
Observing Hydrodynamical Simulation with Deep Learning

Simin Hosseinzadeh

A thesis presented to

Faculty of Physics

Ludwig Maximilians University of Munich

by

Simin Hosseinzadeh

Munich, Germany, 2021

Main Supervisor: Dr. Benjamin Moster

Acknowledgments:

I would like to thank my supervisor, Dr. Benjamin Moster, for his dedicated support and guidance throughout this project. Dr. Moster continuously provided encouragement and was always willing and enthusiastic to assist in any way he could throughout the research project.

Abstract

Numerical simulations are traditionally the most popular tools for studying the physics of galaxy formation. This is mainly due to the complex and non-linear equations governing such systems, preventing exact analytical solutions. However, simulation models are computationally expensive, and simulating a single galactic system can take several months. In the era of machine learning and artificial intelligence, it is not hard to imagine that simulation models can be supplemented with powerful machine learning algorithms to reduce their computational complexity. This is precisely what has been investigated in the present dissertation. We build a learning algorithm that takes a set of images representing some intrinsic properties of galaxies and outputs a new set of images representing galaxies' observational properties. To achieve this goal, we use convolutional neural networks, and in particular, the U-Net architecture. It will be shown that our deep learning method successfully generates accurate images in a time-efficient manner.

Contents

1	Introduction	1
1.1	Cosmology and Galaxies	1
1.2	Numerical Simulations in Astrophysics	3
1.3	Outline	5
2	Deep Learning and Neural Network	6
2.1	What is Deep Learning	6
2.2	Define a Neural Network	9
2.3	Tackle a problem with NN	13
3	Convolutional Neural Network and U-Net	15
3.1	Convolutional Neural Network	16
3.1.1	Convolution Operation	17
3.1.2	Padding	19
3.1.3	Stride	20
3.1.4	Convolution on RGB image	20
3.1.5	Convolutional Layer	21
3.1.6	Pooling Layer	22
3.2	U-Net	22

4	U-Net Meets Astrophysics	26
4.1	Dataset	27
4.2	Preprocessing	29
4.3	Modeling	30
4.4	Evaluation	34
5	Conclusion	39

List of Figures

2.1	DL algorithms can be construed as a subset of ML algorithms which themselves belong to a more broad class of research field called artificial intelligence.	7
2.2	A single neuron performs a two-step computation: 1) Affine transformation and 2) non-linear activation.	10
2.3	From left to right, depicting the ReLu, tanh, and sigmoid activation functions respectively.	11
2.4	A NN with two hidden layers: the values in each node are computed layer by layer.	12
3.1	The convolution between an input image and a feature detector is visualized.	18
3.2	The basic idea of padding is depicted: the pink boarder around the matrix represents the zero-valued pixels.	19
3.3	Stride controls the way the detector slips over the image. Three cases are depicted above.	20
3.4	Convolution on an RGB image is achievable by doing the convolution at each channel independently.	21
3.5	A visualization of how a convolutional layer interacts with an image.	22

3.6	Max Pooling and Average Pooling are applied to a feature map so that a reduce map can be obtained.	23
3.7	Example of a U-Net architecture with 32×32 pixels in the lowest resolution. The blue boxes correspond to multi-channel feature maps. The white boxes represent copied feature maps. The arrows denote the different operations [1].	24
4.1	The feature (left) and label (right) images of galaxies $g3.06e11$ and $g1.37e11$ taken respectively from angles 60° and 84° . The map number for the top left image is 5 (mean metallicity), while the map number for the down left is 6 (mean temperature). The filter numbers for both images are 0 (u), 1 (g) and 2 (r).	28
4.2	A typical structure of a ML algorithm.	29
4.3	The encoding path of our algorithm consists of a series of convolutional and pooling layer along with the dropout technique.	31
4.4	The decoding path of our algorithm consists of a series of transposed convolutional layer along with the dropout technique and skip connections. Notice that, except for the final convolutional layer, which has the sigmoid activation function, the activation functions in all convolutional layers were chosen to be ReLU.	32
4.5	The loss functions for training and validation sets per epoch. There is neither underfitting nor overfitting.	34
4.6	A visualization of some actual and predicted images. Left images are the actual ones, while right images are the predicted ones. From top to down, these images are for galaxies $g2.42e11$, $g2.79e12$, $g3.06e11$ taken from angles 26° , 73° , 60° , respectively.	36
4.7	The relative residual for all images and all filter numbers.	37

-
- 4.8 The left column represents the actual images; the middle column contains the predicted images, and the right column shows the relative residual for these images. The best estimation to the worst estimation are sorted from top to down. 38

Chapter 1

Introduction

1.1 Cosmology and Galaxies

Physical cosmology postulates two underlying assumptions regarding the spatial structure of the Universe: *homogeneity* and *isotropy*. The two assumptions respectively state that, on sufficiently large scales, there is no preferred position and direction in the Universe. By making these assumptions and the general theory of relativity, a cosmological model, called Λ CDM model, can be formulated.

Λ CDM model predicts many properties observed in the Universe and is based on the notions of dark energy (Λ) and cold dark matter (CDM). Although the Λ CDM model has no explicit physical theory for the origin of dark matter or dark energy, it has been widely accepted because of successfully explaining several phenomena such as [2]:

- the existence and structure of the cosmic microwave background radiation
- the large-scale structure in the distribution of galaxies
- the observed abundances of hydrogen, helium, and lithium
- the accelerating expansion of the Universe.

The Λ CDM model explains structure formation as a hierarchical process, meaning that larger structures are formed through a continuous merging of smaller structures. Such a hierarchical process can be exploited to account for galaxy formation. To characterize galaxy formation, three key processes are typically used:

1. **Primordial collapse:** the collapse of individual gas clouds early in the history of the Universe, leading to the formation of galaxies [3].
2. **Hierarchical clustering:** the formation of large galaxies through the merging of smaller ones [4].
3. **Secular evolution:** formation as a result of internal processes, such as the actions of spiral arms and bars [5].

Depending on a galaxy's Hubble type, each of these processes might have a different contribution degree to forming any particular galaxy. For instance, early-type (elliptical and S0) galaxies are thought to have formed mainly due to primordial collapse,¹ while late-type (spiral and irregular) galaxies are formed mainly due to secular evolution [6].²

Although the processes involved in the formation of galaxies have been identified, there is no definitive model of how galaxies form. It is still unclear what the relative importance of the three processes is for each galaxy type. One of the challenges in constructing such a complete model is the need to match observations of galaxies in the current Universe with those in the early Universe. Theoretical simulations can help us achieve this goal.

For example, in the primordial collapse scenario, as a gas cloud shrinks, it turns into a complex and messy mixture of cold dense clouds and hot diffuse gas. It

¹In addition to primordial collapse, mergers are known to play an essential role in their assembly.

²Though, primordial collapse, and mergers can also play a role for them.

is almost impossible to sketch the exact behavior of such complex systems via detailed calculations. Instead, theoretical simulations can provide a good insight into these complex systems using a set of initial conditions and the conservation laws of physics (e.g., conservation laws of mass, energy, and momentum).

1.2 Numerical Simulations in Astrophysics

Numerical simulations are the most popular tools for studying the galaxy formation processes, mainly due to the complex and non-linear nature of equations returned by the Λ CDM. Numerical simulations have been used in several astrophysical research fields such as density profiles of dark matter haloes [7], the existence of dark matter substructure [8], the clustering properties of galaxies [9], and the gas temperature of galaxy clusters [10].

Following [2], four requirements should be satisfied by a successful galaxy formation model:

1. Inclusion of gas and its relevant processes in the model.
2. Achieving a high resolution, to capture all effects shaping the galaxy morphology.
3. Inclusion of the cosmological background.
4. Being computationally efficient (optional).

Numerical simulations for galaxy formation can be divided into two main categories:

- **Collisionless N-body:** simulations that posit an initial configuration over a set of point masses and evolve them according to the laws of gravity [11].

- **Hydrodynamical:** simulations that not only evolve point masses but also incorporate gas physics by including pressure forces and the formation of stars from gas [12].

Both N-body and hydrodynamical methods satisfy only two of the three necessary requirements for galaxy formation models; N-body methods can provide high-resolution models and include cosmological background, and traditional hydrodynamical methods include gas physics and cosmological background. To overcome this difficulty, [13] supplemented hydrodynamical methods with *zoom-in* technique, therefore resulting in a method satisfying all the three required conditions.

The basic idea of the *zoom-in* technique is to increase the resolution by confining most of the particles initially in a selected region within a cosmological volume. This volume includes regions with varying mass and spatial resolution, which decreases with the distance from a selected object. In this way, the resolution would be more optimal; however, the approach is computationally expensive, and simulating a single galactic system takes several months.³

From an algorithmic point of view, there is an alternative way to construe the nature of numerical simulations. A simulation model takes an initial state of a system as input, evolves the given state according to some physical processes, and eventually returns the system's final state as output. There are many machine learning algorithms that are designed to map the input data into the output data, without requiring explicit transformation rules. It is thus reasonable to think that machine learning algorithms can reduce the computational expense and complexity of numerical simulations for galaxy formation. This is in fact what I have done in this thesis.

³In addition to numerical simulations, there is another approach, called the semi-analytic model (SAM), that studies the evolution of dark matter and baryonic matter separately. However, as this research area is beyond the scope of this thesis, I do not discuss it further.

Given a set of images representing some *intrinsic* properties of galaxies, the aim is to achieve a deep learning algorithm that can estimate a new set of images representing some *observational* properties of galaxies.⁴ Both datasets (image sets) were generated via a hydrodynamical simulation incorporating the zoom-in technique. As will be shown in Chapter 4, our deep learning method successfully generated accurate images in a time-efficient manner.⁵

1.3 Outline

The remainder of this dissertation is organized as follows. Chapter 2 investigates the basic concepts of deep learning and explains how to tackle a problem with neural networks systematically. Chapter 3 focuses on a neural network architecture suitable for image datasets, namely convolutional neural networks and in particular U-Net. Chapter 4 concretely investigates the deep learning algorithm that is implemented in this project and evaluates the quality of its learning procedure from different perspectives. Finally, in Chapter 5, I draw some concluding remarks and point out some interesting ideas for further research in this area.

