



universität
wien

MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

Generative models applied to hierarchical data on a sphere

verfasst von / submitted by

Filip Sabo, BSc

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien, 2023 / Vienna, 2023

Studienkennzahl lt. Studienblatt /
degree programme code as it appears on
the student record sheet:

UA 066 910

Studienrichtung lt. Studienblatt /
degree programme as it appears on
the student record sheet:

Masterstudium Computational Science

Betreut von / Supervisor:

Dipl.-Phys. Dr. sc. ETH Oliver Hahn

Mitbetreut von / Co-Supervisor:

Florian List, B.Sc. M.Sc. PhD

Abstract

The last decade has been marked by the rise in popularity of AI and machine learning. Thanks to this, science has shifted its focus more and more towards data exploration. The vast majority of data lies in the Euclidean domain, however, in many research fields, spherical domains are present. This poses many challenges ranging from discretization to successful learning of a machine learning model.

The aim of this thesis is the construction of a generative adversarial network (GAN) on the sphere using HEALPix (Hierarchical Equal Area isoLatitude Pixelisation) as the basis for the discretization of the domain built on DeepSphere, an architecture which already solves aforementioned issues. As an application, we consider the example of predicting the structure of the intergalactic gas given a set of galaxies. It is worth noting that the structure of the model can be applied to any other generative modelling tasks present in other research fields.

Das letzte Jahrzehnt war vor allem durch die zunehmende Popularität von künstlicher Intelligenz und maschinellem Lernen gekennzeichnet. Durch diese Entwicklungen hat sich der Schwerpunkt der Wissenschaft immer mehr auf die Datenerforschung verlagert. Die überwiegende Mehrheit der Daten liegt im Euklidischen Raum, doch in vielen Forschungsbereichen sind sphärische Räume präsent. Dies bringt viele Herausforderungen mit sich, von der Diskretisierung vom Raum bis zum erfolgreichen Lernen eines maschinellen Lernmodells.

Das Ziel dieser Arbeit ist die Konstruktion eines generativen adversen Netzwerks (GAN) auf der Sphäre unter Verwendung von HEALPix (Hierarchical Equal Area isoLatitude Pixelisation) als Grundlage für die Diskretisierung der Domäne auf DeepSphere, einer Architektur, die bereits die oben genannten Probleme löst. Als Anwendung betrachten wir das Beispiel, wo die Struktur des intergalaktischen Gases anhand bekannten Galaxienpositionen vorhergesagt wird. Es ist erwähnenswert, dass die Struktur des Modells auf andere ähnliche Probleme in anderen Forschungsbereichen angewendet werden kann.

Foreword

Ever since I can remember, I have been fascinated by science. During high school, I discovered a second passion of mine – programming. That is the reason why I decided on the following study path: first Physics in the Bachelor's and then Computational Science in my Master's. Soon after my first exposure to machine learning, I quickly realized that this is exactly the field of research I want to work in as it was unbelievably enthralling and fulfilling.

This thesis initially started as an assignment in collaboration with my girlfriend Natalie Grasser at the university during the Data Science course given by professors Oliver Hahn and Florian List. After the submission, we were delighted and yet surprised by the unexpectedly positive praise from our professors, who got in touch with us through a personal email. Since I greatly enjoyed working on this topic, I decided to carry on with it and expand it into my Master's thesis.

In this context, I would like to express my sincere gratitude towards my supervisors, Dipl.-Phys. Dr. sc. ETH Oliver Hahn and Florian List, B.Sc. M.Sc. PhD, for their constant support and a very good cooperation.

Vienna, August 9, 2023
Filip Sabo

Contents

1	Introduction	3
2	Problem setting	5
3	Methodology	9
3.1	Supervised machine learning	9
3.1.1	Basics	9
3.1.2	Neural networks	12
3.1.3	Convolutional neural networks	15
3.2	Discretization of spherical domains	17
3.2.1	Spherical coordinates	17
3.2.2	HEALPix	17
3.2.3	CNNs on a sphere	20
3.3	Generative models	21
3.3.1	Generative adversarial networks (GANs)	22
3.3.2	pix2pix	23
4	Approach	28
4.1	Data set	28
4.2	Creation of training and test set	28
4.3	Normalization of the data sets	30
4.4	Finding a model	30
5	Results	33
5.1	Model performance evaluation	33
5.2	Statistical analysis	35
5.3	Uncertainty quantification	37
6	Conclusions and outlook	40
7	Appendix	46
7.1	Creation of training and test data sets	46
7.2	Normalization of the data sets	49
7.3	Primary architectures	50
7.4	Finding a model - best model	52
7.5	Plotting of results	56
7.6	Statistical analysis	57
7.7	Creation of noisy data	58
7.8	Statistical analysis of noisy data	61

Chapter 1

Introduction

Generative models aim to learn the joint distribution $p(\vec{x}, y)$ or $p(\vec{x})$ if no labels are present) with contrast to discriminative models which describe the conditional distribution, i.e. $p(y|\vec{x})$. To simplify it, discriminative models aim to classify samples whereas generative models try to similarly looking samples. Typically, generative models are excellent at data reconstruction problems (Karras et al. [2018]), such as image-to-image translation tasks (Isola et al. [2016]). They use architectures inspired by convolutional neural networks (CNNs), which have proven to be one of the most successful machine learning models with an impressive track record of breakthroughs in various research domains in recent years, predominantly in analysis of images [Valueva et al., 2020].

Generally, to make computations on domains, we discretize them by introducing a grid. Since the Euclidean domains are the most common, the most straight-forward way is to use square cells with an equal spacing here. However, many research fields involve problems where the data lies on a sphere. Certainly, one of the most prominent is astronomy, since the majority of the real astronomy data lives in a spherical domain.

This is rather unfortunate, since it is challenging to construct a grid for a sphere. One of the most used approaches for pixelisations of spherical manifolds is the so-called HEALPix (Hierarchical Equal Area isoLatitude Pixelisation) [Gorski et al., 2005]. In a nutshell, the sphere is partitioned into a number of curvilinear quadrilaterals and then a subdivision into infinitely small subelements is performed in a hierarchical manner.

CNNs have already been implemented on spherical domains (Perraudin et al. [2019], Lee et al. [2019], Krachmalnicoff and Tomasi [2019], Bidgoli et al. [2022] and Cohen et al. [2019]). However, regarding generative models, there have been only very few applications in spherical domains up until now such as Rozen et al. [2021] which introduced Moser Flow, a generative model for different manifolds such as spheres in the family of the so-called continuous normalizing flows.

The aim of this thesis is to implement pix2pix, a conditional generative adversarial model (GAN), on a sphere [Isola et al., 2016]. This is an extension of GANs, which aim to learn to draw samples from the underlying distribution, i.e. $p(X)$. In contrast, pix2pix tries to learn a conditional distribution of output samples given the input sample, i.e. $p(Y|X)$. Since pix2pix is particularly great at image-to-image translation tasks, it will be applied to predict the structure of the intergalactic gas from galaxies on the full sky using data from the cosmological hydrodynamical simulation CAMELS [Villaescusa-Navarro et al., 2021].

The methodology can also be applied to similar problems in e.g. meteorology, geography or any field which works with computations on spheres. The emphasis will lie on the algorithmic side of the given problem setting.

The first chapter of this thesis explains in detail the given problem setting. Next, we will focus on the underlying basics of supervised machine learning, in particular neural networks and convolutional neural networks, and the theory behind the discretization of spherical domains. We will conclude the chapter focusing on the pix2pix model, which we have chosen for this

task. In the third chapter, we will be laying the groundwork for the creation of the data set we will be using and search for a sufficiently good model. Last but certainly not least, we will have a look at the performance not only qualitatively but also quantitatively.

Chapter 2

Problem setting

In this chapter, we will introduce the example application for the pix2pix, as well as the motivation for this application and the theoretical principles behind it. Specifically, the conditional GAN will be applied to an astronomical dataset of galaxy positions (as obtained from the Rockstar sub-halo catalogs) to infer the structure of the cosmic web through the distribution of intergalactic gas. Understanding the large-scale structure is of huge importance for astrophysicists, since it is not only tightly coupled to the evolution of galaxies and clusters, but also the evolution of the universe as a whole (Zhang et al. [2021], Zhang et al. [2022] and Hasan et al. [2023]).

At the largest scales, the universe can be described by the so-called cosmic web, which is defined as a filamentary network of galaxies, gas, as well as dark matter. In contrast to baryonic matter, i.e. visible matter composed of protons and neutrons, dark matter is a hypothetical form of matter that cannot be observed through electromagnetic radiation, since it does not absorb or emit any photons. Its existence has been up until now inferred solely through its gravitational effects, such as rotation curves of spiral galaxies, gravitational lensing and the motions of galaxies in galaxy clusters, which cannot be explained by any other form of matter [Trimble, 1987].

In order to observe the cosmic web, we must consider very large scales (mega-parsecs $\sim$ millions of light years) at which galaxies appear as tiny particles [Stapelberg, 2022]. Figure 2.1 illustrates the distribution of galaxy clusters beyond the Milky Way as observed by the near-infrared survey 2MASS (Two Micron All-Sky Survey), which scanned the whole sky [Jarrett, 2004]. The galaxies are color-coded by their redshifts, which represent their distances from the Sun.

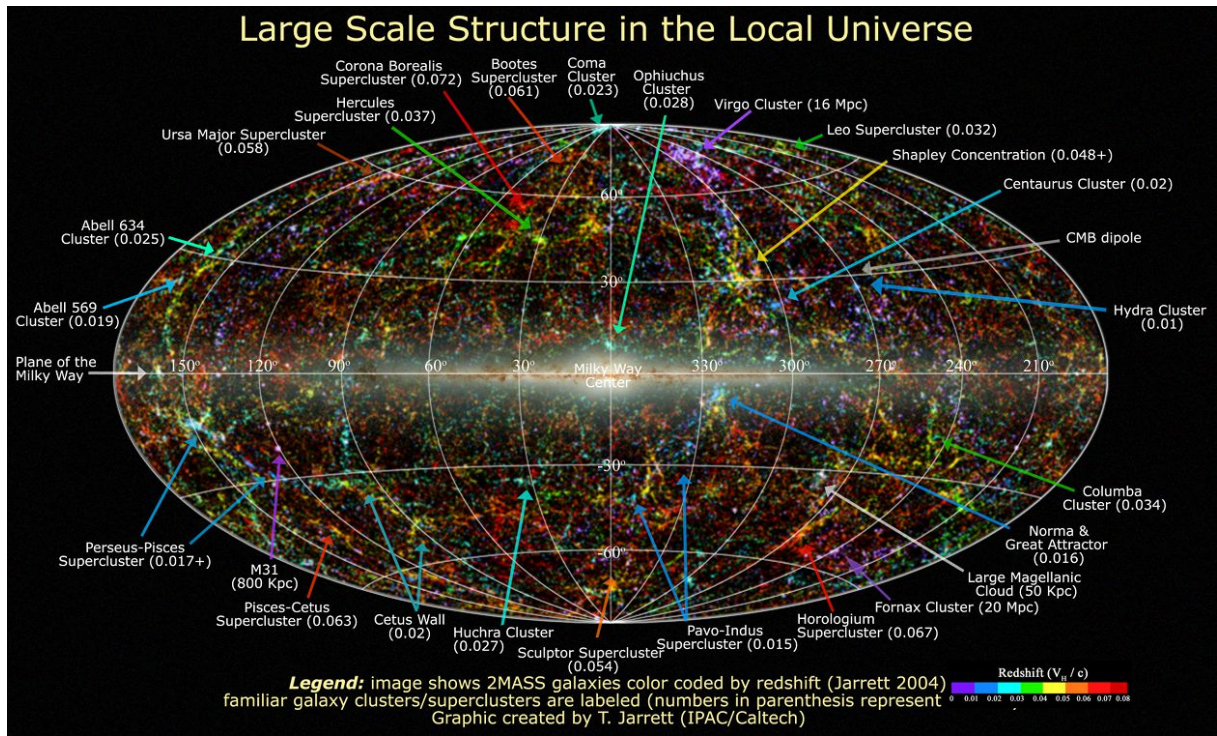


Figure 2.1: Distribution of galaxies and galaxy clusters beyond the Milky Way, as observed by the near-infrared 2MASS survey. The depicted colors of the galaxies represent their redshifts [Jarrett, 2004].

Clearly, the galaxy clusters are not distributed randomly or uniformly, but rather along huge filaments created and bound by gravity. These filaments are separated by cosmic voids, i.e. vast regions containing almost no galaxies. In fact, the huge majority of the cosmic volume turns out to be dominated by cosmic voids. Huge galaxy clusters form high density “nodes” of the cosmic web. Furthermore, along the filaments galaxy clusters are still growing by gravitationally attracting more matter [Stapelberg, 2022].

Figure 2.2 shows the structural constituents of the cosmic web, beginning at smaller scales like the solar system, continuing with the clustering of nearby stellar systems into galaxies, which are further clustered along the filaments of dark matter to form the large-scale structure called the cosmic web. The filamentary distribution of matter resembles a web, as seen on larger scales.

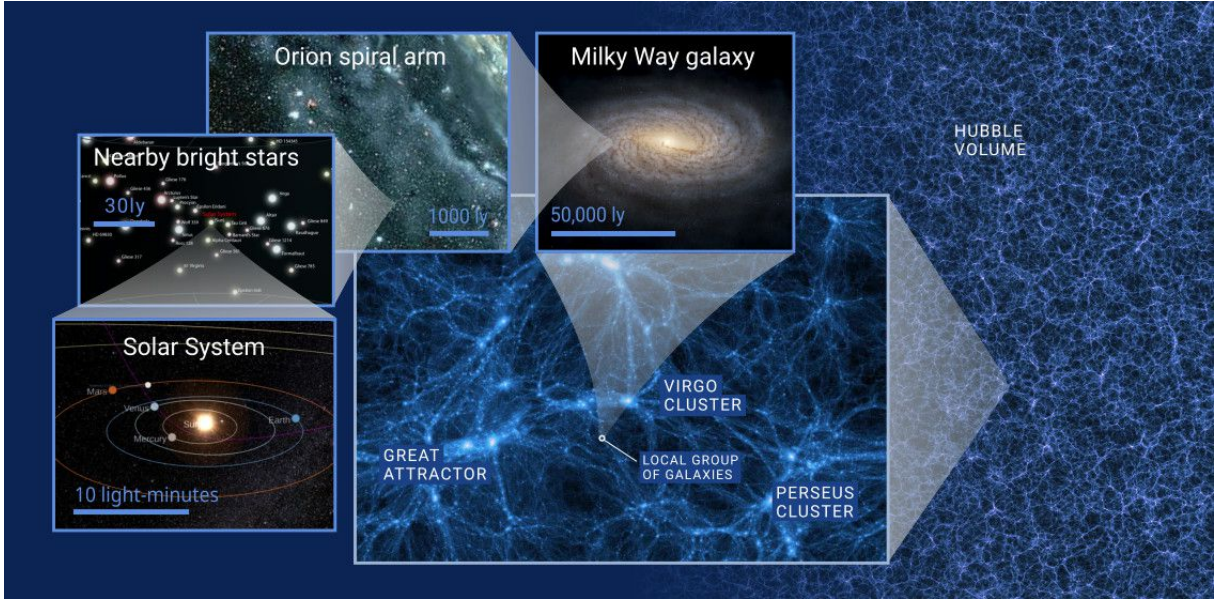


Figure 2.2: Constituents and structure of the cosmic web, from smaller to larger scales [Stapelberg, 2022]

The formation of such large-scale structures in the universe dates back to the Big Bang. The building of the cosmic web requires a long-range force that acts over large scales, namely gravity. Right after the Big Bang, the early universe was very hot and largely homogeneous and isotropic. However, small overdensities due to quantum fluctuations arose, some of which grew large enough to become gravitationally unstable and collapsed as the universe expanded and cooled. These overdensities further attracted more mass, formed stars and filaments of matter, creating large-scale structures (Padmanabhan [1993], Peacock [1999] and Dodelson [2003]).

