

Bachelor Mathematics

*Bachelor thesis*

---

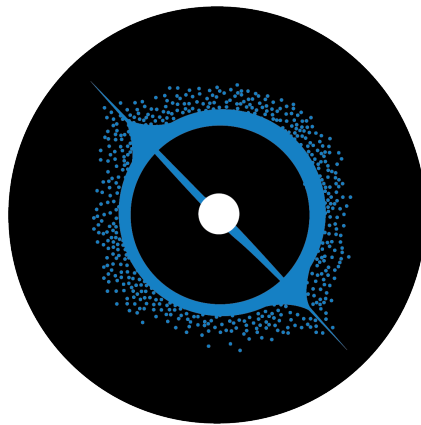
# Convolutional Neural Networks for Galaxy Morphological Classification

---

by

Antón Fidalgo Pérez

Supervisor: dr. Hannah Rocio Santa Cruz Baur



Department of Mathematics  
Faculty of Science



# Abstract

Since the dawn of humanity, we have shown a timeless fascination with the heavens, seeking to understand the universe that surrounds us. This innate curiosity has driven centuries of observation, theory, and exploration.

In today's age of data-intensive science, the study of galaxy morphology plays a fundamental role in understanding the universe. Morphology, from the Greek *morphé*, refers to the classification of galaxies based on their visual structure, such as elliptical, spiral, or irregular galaxies, and provides key insights into large-scale physical processes.

The objective of this thesis is to apply Convolutional Neural Networks to the problem of galaxy morphological classification using astronomical images. A mathematical and computational approach is proposed to enhance the accuracy, scalability, and automation of this task, which has traditionally been performed manually.

This work aims to contribute a complementary tool for the astronomical community that can accelerate the analysis of large-scale datasets from modern sky surveys, and thus support humanity's ongoing effort to understand the universe.

Title: Convolutional Neural Networks for Galaxy Morphological Classification

Author: Antón Fidalgo Pérez, a.fidalgo.perez@student.vu.nl, 2783715

Supervisor: dr. Hannah Rocio Santa Cruz Baur

Department of Mathematics  
Vrije Universiteit Amsterdam  
de Boelelaan 1081, 1081 HV Amsterdam  
<http://www.math.vu.nl/>

# Contents

<b>Nomenclature</b>	<b>4</b>
<b>1 Introduction</b>	<b>5</b>
<b>2 Theoretical Foundations</b>	<b>8</b>
2.1 Mathematical Preliminaries . . . . .	8
2.2 Symmetries, Representations, and Invariance . . . . .	11
<b>3 Convolutional Neural Networks</b>	<b>17</b>
3.1 Formal Definitions . . . . .	17
3.2 The Classification Journey . . . . .	21
3.3 Implementation Details . . . . .	25
<b>4 Discussion</b>	<b>42</b>
<b>5 Conclusion</b>	<b>44</b>
<b>Appendix</b>	<b>45</b>
<b>Bibliography</b>	<b>46</b>

# Nomenclature

$G$  : A group of transformations, such as translations or rotations.

$g \in G$  : A transformation element from the group  $G$ .

$\Omega \subset \mathbb{Z}^2$  : The discrete two-dimensional image domain.

$x \in X(\Omega, \mathbb{C})$  : A signal or image defined over  $\Omega$ , with values in a channel space  $\mathbb{C}$ .

$\mathbb{C}$  : The channel space, for example,  $\mathbb{R}$  for grayscale or  $\mathbb{R}^3$  for RGB images.

$T_{\mathbf{v}}$  : The translation operator by a vector  $\mathbf{v} \in \mathbb{Z}^2$ , defined by  $T_{\mathbf{v}}x(u) = x(u - \mathbf{v})$ .

$\rho(g)$  : The action of a group element  $g \in G$  on a signal  $x$ , written as  $\rho(g)x$ .

$f : \mathcal{X}(\Omega) \rightarrow \mathcal{Y}$  : A function, such as a neural network, mapping signals to outputs.

$G - invariant function$  : A function satisfying  $f(\rho(g)x) = f(x)$  for all  $g \in G$ .

$G - equivariant function$  : A function satisfying  $f(\rho(g)x) = \rho(g)f(x)$  for all  $g \in G$ .

$*$  : The convolution operation, defined by  $(x * \theta)(u) = \sum_v x(v) \theta(u - v)$ .

$\theta$  : The convolution kernel, applied locally over the input.

$S_{\mathbf{v}}$  : A shift operator acting on a function as  $S_{\mathbf{v}}x(u) = x(u - \mathbf{v})$ .

$\mathbb{Z}_n$  : The cyclic group of order  $n$ , representing discrete rotations.

$\sigma$  : An activation function applied elementwise to introduce nonlinearity.

# 1 Introduction

*“Equipped with his five senses, man explores the universe around him and calls the adventure Science.”*

— Edwin Hubble [1]

From the prehistoric structures of Stonehenge, carefully aligned with the movements of the sun and moon, to the observations of eclipses and celestial bodies in ancient Mesopotamia, humanity’s fascination with the heavens comes from the beginning of humankind. Early civilizations tracked the stars and planets not only for agricultural and navigational purposes but also to understand their place in the universe and to connect with the divine. This fascination evolved into more systematic and philosophical inquiries, as seen in the cosmological models of ancient Greece, where thinkers like Ptolemy and Aristotle attempted to explain the structure and mechanics of the heavens.

The Renaissance reignited these pursuits with a spirit of empirical observation, marking a shift in how humanity comprehended the cosmos, not as a distant, unchanging realm governed by mystical forces, but as a dynamic and knowable system. This curiosity continued to inspire our modern exploration of space, as we seek to discover the mysteries of the universe and our place within it [2].

The study of galaxy morphology originated nearly a century ago, at a time when galaxies were still referred to as *nebulae*. In the 18th and 19th centuries, astronomers used the term nebula to describe any diffuse astronomical object, including both gas clouds within the Milky Way and the unresolved, misty shapes visible in early telescopic observations. These so-called spiral or elliptical nebula provoked scientific curiosity due to their diverse and often symmetric shapes. However, it was not until the early 20th century, particularly following Edwin Hubble’s observations of Cepheid variables in the Andromeda nebula in the 1920s, that astronomers realized these objects were not local to our galaxy but were, in fact, entire galaxies: enormous collections of stars and gas held together by gravity, existing far outside our own galaxy [3, 4].

This major conceptual shift transformed our understanding of the universe and initiated the systematic study of galaxy morphology, which involves classifying and interpreting the wide variety of shapes and structures found in galaxies. Early astronomers began this work through visual observation, establishing a framework that would later become a foundation of extragalactic astronomy [4].

Several classification systems were proposed over time, but most fell out of use. The system that endured was developed by Edwin Hubble in the 1920s and 1930s. He categorized galaxies based on their visual morphology into ellipticals, spirals, and a transitional class he called lenticulars,  $S_0$ . This classification was famously illustrated in his "tuning fork" diagram, which organized galaxies along a sequence from round ellipticals to increasingly flattened spirals, and further divided spirals into barred and unbarred types [4].

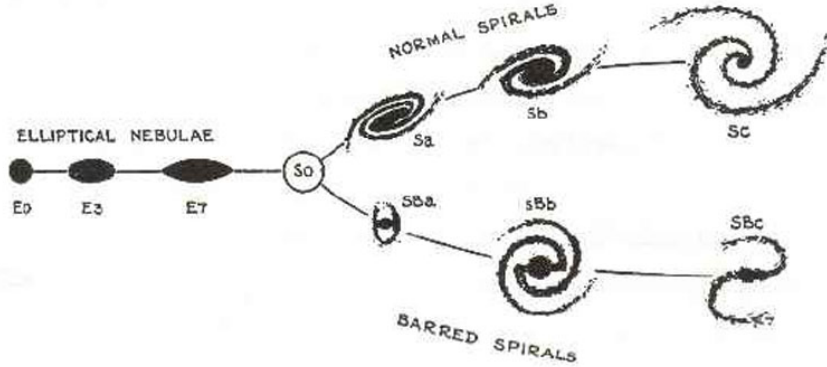


Figure 1.1: Hubble's Original Tuning Fork Diagram [1].

Hubble's approach deliberately avoided overcomplicating the scheme with excessive detail, allowing most of observed galaxies to be divided into broad, practical categories. This visual system was later extended by Sandage (1961) [6] and de Vaucouleurs (1959) [7], and it continues to form the foundation of galaxy classification in modern extragalactic astronomy, particularly for galaxies in the nearby universe [4].

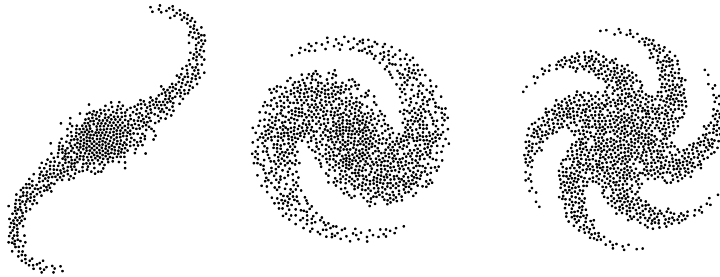
Classifying galaxy types is crucial because understanding their structure allows astronomers to trace how galaxies form, evolve, and interact over cosmic time. Morphological classification allows identifying key physical processes, such as star formation, mergers, or internal dynamics and helps connect individual galaxies to the broader context of cosmic evolution [5].

Galaxy classification has traditionally been a manual process, where trained observers visually inspect images and assign galaxies to morphological categories based on features such as shape, structure, presence of bars or rings, and other characteristics. This method, while subjective, allows for detailed classification and has been applied successfully to tens of thousands of galaxies in major studies [5].

Large-scale efforts like the Galaxy Zoo project [8, 9] expanded this manual approach to include public participation, enabling visual classification of over a million galaxies. However, the enormous volume of data produced by modern sky surveys has driven the development of automated classification techniques. These methods often rely on machine learning algorithms trained on visually classified samples, to replicate human-level decisions in a consistent and scalable way. While still developing, automated systems aim to bring objectivity and efficiency to the classification of vast numbers of galaxies across large datasets.

With this research, we aim to contribute to the development of automated systems for galaxy classification, enabling the efficient and precise analysis of vast astronomical datasets. To achieve this, we will use Convolutional Neural Networks, a machine learning model that has demonstrated exceptional performance in image classification due to its ability to detect and exploit symmetries within visual data.

Our approach will be grounded in a mathematical perspective. We will begin by introducing the foundations of group theory as a basis for understanding CNNs. This theoretical framework will allow us to explore the mathematical principles that underlie the functioning of CNNs. Ultimately, our goal is to develop a CNN model capable of accurately classifying large volumes of galaxy images, adding a small piece to the puzzle of the universe.



## 2 Theoretical Foundations

This section connects the ideas of geometric deep learning with the practical use of Convolutional Neural Networks for classifying galaxy shapes. CNNs work well for image classification because they can take advantage of patterns and symmetries in the data, an idea that comes from group theory. To support this, we introduce key group theory concepts used throughout the thesis. These concepts help explain how a neural network can remain unchanged, invariant, or change predictably, equivariant, when the input image is shifted or rotated, which is crucial since galaxies may appear in different positions or orientations in images.

### 2.1 Mathematical Preliminaries

The notation and theoretical framework adopted in this section follow the conventions and terminology presented in [11, 12].

**Definition 2.1** (Group). A *group*  $G$  is a set of transformations with a binary operation  $\cdot$  satisfying the following axioms:

- **Closure:** For all  $a, b \in G$ , we have  $a \cdot b \in G$ .
- **Associativity:**  $(a \cdot b) \cdot c = a \cdot (b \cdot c)$  for all  $a, b, c \in G$ .
- **Identity:** There exists an element  $e \in G$  such that  $a \cdot e = e \cdot a = a$  for all  $a \in G$ .
- **Inverse:** For each  $a \in G$ , there exists an inverse  $a^{-1} \in G$  such that  $a \cdot a^{-1} = e$ .

