(s)$ that outputs the cost to reach the goal state. The cost-to-go function is iteratively improved using the following update rule:

$$J'(s) = \min_a (g^a(s, A(s,a)) + J(A(s,a))) \tag{3.2}$$

where $A(s,a)$ is the state obtained from taking action a in state s and $g^a(s,s')$ is the cost of transitioning from s to s'. Once the cost-to-go function J is trained, it serves as the heuristic function $h(n)$ in a batch weighted variant of A*, called Batch Weighted A*:

$$f(n) = \lambda g(n) + h(n), \tag{3.3}$$

where $g(n)$ is the path cost from the starting node to node n, $h(n) = J(s_n)$ is the learned heuristic and s_n is the corresponding state to node n. The parameter $\lambda \in [0,1]$ balances the trade-off between path cost so far and heuristic estimate. A higher λ emphasizes exploration, while a

lower value prioritizes nodes closer to the goal according to the heuristic.
Due to the computational cost of using an NN for the heuristic, DeepCubeA processes multiple nodes in parallel during each search iteration. For training, data is generated by starting from the solved cube and applying a sequence of random moves in reverse to produce scrambled states with known optimal solutions. This process ensures that a valid path to the goal is available for each scrambled state. The neural network architecture includes two fully connected hidden layers followed by four residual blocks, with a single linear output unit representing the cost-to-go estimate. To know if the model actually found the shortest path, they compared their results with shortest path solvers. The authors evaluated their method on a test set of 1000 scrambled cube states, including the three states furthest away from the goal state, achieving a 100 % success rate in finding a solution and the shortest path in 60.3 % of the cases. While DeepCubeA achieves strong performance, its efficiency is limited by the cost of evaluating the heuristic function and the need to expand multiple nodes per search step. As the size of the action space increases, the number of required heuristic evaluations per step increases linearly, making the approach computationally expensive.
To address this, Agostinelli et. al. introduced Q* [1], which guides the search process with a Deep Q-Network (DQN). So, for each iteration, only one node is generated. A DQN is a NN that approximates the optimal action-value function $Q(s, a)$, which estimates the minimum cost to reach the goal from state s after taking action a. In Q* Search, the DQN is trained to predict these cost-to-go values for all possible actions from a given state. During search, instead of expanding all successors as in A*, Q* selects the node-action pair with the lowest estimated cost-to-go from a priority queue, generates its successor, and adds new node-action pairs to the queue as needed. This approach reduces the number of neural network evaluations per step to a constant, making the search more efficient, especially in environments with large action spaces.
While DeepCubeA and Q* demonstrate how learning and classical search can be effectively combined, they both rely on supervised training from precomputed trajectories and use domain-specific search infrastructure. In contrast, our approach investigates whether a policy can learn to solve the RC directly via RL, without relying on search or hand-crafted paths.

3.3 Self-Supervision

A fundamentally different approach was presented by Takano et al. in their work on self-supervised learning for solving the RC [33]. Instead of using RL or explicit search algorithms, they trained an NN to learn the inverse of a scrambling sequence. The model learns to undo the steps that scrambled the cube, effectively solving the cube by learning how the current state could have originated from the goal state. This allowed the authors to avoid many of the complexities associated with RL, such as designing suitable reward functions, dealing with sparse rewards, or managing long training horizons.

The problem is initialized in its solved state, and a series of random moves are applied to generate a scrambled configuration. The model is then trained to predict the most recent move that was applied to reach that scrambled state. This means ”unscrambling” the cube by reversing these predictions sequentially. After training, the model is used iteratively. It predicts the last move that was applied to the cube, then the inverse of that move is applied. This process is repeated on the updated cube state. Step by step, the model works backwards through the scramble, gradually returning the cube to its solved state. This approach treats solving the cube as a sequence of predictions without needing reward signals, search algorithms, or labeled data, just the structure of the cube itself.
Despite the simplicity of this approach, it achieved state-of-the-art performance, outperforming DeepCubeA. In particular, the model achieved a slightly lower average solution length and a marginally higher optimality rate while using significantly fewer training examples (2 billion vs. 10 billion). These results highlight the surprising effectiveness of self-supervised learning in structured combinatorial domains. However, the approach relies on the assumption that the model has access to the scrambling sequence during training. This assumption may not hold in more general settings. Since the model effectively learns to reverse a known process, it sidesteps the more general challenge of planning from arbitrary start states without knowing how the state was reached. In contrast, our work explores whether an agent can learn to solve the cube directly from experience without needing to reverse a predefined scrambling process or rely on handcrafted heuristics or search procedures.

3.4 Pure Reinforcement Learning

All of the previous approaches to solving the RC combine RL with other components, such as search algorithms, supervised targets, or handcrafted heuristics. While effective, these methods do not rely solely on the learning signal provided by interaction with the environment. This is somewhat surprising, given that the RC seems well-suited for RL: it has a clearly defined action space, a well-structured goal state, and a natural reward function, zero everywhere except for a reward of one when the cube is solved.
In contrast to these hybrid methods, Lin et al. [26] propose a purely RL–based approach to solving the cube. Specifically, they use policy gradient methods to train an agent directly from experience. They focused their research on the fact that DeepCubeA and the self-supervised method leveraged the fact that a cube that is only slightly scrambled is easier to solve. As they stated, they chose not to explot this because this is not how humans learn to solve the RC. Instead, their training focuses entirely on sampling states from fully scrambled cubes and choosing rewards so that this works. They wanted the rewards to be built based on underlying distance patterns. The core part of their paper is the NX module, which is trained to predict the distance between two arbitrary states of the cube. They experiment with two architectures for this module: a simple feedforward network and a more advanced attention-based model. The

latter performs better, likely because it can capture relationships between individual stickers across the start and end states. The estimated distances from the NX module are then used to shape the reward signal, providing more informative feedback to the agent during training. Using this shaped reward, they train a policy using Proximal Policy Optimization (PPO). Their experiments focus on the $2 \times 2 \times 2$ RC, where they report a success rate of 99.4 % on a set of 50,000 test cases.

Although they use pure RL, their work has some limitations. First, the approach is only tested on the simpler $2 \times 2 \times 2$ cube, which has a much smaller state space than the standard $3 \times 3 \times 3$ version. Second, they strongly emphasize avoiding states near the goal and instead focus on sampling states that are far away. However, this issue may be overstated and not as critical as they suggest. Third, while the overall framework uses RL, the NX module is trained separately in a supervised manner, making the pipeline not completely free of other techniques than RL. In our work, we explore whether a pure RL agent can learn to solve the standard $3 \times 3 \times 3$ cube without relying on a learned reward model. Instead of shaping rewards based on a separately trained network, we use a simple and fixed reward function, aiming to keep the training pipeline clean and fully RL based.

4 Methods

This chapter describes the methods we used to train agents to solve the RC using RL. We begin with a simple baseline approach and build on it by exploring different learning strategies, reward structures, and model architectures. The first section outlines the basic setup, followed by the RL techniques we experimented with. We then present the neural network architectures used to learn the policy. Finally, we describe several training strategies aimed at improving performance and generalization.
Each component is introduced in the context of the RC environment, with a focus on how the choices we made affected learning.

4.1 Base Setup

In our approach, the RC is represented as a tensor of shape $B \times 6 \times 3 \times 3 \times 6$, where B is the batch size, 6 corresponds to the number of sides of the cube, 3×3 refers to the number of facelets per side and the final dimension of six is a one-hot encoding of the facelet color. The actions applied to the cube are the standard QTM actions, as seen in section 2.1.5, with one additional idle action. The idle action leaves the cube unchanged. To generate initial states for training, we start from the solved state and apply a series of random actions. The number of applied actions is sampled uniformly between a fixed lower bound b_l and a fixed upper bound b_u. If 5 random actions have been applied to the cube, we refer to this as 5 scrambles. During the scrambling, we avoided redundant scrambles so that, for example, `R` and `R'` will not follow each other directly. This ensures that the number of scrambles is actually the number of times the cube has been scrambled, we call this true scrambles. For example, if a cube is scrambled four times but two of the moves cancel each other out, the resulting state may effectively be only two scrambles, which could allow the agent to solve it unusually quickly. There could still be shorter paths to solve a state, but this eliminates the most obvious cases.
Initially, the architecture to learn the policy was a simple MLP. It takes the representation of the RC state as input and consists of a sequence of layers, each composed of a linear transformation, followed by a Leaky ReLU activation, and layer normalization. The final output layer maps from the hidden dimension to the number of possible actions. A softmax is applied to produce a probability distribution $\pi(a_t \mid s_t)$ over actions (see figure 4.1).
We use the REINFORCE algorithm to train the policy network. The policy gradient objective

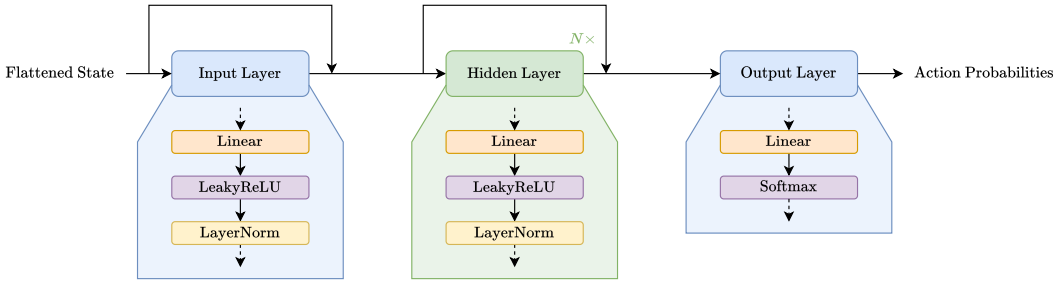

Figure 4.1: The MLP architecture used to learn the policy, consisting of an input layer for flattened cube states, two hidden layers with LeakyReLU activation and regularization components, and a softmax output layer producing action probabilities.