In the context of structure formation, dark matter plays a crucial role due to its gravitational interaction with visible matter. The growth of initial overdensities in the early universe as well as structure formation would not have been possible without the additional gravitational influence of dark matter, which makes up the overwhelming majority of matter in the universe and ultimately made galaxy formation possible [Longair, 2008]. This is because during the early universe, the baryonic fluctuations were damped out, while the dark matter fluctuations were able to grow large enough. Only through the coupling of baryonic and dark matter was it possible for the baryonic fluctuations to grow sufficiently large as well and allow for the formation of larger structures [Longair, 2008]. Several methods have found that dark matter gravitationally clumps together and forms the so-called halos, which attract and embed galaxies. This leads to the formation of gravitationally bound structures and filaments, which make up the cosmic web [Stapelberg, 2022].

Normally, the simulation of the intergalactic gas distribution is rather computationally tedious on cosmological scales. Thus the aim of this thesis is to apply a pix2pix model on a sphere to the positions of galaxies to cheaply infer the gas distribution and hence the structure of the cosmic web. Through this, one can determine how well galaxies can trace the properties of gas and dark matter in the large-scale structures of the universe. These findings can be used to assess the accuracy of cosmic web structures inferred from galaxy positions.

More formally, let us assume that there exists an underlying probability distribution for the discrete galaxy positions as well as for the intergalactic gas distribution being p_{galaxies} and p_{gas} respectively. The goal is to find an approximation of the true underlying conditional distribution $p_{\text{gas}}(Y|X)$, i.e. the distribution of possible gas density fields Y , given a set of observed galaxies X . This shall be done through the application of a conditional GAN on a sphere to the data from the IllustrisTNG project [Nelson et al., 2019]. Mathematically, this

can be formulated accordingly:

Given: $X \sim p_{\text{galaxies}}$

$Y \sim p_{\text{gas}}$

Find: an approximation of $p_{\text{gas}}(Y|X)$

Chapter 3

Methodology

Having some data set, machine learning can be seen as an automatic rules discovery on this set. We could define machine learning as the “non-trivial process of identifying valid and potentially useful patterns in data” [Fayyad et al., 1996]. Specifically, these terms encompass the following meanings:

- Non-trivial: for instance not just the average function,
- Valid: the discovered patterns should also hold for new, previously unseen data samples,
- Potentially useful: they should lead to some benefit to the user or the task.

Moreover, these rules should be as simple and simultaneously as general as possible such that they can be applied to any future or previously unseen data. This balance between complexity and generality is one of the most pivotal cornerstones of machine learning called bias-variance tradeoff. We will come to this later on.

The two most common machine learning types are supervised and unsupervised machine learning. Suppose we have an $N \times M$ data set matrix (or input matrix) X with N being the number of data samples and M the number of features (data properties) and an $N \times M$ label or target matrix Y .

The aim of the supervised machine learning methods is to find a function f which maps X to Y , i.e. $f(X) = Y$. In contrast to this, unsupervised machine learning, also called learning without labels, focuses on tasks such as identifying a compact representation or latent variables of X . The unsupervised machine learning methods are particularly used for exploratory data analysis, feature learning, feature embedding or anomaly detection of “unusual” data points (outliers). Let us now introduce the basics and theory behind the (convolutional) neural networks, the ideas behind their implementation on the sphere and the generative models.

3.1 Supervised machine learning

In this thesis we will generate new images based on a given distribution of galaxies. We opted for neural networks due to their flexibility in learning. Hence, the following chapter will revolve around the basics of the supervised machine learning, but particularly around the basics of neural networks, i.e. what are they actually, how does their structure look like, how exactly do they get trained and wherein lie the challenges. Furthermore, we will talk about a subclass of neural networks, the so-called convolutional neural networks, which have proven to be quite pivotal in tasks which involve image recognition [O’Shea and Nash, 2015].

3.1.1 Basics

Let us now shift our focus to supervised machine learning. Classification and regression are some of the most common supervised machine learning tasks. As stated earlier, we are striving

to learn the function f and its optimal parameters W^* such that we can map the input matrix X to the label matrix Y , i.e. $Y = f(X; W^*)$.

The first is to define a loss function L according to the given problem, which describes the magnitude of the error of the function f given the input matrix with respect to the label matrix Y . Let us formulate the search for this function and its parameters as an optimization problem, i.e. let us find the minimum of the loss function L or in a more formal way:

$$W^* = \underset{W}{\operatorname{argmin}} L(Y, f(X; W)) \quad (3.1)$$

W are the parameters of the function f we are optimizing over. Here are some of the most common loss functions:

- L2-loss: particularly useful for regression

$$L(Y, f(X; W)) = \frac{1}{N} \sum_{i=1}^N (f(X_i; W) - Y_i)^2$$

- L1-loss: particularly useful for regression

$$L(Y, f(X; W)) = \frac{1}{N} \sum_{i=1}^N |f(X_i; W) - Y_i|$$

In order to find the approximate solution of equation 3.1 and if we have a convex and differentiable loss function, we can take advantage of the fact that generally the gradient of a function points into the direction of the fastest increase. Hence, let us go into the opposite direction and define the following iterative algorithm:

- Start at an arbitrary $W_0 \in \mathbb{R}^M$
- For $t=0,1,2,\dots$, do:

$$W_{t+1} = W_t - \eta_t \nabla L(Y, f(X; W_t))$$

This method is also called gradient descent. The parameter η_t stands for the learning rate. The rate of convergence of this iterative method generally depends on the choice of the learning rate. Under mild assumptions, if the learning rate is sufficiently small, gradient descent converges to a stationary point (gradient = 0). For convex objectives, it therefore finds the optimal solution! Although, for non-convex problems, it converges only to a local extremum. In practice, we stop the training if it no longer improves performance on validation set (part of dataset we hold out from training).

We need to bear in mind that the computation of the exact gradient $\nabla L(Y, f(X; W))$ is rather expensive, since it requires the whole dataset X . This means summing gradient contributions over all the points in the dataset. Since modern datasets often contain billions of data points, it would make every single gradient descent step extremely expensive.

Fortunately, we can get a good (unbiased) gradient estimate by evaluating the gradient on only a few points. Hence, let us at every step pick a different minibatch B containing a subset of the dataset and use it as input. Let us summarize this in a pseudocode:

Start at an arbitrary $W_0 \in \mathbb{R}^M$

For $t=0,1,2,\dots$, do:

Pick a number of data samples (X', Y') from (X, Y) uniformly at random and set

$$W_{t+1} = W_t - \eta_t \nabla L(Y', f(X'; W_t)) \quad (3.2)$$

This algorithm is called stochastic gradient descent (SGD). So far, the only measure of the goodness of fit was the loss function. Theoretically, we could choose an infinitely complex model function which would lead to 0 loss. Figure 3.1 shows the resulting mean squared error from models with different complexities which are trying to achieve this 0 loss and were then followingly applied to new, unseen data - and hence shows why this mindset is incorrect.

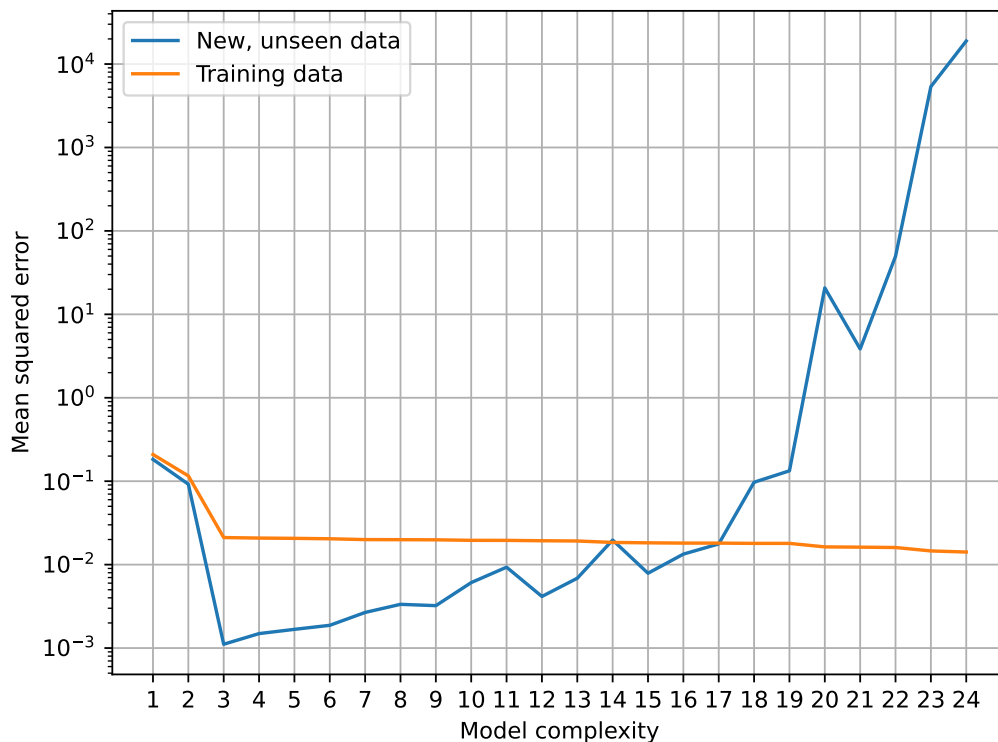


Figure 3.1: Mean squared error (L2-loss) as a function of model complexity

The orange curve shows what happens if we choose a model with too large complexity - the model will try to capture every single detail of the data whilst failing to generalize on previously unseen data. We need to keep in mind that the data often contains noise which the model is desperately trying to adapt to. This is called overfitting.

However, the aim of machine learning lies in finding the balance between simplicity and generality of the model. This is where the so-called bias-variance tradeoff comes into play, with bias being the difference between the true and predicted values, whereas the variance tells us how well the model generalizes when applied to new, in training unseen data.

Therefore, high bias occurs when the model underfits (the model is way too simple), while high variance occurs when we overfit (the model is way too complex). Hence the aim is to keep both of them as low as possible, i.e. having a sufficiently complex model as depicted in figure 3.2, which shows the balance between under- and overfitting.

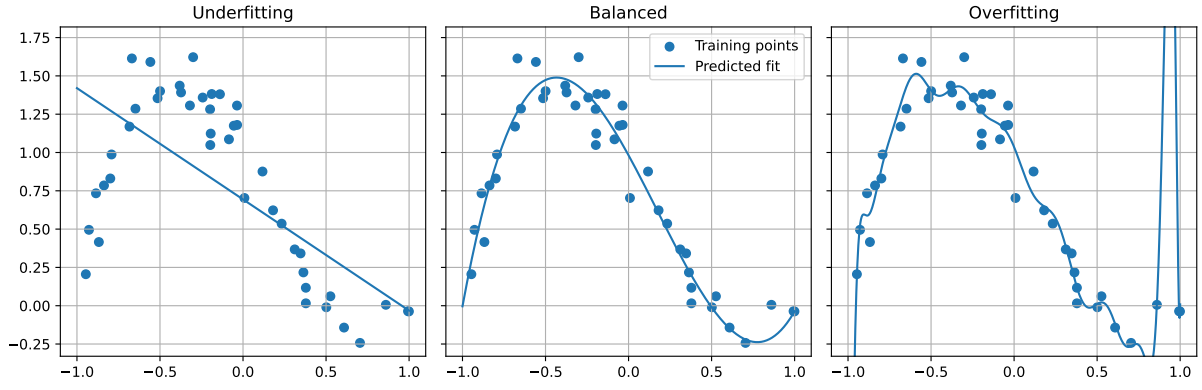


Figure 3.2: Balance between under- and overfitting

This is what a correct supervised learning pipeline looks like: given a set of data (X, Y) , we split into a training set (X_{train}, Y_{train}) and a test set (X_{test}, Y_{test}) . Typical splits are around 70-80% of the data samples going to the training set and the rest to the test set. Next, we fit the model to the training data and evaluate its accuracy on the test set, not on the training set, to see how well it performs on data that it hasn't seen yet.

Moreover, there are multiple ways to combat overfitting. Since it is closely linked with huge weights, we can introduce an additional term in the loss function which punishes large weights. This method is called regularization. We will not go into further details on this topic.

3.1.2 Neural networks

Neural networks have grown in prominence in the recent years. The reason for this is their capability and flexibility to learn complex relationships between features. One can simply see them as a sum of non-linear functions which are nested compositions of (variable) linear functions composed with (fixed) non-linearities, for instance:

$$f(X; W) = \sum_{i=1}^m \varphi(W_i^T X) \quad (3.3)$$

φ are the so-called activation functions, which define the output of each neuron. These activation functions permit the neural network to capture complex relations which would be nearly impossible with just linear functions [Chng, 2022]. A neural network can have more than one hidden layer, i.e. a layer between the input and output layer, and more than one output. Figure 3.3 depicts an example of such a neural network:

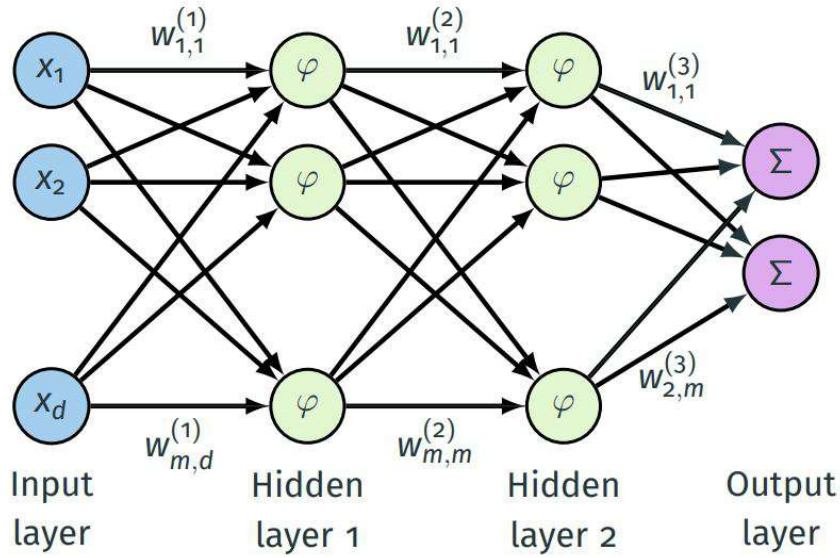


Figure 3.3: An example of a more complex neural network [Tschitschek, 2022]

The following are some of the most well-known and used activation functions:

- Linear:

$$\varphi(z) = z$$

- Sigmoid:

$$\varphi(z) = \frac{1}{1 + \exp(-z)}$$

- Rectified linear units (ReLU):

$$\varphi(z) = \max(0, z)$$

- Leaky ReLU:

$$\varphi(z) = \max(0.01z, z)$$

Suppose we have learned all the weights. How do we then make predictions? The answer to this is an algorithm called forward propagation:

- For the input layer $V^0 = X$
- For each hidden layer $l=1, \dots, L-1$ do:
 - $Z^l = W^l V^{l-1}$
 - $V^l = \varphi(Z^l)$
- For the output layer: $F = W^L V^{L-1}$
- The last step depends on the shape of the output, here are some of the options:
 - $Y = \varphi(F)$ (if the output is a vector, we can run F from the output layer through a final activation function)
 - $y = \sum_{i=1}^n F_i$ (if the output is a scalar, we may sum up F from the output layer)

Now the optimization problem 3.1 has this form:

$$W^* = \underset{W}{\operatorname{argmin}} L(Y, X; W) \quad (3.4)$$

i.e. we jointly optimize over all weights for all layers to minimize loss over the training data. However, this is in general a non-convex optimization problem, i.e. a problem with multiple minima (multiple local and one global). Nevertheless, we can try to find the global minimum with SGD (3.2). This requires computing the gradient

$$\frac{\partial}{\partial W} L(Y, X; W) \quad (3.5)$$

i.e. for each weight between any two connected units i and j we need:

$$\frac{\partial}{\partial W_{i,j}} L(Y, X; W) \quad (3.6)$$

We take use of the chain rule to propagate gradients of intermediate steps, and finally obtain gradients of L with regards to W :

- For the output layer:
 - Compute error $\delta^{(L)} = L'(Y; f)$
 - Gradient $\nabla_{W^{(L)}} L(Y, X; W) = \delta^{(L)} v^{(L-1)T}$
- For each hidden layer $l = L - 1, \dots, 1$:
 - Compute error $\delta^{(l)} = \varphi'(\bar{z}^{(l)}) \odot (W^{(l+1)T} \delta^{(l+1)})$
 - Gradient $\nabla_{W^{(l)}} L(Y, X; W) = \delta^{(l)} v^{(l-1)T}$

Since we have a non-convex problem on our hands, the initialization of weights matters. If we pick inappropriate weights, the training may stall. Although, as soon as we run into a local minimum, the gradient vanishes and we get stuck, so in that case we need to come up with something more advanced. Thus, in practice, we always reach a local minimum with deep neural networks, not a global one. The idea is to keep the variance of weights approximately constant across all layers to avoid vanishing or exploding gradients. Usually, random initialization works well. The following are among the most common weights initialization methods:

- Glorot
 - $W_{i,j} \sim \mathcal{N}(0, \frac{2}{n_{in} + n_{out}})$
- He
 - $W_{i,j} \sim \mathcal{N}(0, \frac{2}{n_{in}})$

Furthermore, the choice of η_t is pivotal in order to achieve the global minimum. Hence, the idea would be to extend the SGD moving not only into direction of the gradient, but also in the direction of the last weight update. This can help to escape local minima. Adam (Adaptive Moment Estimation) [Kingma and Ba, 2017] is probably the most common variant of SGD used for training deep networks currently, as it keeps an individual learning rate and estimates of the first and second moment of the gradient for each weight. The detailed steps of the algorithm can be found in Kingma and Ba [2017].