⁴Chapter 4 discusses the dataset and its physical parameters in more detail.

⁵The source code of this project can be found at: <https://github.com/siminhs/Thesis>.

Chapter 2

Deep Learning and Neural Network

This chapter will investigate the basic concepts of deep learning. As most of these concepts will be used in the next chapters, I will standardize the corresponding notations. To do so, Section 2.1 will first explain what we mean by deep learning, how it is related to machine learning and neural networks, and also the origins of deep learning in statistical modeling. Section 2.2 will then focus on a single neuron and explain how it computes an output based on the data fed into it as the input. The concepts of activation function, hidden layer, loss function, forward and backward propagation will be discussed in the same section. Finally, Section 2.3 will explain how to systematically tackle a problem using the neural networks in an abstract way.

2.1 What is Deep Learning

Machine learning (ML) is a branch of artificial intelligence (AI) focused on building algorithms that improve automatically through experience. That is, ML algo-

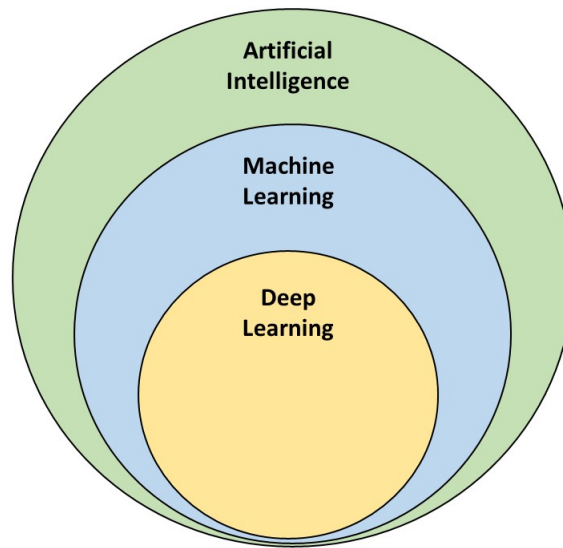


Figure 2.1: DL algorithms can be construed as a subset of ML algorithms which themselves belong to a more broad class of research field called artificial intelligence.

gorithms can be trained on sample data, known as “training data,” to make predictions that can be improved iteratively. Deep learning (DL) is a sub-field of ML, which uses artificial neural networks (NNs) to construct computational models [14]. Hence, DL and NNs are mostly equivalent, and they are used interchangeably in many contexts, see Figure 2.1.

DL has developed rapidly in the past years. Nowadays, it can be applied to a great variety of different data types such as image data [15], video data [16], text data [17], speech data [18], or tabular data [19].

The general scheme of a DL problem is the challenge of predicting some output class given some input data. The DL algorithm is then supposed to *learn* the (hidden) patterns in the input and accurately predict the outcome. The said general scheme is very similar to ML. However, if the input data has the following

three properties, a DL algorithm will probably outperform most ML algorithms [20]:

1. The data is high dimensional; it has a lot of features.
2. Each single feature is not very informative, but a combination of features might be.
3. A large amount of training data is available.

These three criteria lead to a paradigmatic DL application, i.e., *image processing*. In image processing, a training dataset consists of images from which our DL algorithms can recognize patterns. For instance, a DL algorithm can be trained on image datasets to distinguish between cat and dog. In a more general context, there are three main advantages of using DL for computationally expensive problems:

- **Large Datasets:** There exist several high-performance computing DL platforms such as TensorFlow [21], PyTorch [22], and Keras [23] for model fitting. Two key features of these platforms are mini-batch training (the ability to perform the computation in smaller data batches instead of requiring the whole dataset to be loaded into the memory) and automatic differentiation (calculating the gradients used for optimization in a computationally efficient way).
- **Non-Convex Optimization:** In practice, optimization problems are typically non-convex and might have several local extrema. DL algorithms are able to handle such non-convex problems making them powerful tools.
- **Different Data Formats:** Most of the traditional computational methods cannot handle an input training set composed of several data formats (e.g., tabular data and image data). A distinctive feature of NNs is their ability to estimate the joint effects of such mixed data types.

2.2 Define a Neural Network

Generally speaking, a NN can be seen as a calculation rule or an algorithm to obtain an output variable y given some input variables $\mathbf{x} = (x_1, \dots, x_p)$. In more practical terms, NNs are non-linear statistical data computational tools that can be used to model complex relationships between inputs and outputs or to find patterns in data.

A NN is composed of some *nodes*, ordered in *layers*, and connected via *edges*. Each node (also known as a *neuron* or *perceptron*) has a real value and each edge is associated with a real-valued weight [14]. To understand how a NN works, it is instructive to first focus on how a single node works.

At each node, the weighted sum of the values from the previous nodes of the previous layer is computed and then transformed non-linearly (figure 2.2). Thus, a neuron performs a 2-step computation [20]:

1. **Affine Transformation:** the weighted sum of inputs plus a bias term is computed:

$$z = \sum_{i=1}^p w_i x_i + b \quad (2.2.1)$$

where w_i s are the weights, and b is the bias. The weights determine the influence of each feature on our prediction, and the bias indicates what value the prediction should take when all the features are zero.

2. **Non-linear Activation:** a non-linear transformation is applied to the computed weighted sum:

$$\hat{y} = \tau(z) = \tau\left(\sum_{i=1}^p w_i x_i + b\right) \quad (2.2.2)$$

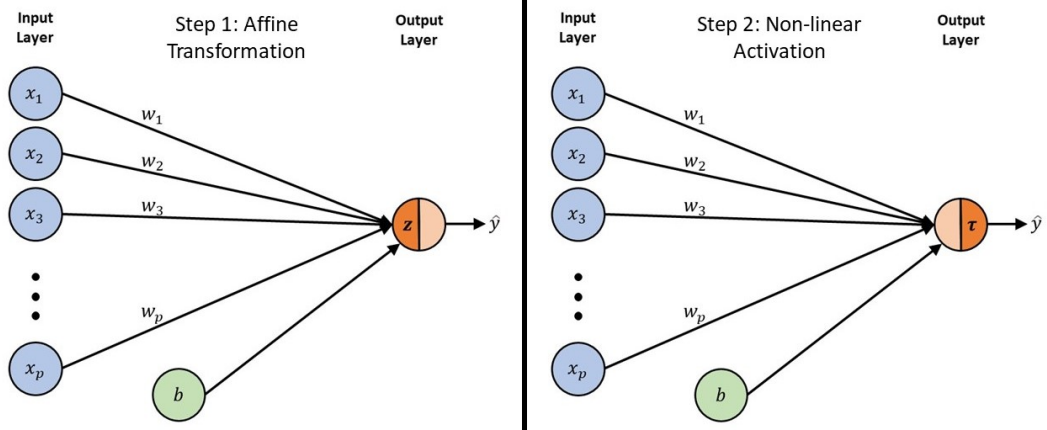


Figure 2.2: A single neuron performs a two-step computation: 1) Affine transformation and 2) non-linear activation.

Remark 2.1. *Collecting all features into a vector $\mathbf{x} \in \mathbb{R}^p$ and all weights into a vector $\mathbf{w} \in \mathbb{R}^p$, the two-step computation done by a single neuron can be expressed in the following compact form:*

$$\hat{y} = \tau(\mathbf{x}^\top \mathbf{w} + b) \quad (2.2.3)$$

The second step of the aforementioned procedure (i.e., the non-linear transformation) is done via *activation functions*. While, in principle, one can define an arbitrary activation function, there are a few standard functions widely used in DL literature. The Rectified Linear Unit (ReLU), Hyperbolic Tangent (Tanh), and the sigmoid functions are three well-known examples in this respect; see figure 2.3.

So far, we have discussed a network with only one single layer that consists of a single neuron. To make the network *deep*, multiple layers, each with several neurons, are added between the input and the output layers. These layers are then called *hidden layers*. The greater the number of layers, the “deeper” the network.

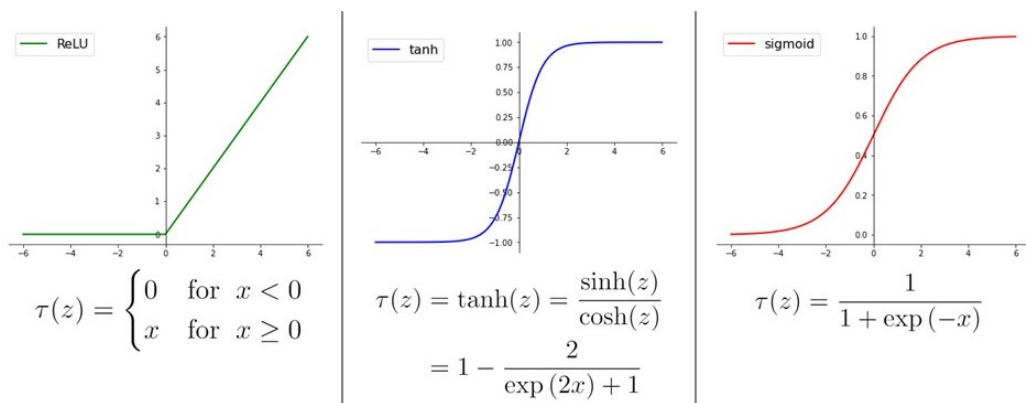


Figure 2.3: From left to right, depicting the ReLu, tanh, and sigmoid activation functions respectively.