is implemented as a loss function:

$$L(\theta) = -\log\left(\pi(a_t \mid s_t)\right) \cdot G_t, \tag{4.1}$$

where a_t is the chosen action at timestep t and G_t is the discounted return from that timestep. This formulation allows standard gradient descent optimizers to maximize the expected return by minimizing the negative log-likelihood weighted by the returns.
To provide a more informative learning signal in the sparse reward environment of the RC, we define a dense reward based on the state *quality*. The quality q_t of a cube state at step t is defined as:

$$q_t = \frac{|\text{correct facelets}|}{|\text{overall facelets}|} \tag{4.2}$$

where a facelet is considered correct if its color matches the corresponding facelet in the goal state. Since the center facelet on each face remains fixed, it is used as a reference for face orientation and color. For the $3 \times 3 \times 3$ RC, the number of overall facelets is $6 \cdot 9 = 54$.
The reward is calculated as the geometric mean of the current quality and the improvement over the best quality seen so far:

$$r_t = \text{GM}(q_t, \max\{q_0, \ldots, q_{t-1}\} - q_t), \tag{4.3}$$

where

$$\text{GM}(x, y) = \sqrt{x \cdot y} \tag{4.4}$$

This reward was designed to address the typically sparse nature of the RC environment. The goal was to provide the agent with a denser learning signal. This allows the agent to receive feedback more frequently, as it receives rewards not only when it solves the cube, but also whenever it reaches a state of higher quality than any previously encountered state. The reward

was intended to encourage the agent to continuously improve the state of the RC. As discussed in Section 2.1.4, the issue with sparse rewards is that the agent may not see enough examples to ever receive any reward signal at all. This is particularly true for the RC, where the probability of reaching the goal state from a scrambled configuration by chance is extremely low. Even for just two scrambles, it is only 0.69 %.
We primarily experimented with scramble ranges of 1, 1–10, and 1–26. The episode length was scaled with the scramble difficulty: 1 step for 1 scramble, 25 steps for 1–10, and 50 steps for 1–26. These values were chosen to give the agent sufficient opportunity to find a solution without making episodes unnecetssarily long. The model was trained using a batch size of 32, with the number of training episodes ranging from 5000 to 100 000 depending on the number of scrambles. For evaluation, we tested on 10 000 randomly scrambled cubes, measuring the *success rate*, which is the proportion of test cases the agent solved within the episode limit. Although the model showed some learning, it did not perform well, as even for one scramble, it did not have a perfect success rate.

4.2 Reinforcement Learning Techniques

This section outlines the RL approaches used to enhance model performance, including alternative reward designs and an actor-critic architecture.

4.2.1 Reward

We begin by examining reward design, focusing on creating denser rewards and rewards that encourage generalization.

Entropy Regularization:

As discussed in Section 4.1, the choice for the reward was primarily motivated to create a dense reward. To evaluate if this is true, we logged the average (over the batch) reward and the frequency of non-zero rewards throughout the episode. We observed that while the frequency of non-zero rewards is initially high, it decreases rapidly, so in an episode that is 50 steps long, it is almost zero after only 20 steps (see figure 4.2).
An obvious choice to create a dense reward is to directly use the quality of the current state as a reward. However, we suspected this might cause the agent to focus solely on improving state quality rather than solving the cube. This could lead the agent to get stuck in local optima. Since avoiding this behavior was one of the reasons we chose the previous reward function in the first place, our expectations were not particularly high.
Another approach to encouraging dense rewards was to add a small, scaled portion of the quality improvement to our standard reward each time the current state improved upon the previous one. The idea behind this was that in the current reward setup, if the initial quality is

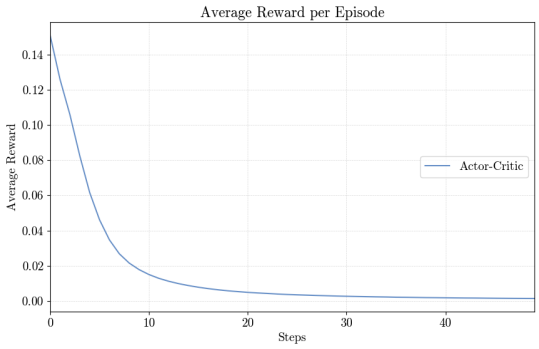

Figure 4.2: The average reward values received over the course of the episode for the standard reward.

high and the agent makes a few moves that reduce it, it will not receive any reward signal since it always performs worse than the initial state. The idea here is simply to make the reward denser. The scaling factor is introduced so that improvements over the last state matter less than improvements over the best previous state.

$$r_t = \mathrm{GM}(q_t, \max(q_{0:t-1}) - q_t) + \lambda_{\mathrm{imp}} \cdot \max(q_{t-1} - q_t, 0) \tag{4.5}$$

This approach could run into a similar issue as the quality-based reward, potentially running into local optima, but less so due to the scaling factor and the fact that we still utilize the standard reward.
As a third approach, we explored another method to make the reward denser while also encouraging the agent to stay away from local optima. This can be achieved by putting more emphasis on exploration. Combined with the observation that the model still did not have a great performance for more than three scrambles, this was the motivation to change the reward to encourage the agent to explore more. For this reason, we added entropy regularization to the reward. The updated reward then became:

$$r_t = \mathrm{GM}(q_t, \max(q_{0:t-1}) - q_t) - \lambda_{\mathrm{er}} \cdot \log p(a), \tag{4.6}$$

where λ_{er} is the entropy regularization weight. As seen in the formula, the entropy regularization term is high when the probability of the chosen action is high. So, it punishes the agent for being too confident in selecting a particular action, encouraging exploration over exploitation.

Sparse Bonus:

To encourage the agent to generalize, we first tried the usual reward for the RC environment, which is the completely sparse reward and gives a reward of one if the cube is solved and zero

otherwise.

$$r_t = \begin{cases} 1 & \text{if } q_t = 1 \\ 0 & \text{otherwise} \end{cases} \tag{4.7}$$

This is an outcome-based reward function, which is a reward that depends only on the final outcome of a task. As shown in [10], outcome-based rewards help with generalization, as they encourage the agent to learn strategies that can adapt more flexibly to task variations. That said, as mentioned in 2.1.4, sparse rewards can pose a challenge in large environments such as the RC.
To address this, we aimed to combine our standard reward and the sparse reward by always giving our reward, just when the cube is solved, we add a bonus to it as an extra incentive for the agent to try to solve the cube instead of running into local optima. So the reward is defined as:

$$r_t = \begin{cases} \mathrm{GM}(q_t, \max q_{0:t-1} - q_t) + b & \text{if } q_t = 1 \\ \mathrm{GM}(q_t, \max q_{0:t-1} - q_t) & \text{otherwise} \end{cases}, \tag{4.8}$$

where b is the bonus and should maximally be about double the usual reward values. The idea behind this approach was to combine the benefits of our standard reward with those of sparse rewards to encourage generalization.
Overall, these reward variations were designed to balance providing informative feedback during training and encouraging the agent to generalize.

4.2.2 Actor Critic

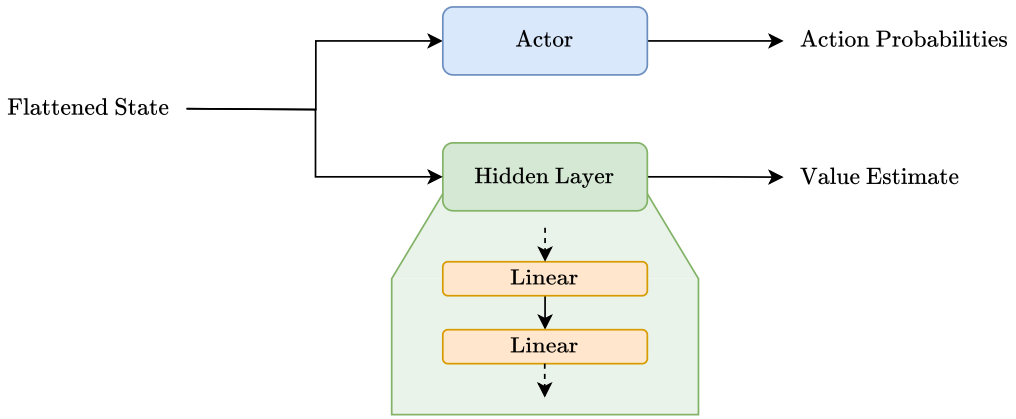

Figure 4.3: The actor-critic architecture used for policy learning, consisting of an actor network that takes flattened cube states as input and outputs action probabilities, and a critic network that estimates state values for policy optimization.

Another approach we explored was to use an actor-critic architecture to learn the policy. One motivation behind this method is that it can help the agent converge more reliably and potentially generalize better across different cube states. The actor-critic setup adds a value function that helps guide the policy during training. In the context of the RC, the critic essentially learns to estimate how ”solvable” a given cube state is. It estimates the relative difficulty of solving the cube from different states. Because the critic estimates long-term returns, it can provide a more stable and informative learning signal than relying solely on the reward signal. Ideally, the critic offers a broader perspective on the cube-solving process by learning which states are closer to or further from the goal, guiding the actor beyond the immediate reward signal. This additional feedback can make training more reliable and help the agent to converge faster.
The critic network used in our experiments is a simple two-layer feedforward model that receives a flattened representation of the cube state as input and outputs a scalar value estimate (see figure 4.3). It is trained to minimize the mean squared error between predicted values and n-step returns. The actor and critic are trained jointly, and the policy gradient is scaled by the advantage computed from these n-step returns. The update rules are as follows:

$$A_t = G_t^{(n)} - V(s_t) \tag{4.9}$$

$$\mathcal{L}_{\text{actor}} = -\log \pi(a_t|s_t) \cdot A_t \tag{4.10}$$

$$\mathcal{L}_{\text{critic}} = (A_t)^2 \tag{4.11}$$

The total loss is the weighted sum of actor and critic losses:

$$\mathcal{L} = \mathcal{L}_{\text{actor}} + \lambda_c \cdot \mathcal{L}_{\text{critic}}, \tag{4.12}$$

where λ_c is a coefficient (set to 0.5 in our experiments) that controls the contribution of the critic loss. This weighting helps balance the actor’s ability to explore with the critic’s capacity to provide useful feedback. We used n-step returns to compute the targets for the critic:

$$G_t^{(n)} = \sum_{k=0}^{n-1} \gamma^k r_{t+k} + \gamma^n V(s_{t+n}), \tag{4.13}$$

which balance bias and variance and help distribute reward information faster than full returns. Updates were applied every n steps or at the end of the episode, depending on which came first.