Since neural networks typically have a lot of parameters, there is a risk of severely overfitting the model. Here are some of the most used countermeasures:

- Regularization:
 - Add penalty term to avoid large weights.

- Dropout [Srivastava et al., 2014]:
 - Randomly ignore neurons in the neural network during each iteration of SGD with a given probability.
- Batch normalization [Ioffe and Szegedy, 2015]:
 - This is a widely used technique which normalizes inputs to each layer according to mini-batch statistics.
- Layer normalization [Ba et al., 2016]:
 - Here we normalize inputs in all neurons in the same layer for each data sample [AssemblyAI, 2022].

Furthermore, predictions should remain unchanged under some transformations of the data, such as translation or rotation. A simple way to ensure that the neural network learns such transformations is by applying data augmentation to the training set, i.e. create new data samples by for instance rotating or adding noise to the existing data training samples.

Finally, here are some of the key parameters of a neural network:

- Batch size:
 - The number of data points in a minibatch
- Iteration:
 - One step of SGD on a minibatch
- Epoch:
 - One full pass over the dataset
 - Number of iterations in each epoch is equal to the ratio of dataset size and batch size

3.1.3 Convolutional neural networks

Convolutional neural networks are neural networks for specialized applications such as image recognition [O'Shea and Nash, 2015]. The difference between CNNs and ANNs (artificial neural networks or just neural networks, see 3.3) is that CNNs have so-called convolutional (hidden) layers. The key idea behind the convolutional layers is the introduction of kernels. These are matrices with specified shapes by the user. By thinking of images as matrices, we can slide a kernel across all the pixels of the image matrix, hence creating a “convolved” version of the image matrix with the help of the dot product between the kernel and the slid on part of the image matrix. This turns out to be quite helpful in recognizing shapes, edges and patterns. Figure 3.4 shows how such kernels work, in this case a 3×3 kernel.

3. Isolatitude distribution for the discrete area elements on the sphere:

- This implies a constant number of pixels on the lines of constant latitude.

There are various possible pixelization techniques, but only HEALPix satisfies all of these requirements simultaneously. Here, the sphere is partitioned into a number of curvilinear quadrilaterals. Furthermore, if there is a mapping of each element of the partition onto a unit square $([0, 1] \times [0, 1])$, then a subdivision into infinitely small subelements exists and hierarchical tree structure is obtained in straight forward manner. Regarding the spherical domains, the base-level discretization is defined by the following parameters:

- The number of base-resolution pixel layers between the north and south poles N_θ
- The number of equatorial base-resolution pixels N_φ

Hence, the number of base level pixels is $N_{pixels} = N_\theta N_\varphi$ and the area of each is one is $A_{pixel} = \frac{4\pi}{N_\theta N_\varphi}$. Moreover, let us define the colatitude θ^* at which the lateral vertices of both polar and equatorial pixels meet:

$$\cos \theta^* = \frac{N_\theta - 1}{N_\theta} \quad (3.7)$$

Figure 3.6 shows a few examples of base-level tessellations:

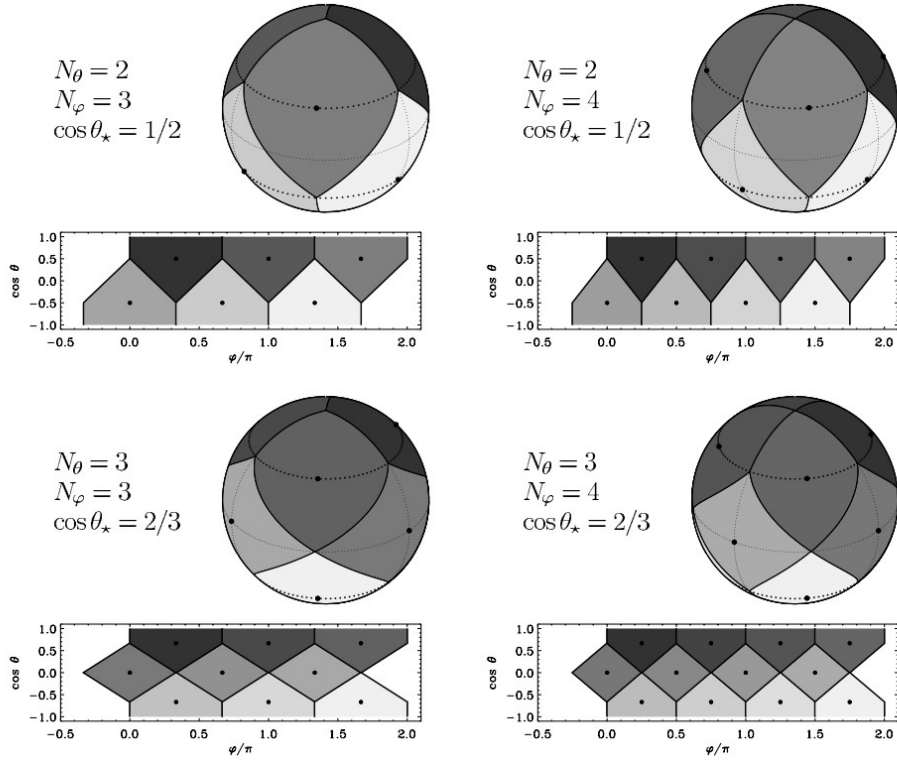


Figure 3.6: Base-level tessellation [Gorski et al., 2005]

Despite the areas remaining constant across pixels, their shapes vary depending on their position. N_{side} stands for the resolution of the sphere, i.e. the number of divisions along the side of a base-resolution pixel that is needed to reach a desired high-resolution partition. Regarding the pixel centers, they can be found on isolatitude lines and are equidistant in azimuth (on each ring). All in all, a HEALPix map has $N_{map} = 12N_{side}^2$ pixels in total. Figure 3.7 shows the discretization of a sphere for $N_{side} = 1, 2, 4, 8$.

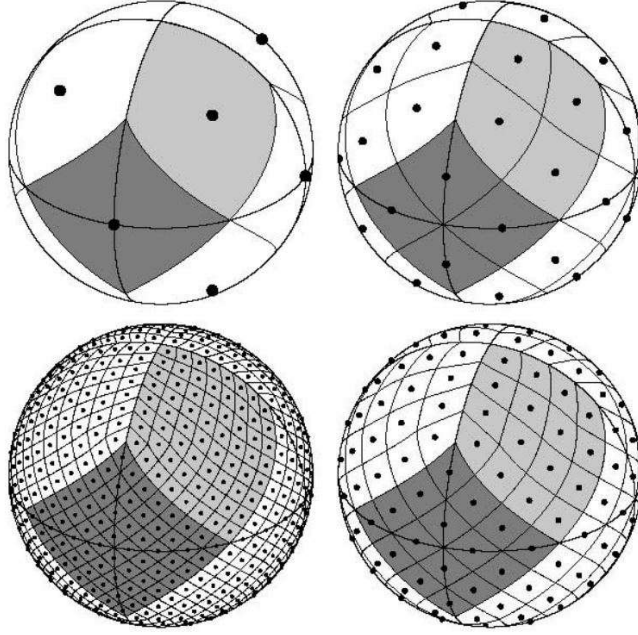


Figure 3.7: Discretizations for $N_{side} = 1, 2, 4, 8$ [Gorski et al., 2005]

As we can see, each pixel is subdivided into $2^{N_{side}}$ children pixels in a hierarchical manner. This fact plays pivotal role in pooling in the convolutional layers. The pixel centers are defined by $p_{center} = (z \equiv \cos \theta, \phi)$ with $\theta \in [0, \pi]$ being the colatitude measured from the north pole and $\phi \in [0, 2\pi]$ standing for the longitude measured eastwards.

There are two possible pixel indexing approaches, called ring and nested. In the ring scheme, one just counts the pixels beginning from the north pole going down to the south while moving along isolatitudinal rings. Particularly Fourier transforms with spherical harmonics can be implemented in a straight-forward manner with the help of this indexing. The nested indexing reproduces the hierarchical structure of the pixels. This pixel indexing proves to be particularly useful in all applications involving nearest-neighbor searches such as convolutions and pooling. Both can be seen in figure 3.8. Further details regarding these two pixel indexing methods can be found in Gorski et al. [2005].

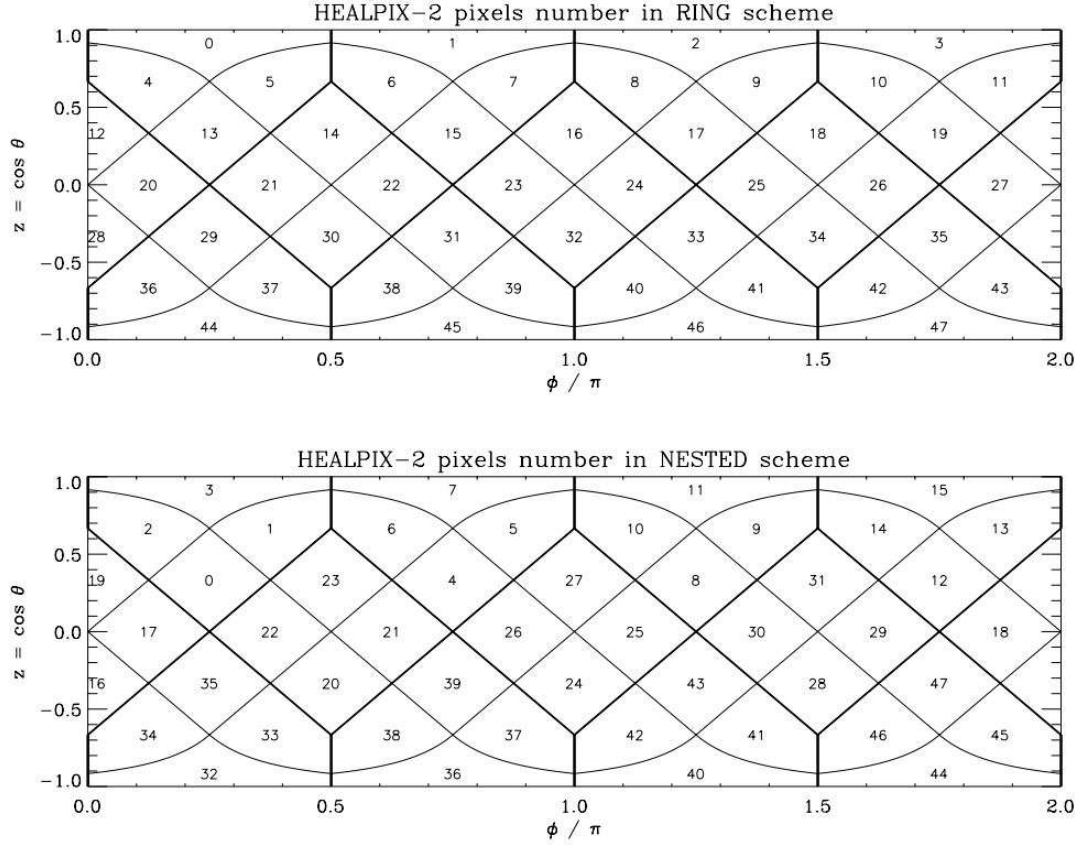


Figure 3.8: Ring and nested pixel indexing [Gorski et al., 2005]

The HEALPix method is implemented in the Python library healpy [Zonca et al., 2019].

3.2.3 CNNs on a sphere

This section is heavily based on [Perraudin et al., 2019]. Now, let us have a look at the theory of CNNs on spheres. Generally, a CNN has these main components [Perraudin et al., 2019]:

1. Convolution
2. Non-linearity
3. Down-sampling operation
4. Pooling operation
5. Normalization

Out of these five, only the convolution and down-sampling need to be rethought when applied to spherical domains. The paper Perraudin et al. [2019] presents the idea of defining the convolution operation on a sphere using a graph. Moreover, the downsampling operations shall be done using the hierarchical pixelisation of the sphere.

Let us now build a graph $G = (V, E, W)$ with V being the set of vertices ($|V| = N_{pix}$, i.e. each pixel is represented by a vertex), E the set of edges and W being the weighted adjacency matrix:

$$W_{ij} = \begin{cases} \exp\left(-\frac{\|\vec{x}_i - \vec{x}_j\|_2^2}{\rho^2}\right) & \text{if vertices } v_i \text{ and } v_j \text{ are neighbours} \\ 0 & \text{else} \end{cases} \quad (3.8)$$

with $\vec{x}_i$ being a vector encoding the 3-dimensional coordinates of v_i and

$$\rho = \frac{1}{|E|} \sum_{(v_i, v_j) \in E} \|\vec{x}_i - \vec{x}_j\|_2 \quad (3.9)$$

i.e. the average Euclidean distance over all connected pixels. One must not forget that the distances between pixels are not equal. Now, let us define the graph Laplacian L which is needed for the Fourier basis of the graph:

$$L = \mathbb{I} - D^{-1/2} W D^{-1/2} \quad (3.10)$$

with $\mathbb{I}$ being the identity matrix and D a diagonal matrix with

$$D_{ii} = d_i \quad \text{and} \\ d_i = \sum_j W_{ij}$$

Due to the nature of graphs, the convolutions on them work in such a way that one needs to filter a graph signal by a kernel. More formally, given a convolution kernel $h : \mathbb{R}_+ \rightarrow \mathbb{R}$ a signal on a graph $\vec{f} \in \mathbb{R}^{N_{pix}}$ can be filtered in the following way:

$$h(L)\vec{f} = U h(\Lambda) U^T \vec{f}, \quad (3.11)$$

with

$$L = U \Lambda U^T \quad (3.12)$$

with U being an orthonormal matrix of eigenvectors and λ a diagonal eigenvalue matrix. However, the eigenvalue decomposition is rather computationally inefficient. A better solution is to use the so-called Chebyshev polynomials, which approximate the eigenvalue decomposition, as filters, i.e.:

$$h_\theta(\tilde{L})\vec{f} = \sum_{k=0}^K \theta_k T_k(\tilde{L})\vec{f} \quad (3.13)$$

with

$$\tilde{L} = \frac{2}{\lambda_{max}} L - \mathbb{I} \quad (3.14)$$

being a rescaled Laplacian with λ_{max} being the largest eigenvalue and $T_k(\cdot)$ being the Chebyshev polynomial of degree k with

$$T_k(\tilde{L}) = 2\tilde{L}T_{k-1}(\tilde{L}) - T_{k-2}(\tilde{L}) \\ T_1(\tilde{L}) = \tilde{L} \\ T_0(\tilde{L}) = \mathbb{I}$$

More details can be found in Perraudin et al. [2019].

3.3 Generative models

In supervised machine learning, there are these two types of models:

- Discriminative models:
 - These models are described by the conditional distribution $p(y|\vec{x})$.
 - In other words, given the vector $\vec{x}$, determine y .

- Logistic regression or SVMs (a problem of finding a decision boundary between two or more classes while maximizing the distance to samples closest to the boundary) are just some of the examples for discriminative models.
- Generative models:
 - These models get described by the joint distribution $p(\vec{x}, y)$.
 - Generative adversarial networks is just one of the examples for generative models.

Generally, it is way harder to train a generative model. The aim of this thesis is to infer the structure of the cosmic web based on the spatial distribution of the galaxies. So, given some input image of galaxy positions, we want to obtain some output image of the gas distribution. The pix2pix model [Isola et al., 2016], a conditional generative adversarial network, is perfectly suitable for image-to-image translation, hence we decided to go with this model. In this chapter, we will briefly talk about the basics of unconditional and conditional GANs and how they get trained.

