Some Mathematical Foundations of Convolutional Neural Networks

---

A Thesis
Presented to
The Division of Mathematical and Natural Sciences
Reed College

In Partial Fulfillment
of the Requirements for the Degree
Bachelor of Arts

Kevin Thomas

May 2025

Approved for the Division
(Mathematics)

_____

Kyle Ormsby

Acknowledgements

I would like to thank my thesis advisor, Professor Kyle Ormsby, for his guidance, feedback, and support throughout this project. His insight helped shape the direction and clarity of this work.

I am also grateful to the Division of Mathematical and Natural Sciences at Reed College for providing the academic environment and resources necessary to complete this thesis.

Finally, I thank my family for their constant and unconditional support and encouragement during my studies.

List of Abbreviations

CNN	Convolutional Neural Networks
GDL	Geometric Deep Learning
G-CNN	Group Equivariant - Convolutional Neural Networks
SO(3)	Special Orthogonal Group in 3 Dimensions
ResNet	Residual Networks
ReLU	Recitifed Linear Unit
SGD	Stochastic Gradient Descent

Table of Contents

Abstract

This thesis explores the structure and behavior of Convolutional Neural Networks (CNNs), with a focus on their ability to process shifted input data. It presents the mathematical basis for CNNs using tools like circulant and shift matrices, showing how convolution operations are linear and shift-equivariant. While effective for translation-based tasks, standard CNNs perform poorly on transformations such as rotations and reflections. To address this, the thesis applies concepts from Geometric Deep Learning (GDL) and implements Group Equivariant CNNs (G-CNNs) that handle a broader set of symmetries. These models are tested on a Rubik's Cube solver, where various architectures—including standard CNNs, deeper ResNets, and symmetry-aware G-CNNs—are compared.

Introduction

Motivation

Convolutional Neural Networks (CNNs) have become foundational in deep learning, particularly for tasks involving spatially structured data such as images and videos. Their strength lies in their ability to exploit translational symmetry through shared weights and local connectivity, properties that can be formalized mathematically using tools such as circulant matrices and shift operators. However, standard CNNs are inherently limited to translation-based equivariance and often require significant data augmentation to handle other transformations like rotation and reflection.

Geometric Deep Learning (GDL) extends the CNN paradigm by incorporating broader symmetry groups into the model architecture. By embedding group-theoretic structure, GDL allows models to remain equivariant under a wider set of transformations. This is particularly relevant for structured problems like the Rubik's Cube, which exhibits a high degree of rotational symmetry. Solving the cube requires understanding not just the local arrangement of elements, but also how those arrangements change under group actions. This makes the Rubik's Cube an ideal candidate for studying the practical implications of symmetry-aware neural networks.

Objectives

The primary goal of this thesis is to explore the mathematical and computational foundations of CNNs and apply these insights to a structured problem domain—solving the Rubik's Cube. Specific objectives include:

- To understand the mathematical basis of convolution, shift equivariance, and their representation using circulant and shift matrices.
- To examine how symmetry groups can be used to generalize CNNs through the framework of geometric deep learning.
- To implement and compare different neural network architectures for the Rubik's Cube solver, including standard 3D CNNs, residual networks (ResNets), and Group Convolutional Neural Networks (G-CNNs).

Structure of the Thesis

The thesis is organized into four chapters:

- **Chapter 1** introduces the mathematical foundations of CNNs, focusing on convolution operations, shift equivariance, circulant matrices, and their algebraic representations.
- **Chapter 2** presents the theoretical background of geometric deep learning, group actions, and equivariance, and explains how G-CNNs extend traditional CNNs. It also discusses the vanishing gradient problem in deep networks and introduces ResNet architectures as a solution.
- **Chapter 3** describes the design and implementation of a Rubik's Cube solver. It includes the cube representation, data generation process, and model architectures. The chapter evaluates the performance of standard CNNs, ResNets, and G-CNNs in solving scrambled cube configurations.
- The concluding chapter summarizes the key findings of the thesis and reflects on the connection between mathematical theory and practical implementation.

Chapter 1

Convolutional Neural Networks and their Mathematics

Convolutional Neural Networks (CNNs) are a specialized class of deep learning models designed for structured data, such as images, where spatial relationships between features are crucial. CNNs leverage convolutional layers to extract hierarchical representations of data while maintaining shift equivariance. This chapter provides an in-depth overview of CNN architecture, training methodologies, and mathematical properties.

1.1 Introduction to CNNs

Deep learning has revolutionized artificial intelligence by allowing models to automatically learn representations from raw data. Neural networks, inspired by the human brain, consist of interconnected layers of neurons that process input data hierarchically. Among these, CNNs have become the dominant architecture for computer vision tasks.

Key motivations for CNNs:

- **Grid-structured data processing**: CNNs are particularly suited for images, videos, and other spatially structured data.
- **Feature extraction from localized regions of input**: Convolutional layers extract meaningful features by applying small learnable filters across the input.
- **Robustness to spatial translations**: Through convolutional and pooling layers, CNNs maintain translational equivariance and invariance.

A very good comprehensive guide on CNNs can be found at Kromydas (2023).

1.1.1 Architecture of CNNs

A typical CNN consists of multiple layers that transform input data into increasingly complex feature representations. Figure 1.1 provides an overview of a CNN architecture.

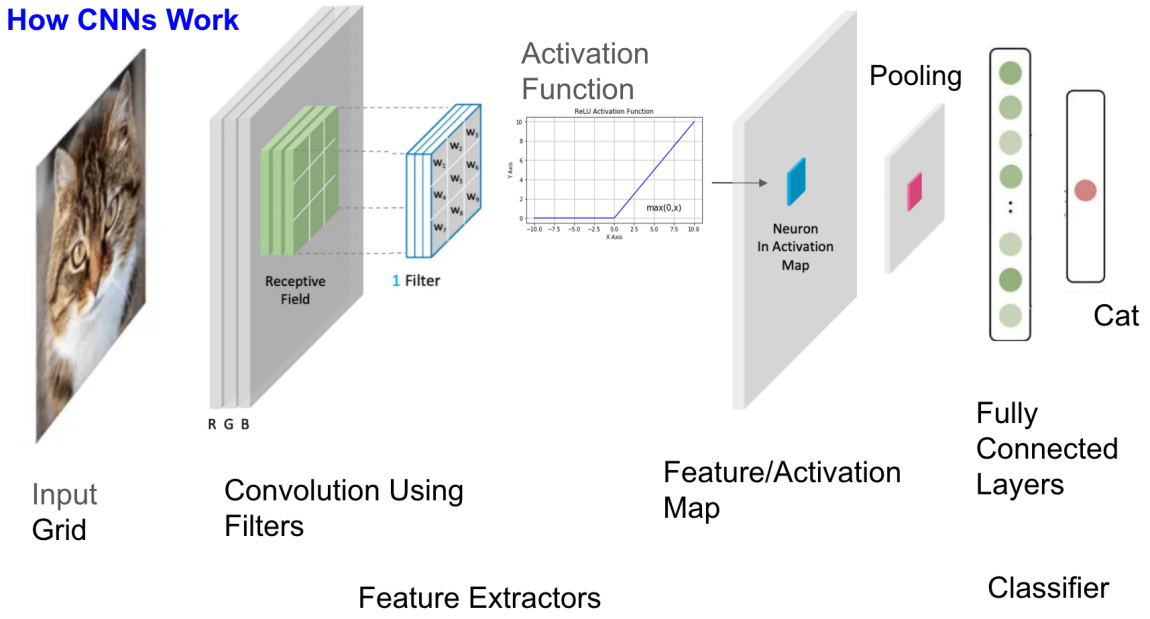

Figure 1.1: Components of a CNN

Image adapted from Kromydas (2023)

Input Representation

CNNs operate on structured data such as images, represented as multi-dimensional arrays. For example, RGB images are stored as 3D tensors with width, height, and depth corresponding to color channels.

Convolutional Layers

The convolution operation extracts features by applying a sliding filter (kernel) over the input. Apply a sliding filter to input data aggregates weighted values to detect patterns like edges or textures. This operation essentially performs a dot product. A sequence of convolution operations are shown in figure 1.2. This figure illustrates a 3×3 convolution over a 4×4 input. In each input panel, the numbers in the lower-right corners represent the filter weights applied to the highlighted region to compute the corresponding output value.

Activation Functions

Activation functions introduce non-linearity into the model. The most commonly used function in CNNs is the Rectified Linear Unit (ReLU), defined as:

$$\sigma(x) = \max(0, x)$$

where:
- x is the input to the activation function, typically the output of a neuron before applying the activation.

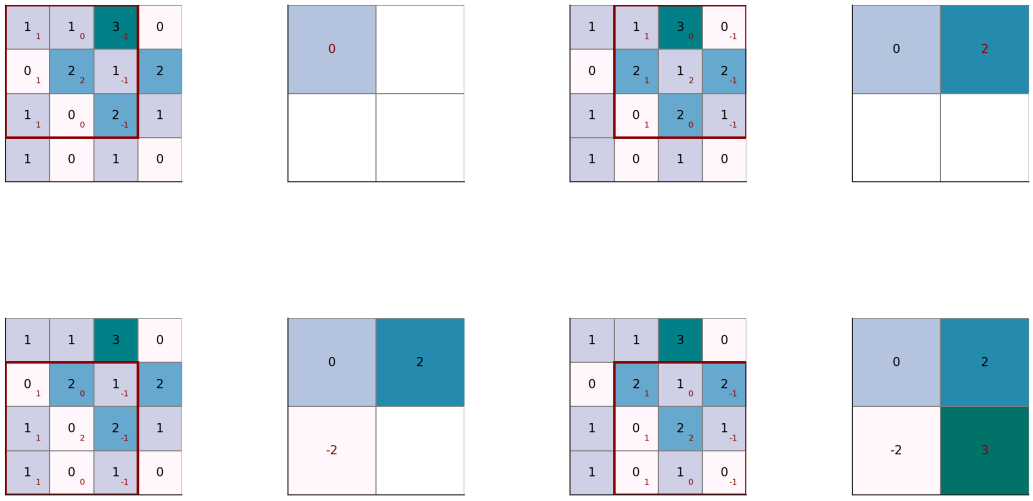

Figure 1.2: Convolution Operation

- $\sigma(x)$ is the transformed output after applying the ReLU function.

The function operates by returning zero for all negative input values ($x < 0$) and returning the input value itself for non-negative inputs ($x \geq 0$).

Feature Maps

As convolutional layers apply filters to the input, they produce feature maps, which highlight the presence of learned patterns such as edges, textures, or object parts. Each feature map corresponds to a different filter response, preserving spatial relationships in the image.