4.3 Neural Network Architectures

This section describes how we applied different network architectures: loop-based MLP, RNNs, CNNs, and GNNs to the RC environment.

4.3.1 Multilayer Perceptron

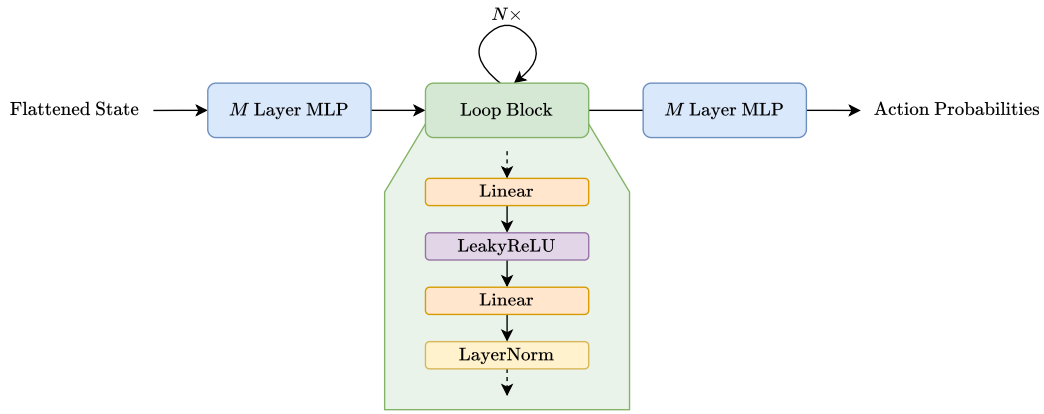

Figure 4.4: The loop-based architecture used for policy learning, consisting of an MLP block that contains linear layers, LeakyReLU activation, and layer normalization and is repeatedly applied to the flattened input and additional MLPs.

One problem we observed with the MLP was that it seemed to be memorizing solutions instead of generalizing across states. An idea to address this is to increase the depth of the model. However, this also increases parameter size and may result in the model memorizing even more, as models with more parameters are more likely to overfit [11]. Instead, we explored a model where a fixed MLP block is repeatedly applied to one state. This mimics the way RNNs work but without the sequence handling or explicit hidden states. Repeating the same block may encourage the model to learn patterns that work across different states, rather than fitting to specific examples. This could lead the model to discover general rules for improving the cube, regardless of the exact configuration.
The core component of the model is a single MLP block, composed of a linear transformation, layer normalization, and LeakyReLU activations. This block is applied iteratively several times to the same cube representation, with all iterations sharing the same weights (see figure 4.4). The number of times the ”loop” is executed is controlled by the hyperparameter n_{loop}. Because the weights are shared, this approach introduces no new parameters beyond the core block, so it is more parameter-efficient than simply stacking additional layers. The repeated application of the same transformation may also act as a form of implicit regularization.

4.3.2 Recurrent Neural Network

The motivation behind using an RNN is that, unlike feedforward networks, RNNs process data sequentially. This allows the policy to condition its decisions not only on the current cube state but also on the sequence of previous states and actions. This way, the policy does not have to decide on each action independent from the previous steps. Using an RNN is also motivated by how humans solve the RC, as they do not only consider the cube as they see it now, but also

remember previous moves they made.
One hope is that the agent will learn to avoid repeating the same moves or getting stuck in loops because it now has the history of previous states available. To this end, we inserted a GRU in between smaller MLPs in our architecture (see figure 4.5) to capture the dependencies between individual time steps. The hidden state was initialized with zeroes, and we experimented with different depths for the RNN.
Since unstable gradients are a common issue for RNNs [28], we used gradient clipping with a

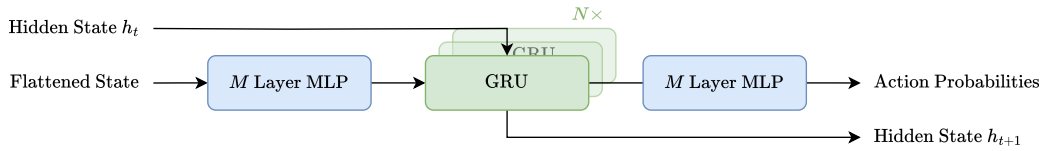

Figure 4.5: The GRU-based architecture used to learn the policy, consisting of MLPs and a GRU layer for temporal processing of cube states to produce action probabilities.

threshold of 1.0. We also logged the gradient norms during training to validate that they were neither exploding nor vanishing. Additionally, we experimented with substituting the GRU with an LSTM since these are a more advanced version of GRUs and may be able to capture the long range dependencies better.

4.3.3 Convolutional Neural Network

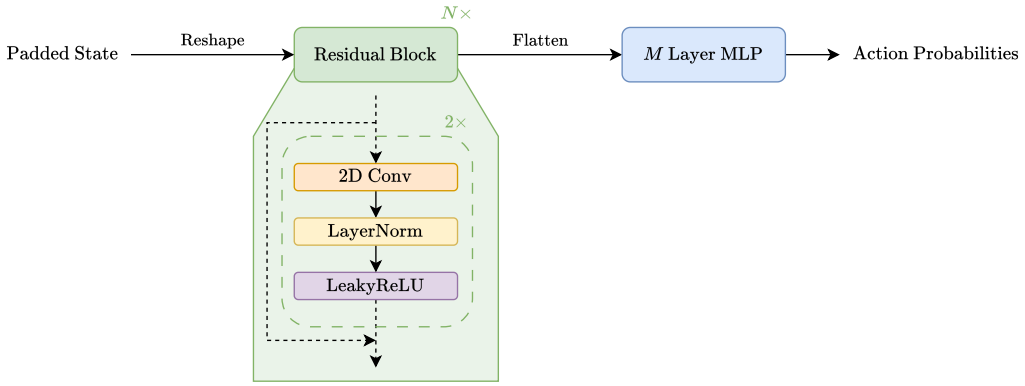

Figure 4.6: The CNN architecture used for policy learning, consisting of a ŗ with two residual blocks processing padded cube states, followed by flattening and an MLP that outputs action probabilities.

Neither MLPs nor RNNs make use of spatial structure by default. Instead, they have to learn spatial relationships explicitly from the data. As a result, the cube state has to be flattened into a vector, which discards important spatial information, such as the relationships between

neighboring facelets. This loss of structure can make it more difficult for the model to learn meaningful representations. CNNs, on the other hand, are specifically designed to preserve this spatial information, allowing them to take local dependencies into account.
To take advantage of this, we experimented with a CNN-based policy model. We used a Residual Neural Network (ResNet) architecture [15] consisting of two residual blocks with a kernel size of 3, followed by two fully connected layers (see figure 4.6). To make the cube representation suitable for convolutional processing, we added padding around each face of the cube. For each face, we included the top, bottom, left, and right rows of its adjacent neighbors, as can bee seen in figure 4.7. Additionally, we concatenated the one-hot color encoding (6 dimensions)

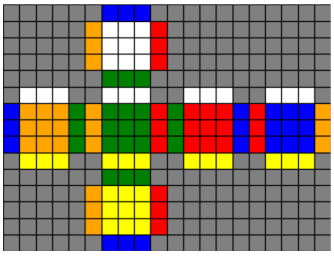

Figure 4.7: An example padded state, used as input for the CNN.

with a one-hot vector indicating the current face (also 6 dimensions). This helps provide global context to each face, since otherwise the network processes each face largely independently and would lack information about the cube's overall structure. We added one final dimension to the vector to indicate grid cells that correspond to the corners of the cube (so they do not have a corresponding facelet in the actual cube). This resulted in a feature vector of size 13 for each facelet: 6 for the color, 6 for the face identity, and 1 to indicate whether it is a corner. With the added padding, the overall input shape changed from $(B \times 6 \times 3 \times 3 \times 6)$ to $(B \times 6 \times 5 \times 5 \times 13)$. This setup let the network consider each face of the cube in its local context and allowed the convolutional layers to recognize spatial patterns between adjacent facelets.
Since CNNs process data locally, the model does not directly produce a single global action distribution. Instead, it makes a local prediction for each face: a probability distribution over rotating that face left or right. Afterwards, these local distributions are combined into a single global action distribution over all possible face turns, from which the final action is drawn. This design reflects the fact that the model considers each face separately, while still enabling global decision-making across the cube.

4.3.4 Graph Neural Networks

We also experimented with using a GNN to learn the policy. To represent the RC state as input to the network, we encoded it as a graph. Each facelet on the cube is represented by a node, resulting in 54 facelet nodes. We added 6 color nodes (one for each color) and 6 side

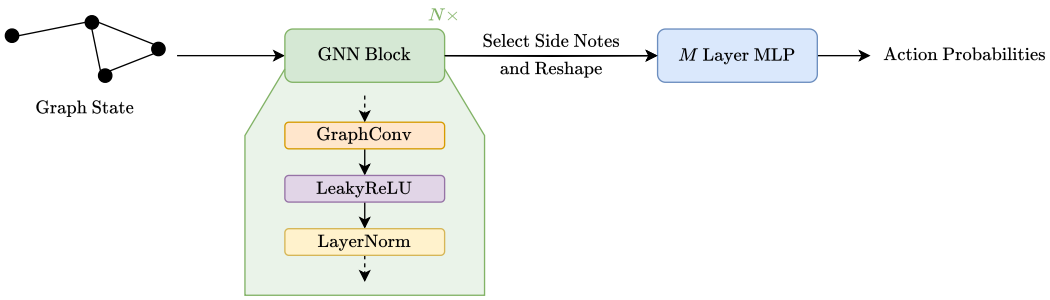

Figure 4.8: The GNN architecture used for policy learning, consisting of a GNN block that processes graph-structured cube states through GraphConv layers, LeakyReLU activation, and layer normalization, followed by side node selection and reshaping before an MLP outputs action probabilities.

nodes (one for each face of the cube), leading to a total of 66 nodes in the graph. Edges were added such that each facelet node was connected to two additional nodes: one representing its current color and one representing the face it is located on (see figure 4.9). This graph structure enables the GNN to aggregate both spatial information (via sides) and semantic information (via colors), capturing the relevant relational information of the cube's configuration. Since the model receives the cube as a graph, it can more directly learn its underlying structure. The hope was that by understanding these structural relations, the network would be better able to generalize across cube states instead of memorizing specific configurations.