Figure 2.4 depicts a sample NN architecture with two hidden layers. The values in each node of this network are computed layer by layer. Firstly, the values of each neuron at the hidden layer one are computed based on the values of the input nodes (affine transformation plus the non-linear activation). Then, the neurons' values in the second hidden layer are computed based on the first hidden layer. Finally, the output is computed based on the second hidden layer's values.

Remark 2.2. *Not all nodes in one layer have to be connected with all nodes of its adjacent layers. One can flexibly specify how a node should be connected with the other nodes.*

Remark 2.3. *The activation functions need to be non-linear. Otherwise, the network can only learn linear decision boundaries. In other words, in this situation, adding different layers to the networks cannot increase the model complexity, and the prediction would be a linear combination of the input as in the single-layer perceptron.*

Similar to other ML algorithm, a key ingredient of a NN is to fit a given data with our model: we want to systematically find optimal values for our model parameters

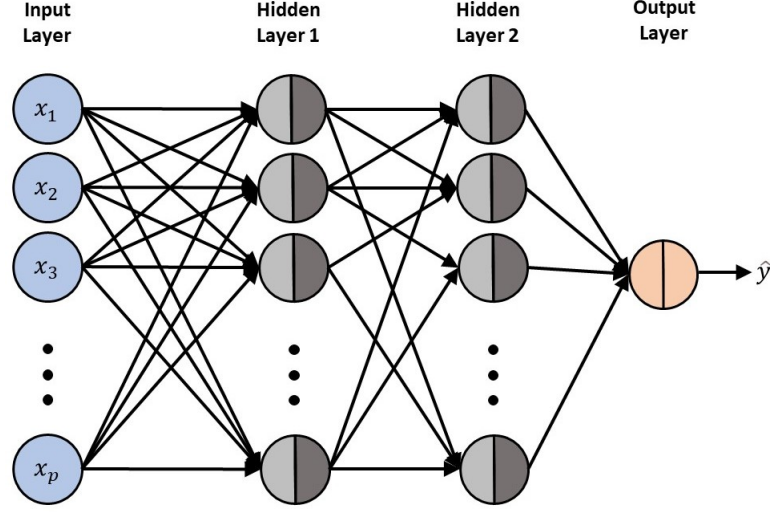


Figure 2.4: A NN with two hidden layers: the values in each node are computed layer by layer.

(weights and bias terms). To this end, let us first introduce the notions of the loss function and the empirical risk. The *loss function* is a measure of fitness that quantifies the distance between the actual value of target y and its corresponding predicted value \hat{y} . The loss is usually a non-negative number where smaller values indicate better predictions. Given a loss function and n observations, the *empirical risk* is sum of the loss contribution of all observations. More precisely, given n observations $(y_i, \mathbf{x}_i), i = 1, \dots, n$, where $\hat{y}_i = f(\mathbf{x}_i)$ is the model prediction for the i -th observation, the empirical risk is computed as follows [24]:

$$\mathcal{R}_{\text{emp}}(f) = \sum_{i=1}^n \mathcal{L}(y_i, f(\mathbf{x}_i)). \quad (2.2.4)$$

While the negative (log-)likelihood is usually taken as the loss function in many settings, the mean squared error (MSE) is used as the loss in regression tasks:

$$\mathcal{L}(y_i, \hat{y}_i) = \mathcal{L}(\mathbf{w}, b) = \frac{1}{2} (\hat{y}_i - y_i)^2. \quad (2.2.5)$$

Having these notions in hand, the optimal model parameters are those which minimize the empirical risk function. In other words, when training a model, the

goal is to find parameters (\mathbf{w}^*, b^*) under the following condition:

$$\mathbf{w}^*, b^* = \underset{\mathbf{w}, b}{\operatorname{argmin}} \mathcal{R}_{\text{emp}}(f) = \underset{\mathbf{w}, b}{\operatorname{argmin}} \mathcal{R}_{\text{emp}}(\mathbf{w}, b) \quad (2.2.6)$$

The key technique for optimizing nearly any DL model consists of iteratively reducing the empirical risk by computing the gradient of the risk and updating the parameters in the direction that incrementally lowers this quantity. This idea is the essence of the *gradient descent* algorithm [20]. Consequently, training of NNs is composed of two iterative steps:

- **Forward Propagation:** The information of the inputs flows through the model to produce a prediction, based on which the empirical loss is computed.
- **Backward Propagation:** The information of the prediction error flows backward through the network and is used to update the weights (and bias terms) in a way that the risk reduces.

2.3 Tackle a problem with NN

As described in the previous section, the central idea of NNs is based on the foundations of ML. Hence, each NN can be characterized in terms of three elements: 1) hypothesis space¹, 2) the risk function, and 3) optimization method.

To tackle a specific problem with a NN, there are some usual techniques (e.g., increasing the complexity of a NN by adding further hidden layers or adding more complex types of layers). Apart from these techniques, one can systematically proceed with the following three steps to achieve the said goal [14]:

1. **Constructing a network architecture:** the hypothesis space of such an architecture should match with the data structure.

¹The hypothesis space \mathcal{H} describes the space of all possible models $f: \mathcal{H} = \{f : \mathcal{X} \rightarrow \mathbb{R}^g\}$, where g denotes the target dimension.

2. **Introducing a suitable risk:** in accordance with our computational goal, we should introduce a suitable risk function the minimization of which reflects our optimal solution.
3. **Exploiting a suitable optimization method:** to minimize the risk function, we should use an optimization algorithm to find the optimal weights and bias terms.

As will be shown in the next chapters, for this thesis (i.e., generating a set of images from another set of images), the said three steps convert into the following demands:

1. **CNN and U-Net:** for image processing tasks, convolutional layers yielding convolutional neural networks (CNNs) are proved to be very useful.
2. **Regression and not classification:** we would like to generate images based on images. So, the goal is not to classify or detect particular objects in the images. Hence, we define our loss function to be MSE.
3. **ADAM optimizer:** Adam is an optimization algorithm based on stochastic gradient descent and is widely used for training deep NNs [25]. We will use ADAM optimizer to minimize the risk.

Chapter 3

Convolutional Neural Network and U-Net

The last chapter sketched the general idea behind NNs and DL and asserted that to tackle a specific problem with a NN, one must carefully choose a suitable network architecture whose hypothesis space matches the data structure. As this thesis's main task is to work with image data, we focus on a particular class of neural networks called convolutional neural networks (CNNs). Due to reasons that will be explained later throughout this chapter, CNNs are suitable tools for analyzing images, with applications in image and video recognition, recommender systems, image classification, medical image analysis, and natural language processing. To explain the core ideas of CNNs, Section 3.1 will introduce the general operations and conceptions related to CNNs. After that, in Section 3.2, we will study the structure of a particular type of CNNs, called U-Net.

3.1 Convolutional Neural Network

CNNs are regularized versions of standard NNs (i.e., multilayer perceptrons) designed to be applied to image data. Standard NN is usually fully-connected (i.e., all neurons in one layer are connected to all neurons in the adjacent layers), and because of this, they are prone to *overfit*¹ data, especially when our data is very high-dimensional.

The basic idea of CNNs is to automatically extract particular features and hierarchical patterns from the given (image) data and use them to estimate optimal predictions and effects. By doing so, CNNs helps to overcome the risk of overfitting when working with images.

The input layer of a CNN is usually an image, each pixel of which is construed as one input node of the network. Like a standard NN, the input layer is then given to some hidden layers followed by an output layer. A CNN's hidden layers typically consist of a series of convolutional and pooling layers, whose roles are respectively to extract feature maps from a previous layer and reduce the dimensionality of the data input by selecting more robust features.

The following subsections will explain how convolutional and pooling layers operate on a given data, and in particular, on an RGB image.

¹A statistical model is said to overfit data whenever its parameters are too closely related to a particular training dataset [20]. In such a situation, the model cannot generalize well to new “unseen” data, such as the test dataset.

3.1.1 Convolution Operation

In purely mathematical terms, *convolution* is an operation taking two functions as input and returning a third function as output:

$$(f * g)(t) \stackrel{\text{def}}{=} \int_{-\infty}^{\infty} f(\tau)g(t - \tau)d\tau \quad (3.1.1)$$

where, f, g are the two input functions, and $(f * g)$ is the output function.

When it comes to neural networks, the notion of convolution needs to be modified since the convolution operation is now applicable to matrices instead of functions². Let \mathbf{X} be a 3×3 matrix representing a binary image, i.e., each pixel of the image is either black or white corresponding to zero and one. Also, let \mathbf{W} be a 2×2 matrix representing a filter or feature detector. Considering the feature detector as a window consisting of 4 cells, the convolution operation between \mathbf{X} and \mathbf{W} can be described as the following steps (also see Figure 3.1):

1. Place the feature detector over the input image beginning from the top-left corner and perform a dot product between the matching cells of the image and the feature detector.
2. Then, move the feature detector one cell to the right and repeat the same calculation.
3. After going through the whole first row, move the detector over to the next row and go through the same process.
4. Repeat this process until covering all the matching cells.

²The convolution between two matrices can be understood as the convolution between *representations* of two functions in an abstract space. This is similar to quantum mechanics, where we can denote a quantum state either as a matrix (density operator) or as a function (wavefunction).