In the context of CNNs, group elements can represent transformations such as translations or rotations of an image.  $\mathbb{Z}_n$  refers to the cyclic group of integers modulo  $n$ . It consists of the set  $\mathbb{Z}_n = \{0, 1, 2, \dots, n-1\}$ , with the group operation being addition modulo  $n$ . Formally, for any  $a, b \in \mathbb{Z}_n$ , their group operation is  $a + b \pmod n$ .

**Example 2.2** (Translation). In the figure below, we illustrate a translation using two versions of the same galaxy image. The galaxy shifts within the frame, but its local structure remains unchanged. This observation is crucial in the design of CNNs, which should not be altered by translations. By learning features that are not dependent on an object’s position, CNNs can recognize patterns regardless of their location within the image.



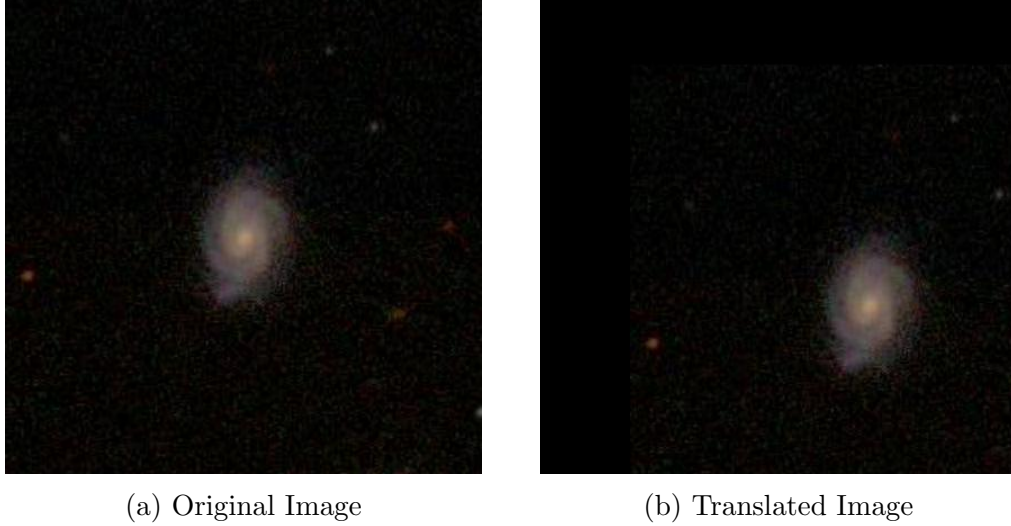


Figure 2.1: Translation of a Galaxy [9, 10]

**Definition 2.3** (Subgroup). A subset  $H \subseteq G$  is called a *subgroup* if it forms a group under the same binary operation. Formally,  $H \leq G$  if:

- $H \neq \emptyset$ ,
- For all  $x, y \in H$ ,  $x \cdot y^{-1} \in H$ .

In practice, this might mean considering only translations and ignoring rotations in a symmetry group.

**Definition 2.4** (Generator). A *generator* of a group is an element (or set of elements) from which all other elements of the group can be derived through repeated composition. For a single generator  $g$ , we write:

$$\langle g \rangle = \{g^n \mid n \in \mathbb{Z}\}.$$

Horizontal and vertical translations can generate the full translation group on a 2-dimensional image grid.

**Definition 2.5** (Abelian Group). A group  $G$  is called *Abelian* (or commutative) if:

$$a \cdot b = b \cdot a \quad \text{for all } a, b \in G.$$

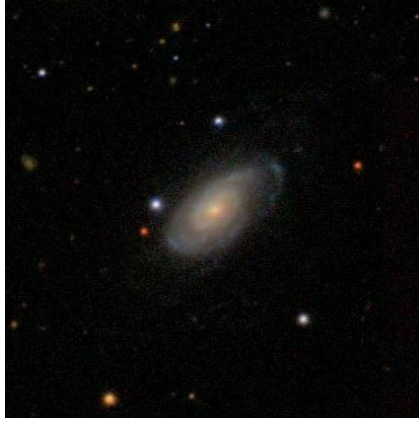
Translations in Euclidean space form an Abelian group. This property simplifies the design of certain CNN architectures.

**Definition 2.6** (Group Action). Let  $G$  be a group and  $A$  a set. A *group action* of  $G$  on  $A$  is a function  $G \times A \rightarrow A$ ,  $(g, a) \mapsto g \cdot a$ , satisfying the following properties for all  $g_1, g_2 \in G$  and  $a \in A$ :

- $g_1 \cdot (g_2 \cdot a) = (g_1 \cdot g_2) \cdot a$ ,
- $e \cdot a = a$ , where  $e$  is the identity element of  $G$ .

**Definition 2.7** (Homomorphism and Isomorphism). Given two groups  $G, H$  and respective group actions, a *homomorphism* is a structure-preserving map  $\phi : G \rightarrow H$  such that  $\phi(a \cdot b) = \phi(a) \cdot \phi(b)$ . An *isomorphism* is a bijective homomorphism, meaning the two groups are structurally identical.

**Definition 2.8** (Orbit). Let  $G$  be a group acting on a set  $X$ . The *orbit* of an element  $x \in X$  under the action of  $G$  is the set  $\text{Orbit}(x) = \{g \cdot x \mid g \in G\}$ .



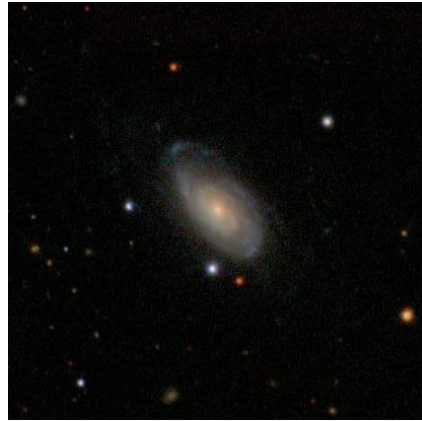
(a) Original Image



(b) 90° Rotation



(c) 180° Rotation



(d) 270° Rotation

Figure 2.2: Orbit of a Galaxy under the Rotation Group  $\mathbb{Z}_4$  [9, 10]

**Example 2.9.** The images above show the progression of a rotation group action on an image of a galaxy. Each image has been rotated by 90 degrees in a clockwise direction, and the four positions represent the orbit of the original image under the rotation group  $\mathbb{Z}_4$ . The four images ( $0^\circ$ ,  $90^\circ$ ,  $180^\circ$ ,  $270^\circ$ ) form an orbit under the group action of rotations, where each element of the group corresponds to a distinct rotation.

In this section, we introduced some basic concepts from group theory to help describe how transformations like rotations and translations work. These ideas will help us understand how neural networks, especially CNNs, work. The next chapters will build on this foundation to show how these ideas are used in real models.

## 2.2 Symmetries, Representations, and Invariance

A core strength of Convolutional Neural Networks lies in their ability to exploit symmetries in data. In the context of galaxy morphology classification, a particularly relevant example is their translation equivariance. This property ensures that if a galaxy shifts to a different location within the image, the CNN's internal representation shifts accordingly, allowing the network to detect features consistently across the domain. Such behavior is essential in astronomical data analysis, where galaxies can appear anywhere in the frame, and precise centering is not always guaranteed.

In figure 2.1 above, we illustrate translational invariance using two versions of the same galaxy image. Although the galaxy's position changes within the frame, its morphological features remain identical. CNNs exploit this symmetry by applying shared filters across spatial locations, ensuring consistent feature detection regardless of object position.

Formally, let  $x \in X(\Omega, \mathbb{C})$  be a signal representing an image defined on a discrete grid  $\Omega \subset \mathbb{Z}^2$ , with values in a channel space  $\mathbb{C}$ . A translation by a vector  $\mathbf{v} \in \mathbb{Z}^2$  acts on the signal via the group action:

$$T_{\mathbf{v}}x(u) = x(u - \mathbf{v}), \quad \forall u \in \Omega.$$

Here,  $T_{\mathbf{v}}$  denotes the translation operator, and  $u$  is a pixel coordinate. As shown in Figure 2.1, the same spiral galaxy is presented in two locations: one centered, and one translated by  $\mathbf{v} = (80, -60)$ . The features of the galaxy remain perceptually and structurally identical.

In the context of geometric deep learning, we consider a domain  $\Omega$ , which could represent, for example, the pixel grid of an image. A function defined on this domain assigns a value (such as brightness or color) to each pixel, effectively turning the image into a mathematical object. To illustrate this, consider the image in Figure 2.3.

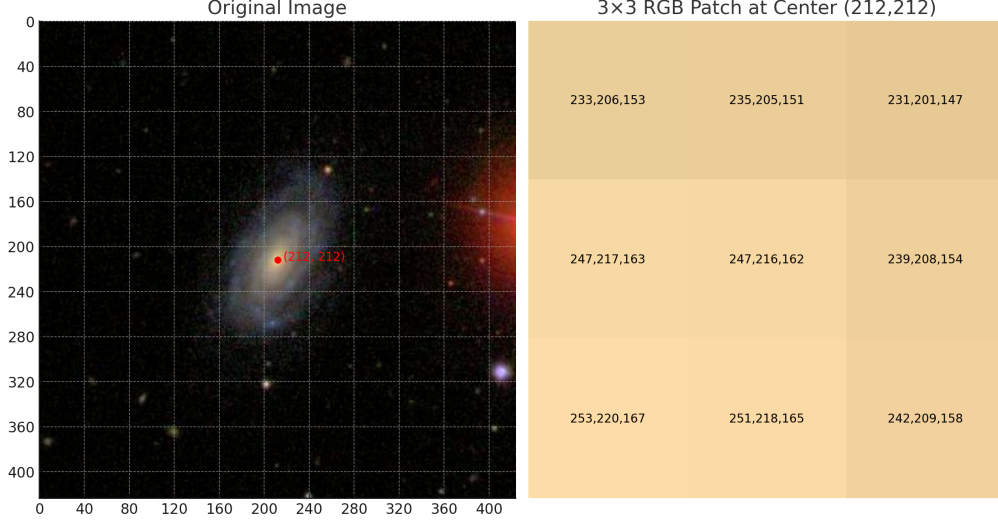


Figure 2.3: Image domain  $\Omega$  as a grid of pixel coordinates [9, 10].

On the left, we show a galaxy image, and on the right, a zoomed-in  $3 \times 3$  patch centered at pixel (212,212), where each pixel is annotated with its RGB values. We define the domain as  $\Omega = \{(x, y) \mid 0 \leq x < 424, 0 \leq y < 424\}$  and the corresponding function as  $f : \Omega \rightarrow \mathbb{R}^3$ , where:

$$f(x, y) = \begin{bmatrix} R(x, y) \\ G(x, y) \\ B(x, y) \end{bmatrix}$$

maps each pixel to its red, green, and blue intensity values. For instance, the function at the center of the patch evaluates to:

$$f(212, 212) = \begin{bmatrix} 247 \\ 216 \\ 162 \end{bmatrix}.$$

Having established how we represent an image as a function over a discrete domain, we can now examine how such functions behave under transformations of the domain. Specifically, we are interested in how functions respond when a group  $G$  acts on  $\Omega$  (for example, by translating or rotating the pixel grid). Two key properties in this setting are invariance and equivariance, which describe whether and how the function's output changes when its input is transformed.