3.3.1 Generative adversarial networks (GANs)

The generative adversarial network aims to learn to generate new samples based on the given training data set. Figure 3.9 shows the capabilities of a GAN - these people do not really exist as they have been generated by a GAN.



Figure 3.9: Capabilities of GAN models [Karras et al., 2018]

Regarding the structure of the model, normally it consists of two neural networks: the generator and the discriminator. The former aims to create data which should be similar to the training data. The latter should differentiate between “real” and “fake” data by assigning a scalar to each data set from 0 to 1. This scalar stands for a probability corresponding to the realness of the data sample with 0 being fake and 1 being real.

This is approximately what the learning process looks like: The generator G takes in a random noise vector $\vec{z}$ and generates from it some fake data sets which are mixed with the real ones. These are then fed to the discriminator D which assigns a probability to each one of them. After the assignment of the probabilities is done, it is revealed whether the discriminator was correct or not. Based on this cross check, the parameters are adjusted in either the discriminator (wrong assessment) or the generator (correct assessments). However, in practise, the discriminator and generator weights are updated after each training step. Figure 3.10 shows a graphical illustration of the learning process of a GAN.

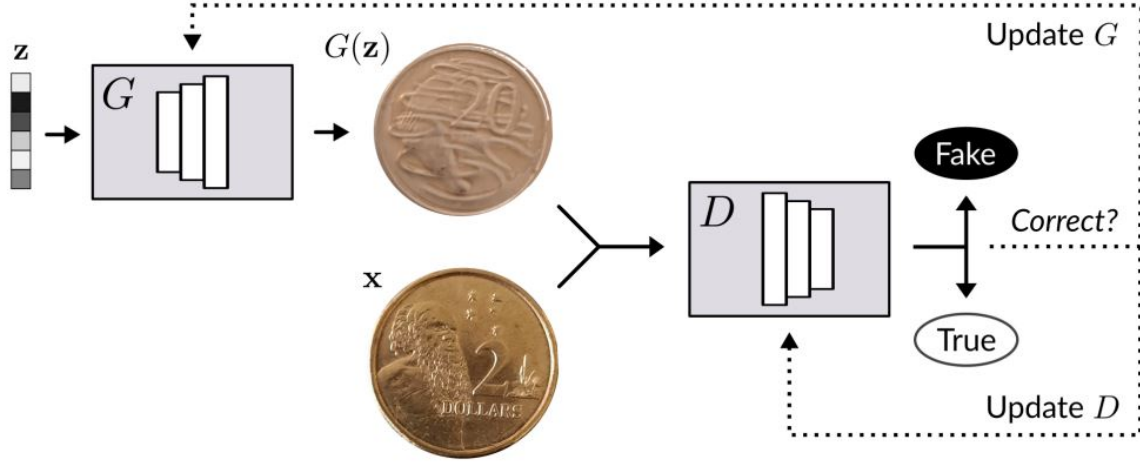


Figure 3.10: GAN's learning process [List, 2021]

We can imagine these two networks as two players who are competing against each other whilst simultaneously improving the “opponent”. The generator learns a mapping from a random noise vector $\vec{z}$ (which is the only input to the generator) to the output data sample $\vec{y}'$, i.e. $G : \vec{z} \mapsto \vec{y}'$, which should be drawn from a distribution that closely resembles the ground truth distribution. $p_{\vec{y}}$ and $p_{\vec{z}}$ stand for the corresponding distributions. The loss function has the following form [Isola et al., 2016]:

$$L(W^G, W^D) = \mathbb{E}_{\vec{y} \sim p_{\vec{y}}} \log D(\vec{y}; W^D) + \mathbb{E}_{\vec{z} \sim p_{\vec{z}}} \log (1 - D(G(\vec{z}; W^G); W^D))$$

The first part of the loss functions consists of the expectation value $\mathbb{E}$ for the probabilities of the real data sets $D(\vec{y})$. The second part of the loss function computes the expectation value of the negative probability of the by generator generated fake data sets.

To obtain the optimal weight vectors W_*^G and W_*^D for the generator and discriminator respectively, we need to first think about the structure of the loss function. Clearly, the aim of the discriminator is to correctly classify all the real data sets, i.e.

$$D(\vec{y}; W^D) = 1 \implies \mathbb{E}_{\vec{y} \sim p_{\vec{y}}} \log D(\vec{y}; W^D) \approx 0$$

and incorrectly classify all the fake data sets, i.e.

$$D(G(\vec{z}; W^G); W^D) = 0 \implies \mathbb{E}_{\vec{z} \sim p_{\vec{z}}} \log (1 - D(G(\vec{z}; W^G); W^D)) \approx -\infty$$

hence, the generator and the discriminator are trying to “pull” into opposite directions. Let us now formulate this findings as an optimization problem:

$$(W_*^G, W_*^D) = \arg \min_{W^G} \max_{W^D} L(W^G, W^D)$$

The ideal generator, i.e. a generator which would perfectly fool the discriminator every single time, corresponds to the underlying data distribution, namely $p_{\vec{y}}$.

3.3.2 pix2pix

Pix2pix is a conditional GAN learning a conditional generative model (Isola et al. [2016], Mirza and Osindero [2014]). This makes cGANs suitable for image-to-image translation tasks, where

we condition on an input image and generate a corresponding output image. The figure 3.11 shows the capabilities of a pix2pix model.

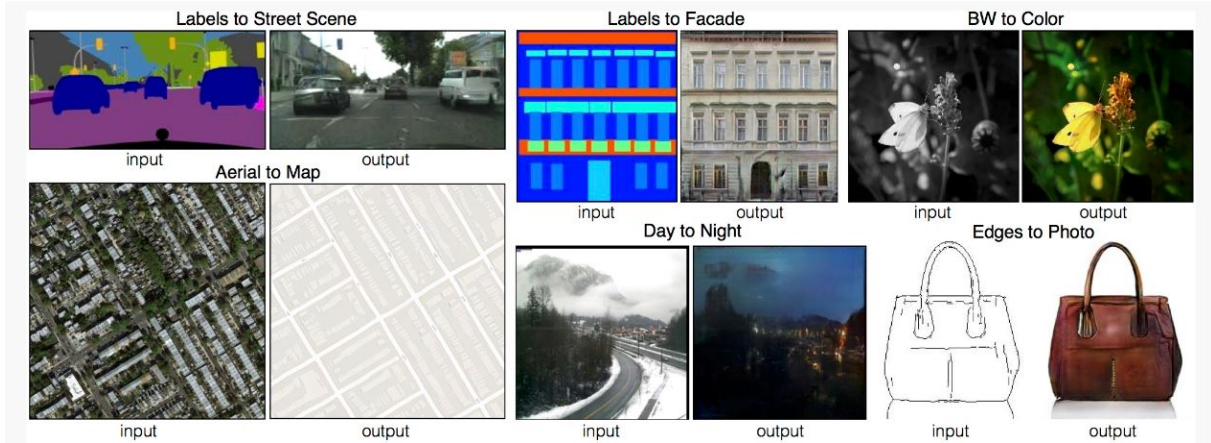


Figure 3.11: Capabilities of cGAN models [Isola et al., 2016]

The training approach is the same as for the unconditional GANs - the generator creates some outputs with regards to the input, mixes them with the real data sets and the discriminator assesses whether the given tuples are real or fake. Followingly, the parameters of either the discriminator or the generator are adjusted based on the correctness of the predictions. Although, just as previously, in practice both are adjusted.

Conditional GANs learn a mapping from observed (input) image $\vec{x}$ and random noise vector $\vec{z}$ to an output image $\vec{y}$, i.e. $G : \{\vec{x}, \vec{z}\} \mapsto \vec{y}$. Here the loss function of a conditional GAN [Isola et al., 2016]:

$$\begin{aligned}
 L(W^G, W^D) = & \mathbb{E}_{\vec{x} \sim p_{\vec{x}}, \vec{y} \sim p_{\vec{y}}} \log D(\vec{x}, \vec{y}; W^D) \\
 & + \mathbb{E}_{\vec{x} \sim p_{\vec{x}}, \vec{z} \sim p_{\vec{z}}} \log (1 - D(\vec{x}, G(\vec{x}, \vec{z}; W^G); W^D)) \\
 & + \lambda \mathbb{E}_{\vec{x} \sim p_{\vec{x}}, \vec{y} \sim p_{\vec{y}}, \vec{z} \sim p_{\vec{z}}} \|\vec{y} - G(\vec{x}, \vec{z}; W^G)\|_1
 \end{aligned} \tag{3.15}$$

The yellow marked parts in the loss functions show the differences with regards to the GAN loss function. Now we have an additional input distribution $p_{\vec{x}}$. As we can see, the loss function will also be rewritten in such a way that the discriminator and generator are conditioned on an input. Additionally, we have a regularization term which enables the generator to be near to the ground truth. We use the L1 loss instead of the L2 loss to avoid blurring.

The optimal parameters W^D and W^G are computed analogously to the unconditional GAN:

$$(W_*^G, W_*^D) = \arg \min_{W^G} \max_{W^D} L(W^G, W^D)$$

However, in practice, we take a slightly different approach. First, since we are dealing with two separate neural networks, let us split up equation 3.15 into two loss functions which are only discriminator- (discriminator loss) and generator-dependent (generator loss), i.e.

$$\begin{aligned}
 L_D(W^G, W^D) = & \mathbb{E}_{\vec{x} \sim p_{\vec{x}}, \vec{y} \sim p_{\vec{y}}} \log D(\vec{x}, \vec{y}; W^D) \\
 & + \mathbb{E}_{\vec{x} \sim p_{\vec{x}}, \vec{z} \sim p_{\vec{z}}} \log (1 - D(\vec{x}, G(\vec{x}, \vec{z}; W^G); W^D)) \\
 L_G(W^G, W^D) = & \mathbb{E}_{\vec{x} \sim p_{\vec{x}}, \vec{z} \sim p_{\vec{z}}} \log (1 - D(\vec{x}, G(\vec{x}, \vec{z}; W^G); W^D)) \\
 & + \lambda \mathbb{E}_{\vec{x} \sim p_{\vec{x}}, \vec{y} \sim p_{\vec{y}}, \vec{z} \sim p_{\vec{z}}} \|\vec{y} - G(\vec{x}, \vec{z}; W^G)\|_1
 \end{aligned}$$

Furthermore, the second part of L_D and the first part of L_G does not perform well in practice, particularly if the discriminator correctly classifies the fake data sets, the gradient of the generator loss vanishes [Goodfellow, 2017], as the figure 3.12 shows.

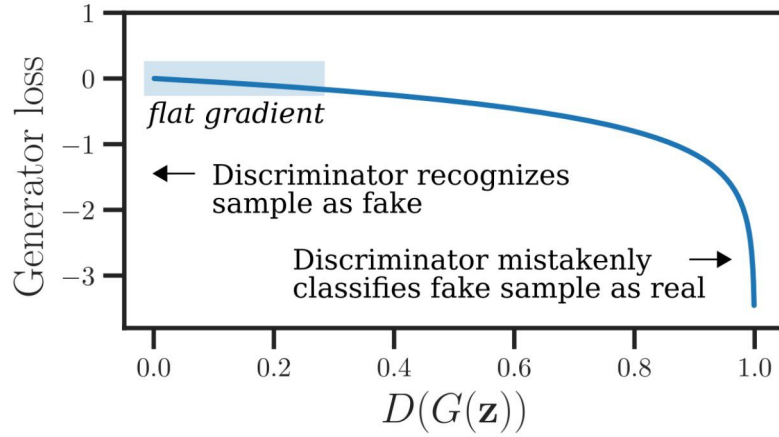


Figure 3.12: The effect of vanishing gradients [List, 2022]

In order to solve this, we must reformulate the problem: instead of minimizing the probability of discriminator's correct assessments, let us maximize the probability of discriminator's wrong assessments, as the figure 3.13 shows.

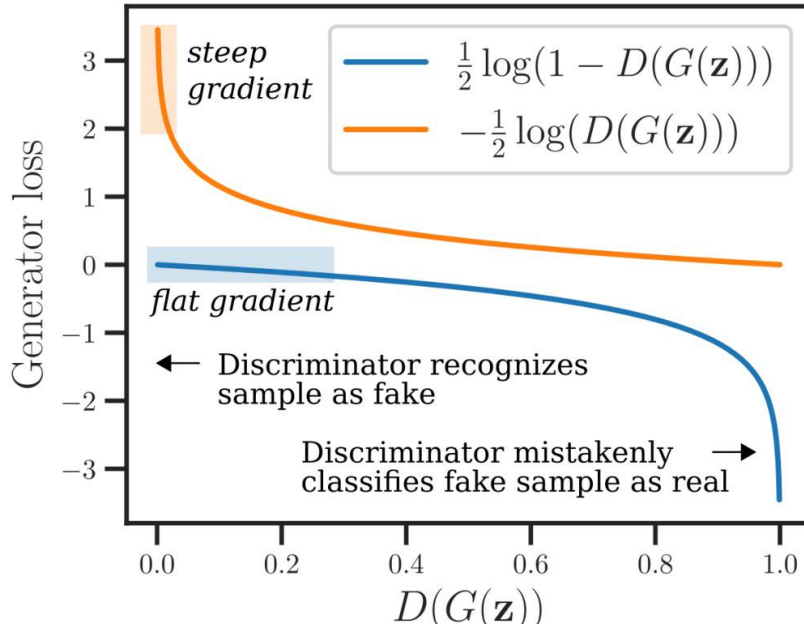


Figure 3.13: The solution of vanishing gradients [List, 2022]

Hence, instead of

$$\log(1 - D(\vec{x}, G(\vec{x}, \vec{z}; W^G); W^D))$$

let us use [Goodfellow, 2017]

$$-\log(D(\vec{x}, G(\vec{x}, \vec{z}; W^G); W^D)) \quad (3.16)$$

Furthermore, we also need to invert the sign of first part of L_D since this term has a similar structure to 3.16. Both loss functions take thus the following forms [Goodfellow, 2017]:

$$\begin{aligned} L_D(W^G, W^D) = & -\mathbb{E}_{\vec{x} \sim p_{\vec{x}}, \vec{y} \sim p_{\vec{y}}} \log D(\vec{x}, \vec{y}; W^D) \\ & -\mathbb{E}_{\vec{x} \sim p_{\vec{x}}, \vec{z} \sim p_{\vec{z}}} \log(D(\vec{x}, G(\vec{x}, \vec{z}; W^G); W^D)) \end{aligned}$$

$$L_G(W^G, W^D) = -\mathbb{E}_{\vec{x} \sim p_{\vec{x}}, \vec{z} \sim p_{\vec{z}}} \log(D(\vec{x}, G(\vec{x}, \vec{z}; W^G); W^D)) \\ + \lambda \mathbb{E}_{\vec{x} \sim p_{\vec{x}}, \vec{y} \sim p_{\vec{y}}, \vec{z} \sim p_{\vec{z}}} \|\vec{y} - G(\vec{x}, \vec{z}; W^G)\|_1$$

Finally, we need to minimize these two loss functions in order to obtain the optimal generator and discriminator. Additionally, we need to slightly adjust the learning process. Regarding the generator, the generator produces an output from an input image. This pair is fed to the discriminator and a scalar probability is the output. These scalar probabilities of an entire batch of samples are summarized in a vector. A cross entropy loss function ($-\mathbb{E} \log$) is applied to this vector and a vector of the corresponding size filled with 1s. The reason for the 1s is the fact that we aim to completely fool the discriminator, i.e. we want the discriminator to output 1 for all generated samples. Simultaneously, we compute the mean absolute error between the target image and the output of the generator and multiplies it with λ , which we add to the result of cross entropy loss function. Based on the result, we update the gradients.

Regarding the discriminator, the generator produces an output from an input image. This pair is fed to the discriminator which outputs the probabilities. These probabilities are fed inserted with a vector filled with 0s (discriminator correctly discovers fake data sets) to the cross entropy loss function. Additionally, the ground truth pair of output and input images are fed as well to the discriminator which output probabilities. Together with a vector filled with 1s (discriminator correctly discovers real data sets), these probabilities are inserted into the cross entropy function. Finally, we add the results of the cross entropy loss functions and apply the according gradients to the discriminator. The learning process for the generator and the discriminator are visually summarized in figure 3.14.