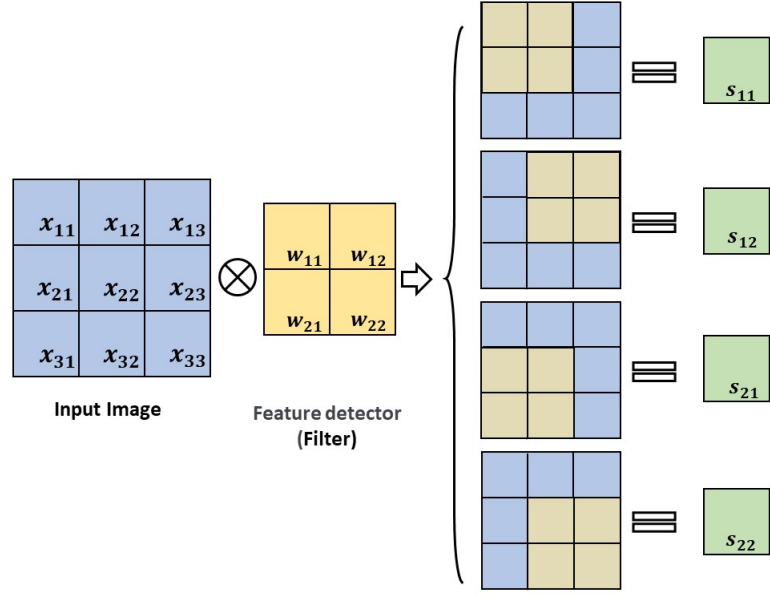


Figure 3.1: The convolution between an input image and a feature detector is visualized.

Hence, the convolution of the two matrices results in another matrix called *feature map* whose elements are computed as:

$$s_{11} = x_{11} \cdot w_{11} + x_{12} \cdot w_{12} + x_{21} \cdot w_{21} + x_{22} \cdot w_{22}$$

$$s_{12} = x_{12} \cdot w_{11} + x_{13} \cdot w_{12} + x_{22} \cdot w_{21} + x_{23} \cdot w_{22}$$

$$s_{21} = x_{21} \cdot w_{11} + x_{22} \cdot w_{12} + x_{31} \cdot w_{21} + x_{32} \cdot w_{22}$$

$$s_{22} = x_{22} \cdot w_{11} + x_{23} \cdot w_{12} + x_{32} \cdot w_{21} + x_{33} \cdot w_{22}$$

Remark 3.1. In general, by convolving input matrix $\mathbf{X}_{n \times n}$ with feature detector $\mathbf{W}_{f \times f}$, the resulting feature map is matrix $\mathbf{Y}_{m \times m}$ where $m = n - f + 1$.

It is important to emphasize that, because of reducing the dimension of the input image, the feature map leads to losing information. However, this information loss

is not a bad or unwanted phenomenon. In fact, the very purpose of the feature detector is to catch the essential information of the input image by excluding the unnecessary parts.

In practice, the convolution operation comes with a few *hyperparameters* (degrees of freedom which can be tuned so we can have an optimal gain from the network). *Padding* and *stride* are two examples of such hyperparameters, which will be introduced in the following.

3.1.2 Padding

In the standard convolution, pixels located in the corner of the input image are covered only once, while middle pixels are covered more than once. Such an asymmetric treatment toward the pixels causes the CNNs to lose the information laid on corners of the image. “Padding” has been introduced precisely to overcome this issue. The idea is to add some zero-valued dummy pixels to the image borders so that the corner and middle pixels get covered more equally; see Figure 3.2. Padding is said to *same* whenever the feature map has the same dimensions as the input image.

0	0	0	0	0
0	x_{11}	x_{12}	x_{13}	0
0	x_{21}	x_{22}	x_{23}	0
0	x_{31}	x_{32}	x_{33}	0
0	0	0	0	0

Figure 3.2: The basic idea of padding is depicted: the pink boarder around the matrix represents the zero-valued pixels.

3.1.3 Stride

Stride is another hyperparameter of convolution that adjusts the amount of the detector movement over the input image at each step. For example, if the stride is set to two, the detector will jump two pixels at each step. See figure 3.3 for further detail.

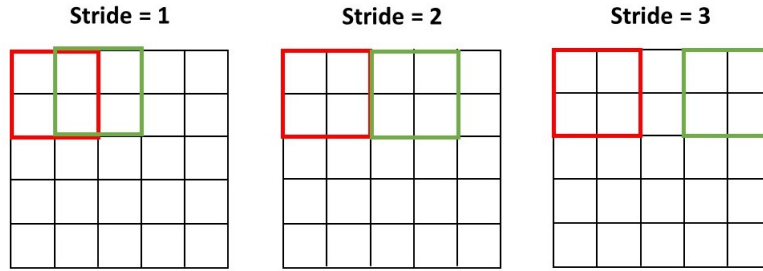


Figure 3.3: Stride controls the way the detector slips over the image. Three cases are depicted above.

Remark 3.2. Applying a $f \times f$ feature detector to an $n \times n$ input image with padding p and stride s results in a feature map with the following dimensions:

$$\left\lfloor \frac{n + 2p - f}{s} + 1 \right\rfloor \times \left\lfloor \frac{n + 2p - f}{s} + 1 \right\rfloor \quad (3.1.2)$$

3.1.4 Convolution on RGB image

While our discussion has been so far limited to binary images, in practice, we should typically deal with colored images (also known as RGB images).

To perform convolution on an RGB image, the idea is to consider three *channels* representing the three colors (i.e., red, green, and blue) constructing the image. The convolution is then performed at each channel independently via three independent feature detectors. Consequently, the number of channels in the input image must be the same as the number of detectors to be applied.

For a color image with three channels for red, green and blue, these three features maps are then stacked to create the final output of the convolution. Interestingly, one has the option of extracting different types of features from each channel. To do so, one should use different detectors at each channel and then stack the results.

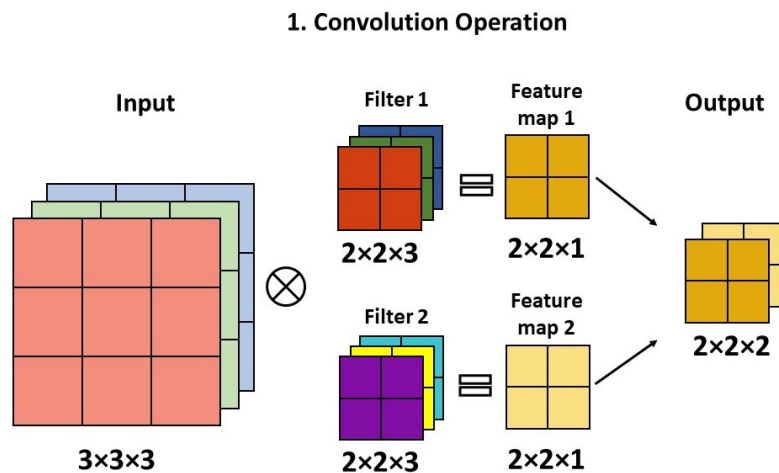


Figure 3.4: Convolution on an RGB image is achievable by doing the convolution at each channel independently.

3.1.5 Convolutional Layer

The convolution operation in CNNs plays the similar role as the weighted sum in standard NNs. This means that, in addition to the “linear” convolution, we need a non-linear transformation which can increase our model complexity. At each *convolutional layer*, a 2-step computations is thus performed:

1. **Convolution Operation**
2. **Non-linear Activation**

The commonly-used activation functions in CNNs (similar to NNs) are ReLU and Tanh functions, see Figure 3.5.

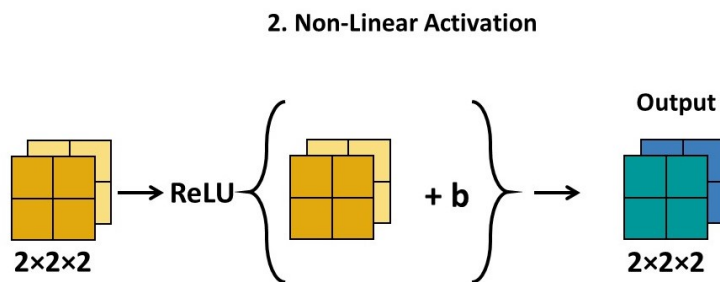


Figure 3.5: A visualization of how a convolutional layer interacts with an image.

3.1.6 Pooling Layer

In a CNN, a convolutional layer is usually followed by a *pooling layer*. The pooling layer reduces the size of the feature map. Hence, a pooling layer can be used to speed up computation. In the literature, there are two commonly-used types of pooling:

- **Max Pooling:** the maximum value from each patch of a feature map is selected to create a reduced map.
- **Average Pooling:** the average value from each patch of a feature map is selected to create a reduced map.

3.2 U-Net

U-Net is a particular type of CNNs originally developed for biomedical image segmentation in 2015 [1]. The architecture of U-Net is designed to work with fewer training images and to yield more precise segmentation. It is an effective tool, especially when the input and output images are of similar sizes, e.g., in image generation tasks such as super-resolution and colorization. Some variants and applications of U-Net are:

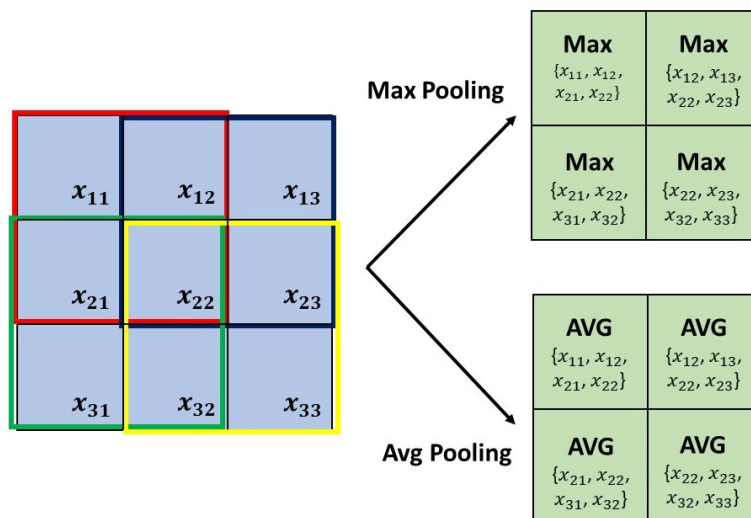


Figure 3.6: Max Pooling and Average Pooling are applied to a feature map so that a reduce map can be obtained.