**Definition 2.10** ( $G$ -invariance). A function  $f : \mathcal{X}(\Omega) \rightarrow \mathcal{Y}$  is called  $G$ -invariant if  $f(\rho(g)x) = f(x)$  for all  $g \in G$ , where  $\rho(g)$  denotes the action of the group element  $g$  on the signal  $x$ . Invariant functions produce the same output regardless of how the input is transformed under  $G$ .

**Definition 2.11** ( $G$ -equivariance). A function  $f : \mathcal{X}(\Omega) \rightarrow \mathcal{X}(\Omega)$  is called  $G$ -equivariant if  $f(\rho(g)x) = \rho(g)f(x)$  for all  $g \in G$ . In this case, the output of  $f$  transforms in the same way as the input under the group action.

Convolutional Neural Networks are often mistakenly referred to as translation-invariant. In reality, CNNs are translation-equivariant: translating the input results in a corresponding translation in the output feature maps. This is because convolution is compatible with the action of translation: convolving a function with another does not destroy the structure of translations but rather preserves it in a predictable way.

**Definition 2.12** (Convolution). The *convolution* of a function  $x$  with a kernel  $\theta$  is defined as

$$(x * \theta)(u) = \sum_v x(v) \theta(u - v)$$

where  $x$  is the input signal (e.g., a matrix),  $\theta$  is the convolution kernel (or filter), and  $u$  is the output position.

**Lemma 2.13** (Linearity of Convolution). Let  $x_1, x_2$  be discrete signals, let  $\theta$  be a kernel, and let  $a, b \in \mathbb{R}$ . Then convolution is linear:

$$a(x_1 * \theta)(u) + b(x_2 * \theta)(u) = ((ax_1 + bx_2) * \theta)(u)$$

for all  $u \in \mathbb{Z}$ .

*Proof.* By the definition of convolution, we have

$$(x_1 * \theta)(u) = \sum_{v \in \mathbb{Z}} x_1(v) \theta(u - v), \quad (x_2 * \theta)(u) = \sum_{v \in \mathbb{Z}} x_2(v) \theta(u - v).$$

Therefore,

$$\begin{aligned} a(x_1 * \theta)(u) + b(x_2 * \theta)(u) &= a \sum_{v \in \mathbb{Z}} x_1(v) \theta(u - v) + b \sum_{v \in \mathbb{Z}} x_2(v) \theta(u - v) \\ &= \sum_{v \in \mathbb{Z}} (ax_1(v) + bx_2(v)) \theta(u - v) = ((ax_1 + bx_2) * \theta)(u). \end{aligned}$$

□

**Theorem 2.14** (Translation Equivariance of Convolution). *The convolution operation is translation-equivariant. That is, shifting the input results in a corresponding shift in the output feature maps. Formally, if the convolution is defined as*

$$(x * \theta)(u) = \sum_v x(v)\theta(u - v),$$

*then for any shift  $S_v$  by  $v$ , the following holds:*

$$S_v(x * \theta) = (S_v x) * \theta,$$

*where  $S_v x(u) = x(u - v)$  denotes the shifted signal. Thus, convolution commutes with shifts, ensuring translational-equivariance.*

*Proof.* Let  $x$  be the input signal and  $\theta$  the convolution kernel. The convolution of  $x$  and  $\theta$  at position  $u$  is given by:

$$(x * \theta)(u) = \sum_v x(v)\theta(u - v).$$

Now, to demonstrate translation-equivariance, we apply a shift  $S_v$  to the output feature map  $(x * \theta)$ . A shift by  $v$  means that for any position  $u$ , the new output at position  $u$  is the output of the original convolution evaluated at  $u - v$ :

$$S_v(x * \theta)(u) = (x * \theta)(u - v).$$

Substituting the definition of convolution, we obtain:

$$S_v(x * \theta)(u) = \sum_{v'} x(v')\theta((u - v) - v').$$

Simplifying the expression inside the kernel:

$$S_v(x * \theta)(u) = \sum_{v'} x(v')\theta(u - (v + v')).$$

Now, consider shifting the input signal  $x$  by  $v$ , which gives  $(S_v x)(u) = x(u - v)$ . The convolution of the shifted signal with  $\theta$  is:

$$(S_v x) * \theta(u) = \sum_{v'} (S_v x)(v')\theta(u - v').$$

Since  $(S_v x)(v') = x(v' - v)$ , we have:

$$(S_v x) * \theta(u) = \sum_{v'} x(v' - v)\theta(u - v').$$

Let us perform a change of variables by setting  $v'' = v' - v$ , so that  $v' = v'' + v$ . Substituting, we get:

$$(S_v x) * \theta(u) = \sum_{v''} x(v'') \theta(u - (v + v'')).$$

Thus, we find:

$$S_v(x * \theta)(u) = (S_v x) * \theta(u).$$

Since both the left-hand side and right-hand side are identical, we have shown that shifting the output of the convolution is equivalent to first shifting the input and then performing the convolution. This proves that the convolution operation is equivariant under shifts.

□

**Definition 2.15** (Tensor). A *tensor* is a multidimensional array of numerical values that generalizes the concepts of scalars, vectors, and matrices. Specifically:

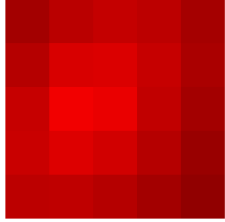
- A **scalar** is a 0-dimensional tensor (a single number).
- A **vector** is a 1-dimensional tensor (e.g.,  $[x_1, x_2, x_3]$ ).
- A **matrix** is a 2-dimensional tensor (a rectangular array of numbers).
- Higher-dimensional tensors extend this idea to more axes.

In the context of image data:

- A grayscale image is a 2-dimensional tensor, where each entry corresponds to a pixel intensity.
- An RGB image is a 3-dimensional tensor of shape  $(H, W, 3)$ , where  $H$  and  $W$  are the height and width, and 3 corresponds to the red, green, and blue channels.

**Example 2.16.** To demonstrate how image data can be treated as a structured numerical object, we begin with an example from our dataset. Each galaxy image is converted to an array, representing images as 3-dimensional tensors.

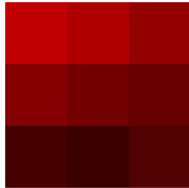
For simplicity, we considered only the red channel of the image in Figure 2.3. This gives us a 2D array where each entry corresponds to the red intensity of a pixel. Below is the  $5 \times 5$  patch from the center of the red channel of the image:

$$\text{Patch} = \begin{bmatrix} 0.64 & 0.73 & 0.77 & 0.74 & 0.65 \\ 0.71 & 0.85 & 0.86 & 0.78 & 0.67 \\ 0.78 & 0.95 & 0.92 & 0.76 & 0.64 \\ 0.79 & 0.87 & 0.82 & 0.71 & 0.60 \\ 0.74 & 0.75 & 0.71 & 0.64 & 0.57 \end{bmatrix}$$


To this patch, we apply a simple convolution operation. Convolution involves sliding a small matrix, called a kernel, across the patch and computing a weighted sum of overlapping values. This is a basic operation in many image processing and machine learning models. In our case, we use the following  $3 \times 3$  kernel:

$$\text{Kernel} = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

This kernel emphasizes changes in intensity from top to bottom. Applying this kernel to the  $5 \times 5$  patch, we obtain the following  $3 \times 3$  output:

$$\text{Output} = \begin{bmatrix} 0.51 & 0.39 & 0.16 \\ 0.06 & -0.09 & -0.18 \\ -0.45 & -0.53 & -0.40 \end{bmatrix}$$


Each number in the output represents the result of placing the kernel over a  $3 \times 3$  region of the patch and computing a weighted sum. This example shows how we can move from raw image data to numerical patterns that reveal local structure in the image, an idea that forms the basis for more complex operations in Convolutional Neural Networks.

In this chapter, we have introduced the main ideas needed to understand how symmetry, group actions, and functional representations on discrete spaces provide a strong basis for modeling image data. By discussing the difference between invariance and equivariance, and explaining how convolution behaves under translation, we have shown how concepts from group theory support many modern methods in image analysis. With this foundation, we can now take a closer look at Convolutional Neural Networks, to see how their structure is based on these mathematical ideas and how they learn to build hierarchical and spatially meaningful representations.



# 3 Convolutional Neural Networks

A Convolutional Neural Network is a type of neural network that is especially well-suited for processing data such as images. Instead of treating the input as a one-dimensional vector, it uses the two-dimensional structure of the image to apply small filters, kernels, that detect local patterns. Kernels are applied across the image in a consistent way, allowing the network to recognize features regardless of their position. In mathematical terms, it can be described as a sequence of layers, each transforming the input data into a more abstract and useful representation.

## 3.1 Formal Definitions

In this section, we provide the formal definitions and mathematical descriptions of the components that constitute a Convolutional Neural Network. We follow the formalism developed in [11], with additional images and definitions from [13].

**Definition 3.1** (Layer). A *layer* in a neural network is a function  $f_\ell : \mathcal{X}_{\ell-1} \rightarrow \mathcal{X}_\ell$  that maps an input tensor to an output tensor.

Layers may perform operations such as convolution, explained in previous sections, or activation, pooling, or normalization, which will be defined later. These operations transform the input data, extract relevant features, and help the network to capture complex structures.

**Definition 3.2** (CNN Architecture). A *CNN architecture* is a function  $f_\Theta : \mathcal{X}_0 \rightarrow \mathcal{Y}$ , defined as a composition of layers:

$$f_\Theta = f_L \circ f_{L-1} \circ \cdots \circ f_1,$$

where each  $f_\ell$  may involve operations such as convolution, nonlinearity (activation functions), and subsampling (e.g., pooling). The parameter set  $\Theta = \{\theta^{(1)}, \dots, \theta^{(L)}\}$  consists of all learnable weights and biases across the layers.

**Definition 3.3** (Perceptron). A *perceptron* is a function  $y = f(x) = w^\top x + b$ , where  $x \in \mathbb{R}^n$  is the input,  $w \in \mathbb{R}^n$  is a weight vector, and  $b \in \mathbb{R}$  is a bias term. In its basic form, it outputs a linear combination of its inputs.

**Example 3.4.** Consider a simple perceptron, illustrated in the figure below. This structure computes the output  $y$  directly from the inputs, without any intermediate processing:

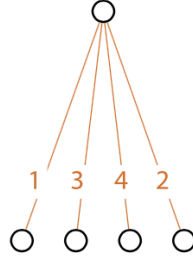


Figure 3.1: Simple Perceptron [13]

The function represented by this simple perceptron is:  $y = 1x_1 + 3x_2 + 4x_3 + 2x_4$ . At first glance, one might expect that composing multiple perceptrons would result in a more powerful model. However, this is not necessarily the case. Building on our previous example, consider the following composite perceptron:

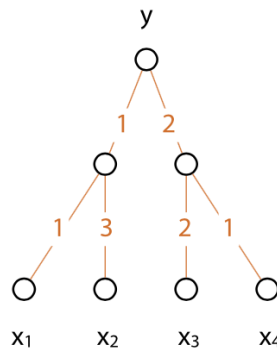


Figure 3.2: Composite Perceptron [13]

The function computed by this composite structure is:  $y = 1(1x_1 + 3x_2) + 2(2x_3 + 1x_4)$ . After simplification, this still results in a linear function. This illustrates that composing perceptrons without non-linear activation functions does not increase the power of the network, the overall function remains linear. Therefore, an important concept in neural networks is the activation function, which enables models to learn non-linear patterns in data. By introducing nonlinearity, activation functions give neural networks the ability to represent complex structures in data, something essential for solving real-world tasks like image recognition.