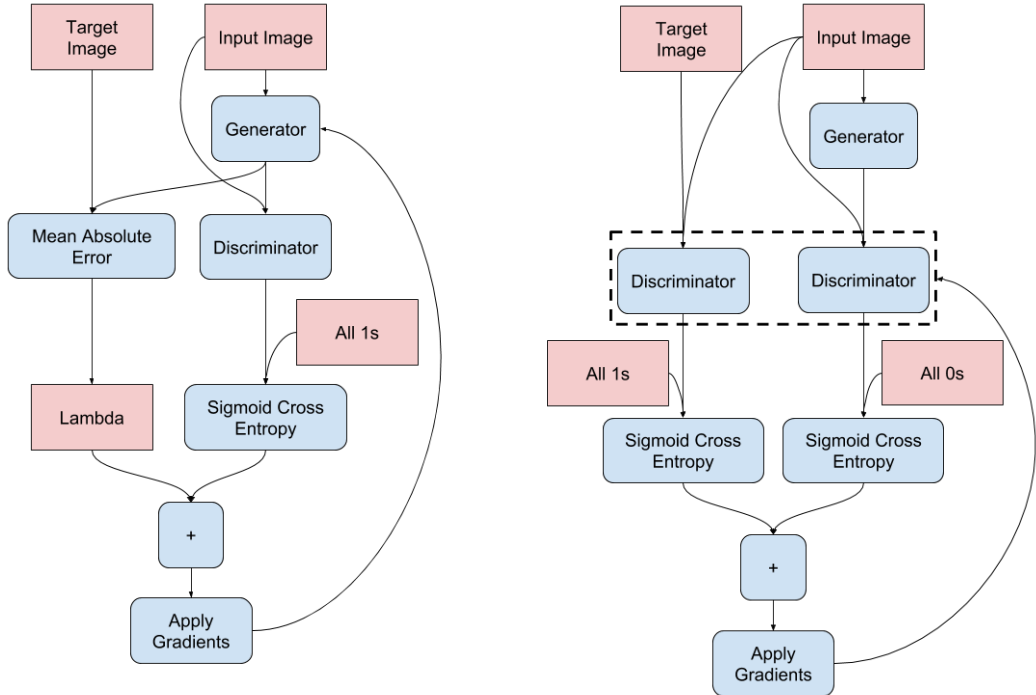


Figure 3.14: Learning processes in pix2pix, left for the generator and right for the discriminator [Abadi et al., 2022]

Generally, to achieve better performances on the data set, we use a deep convolutional neural network for the discriminator and generator. Furthermore, in this thesis, the generator uses the so-called decoder-encoder structure. U-Net is one possible variant (figure 3.15) of this architecture. In this case, in every step of the upsampling path, we use the already acquired corresponding map from the encoder and concatenate them [Ronneberger et al., 2015]. There is a possibility to use an addition instead of a concatenation, however one must

use the corresponding map from the encoding part such that the dimensions work out. The difference between a conventional encoder-decoder structure and U-net is the use of the so-called skip-connections in U-nets which bypass some layers and propagate the output of these layers on to not necessarily the following layer [Dutta, 2021], which can be seen in figure 3.15.

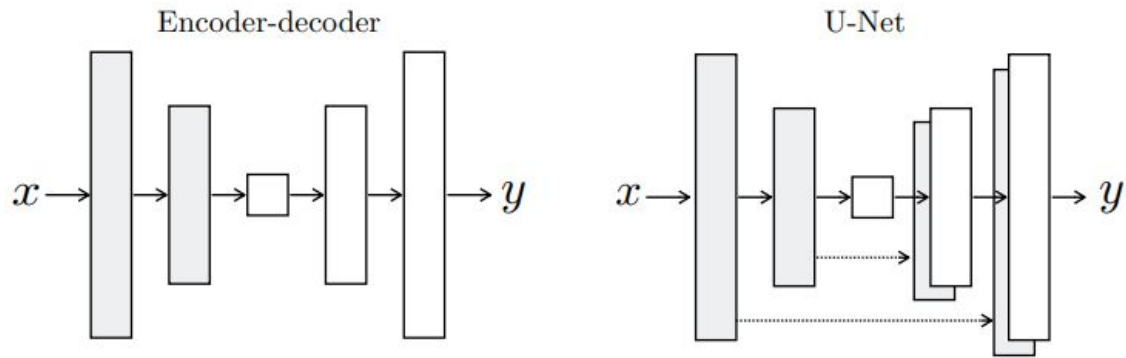


Figure 3.15: Different architectures [Isola et al., 2016]

Chapter 4

Approach

4.1 Data set

The repository with all the code can be found in [Sabo, 2023]. For this thesis, we have used data from the cosmological hydrodynamical simulation CAMELS [Villaescusa-Navarro et al., 2021]. These simulations showcase the evolution of a mock universe, beginning shortly after the Big Bang until the present time, considering all relevant physical processes that influence structure growth and galaxy formation. Concretely, for the galaxy positions, we have used 16 different 'halo finder' outputs, which contain a catalog of all gravitationally bound structures found in the simulation snapshot with a box size of 25 Mpc/h. From this data set, we are only interested in the positions (x, y and z coordinates) of the galaxies.

Regarding the fields (since we want this thesis to be as general as possible, we can take instead of the gas particle density any other fields such as temperature fields), we have taken the files which contain a subset of the fields that are present in a full snapshot corresponding with the galaxy positions files. Regarding the data, we will need the position of the gas particles, their densities, their metallicities, their internal energies, their electron abundance and the box size of the given simulation.

Thanks to this data, we can derive the density, metallicity and temperature fields by using the `NearestNDInterpolator` from `scipy.interpolate` to interpolate the field values at the gas cell particle positions to the image pixels. It is worth noting, this can be done fully analogously with any field property, gas density, metallicity and temperature are just the most common examples in the astrophysics.

4.2 Creation of training and test set

The whole code can be found in section 7.1. After loading in all the files, we must decide what kind of resolution (N_{side} , see 3.7) the data sets should have. We have opted for

$$N_{side} = 64$$

due to a great balance between the training time of neural networks and the quality of resolution of the images. This corresponds to

$$N_{pixels} = 49152.$$

Now, let us perform the cuts to generate data samples. First, let us set up a square grid, a set of points with equal spacing between each other, with periodic boundary conditions, i.e.

$$f(\vec{x} + L) = f(\vec{x})$$

with L being the box size. Concretely, these points will go from the origin up to 25 with a spacing of 5 in all 3 directions, i.e. that sums up to

$$\left(\frac{25 - 0}{5}\right)^3 = 5^3 = 125$$

points on the grid per simulation. We can imagine these points as kind of mini observatories from where “mini astronomers” observe the surrounding night sky in their vicinity. Next, we must define this vicinity, i.e. defining the radius of two spheres (r_{min} and r_{max}). The sampling process can be seen in figure 4.1.

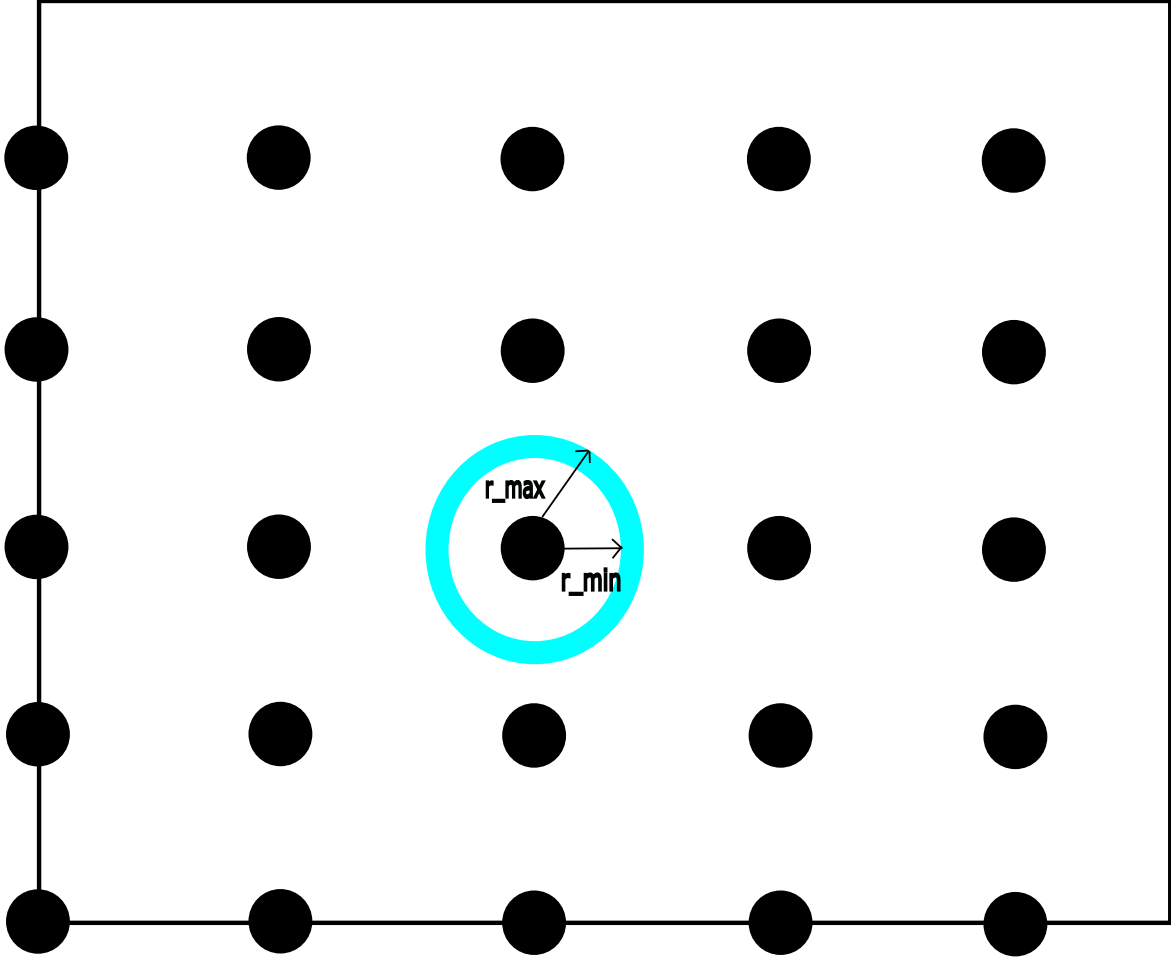


Figure 4.1: The idea behind sampling

The galaxies which are between these two spheres will be deemed as visible to the “mini observers”. In each pixel, we count how many visible galaxies fall into it. We decided for the following two radii:

$$\begin{aligned} r_{min} &= 8 \text{ Mpc/h} \\ r_{max} &= 10 \text{ Mpc/h} \end{aligned}$$

Regarding the fields, we need not only the two radii, but also a number of intermediary steps n_{steps} we want to take in the interval $[r_{min}, r_{max}]$. We decided for

$$n_{steps} = 20$$

Having defined the number of steps, we let the saved field `NearestNDInterpolator` model run on the points on the sphere at each step and averaged them out. This is the field corresponding to the galaxies between the two spheres.

Additionally, we perform data augmentation on the training set. Standard augmentation techniques in Euclidean spaces are rotation or translation. In this case, we will use a west-to-south flip (rotation around the prime meridian), a north-to-south flip (rotation around the equator) and both simultaneously. All of these augmentations produce new valuable data samples out of the existing ones. The DeepSphere convolutions layers require all data samples to have the nested pixel indexing (see Figure 3.8). From the 16 files, 14 have been assigned to the training data set and the remaining 2 to the test set. All in all, these sets have the following number of samples:

Training set:

$$125 \cdot 14 \cdot 4 = 7000 \quad \text{number of grid points, number of files, data augmentation}$$

Test set:

$$125 \cdot 2 = 250 \quad \text{number of grid points, number of files}$$

Both sets have the following shapes:

Training set:

$$(2, 7000, 49152, 1) \quad \text{galaxy+field,samples, pixels, 1}$$

Test set:

$$(2, 250, 49152, 1) \quad \text{galaxy+field,samples, pixels, 1}$$

4.3 Normalization of the data sets

The whole code can be found in section 7.2. Up until now, we have just summed up all the visible galaxies or averaged out the fields, hence the values vary significantly. Any machine learning model would have difficulties finding a function which would infer the corresponding fields given some galaxies. In order to solve this issue, we need to normalize the sets.

We need to perform the normalization over the training and test set simultaneously, though separately over the input and output samples. Regarding the galaxy positions data set, first we find out the minimum and the maximum value in the entire data set. Once we have done that, we linearly interpolate the values to (0, 1) given the minimum and maximum value with the help of the function `np.interp()`.

Concerning the fields data set, let us first compute the logarithm of the data. This is due to the fact that the values span multiple orders of magnitude. Next, we proceed just as previously - first compute the minimum and maximum and then linearly interpolate using `np.interp()`.

Last but not least, in order to have just two data sets (training and test set), let us concatenate the fields. Thus we will get sets of the following shapes:

Training set:

$$(7000, 49152, 4) \quad \text{samples, pixels, galaxies+density+metallicity+temperature}$$

Test set:

$$(250, 49152, 4) \quad \text{samples, pixels, galaxies+density+metallicity+temperature}$$

4.4 Finding a model

Multiple models have been used throughout the thesis. The biggest issue of the majority of models was the occurrence of the so-called artifacts. These are checkerboard patterns which emerge from the overlaps in deconvolution path [Odena et al., 2016]. Petroff et al. [2020],

who built a GAN based on DeepSphere CMB Foreground Cleaning, has shown a way how to circumvent this issue. Instead of using the transposed convolution (deconvolution), they upsample each pixel with the nearest-neighbor upsampling and subsequently, a convolution is performed on the higher resolution images. The primary model in this thesis is based on the model proposed by Petroff et al. [2020]. Tables 7.1 and 7.2 present the used architectures for the discriminator and generator respectively. We use the Adam optimizer with a learning rate of $2 \cdot 10^{-4}$ and $\beta_1 = 0.5$. A couple of variants have been tried out such as:

1. Relu activation after concatenation instead of after UpSampling in the generator
2. Same generator as in option 1, the sequence Sphere Convolution - Batch Normalization - Leaky ReLU after Graph Pooling twice in the discriminator
3. Same discriminator as in option 2, UpSampling and activation switched in the bottleneck
4. Same discriminator as in option 2, generator same as in option 3 without Linear Combination (adds a bias in the layer) before bottleneck
5. Same discriminator as in option 2, batch normalization before leaky ReLU in the down-sampling and before concatenation in the UpSampling part of the generator
6. Layer normalization instead of batch normalization in discriminator and generator from option 5
7. Layer normalization dropped (i.e. generator as in option 4)
8. Discriminator as in option 2 (Sphere Convolution - Batch Normalization - Leaky ReLU twice), in generator Sphere Convolution - Leaky ReLU twice
9. Discriminator as in option 8, generator has the sequence Sphere Convolution - Batch Normalization - Leaky ReLU twice after Graph Pooling
10. Same generator and discriminator as in option 9, no sigmoid at the end of both
11. Same generator and discriminator as in option 9, fixed filter size (6)
12. Same generator and discriminator as in option 9, larger fixed filter size (30)

The models have been trained for 200 epochs with a batch size of 1 and $\lambda = 100$ in the loss function. Option 8 has produced the best results out of these models. Its code ([Abadi et al., 2022], [Brownlee, 2021], [Perraudin et al., 2019], [Petroff et al., 2020] and [pawangfg, 2022]) can be found in the section 7.4. Furthermore, we have tried out these variants of option 8:

1. $\lambda = 10$ in the loss function
2. Addition of layers instead of concatenation in the upsampling part of the generator, $\lambda = 100$
3. Addition of layers instead of concatenation in the upsampling part of the generator, $\lambda = 10$

These have been by far the best models. Just as previously, all of them have been trained for 200 epochs with a batch size of 1. Since both models with $\lambda = 100$ have proven to be superior, we went ahead with these two. Furthermore, since we have already created data samples with other fields such as metallicity and temperature, we tried out a model which outputs 3 (density, metallicity, temperature) channels instead of one. Here are the final 4 models:

1. Concatenation in the upsampling part of the generator
2. Addition of layers instead of concatenation in the upsampling part of the generator
3. 3 channels (density, metallicity, temperature) as output of the generator with concatenation in the upsampling part of the generator
4. 3 channels (density, metallicity, temperature) as output of the generator with addition in the upsampling part of the generator

These models have been trained for 400 epochs with a batch size of 1. In this case, the single output models have proven to be superior for the same number of epochs. Possible reasons for this might be that the models with 3 output channels need either significantly more parameters compared to single output models or significantly more epochs to run. Since none of these conditions have been satisfied, the single output models have shown better performances.