- Pixel-wise regression used for pansharpening
- Learning dense volumetric segmentation from sparse annotation (3D U-Net)
- Image segmentation via combining it with VGG11 Encoder (TernausNet)

To explain the U-Net's main idea, it is instructive to compare its architecture with a usual CNN used for a classification task. In a typical CNN, the original image is taken as input and converted into a downsampled feature map via successively applying convolutional and pooling layers to it. By doing so, a CNN extracts the most useful information of the image that can be used for a classification task. Another way to explain this process is to say that a CNN reduces the spatial information of the original image while increasing the feature information.

U-Net's main idea is to supplement the aforementioned downsampling process with an upsampling procedure that increases the spatial information and decreases the feature information. By doing so, the U-Net's output is a generated image with

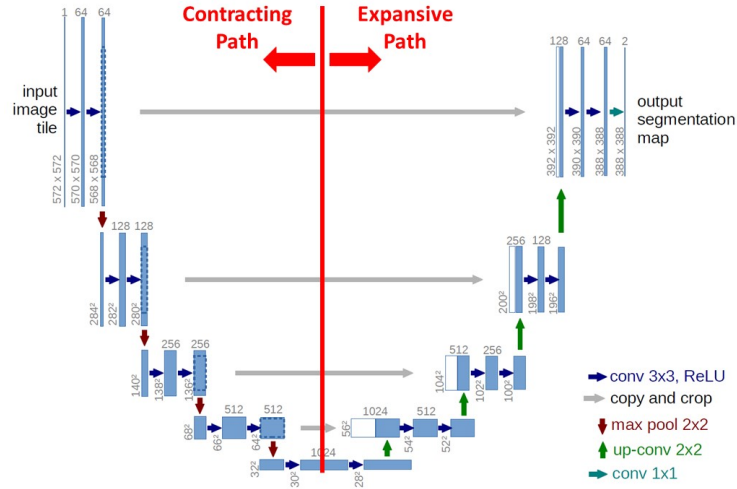


Figure 3.7: Example of a U-Net architecture with 32×32 pixels in the lowest resolution. The blue boxes correspond to multi-channel feature maps. The white boxes represent copied feature maps. The arrows denote the different operations [1].

almost the same dimensions as the input image. More importantly, the upsampling procedure allows the network to propagate context information to higher resolution layers. Consequently, U-Net is a U-shaped network composed of two paths:

1. downsampling/encoder path forming the left-hand side of the “U”
2. upsampling/decoder path forming the right-hand side of the “U”

Figure 3.7 visualizes an example for a U-Net architecture taking $572 \times 572 \times 1$ input images and returning $388 \times 388 \times 2$ output images. As can be seen, the input images are first downsampled in the contracting path (left) and then upsampled in the expansive path (right). To contract the image, several convolutional and (max) pooling layers are applied to the image in the first stage. Several upsampling and convolutional layers are then applied to the contracted image to expand it in the

second stage.

Remark 3.3. *The pooling layers are typically used to reduce an image's dimensions. Therefore, we need to replace the pooling operations with new upsampling operations to increase the output's resolution in the upsampling path.*

To expand the compressed image, one should use several transposed convolutions. The basic idea of these upsampling operations is to add pixels around and in-between the existing pixels to eventually reach the desired resolution at the end of the expansive path.

U-Nets have typically an additional component, called *skip connection*, enabling them to estimate more fine details. The skip connections cross from the same sized part in the downsampling path to the upsampling path, as depicted with the grey arrows in Figure 3.7. These connections allow the expansive path to be aware of the original pixels inputted into the model so it can combine the feature information with the high-dimensional spatial information from the contracting path.

Chapter 4

U-Net Meets Astrophysics

This chapter will concretely introduce the DL algorithm that I implemented in my project.¹ The rough idea is as follows. Given a set of *feature images* (which have been simulated based on some intrinsic properties of galaxies), the aim is to build a DL algorithm that generates a set of *label images* (which have been simulated based on some observational properties of galaxies). To do so, the algorithm should *learn* the similarity relationships between the two image sets. As will be shown in the following sections, a U-Net can successfully learn and estimate such relationships. It is worth mentioning that generating label images via numerical simulations is a highly time-consuming task. However, according to my experiments, DL can generate such images very accurately in a time-efficient manner. The structure of this chapter is as follows. Section 4.1 focuses on the given datasets and provides some physical and numerical details about them. Section 4.2 explains some preprocessing steps that prepare the dataset for the next step of the algorithm. Section 4.3 introduces the U-Net structure that I utilized. Finally, in Section 4.4, I examine different aspects of the trained algorithm, such as the quality of the predictions, the residuals, and the model loss.

¹The source code of my project can be found at: <https://github.com/siminh/Thesis>.

4.1 Dataset

Generally speaking, we have two sets of feature and label images from 20 galaxies taken from 11 different angles² ranging from 0° to 90°. Therefore, our dataset consists of 219 *samples* (i.e., image pairs). In the python code developed for this thesis, galaxies' names and images' angles are denoted by `system_names` and `projection_names`, respectively.

The objective is to train an algorithm that can generate label images from feature images. In other words, the algorithm should take feature images as input and return label images as output. To this end, the algorithm must learn the structural relationships holding among both image datasets.

In our dataset, each feature image is represented by 6 physical variables, including the stellar mass surface density, gas surface density, star formation rate, mean age, mean metallicity, and mean temperature. These variables are denoted by `map_names` in my python codes.

On the other hand, each label image is associated with 5 color filters, including ultraviolet (u), green (g), red (r), near-infrared (i), and infrared (z), denoted by `filter_names`. In practice, these filters are used to block out the incoming light from galaxies at all wavelengths except those around the wavelength they are designed to see. The following table shows the wavelengths at which these filters work the best [26].

²There is one exception here: due to some technical reasons, we have 10 sets of images for galaxy g1.77e12.

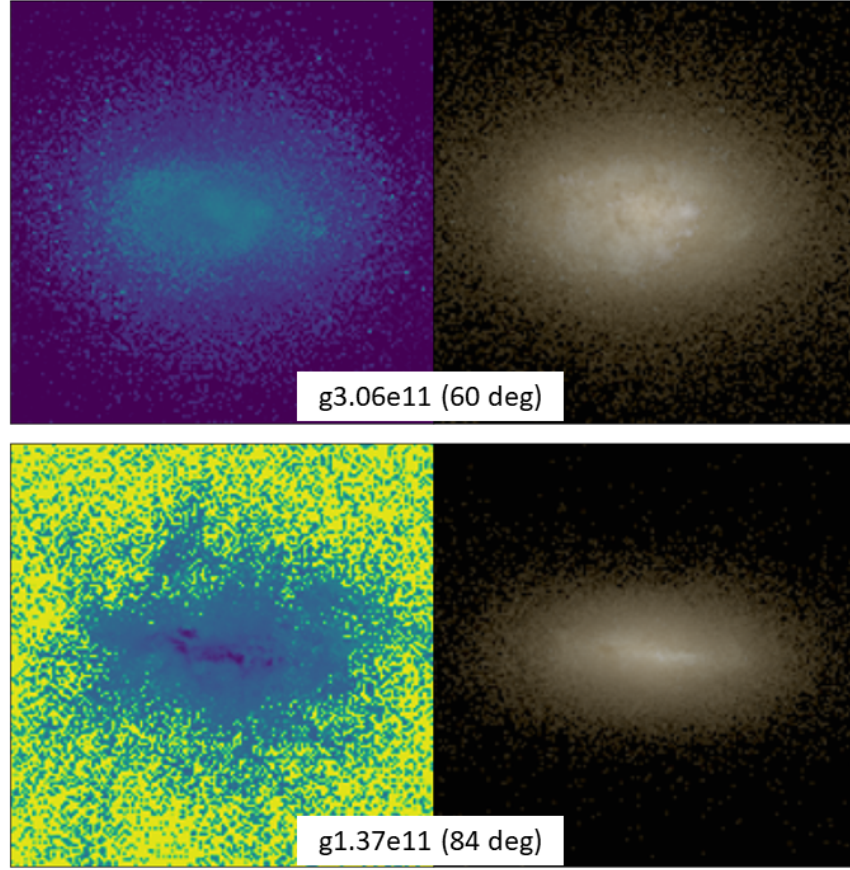


Figure 4.1: The feature (left) and label (right) images of galaxies $g3.06e11$ and $g1.37e11$ taken respectively from angles 60° and 84° . The map number for the top left image is 5 (mean metallicity), while the map number for the down left is 6 (mean temperature). The filter numbers for both images are 0 (u), 1 (g) and 2 (r).

Filter	Wavelength (Angstroms)
Ultraviolet (u)	3543
Green (g)	4770
Red (r)	6231
Near Infrared (i)	7625
Infrared (z)	9134

Note that the datasets were given in terms of two “HDF5” (Hierarchical Data Format) files, one for feature images and the other for label images. See Figure 4.1 in order to get a better insight into these images.

4.2 Preprocessing

A typical ML algorithm works as Figure 4.2. We take a dataset as input and perform some preprocessing steps to prepare the dataset for the next steps. In the estimation step, the algorithm predicts some outputs, the values of which are supposed to be as close as possible to the real labels. The algorithm iteratively compares the predictions with the real values by taking some feedback from the outputs. This section focuses on the preprocessing part of my algorithm.