**Definition 3.5** (Activation Function). An *activation function* is a non-linear function  $\sigma : \mathbb{R} \rightarrow \mathbb{R}$  that is applied elementwise to the output of a layer in a neural network. Given an input vector  $z = (z_1, z_2, \dots, z_n)$ , the activation function produces  $\sigma(z) = (\sigma(z_1), \sigma(z_2), \dots, \sigma(z_n))$ .

Two commonly used activation functions are:

The Sigmoid function is defined as  $\sigma(z) = (1 + e^{-z})^{-1}$  which maps real-valued inputs to the interval  $(0, 1)$ . It is often used in binary classification and scenarios requiring probabilistic interpretation.

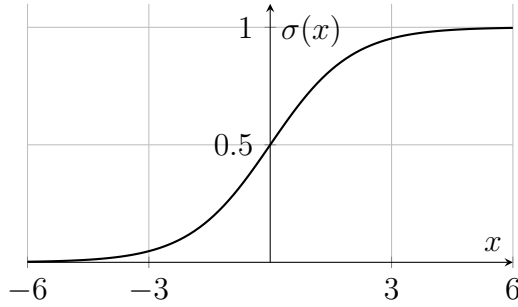


Figure 3.3: Sigmoid Activation Function

The ReLU function is defined as  $\sigma(z) = \max(0, z)$ , which outputs zero for negative inputs and returns the input itself for positive values. It is computationally efficient and helps mitigate the vanishing gradient problem.

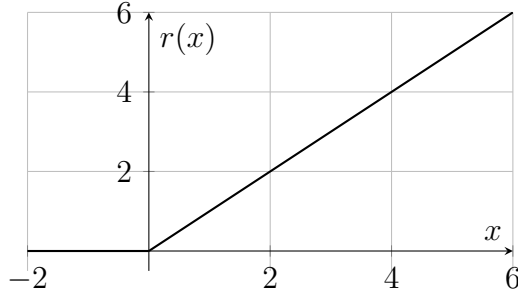


Figure 3.4: ReLU Activation Function

**Definition 3.6** (Pooling Layer). A *pooling layer* is a downsampling operation applied to feature maps to reduce their spatial dimensions (height and width) while retaining important information. Given an input feature map  $x \in \mathbb{R}^{H \times W}$ , a pooling operation partitions  $x$  into (possibly overlapping) regions and applies an aggregation function to each region.

The most frequently used pooling operations are max pooling and average pooling. Max pooling selects the largest value from each region, defined as:

$$y_{i,j} = \max_{(m,n) \in \mathcal{R}_{i,j}} x_{m,n},$$

whereas average pooling calculates the mean of the values within each region, expressed as:

$$y_{i,j} = \frac{1}{|\mathcal{R}_{i,j}|} \sum_{(m,n) \in \mathcal{R}_{i,j}} x_{m,n}.$$

Here,  $\mathcal{R}_{i,j}$  denotes the set of spatial indices in the pooling region corresponding to output location  $(i,j)$ , and  $y \in \mathbb{R}^{H' \times W'}$  represents the resulting pooled feature map.

**Example 3.7** (Max and Average Pooling). Let the input feature map be:

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

Applying  $2 \times 2$  pooling with stride 2 results in four  $2 \times 2$  regions. Computing the max and average pooling we get the following result:

$$\text{Max} = \begin{bmatrix} 6 & 4 \\ 3 & 9 \end{bmatrix} \quad \text{Average} = \begin{bmatrix} 3.75 & 2.25 \\ 1.5 & 7.0 \end{bmatrix}$$

**Definition 3.8** (Feature Map). A *feature map* is the output of applying a specific convolutional filter to an input tensor, followed by a non-linear activation function.

Suppose the input  $x \in \mathbb{R}^{H \times W \times C}$  is an image with height  $H$ , width  $W$ , and  $C$  channels. Let  $\theta^{(k)}$  denote the  $k$ -th convolutional kernel (a learnable filter), and let  $b^{(k)} \in \mathbb{R}$  be its associated bias. The result of applying this kernel to the input via convolution, followed by a non-linear activation function  $\sigma$ , is the  $k$ -th feature map:

$$f^{(k)}(x) = \sigma(x * \theta^{(k)} + b^{(k)}) \in \mathbb{R}^{H' \times W'},$$

where  $*$  denotes the convolution operation, and  $H' \times W'$  is the spatial size of the output. Each kernel captures a different pattern or feature in the input, so the set of all feature maps represents different learned aspects of the input data.

Having outlined the core components that build a CNN, we are now ready to explore how these elements work together to transform raw input images into accurate galaxy classifications.

## 3.2 The Classification Journey

In this section, we follow the journey of a galaxy image as it makes its way from observational data to a final morphological classification. The transformation takes place along the layers of our Convolutional Neural Network, where the image is gradually processed through a sequence of well-defined stages. Along the way, key features are extracted, patterns are recognized, and the network builds an internal representation of the galaxy's structure. By tracing this process, we gain insight into how the network learns to convert visual data into a scientifically meaningful classification.

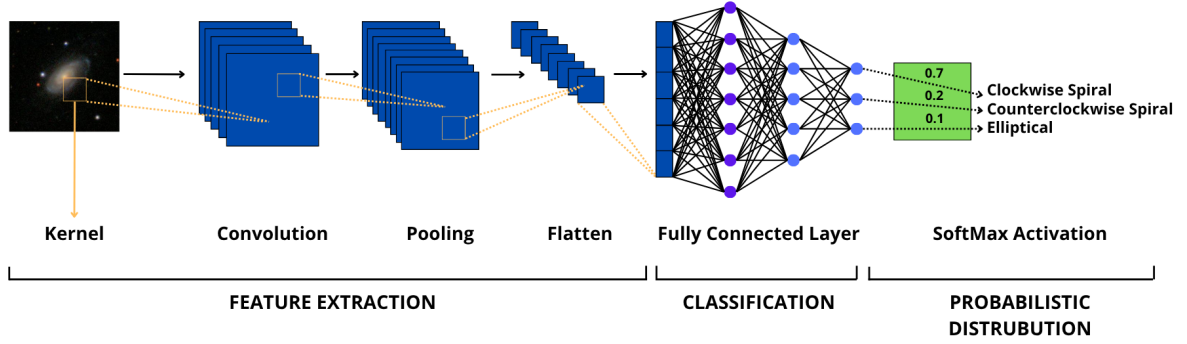


Figure 3.5: CNN architecture for galaxy classification

### Feature Extraction

The image of a galaxy begins its journey as raw observational data, captured and stored by a telescope in a digital format, typically as a color image with three channels corresponding to red, green, and blue light. We take this image and prepare it for analysis by representing it as a 3-dimensional tensor as explained in Definition 2.15. This tensor representation forms the input to our network.

The first step in the feature extraction process is convolution. Convolutional layers apply a family of learnable kernels  $\{\theta^{(k)}\}_{k=1}^K$ , each defined on a small support in  $\mathbb{Z}^2$ . For each kernel, the discrete convolution operation is defined by:

$$(y^{(k)})(u) = (x * \theta^{(k)})(u) = \sum_{v \in \Omega} x(v) \theta^{(k)}(u - v).$$

As established in Theorem 2.14, this operation is translation-equivariant: applying a shift operator  $S_v$  to the input commutes with convolution, i.e.,

$$S_v(x * \theta^{(k)}) = (S_v x) * \theta^{(k)}.$$

This ensures that feature detection is consistent, a property crucial for images where galaxy morphology must be recognized regardless of its position within the frame. Following convolution, the resulting feature maps are passed through an elementwise non-linear activation function  $\sigma : \mathbb{R} \rightarrow \mathbb{R}$ , typically ReLU, defined by  $\sigma(z) = \max(0, z)$  as we did in Definition 3.5. This operation enables the network to approximate complex, nonlinear functions and capture relationships in the data:

$$z^{(k)}(u) = \sigma(y^{(k)}(u)).$$

The next feature extraction step is pooling. Pooling is applied to the activated feature maps  $z^{(k)}$  to manage the spatial dimensions and improve computational efficiency. This operation reduces the spatial resolution of the feature maps while preserving the most important features and without reducing the number of channels. For instance, using max pooling as in Example 3.7:

$$p^{(k)}(i, j) = \max_{v \in R_{i,j}} z^{(k)}(v)$$

where  $p^{(k)}(i, j)$  is the output of the pooling operation at location  $(i, j)$  in the  $k$ -th channel, and  $z^{(k)}(v)$  are the activation values within the region  $R_{i,j}$ . This sequence of convolution, activation, and pooling forms a repeated pattern, stacking multiple layers to build hierarchical feature representations. Mathematically, we denote the composition of these layers at depth  $\ell$  as:

$$x^{(\ell)} = P\left(\sigma(x^{(\ell-1)} * \theta^{(\ell)})\right).$$

As  $\ell$  increases, these layers extract progressively more abstract and complex features, essential for capturing the morphological diversity of galaxies. This high-dimensional feature map has to be transformed for our fully connected layers. Therefore, the tensor is reshaped using a flattening operation, mapping:  $X(\Omega, K) \rightarrow \mathbb{R}^N$ , where  $N$  denotes the total number of activations aggregated across all channels.

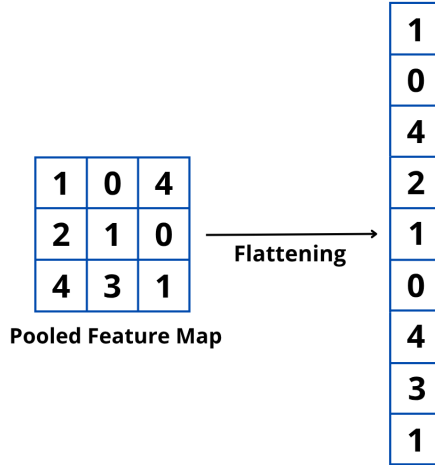


Figure 3.6: Flattening pooled features into a vector.

## Classification

Following flattening, the data is passed through fully connected (dense) layers. These layers perform transformations with learnable parameters, followed by non-linear activations, to combine the extracted features across the entire image:

$$h^{(m)} = \sigma(W^{(m)}h^{(m-1)} + b^{(m)}),$$

where  $W^{(m)}$  and  $b^{(m)}$  denote the weights and biases of layer  $m$ . Dense layers are specifically designed to operate on one-dimensional data, making flattening a necessary step. In the context of galaxy classification, dense layers synthesize the learned features, such as spiral arms or elliptical structures, into a final decision about the galaxy's morphological class.

## Probabilistic Distribution

To quantify how well the predicted distribution  $\hat{y}$  matches the true distribution  $y$ , that was one-hot encoded, we derive the loss from the negative log-likelihood. Instead of representing the true class label as a scalar  $y \in \{1, \dots, C\}$ , we express it as a one-hot encoded vector  $(y_1, y_2, \dots, y_C) \in \{0, 1\}^C$ , where:

$$y_i = \begin{cases} 1 & \text{if } i \text{ is the correct class,} \\ 0 & \text{otherwise.} \end{cases}$$

Since only one element of the vector is 1 (and the rest are 0), we can express the likelihood as:

$$P(y \mid x; \theta) = \prod_{i=1}^C \hat{y}_i^{y_i},$$

and the corresponding negative log-likelihood gives the categorical cross-entropy loss:

$$\mathcal{L}_{\text{CE}}(y, \hat{y}) = - \sum_{i=1}^C y_i \log(\hat{y}_i).$$

For a single example where class  $c$  is correct, this simplifies to:

$$\mathcal{L}_{\text{CE}} = -\log(\hat{y}_c),$$

which penalizes the model for assigning low probability to the correct class. The loss grows logarithmically as the predicted probability decreases.