Out of the two single output models, the version with concatenation has wound up being the slightly better version. The version with addition has generated small dark holes in some of the data samples even though the other model had also shown this frailty to some extent. A possible reason for the slight difference might be the fact that the model with concatenation has around 6% more parameters in the generator.

Chapter 5

Results

5.1 Model performance evaluation

Some of the results in this paper have been derived using the healpy [Zonca et al., 2019] and HEALPix package [Gorski et al., 2005]. The code which generates the plots can be found in Section 7.4. The model ([Abadi et al., 2022], [Brownlee, 2021], [Perraudin et al., 2019], [Petroff et al., 2020] and [pawangfg, 2022]) has been run for 400 epochs.

Figure 5.1 shows the performance on 5 random samples from the training set, whereas figure 5.2 depicts the performance on 5 random samples from the test set. In the left column, we have the input galaxy distribution field (which we condition on), the middle column the generated gas density map, and the right column the ground truth gas density field from the CAMELS simulation [Villaescusa-Navarro et al., 2021].

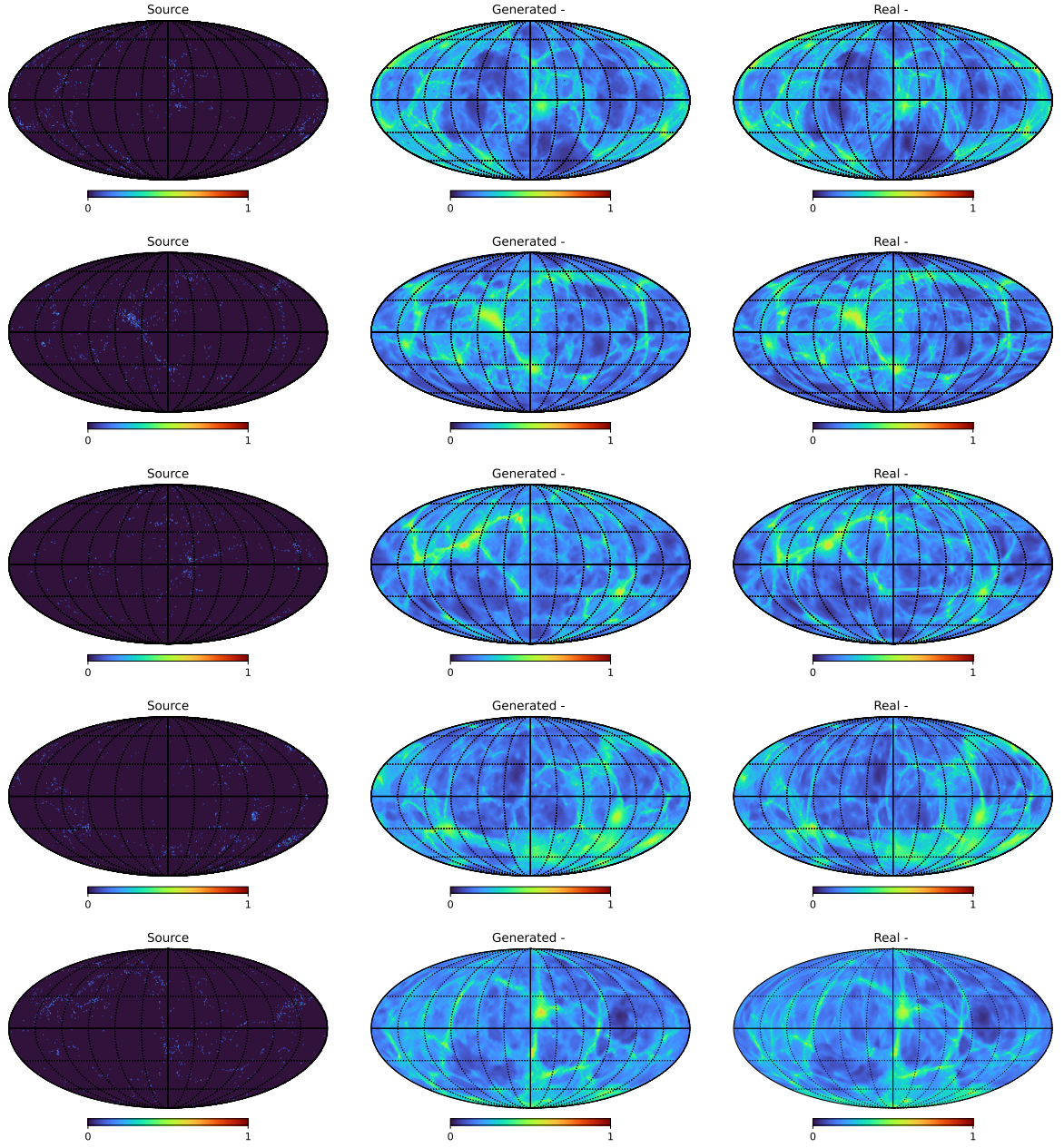


Figure 5.1: Performance of the trained pix2pix model on the training set. Left: the spatial distribution of the input galaxies. Middle: Generated cosmic web by the model. Right: The corresponding real cosmic web.

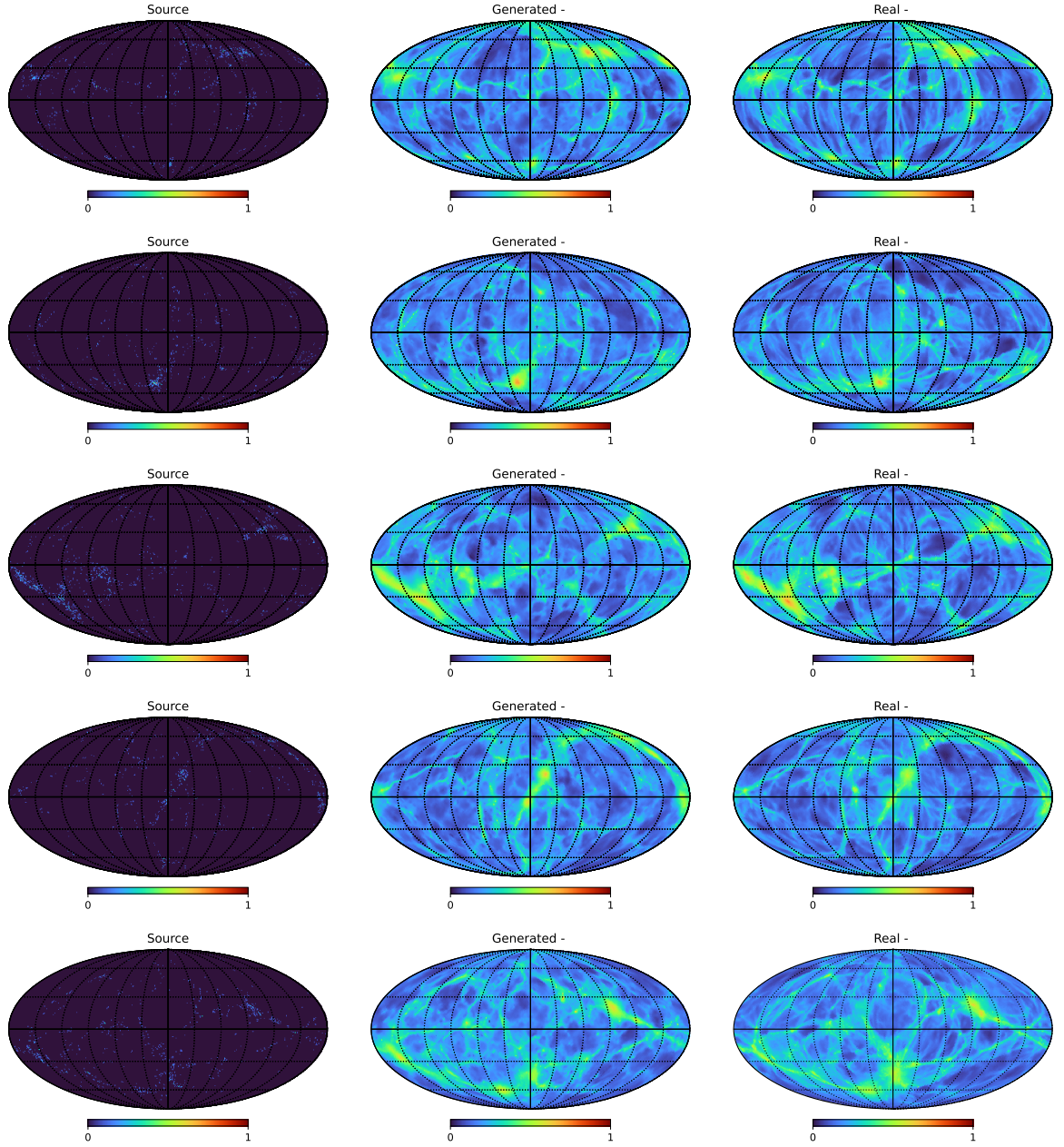


Figure 5.2: Performance of the trained pix2pix model on the test set. Left: the spatial distribution of the input galaxies. Middle: Generated cosmic web by the model. Right: The corresponding real cosmic web.

Visually, the model appears to have worked very well in predicting the structure of the corresponding cosmic web, despite having a very sparse input matrices. However, some filaments, in particular some of which are present in regions that are lacking in galaxies, do not look like those of realistic samples. Furthermore, a few dark holes are present particularly in the results for the test set. In order to quantitatively assess how well the model has worked, let us have a look at the statistical analysis.

5.2 Statistical analysis

In the statistical analysis, we are interested in the power spectra of the generated, real image and their ratio. Additionally, a PDF (probability density function; tells us the probability of the occurrence of a given value) of the generated and real values will be plotted as well. The code which generates the plots can be found in section 7.6.

Figure 5.3 shows the performance on 5 random samples from the training set, whereas figure 5.4 depicts the performance on 5 random samples from the test set. Just as previously, in the left column, we have the input galaxy distribution field, in the second one the generated gas density map, in the third one the ground truth gas density field from the CAMELS simulation [Villaescusa-Navarro et al., 2021], in the fourth one a plot of the power spectra of the generated, real image and their ratio and finally, in the last one, the PDFs of the values of the real and generated image.

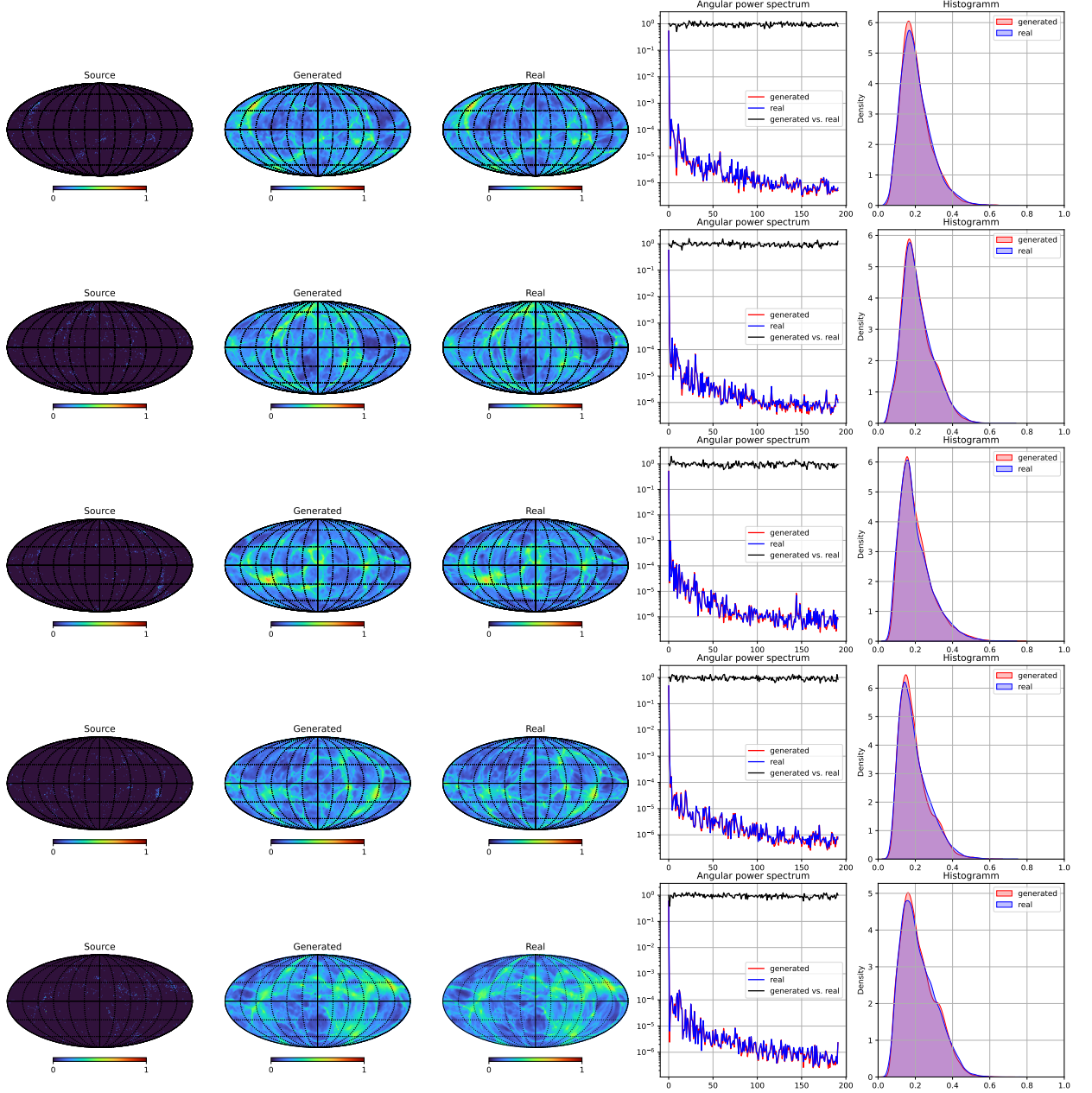


Figure 5.3: Statistical analysis of the model on the training set. Columns from left to right; First: the spatial distribution of the input galaxies. Second: Generated cosmic web by the model. Third: The corresponding real cosmic web. Fourth: the angular power spectrum of the real and generated cosmic webs. Fifth: the histogram of values of the real and generated cosmic webs.