Pooling Layers

Pooling operations reduce the spatial resolution of feature maps by aggregating information over local neighborhoods, which helps to lower computational cost and introduce a degree of translational invariance.

One of the most common pooling strategies is **max pooling**, which selects the maximum activation within a local region of the input. This operation retains the most dominant features within each neighborhood while discarding less informative activations. Because small translations in the input often leave the local maximum unchanged, max pooling introduces approximate translational invariance in the resulting feature maps.

Fully Connected Layers

The final layers of a CNN are fully connected (dense) layers (figure 1.3), which map the extracted features to the desired output, such as class probabilities.

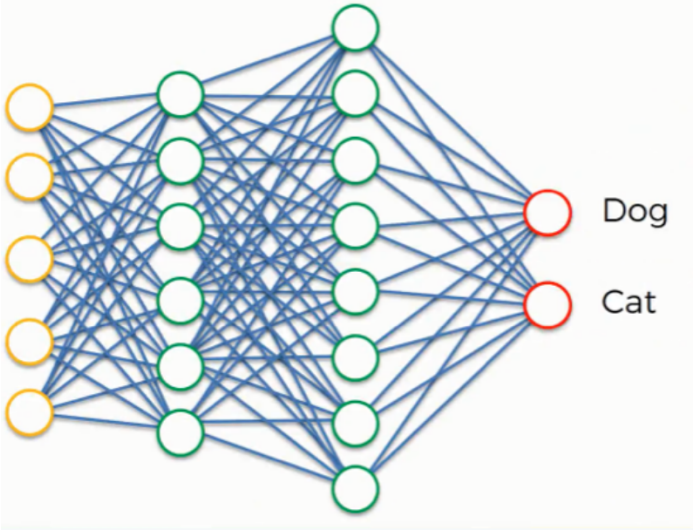

Figure 1.3: Fully Connected Layers

Image source: (SuperDataScience Team, 2021, Fully connected network)

1.1.2 Training CNNs

CNNs are trained using gradient-based optimization methods, where weights are iteratively updated to minimize a loss function.

Initialization

The initial values of weights significantly impact convergence. Common strategies include random initialization where small random values are assigned to weights.

Stochastic Gradient Descent and Backpropagation

CNNs learn by adjusting their internal weights to minimize prediction errors. This is typically done using an optimization method called **gradient descent**, which updates the weights in the direction that reduces a loss function — a scalar value measuring the difference between the network's predictions and the true targets.

In practice, CNNs are typically trained using a variant called **Stochastic Gradient Descent (SGD)**. Instead of computing the gradient over the entire dataset, SGD estimates the gradient using small, randomly selected mini-batches of data. This introduces stochasticity into the updates, making them noisier but often faster and more effective at escaping local minima.

The gradients are computed using **backpropagation**, which applies the chain rule to compute how the loss changes with respect to each weight. The update rule for weight W_k is given by:

$$W_k \leftarrow W_k - \eta \frac{\partial L}{\partial W_k}$$

where:

- W_k is the weight of the k-th filter.
- η is the learning rate, which determines the step size of each update.
- $\frac{\partial L}{\partial W_k}$ is the gradient of the loss function with respect to the weight vector W_k; it consists of partial derivatives indicating the direction and rate of change for each parameter in W_k.

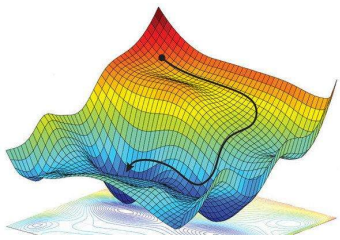

Figure 1.4: Stochastic Gradient Descent

Image source: (GL Academy, 2025, Image depicting non-convex optimization and stochastic gradient descent)

By repeatedly applying this update, the network gradually converges toward a set of weights that minimize the loss (Figure 1.4). The learning rate plays a critical role: if it's too large, training may become unstable; if too small, convergence can be excessively slow.

Stopping Criteria

Training in a neural network stops when certain conditions are met to prevent overfitting or unnecessary computations. One common criterion is early stopping, where training halts if the validation loss stops decreasing, indicating no further improvement. Alternatively, training can end after a predefined number of epochs, ensuring a fixed amount of learning iterations.

1.2 Mathematical Properties of CNNs

Convolution is a fundamental operation in deep learning, particularly in CNNs, enabling networks to learn spatially localized patterns in input data. It plays a crucial role in ensuring translation equivariance, meaning that a shift in the input results in a corresponding shift in the output. This section explores the mathematical foundations that make CNNs effective, focusing on shift equivariance, convolution operations, and the role of circulant matrices. By understanding how convolutions are represented using shift matrices and polynomial structures, we gain deeper insights into the efficiency of CNNs and the potential for further optimization, such as using the Fast Fourier Transform (FFT).

1.2.1 Convolution Operations

CNNs apply localized filters across an input to extract spatially dependent features. Mathematically, if $x \in \mathbb{R}^n$ is an input vector and $\theta \in \mathbb{R}^n$ is a filter (also called a kernel), convolution $x * \theta \in \mathbb{R}^n$ is defined as:

$$(x * \theta)_i = \sum_{j=0}^{n-1} x_j \, \theta_{(i-j) \bmod n}$$

for $i = 0, \ldots, n-1$. Here, n is the length of the input vector. This operation computes a weighted sum of input values, where the weights are determined by the filter coefficients. This definition extends naturally to two dimensions.

1.2.2 Shift Equivariance in Convolutions

Shift equivariance refers to the property where a transformation (such as translation) applied to an input is preserved in the output. Mathematically, if $f \colon X \to Y$ is a function representing a convolutional neural network operation, and T is a shift operator acting on elements of the domain X, then shift equivariance means:

$$(Tf)(x) = f(Tx)$$

Here, Tx denotes a shifted version of the input x, and $(Tf)(x)$ denotes the output of f shifted accordingly. This means that if the input shifts by a certain amount, the output feature map also shifts in the same manner.

CNNs maintain shift equivariance because:

- Convolutional filters slide over the input, applying the same transformation at every location.
- The weights of a convolutional filter are shared across spatial positions, ensuring consistency in feature extraction.
- No spatial information is lost during convolution, meaning the relative positioning of features remains intact.

This is a key property that enables CNNs to detect patterns regardless of their position in the input. Since convolutions operate locally and apply the same filter across different spatial locations, they maintain equivariance to shifts, meaning features detected in one region of an input will be similarly detected in shifted versions of the input.

Theorem 1. *Let $x \in \mathbb{R}^n$ and $\theta \in \mathbb{R}^n$. Define the shift operator T by $(Tx)_i = x_{(i-1) \bmod n}$. Then convolution $*$ satisfies*

$$T(x * \theta) = (Tx) * \theta.$$

Proof. First computing the shifted convolution:

$$(T(x * \theta))_i = (x * \theta)_{(i-1) \bmod n} = \sum_{j=0}^{n-1} x_j \, \theta_{(i-1-j) \bmod n}.$$

Next computing the convolution of the shifted input:

$$((Tx) * \theta)_i = \sum_{j=0}^{n-1} x_{(j-1) \bmod n} \, \theta_{(i-j) \bmod n}.$$

Letting $k = (j-1) \bmod n$, we obtain:

$$\sum_{k=0}^{n-1} x_k \, \theta_{(i-1-k) \bmod n},$$

which matches the previous expression. Thus,

$$T(x * \theta) = (Tx) * \theta.$$

This confirms that convolution is shift-equivariant. □

An intuitive visualization of this property is shown in Figure 1.5, where the shift and blur operations commute.

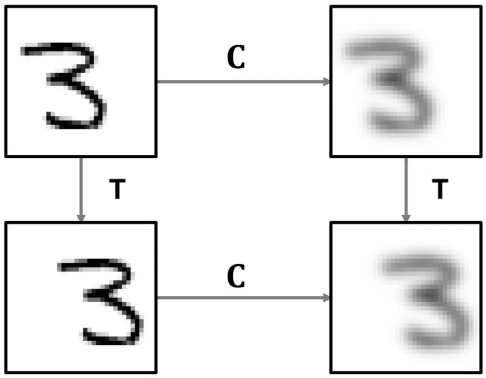

Figure 1.5: Illustration of Shift Equivariance

Image adapted from source: (Bronstein, 2020, Illustration of shift equivariance as the interchangeability of shift and blur operations)

1.2.3 Shift/Translational Invariance and Pooling Operations

While shift equivariance ensures that translations in the input are reflected in the output, **shift invariance** ensures that the output remains the same despite shifts in the input:

$$f(Tx) = f(x)$$

where T is a shift operator. Pooling operations, such as max pooling and average pooling, introduce approximate shift invariance by reducing spatial dimensions while preserving essential features. Pooling achieves this by selecting the most dominant features within a local receptive field, removing explicit spatial information.

Formally, if $x : \Omega \to \mathbb{R}$ is a discrete feature map defined on a domain Ω, and $u \in \Omega$ is a spatial location, then max pooling is defined as:

$$(Px)(u) = \max_{v \in N(u)} x(v)$$

where:

- Px is the pooled feature map (i.e., the result of applying the pooling operator to x),
- $N(u) \subset \Omega$ is the neighborhood around location u,
- $x(v)$ is the feature value at location $v \in N(u)$.

Pooling leads to a loss of precise location information but increases robustness to small input variations. Fully connected layers further enforce shift invariance by treating the extracted feature maps as unordered feature vectors rather than preserving spatial structure.

While convolutions preserve spatial structure, pooling ensures that extracted features remain robust to minor translations. An illustration of max and average pooling is shown in Figure 1.6.

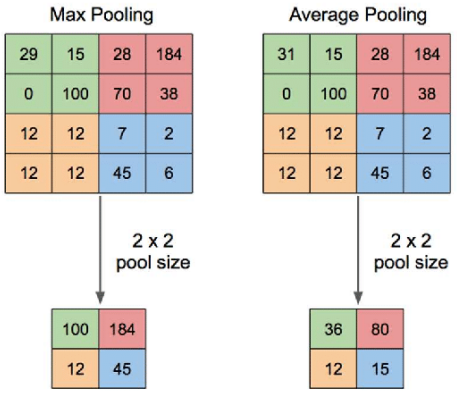

Figure 1.6: Illustration of Max and Average Pooling

Image source: (Yani et al., 2019, Illustration of Max Pooling and Average Pooling)

1.2.4 Circulant Matrices and Convolutions

Circulant matrices play a fundamental role in convolution operations, as they represent shift-equivariant linear transformations.