The final dense layer uses the softmax function to produce a probability distribution over the morphological classes:

$$\hat{y}_c = \frac{\exp(h_c^{(L)})}{\sum_k \exp(h_k^{(L)})}$$

where  $L$  denotes the last layer of the network. Softmax transforms the network's raw outputs into non-negative values that sum to one by exponentiating each output and dividing by the total sum. This ensures the outputs can be interpreted as probabilities, suitable for multi-class classification tasks. Below, an example of how this works:

$$\begin{bmatrix} 0.5 \\ 3.0 \\ 1.5 \\ 2.0 \\ 0.8 \end{bmatrix} \longrightarrow \boxed{\frac{\exp(h_c^{(L)})}{\sum_k \exp(h_k^{(L)})}} \longrightarrow \begin{bmatrix} 0.05 \\ 0.56 \\ 0.13 \\ 0.20 \\ 0.06 \end{bmatrix}$$

With this, the classification journey through our Convolutional Neural Network would be finished. After passing through convolution, activation, pooling, flattening, and dense layers, the softmax function produces a probability distribution over our labels (the different morphological classes). The highest probability in this distribution determines the network's final prediction, such as "Clockwise Spiral", to the input image. This result completes the process of transforming visual data into a mathematically meaningful classification, demonstrating the usefulness and accuracy of our method for classifying galaxy shapes.



### 3.3 Implementation Details

Having concluded the mathematical explanation of Convolutional Neural Networks, it is now time to focus on the second major goal of this thesis: the development of a program for galaxy morphological classification, capable of accurately classifying large volumes of image data.

Our implementation is written in Python and uses several open-source libraries: Pandas for data handling, scikit-learn for preprocessing and evaluation [14], scikit-image for image processing [15], and TensorFlow [16] for building and training the deep learning model. The dataset was obtained from the Galaxy Zoo project [8, 9], originally containing around 500,000 galaxy images. To make it more manageable and ensure balanced classes, we selected 120,000 images evenly distributed among eight morphological categories. For consistency, we kept the same galaxy classes.

The CNN used in this work includes three main convolutional blocks with increasing filter sizes: 32, 64, and 128. This structure helps the network first detect simple patterns and later more complex features. Each block has two convolutional layers with ReLU activation as explained in Definition 3.5. Max pooling (Definition 3.6) reduced the size of feature maps and kept the most important information, while dropout layers helped prevent overfitting. After these blocks, a global average pooling layer (Definition 3.6) reduced data size while preserving essential features. Then, a dense layer with 256 neurons learned higher-level representations. The final softmax output layer had eight neurons, one for each galaxy class. We trained the model using the Adam optimizer (which adapts the learning rate) and the categorical crossentropy loss function, suitable for multi-class classification. These were our results:

<b>Morphological Classes</b>	<b>Precision</b>	<b>Recall</b>	<b>F1</b>	<b>Accuracy</b>	<b>AUC</b>
Clockwise Spiral	0.9369	0.9888	0.9621	0.9888	0.9955
Anticlockwise Spiral	0.9378	0.9828	0.9598	0.9828	0.9948
Disk	0.9447	0.9780	0.9610	0.9780	0.9996
Merger	0.9023	0.9732	0.9364	0.9732	0.9967
Edge	0.7768	0.9051	0.8360	0.9051	0.9867
Elliptical	0.8179	0.8616	0.8392	0.8616	0.9896
<b>Average</b>	<b>0.8861</b>	<b>0.9483</b>	<b>0.9158</b>	<b>0.9483</b>	<b>0.9938</b>
<b>Non-Morphological Classes</b>	<b>Precision</b>	<b>Recall</b>	<b>F1</b>	<b>Accuracy</b>	<b>AUC</b>
Cosmological Structure	0.6769	0.5502	0.6070	0.5502	0.9012
Uncertain	0.6299	0.3084	0.4141	0.3084	0.9417
<b>Average</b>	<b>0.6534</b>	<b>0.4293</b>	<b>0.5105</b>	<b>0.4293</b>	<b>0.9215</b>
<b>Total Average</b>	<b>0.8279</b>	<b>0.8185</b>	<b>0.8145</b>	<b>0.8185</b>	<b>0.9757</b>

Table 3.1: Evaluation metrics per class: Precision, Recall, F1, Accuracy, and AUC.

We evaluated the model on a validation set using standard classification metrics, including precision, recall, F1 score, accuracy, and AUC. While the global accuracy was 86.17%, a closer analysis reveals a significant performance gap across different classes.

In particular, well-defined galaxy morphologies such as disk, anticlockwise spiral, and clockwise spiral achieved outstanding results, with accuracy values of 97.8%, 98.3%, and 98.9%, respectively, and AUC scores exceeding 0.99, indicating near-perfect separability from other classes. These high-performing categories also showed consistently high precision, recall, and F1 scores above 0.94, demonstrating the model’s strong ability to detect structured patterns.

On the other hand, performance dropped substantially for ambiguous types. The uncertain class had an accuracy of only 30.8% and an F1 score of 0.41, while cosmological structure showed a similarly low accuracy of 55.0%. These classes likely suffer from labeling noise, as the uncertain label was artificially defined and cosmological structures, like huge groups of galaxies and giant empty spaces, lack clear morphological traits. Further analysis may reveal that misclassifying uncertain cases is not an error but a strength, as the model could be correctly identifying their true morphology.

To better assess the model’s capability on well-defined morphological classes, we recalculated the average accuracy excluding the problematic categories mentioned above. The resulting average accuracy across the remaining six classes was approximately 94.8%, a substantial improvement over the macro average of 81.85% given on the table. This more focused evaluation confirms that the model performs exceptionally well on structured and meaningful categories.

Furthermore, we tested the model under more challenging conditions by applying random rotations to a separate test set. Even under these transformations, the model retained a strong overall accuracy of 83.29%, further supporting the explanation of rotation invariance introduced in earlier chapters.

Because of limited local hardware (weak GPU), we used a virtual machine with two *NVIDIA RTX 3060* GPUs, compatible with *CUDA 12.5*. This setup allowed us to use TensorFlow’s GPU acceleration, greatly speeding up training. We connected to the virtual machine using *SSH* and transferred all project files via *rsync*. Training and evaluation were done remotely through the virtual machine terminal, and the full training process took less than one hour.

In summary, the results are promising, though there is room for improvement. Future work may include expanding the dataset, applying data augmentation, and refining the architecture. Additionally, excluding the cosmological structure class, which may not represent a galaxy type, and improving the manually created uncertain label could further enhance the program’s performance and lead to more reliable classification results. Below, we present the implementation of the program itself:

# Convolutional Neural Network for Galaxy Morphological Classification

## Import and Set Up

We import all the necessary libraries for data manipulation, visualization, and deep learning. These will be used to build and train our Convolutional Neural Network (CNN) for galaxy classification.

```
# Data manipulation and visualization
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import cv2
import random
from scipy.ndimage import rotate

# Progress bar
from tqdm import tqdm

# Deep learning libraries
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.models import Sequential
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.losses import CategoricalCrossentropy
from tensorflow.keras.layers import Conv2D, MaxPooling2D,
BatchNormalization, Dropout
from tensorflow.keras.layers import GlobalAveragePooling2D, Dense,
Input
from tensorflow.keras.callbacks import EarlyStopping,
ReduceLROnPlateau
from tensorflow.keras.utils import to_categorical

from sklearn.metrics import classification_report, confusion_matrix,
roc_auc_score, precision_recall_fscore_support, accuracy_score
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
```

## Load the Data

We load the dataset containing galaxies with spectroscopic data

```
train = pd.read_csv("./Galaxies_Spectra.csv")
```

## Define Labels and Threshold

Here, we specify the columns representing class probabilities and their corresponding labels. We also define a threshold to determine which class is assigned.

```
# Columns containing classification probabilities
prob_cols = ['P_CW', 'P_ACW', 'P_EL', 'P_EDGE', 'P_DK', 'P_MG',
             'P_CS']

# Corresponding labels
labels = ['SPIRAL_CW', 'SPIRAL_ACW', 'ELLIPTICAL', 'EDGE', 'DISK',
          'MERGER', 'COSMOLOGICAL_STRUCTURE']

# Minimum probability required to assign a label
threshold = 0.5
```

## Assign Labels Based on Probabilities

We assign each galaxy a label based on the highest probability that exceeds the threshold. If no class meets the threshold, the label is set to "UNCERTAIN".

```
# Extract probability values
probs = train[prob_cols].values

# Find max probability and corresponding class index for each row
max_probs = probs.max(axis=1)
max_indices = probs.argmax(axis=1)

# Assign label if max probability >= threshold, otherwise 'UNCERTAIN'
assigned_labels = np.where(
    max_probs >= threshold,
    np.array(labels)[max_indices],
    'UNCERTAIN'
)

# Add the labels to the dataframe
train['Assigned_Label'] = assigned_labels

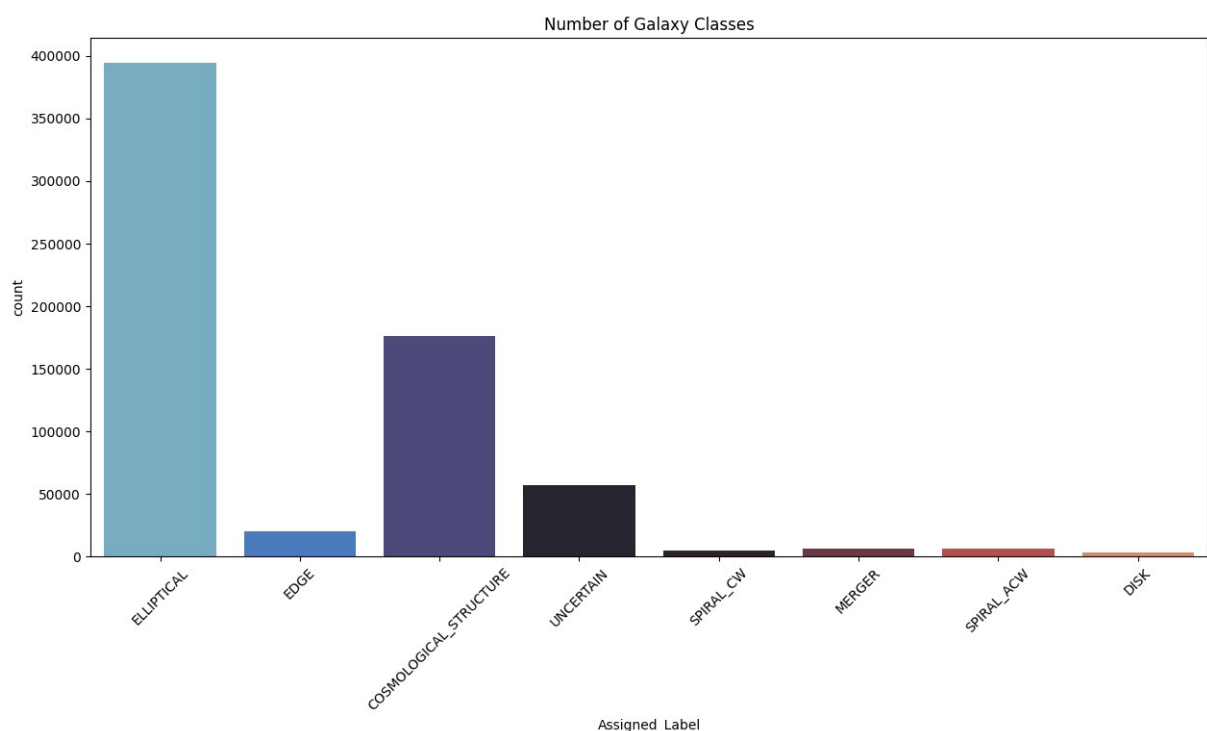
# Display examples
print(train[['Assigned_Label'] + prob_cols].head())
```