For the architecture, we used a Graph Convolutional Network (GCN)-based model [18] with

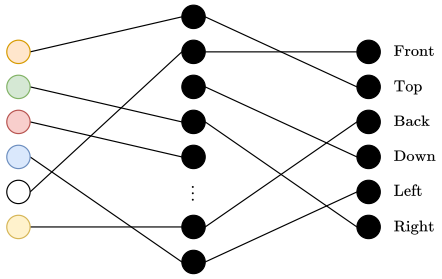

Figure 4.9: An abbreviated example graph to represent a RC state.

three stacked graph convolution layers. Each layer was followed by a LeakyReLU activation, and layer normalization. After the GCN layers, the node embeddings were reshaped, and we selected the six side nodes for further processing. These were passed through a sequence of fully connected layers, followed by a final linear layer to produce logits for the action probabilities. The network returns a probability distribution over the possible actions (see figure 4.8). Because we only select the side nodes, each face is processed individually, similar to the CNN approach. The model then collects these outputs to form a single global distribution, from

which the final action is sampled.
While the connectivity in our proposed graph is well-defined, there is no obvious definition for the initial node features. That is why we tried different alternatives for the node features. One approach is to assign each node a unique index. Another is to use a one-hot encoding based on node type: color nodes were encoded as $(1, 0, 0)$, side nodes as $(0, 1, 0)$, and facelet nodes as $(0, 0, 1)$. We also tried to give color and side nodes their usual one-hot encodings and assign facelet nodes the one-hot encoding of the color they currently hold. Finally, we experimented with learned embeddings, where node features were directly mapped to the hidden size through a trainable embedding layer.
We hypothesized that this graph representation would help the agent generalize better by capturing the underlying structure of the cube rather than just memorizing specific configurations. By learning how facelets relate to sides and colors in a more abstract way, the network might be more prepared to transfer knowledge across different cube states.

4.4 Training Strategies

In addition to experimenting with different RL techniques and model architectures, we also conducted some specific to the training process. These experiments were primarily applied to the base MLP model.
One such strategy was *curriculum learning*, where the idea is to increase task difficulty gradually [5]. So, for the first episodes, the model is trained on easier instances of the problem, for the RC, this would mean cubes that are scrambled only a few times. The difficulty then gradually increases over the course of the training so that the model is trained on the full problem in the last episodes. This approach allows the model to first learn simple patterns before tackling the full complexity of the task. We focused on learning up to 10 scrambles, since the model cannot consistently solve longer scrambles, making it difficult to evaluate performance on the upper end of the scramble range for e.g. 1-26 scrambles.

- **Curriculum Slow:** We began with scrambles ranging from 1 to 3 moves. This range was incremented in seven stages, increasing by one scramble each time until the final stage included scrambles between 8 and 10 moves. Each stage was trained for 1.5× the number of episodes used in the previous stage.
- **Curriculum Fast:** A second setup started with scrambles from 1 to 2 moves, and this range increased in four stages by two moves at each step, eventually reaching 9 to 10 scrambles. The episode count again increased by a factor of 1.5× between stages.

Something else we tried to encourage generalization over memorization was *goal conditioning*. In this setting, we generated random goal states and paired them with distinct initial states, then concatenated the two as input to the model. The idea was to teach the agent to reach a specified goal from a given start state rather than always solving toward a fixed solved state [3]. Always

solving toward different goals may help the model to implicitly learn the cube's structure and dynamics, making it easier to navigate the state space.
To test whether this setup worked, we ran an experiment using only a single scramble, meaning the agent needed to learn two specific moves to reach the goal. Despite the simplicity, this seemed to be too complex for our current MLP, as it performed no better than random guessing.
To rule out implementation issues, we ran a control experiment where the goal state was always the solved cube (scrambled zero times), and the start state was scrambled once. In this case, the model quickly learned to solve the task as expected. This confirmed that the goal conditioning setup worked correctly in terms of implementation and that the failure on two step cases was likely due to the increased complexity of the learning task rather than a bug.
Since even these lightly scrambled instances could not be solved within 10 000 episodes, scaling this approach to full scrambles (up to 26 moves) did not seem feasible, and the idea was abandoned.

5 Evaluation

In this chapter, we evaluate the models and training setups that were previously described. We start with the MLP baseline, looking at how different architecture choices and hyperparameters affect performance. We then compare it to other architectures like RNNs, CNNs, and GNNs. Then the chapter focuses on RL techniques, reward design, and training strategies.
Unless otherwise specified, we train our model for a total of 50 000 episodes, each with a maximum length of 50 steps, and evaluate our success rate on 10 000 scrambled cubes. Our standard scramble range is 1-26 scrambles. All presented scores are averaged over four runs, with both the mean and standard deviation reported.

5.1 Baseline Optimization

Our experiments with the Multi-Layer Perceptron (MLP) architecture served as a foundation for exploring how different hyperparameters influence learning performance on the RC task. We began by tuning the number of layers and the size of the hidden layers under the assumption that increasing the number of parameters will lead to an improved success rate, as the model will have more capacity for memorizing. As shown in Table 5.1, deeper and wider

Experiment	Value	Success Rate
Number of Layers	**10**	**0.1344 ± 0.0087**
	7	0.1221 ± 0.0027
	5	0.1227 ± 0.0025
	3	0.1252 ± 0.0085
Hidden Size	**256**	**0.1344 ± 0.0087**
	128	0.1196 ± 0.0043
	64	0.1128 ± 0.0051

Table 5.1: Comparison between different model capacities in terms of the network depth and hidden size with respect to their success rate.

networks generally led to slightly better performance. The best results were achieved with 10 layers and a hidden size of 256. Although performance improved with larger models, the gains were relatively small. A possible explanation for this is supported by the fact that the model performs best for 1-3 scrambles and not as well for larger scrambles. This is likely due to the large increase in the number of possible cube states as the scramble depth increases. For example, with 1 scramble, there are 12 states; with 2 scrambles, there are 144; and with 3, there are 1728 (see figure 5.1). Thus, solving cubes scrambled with 4 or more moves requires the model to generalize across exponentially many configurations. This severely limits the

effectiveness of simply scaling up model size.

To address overfitting and promote generalization, we evaluated two standard regularization

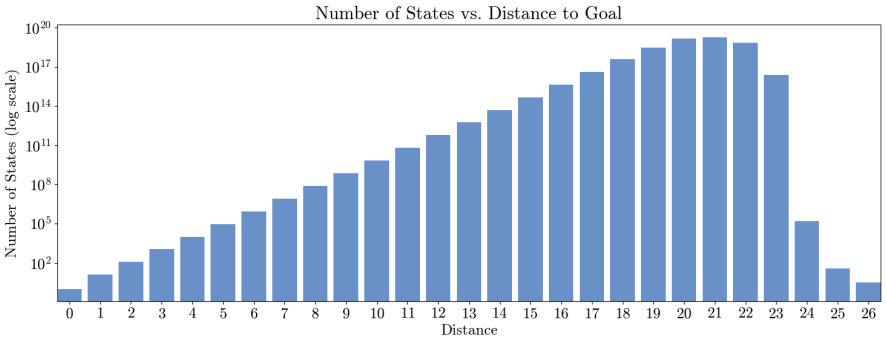

Figure 5.1: Plot of number of states vs. distance (log-scaled) [29].

techniques: *dropout* and *weight decay*. Dropout works by randomly deactivating a set percentage of the neurons during each training step. This forces the network to learn more robust features that do not rely too heavily on any single neuron, thereby reducing overfitting [31]. Another strategy we explored was increasing the *weight decay*, a regularization method that penalizes large weight values by adding a term to the loss function proportional to the magnitude of the weights. This technique encourages the model to favor simpler, more generalizable representations and has been shown to improve generalization by reducing overfitting to the training data [22]. However, in our experiments, increasing the dropout rate slightly worsened performance, so we ultimately decided not to use it. This may be due to the inherent training instability in RL, where the learning signal is already noisy and variable. The added randomness from dropout may have further disrupted the learning process, outweighing any potential benefits to generalization. Adjusting the weight decay did not have much impact on the performance, which may be due to the fact that the model's weights did not grow excessively during training, reducing the influence of the regularization term.

Experiment	Value	Success Rate
Dropout	**0**	**0.1344 ± 0.0087**
	0.25	0.1244 ± 0.0051
	0.5	0.1139 ± 0.0062
	0.75	0.1202 ± 0.0069
Weight Decay	1e-2	0.1271 ± 0.0074
	1e-3	**0.1344 ± 0.0087**
	1e-4	0.1303 ± 0.0057

Table 5.2: Comparison of dropout and weight decay settings regarding their effect on success rate.

Next, we varied the learning rate. As shown in Table 5.3, performance improved as the learning rate increased up to $5 \cdot 10^{-5}$, after which it began to degrade slightly.

During earlier experiments, we observed that the model struggled particularly with solving

Experiment	Value	Success Rate
Learning Rate	1e-6	0.1150 ± 0.0063
	5e-6	0.1226 ± 0.0025
	1e-5	0.1344 ± 0.0087
	3e-5	0.1511 ± 0.0119
	5e-5	$\mathbf{0.1617 \pm 0.0104}$
	7e-5	0.1507 ± 0.0059
	1e-4	0.1471 ± 0.0054

Table 5.3: Comparison between different learning rate values regarding their effect on success rate.

cubes that had been scrambled often. This motivated us to experiment with the discount factor, which controls how much future rewards are valued relative to immediate ones, as more complex states may get rewards later in the episode than shorter scrambles. Due to our short episode lengths (typically 50 steps), we tested relatively low discount values ranging from 0.65 to 0.85. The results suggest that, in general, larger discount factors (e.g., 0.75–0.85) lead to a

Experiment	Value	Success Rate
Discount Factor	0.65	0.1260 ± 0.0075
	0.7	0.1210 ± 0.0095
	0.75	$\mathbf{0.1344 \pm 0.0087}$
	0.8	0.1319 ± 0.0052
	0.85	0.1321 ± 0.0029

Table 5.4: Comparison between different discount factors regarding their effect on success rate.

slightly improved performance. One possible reason is that this slightly favors scrambles that are further from the solution, while the model already performs relatively well on those that are closer.