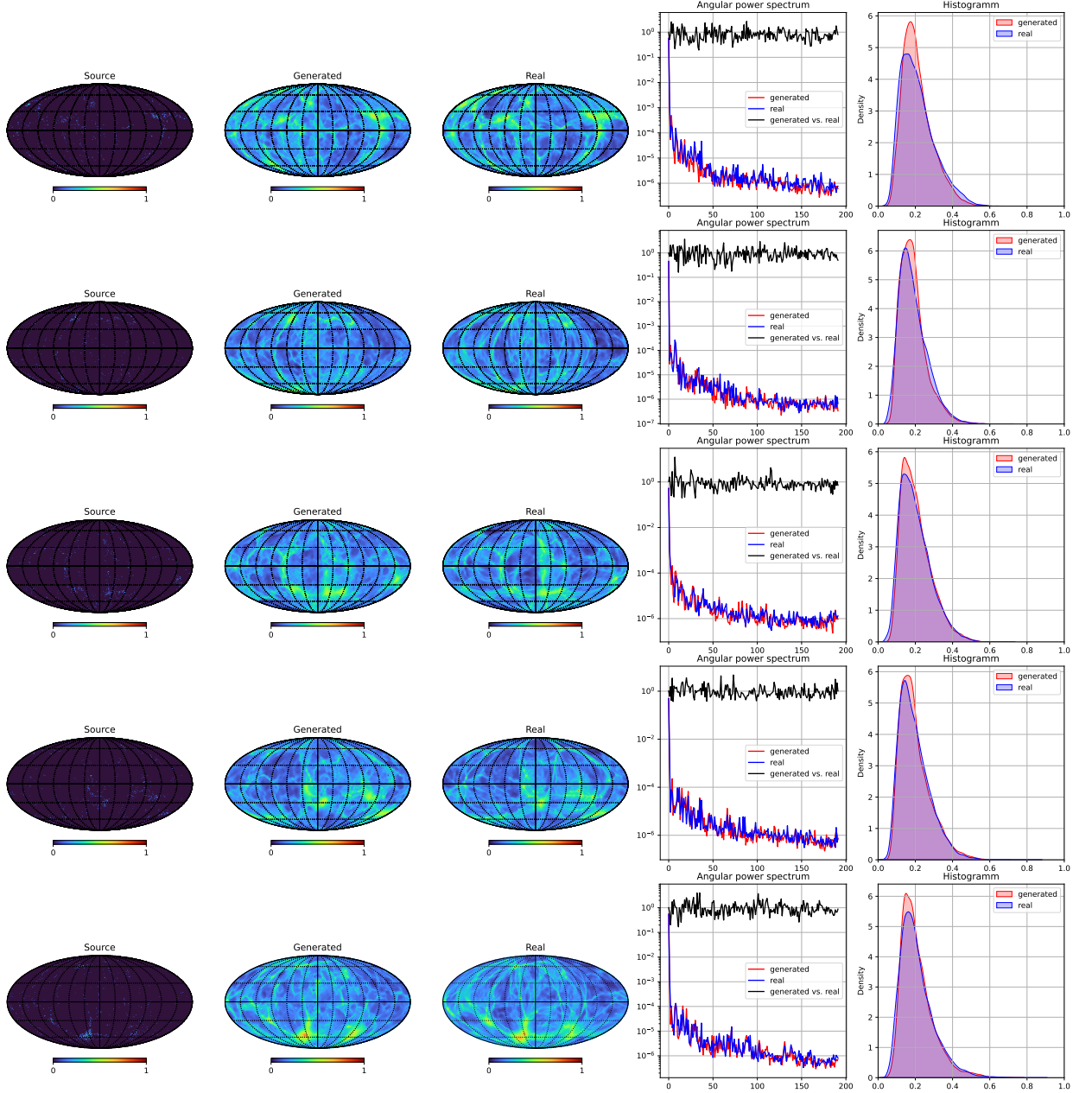


Figure 5.4: Statistical analysis of the model on the test set. Columns from left to right; First: the spatial distribution of the input galaxies. Second: Generated cosmic web by the model. Third: The corresponding real cosmic web. Fourth: the angular power spectrum of the real and generated cosmic webs. Fifth: the histogram of values of the real and generated cosmic webs.

Regarding the training set, the PDFs and power spectra overlap almost perfectly. Furthermore, the ratio of the power spectra only confirms the results of the PDFs since the majority of values is around 1. Concerning the test data set, the power spectra seemingly overlap quite well. Moreover, the shape of the PDFs is generally quite well conserved, however the model tends to overestimate the true value visible since the peaks of the PDFs do not match entirely. This may possibly imply slight overfitting.

5.3 Uncertainty quantification

Now, let us try to see how stable the model is, i.e. how it reacts to small perturbations of the galaxy positions. Seemingly, the model predicts always the same gas density for a given galaxy distribution, despite the fact that the generator starts with random noise vector. Hence, one

would expect multiple different results. The aim of this experiment is to pinpoint regions in the reconstructed maps which remain unaffected by the small perturbations (stable regions) and those which do not (unstable regions).

First, we need to generate a new data set with noise. It is necessary to generate N noisy variants to a couple of data samples in order to see how the model behaves on average. The perturbations need to be applied to the galaxy positions, i.e. before the cuts, and not to the galaxy arrays after the cuts. Let us use the Gaussian distribution to generate noise. Thus we need to define the mean μ and the standard deviation σ , so let us make the choices:

$$\begin{aligned}\mu &= 0 \\ \sigma &= 5 \cdot 10^{-2}\end{aligned}$$

Let us now add noise to the galaxy positions in the following manner:

$$\begin{aligned}(\phi + \Delta\phi, \theta + \Delta\theta \sin \theta) \\ \Delta\phi \sim N(\mu, \sigma) \\ \Delta\theta \sim N(\mu, \sigma)\end{aligned}$$

ϕ and θ live in the following intervals:

$$\begin{aligned}\phi &\in [0, 2\pi] \\ \theta &\in [0, \pi]\end{aligned}$$

However, due to the addition of noise it may well happen for θ that we get a value which is outside of either of the interval. Hence, let us wrap the coordinate in the following way:

$$\begin{aligned}\text{if } \theta < 0: \\ \theta_{new} &= -\theta_{old} \\ \text{if } \theta > \pi: \\ \theta_{new} &= 2\pi - \theta_{old}\end{aligned}$$

Alternatively, if the noise gets unexpectedly big, we can wrap the coordinates in the followingly:

$$\theta_{new} = \theta_{old} \mod 2\pi$$

We will stick to the first wrapping method since the noise will not get that large due to the chosen σ . Let us now perform the discussed for the first 5 grid points for only the test data set 100 times. The code can be found in section 7.7.

Now, having created the noisy data, let us have a look at the performance of the model when exposed to it. The figure 5.5 shows the results of selected 5 galaxy positions distributions out of the test set - in the first column we have the (noiseless) source data, in the second one the true (noiseless) corresponding cosmic webs, in the third one the average of 100 generated cosmic webs out of the noisy source data, in the fourth the standard deviation of them and in the last one relative variance, i.e. standard deviation/mean. The code can be found in section 7.8.

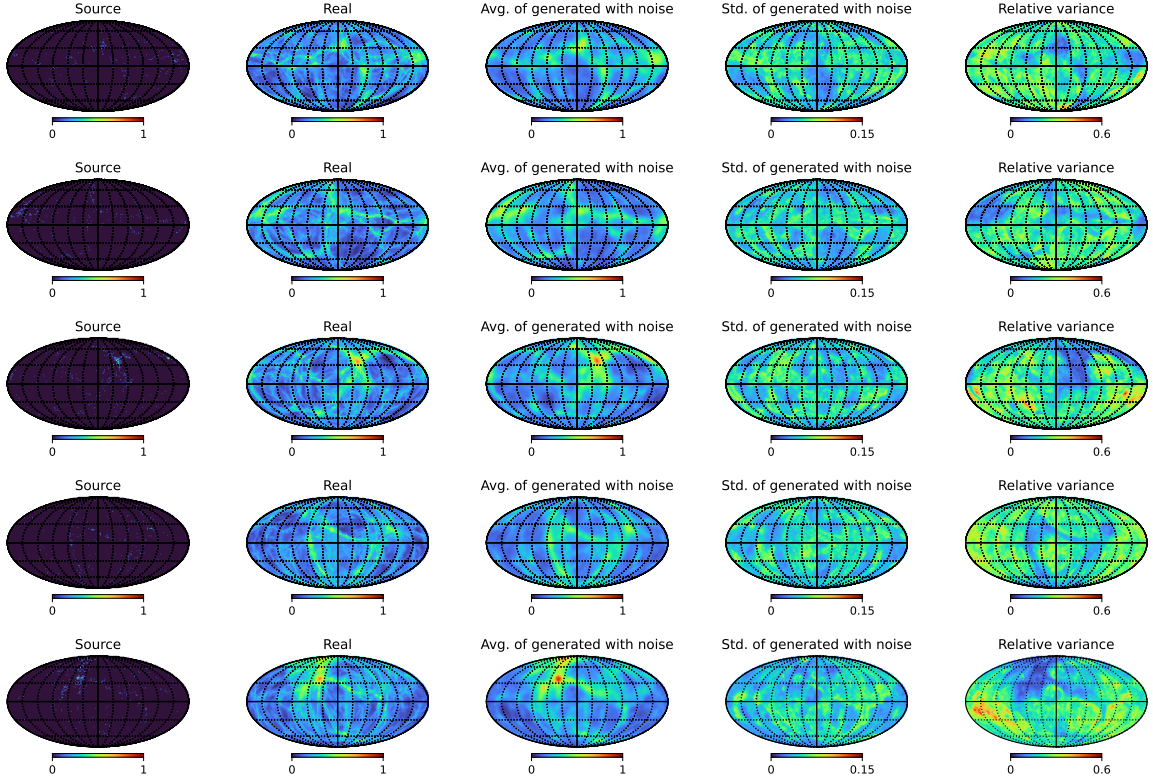


Figure 5.5: Statistical analysis of the model on the noisy test set. Columns from left to right; First: the spatial distribution of the input galaxies. Second: The corresponding real cosmic web. Third: the average of generated cosmic webs. Fourth: the standard deviation of generated cosmic webs. Fifth: the relative variance of generated cosmic webs.

Clearly, the sections with the highest amount of galaxies remain unchanged, even when noise is present. Furthermore, the biggest differences appear in the regions with lower amounts of galaxies. The model correctly predicts regions with higher densities of galaxies, but performs significantly worse in regions with low densities. Regarding the relative variance, it is evident that the regions with largest values have only a few galaxies, if any. The model tries to guess the structure of the cosmic web in these regions and understandably, these guesses are quite unstable, i.e. small perturbations can change them significantly.

Chapter 6

Conclusions and outlook

In this thesis, we have implemented a conditional generative adversarial network on the sphere to infer the intergalactic gas distribution given the distribution of discrete galaxy positions. This offers a computationally more attractive alternative to computing it with simulations, which is a rather tedious task.

All in all, the proof of concept worked quite well as the mapping from galaxy positions to cosmic webs has been to large extent successfully determined. This applies particularly for galaxies along the filaments as the mapping there has been generally correctly predicted. Nonetheless, the structure especially in cosmic voids and less dense regions have not looked like realistic structures -this makes sense though as there almost no data in these regions and the model is forced to guess.

Regarding the statistical analysis, the power spectra and PDFs for the training set overlap almost perfectly. This means that the magnitudes of the pixels have to large extent been predicted correctly. For the test set though, the peaks of the PDFs for the generated images point to a systematic overestimation of the values of the true pixels. This may well imply slight overfitting of the model.

Comparing our thesis to the current research, there are generally very few applications of conditional GANs in the astronomy or on spheres. Some of the examples are Perraudin et al. [2021], who have trained a cGAN model which outputs mass maps given some cosmological parameters, List et al. [2019], who have trained a cGAN model to predict gas density with dark matter annihilation feedback (DMAF) given gas density without DMAF, Zhao et al. [2019], who have trained a mask-pix2pix model, which recovers overexposed astronomical photos and Tröster et al. [2019], who trained a cGAN which generates the corresponding gas pressure given the matter density. However, there a lot of papers, Rodríguez et al. [2018], Ullmo et al. [2021] and Li et al. [2021] just to name a few, which use GAN-based approaches to generate new statistically consistent samples of cosmic webs.

Concerning the uncertainty quantification, despite the added noise, the model predicted generally very well the structure of the cosmic web along the filaments. The biggest differences can be observed in the less dense regions, particularly in cosmic voids, which was to be expected, since we have already seen how the model has struggled in these regions due to the lack of data. Clearly, the model is quite unstable in these regions.

This model can be expanded into various directions in the future, which would be:

1. avoid overfitting with the current resolution,
2. try out larger resolution,
3. training the current model on different fields,
4. more than one output channel.

Let us now go over each point in more detail. First, one may look for ways to combat this slight overfitting. Possible solution could be running the model for less epochs as we are

running it for 400 epochs. Another way would be introducing Batch Normalization layers in the generator (reminder: there are currently no Batch Normalization layers in the generator, only in the discriminator, which has less parameters) or Dropout layers in the generator and/or in the discriminator.

Currently, the resolution of the images is quite low (only $N_{side} = 64$). Just for a comparison, Petroff et al. [2020] is using $N_{side} = 512$. The reason for the low resolution has been the run time of the model with larger resolutions and the tight schedule for this thesis. It would be interesting to see whether the results would improve or worsen significantly. Furthermore, it would also be of great interest to see if it would have any effect on the current slight overfitting.

In this thesis, we have presented only the results for the density field. In the future, it would be of great interest to train this model fully analogously on different fields such as temperature or metallicity. Moreover, it would be fascinating to see how the results for different fields compare with the results presented here.

Last but certainly not least, since we have already mentioned training the model on different fields, maybe a more elegant solution would be having a single model with more output channels, for instance [density, metallicity, temperature]. We have tried already this variant in the course of this thesis, however the single output model has proven to be superior, hence this type of models would require more research than it was dedicated here. Something similar has already been done by Tamosiunas et al. [2021] who have trained a GAN with three output channels, namely for the dark matter, gas and internal energy distribution.

All in all, it can be concluded that applying a conditional GAN on a sphere to galaxy positions to infer the structure of the cosmic web, which encompasses the intergalactic gas and dark matter distribution, is a viable option for studying the large-scale structure of the universe.

Bibliography

- Tero Karras, Samuli Laine, and Timo Aila. A style-based generator architecture for generative adversarial networks. *CoRR*, abs/1812.04948, 2018. URL <http://arxiv.org/abs/1812.04948>.
- Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A. Efros. Image-to-image translation with conditional adversarial networks. *CoRR*, abs/1611.07004, 2016. URL <http://arxiv.org/abs/1611.07004>.
- M.V. Valueva, N.N. Nagornov, P.A. Lyakhov, G.V. Valuev, and N.I. Chervyakov. Application of the residue number system to reduce hardware costs of the convolutional neural network implementation. *Mathematics and Computers in Simulation*, 177:232–243, 2020. ISSN 0378-4754. doi: <https://doi.org/10.1016/j.matcom.2020.04.031>. URL <https://www.sciencedirect.com/science/article/pii/S0378475420301580>.
- K. M. Gorski, E. Hivon, A. J. Banday, B. D. Wandelt, F. K. Hansen, M. Reinecke, and M. Bartelmann. HEALPix: A framework for high-resolution discretization and fast analysis of data distributed on the sphere. *The Astrophysical Journal*, 622(2):759–771, apr 2005. doi: 10.1086/427976. URL <https://doi.org/10.1086%2F427976>.
- N. Perraudin, M. Defferrard, T. Kacprzak, and R. Sgier. DeepSphere: Efficient spherical convolutional neural network with HEALPix sampling for cosmological applications. *Astronomy and Computing*, 27:130–146, apr 2019. doi: 10.1016/j.ascom.2019.03.004. URL <https://doi.org/10.1016%2Fj.ascom.2019.03.004>.
- Yeonkun Lee, Jaeseok Jeong, Jongseob Yun, Wonjune Cho, and Kuk-Jin Yoon. Spherephd: Applying cnns on a spherical polyhedron representation of 360 degree images, 2019.
- N. Krachmalnicoff and M. Tomasi. Convolutional neural networks on the HEALPix sphere: a pixel-based algorithm and its application to CMB data analysis. *Astronomy & Astrophysics*, 628:A129, aug 2019. doi: 10.1051/0004-6361/201935211. URL <https://doi.org/10.1051%2F0004-6361%2F201935211>.
- Navid Mahmoudian Bidgoli, Roberto G. de A. Azevedo, Thomas Maugey, Aline Roumy, and Pascal Frossard. OSLO: On-the-sphere learning for omnidirectional images and its application to 360-degree image compression. *IEEE Transactions on Image Processing*, 31: 5813–5827, 2022. doi: 10.1109/tip.2022.3202357. URL <https://doi.org/10.1109%2Ftip.2022.3202357>.
- Taco S. Cohen, Maurice Weiler, Berkay Kicanaoglu, and Max Welling. Gauge equivariant convolutional networks and the icosahedral cnn, 2019.
- Noam Rozen, Aditya Grover, Maximilian Nickel, and Yaron Lipman. Moser flow: Divergence-based generative modeling on manifolds, 2021. URL <https://arxiv.org/abs/2108.08052>.