	Assigned_Label	P_CW	P_ACW	P_EL	P_EDGE	P_DK	P_MG	P_CS
0	ELLIPTICAL	0.034	0.000	0.610	0.153	0.153	0.051	0.186
1	ELLIPTICAL	0.000	0.167	0.611	0.222	0.000	0.000	0.389
2	ELLIPTICAL	0.029	0.000	0.735	0.147	0.074	0.015	0.176
3	ELLIPTICAL	0.019	0.000	0.885	0.058	0.019	0.019	0.077
4	ELLIPTICAL	0.000	0.000	0.712	0.220	0.068	0.000	0.220

## Visualize the Distribution of Assigned Classes

We visualize the number of galaxies assigned to each class, which gives us an idea of class imbalance or uncertainty.

```
# Plot the count of each assigned label
plt.figure(figsize=(15, 7))
sns.countplot(data=train, x='Assigned_Label', hue='Assigned_Label',
palette='icefire', legend=False)
plt.title("Number of Galaxy Classes")
plt.xticks(rotation=45)
plt.show()
```



## Define Paths for Spectra Matrices

Here, we specify the directories where the spectral matrix files for each galaxy are stored.

```
train_matrix_dir = "./Spectra_matrices/Spectra_matrices"
```

## Load Mapping Between Filenames and Galaxy IDs

This CSV contains the mapping between matrix filenames and galaxy IDs, which is needed to align the matrix data with the labels.

```
mapping_file = "./gz2_filename_mapping.csv"
mapping_df = pd.read_csv(mapping_file)
```

## Balance and Sample the Dataset

To ensure balanced training data, we sample 15,000 galaxies from each class. This helps the model learn more equally across all morphological types.

```
np.random.seed(42)

# Sample 15,000 examples from each class (with replacement if not
# enough samples)
train = train.groupby('Assigned_Label').apply(
    lambda x: x.sample(n=15000, replace=len(x) < 15000)
).reset_index(drop=True)

print(train['Assigned_Label'].value_counts())
```

Assigned_Label	
COSMOLOGICAL_STRUCTURE	15000
DISK	15000
EDGE	15000
ELLIPTICAL	15000
MERGER	15000
SPIRAL_ACW	15000
SPIRAL_CW	15000
UNCERTAIN	15000

Name: count, dtype: int64

```
/tmp/ipykernel_2082/1733161426.py:4: FutureWarning:
DataFrameGroupBy.apply operated on the grouping columns. This behavior
is deprecated, and in a future version of pandas the grouping columns
will be excluded from the operation. Either pass
`include_groups=False` to exclude the groupings or explicitly select
the grouping columns after groupby to silence this warning.
  train = train.groupby('Assigned_Label').apply(
```

## Merge Spectra Matrix Mapping

We merge the galaxy dataset with the matrix ID mapping file to link each galaxy to its corresponding .npy matrix file.

```
train_df = pd.merge(train, mapping_df, left_on='OBJID',
right_on='objid')
```

## Load Matrix Data and Labels

This section reads each galaxy's spectral matrix and its label. These matrices are stored in .npy files and represent the input to our CNN.

```
rib_data = [] # List to store matrix data
labels = [] # List to store labels
```

```
# Iterate through each galaxy and load its matrix
for _, row in tqdm(train_df.iterrows(), total=len(train_df)):
    asset_id = row['asset_id']
    rib_path = os.path.join(train_matrix_dir, f"{asset_id}.npy")

    if os.path.exists(rib_path):
        rib_matrix = np.load(rib_path)
        rib_data.append(rib_matrix)
        labels.append(row['Assigned_Label'])
```

## Preprocess the Input and Labels

Once we have the data and labels, we convert them to NumPy arrays and encode the labels into one-hot format for training.

```
# Convert list of matrices to numpy array
X = np.array(rib_data)

# Convert labels to numpy array
y = np.array(labels)

# Encode class labels
label_encoder = LabelEncoder()
y_encoded = label_encoder.fit_transform(y)

# One-hot encode the encoded labels
y_one_hot = to_categorical(y_encoded)
```

## Train-Test Split

We split the dataset into training and validation subsets using stratified sampling to preserve class balance.

```
X_train, X_val, y_train, y_val = train_test_split(X, y_one_hot,
test_size=0.2, stratify=y_one_hot, random_state=42)
```

## Define the CNN Model

This cell defines our Convolutional Neural Network (CNN) model using the Keras Sequential API. The architecture consists of three convolutional blocks with increasing filter sizes (32, 64, 128).

Each block contains two convolutional layers with ReLU activation and batch normalization, followed by max pooling and dropout for regularization.

After feature extraction, a global average pooling layer reduces the spatial dimensions before passing data to a dense layer with 256 neurons. A final softmax layer outputs class probabilities for 8 classes.

```
model = Sequential([
    Input(shape=X_train.shape[1:]),

    # Block 1
    Conv2D(32, (3, 3), padding='same', activation='relu'),
    BatchNormalization(),
    Conv2D(32, (3, 3), padding='same', activation='relu'),
    BatchNormalization(),
    MaxPooling2D((2, 2)),
    Dropout(0.25),

    # Block 2
    Conv2D(64, (3, 3), padding='same', activation='relu'),
    BatchNormalization(),
    Conv2D(64, (3, 3), padding='same', activation='relu'),
    BatchNormalization(),
    MaxPooling2D((2, 2)),
    Dropout(0.3),

    # Block 3
    Conv2D(128, (3, 3), padding='same', activation='relu'),
    BatchNormalization(),
    Conv2D(128, (3, 3), padding='same', activation='relu'),
    BatchNormalization(),
    MaxPooling2D((2, 2)),
    Dropout(0.4),

    # Classifier
    GlobalAveragePooling2D(),
    Dense(256, activation='relu'),
    Dropout(0.5),
    Dense(8, activation='softmax')
])
```

```
I0000 00:00:1750005824.121705    2082 gpu_device.cc:2019] Created
device /job:localhost/replica:0/task:0/device:GPU:0 with 10274 MB
memory:  -> device: 0, name: NVIDIA GeForce RTX 3060, pci bus id:
0000:87:00.0, compute capability: 8.6
I0000 00:00:1750005824.123836    2082 gpu_device.cc:2019] Created
device /job:localhost/replica:0/task:0/device:GPU:1 with 10274 MB
memory:  -> device: 1, name: NVIDIA GeForce RTX 3060, pci bus id:
0000:88:00.0, compute capability: 8.6
```



## Compile the CNN Model

This cell compiles the CNN model with the Adam optimizer (learning rate = 0.001), using categorical crossentropy as the loss function and accuracy as the evaluation metric.

Additionally, two callbacks are defined to improve training:

EarlyStopping stops training if validation performance stops improving for 10 epochs and restores the best model weights, while ReduceLROnPlateau reduces the learning rate by a factor of 0.5 if the validation metric stops changing significantly for 5 epochs.

```
# Compile model
model.compile(optimizer=Adam(learning_rate=1e-3),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Callbacks
early_stop = EarlyStopping(patience=10, restore_best_weights=True,
                           verbose=1)
reduce_lr = ReduceLROnPlateau(factor=0.5, patience=5, verbose=1)
```

## Train the Model

We now train the model for 100 epochs using the training data. Validation data is used to monitor generalization performance.

```
# Train the model
history = model.fit(
    X_train, y_train,
    validation_data=(X_val, y_val),
    epochs=100,
    batch_size=32,
    callbacks=[early_stop, reduce_lr]
)
```

Epoch 1/100

```
WARNING: All log messages before absl::InitializeLog() is called are
written to STDERR
I0000 00:00:1750005858.391506    2300 service.cc:152] XLA service
0x7f965801b770 initialized for platform CUDA (this does not guarantee
that XLA will be used). Devices:
I0000 00:00:1750005858.392432    2300 service.cc:160]   StreamExecutor
device (0): NVIDIA GeForce RTX 3060, Compute Capability 8.6
I0000 00:00:1750005858.392821    2300 service.cc:160]   StreamExecutor
device (1): NVIDIA GeForce RTX 3060, Compute Capability 8.6
I0000 00:00:1750005859.343480    2300 cuda_dnn.cc:529] Loaded cuDNN
version 90800
```

3/1276 ————— 50s 39ms/step - accuracy: 0.2882 -  
loss: 2.2390

I0000 00:00:1750005873.585632 2300 device\_compiler.h:188] Compiled  
cluster using XLA! This line is logged at most once for the lifetime  
of the process.

1276/1276 ————— 98s 56ms/step - accuracy: 0.3467 -  
loss: 1.6211 - val\_accuracy: 0.3624 - val\_loss: 1.6829 -  
learning\_rate: 0.0010

Epoch 2/100

1276/1276 ————— 47s 37ms/step - accuracy: 0.5181 -  
loss: 1.1538 - val\_accuracy: 0.6867 - val\_loss: 0.9154 -  
learning\_rate: 0.0010

Epoch 3/100

1276/1276 ————— 48s 37ms/step - accuracy: 0.7345 -  
loss: 0.7522 - val\_accuracy: 0.7326 - val\_loss: 0.7390 -  
learning\_rate: 0.0010

Epoch 4/100

1276/1276 ————— 48s 38ms/step - accuracy: 0.7686 -  
loss: 0.6519 - val\_accuracy: 0.8024 - val\_loss: 0.5624 -  
learning\_rate: 0.0010

Epoch 5/100

1276/1276 ————— 47s 37ms/step - accuracy: 0.7824 -  
loss: 0.6060 - val\_accuracy: 0.7353 - val\_loss: 0.7547 -  
learning\_rate: 0.0010

Epoch 6/100

1276/1276 ————— 47s 37ms/step - accuracy: 0.7960 -  
loss: 0.5682 - val\_accuracy: 0.8104 - val\_loss: 0.5279 -  
learning\_rate: 0.0010

Epoch 7/100

1276/1276 ————— 48s 37ms/step - accuracy: 0.7994 -  
loss: 0.5505 - val\_accuracy: 0.8155 - val\_loss: 0.5045 -  
learning\_rate: 0.0010

Epoch 8/100

1276/1276 ————— 47s 37ms/step - accuracy: 0.8099 -  
loss: 0.5279 - val\_accuracy: 0.8189 - val\_loss: 0.5013 -  
learning\_rate: 0.0010

Epoch 9/100

1276/1276 ————— 57s 45ms/step - accuracy: 0.8122 -  
loss: 0.5172 - val\_accuracy: 0.8266 - val\_loss: 0.4865 -  
learning\_rate: 0.0010

Epoch 10/100

1276/1276 ————— 52s 41ms/step - accuracy: 0.8184 -  
loss: 0.4973 - val\_accuracy: 0.8076 - val\_loss: 0.5359 -  
learning\_rate: 0.0010