Combining the best-performing hyperparameters from all previous experiments, we trained our final MLP model. While the MLP could learn to reliably solve simpler scrambles, its

Experiment	Value	Success Rate
Best MLP Model	Num Layers=10 Hidden Size=256 Dropout=0 LR=5e-5 Discount=0.75 Weight Decay=1e-3	$\mathbf{0.1617 \pm 0.0104}$

Table 5.5: Performance of the most successful MLP configuration found so far.

performance quickly declined for more complex states. This suggests that the architecture struggles to generalize and could benefit from a more expressive model.

5.2 Architectures

This section examines how different neural network architectures affect the agent's ability to solve the RC. Starting from a simple loop-based MLP, we test recurrent, convolutional, and graph-based models.

5.2.1 Loop-Based

In this section, we evaluate the performance of the loop-based architecture introduced in Section 4.3.1, where a single MLP block with shared weights is applied repeatedly to the same input state. This approach aims to improve parameter efficiency and potentially encourage better generalization by implicitly regularizing the network through weight sharing.
We observed that the loop-based MLP took noticeably longer to converge compared to the

Experiment	Value	Success Rate
Loops	10	0.1158 ± 0.0025
	7	0.1183 ± 0.0018
	5	$\mathbf{0.1219 \pm 0.0023}$
	3	0.1217 ± 0.0025

Table 5.6: Comparison of success rates for different loop counts in the repeated MLP block.

standard MLP, and performance was slightly worse. One reason could be that sharing weights across multiple iterations results in fewer parameters, which reduces the model's capacity to memorize. We also observed that increasing the number of loops tended to decrease performance, suggesting that the loop structure may actually hinder learning. One possible explanation is that more loops lead to larger or more unstable gradients. Although we applied gradient clipping to counter this, the training process still appeared less stable with higher loop counts.
Overall, while the loop approach offers a parameter-efficient alternative, it does not outperform the standard MLP. The results suggest that while looping hinders memorization, this is not enough to improve generalization, ultimately leading to worse performance. Improving training stability and combining loops with other techniques might help boost its performance.

5.2.2 Recurrent Neural Networks

Now, we examine the effects of integrating a recurrent component into the policy network, as described in Section 4.3.2. The main motivation behind this architecture was to give the agent access to temporal context, potentially allowing it to avoid repeating previous actions and escape local loops more effectively. The experiments showed that adding recurrence did not improve performance. The best result was achieved using a single GRU layer, but even that performed worse than the baseline MLP. We attribute this slight decrease in performance to the fact that RNNs can sometimes hinder the memorization process by introducing variability.

Experiment	Value	Success Rate
Number of GRU Layers	7	0.1074 ± 0.0021
	5	0.1161 ± 0.0031
	3	0.1286 ± 0.0077
	1	$\mathbf{0.1391 \pm 0.0065}$
LSTM	1	0.1191 ± 0.0038
MLP		$\mathbf{0.1617 \pm 0.0104}$

Table 5.7: Comparison of success rates for different RNN architectures and layer depths.

Unlike MLPs, which always receive the same input for the same cube state, RNNs take in addition to the RC state, also the hidden state as input. This hidden state changes over time, so especially for the same cube state the hidden state can be different in another iteration. This results in more variety, which makes it harder for the agent to memorize. Nevertheless, this is also not enough for the agent to generalize to states further away, as can be seen by the decrease in success rate when increasing the recurrence (number of GRU layers). The performance also declined as more recurrent layers were added. This supports the idea that increased recurrence may have made training more difficult. While we applied gradient clipping to control unstable gradients, the added recurrence might still have introduced noise or instability into the training process. Substituting the GRU with an LSTM led to even worse results, possibly due to the increased complexity and stronger recurrence introduced by the additional gating mechanisms. Despite our hypotheses, the agent's behavior did not significantly change compared to the MLP. In cases where it got stuck in loops, the agent sometimes backtracked one or two steps but often returned to the previous position, likely because it did not receive any reward signal while doing so.
In summary, while the RNN-based models introduce temporal context, they did not lead to meaningful gains in performance and may have even hindered learning by disrupting the memorization process. However, similar to what we observed in the loop results, this was not enough to encourage more generalization.

5.2.3 Convolutional Neural Networks

We also experimented with a convolutional architecture, as described in Section 4.3.3. Our hypothesis was that the CNN would better leverage the spatial structure of the cube by recognizing local patterns between facelets. Additionally, since the network makes local per side decisions, this approach was different compared to the MLP and RNN models, and we were curious whether this could help with generalization. The CNN architecture, however,

Experiment	Success Rate
CNN	0.1219 ± 0.0033

Table 5.8: Success rate of the CNN-based policy model.

resulted in even more of a drop in performance than the GRUs. This could be due to the local per side decision making process not capturing global information of the RC. Since actions applied to the cube result in changes to almost all sides and not only the side that was turned, this might miss important interactions between different parts of the cube. Another possible factor is that the padding and the larger one hot vectors add complexity, which may hinder the memorization process.
In summary, although the CNN model introduced useful spatial biases, these alone were not sufficient to improve performance. The lack of global context and increased input complexity outweighed the advantages of local decision-making.

5.2.4 Graph Neural Networks

To evaluate the potential of graph-based representations for the RC, we experimented with a GNN architecture as described in Section 4.3.4. In this setup, we investigated two aspects: the effect of different initial node features and the impact of the number of graph convolution layers. All experiments were conducted on cubes scrambled with a single move, using 10 000 training episodes where each episode consisted of just one step.

The choice of initial node features turned out to be important for the performance. For

Experiment	**Value**	**Success Rate**	**Similarity** (during Training)
Initial Features	numbered vectors	0.0862 ± 0.0094	0.9999 ± 0
	one-hot category	0.0852 ± 0.0022	1 ± 0
	initial one-hot	0.4222 ± 0.0015	0.8882 ± 0.0051
	embedding	$\mathbf{0.5824 \pm 0.0030}$	$\mathbf{0.9347 \pm 0.0026}$
Number of graph convolutions	1	0.0836 ± 0.0014	1 ± 0
	2	0.5812 ± 0.0011	0.9567 ± 0.0013
	3	$\mathbf{0.5824 \pm 0.0030}$	$\mathbf{0.9347 \pm 0.0026}$

Table 5.9: Comparison of GNN training outcomes based on initial node features and number of convolution layers.

both naive approaches, assigning unique indices (numbered vectors) and one-hot encodings by node type (facelet, side, or color), the success rates remained close to random guessing (approximately 8.5 %). This suggests that these features were not expressive enough to allow the GNN to learn any useful structure. In contrast, one-hot encodings based on actual facelet colors and learned embeddings allowed the model to achieve success rates of 42.2 % and 58.2 %, respectively. These results demonstrate that when the input features reflect semantically meaningful aspects of the cube, the GNN can partially learn to solve the task, at least for a single scramble. However, this still significantly lagged behind other architectures that were able to consistently solve cubes scrambled up to three times.
To understand the model's behavior in more detail, we analyzed the output logits and found that, during training, the predicted probabilities for rotating each side clockwise or counterclockwise were nearly identical for large parts of the training. This led us to investigate whether

oversmoothing was occurring. We measured the cosine similarity of the side node embeddings after the final GCN layer and found that they were indeed very similar, with values of about 90 %. This indicates a strong degree of oversmoothing, which limits the GNN's ability to make distinct decisions based on nuanced differences in cube configurations. Oversmoothing creates a challenge for the final MLP at the end of our architecture. Since the inputs to the MLP (the side node embeddings) are nearly identical, the network is left with two options: either amplify minor differences via large weight magnitudes, potentially leading to instability and large gradients or predict the same actions. In our experiments, the latter case applied, as the gradients remained stable, but the output logits showed little variation.
We hypothesized that reducing the number of graph convolution layers would reduce oversmoothing. However, using fewer layers increased the cosine similarity. This indicates that the reduced depth prevented the network from learning meaningful distinctions between the faces of the cube, resulting in more similar embeddings. Since the six side nodes are structurally equivalent, fewer layers failed to capture the distinctions necessary for action selection, making the task for the subsequent MLP more difficult. We also observed that with only a single convolution, the success rate dropped back to about random levels. This suggests that connectivity alone is insufficient for the model to learn effectively. Since action decisions are made based on the side nodes, these nodes must receive information from the color nodes. In the case of only one graph convolution, this information does not yet propagate to the side nodes. Only with at least two convolutional layers meaningful information flow occurs, enabling the model to start learning.
Another potential issue with the graph encoding is that the GNN may struggle to differentiate between center facelets and others, making it unclear which facelets can actually be rotated. Including this distinction in the initial node features could address this limitation.
These findings suggest that the GNN struggles to gain an understanding of the RC. One limitation is that, due to oversmoothing, the model is unable to effectively aggregate and distinguish information across the entire cube. As a result, the side nodes (where the decisions are made in our architecture) end up with representations that are too similar to allow for robust policy learning.

5.3 Reinforcement Learning Techniques

This section evaluates the impact of specific reward design choices and compares actor-critic architectures against standard REINFORCE.

5.3.1 Reward

To address the challenges identified in Section 4.2.1, we investigated several reward function variations. We had two goals: first, to make the reward signal denser, and second, to encourage the agent to generalize beyond memorizing short solution paths. During our experiments, we

also observed specific agent behaviors that further supported our hypothesis that our standard reward function might lead to local optima.