Definition and Properties

A matrix L is circulant if each row is a cyclic permutation of the row above it. Formally, an $N \times N$ circulant matrix takes the form:

$$L = \begin{bmatrix} \theta_0 & \theta_1 & \cdots & \theta_{N-1} \\ \theta_{N-1} & \theta_0 & \cdots & \theta_{N-2} \\ \vdots & \vdots & \ddots & \vdots \\ \theta_1 & \theta_2 & \cdots & \theta_0 \end{bmatrix}$$

where $\theta = (\theta_0, \theta_1, ..., \theta_{N-1})$ is the first row of L.

Circulant matrices have a highly regular structure: they commute with each other under matrix multiplication. A proof of this property is provided in Theorem 2, following the definition of the shift matrix. These matrices can also be transformed into a diagonal form, making them easier to work with and simplifying computations like matrix multiplication.. This makes them easy to analyze and efficient to compute with, especially in applications involving repeated structure like convolution (Bronstein et al. (2021b)). Convolution with a fixed filter corresponds to multiplying by a circulant matrix, which can be diagonalized. This allows convolution to be computed as a simple scaling operation in the right basis, greatly simplifying computation.

Shift Matrices and Equivariance

The **shift matrix** S is a circulant matrix that cyclically shifts a vector $v \in \mathbb{R}^N$ to the right:

$$S = \begin{bmatrix} 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 0 & 0 & \dots & 0 \end{bmatrix}$$

This matrix plays a central role in defining circulant matrices, since any circulant matrix can be written as a linear combination of powers of S.

Theorem 2. *Circulant matrices commute under matrix multiplication.*

Proof. Let $C(\theta)$ and $C(\phi)$ be two circulant $N \times N$ matrices generated by

$$\theta = (\theta_0, \theta_1, \dots, \theta_{N-1}), \quad \phi = (\phi_0, \phi_1, \dots, \phi_{N-1}).$$

Each circulant matrix can be written as a linear combination of powers of the shift matrix S:

$$C(\theta) = \sum_{k=0}^{N-1} \theta_k S^k, \quad C(\phi) = \sum_{l=0}^{N-1} \phi_l S^l.$$

All powers of S are taken modulo N, because $S^N = I$ and the powers form a cyclic group. Multiplying the matrices gives:

$$C(\theta)C(\phi) = \sum_{k=0}^{N-1} \sum_{l=0}^{N-1} \theta_k \phi_l S^{k+l},$$

and similarly,

$$C(\phi)C(\theta) = \sum_{l=0}^{N-1} \sum_{k=0}^{N-1} \phi_l \theta_k S^{l+k}.$$

Since scalar multiplication commutes and powers of S are taken modulo N, the terms S^{k+l} and S^{l+k} refer to the same elements in both sums. It follows that

$$C(\theta)C(\phi) = C(\phi)C(\theta).$$

□

Theorem 3. *If $L = C(\theta)$, a circulant matrix, then L is shift-equivariant.*

Proof. ($\Longrightarrow$) If $L = C(\theta)$, then L is shift-equivariant:

Circulant matrices commute with respect to multiplication. As both S and $C(\theta)$ are circulant,

$$SC(\theta) = C(\theta)S$$

Therefore:

$$SL = LS$$

This shows that L is shift-equivariant.

($\Longleftarrow$) If L is shift-equivariant, then $L = C(\theta)$ for some $\theta \in \mathbb{R}^n$:
Assume $SL = LS$, meaning L is shift-equivariant. All circulant matrices share the same eigenbasis, specifically the Fourier basis. As defined by Bronstein et al. (2021, Section 4.2, p. 38), the Fourier basis matrix Φ is an orthonormal[1] matrix composed of discrete Fourier transform eigenvectors. This means that the columns of Φ are eigenvectors common to all circulant matrices, and Φ^* is its conjugate transpose (also inverse, as Φ is unitary). Since L commutes with the circulant shift operator S, it is simultaneously diagonalizable with S in the Fourier basis:

$$L = \Phi \, \mathrm{diag}(\hat{\theta}) \, \Phi^*$$

Here, $\hat{\theta}$ denotes the eigenvalues of the matrix L in the Fourier domain. Any matrix diagonalizable by Φ corresponds exactly to a circulant matrix $C(\theta)$ in the original domain. Consequently, L must itself be circulant. Hence, both directions of the proof hold. □

This result shows that all convolution operations (implemented via circulant matrices) are shift-equivariant. The next theorem confirms whether the converse that every shift-equivariant linear transformation being a convolution is true.

Theorem 4. *Every shift equivariant linear transformation is a convolution.*

Proof. We begin by examining a simple 3×3 case to illustrate the idea. This example helps build intuition before we generalize to arbitrary size. Consider the 3×3 matrix:

$$L = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}.$$

Let S be the right cyclic shift matrix. Multiply L by S:

$$LS = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} c & a & b \\ f & d & e \\ i & g & h \end{bmatrix}.$$

Now, multiply S by L:

$$SL = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} = \begin{bmatrix} d & e & f \\ g & h & i \\ a & b & c \end{bmatrix}.$$

Equating $LS = SL$, we match the corresponding entries to get:

$$\begin{aligned} c = d, \quad a = e, \quad b = f, \\ f = g, \quad d = h, \quad e = i, \\ i = a, \quad g = b, \quad h = c. \end{aligned}$$

From these equations:

$$\begin{aligned} c = d = h \\ a = e = i \\ b = f = g \end{aligned}$$

Thus, L must have the form:

$$L = \begin{bmatrix} e & f & c \\ c & e & f \\ f & c & e \end{bmatrix},$$

which is a circulant matrix.

We now extend this result to the general case.

[1] Φ is an orthonormal matrix, meaning its columns form a set of vectors that are perpendicular (orthogonal) and have unit length (normalized). Consequently, its conjugate transpose Φ^* is also its inverse.

Let L be a linear transform that commutes with the shift matrix S. Assume that both L and S are $N \times N$ matrices. Multiplying L with S cyclically shifts the columns of L one position to the right. Each entry of LS is given by:

$$(LS)_{ij} = L_{i,(j-1) \bmod N}$$

Similarly, each entry of SL is given by:

$$(SL)_{ij} = L_{(i+1) \bmod N,j} \quad \text{(In this case, each row of } L \text{ shifts upward cyclically).}$$

Shift equivariance requires that $LS = SL$, hence:

$$L_{i,(j-1) \bmod N} = L_{(i+1) \bmod N,j}$$

This is the definition of a circulant matrix. Therefore, any linear operator L that commutes with S is circulant, which shows that L is a convolution. This confirms that convolutional layers inherently preserve shift equivariance. Circulant matrices replace convolution operation with matrix multiplication. □

The results above show that in one-dimensional settings, every linear transformation that is shift-equivariant must be a convolution, and such transformations are represented by circulant matrices. To generalize the notion of shift equivariance to 2D-images, we now introduce matrices that shift along rows and columns separately.

1.2.5 H and V Matrices: Extending Shift Equivariance

In one-dimensional domains, shift-equivariant linear operators are represented by circulant matrices. For two-dimensional data, such as images (which are naturally stored as $n \times n$ matrices), we need to define shift operators along both axes: horizontally (left-right) and vertically (up-down). These operators allow us to generalize the notion of shift equivariance from 1D to 2D.

Let $X \in \mathbb{R}^{n \times n}$ represent an image. The two matrices that encode cyclic shifts are defined below:

- The **horizontal shift matrix** $H \in \mathbb{R}^{n \times n}$ acts from the right: it shifts each row of the image one position to the left (i.e., a circular rightward shift across columns). That is,
$$(XH)_{i,j} = X_{i,(j+1) \bmod n}.$$
- The **vertical shift matrix** $V \in \mathbb{R}^{n \times n}$ acts from the left: it shifts each column of the image one position downward (i.e., a circular upward shift across rows). That is,
$$(VX)_{i,j} = X_{(i-1) \bmod n,j}.$$

Matrix representation of H and V: To represent these shifts using matrix multiplication, we define H and V as permutation matrices.
A one-step rightward circular shift is implemented by multiplying on the right with the matrix:

$$H[i,j] = \begin{cases} 1 & \text{if } j = (i+1) \bmod n, \\ 0 & \text{otherwise.} \end{cases}$$

For example, when $n = 3$, the matrix H becomes:

$$H = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}.$$

This matrix cyclically shifts each row of a matrix one step to the right.
Similarly, a one-step downward shift is implemented by multiplying on the left with:

$$V[i,j] = \begin{cases} 1 & \text{if } j = (i-1) \bmod n, \\ 0 & \text{otherwise.} \end{cases}$$

For $n = 3$, the matrix V takes the form:

$$V = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}.$$

This matrix cyclically shifts each column of a matrix one step downward.

Numerical example: To better understand how the shift matrices H and V act on 2D data, consider the following 3×3 matrix:

$$X = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}.$$

XH shifts each row to the right:

$$XH = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 3 & 1 & 2 \\ 6 & 4 & 5 \\ 9 & 7 & 8 \end{bmatrix}.$$

VX shifts each column down:

$$VX = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} = \begin{bmatrix} 7 & 8 & 9 \\ 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}.$$

HX shifts each column up:

$$HX = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} = \begin{bmatrix} 4 & 5 & 6 \\ 7 & 8 & 9 \\ 1 & 2 & 3 \end{bmatrix}$$

XV shifts each row to the left:

$$XV = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \cdot \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 \\ 5 & 6 & 4 \\ 8 & 9 & 7 \end{bmatrix}$$

Commuting with H and V: In two dimensions, a linear operator $L \in \mathbb{R}^{n\times n}$ is shift-equivariant if it commutes with both the horizontal and vertical shift matrices, H and V. That is,

$$LH = HL \quad \text{and} \quad LV = VL.$$

Such matrices preserve the structure of the data under circular shifts in both directions. A key example of this is the class of **doubly circulant matrices**, which are circulant along both rows and columns.
A general form of a doubly circulant matrix for $n = 3$ is:

$$L = \begin{bmatrix} \alpha & \beta & \gamma \\ \gamma & \alpha & \beta \\ \beta & \gamma & \alpha \end{bmatrix}.$$

This structure ensures that each row is a cyclic shift of the one above it. Such matrices commute with H and V.