Epoch 11/100

1276/1276 ————— 52s 41ms/step - accuracy: 0.8237 -  
loss: 0.4834 - val\_accuracy: 0.8052 - val\_loss: 0.5414 -  
learning\_rate: 0.0010

```
Epoch 12/100
1276/1276 _____ 52s 41ms/step - accuracy: 0.8254 -
loss: 0.4801 - val_accuracy: 0.8273 - val_loss: 0.4850 -
learning_rate: 0.0010
Epoch 13/100
1276/1276 _____ 52s 41ms/step - accuracy: 0.8265 -
loss: 0.4724 - val_accuracy: 0.8262 - val_loss: 0.5075 -
learning_rate: 0.0010
Epoch 14/100
1276/1276 _____ 51s 40ms/step - accuracy: 0.8339 -
loss: 0.4529 - val_accuracy: 0.8244 - val_loss: 0.4963 -
learning_rate: 0.0010
Epoch 15/100
1276/1276 _____ 53s 41ms/step - accuracy: 0.8374 -
loss: 0.4403 - val_accuracy: 0.8012 - val_loss: 0.5571 -
learning_rate: 0.0010
Epoch 16/100
1276/1276 _____ 52s 41ms/step - accuracy: 0.8352 -
loss: 0.4423 - val_accuracy: 0.8398 - val_loss: 0.4531 -
learning_rate: 0.0010
Epoch 17/100
1276/1276 _____ 51s 40ms/step - accuracy: 0.8390 -
loss: 0.4355 - val_accuracy: 0.8177 - val_loss: 0.5158 -
learning_rate: 0.0010
Epoch 18/100
1276/1276 _____ 51s 40ms/step - accuracy: 0.8437 -
loss: 0.4255 - val_accuracy: 0.8444 - val_loss: 0.4421 -
learning_rate: 0.0010
Epoch 19/100
1276/1276 _____ 50s 40ms/step - accuracy: 0.8437 -
loss: 0.4241 - val_accuracy: 0.8458 - val_loss: 0.4368 -
learning_rate: 0.0010
Epoch 20/100
1276/1276 _____ 51s 40ms/step - accuracy: 0.8470 -
loss: 0.4130 - val_accuracy: 0.8358 - val_loss: 0.4557 -
learning_rate: 0.0010
Epoch 21/100
1276/1276 _____ 51s 40ms/step - accuracy: 0.8509 -
loss: 0.4121 - val_accuracy: 0.8435 - val_loss: 0.4386 -
learning_rate: 0.0010
Epoch 22/100
1276/1276 _____ 51s 40ms/step - accuracy: 0.8500 -
loss: 0.4066 - val_accuracy: 0.8433 - val_loss: 0.4366 -
learning_rate: 0.0010
Epoch 23/100
1276/1276 _____ 51s 40ms/step - accuracy: 0.8551 -
loss: 0.3956 - val_accuracy: 0.8359 - val_loss: 0.4600 -
learning_rate: 0.0010
Epoch 24/100
```

```
1276/1276 ————— 51s 40ms/step - accuracy: 0.8537 -  
loss: 0.3951 - val_accuracy: 0.8391 - val_loss: 0.4466 -  
learning_rate: 0.0010  
Epoch 25/100  
1276/1276 ————— 50s 39ms/step - accuracy: 0.8558 -  
loss: 0.3845 - val_accuracy: 0.8526 - val_loss: 0.4287 -  
learning_rate: 0.0010  
Epoch 26/100  
1276/1276 ————— 51s 40ms/step - accuracy: 0.8577 -  
loss: 0.3851 - val_accuracy: 0.8365 - val_loss: 0.4920 -  
learning_rate: 0.0010  
Epoch 27/100  
1276/1276 ————— 51s 40ms/step - accuracy: 0.8567 -  
loss: 0.3854 - val_accuracy: 0.8448 - val_loss: 0.4619 -  
learning_rate: 0.0010  
Epoch 28/100  
1276/1276 ————— 51s 40ms/step - accuracy: 0.8611 -  
loss: 0.3733 - val_accuracy: 0.8302 - val_loss: 0.5041 -  
learning_rate: 0.0010  
Epoch 29/100  
1276/1276 ————— 51s 40ms/step - accuracy: 0.8611 -  
loss: 0.3724 - val_accuracy: 0.8448 - val_loss: 0.4585 -  
learning_rate: 0.0010  
Epoch 30/100  
1276/1276 ————— 0s 37ms/step - accuracy: 0.8602 - loss:  
0.3710  
Epoch 30: ReduceLR0nPlateau reducing learning rate to  
0.00050000000237487257.  
1276/1276 ————— 51s 40ms/step - accuracy: 0.8602 -  
loss: 0.3710 - val_accuracy: 0.8470 - val_loss: 0.4571 -  
learning_rate: 0.0010  
Epoch 31/100  
1276/1276 ————— 51s 40ms/step - accuracy: 0.8738 -  
loss: 0.3391 - val_accuracy: 0.8590 - val_loss: 0.4162 -  
learning_rate: 5.0000e-04  
Epoch 32/100  
1276/1276 ————— 51s 40ms/step - accuracy: 0.8777 -  
loss: 0.3259 - val_accuracy: 0.8503 - val_loss: 0.4401 -  
learning_rate: 5.0000e-04  
Epoch 33/100  
1276/1276 ————— 51s 40ms/step - accuracy: 0.8797 -  
loss: 0.3196 - val_accuracy: 0.8572 - val_loss: 0.4143 -  
learning_rate: 5.0000e-04  
Epoch 34/100  
1276/1276 ————— 51s 40ms/step - accuracy: 0.8826 -  
loss: 0.3210 - val_accuracy: 0.8489 - val_loss: 0.4433 -  
learning_rate: 5.0000e-04  
Epoch 35/100  
1276/1276 ————— 51s 40ms/step - accuracy: 0.8833 -
```

```
loss: 0.3113 - val_accuracy: 0.8531 - val_loss: 0.4497 -  
learning_rate: 5.0000e-04  
Epoch 36/100  
1276/1276 _____ 51s 40ms/step - accuracy: 0.8874 -  
loss: 0.3050 - val_accuracy: 0.8555 - val_loss: 0.4388 -  
learning_rate: 5.0000e-04  
Epoch 37/100  
1276/1276 _____ 52s 40ms/step - accuracy: 0.8913 -  
loss: 0.2948 - val_accuracy: 0.8617 - val_loss: 0.4053 -  
learning_rate: 5.0000e-04  
Epoch 38/100  
1276/1276 _____ 51s 40ms/step - accuracy: 0.8861 -  
loss: 0.3079 - val_accuracy: 0.8600 - val_loss: 0.4354 -  
learning_rate: 5.0000e-04  
Epoch 39/100  
1276/1276 _____ 52s 40ms/step - accuracy: 0.8900 -  
loss: 0.2980 - val_accuracy: 0.8588 - val_loss: 0.4273 -  
learning_rate: 5.0000e-04  
Epoch 40/100  
1276/1276 _____ 51s 40ms/step - accuracy: 0.8884 -  
loss: 0.3016 - val_accuracy: 0.8544 - val_loss: 0.4594 -  
learning_rate: 5.0000e-04  
Epoch 41/100  
1276/1276 _____ 51s 40ms/step - accuracy: 0.8875 -  
loss: 0.2982 - val_accuracy: 0.8597 - val_loss: 0.4357 -  
learning_rate: 5.0000e-04  
Epoch 42/100  
1275/1276 _____ 0s 37ms/step - accuracy: 0.8901 - loss:  
0.2903  
Epoch 42: ReduceLR0nPlateau reducing learning rate to  
0.0002500000118743628.  
1276/1276 _____ 52s 40ms/step - accuracy: 0.8901 -  
loss: 0.2903 - val_accuracy: 0.8538 - val_loss: 0.4502 -  
learning_rate: 5.0000e-04  
Epoch 43/100  
1276/1276 _____ 52s 40ms/step - accuracy: 0.8962 -  
loss: 0.2763 - val_accuracy: 0.8592 - val_loss: 0.4446 -  
learning_rate: 2.5000e-04  
Epoch 44/100  
1276/1276 _____ 51s 40ms/step - accuracy: 0.8973 -  
loss: 0.2763 - val_accuracy: 0.8607 - val_loss: 0.4291 -  
learning_rate: 2.5000e-04  
Epoch 45/100  
1276/1276 _____ 52s 40ms/step - accuracy: 0.8986 -  
loss: 0.2705 - val_accuracy: 0.8616 - val_loss: 0.4302 -  
learning_rate: 2.5000e-04  
Epoch 46/100  
1276/1276 _____ 52s 40ms/step - accuracy: 0.9043 -  
loss: 0.2574 - val_accuracy: 0.8594 - val_loss: 0.4384 -
```

```

learning_rate: 2.5000e-04
Epoch 47/100
1276/1276 ————— 0s 37ms/step - accuracy: 0.9004 - loss:
0.2621
Epoch 47: ReduceLROnPlateau reducing learning rate to
0.0001250000059371814.
1276/1276 ————— 51s 40ms/step - accuracy: 0.9004 -
loss: 0.2621 - val_accuracy: 0.8578 - val_loss: 0.4545 -
learning_rate: 2.5000e-04
Epoch 47: early stopping
Restoring model weights from the end of the best epoch: 37.

```

## Evaluate the Model

Once training is complete, we evaluate the model's performance on the validation dataset.

```

# Evaluate accuracy and loss on the validation set
val_loss, val_accuracy = model.evaluate(X_val, y_val)

print(f"Validation Loss: {val_loss:.4f}")
print(f"Validation Accuracy: {val_accuracy:.4f}")

319/319 ————— 4s 13ms/step - accuracy: 0.8625 - loss:
0.4080
Validation Loss: 0.4053
Validation Accuracy: 0.8617

```

## Plot Training and Validation History

We visualize how training and validation accuracy/loss evolve over time to assess learning behavior.

```

# Plot training and validation metrics
plt.figure(figsize=(15, 7))

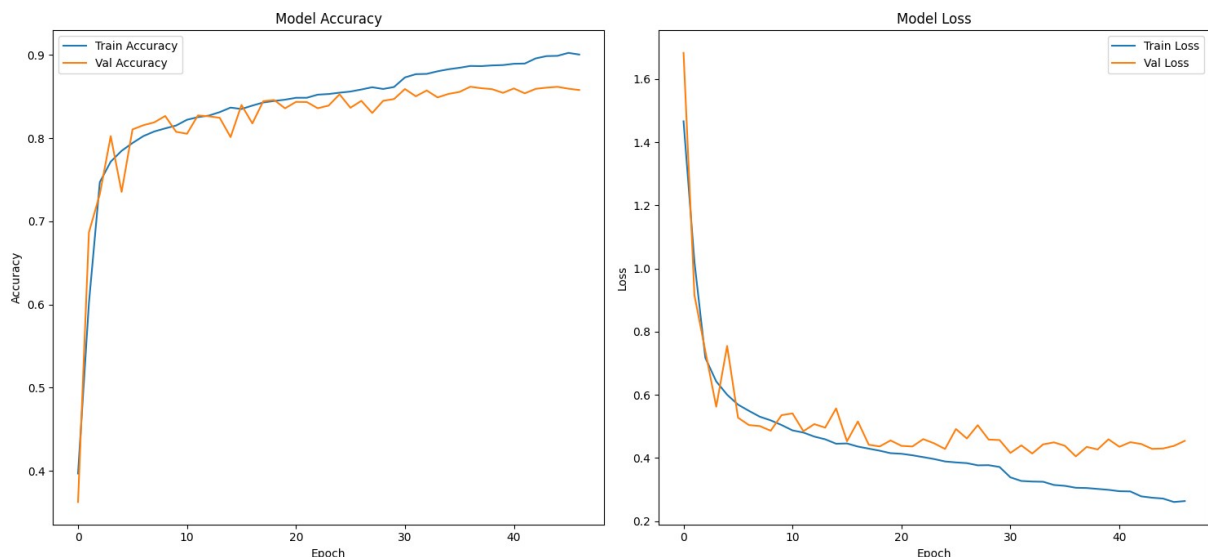
# Accuracy over epochs
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Val Accuracy')
plt.title('Model Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()

# Loss over epochs
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Val Loss')
plt.title('Model Loss')

```

```
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
```

```
plt.tight_layout()
plt.show()
```



## Function to apply random transformations

For each image, the function randomly picks one of four rotations (0°, 90°, 180°, 270°).

```
def random_transform(matrix):
    k = random.choice([0, 1, 2, 3]) # 0°, 90°, 180°, 270°
    transformed = np.rot90(matrix, k=k)
    return transformed
```

## Evaluating Model Robustness to Rotations

To assess the model's robustness, we create a test set by applying *random\_transform* to the original images. We then evaluate the CNN's performance on this rotated test set.