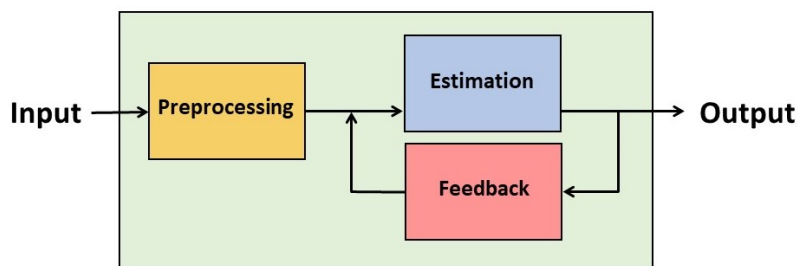


Figure 4.2: A typical structure of a ML algorithm.

I separately performed the preprocessing step once for the feature images and once for label image.

First of all, we defined some global values denoted by `min_value` and `max_value` and used them for globally normalizing all features and labels. We also resized all images into 192×192 pixels.

For normalizing the features, we computed the minimum and maximum values of all the 6 maps and put them in two NumPy arrays: (`min_feature` and `max_feature`). Based on the `min_feature`, we defined `cut_feature` as

10 `min_feature`. For normalizing the labels, we also considered minimum and maximum values and denoted them by `min_label` and `max_label`.

After normalizing and resizing all the images, we defined two empty NumPy arrays and concatenated features and labels with them. As a result, we obtained two arrays X and y with shapes $(219, 192, 192, 6)$ and $(219, 192, 192, 5)$, respectively. Notice that 219 shows the total number of samples, $(192, 192)$ represents the image pixels, 6 and 5 show the number of channels in the features and labels, respectively.

4.3 Modeling

As explained in Chapter 3, U-Nets are powerful tools when the input and output have (almost) similar sizes. This demand is satisfied by the present problem because the features and labels have the shapes of $(219, 192, 192, 6)$ and $(219, 192, 192, 5)$.

The **encoding path** of our U-Net takes images of size $192 \times 192 \times 6$ as input. These images were then applied to a series of convolutional and pooling layers whose duty was to decrease the height and width of each image and increase the channels. Note that a convolutional layer has many learnable parameters, whereas a pooling layer has no learnable parameter. The latter is just used to reduce the image size. Figure 4.3 depicts the structure of the encoding path of our U-Net.

Remark 4.1. *Besides the mentioned layers, we exploited the dropout technique, which is a regularization method used to avoid overfitting. The idea is to randomly throw away a preset ratio of the weights at each NN layer.*

The **decoding path** of our U-Net acts in an opposite manner with respect to the encoding path. It takes $24 \times 24 \times 256$ objects as input and applies a series of transposed convolutional and concatenating layers (i.e., skip connections) to them.



As a result, there is an increase in the objects' height and width while a decrease in the number of channels. So, we reach the size of $192 \times 192 \times 5$ at the end of the decoding path. Figure 4.4 depicts the structure of the decoding path of our U-Net.

It is important to note that it is not the case that a NN is updating its weight for each given sample (image). Instead, using the notions of *batch* and *epoch*, a NN performs the updates more efficiently and stably. In simple words, the number of batches refers to the number of samples taken by the algorithm before updating the weights. The number of batches can be chosen between one and the total number of samples. The epoch refers to the number of complete passes through the training dataset. The epoch can be chosen between one and infinity.

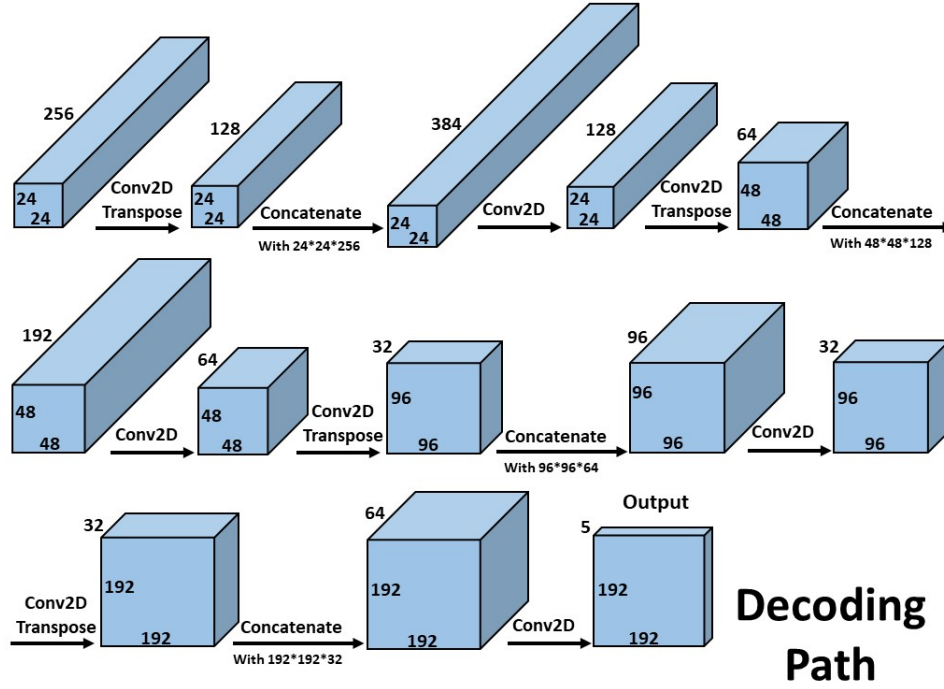


Figure 4.4: The decoding path of our algorithm consists of a series of transposed convolutional layer along with the dropout technique and skip connections. Notice that, except for the final convolutional layer, which has the sigmoid activation function, the activation functions in all convolutional layers were chosen to be ReLU.

To understand the difference between these two notions, let us consider an example of a dataset with 300 samples. By choosing a batch size of 10, the model divides the dataset into 30 batches (each of which containing 10 samples) and updates its weights after analyzing each 10 samples. By choosing the epoch to be 500, the model will go through the whole training set for 500 times and updates its weights for $500 \times 30 = 15000$ times during the training process.

Keras enables one to choose the batch size and the epoch straightforwardly. For my algorithm these values were set by `batch_size=10, epochs=300`.

We split our data into 80% for the training set and 20% for the validation set. By

calculating the loss for each of these sets, we examined the algorithm's quality and cared about the risk of underfitting and overfitting. As the estimation problem that we are dealing with is inherently a regression problem, we used the MSE (mean squared error) as the measure for the loss:

$$\mathcal{L}_{MSE} = \frac{1}{N} \sum_{i=1}^N (\mathbf{y}_i - \hat{\mathbf{y}}_i)^\top (\mathbf{y}_i - \hat{\mathbf{y}}_i), \quad (4.3.1)$$

where N is the number of samples.

Early stopping is a regularization technique used to avoid overfitting when training an algorithm with an iterative method [27]. This technique provides control over the number of iterations that can be run before the algorithm begins to overfit. The idea is to iteratively evaluate a specified quantity and stop the training process whenever the quantity's value does not improve anymore.

In Keras, early stopping is implemented via adjusting two parameters, `monitor` and `patience`. The former determines the quantity to be evaluated iteratively, and the latter denotes the number of epochs with no improvement after which training will be stopped. Both of these parameters can be adjusted within a function called `callbacks`. For our example, I chose the validation loss as the monitoring parameter with the patience to be 35:

```
tf.keras.callbacks.EarlyStopping(patience = 35, monitor = 'val_loss')
```

Besides implementing early stopping, `callbacks` contains some other functionalities to automate tasks after every epoch bringing about more controls over the whole training process. In particular, `ModelCheckpoint` saves the model after every epoch and can overwrite it whenever a better model is found. We used this functionality by setting `save_best_only = True`.

Finally, we fitted our model via `model.fit` which is a built-in function in Keras. The fitted model was then asked to predict some *unseen* data for us via `y_pre = model.predict`. That is, `y_pre` contains the predicted values corresponding to actual values `y`, and I would like these two values to be as close as

possible.

4.4 Evaluation

Having explained the implementation of this proposed model, it is now time to evaluate the quality of the learning algorithm. Figure 4.5 depicts the loss per epoch both for the training and validation sets. The results show that the model is fitted successfully without being underfitted or overfitted. This can be inferred from the following observations:

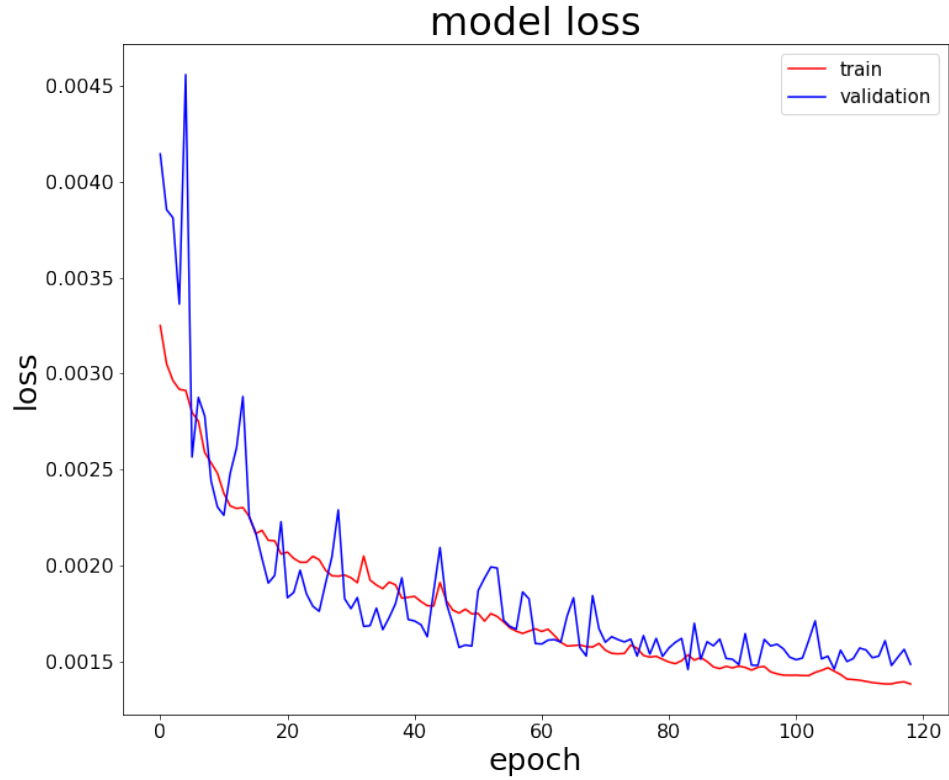


Figure 4.5: The loss functions for training and validation sets per epoch. There is neither underfitting nor overfitting.