Numerical verification: Let

$$L = \begin{bmatrix} 2 & 3 & 4 \\ 4 & 2 & 3 \\ 3 & 4 & 2 \end{bmatrix}, \quad H = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}.$$

Then:

$$LH = \begin{bmatrix} 3 & 4 & 2 \\ 2 & 3 & 4 \\ 4 & 2 & 3 \end{bmatrix}, \quad HL = \begin{bmatrix} 3 & 4 & 2 \\ 2 & 3 & 4 \\ 4 & 2 & 3 \end{bmatrix}.$$

Since $LH = HL$, this confirms that L commutes with H. The same holds for the vertical shift matrix V, verifying that L also satisfies $LV = VL$. The next section formalizes how these operators and introduces their algebraic structure using matrix polynomials.

1.2.6 Polynomial Representation of Shift-Equivariant Matrices

Circulant matrices are fundamental in the study CNNs due to their shift-equivariant properties. These matrices can be expressed as polynomials in terms of shift operators, capturing their essential structure in a concise algebraic form (Kra & Simanca, 2012, page 371).

Circulant Matrices as Polynomials in the Shift Matrix

A circulant matrix L can be expressed as a polynomial in the horizontal shift matrix H:

$$L = a_0 I + a_1 H + a_2 H^2 + \cdots + a_{N-1} H^{N-1}.$$

where H is the matrix that cyclically shifts rows to the right.
Alternatively, using an indeterminate X, the matrix can be represented as:

$$P_L(X) = a_0 + a_1 X + a_2 X^2 + \cdots + a_{N-1} X^{N-1}.$$

where X acts as a formal shift operator. A similar representation can be done using the vertical shift matrix V.

$$L = a_0 I + a_{N-1} V + a_{N-2} V^2 + \cdots + a_1 V^{N-1}.$$

This polynomial representation provides an intuitive framework for analyzing the properties of circulant matrices.

Properties of Shift Matrices H and V

The horizontal shift matrix H and vertical shift matrix V are used to model circular row-wise and column-wise shifts, respectively. To understand their structure and algebraic properties, we use the horizontal shift matrix H, which performs a circular left shift when applied to each row of a matrix.

The matrix $H \in \mathbb{R}^{n \times n}$ satisfies the following identities:

$$H^0 = I, \qquad H^n = I,$$

where I is the identity matrix of size $n \times n$. The second identity reflects that applying n shifts returns each row to its original state. More generally, the power H^k corresponds to a k-step circular left shift, forming a cyclic group of order n under matrix multiplication.

Example: Horizontal Shift Matrix for $n = 3$ For $n = 3$, the shift matrix H and its powers are:

$$H^1 = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}, \quad H^2 = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}.$$

Example: Vertical Shift Matrix for $n = 3$ For $n = 3$, the shift matrix V and its powers are:

$$V^1 = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}, \quad V^2 = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}.$$

Example: Circulant Matrix via Shift Powers A circulant matrix is one where each row is a cyclic shift of the previous. Such matrices can be expressed as a linear combination of powers of H. For example:

$$L = \begin{bmatrix} a_0 & a_1 & a_2 \\ a_2 & a_0 & a_1 \\ a_1 & a_2 & a_0 \end{bmatrix} = a_0 I + a_1 H + a_2 H^2.$$

Expanding each term:

$$a_0 \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} + a_1 \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} + a_2 \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} a_0 & a_1 & a_2 \\ a_2 & a_0 & a_1 \\ a_1 & a_2 & a_0 \end{bmatrix}.$$

This demonstrates how shift matrices like H serve as generators for circulant structure in linear operators.

Similar constructions can be applied to the vertical shift matrix V, which governs column-wise circular shifts.

$$L = \begin{bmatrix} a_0 & a_1 & a_2 \\ a_2 & a_0 & a_1 \\ a_1 & a_2 & a_0 \end{bmatrix} = a_0 I + a_2 V + a_1 V^2.$$

$$L = a_0 \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} + a_2 \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} + a_1 \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} a_0 & a_1 & a_2 \\ a_2 & a_0 & a_1 \\ a_1 & a_2 & a_0 \end{bmatrix}.$$

Theorem 5. *Every circulant matrix L can be written as a polynomial in the shift matrix H.*

Proof. 1. By definition, a circulant matrix satisfies $HL = LH$.
2. Each row of L is obtained by cyclically shifting the first row.
3. This implies that L can be expressed as a sum of powers of H:

$$L = P(H) = a_0 I + a_1 H + a_2 H^2 + \cdots + a_{N-1} H^{N-1}.$$

Thus, L is a polynomial in H, proving the claim. □

A similar proof works for the shift matrix V too.

Commutativity with Vertical Shifts and Doubly Circulant Matrices

For shift-equivariant operations over 2D domains (e.g., images), a transformation matrix must commute with both the horizontal shift matrix H and the vertical shift matrix V. That is, a matrix $L \in \mathbb{R}^{n \times n}$ satisfies shift equivariance in both spatial dimensions if and only if:

$$LH = HL, \qquad LV = VL.$$

A matrix that commutes with both H and V is called **doubly circulant**, meaning it is circulant in both rows and columns. Such matrices preserve the structure of translations along both axes.

This implies that L must be a polynomial in both H and V, leading to the general form:

$$L = \sum_{k=0}^{N-1} \sum_{\ell=0}^{N-1} a_{k,\ell} H^k V^\ell.$$

Expanding this equation for a 3×3 matrix, we have:

$$\begin{aligned}
L = {} & a_{00}H^0V^0 + a_{01}H^0V^1 + a_{02}H^0V^2 + \\
& a_{10}H^1V^0 + a_{11}H^1V^1 + a_{12}H^1V^2 + \\
& a_{20}H^2V^0 + a_{21}H^2V^1 + a_{22}H^2V^2 \\
L = {} & a_{00}\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} + a_{01}\begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} + a_{02}\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} \\
& + a_{10}\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} + a_{11}\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} + a_{12}\begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \\
& + a_{20}\begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} + a_{21}\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} + a_{22}\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \\
L = {} & \begin{bmatrix} a_{00}+a_{11}+a_{22} & a_{02}+a_{10}+a_{21} & a_{01}+a_{12}+a_{20} \\ a_{01}+a_{12}+a_{20} & a_{00}+a_{11}+a_{22} & a_{02}+a_{10}+a_{21} \\ a_{02}+a_{10}+a_{21} & a_{01}+a_{12}+a_{20} & a_{00}+a_{11}+a_{22} \end{bmatrix} \\
L = {} & \begin{bmatrix} \alpha & \beta & \gamma \\ \gamma & \alpha & \beta \\ \beta & \gamma & \alpha \end{bmatrix}
\end{aligned}$$

where L is a doubly circulant matrix. This representation naturally generalizes the concept of shift-equivariant operations to two-dimensional convolutional layers in CNNs.

By explicit computation, one can see that a doubly circulant matrix has the same structure whether represented using H, V, or their combinations. This shows that a matrix is doubly circulant if and only if it is circulant. A doubly circulant matrix satisfies both row-wise and column-wise cyclic shifts as shown below.

$$L = \begin{bmatrix} a_0 & a_1 & a_2 \\ a_2 & a_0 & a_1 \\ a_1 & a_2 & a_0 \end{bmatrix}.$$

The polynomial representation of shift-equivariant operators not only provides theoretical clarity but also allows for efficient implementation of convolutions. These

structures explain why CNNs excel at detecting patterns that are spatially translated.

However, this framework is inherently limited to translations. Real-world images often involve richer transformations—such as rotations, reflections, or scalings—that fall outside the scope of translation equivariance. These limitations motivate the need for models that generalize beyond standard convolutional layers.

1.3 Challenges with Standard CNNs

CNNs can operate efficiently due to various mathematical principles, including shift-equivariant structures, polynomial representations of circulant matrices, and fast computational techniques like FFT. These mathematical foundations not only enable convolution operations but also optimize their performance, making CNNs both powerful and scalable. Even though all these methods make CNNs highly effective for image processing, they struggle with transformations beyond simple translations. Standard CNNs do not inherently generalize to rotated or reflected versions of the same object, making them sensitive to such variations. As a result, extensive data augmentation is often required to improve robustness.

To overcome these limitations, researchers have developed Geometric Deep Learning, which extends CNNs by incorporating group equivariance. This enables models to handle more complex transformations, such as rotations and reflections, without relying solely on data augmentation. A key advancement in this field is the introduction of Group Convolutional Neural Networks (G-CNNs), which leverage symmetry-aware architectures to improve generalization across different geometric transformations.

1.4 Summary

This chapter introduced the fundamental architecture and training principles of CNNs, focusing on their ability to extract hierarchical features while maintaining shift equivariance. A key mathematical tool in this context is the circulant matrix, which captures the structure of convolutions and enables efficient computation through polynomial representations. These properties make CNNs highly effective for translation-invariant tasks. However, CNNs struggle with more general transformations, such as rotations and scalings, which are essential in many real-world applications. This limitation motivates the development of geometric deep learning methods that extend equivariance beyond translations. The next chapter explores the mathematical foundations necessary to generalize CNNs to broader classes of symmetries, enabling more robust feature extraction and improved performance in tasks requiring rotational and scale invariance.

Chapter 2

Geometric Deep Learning and Group Equivariant CNNs

Traditional Convolutional Neural Networks (CNNs) are highly effective for learning translational symmetries in data. However, they struggle with other transformations such as rotations and reflections. Geometric Deep Learning extends CNNs by incorporating group equivariance, allowing models to natively handle broader symmetry transformations. Extensive background on this work is available at (Bronstein et al., 2021b, sections 5.1 and 5.2) and Bronstein et al. (2017). A GDL course also provides lecture recordings on this subject (Bronstein et al. (2021a)). This chapter discusses the mathematical foundations of group-equivariant Convolutional Neural Networks (G-CNNs), the transform and convolve approach, and the role of residual connections in stabilizing deeper networks.

2.1 Symmetry Groups and Group Equivariance

In the real world, objects don't just shift around—they rotate, flip, and scale too. But standard CNNs only handle shifts well, which is why they struggle with things like rotated or mirrored images. This section dives into how we can use symmetry groups to make neural networks smarter about these transformations.

2.1.1 Limitations of Standard CNNs

While CNNs maintain shift equivariance, meaning that a shift in input results in a corresponding shift in output, they do not inherently generalize to other transformations such as:

- **Rotations**: A rotated object may not activate the same filters.
- **Reflections**: CNNs do not recognize flipped versions of learned features.
- **Scaling**: Different object sizes require multi-scale feature extraction.

These limitations necessitate extensive **data augmentation**, artificially generating transformed copies of training images. However, this approach is computationally expensive and does not guarantee full generalization.