The goal is to demonstrate that the CNN maintains strong performance despite the transformations, indicating that it has learned rotation-invariant features.

```
# Apply transformation to the entire validation set
X_test_augmented = np.array([random_transform(matrix) for matrix in
X_val])

# Predict on the augmented test set
y_test_pred = model.predict(X_test_augmented)
```

```

# Convert predictions and true labels from one-hot to class indices
y_test_pred_labels = np.argmax(y_test_pred, axis=1)
y_test_true_labels = np.argmax(y_val, axis=1)

# Calculate accuracy
matches = np.sum(y_test_pred_labels == y_test_true_labels)
percentage_matches = (matches / len(y_test_true_labels)) * 100

print(f"Percentage of test set matches (after augmentation):
{percentage_matches:.2f}%")

319/319 ————— 4s 11ms/step
Percentage of test set matches (after augmentation): 83.29%

```

## Evaluating Model Performance on the Validation Set

To assess the model's performance, we evaluate it on the validation set using detailed per-class metrics.

This includes precision, recall, F1 score, accuracy, and AUC for each morphological class. The model's predicted probabilities are compared against the true one-hot encoded labels to compute class-wise and macro-averaged performance.

```

y_true = np.argmax(y_val, axis=1)
y_pred_probs = model.predict(X_val)
y_pred = np.argmax(y_pred_probs, axis=1)

# Class names from LabelEncoder
class_names = label_encoder.classes_

# Confusion matrix
conf_matrix = confusion_matrix(y_true, y_pred)

# Per-class precision, recall, and F1
precision, recall, f1, _ = precision_recall_fscore_support(
    y_true, y_pred, labels=np.arange(len(class_names))
)

# Per-class accuracy from confusion matrix
per_class_accuracy = conf_matrix.diagonal() / conf_matrix.sum(axis=1)

# AUC
try:
    auc_macro = roc_auc_score(y_val, y_pred_probs, average='macro',
multi_class='ovr')
    auc_per_class = roc_auc_score(y_val, y_pred_probs, average=None,
multi_class='ovr')
except ValueError:
    auc_macro = None
    auc_per_class = [None] * len(class_names)

```



```

# Summary table
summary_df = pd.DataFrame({
    'Class': class_names,
    'Precision': precision,
    'Recall': recall,
    'F1': f1,
    'Accuracy': per_class_accuracy,
    'AUC': auc_per_class
})

# Average metrics
summary_df.loc['Average'] = [
    'Average',
    precision.mean(),
    recall.mean(),
    f1.mean(),
    per_class_accuracy.mean(),
    auc_macro
]

display(summary_df)
319/319 ————— 4s 11ms/step

```

	Class	Precision	Recall	F1	Accuracy	AUC
0	COSMOLOGICAL_STRUCTURE	0.676883	0.550178	0.606989	0.550178	0.901202
1	DISK	0.944681	0.977974	0.961039	0.977974	0.999630
2	EDGE	0.776772	0.905097	0.836039	0.905097	0.986739
3	ELLIPTICAL	0.817910	0.861635	0.839204	0.861635	0.989597
4	MERGER	0.902261	0.973249	0.936412	0.973249	0.996672
5	SPIRAL_ACW	0.937763	0.982812	0.959759	0.982812	0.994783
6	SPIRAL_CW	0.936911	0.988759	0.962137	0.988759	0.995536
7	UNCERTAIN	0.629870	0.308426	0.414088	0.308426	0.941670
Average	Average	0.827881	0.818516	0.814458	0.818516	0.975729

# 4 Discussion

## Comparative Analysis with Previous Work

A similar project was carried out by Zhu et al. (2019), titled *Galaxy morphology classification with deep convolutional neural networks* [19]. Like this thesis, their work used images from the Galaxy Zoo 2 project [9], based on data from the Sloan Digital Sky Survey [20]. While both studies applied CNNs to classify galaxy morphologies, there are clear differences in dataset size, number of classes, architecture, and hardware used.

Zhu et al. began with 61,578 galaxy images, each labeled using modified vote fractions from Galaxy Zoo 2. After applying strict thresholds for clean classification, their dataset was reduced to 28,790 images, covering five morphology types: completely round, in-between, cigar-shaped, edge-on, and spiral. These were split into 25,911 for training and 2,879 for testing. In this thesis, a dataset of 120,000 images was used and divided into eight categories, including spiral (clockwise and anti-clockwise), elliptical, edge, disk, merger, cosmological structure, and uncertain. The data was split into 96,000 training and 24,000 validation images using an 80:20 stratified split. These classes were selected to follow Hubble’s sequence and de Vaucouleurs’ refinement as close as possible.

They implemented a variant of residual networks (ResNets) and trained it on a *Tesla V100 GPU*, with the total training time reported as 31.5 hours. In this project, a simpler CNN with three convolutional blocks was developed, trained on a regular laptop with a virtual machine with two *RTX 3060 GPUs*. Training was completed in under one hour.

Class	ResNets Zhu et al. (2019)	Our Model
Clockwise Spiral (Spiral)	0.9770	<b>0.9888</b>
Anticlockwise Spiral (Spiral)	0.9770	<b>0.9828</b>
Disk (In-Between)	0.9442	<b>0.9780</b>
Merger		<b>0.9732</b>
Edge (Edge-on)	<b>0.9436</b>	0.9051
Elliptical (Completely round)	<b>0.9668</b>	0.8616

Table 4.1: Accuracy comparison between ResNets (Zhu\_2019) and our model.

In this thesis, the model achieved a validation accuracy of 86.2% and 83.3% on a rotation-augmented test set that included all label categories. Performance was especially strong for morphological classes, with average AUC scores above 0.98 and F1 scores exceeding 0.83.

Excluding the merger class, which does not represent a true morphological type, and the artificially defined uncertain class, the average accuracy across our remaining six morphological categories reached 94.8%, as shown in Table 3.1. In comparison, the project by Zhu et al. reported an average accuracy of 88.36% across their five morphological classes described earlier. This difference highlights the improved performance of our model on clearly defined galaxy morphologies.

In summary, while Zhu et al.’s project demonstrates the effectiveness of deep learning when applied with large datasets and advanced hardware, this thesis shows that it is possible to achieve strong results with a simpler model and limited resources. Moreover, by attempting to follow the structure of Hubble’s tuning fork and de Vaucouleurs’ refinements, this project aims not only for technical performance, but also for meaningful classification that aligns with established astronomical frameworks. Notably, the model developed in this work achieves better accuracy for most of the classes compared to Zhu et al.’s approach, highlighting the potential of efficient, well-structured models even under constrained conditions.

## 5 Conclusion

And with this, we conclude our study on the usefulness of Convolutional Neural Networks for morphological classification of galaxies. This thesis has drawn from various scientific fields, exploring the intersection between mathematics, machine learning, and astronomy.

The process began with the most basic concepts of group theory and culminated in the development of a program capable of automatically classifying galaxies, offering potential support to the astronomical community in the task of analyzing large volumes of data. This journey has not only allowed for the implementation of a useful tool but also provided an opportunity to reflect on the mathematics underlying these machine learning models.

As possible lines for improvement, one could delve even deeper into the mathematical analysis of each stage of the model. Furthermore, from a computational perspective, the training set and the model's complexity could be increased to improve classification accuracy. Another interesting direction would be to explore more sophisticated data augmentation techniques and more precise labelling.

In short, this work opens the door to new research at the intersection of machine learning and astronomy, demonstrating how mathematics can serve as a bridge to tackle large-scale scientific problems and contribute to our understanding of the universe.



# Appendix

## Notes on Contributions and Tools

Throughout this thesis, several external contributions and tools have been employed to support the development and presentation of the work. The cover design, as well as the galaxy image shown at the end of the Introduction and Conclusion chapters, were kindly designed by Francesca Gonzalez. Gonzalo Vela provided help with connecting to the virtual machine via SSH and using rsync for file transfers. Both friends of the author.

For the development of the Convolutional Neural Network model, the project idea began with an example implementation available at Kaggle [17], upon which our final program was constructed and adapted for the specific needs of galaxy morphology classification.

During the coding process, Microsoft’s GitHub Copilot [18] was used as a debugging tool. Regarding the preparation of this document, some images were transformed using image editing from ChatGPT [10] (as already referenced on the project), which was also employed for assistance with LaTeX code, formatting, and grammar checking in some paragraphs to ensure consistency and readability.

While these tools and contributions provided valuable support, all final implementations, analysis, and conclusions presented in this thesis reflect the author’s own work and understanding.

# Bibliography

- [1] Hubble, E. 1943, ApJ, 97, 112
- [2] Vermij, R. G. 2020, *A History of Western Science*, Routledge, 1st ed.
- [3] NASA Hubble Cepheid. *Hubble Views the Star that Changed the Universe*, <https://science.nasa.gov>
- [4] Buta, R. J. 2014, *Galaxy Morphology, Chapters 1–3*, in *Planets, Stars and Stellar Systems, Vol. 6*, ed. T. D. Oswalt & W. C. Keel (Dordrecht: Springer)
- [5] Buta, R. J. 2014, *Galaxy Morphology, Chapters 16–17*, in *Planets, Stars and Stellar Systems, Vol. 6*, ed. T. D. Oswalt & W. C. Keel (Dordrecht: Springer)
- [6] Sandage, A. 1961, *The Hubble Atlas of Galaxies*, Carnegie Inst. of Wash. Publ. 618
- [7] de Vaucouleurs, G. 1959, Handbuch der Physik, 53, 275
- [8] Willett, K. W., Lintott, C. J., Bamford, S. P., Masters, K. L., Simmons, B. D., Casteels, K. R. V., Edmonson, E. M., Fortson, L. F., Kaviraj, S., Keel, W. C., Melvin, T., Nichol, R. C., Raddick, M. J., Schawinski, K., Simpson, R. J., Skibba, R. A., Smith, A. M., & Thomas, D. 2013, MNRAS, <https://doi.org>
- [9] Galaxy Zoo Team. *Spectra and No Spectra Datasets*, <https://data.galaxyzoo.org>
- [10] OpenAI. *ChatGPT*, <https://openai.com>
- [11] Bronstein, M. M., Bruna, J., LeCun, Y., Szlam, A., & Vandergheynst, P. *Geometric Deep Learning*, <https://geometricdeeplearning.com>

- [12] Dummit, D. S., & Foote, R. M. 2004, *Abstract Algebra*, 3rd ed., John Wiley & Sons
- [13] Vrije Universiteit Amsterdam. *Machine Learning*, Department of Computer Science
- [14] Pedregosa, F., Varoquaux, G., Gramfort, A., et al. (2011). Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*
- [15] Van der Walt, S., Schönberger, J. L., Nunez-Iglesias, J., et al. (2014). Scikit-image: Image processing in Python
- [16] Abadi, M., Barham, P., Chen, J., et al. (2016). TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*
- [17] Kanncaa1. *PyTorch Tutorial for Deep Learning Lovers*: <https://www.kaggle.com/>
- [18] GitHub Copilot. Microsoft. Available at: <https://github.com/features/copilot>
- [19] X.-P. Zhu, J.-M. Dai, C.-J. Bian, Y. Chen, S. Chen, and C. Hu. *Galaxy morphology classification with deep convolutional neural networks*. arXiv:1807.10406 (2019)
- [20] D. G. York et al. *The Sloan Digital Sky Survey: Technical Summary*. The Astronomical Journal, 120(3):1579–1587, 2000. DOI: 10.1086/301513