- The losses are decreasing functions of the epoch: the more the model updates

the weights, the less the losses are.

- The training loss reaches a small value at the end of the learning process: the model is not underfitted.
- The validation loss is very close to the training loss, and hence the model has good generalizability: the model is not overfitted.

To check what has been learned by the model, let us compare some images generated (predicted) images with their corresponding actual images. Figure 4.6 depicts some of these images.

In addition to checking the loss and comparing the actual and predicted images, one can examine the *residuals* of the estimation. Let \mathbf{y}_i and $\hat{\mathbf{y}}_i$ be the actual and predicted values of the i -th sample (image). The relative residual of the estimation is computed as follows:

$$\epsilon_i = \left\| \frac{\mathbf{y}_i - \hat{\mathbf{y}}_i}{\mathbf{y}_i} \right\|_2. \quad (4.4.1)$$

In my python code, function `res_calculator` calculates the relative residual for each sample and each filter. Figure 4.7 shows the said residuals for all the images and all filter numbers.

Interestingly, the ultraviolet filter has the lowest residuals and hence the best estimations nearly for all images. The worst estimation with the highest residuals corresponds to the infrared filter. More generally, it seems that the accuracy of our estimation is correlated with the wavelength interval allowed by the filters. Providing the physical reason(s) for such a phenomenon can be an interesting topic of further research.

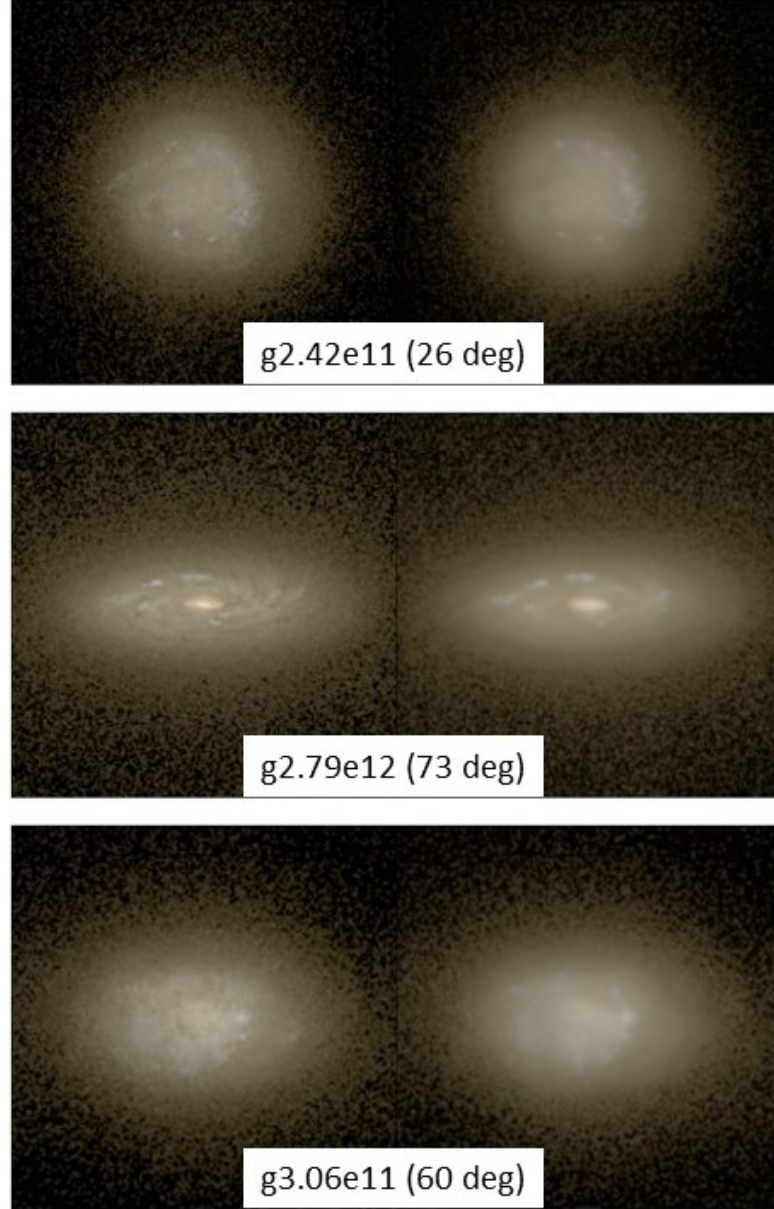


Figure 4.6: A visualization of some actual and predicted images. Left images are the actual ones, while right images are the predicted ones. From top to down, these images are for galaxies $g2.42e11$, $g2.79e12$, $g3.06e11$ taken from angles 26° , 73° , 60° , respectively.

Another way to analyze the implications of the relative residuals is to check the images with the lowest and highest residuals. For the present case, the lowest residual (the best estimation) and the highest residual (the worst estimation) correspond to image numbers 52 and 75. These image numbers correspond to galaxies $g1.59e11$ and $g1.92e12$ taken from angles 78° and 90° .

Remark 4.2. *The lowest and highest residuals remain the same for all filters.*

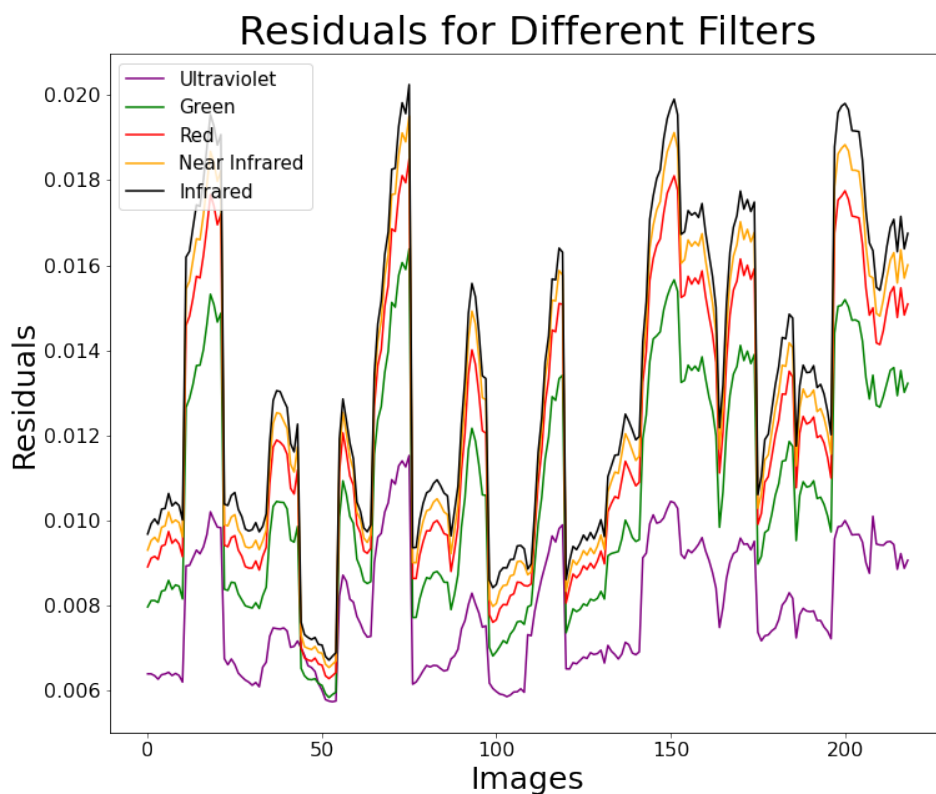


Figure 4.7: The relative residual for all images and all filter numbers.

Finally, visualization of some actual images, their corresponding predicted images, and their relative residuals are shown in Figure 4.8.