2.1.2 Extending Equivariance Beyond Translation

In mathematics, symmetries are described using groups, which capture how transformations like rotations, reflections, and shifts interact. The symmetric group S_n, for example, represents all possible permutations of n elements, illustrating the power of group theory in structuring transformations. More relevant to images, groups like the rotation group $SO(2)$ or the dihedral group D_n describe transformations that preserve visual structures.

Group-equivariant neural networks (G-CNNs) extend traditional CNNs by leveraging group theory, ensuring that transformations of the input correspond to predictable transformations of the output. Mathematically, for a transformation g from a symmetry group G:

$$f(g \cdot x) = g \cdot f(x), \quad g \in G.$$

In this equation, x represents the input (e.g., an image), g is a transformation from a symmetry group G (such as a rotation or reflection), and f is the function (the neural network) that processes x.
If an input image is rotated or flipped, the network's response transforms in a predictable way rather than needing to relearn features from scratch. This means that instead of only detecting patterns in fixed orientations, G-CNNs enforce equivariance to predefined transformation groups such as rotations and reflections.

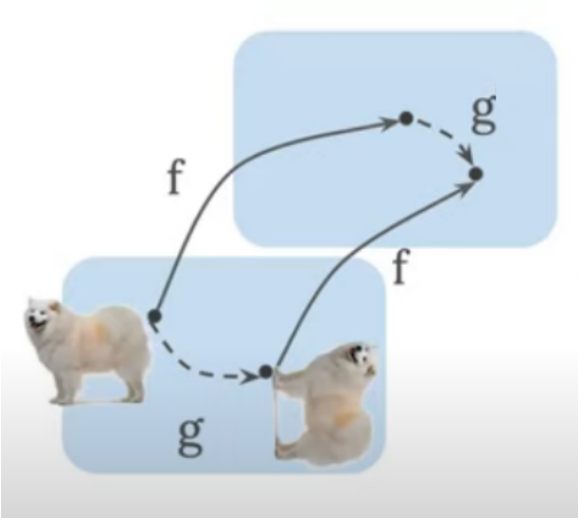

Figure 2.1: Illustration of Group Equivariance
Image adapted from: (van der Pol, 2021, Image depicting Equivariance)

Figure 2.1 visually demonstrates the idea of group equivariance. The image undergoes a transformation g (such as a rotation), and then the function f (representing the CNN) processes it. If the network is equivariant, applying f before or after the transformation leads to a consistent and predictable change in the output.

Mathematically, group convolution is typically defined as follows (Bronstein et al., 2021):

$$(x *_G \theta)(g) = \sum_{u \in \Omega} x(u) \cdot \theta(g^{-1} \cdot u)$$

where:

- Ω is the base domain on which the input is defined and on which the group acts,
- $x : \Omega \to \mathbb{R}$ is the input (e.g., an image),
- $\theta : \Omega \to \mathbb{R}$ is the convolutional filter,
- $g \in G$ is a transformation from the symmetry group (e.g., a rotation or reflection) and g^{-1} is the inverse of transformation

This formulation ensures that the convolution respects the group symmetry by transforming the filter θ under the action of g.

Theorem 6 (Equivariance of G-Convolution)**.** *Let G be a group acting on a domain Ω, and let $x : \Omega \to \mathbb{R}$ be an input function. Let $\theta : \Omega \to \mathbb{R}$ be a filter. Define the* G-convolution *of x with θ as*

$$(x *_G \theta)(g) := \sum_{u \in \Omega} x(u) \cdot \theta(g^{-1}u), \quad \textit{for all } g \in G.$$

Then G-convolution is equivariant with respect to the group action on inputs. That is, for all $g_1, g \in G$,

$$((g_1 \cdot x) *_G \theta)(g) = (x *_G \theta)(g_1^{-1}g),$$

where the group action on inputs is defined as

$$(g_1 \cdot x)(u) = x(g_1^{-1}u).$$

Proof. We compute the convolution of the transformed input $g_1 \cdot x$:

$$\begin{aligned}((g_1 \cdot x) *_G \theta)(g) &= \sum_{u \in \Omega} (g_1 \cdot x)(u) \cdot \theta(g^{-1}u) \\ &= \sum_{u \in \Omega} x(g_1^{-1}u) \cdot \theta(g^{-1}u).\end{aligned}$$

Let $v = g_1^{-1}u$, so that $u = g_1 v$. Since G acts bijectively on Ω, we can change variables in the sum:

$$= \sum_{v \in \Omega} x(v) \cdot \theta(g^{-1}g_1 v) = \sum_{v \in \Omega} x(v) \cdot \theta((g_1^{-1}g)^{-1}v) = (x *_G \theta)(g_1^{-1}g).$$

Thus, we conclude:

$$((g_1 \cdot x) *_G \theta)(g) = (x *_G \theta)(g_1^{-1}g),$$

which proves that G-convolution is equivariant under the group action. □

Having established the theoretical foundation for group equivariance, we now turn to how these principles are implemented in practice through group convolutions and the transform-and-convolve approach.

2.2 Group Convolutions and the Transform and Convolve Approach

In a standard CNN, convolution applies a filter across spatial locations. In a Group CNNs, convolution needs to be extended to operate over transformations from a symmetry group. Instead of using only one fixed filter, one approach for G-CNNs transforms the filter itself under multiple orientations before applying convolution. This allows the network to learn features that are naturally equivariant under rotations, reflections, or other group operations.

2.2.1 Transform and Convolve Approach

The Transform and Convolve approach enables a CNN to learn equivariant representations by modifying its filters according to group transformations before applying convolution. The process involves:

- Canonical Filter: A base filter θ is initialized and optimized during training.
- Generate Transformed Filters: Multiple transformed versions of θ are created using group transformations g.
- Apply Convolution: Each transformed filter is convolved with the input.
- Combine Results: The feature maps from different transformations are aggregated.

Rather than computing the full group convolution in one step, modern architectures often implement it via the Transform-and-Convolve strategy. The filter θ is first transformed under several group elements $g \in G$, and each transformed filter is then convolved with the input x using standard convolution. This enables the network to share weights across symmetrical configurations and maintain equivariance.

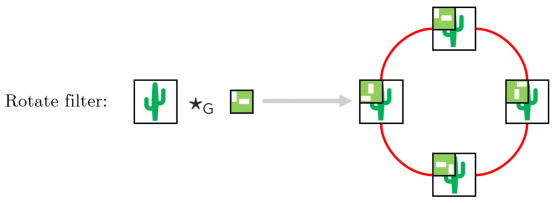

Figure 2.2: Filter Rotation for Group Convolution

Image source: (Van Engelenburg, 2020, G-convolution on a planar input)

Figure 2.2 illustrates the Transform and Convolve approach in action. Instead of using a single fixed filter, the filter is rotated under a group of transformations before being applied to the input. This ensures that the network can detect features consistently across different orientations, making it naturally equivariant to rotations. The blog by Van Engelenburg (2020), highlights how G-CNNs reduce redundancy in learning by sharing parameters across transformed versions of features.

In G-CNNs, pooling works by grouping together similar transformations, like rotations or reflections, to keep important patterns while reducing the size of the data. Subgroup pooling pools over a smaller set of transformations, while coset pooling makes features fully invariant to certain transformations (Cohen & Welling, 2016, section 6.3). Nonlinearity is applied to each feature individually, meaning it doesn't interfere with how the network recognizes patterns across transformations. Both pooling and nonlinearity are designed to keep the network's ability to track patterns consistently, no matter how they move or change.

Group convolutions improve a network's ability to recognize patterns across transformations. We also need deeper networks (with multiple convolution layers) for hierarchical feature learning, capturing low-level edges in early layers and more complex patterns in deeper layers. However, simply stacking more layers introduces challenges like vanishing gradients and feature degradation. To address these issues, Residual Networks (ResNets) introduce skip connections that help stabilize learning and enable deeper architectures. The next section explores how ResNets improve feature propagation and complement equivariant architectures.

2.3 ResNet and Feature Stability

Deeper networks often suffer from vanishing gradients and feature degradation, where early layers stop learning meaningful representations. Residual Networks (ResNets) He et al. (2016) address this by introducing skip connections, enabling direct feature reuse across layers.

2.3.1 Vanishing Gradient Problem

G-CNNs update weights using Stochastic Gradient Descent (SGD) during backpropagation. As networks get deeper, gradients shrink, making learning inefficient. This **vanishing gradient problem** occurs because gradients are repeatedly multiplied by small values, leading to near-zero updates in early layers. It was highlighted in early work on long-term dependencies Bengio et al. (1994).

Example Weight Updates in a Two-Layer CNN

Consider a convolutional neural network (CNN) with two layers:

- **Layer 1**: Has weights W_1, takes input x, and produces activation h_1.
- **Layer 2**: Has weights W_2, takes h_1 as input, and produces output $\hat{y}$.

In a simple two-layer case, the forward pass is:

$$h_1 = \text{ReLU}(W_1 \cdot x)$$

$$\hat{y} = \text{ReLU}(W_2 \cdot h_1)$$

We define the loss using Mean Squared Error (MSE):

$$L = (\hat{y} - y)^2$$

During backpropagation, the gradient with respect to W_2 is proportional to:

$$\frac{\partial L}{\partial W_2} \propto \text{ReLU}'(W_2 \cdot h_1) \cdot h_1$$

(here, $\text{ReLU}'(z)$ denotes the derivative of the ReLU function with respect to its input z, equal to 1 if $z > 0$ and 0 otherwise)
However, the gradient with respect to W_1 must pass through both layers:

$$\frac{\partial L}{\partial W_1} \propto \text{ReLU}'(W_2 \cdot h_1) \cdot \text{ReLU}'(W_1 \cdot x) \cdot x$$

Each multiplication by a derivative — especially when ReLU outputs zero — can shrink the gradient. As more layers are added, this effect compounds. The result is that gradients reaching early layers become vanishingly small, and those layers stop learning effectively. This issue arises in both standard CNNs and group-equivariant CNNs. Without intervention, stacking many layers leads to stagnation in training. The next section discusses how residual connections can alleviate this problem by ensuring better gradient flow in deep architectures.

2.3.2 Residual Learning with Skip Connections

To address the vanishing gradient problem, Residual Networks (ResNets He et al. (2016)) introduce **skip connections** that allow activations and gradients to bypass intermediate layers. Instead of learning a full mapping $H(x)$ directly, the network learns a residual function $F(x)$ and adds it back to the input:

$$H(x) = x + F(x) \quad \text{with} \quad F(x) = \text{ReLU}(W_2 \cdot \text{ReLU}(W_1 \cdot x)).$$

Here:

- x is the input to the residual block.
- $F(x)$ represents the residual function, typically a stack of convolutional layers with activation.
- The identity connection ensures that gradients can bypass $F(x)$, preventing vanishing gradients.