One example of such behavior came from comparing two evaluation trajectories, one starting

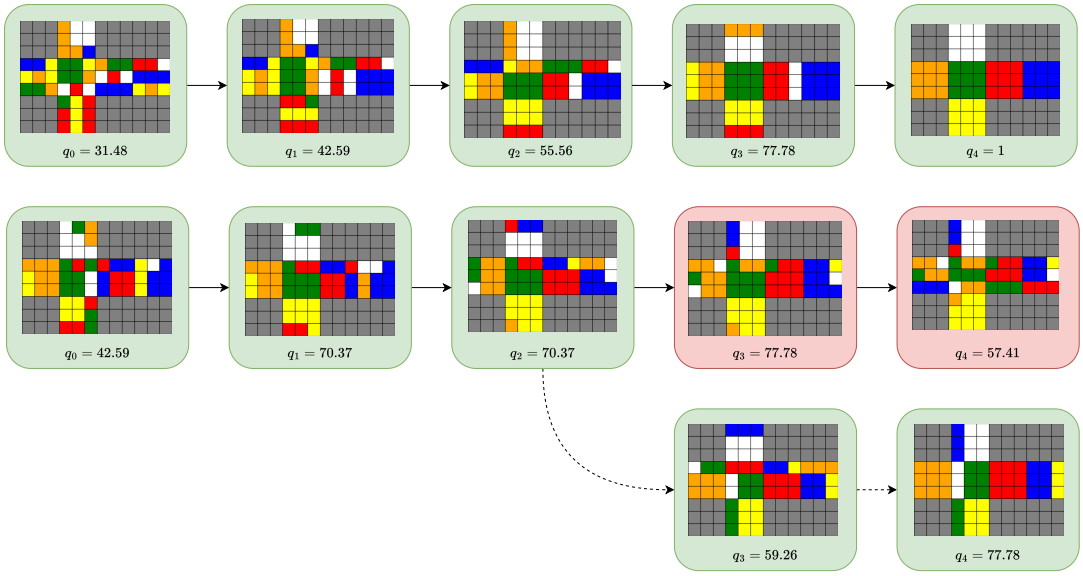

Figure 5.2: The 4-scramble trajectory (top) and the 5-scramble trajectory (middle) plus the states that would have been correct for the 5-scramble trajectory (bottom).

from a state scrambled four times and the other from a state scrambled five times. We refer to these as the *4-scramble trajectory* and *5-scramble trajectory*, respectively (see figure 5.2). In the 5-scramble trajectory, the agent initially performed well. The first two actions increased or maintained the cube's state quality q, bringing the cube to a state with a distance of three moves to the goal state. However, in the third step, it chose an action that led to a slight increase in quality rather than selecting the correct move, which, although momentarily decreasing the quality, would have brought the cube closer to being solved. From there, the agent continued to move further away from the solution, eventually ending the episode far from the goal state. However, the 4 scramble trajectory solves the initial state correctly and efficiently in 4 steps, where the quality of the state consistently increases with each step. This behavior suggests that the reward design may place too much emphasis on increasing quality rather than solving the cube, making the agent vulnerable to getting stuck in states with higher quality.

Another interesting behavior we saw was when the agent reached a state with a very high quality ($q = 0.8148$), which can be seen in figure 5.3. This state, however, was actually 17 moves away from the goal state. When the agent applies an action to this state, the quality will drop. So, under our standard reward setup, it then receives no reward for several steps, as it must pass through many intermediate states before reaching one with higher quality. Without any intermediate reward signals, it is unlikely that the agent will ever reach such a state. This

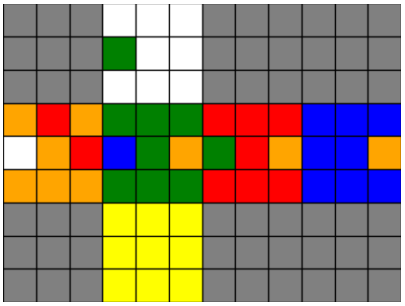

Figure 5.3: A state with $q = 0.8148$ and distance to the goal of 17 moves.

also further motivated our exploration of alternative reward strategies that could counter/avoid such local optima and guide the agent more effectively in these situations. So we experimented

Experiment	Value	Success Rate
Standard Reward (best MLP model)		0.1617 ± 0.0104
Quality Reward		0.1073 ± 0.0075
Improvement over last	1e-1	0.1604 ± 0.0070
	1e-2	0.1663 ± 0.0105
	1e-3	0.1583 ± 0.0130
Entropy Regularization	1e-2	0.1604 ± 0.0031
	1e-3	0.1761 ± 0.0046
	1e-4	0.1529 ± 0.0054
Sparse Reward		0.1035 ± 0.0075
Bonus Reward	0.0 entropy	0.1945 ± 0.0071
	1e-3 entropy	$\mathbf{0.2048 \pm 0.0044}$

Table 5.10: Comparison between different reward functions regarding their success rates, using the bonus reward with entropy regularization.

with a variety of reward designs, as summarized in Table 5.10.

Quality reward: Directly using the cube state quality as the reward led to poor results. The agent quickly ran into local optima, often preferring to remain in states that had a high quality rather than exploring paths that temporarily decreased quality but ultimately led to a solution.

Improvement over last reward: This variation added a small bonus whenever the current state improved upon the previous one. While it made the reward signal denser, the success rate increased only slightly. The likely reason is that this reward also encourages local improvements, which again reinforces behavior that avoids exploratory steps that may initially reduce quality but are necessary to solve the cube.

Entropy regularization: Adding an entropy term to the reward helped promote exploration by penalizing overconfident action choices. With proper tuning, this led to more robust policies, improving success rates up to 17.61 % with a regularization weight of 10^{-3}. However, even with this improvement, the agent's generalization remained limited, as performance dropped significantly on more difficult scrambles, indicating that the agent still can not generalize to these.

Sparse reward: As expected, the sparse reward performed poorly. The agent rarely encountered successful examples during training, which severely limited its learning progress.
Bonus reward: This approach merged our standard reward with an additional bonus upon solving the cube. It improved performance further, likely by giving the agent more explicit motivation to reach the goal state, instead of purely improving the quality. While this bonus helped avoid local optima to some extent, it still was not sufficient for the agent to fully generalize. Nevertheless, through this bonus, the structure of the problem changes. With the bonus

Experiment	Value	Success Rate
Discount Factor	0.75	0.2048 ± 0.0044
	0.8	0.2091 ± 0.0057
	0.85	0.2150 ± 0.0031
	0.9	$\mathbf{0.2187 \pm 0.0034}$

Table 5.11: Comparison between different discount factors regarding their effect on success rate, using the bonus reward with entropy regularization.

reward, longer scrambles can achieve higher returns only later in the episode compared to shorter scrambles, since they receive the bonus only when the cube is actually solved, which naturally takes longer for longer scrambles. Previously, for values higher than $\gamma = 0.75$, the success rate began to decrease again. Now, however, we can try higher discount values, as they place more emphasis on rewards that come later in the episode. The results in Table 5.11 support this: where success rates previously declined for higher values, they now slightly increase.
In summary, our experiments confirmed that designing a well-balanced reward function is important in environments like the RC. Dense rewards help the agent to consistently receive feedback during training, but they can not generalize beyond simple cube states. Sparse rewards, on the other hand, encourage generalization but suffer from insufficient feedback. The most promising results came from combining the two approaches by using the bonus reward with entropy regularization. However, the agent still struggled with more complex cube states.

5.3.2 Actor Critic

As architectural changes alone did not yield improvements, we next experimented with an actor-critic setup as described in Section 4.2.2, motivated by its potential to stabilize training and improve convergence through the use of a learned value function. In practice, the actor-critic architecture led to slightly smoother and more stable training than standard REINFORCE. Particularly in the early stages, we observed more consistent improvements in the policy, likely due to the critic offering more informative gradients. When comparing behaviors, the actor in this setup became more confident more quickly, action probabilities for the chosen moves achieved high values earlier and stayed stable throughout training, as can be seen in figure 5.4. However, the performance improved only slightly. The agent still struggled to generalize

Experiment	Value	Success Rate
Number of Steps	15	0.2034 ± 0.0027
	10	$\mathbf{0.2066 \pm 0.0051}$
	7	0.1960 ± 0.0030
	5	0.1899 ± 0.0036
	3	0.1734 ± 0.0011

Table 5.12: Comparison between different numbers of steps in actor-critic training with respect to their success rates (using the bonus reward with entropy regularization and a discount of 0.75).

beyond short scramble lengths. For longer scrambles, performance dropped again, indicating that the critic's guidance was not enough to help the agent to improve generalization.

One potential reason for the limited impact of the critic is the high-dimensional nature of the

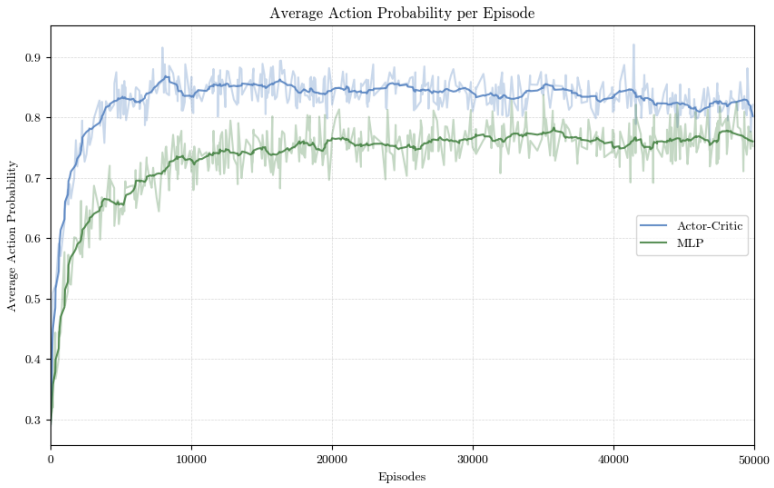

Figure 5.4: Comparison of average action probabilities per episode, using a critic and without one.

RC environment, which makes it hard to estimate long-term values reliably. When the critic fails to generalize and instead learns faulty estimates, it can mislead the actor by reinforcing suboptimal behavior. This was especially noticeable when both networks settled into local optima, where the critic overestimated the value of familiar but ineffective states. To explore this further, we visualized the critic's predictions across episodes, which showed that the patterns were inconsistent. Often, the critic produced noisy outputs, failing to capture the long-term structure needed to guide learning effectively.

In conclusion, while the Actor-Critic method offered smoother training and reduced variance, it still suffered from an inability to generalize to more complex states. More expressive value functions, improved exploration strategies or richer critic architectures may be necessary to make further progress.