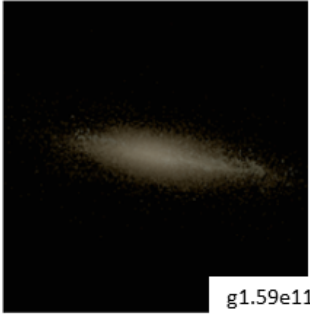
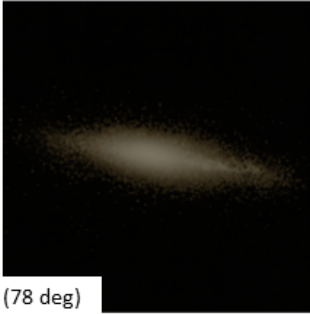
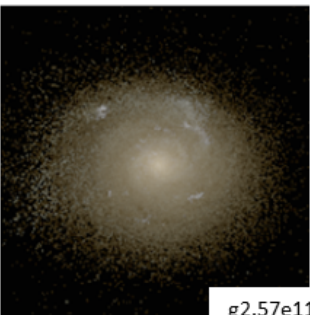
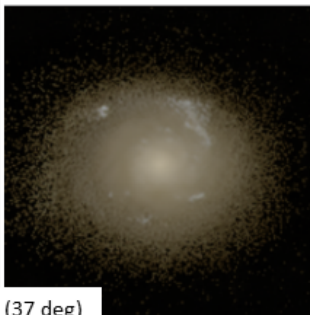
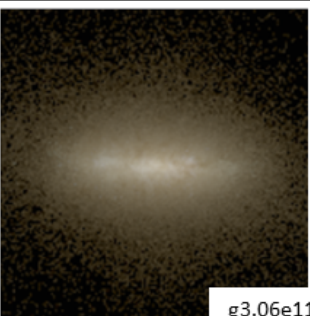
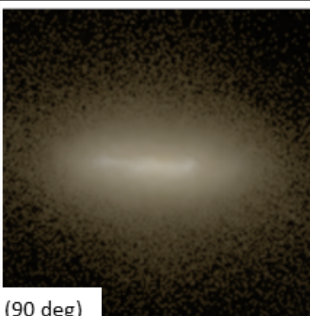
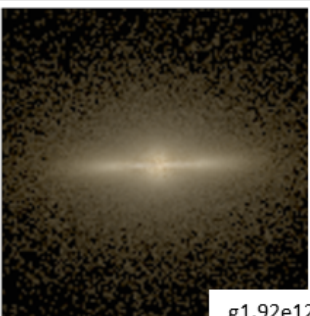
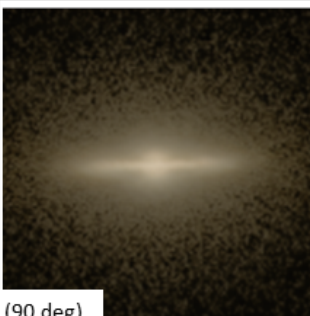
Actual Image	Predicted Image	Relative Residuals
		$\begin{bmatrix} \epsilon_u \\ \epsilon_g \\ \epsilon_r \\ \epsilon_i \\ \epsilon_z \end{bmatrix} = \begin{bmatrix} 0.00575 \\ 0.00584 \\ 0.00628 \\ 0.00654 \\ 0.00672 \end{bmatrix}$
		$\begin{bmatrix} \epsilon_u \\ \epsilon_g \\ \epsilon_r \\ \epsilon_i \\ \epsilon_z \end{bmatrix} = \begin{bmatrix} 0.00598 \\ 0.00691 \\ 0.00767 \\ 0.00805 \\ 0.00849 \end{bmatrix}$
		$\begin{bmatrix} \epsilon_u \\ \epsilon_g \\ \epsilon_r \\ \epsilon_i \\ \epsilon_z \end{bmatrix} = \begin{bmatrix} 0.00695 \\ 0.00833 \\ 0.00928 \\ 0.00966 \\ 0.01002 \end{bmatrix}$
		$\begin{bmatrix} \epsilon_u \\ \epsilon_g \\ \epsilon_r \\ \epsilon_i \\ \epsilon_z \end{bmatrix} = \begin{bmatrix} 0.01153 \\ 0.01638 \\ 0.01849 \\ 0.01951 \\ 0.02024 \end{bmatrix}$

Figure 4.8: The left column represents the actual images; the middle column contains the predicted images, and the right column shows the relative residual for these images. The best estimation to the worst estimation are sorted from top to down.

Chapter 5

Conclusion

This thesis began with an overview surrounding simulation models for galaxy formation. By sketching the differences between the N-body and hydrodynamical simulation models, I explained why it is desirable to exploit ML tools in this research field. I then described the basic concepts of deep learning and neural networks and continued the discussion by focusing on a specific convolutional neural network architecture, namely the U-Net model. Finally, I implemented a DL algorithm that takes a set of galaxy images as input and generates a new set of images with high accuracy and low computational complexity.

The input and output of the learning algorithm can be seen as the intrinsic and observational properties of galaxies, respectively. Therefore, the DL algorithm has learned to estimate the latter properties from the former without knowing the physical rules governing galactic systems. Apart from such a successful achievement, I found an interesting pattern in the model accuracy when evaluating the residuals in Figure 4.7. Apparently, there is a correlation between the residuals and the filters' wavelengths, suggesting that the estimation of the observational properties in higher frequencies (e.g., ultraviolet filter) is easier for the algorithm.

Some interesting directions for further research in this field are:

- **Optimization:** The current algorithm can be optimized more efficiently by exploiting advanced regularization methods available in the DL literature.
- **Physical Explanation:** As mentioned above, there is a correlation between the model residuals and the filter frequencies. Why is this so? Is there any physical explanation for such a phenomenon?
- **Evaluation and Feature Importance:** For model evaluation, I only considered the relative residuals as a sign of the accuracy. One can supplement my evaluation analysis with other statistical measures (such as variance, skewness) to extract more informative signs in the scenario. In a similar vein, one can evaluate the feature importance of the input data to find the most significant intrinsic properties of galaxies required to infer their observational properties.
- **Causal Analysis:** Even more technically, one can implement a causal analysis to infer the causal dependencies between different variables in the input and output data. Such a causal analysis might provide us with a deeper physical insight into the underlying physical mechanisms governing galactic systems.

Bibliography

- [1] O. Ronneberger, P. Fischer, and T. Brox, “U-net: Convolutional networks for biomedical image segmentation,” in *International Conference on Medical image computing and computer-assisted intervention*, pp. 234–241, Springer, 2015.
- [2] B. P. Moster, *Hydrodynamic Simulations of Cosmological Galaxy Merger Trees*. PhD thesis, 2010.
- [3] A. Loeb and F. A. Rasio, “Collapse of primordial gas clouds and the formation of quasar black holes,” *arXiv preprint astro-ph/9401026*, 1994.
- [4] S. D. White and C. S. Frenk, “Galaxy formation through hierarchical clustering,” *The Astrophysical Journal*, vol. 379, pp. 52–79, 1991.
- [5] V. P. Debattista, L. Mayer, C. M. Carollo, B. Moore, J. Wadsley, and T. Quinn, “The secular evolution of disk structural parameters,” *The Astrophysical Journal*, vol. 645, no. 1, p. 209, 2006.
- [6] “Galaxy formation.” <https://astronomy.swin.edu.au/cosmos/G/Galaxy+Formation>. Accessed: 2021-02-12.
- [7] J. F. Navarro, C. S. Frenk, and S. D. White, “A universal density profile from hierarchical clustering,” *The Astrophysical Journal*, vol. 490, no. 2, p. 493, 1997.

-
- [8] J. Diemand, M. Kuhlen, and P. Madau, “Formation and evolution of galaxy dark matter halos and their substructure,” *The Astrophysical Journal*, vol. 667, no. 2, p. 859, 2007.
- [9] B. P. Moster, R. S. Somerville, C. Maulbetsch, F. C. Van Den Bosch, A. V. Maccio, T. Naab, and L. Oser, “Constraints on the relationship between stellar mass and halo mass at low and high redshift,” *The Astrophysical Journal*, vol. 710, no. 2, p. 903, 2010.
- [10] A. E. Evrard, “Formation and evolution of x-ray-clusters-a hydrodynamic simulation of the intracluster medium,” 1990.
- [11] J. Stadel, D. Potter, B. Moore, J. Diemand, P. Madau, M. Zemp, M. Kuhlen, and V. Quilis, “Quantifying the heart of darkness with ghalo—a multibillion particle simulation of a galactic halo,” *Monthly Notices of the Royal Astronomical Society: Letters*, vol. 398, no. 1, pp. L21–L25, 2009.
- [12] J. F. Navarro and S. D. White, “Simulations of dissipative galaxy formation in hierarchically clustering universes—i: Tests of the code,” *Monthly Notices of the Royal Astronomical Society*, vol. 265, no. 2, pp. 271–300, 1993.
- [13] N. Katz and S. D. White, “Hierarchical galaxy formation-overmerging and the formation of an x-ray cluster,” *The Astrophysical Journal*, vol. 412, pp. 455–478, 1993.
- [14] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep learning*, vol. 1. MIT press Cambridge, 2016.
- [15] D. Shen, G. Wu, and H.-I. Suk, “Deep learning in medical image analysis,” *Annual review of biomedical engineering*, vol. 19, pp. 221–248, 2017.

- [16] J. Ngiam, A. Khosla, M. Kim, J. Nam, H. Lee, and A. Y. Ng, “Multimodal deep learning,” in *ICML*, 2011.
- [17] K. Kowsari, D. E. Brown, M. Heidarysafa, K. J. Meimandi, M. S. Gerber, and L. E. Barnes, “Hdltex: Hierarchical deep learning for text classification,” in *2017 16th IEEE international conference on machine learning and applications (ICMLA)*, pp. 364–371, IEEE, 2017.
- [18] L. Deng, J. Li, J.-T. Huang, K. Yao, D. Yu, F. Seide, M. Seltzer, G. Zweig, X. He, J. Williams, *et al.*, “Recent advances in deep learning for speech research at microsoft,” in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 8604–8608, IEEE, 2013.
- [19] I. Shavitt and E. Segal, “Regularization learning networks: deep learning for tabular datasets,” *arXiv preprint arXiv:1805.06440*, 2018.
- [20] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola, “Dive into deep learning. 2020,” *URL <https://d2l.ai>*, 2020.
- [21] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, “Tensorflow: A system for large-scale machine learning,” in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pp. 265–283, 2016.
- [22] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, “Automatic differentiation in pytorch,” 2017.
- [23] A. Gulli and S. Pal, *Deep learning with Keras*. Packt Publishing Ltd, 2017.
- [24] V. Vapnik, “Principles of risk minimization for learning theory,” in *Advances in neural information processing systems*, pp. 831–838, 1992.

- [25] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv preprint arXiv:1412.6980*, 2014.
- [26] “Sdss filters.” <http://skyserver.sdss.org/dr1/en/proj/advanced/color/sdssfilters.asp>. Accessed: 2021-02-12.
- [27] L. Prechelt, “Early stopping-but when?,” in *Neural Networks: Tricks of the trade*, pp. 55–69, Springer, 1998.

Declaration:

I hereby declare that this thesis is my own work, and that I have not used any sources and aids other than those stated in the thesis.

Munich 28.02.2021