The output of the block is:

$$\hat{y} = \text{ReLU}(x + F(x)).$$

During backpropagation, the Jacobian (matrix of partial derivatives) of output $H(x)$ with respect to the input is:

$$\nabla H(x) = I + \nabla F(x),$$

where I is the identity matrix. The additive identity ensures that the gradient is never completely suppressed, even when $\nabla F(x)$ is small. Figure 2.3 illustrates this concept. Instead of passing the input x only through convolutional layers, the network also includes a shortcut (identity) connection that directly adds x to the transformed output $F(x)$. This addition ensures that even if $F(x)$ produces small gradients, the identity path preserves strong gradient flow, stabilizing deep network training.

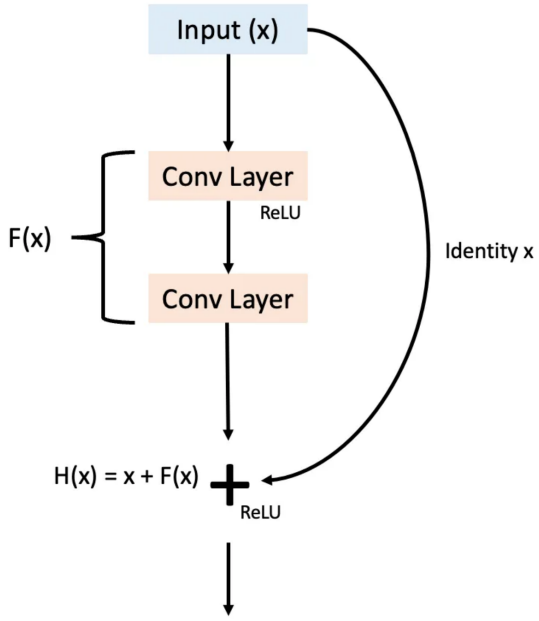

Figure 2.3: Residual Learning in Deep Networks

Image source: (Martins, 2023, Figure showing residual connections)

Residual connections not only preserve gradient flow, but also shift the design philosophy of CNNs. Traditionally, each stage of a CNN applied a sequence of nonlinearity and pooling to a learned feature transformation:

$$H(x) = P(\sigma(F(x))).$$

where σ is a pointwise activation (such as ReLU) and P is a pooling operation.
With residual learning, the output instead combines the input and the transformed features before pooling ((Bronstein et al., 2021b, Section 5.1)):

$$H(x) = P\big(x + \sigma(F(x))\big).$$

Together, these components—group-equivariant convolutions and residual connections—form the foundation for building deep, stable architectures that can generalize across symmetry transformations.

2.4 Summary

Traditional CNNs effectively capture translational symmetries but struggle with rotations and reflections. Geometric Deep Learning extends CNNs by incorporat-

ing group equivariance, enabling predictable transformations in response to input changes. Group-equivariant CNNs (G-CNNs) achieve this by using group convolutions, where filters are transformed under symmetry groups before applying convolution. This approach generalizes features across orientations without relying on data augmentation. However, deep networks suffer from vanishing gradients, making early layers ineffective. Residual Networks (ResNets) solve this by introducing skip connections, allowing direct gradient flow and stabilizing training. Instead of learning direct mappings, ResNets optimize residual functions, ensuring deeper networks remain trainable. This combination of G-CNNs and ResNets enhances feature learning across transformations while maintaining stability in deep architectures.

Building on these principles, the next chapter explores the Rubik's Cube as a testbed for geometric deep learning. The cube's transformations naturally align with the symmetry group $SO(3)$, making it an ideal candidate for studying equivariant neural networks. We will implement a 3D Rubik's Cube solver using group-equivariant architectures, leveraging the structure of rotation groups to improve generalization. Additionally, we will investigate a ResNet-based approach to enhance stability and efficiency in learning cube transformations. This fusion of symmetry-aware deep learning and residual networks aims to develop a robust model for solving the Rubik's Cube efficiently.

Chapter 3

Rubik's Cube Solver Implementation

In this chapter, the implementation of a Rubik's Cube (Rubik (1974)) solver using deep learning techniques is discussed. A Rubik's Cube is a 3D combination puzzle consisting of six colored faces that must be restored to uniform colors through a sequence of rotations. There have been several approaches to solving the Rubik's Cube using neural networks. McAleer et al. (McAleer et al. (2018)) applied deep Q-learning to train a network capable of solving the cube from scratch. Gottlieb and Burch (Gottlieb & Burch (2019)) explored a hybrid approach combining Monte Carlo Tree Search with neural guidance. The most effective method to date, DeepCubeA (Agostinelli et al. (2019)), uses a deep value network to solve the cube from any state without demonstrations.

The approach in this work is to explore the mathematical structure of the cube's symmetry group and incorporate it into the model architecture. It begins with a standard 3D convolutional neural network and is later extended to a Group-Convolutional Neural Network (G-CNN) to leverage the cube's intrinsic rotational symmetries. The goal is to investigate how embedding group-theoretic structure into the network—via equivariant convolutions—affects learning efficiency and generalization.

3.1 Rotational Symmetries and SO(3)

The rotational properties of the Rubik's Cube are characterized using the special orthogonal group in three dimensions, denoted $SO(3)$, which consists of all orientation-preserving rotations in $\mathbb{R}^3$.

A standard cube possesses exactly 24 distinct orientation-preserving symmetries, forming a finite subgroup $G \subset SO(3)$. Each element $g \in G$ corresponds to a rotation that maps the cube onto itself. These 24 discrete symmetries play a crucial role in defining group-equivariant operations for learning tasks involving 3D spatial data.

3.2 Group Convolution Framework

Group convolution extends the traditional convolution operation by incorporating transformations from a symmetry group. In the 3D setting, a convolutional kernel $\kappa(\mathbf{y})$ is typically applied at offsets $\mathbf{y}$ across the input. In a group-convolutional setting, a rotated version of the kernel is defined for each group element $g \in G$ as $\kappa_g(\mathbf{y}) = \kappa(g^{-1} \cdot \mathbf{y})$, and convolved with the input accordingly.

The outputs obtained from each transformed kernel can be aggregated, either by summing or averaging, or retained as separate feature channels. This ensures equivariance of the convolution operation—applying a group transformation to the input results in a corresponding transformation of the output. When applied to Rubik's Cube configurations, this technique enables the model to generalize across multiple orientations, significantly enhancing learning efficiency.

3.3 Details about the Implementation

3.3.1 Codebase

The code for the Rubik's cube is available at https://github.com/kvnjthms/rubiks.

3.3.2 Cube representation

The Rubik's Cube is represented as a $5 \times 5 \times 5$ tensor 3D tensor using PyTorch (Paszke et al. (2019)). Each of the six faces of the cube is embedded as a 3×3 patch on the outer boundary. These six patches correspond to the front, back, left, right, top, and bottom faces of the cube and are assigned distinct integer values to represent different face colors. The center $3 \times 3 \times 3$ region of the tensor is left empty and does not contribute to the cube's state.

The use of a padded $5 \times 5 \times 5$ structure—rather than the minimal $3 \times 3 \times 3$ representation—facilitates efficient and consistent manipulation of the cube's state during transformations. Each face rotation can be modeled as a slice-wise tensor rotation along one of the three spatial axes, without needing to explicitly track or remap adjacent faces. This allows cube moves to mimic the physical mechanics of a real Rubik's Cube while preserving tensor structure and symmetry.

Figure 3.1 shows how a 3×3 face is embedded in a 5×5 tensor.

3.3.3 Data Generation for Training

Training data is generated by applying a random sequence of up to 10 legal Rubik's Cube moves to the solved state, resulting in a scrambled cube configuration. For each generated state, the reverse of the final move in the sequence is recorded as the target label. This reverse move serves as a one-step corrective action that brings the cube

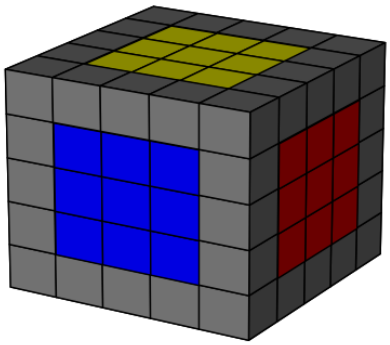

Figure 3.1: Cube faces embedded in a $5 \times 5 \times 5$ grid

closer to the solved state. Although this is not necessarily the globally optimal move, it is guaranteed to be a valid move on the path back to the solution and provides a consistent, easy-to-generate inputs.

This setup creates a one-step supervised learning task, where the network is trained to predict a single corrective move rather than solving the entire cube outright. To ensure dataset diversity and avoid redundant examples, each cube state is flattened and stored in a hash set to filter duplicates before inclusion.

While this approach is computationally efficient and allows for scalable data generation without external solvers, it has limitations. It does not expose the model to more complex solving strategies or multi-step dependencies. Additionally, it introduces bias toward recently applied moves, which may limit long-horizon planning capabilities. Despite this, the method provides a broad variety of cube states and helps promote generalization during early training.

3.3.4 Output Visualization

A visual representation of the cube can also be shown in the program. The diagram 3.2 shows a scrambled Rubik's Cube where each face contains a mixture of different colors.

The diagram 3.3 displays a solved cube, with each face uniformly colored. If the solve was unsuccessful, we will see a cube with colors in disoriented state.

Several models were explored: a basic 3D convolutional neural network (CNN), a deeper model with multiple convolutional layers, a ResNet-based architecture, and a group-equivariant model that incorporates the cube's symmetries. All these models

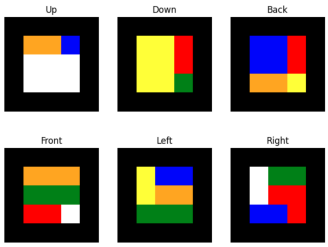

Figure 3.2: Scrambled Cube Representation

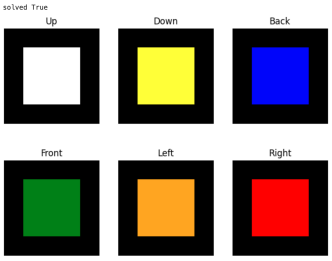

Figure 3.3: Solved Cube Representation

use the following parameters for training and evalulation 3.1.

Parameters	
Epochs	100
Training Set Size	100,000
Evaluation Set size	1000
Learning Rate	0.001
Number of Moves to Scramble Cube	Upto 10

Table 3.1: Training and Evaluation Configuration.