5.4 Training Strategies

In this section, we evaluate how different training strategies impact performance, including curriculum learning and the range of scrambles used during training.

5.4.1 Curriculum Learning

To investigate whether gradually increasing task difficulty could improve learning outcomes, we experimented with two forms of curriculum learning, as described in Section 4.4: *Curriculum Slow* and *Curriculum Fast*. The goal was to guide the agent through progressively harder instances of the RC task, beginning with short scrambles and ending with more complex states. Both curriculum schedules, as well as the model that did not use curriculum learning, were trained for 20 000 episodes. The model that did not use curriculum learning saw states that were scrambled between 1-10 times during training. As shown in Table 5.13, curriculum

Experiment	Value	Success Rate
9-10	No	0.0087 ± 0.0014
	Curriculum Slow	$\mathbf{0.0088 \pm 0.0011}$
8-10	**No**	$\mathbf{0.0056 \pm 0.0007}$
	Curriculum Fast	0.0049 ± 0.0004

Table 5.13: Comparison of success rates with and without curriculum learning. The search is performed on the scramble ranges 8-10 and 9-10.

learning did not significantly improve success rates. This outcome was consistent across both curriculum strategies, with no meaningful difference between the slower and faster schedules. There are a few possible explanations for why curriculum learning was ineffective in our setup. First, it is possible that the training duration within each stage was not long enough for the agent to fully master the current difficulty level before moving on to the next. In such a case, the curriculum would effectively just present a sequence of increasingly difficult tasks that are all partially learned. This might even result in interference, where what the policy learned for easier tasks is overwritten or contradicted by the partial learning on harder ones. Second, even with more time for the single stages, the complexity of the cube states in the 8–10 or 9–10 scramble ranges may have remained too high for the model to learn meaningful policies.

5.4.2 Number of scrambles

We also investigated how the choice of scramble range during training affects the model's ability to generalize to different levels of difficulty. Since the base MLP model consistently failed to solve cube states scrambled more than 13 times, we, like in the curriculum learning experiments, chose not to explore wider scramble ranges such as 13–26 or 1–26. Instead, we focused on shorter scrambles, where the model still showed potential to learn meaningful strategies. All models were trained for 20 000 episodes. As shown in Table 5.14, the training

Experiment	Training Range	Success Rate
4-10 Scrambles	**1-10**	**0.1235 ± 0.0083**
	4-10	0.1219 ± 0.0127
7-10 Scrambles	1-10	0.0158 ± 0.0011
	4-10	**0.0159 ± 0.0006**
	7-10	0.0109 ± 0.0013

Table 5.14: Comparison between different scramble ranges during training regarding their impact on success rate.

range influenced performance. For the 4–10 scramble range, models trained on both 1–10 and 4–10 performed similarly, suggesting that excluding very easy examples (1–3 moves) did not hurt performance in this range. The effect became more pronounced when evaluating the harder 7–10 scramble range. The model trained on 1–10 scrambles outperformed the one trained only on 7–10, and the 4–10 model again had a similar performance as the 1–10 model. These results suggest that training on easier scrambles helps the model form generalizable solving strategies that it can then apply to harder cases. When only exposed to the most complex examples, the model seems to struggle with identifying useful patterns, leading to weaker generalization. At the same time, starting training at scramble length 4 still appears to provide enough structure for the model to learn effectively, as these states are still solved in about 50 % of cases. This implies that while exposure to easier examples is important, there is some flexibility in how easy those examples need to be for the model to benefit.

5.5 2x2x2

To better understand the limitations and potential of our model, we also evaluated it on the $2 \times 2 \times 2$ RC (see figure 5.5). It is a simpler variant of the $3 \times 3 \times 3$ RC environment. While the action space remains the same, the overall state space is significantly smaller, with only 3 674 160 possible configurations [27], and a maximum distance of 14 moves from any state to the solved configuration [17] (compared to 26 for the $3 \times 3 \times 3$ cube). We trained the model for a total of 50 000 episodes, each with 50 steps, using scramble lengths between 1 and 14 moves. For evaluation, we tested the model on 10 000 scrambled cube states. The used model was the actor-critic architecture combined with bonus rewards and entropy regularization. As shown in Table 5.15, the model achieved a substantial improvement in performance, solving over 61 % of scrambled cubes. This improvement is not only due to the absence of long scrambles (i.e., 15–26 moves), which were excluded due to the nature of the $2 \times 2 \times 2$ cube. While on the $3 \times 3 \times 3$ cube, the agent only gets meaningful success rates until about 9 or 10 scrambles, for the $2 \times 2 \times 2$ cube, it achieves even

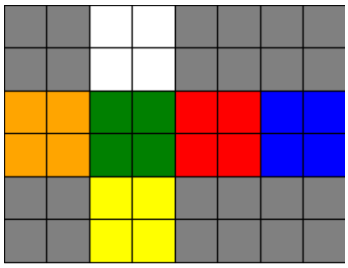

Figure 5.5: The grid of the solved $2 \times 2 \times 2$ cube.

Experiment	Architecture	Success Rate
2x2x2	Actor-critic	0.6154 ± 0.0018
2x2x2	GRU ($\gamma = 0.75$)	0.7633 ± 0.0249
2x2x2	**GRU($\gamma = 0.85$)**	$\mathbf{0.8004 \pm 0.0420}$

Table 5.15: Comparison of different setups on the $2 \times 2 \times 2$ RC regarding their success rate.

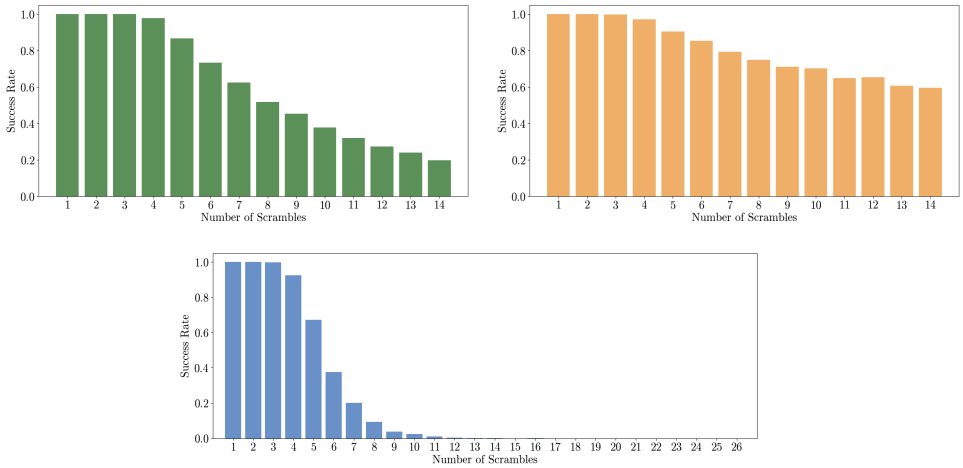

Figure 5.6: Success Rates for different Scramble Depths, using an actor-critic MLP architecture for the $3 \times 3 \times 3$ RC (blue) and for the $2 \times 2 \times 2$ RC (green). We also use a GRU to solve the $2 \times 2 \times 2$ RC (orange). The values are again averaged over four runs.

for 14 scrambles a success rate of 18 % to 20 % (see figure 5.6 (blue and green)).
Since we obtained such good results on the $2 \times 2 \times 2$ cube, we decided to try other architectures as well. Among them the easiest to adapt is the GRU, so we ran the experiment again using the bonus reward with entropy regularization and our GRU architecture. Surprisingly, as shown in Table 5.15, this setup achieved a significantly higher success rate than the actor-critic MLP architecture, reaching even for 14 scrambles a success rate of over 60 % (see figure 5.6 (orange)). This is unexpected, as in previous experiments the GRU performed significantly worse than the MLP setup (Table 5.7). One possible explanation for its strong performance now is that the $3 \times 3 \times 3$ cube was too complex, whereas now it can handle this environment and fully benefit from its sequential processing capabilities.
The results indicate that the $2 \times 2 \times 2$ cube's lower complexity, in terms of both its state space and solution depth, makes it far more manageable for our model. In this simpler setting, the agent seems better able to extract generalizable solving strategies, which are harder to learn or apply effectively in the more complex $3 \times 3 \times 3$ environment. This suggests that our model can generalize to some extent, but the $3 \times 3 \times 3$ cube may simply be too complex.

6 Conclusion

This thesis investigated whether the RC can be solved effectively using only RL, specifically policy gradient methods, without relying on supervised guidance, handcrafted heuristics, or search-based planning. We conducted extensive experiments across different neural network architectures, reward designs, and training strategies to evaluate both the strengths and limitations of policy gradient methods in this challenging environment.

Our experiments with policy gradient methods in the RC environment led to several important findings. Testing multiple neural network architectures like MLPs, RNNs, CNNs, and GNNs showed that none substantially outperformed the baseline MLP, with the best configuration achieving, without any additional RL techniques, a success rate of approximately 16.17 % on standard $3 \times 3 \times 3$ cubes scrambled with 1-26 moves. RNNs struggled to capture temporal dependencies, CNNs had difficulties modeling global cube interactions despite their spatial awareness, and GNNs suffered from oversmoothing that hindered decision-making.

A main problem we encountered was the agents' tendency to memorize short solution paths rather than develop strategies that can generalize to more complex states. Performance declined rapidly beyond 3-4 scrambles, likely because the exponential growth of the state space (see Figure 5.1) exceeded the model's learning capacity. This suggests that the models only learned shallow patterns rather than the cube's underlying structure.

Dense reward functions were essential for providing learning signals, but often led agents into local optima where they focused on maintaining high-quality states rather than progressing toward a solution. The most effective enhancement of our standard reward combined it with characteristics from sparse and dense rewards by using bonuses and entropy regularization. Actor-critic methods improved training stability, but did not lead to significant gains in overall performance.

Curriculum learning had limited impact, possibly because not enough training time was dedicated to each difficulty level or due to sudden jumps in complexity. However, training on simpler examples (1–10 scrambles) improved generalization more than training only on difficult cases (7-10 scrambles).

The contrast in results between $2 \times 2 \times 2$ cubes and $3 \times 3 \times 3$ cubes showed how sensitive our approach is to problem complexity. The smaller state space (3.67×10^6 vs. 4.3×10^{19} configurations) and shorter maximum solution depth (14 vs. 26 moves) made it possible for the success rate to rise to 80.04 %. This indicates that our policy gradient methods can learn well in simpler environments, but struggle when complexity increases.

While our approach did not achieve competitive performance, this work presents a structured comparison of neural architectures and their limitations in this domain. Our reward function

analysis shows how dense signals aid learning in sparse settings but can risk running into local optima. Adding a small bonus added motivation to solve the cube, but still was not enough to support generalization to more complex solution paths.
These results highlight the importance of architecture choice, reward shaping, and problem formulation. Despite not achieving optimal performance, we obtained some meaningful results, with the best success rate reaching 21.87 % using only basic RL techniques and simple architectures. This suggests opportunities for improvement for future work that employs more sophisticated and complex techniques designed to address the specific problems we encountered.

List of Acronyms

A*	A*
BPTT	Backpropagation Through Time
CNN	Convolutional Neural Network
DAVI	Deep Approximate Value Iteration
DQN	Deep Q-Network
GCN	Graph Convolutional Network
GNN	Graph Neural Network
GRU	Gated Recurrent Unit
HTM	Half-Turn Metric
IDA*	Iterative Deepening A*
LSTM	Long Short-Term Memory
MDP	Markov Decision Process
MLP	Multi Layer Perceptron
MP	Markov Process
MRP	Markov Rewards Process
NN	Neural Network
PDB	Pattern Databases
QTM	Quarter-Turn Metric
RC	Rubik's Cube
ReLU	Rectified Linear Unit
ResNet	Residual Neural Network
RL	Reinforcement Learning
RNN	Recurrent Neural Network
RTA*	Real Time A*
SGD	Stochastic Gradient Descent
TD	Temporal Difference

List of Symbols

Reinforcement Learning

$s, s_t, S_t, s_{\text{term}}$	state, state at time t, random variable S_t, terminal state
a, a_t, A_t	action, action at time t, random variable A_t
r, r_t, R_t	reward, reward at time t, random variable R_t
q, q_t	quality, quality at time t
$\mathcal{S}, \mathcal{S}_T$	set of all states, set of terminal states
$\mathcal{A}$	set of all actions
$\mathcal{R}$	set of all possible rewards
$p(s', r \mid s, a)$	state transition and reward dynamics function
T	episode length
$r(s_t, a_t)$	reward signal function
γ	discount factor
$\pi, \pi_\theta(a \mid s), \pi_*$	policy, parameterized policy, optimal policy
$v_\pi(s), \hat{v}(s, \omega)$	state-value function under policy π, estimated value function with parameters ω
$q_\pi(s, a)$	action-value function under policy π
θ, ω	policy parameters, value function parameters
α, β	learning rate for actor and critic updates
$\mu(s)$	stationary distribution over states under policy
$\tau = (s_0, a_0, s_1, a_1, \ldots)$	trajectory (sequence of states and actions)
$J(\theta) = \mathbb{E}[R(\tau)]$	objective function, expected return
$b(s)$	baseline function for variance reduction in policy gradient
δ_t	temporal difference error
G_t, G	return at time t

Neural Networks

$W, W^{(k)}$	weight matrix, weight matrix at layer k
b	bias vector
$\sigma, g^{(1)}, g^{(2)}$	non-linear activation functions
θ	network parameters
L	loss function
η	learning rate
D	input dimension
K	output dimension

h_t	hidden state at time t
c_t	cell state at time t (LSTM)
z_t	update gate in GRU
r_t	reset gate in GRU
$\hat{h}_t$	candidate hidden state in GRU
f	input function/image
g	kernel/filter function
i, j	pixel indices
m, n	kernel coordinate indices
$G = (V, E)$	graph with nodes and edges
V	set of nodes in graph
E	set of edges in graph
v	node in graph
x_v	feature vector of node v
$m_v^{(k)}$	message for node v at layer k
$\mathcal{N}(v)$	neighbors of node v
k	layer index

Search Methods

n	node in search graph
$h(n)$	heuristic function estimating cost to goal from node n
$g(n)$	cost function from start node to node n
$f(n)$	evaluation function for node n
s_n	corresponding state to node n
$A(s, a)$	next state function, state after applying action a in state s
$g^a(s, s')$	cost of transition from state s to s' by action a
$J(s)$	cost-to-go function approximated by DeepCubeA
$J'(s)$	updated cost-to-go function
λ	trade-off parameter between path cost and heuristic estimate

List of Figures

List of Tables

List of References

[1] F. Agostinelli, S. S. Shperberg, A. Shmakov, S. McAleer, R. Fox, and P. Baldi, "Q* search: Heuristic search with deep q-networks," in *ICAPS Workshop on Bridging the Gap between AI Planning and Reinforcement Learning*, 2024.

[2] S. A. e. a. Agostinelli F. McAleer S., "Solving the rubik's cube with deep reinforcement learning and search," Jul. 15, 2019.

[3] M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, O. Pieter Abbeel, and W. Zaremba, "Hindsight experience replay," *Advances in neural information processing systems*, vol. 30, 2017.

[4] E. Aronson, T. D. Wilson, and S. R. Sommers, *Social psychology*. Pearson Education India, 2005.

[5] Y. Bengio, J. Louradour, R. Collobert, and J. Weston, "Curriculum learning," in *Proceedings of the 26th annual international conference on machine learning*, 2009, pp. 41–48.

[6] E. Benhamou, "Variance reduction in actor critic methods (ACM)," *CoRR*, vol. abs/1907.09765, 2019. arXiv: 1907.09765. [Online]. Available: http://arxiv.org/abs/1907.09765.

[7] C. M. Bishop and H. Bishop, *Deep learning: Foundations and concepts*. Springer Nature, 2023.

[8] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst, "Geometric deep learning: Going beyond euclidean data," *IEEE Signal Processing Magazine*, vol. 34, no. 4, pp. 18–42, Jul. 2017, ISSN: 1558-0792. DOI: 10.1109/msp.2017.2693418. [Online]. Available: http://dx.doi.org/10.1109/MSP.2017.2693418.

[9] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[10] T. Chu, Y. Zhai, J. Yang, S. Tong, S. Xie, D. Schuurmans, Q. V. Le, S. Levine, and Y. Ma, *Sft memorizes, rl generalizes: A comparative study of foundation model post-training*, 2025. arXiv: 2501.17161 [cs.AI]. [Online]. Available: https://arxiv.org/abs/2501.17161.

[11] S. Geman, E. Bienenstock, and R. Doursat, "Neural networks and the bias/variance dilemma," *Neural computation*, vol. 4, no. 1, pp. 1–58, 1992.

[12] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *International conference on machine learning*, PMLR, 2017, pp. 1263–1272.

[13] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[14] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968. DOI: 10.1109/TSSC.1968.300136.

[15] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[16] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[17] D. Joyner, *Adventures in group theory: Rubik's Cube, Merlin's machine, and other mathematical toys*. JHU Press, 2008.

[18] T. N. Kipf and M. Welling, *Semi-supervised classification with graph convolutional networks*, 2017. arXiv: 1609.02907 [cs.LG]. [Online]. Available: https://arxiv.org/abs/1609.02907.

[19] R. E. Korf, "Depth-first iterative-deepening: An optimal admissible tree search," *Artificial Intelligence*, vol. 27, no. 1, pp. 97–109, 1985, ISSN: 0004-3702. DOI: https://doi.org/10.1016/0004-3702(85)90084-0. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0004370285900840.

[20] R. E. Korf, "Real-time heuristic search," *Artificial Intelligence*, vol. 42, no. 2, pp. 189–211, 1990, ISSN: 0004-3702. DOI: https://doi.org/10.1016/0004-3702(90)90054-4. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0004370290900544.

[21] R. 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*, ser. AAAI'97/IAAI'97, Providence, Rhode Island: AAAI Press, 1997, pp. 700–705, ISBN: 0262510952.

[22] A. Krogh and J. Hertz, "A simple weight decay can improve generalization," *Advances in neural information processing systems*, vol. 4, 1991.

[23] D. Kunkle and G. Cooperman, "Twenty-six moves suffice for rubik's cube," Jul. 2007, pp. 235–242. DOI: 10.1145/1277548.1277581.

[24] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998. DOI: 10.1109/5.726791.

[25] Q. Li, Z. Han, and X.-M. Wu, *Deeper insights into graph convolutional networks for semi-supervised learning*, 2018. arXiv: 1801.07606 [cs.LG]. [Online]. Available: https://arxiv.org/abs/1801.07606.

[26] Y. Lin and S. Liang, *Solving rubik's cube without tricky sampling*, 2024. arXiv: 2411.19583 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2411.19583.

[27] Z. Lyu, Z. Liu, A. Khojandi, and A. Junfang Yu, "Q-learning and traditional methods on solving the pocket rubik's cube," *Computers and Industrial Engineering*, vol. 171, p. 108 452, 2022, ISSN: 0360-8352. DOI: https://doi.org/10.1016/j.cie.2022.108452. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0360835222004867.

[28] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training recurrent neural networks," in *International conference on machine learning*, Pmlr, 2013, pp. 1310–1318.

[29] T. Rokicki, "Towards god's number for rubik's cube in the quarter-turn metric," *The College Mathematics Journal*, vol. 45, no. 4, pp. 242–253, 2014.

[30] T. Rokicki, H. Kociemba, M. Davidson, and J. Dethridge, "The diameter of the rubik's cube group is twenty," *siam REVIEW*, vol. 56, no. 4, pp. 645–670, 2014.

[31] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 1929–1958, Jan. 2014, ISSN: 1532-4435.

[32] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, second edition. MIT Press, Nov. 13, 2018, ISBN: 978-0-262-03924-6.

[33] K. Takano, "Self-supervision is all you need for solving rubik's cube," 2023. arXiv: 2106.03157 [cs.LG]. [Online]. Available: https://arxiv.org/abs/2106.03157.

[34] H. Van Hasselt, "Reinforcement learning in continuous state and action spaces," in *Reinforcement Learning: State-of-the-Art*, Springer, 2012, pp. 207–251.