3.4 Implementation of 3D-Convolution

The neural network uses a 3D convolutional layer to process the Rubik's Cube, which is represented as a $5 \times 5 \times 5$ tensor with labeled face values. A single 3D convolutional layer with 16 filters captures spatial patterns from this input, followed by two fully connected layers for predicting the next move among 12 possible options. These 12 options represent clockwise and counter-clockwise rotations of each of the six faces of the Rubik's Cube: Right (R, RP), Left (L, LP), Up (U, UP), Down (D, DP), Front (F, FP), and Back (B, BP).

The model is trained using cross-entropy loss, where the output is a probability distribution over all 12 moves, and the true label is the correct move (in this case,

the reverse of the final scrambling action). Cross-entropy quantifies how well the predicted distribution aligns with the true move: it penalizes high confidence in incorrect moves and rewards confident predictions of the correct one. Gradients are computed using automatic differentiation and parameters are updated manually using gradient descent. Training proceeds over 100 epochs.

The evaluation accuracy and loss for a particular run is summarized in Figure 3.4.

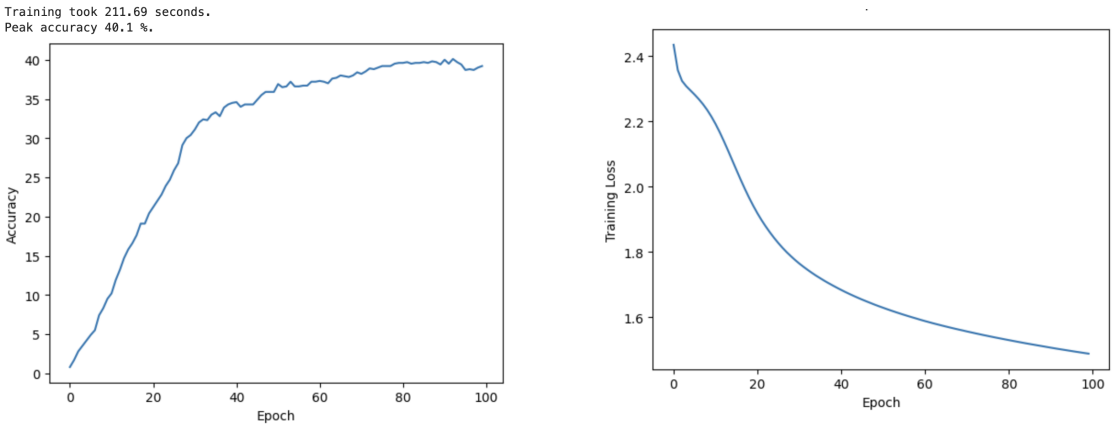

Figure 3.4: Prediction Accuracy and Training Loss with 3D-Convolution

In this run the model achieved a peak accuracy of approximately 40.1%. While this indicates that the network is learning meaningful patterns, the performance suggests that its current capacity may be insufficient to fully capture the complexity of the Rubik's Cube solving task. Further improvements could involve using deeper architectures, incorporating enhanced data augmentation, or leveraging more structured inductive biases such as group-equivariant convolutions.

3.5 Exploring Deeper Networks and ResNet Enhancements

To improve the Rubik's Cube solver's performance, modifications to the neural network architecture were explored, specifically by increasing the number of convolutional layers.

3.5.1 Increasing the Number of Convolutional Layers

In this phase of experimentation, the standard model was extended by stacking multiple 3D convolutional layers, with the goal of extracting higher-order spatial features and modeling complex cube transformations. Deeper networks theoretically allow the representation of intricate dependencies such as interactions between multiple faces and long-range patterns in scrambled states.

However, training the deeper architecture proved challenging. As shown in Figure 3.5, the model exhibited a long period of stagnation during the first 30 epochs, with no noticeable gain in accuracy and minimal reduction in training loss. This delay is indicative of optimization difficulties, possibly due to vanishing gradients in the early layers. Eventually, the model began to learn, reaching a peak accuracy of 45.4% with this run. As can be expected, the time for training is much longer compared to the previous model with a single convolution layer.

This result motivated the need for improved architectural elements such as skip connections using ResNet in deeper 3D convolutional networks.

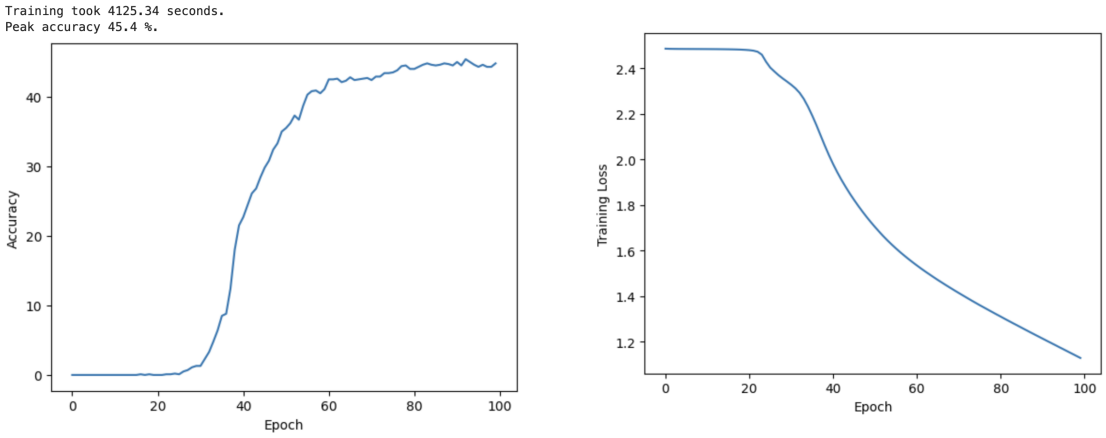

Figure 3.5: Prediction Accuracy and Training Loss with Multiple Convolution Layers.

3.5.2 Residual Network (ResNet) Implementation

To address the training instability observed in deeper networks, a ResNet-inspired architecture was implemented using 3D convolutional layers. Skip connections—paths were introduced through which the input to a layer can be added directly to its output.

The implemented residual blocks consist of two or more 3D convolutional layers with ReLU activations, followed by the addition of the block's input to its output. The Rubik's Cube input, structured as a $5 \times 5 \times 5$ tensor, passes through an initial convolution layer and several residual blocks, preserving spatial dimensions via appropriate padding. The final feature map is flattened and passed through a fully connected layer to produce a prediction over the 12 possible moves.

While the introduction of ResNet allowed the model to resume effective training, the observed improvement in evaluation accuracy was relatively modest compared to the CNN with multiple layers. The model is also slow in convergence as expected.

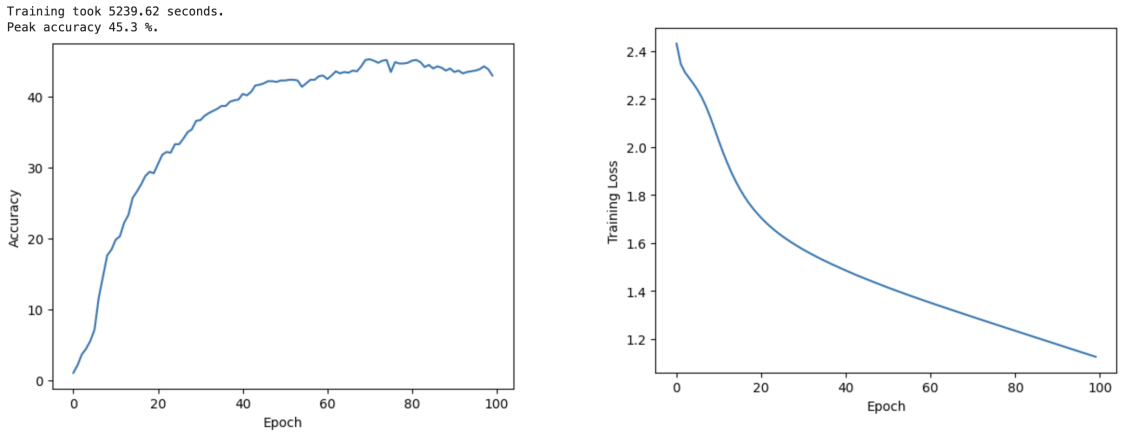

Figure 3.6: Prediction Accuracy and Training Loss with ResNet

3.6 G-CNN Implementation

The G-CNN implementation achieves equivariance to 3D rotations by transforming a single convolutional kernel into multiple rotated versions before applying convolution. Specifically, it begins with a base 3D kernel, which is rotated into six distinct orientations corresponding to the major rotational symmetries of a cube—representing different "up" faces. These rotations are performed in 90-degree increments around defined axes. For each of the six orientations, a separate 3D convolution is applied to the input. The resulting feature maps are then aggregated—via summation—to form a final feature representation that integrates responses across all rotational views. This aggregation preserves equivariance, allowing the network to produce consistent representations for rotated inputs.

Initially six orientations were considered because of the six faces of the cube. This was then extended to the complete 24 symmtries of the cube. Both of these are discussed in the sections below.

3.6.1 G-CNN with Six Orientations

This run demonstrates that incorporating six rotational symmetries leads to a modest improvement over the baseline multi-layer CNN. As shown in Figure 3.7, the G-CNN with six orientations achieved a peak accuracy of 46.7%. The average time this model takes in around half of what is taken by the deep models. More notably, the G-CNN stabilized around epoch 40, indicating more efficient convergence. These results suggest that encoding symmetry through equivariant convolution offers gains in both accuracy and training efficiency, even without additional depth.

3.6.2 G-CNN with 24 SO(3) Orientations

In addition to the six orientations discussed in the previous section, each orientation was rotated 4 times - $0°, 90°, 180°$ and $270°$. This gives the 24 orientations of the

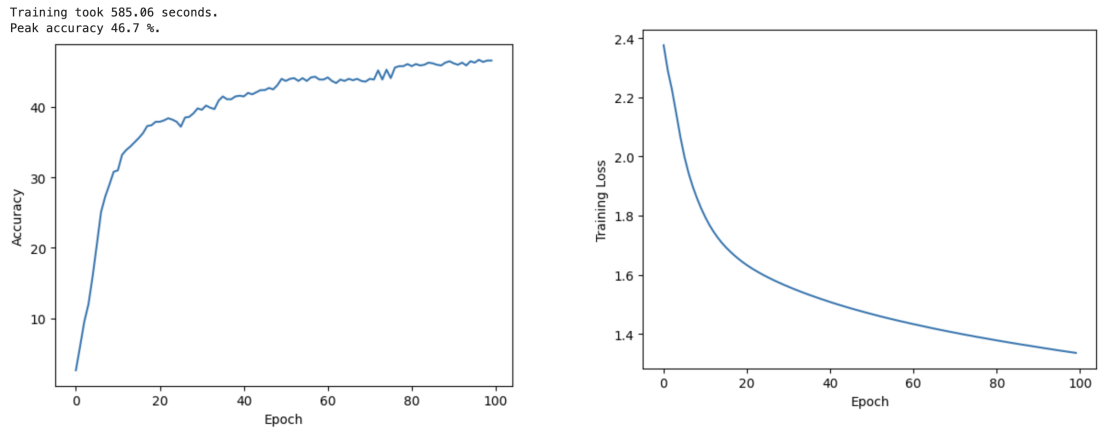

Figure 3.7: Prediction Accuracy and Training Loss with G-CNN and 6 symmetries

cube. The resulting accuracy and training times are shown in figure 3.8. It can be seen here too that the the G-CNN stabilized around epoch 40, indicating more efficient convergence.

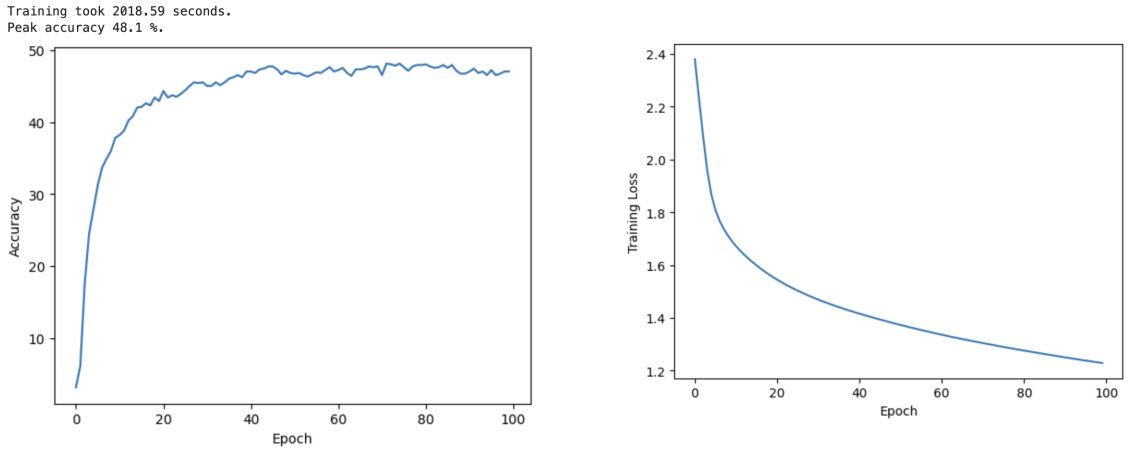

Figure 3.8: Prediction Accuracy and Training Loss with G-CNN and 24 symmetries

3.7 Comparison of Various Models

A training accuracy and time to train for the various methods is summarized in Table 3.2. These results are from 5 runs of each model[1]. While there is some variation across runs, the values shown here represent a typical outcome observed during experimentation.

[1] All experiments were executed locally using JupyterLab on a CPU-only laptop.

Network Type	Peak Accuracy (%)	Training Time Avg. (s)	Notes
3D-Conv 1 layer	42.10	228	
3D-Conv 5 layers	45.30	4794	Slow to start training.
3D-Conv 5 layers ResNet	49.50	5010	Addresses slow start problem.
G-CNN 6 symmetries	49.80	760	Faster convergance.
G-CNN 24 symmetries	48.90	2528	Faster convergance.

Table 3.2: Comparison of models on classification accuracy and training time.

3.8 Future work

This chapter presents an initial implementation of symmetry-aware neural networks for a Rubik's Cube solver. Like most deep learning models, there are several areas that could be explored further:

- Making sure the generated training data is optimal
- Trying alternative optimizers and training strategies to improve convergence and stability
- Tuning hyperparameters such as learning rate, batch size, layer depths, kernel sizes, activation functions and more
- Extending the model to predict multi-step move sequences rather than single corrections

These directions can help refine the current approach and support further study of how group equivariance impacts model performance.

3.9 Summary

This chapter presented the implementation of a Rubik's Cube solver using various deep learning architectures, informed by the cube's geometric structure. The cube was represented as a 3D tensor, and training data was generated using random move sequences paired with supervised labels. A standard 3D CNN was first implemented, followed by a ResNet-enhanced model to address training instability caused by vanishing gradients. Finally, a Group Convolutional Neural Network (G-CNN) was introduced to incorporate rotational symmetry directly into the network. Comparative evaluations showed that while ResNet improved stability, the G-CNN achieved better generalization and slightly higher accuracy. These results demonstrated the value of symmetry-aware architectures in learning structured spatial tasks.

Conclusion

This thesis examined the mathematical foundations of Convolutional Neural Networks (CNNs) and explored how those foundations can inform the design of models for solving structured problems such as the Rubik's Cube. Initially, the focus was on understanding how convolution operations achieve translational equivariance, and how this property can be formalized using circulant and shift matrices. The algebraic structure of these matrices, along with their polynomial representations, provided insight into how CNNs operate as shift-equivariant linear systems. This perspective helped clarify why CNNs are particularly effective on data with regular spatial structure.

Building on this mathematical foundation, the idea of extending equivariance to broader classes of transformations was introduced through the lens of geometric deep learning. Group Convolutional Neural Networks (G-CNNs) were studied as a way to incorporate group symmetries—such as rotations and reflections—directly into the architecture. The use of group actions and the rotation group $SO(3)$ offered a framework for designing models that respect the symmetries inherent in many physical and combinatorial systems, including the Rubik's Cube.

The theoretical ideas were applied to the Rubik's Cube by representing the cube as a 3D tensor and training a model to predict optimal moves in a supervised learning setting. A standard 3D CNN was implemented as a baseline, and its performance was evaluated over multiple training epochs. In an effort to improve this model, additional convolutional layers were added. However, this led to training instability due to the vanishing gradient problem. To address this, a residual network was implemented with skip connections that helped stabilize learning, though the performance improvement was relatively limited.

Subsequently, a G-CNN was implemented by rotating the convolutional kernel into six different orientations and aggregating the resulting feature maps. This was followed by the complete 24 SO(3) orientations. In summary, this work explored a progression of neural network architectures for Rubik's Cube solving, guided by mathematical insights into symmetry and equivariance. Starting with standard 3D convolutions, successive modifications including residual connections and group-equivariant convolutions were implemented. Each design provided a different perspective on how network structure interacts with the underlying properties of the task. While the im-

provements in accuracy were incremental, they were accompanied by gains in training efficiency. These results suggest that incorporating symmetry into model architecture is a reasonable direction to pursue when working with structured spatial data. Future work could explore combining rotation group of the cube with ResNet.

References

Agostinelli, F., McAleer, S., Shmakov, A., & Baldi, P. (2019). Solving the rubik's cube with deep reinforcement learning and search. *Nature Machine Intelligence*, *1*(8), 356–363.

Bengio, Y., Simard, P., & Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, *5*(2), 157–166.

Bronstein, M. (2020). Deriving convolution from first principles. *Experfy Insights*. Accessed: 2025-03-11. `https://resources.experfy.com/ai-ml/deriving-convolution-from-first-principles/`

Bronstein, M. (2023). Deriving convolution from first principles - Experfy insights. `https://resources.experfy.com/ai-ml/deriving-convolution-from-first-principles/`

Bronstein, M. M., Bruna, J., Cohen, T., & Veličković, P. (2021a). Geometric deep learning. `https://geometricdeeplearning.com`. Accessed: 2025-04-22.

Bronstein, M. M., Bruna, J., Cohen, T., & Veličković, P. (2021b). Geometric deep learning: Grids, groups, graphs, geodesics, and gauges. https://arxiv.org/abs/2104.13478.

Bronstein, M. M., Bruna, J., LeCun, Y., Szlam, A., & Vandergheynst, P. (2017). Geometric deep learning: Going beyond euclidean data. *IEEE Signal Processing Magazine*, *34*(4), 18–42. `http://dx.doi.org/10.1109/MSP.2017.2693418`

Cohen, T. S., & Welling, M. (2016). Group equivariant convolutional networks. https://arxiv.org/pdf/1602.07576. `https://arxiv.org/abs/1602.07576`

GL Academy (2025). Non-convex optimization: Utilizing stochastic gradient descent to find a local optimum. Accessed: 2025-03-11. `https://dtmvamahs40ux.cloudfront.net/gl-academy/course/course-1281-Non-convex-optimization-We-utilize-stochastic-gradient-descent-to-find-a-local-optimum.jpg`

Gottlieb, Y., & Burch, N. (2019). Learning to solve combinatorial optimization problems on graphs: A graph neural network approach. *arXiv preprint arXiv:1906.07391*.

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, (pp. 770–778).

Kra, I., & Simanca, S. R. (2012). On circulant matrices. *Notices of the American Mathematical Society*, *59*(3), 368–377. https://www.ams.org/notices/201203/rtx120300368p.pdf

Kromydas, B. (2023). Convolutional Neural Network (CNN): A Complete Guide — learnopencv.com. https://learnopencv.com/understanding-convolutional-neural-networks-cnn/. [Accessed 03-11-2024].

Martins, C. (2023). Understanding residual connections in neural networks. https://cdanielaam.medium.com/understanding-residual-connections-in-neural-networks-866b94f13a22

McAleer, S., Agostinelli, F., Shmakov, A., & Baldi, P. (2018). Solving the rubik's cube without human knowledge. *arXiv preprint arXiv:1805.07470*. Accessed: 2025-03-11. https://arxiv.org/abs/1805.07470

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., & Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. https://pytorch.org/

Rubik, E. (1974). Rubik's cube. https://rubiks.com.

SuperDataScience Team (2021). Convolutional neural networks (cnn): Step 4 - full connection. Accessed: 2025-03-11. https://www.superdatascience.com/blogs/convolutional-neural-networks-cnn-step-4-full-connection

van der Pol, E. (2021). Geometric deep learning and reinforcement learning. Seminar presented at the African Master's in Machine Intelligence (AMMI) Geometric Deep Learning Course. https://www.elisevanderpol.nl/posts/talk2021b/

Van Engelenburg, C. (2020). Geometric Deep Learning: group equivariant convolutional networks. https://medium.com/swlh/geometric-deep-learning-group-equivariant-convolutional-networks-ec687c7a7b41

Yani, M., Irawan, S., & Setianingsih, C. (2019). Application of transfer learning using convolutional neural network method for early detection of terry's nail. *Journal of Physics: Conference Series*, *1201*, 012052.