- Francisco Villaescusa-Navarro, Daniel Anglés-Alcázar, Shy Genel, David N. Spergel, Rachel S. Somerville, Romeel Dave, Annalisa Pillepich, Lars Hernquist, Dylan Nelson, Paul Torrey, Desika Narayanan, Yin Li, Oliver Philcox, Valentina La Torre, Ana Maria Delgado, Shirley Ho, Sultan Hassan, Blakesley Burkhart, Digvijay Wadekar, Nicholas Battaglia, Gabriella Contardo, and Greg L. Bryan. The CAMELS project: Cosmology and astrophysics with machine-learning simulations. *The Astrophysical Journal*, 915(1):71, jul 2021. doi: 10.3847/1538-4357/abf7ba. URL <https://doi.org/10.3847%2F1538-4357%2Fabf7ba>.
- Youcai Zhang, Xiaohu Yang, and Hong Guo. Connections between galaxy properties and halo formation time in the cosmic web. , 507(4):5320–5330, November 2021. doi: 10.1093/mnras/stab2487.
- Benjamin Zhang, Khee-Gan Lee, Alex Krolewski, Jingjing Shi, Benjamin Horowitz, and Robin Kooistra. Intrinsic Alignments Between Galaxies and the Cosmic Web at $z \sim 1 - 2$ in the IllustrisTNG Simulations. *arXiv e-prints*, art. arXiv:2211.09331, November 2022. doi: 10.48550/arXiv.2211.09331.
- Farhanul Hasan, Joseph N. Burchett, Alyssa Abeyta, Douglas Hellinger, Nir Mandelker, Joel R. Primack, S. M. Faber, David C. Koo, Oskar Elek, and Daisuke Nagai. The Evolving Effect Of Cosmic Web Environment On Galaxy Quenching. *arXiv e-prints*, art. arXiv:2303.08088, March 2023. doi: 10.48550/arXiv.2303.08088.
- Virginia Trimble. Existence and nature of dark matter in the universe. , 25:425–472, January 1987. doi: 10.1146/annurev.aa.25.090187.002233.
- Sebastian Stapelberg. The cosmic web of galaxies, dark matter and how it emerged, Dec 2022. URL https://structures.uni-heidelberg.de/blog/posts/2022_12_cw/index.php.
- Thomas Jarrett. Large scale structure in the local universe — the 2mass galaxy catalog. *Publications of the Astronomical Society of Australia*, 21(4):396–403, 2004. doi: 10.1071/as04050. URL <https://doi.org/10.1071%2Fas04050>.
- Thanu Padmanabhan. *Structure Formation in the Universe*. Cambridge University Press, 1993. ISBN 9780521424868.
- John A. Peacock. *Cosmological Physics*. Cambridge University Press, 1999. ISBN 9780511804533.
- Scott Dodelson. *Modern Cosmology*. Academic Press, 2003. ISBN 9780122191411.
- Malcolm S. Longair. *Galaxy Formation*. Springer, 2008. ISBN 978-3-540-73478-9.
- Dylan Nelson, Volker Springel, Annalisa Pillepich, Vicente Rodriguez-Gomez, Paul Torrey, Shy Genel, Mark Vogelsberger, Ruediger Pakmor, Federico Marinacci, Rainer Weinberger, Luke Kelley, Mark Lovell, Benedikt Diemer, and Lars Hernquist. The IllustrisTNG simulations: public data release. *Computational Astrophysics and Cosmology*, 6(1):2, May 2019. doi: 10.1186/s40668-019-0028-x.
- Usama Fayyad, Gregory Piatetsky-Shapiro, and Padhraic Smyth. Knowledge discovery and data mining: Towards a unifying framework. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, page 82–88. AAAI Press, 1996.
- Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks, 2015.

- Zhe Ming Chng. Using activation functions in neural networks, Aug 2022. URL <https://machinelearningmastery.com/using-activation-functions-in-neural-networks/>.
- Sebastian Tschiatschek. Lecture notes for the lecture: Introduction to machine learning, 2022.
- Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017. URL <https://arxiv.org/abs/1412.6980>.
- Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, jan 2014. ISSN 1532-4435.
- Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.
- Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 2016.
- AssemblyAI. What is layer normalization? — deep learning fundamentals, 2022. URL <https://www.youtube.com/watch?v=2V3Uduw1zwQ>.
- Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.
- Andrea Zonca, Leo Singer, Daniel Lenz, Martin Reinecke, Cyrille Rosset, Eric Hivon, and Krzysztof Gorski. healpy: equal area pixelization and spherical harmonics transforms for data on the sphere in python. *Journal of Open Source Software*, 4(35):1298, March 2019. doi: 10.21105/joss.01298. URL <https://doi.org/10.21105/joss.01298>.
- Florian List. Chasing hot stuff in the universe with cool stuff on earth: Numerical and deep learning methods for dark matter annihilation, 2021. URL <https://ses.library.usyd.edu.au/handle/2123/26583>.
- Mehdi Mirza and Simon Osindero. Conditional generative adversarial nets, 2014.
- Ian Goodfellow. Nips 2016 tutorial: Generative adversarial networks, 2017.
- Florian List. Lecture notes for the lecture: Wm-c-dat data science in the astrophysics (pi), 2022.
- Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. pix2pix: Image-to-image translation with a conditional gan : Tensorflow core, Dec 2022. URL <https://www.tensorflow.org/tutorials/generative/pix2pix>. Tutorial on pix2pix.
- Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation, 2015.
- Bhumika Dutta. What are skip connections in neural networks?, Sep 2021. URL <https://www.analyticssteps.com/blogs/what-are-skip-connections-neural-networks>.

- Filip Sabo. GANs on spheres, June 2023. URL https://github.com/fsabo/GANs_on_spheres.
- Augustus Odena, Vincent Dumoulin, and Chris Olah. Deconvolution and checkerboard artifacts, Oct 2016. URL <https://distill.pub/2016/deconv-checkerboard/>.
- Matthew A. Petroff, Graeme E. Addison, Charles L. Bennett, and Janet L. Weiland. Full-sky cosmic microwave background foreground cleaning using machine learning. *The Astrophysical Journal*, 903(2):104, nov 2020. doi: 10.3847/1538-4357/abb9a7. URL <https://doi.org/10.3847%2F1538-4357%2Fabb9a7>.
- Jason Brownlee. How to develop a pix2pix gan for image-to-image translation, January 2021. URL <https://machinelearningmastery.com/how-to-develop-a-pix2pix-gan-for-image-to-image-translation/>.
- pawangfg. Image-to-image translation using pix2pix, June 2022. URL <https://www.geeksforgeeks.org/image-to-image-translation-using-pix2pix/>.
- Nathanaël Perraudin, Sandro Marcon, Aurelien Lucchi, and Tomasz Kacprzak. Emulation of cosmological mass maps with conditional generative adversarial networks, 2021.
- Florian List, Ishaan Bhat, and Geraint F Lewis. A black box for dark sector physics: predicting dark matter annihilation feedback with conditional GANs. *Monthly Notices of the Royal Astronomical Society*, 490(3):3134–3143, oct 2019. doi: 10.1093/mnras/stz2759. URL <https://doi.org/10.1093%2Fmnras%2Fstz2759>.
- Dong Zhao, Long Xu, Linjie Chen, Yihua Yan, and Ling-Yu Duan. Mask-pix2pix network for overexposure region recovery of solar image. *Advances in Astronomy*, 2019:5343254, Sep 2019. ISSN 1687-7969. doi: 10.1155/2019/5343254. URL <https://doi.org/10.1155/2019/5343254>.
- Tilman Tröster, Cameron Ferguson, Joachim Harnois-Déraps, and Ian G McCarthy. Painting with baryons: augmenting n-body simulations with gas using deep generative models. *Monthly Notices of the Royal Astronomical Society: Letters*, 487(1):L24–L29, may 2019. doi: 10.1093/mnrasl/slz075. URL <https://doi.org/10.1093%2Fmnrasl%2Fslz075>.
- Andres C. Rodríguez, Tomasz Kacprzak, Aurelien Lucchi, Adam Amara, Raphaël Sgier, Janis Fluri, Thomas Hofmann, and Alexandre Réfrégier. Fast cosmic web simulations with generative adversarial networks. *Computational Astrophysics and Cosmology*, 5(1), nov 2018. doi: 10.1186/s40668-018-0026-4. URL <https://doi.org/10.1186%2Fs40668-018-0026-4>.
- Marion Ullmo, Aurélien Decelle, and Nabila Aghanim. Encoding large-scale cosmological structure with generative adversarial networks. *Astronomy & Astrophysics*, 651:A46, jul 2021. doi: 10.1051/0004-6361/202039866. URL <https://doi.org/10.1051%2F0004-6361%2F202039866>.
- Yin Li, Yueying Ni, Rupert A. C. Croft, Tiziana Di Matteo, Simeon Bird, and Yu Feng. AI-assisted superresolution cosmological simulations. *Proceedings of the National Academy of Sciences*, 118(19), may 2021. doi: 10.1073/pnas.2022038118. URL <https://doi.org/10.1073%2Fpnas.2022038118>.
- Andrius Tamosiunas, Hans A Winther, Kazuya Koyama, David J Bacon, Robert C Nichol, and Ben Mawdsley. Investigating cosmological GAN emulators using latent space interpolation. *Monthly Notices of the Royal Astronomical Society*, 506(2):3049–3067, jul 2021. doi: 10.1093/mnras/stab1879. URL <https://doi.org/10.1093%2Fmnras%2Fstab1879>.

Chapter 7

Appendix

7.1 Creation of training and test data sets

```
import numpy as np
import h5py
import healpy as hp
import tqdm
import pandas as pd
from numba import njit
from scipy.interpolate import NearestNDInterpolator
#%%
def file_to_pandas(name):
    data=[]
    with open(name,'r') as f:
        my_list = f.read()
        for line in my_list.splitlines():
            data.append(line.split())
    return pd.DataFrame(np.array(data[64:]).astype(float),columns=data[0
    ])
#%%
PATH="../../../mnt/data/filipsabo"
n_files=16
m_p=1.6726*10**(-24) # proton mass
k_B=1.3807*10**(-16) # Boltzmann constant
BoxSizes=[]
pos_gs=[]
rho_gs=[]
interp_rhos=[]
galaxies_datasets=[]
metal_gs=[]
energies=[]
electrons=[]
temp_models=[]
metal_models=[]
print("importing files...")
for i in tqdm.tqdm(range(n_files)):
    with h5py.File(PATH+'snap/snap_033_{}.hdf5'.format(i)) as f:
        BoxSize = f['Header'].attrs[u'BoxSize']/1e3 # Mpc/h
        pos_g = f['PartType0/Coordinates'][:]/1e3 # Mpc/h
        pos_g = pos_g.astype(np.float32) # positions as float32
        pos_g = np.fmod(pos_g,BoxSize) # periodically wrap
        rho_g = f['PartType0/Density'][:]/1e10 # 1e10 h-1 Msun / ( h-1 ckpc)^3
        metal_g = f['PartType0/GFM_Metallicity'][:]/
        energy = f['PartType0/InternalEnergy'][:]/
        electron = f['PartType0/ElectronAbundance'][:]/
```

```

BoxSizes.append(BoxSize)
pos_gs.append(pos_g)
rho_gs.append(rho_g)
metal_gs.append(metal_g)
energies.append(energy)
electrons.append(electron)
interp_rhos.append(NearestNDInterpolator(pos_gs[-1], rho_gs[-1],
                                         tree_options={'boxsize' :
                                                         BoxSizes[-1]}))
temp_models.append(NearestNDInterpolator(pos_gs[-1], 2/3*4/(1+3*0.76+
4*0.76*electrons[-1])*m_p*
energies[-1]/k_B*10**10,
                                         tree_options={'boxsize' :
                                                         BoxSizes[-1]}))
metal_models.append(NearestNDInterpolator(pos_gs[-1], metal_gs[-1],
                                         tree_options={'boxsize' :
                                                         BoxSizes[-1]}))
galaxies_datasets.append(file_to_pandas(PATH+'hlist/hlist_{i}_1.
00000.list'.format(i)))

###
nside = 64
npix = hp.nside2npix(nside)
indices=np.arange(hp.nside2npix(nside))
###
theta, phi = hp.pix2ang(nside, np.arange(npix), nest=True)
theta_phi = np.stack((theta, phi), -1)
pi_minus_theta = np.pi - theta
pi_minus_theta_phi = np.stack((pi_minus_theta, phi), -1)
two_pi_minus_phi = (2*np.pi - phi)%(2*np.pi)
two_pi_minus_theta_phi = np.stack((theta, two_pi_minus_phi), -1)
###
@njit
def build_inds(minus_array, th_ph):
    inds_i = []
    inds_j = []
    for i, p1 in enumerate(minus_array):
        for j, p2 in enumerate(th_ph):
            if np.sum(np.abs(p1 - p2)) < 1e-10:
                inds_i.append(i)
                inds_j.append(j)
    return inds_i, inds_j
###
print("theta")
inds_i_1, inds_j_1 = build_inds(pi_minus_theta_phi, theta_phi)
print("phi")
inds_i_2, inds_j_2 = build_inds(two_pi_minus_theta_phi, theta_phi)
###
def gen_query_points( x0, r, npix ):
    theta, phi = hp.pix2ang(nside, range(npix))
    pos_healpy = np.zeros((npix, 3))
    pos_healpy[:,0] = x0[0] + r * np.sin(theta) * np.cos(phi)
    pos_healpy[:,1] = x0[1] + r * np.sin(theta) * np.sin(phi)
    pos_healpy[:,2] = x0[2] + r * np.cos(theta)
    return pos_healpy
###
def gen_avg_maps(x0, rmin, rmax, nsteps, interp):
    rr = np.linspace(rmin, rmax, nsteps)
    rho_img = np.zeros(hp.nside2npix(nside))
    for r in rr:
        pos_healpy = gen_query_points( x0=x0, r=r, npix=npix)
        rho_slice = interp( pos_healpy )
        rho_img += rho_slice
    return rho_img/nsteps

```

```

%%
def get_spherical(galaxy, x0, size):
    gal_pos = galaxy[["x(17)", "y(18)", "z(19)"]].to_numpy()

    rgal_pos = gal_pos - x0
    rgal_pos[rgal_pos < -size/2] += size
    rgal_pos[rgal_pos > size/2] -= size

    r_gal = np.sqrt((rgal_pos**2).sum(axis=1))
    theta_gal = np.arccos(rgal_pos[:, 2]/r_gal)
    phi_gal = np.arctan2(rgal_pos[:, 1], rgal_pos[:, 0])

    return np.c_[r_gal, theta_gal, phi_gal]

%%
def get_counts(theta, phi):
    # convert to HEALPix indices
    indices = hp.ang2pix(nside, theta, phi)

    idx, counts = np.unique(indices, return_counts=True)

    # fill the fullsky map
    hpx_map = np.zeros(npix, dtype=int)
    hpx_map[idx] = counts
    return hpx_map

%%
def perform_cuts(x0, galaxy, gas, sizes, augmentation=True):
    galaxy_array = []
    gas_array = []
    for i in tqdm.tqdm(range(len(galaxy))):
        for j in range(x0.shape[0]):
            galaxy_positions_spherical = get_spherical(galaxy[i], x0[j, :],
                                                       sizes[i])

            ii = (galaxy_positions_spherical[:, 0] >= r_min) & (
                galaxy_positions_spherical[:, 0] <= r_max)
            galaxy_array.append(hp.pixelfunc.reorder(get_counts(
                galaxy_positions_spherical[ii, 1],
                galaxy_positions_spherical[ii, 2]), r2n=True))

            gas_array.append(hp.pixelfunc.reorder(gen_avg_maps(x0[j],
                                                                r_min, r_max, 20, gas[i]
                                                                ), r2n=True))

    galaxy_array = np.array(galaxy_array)
    gas_array = np.array(gas_array)
    if augmentation:
        for i in tqdm.tqdm(range(galaxy_array.shape[0])):
            img = galaxy_array[i, :]
            galaxy_array = np.r_[galaxy_array, img[inds_j_1][None, :]]
            galaxy_array = np.r_[galaxy_array, img[inds_j_2][None, :]]
            galaxy_array = np.r_[galaxy_array, img[inds_j_1][inds_j_2][
                None, :]]

            img_1 = gas_array[i, :]
            gas_array = np.r_[gas_array, img_1[inds_j_1][None, :]]
            gas_array = np.r_[gas_array, img_1[inds_j_2][None, :]]
            gas_array = np.r_[gas_array, img_1[inds_j_1][inds_j_2][None, :]]

    return galaxy_array[:, :, None].astype(np.float32), gas_array[:, :, None]
        .astype(np.float32)

```

```

%%
x_grid=np.arange(0,25,5)
x0s=np.array(np.meshgrid(x_grid,x_grid,x_grid)).T.reshape(-1,3)
r_min=8
r_max=10

print("performing cuts for the training set...")
train_set=perform_cuts(x0s,galaxies_datasets[:-2],interp_rhos[:-2],
                        BoxSizes[:-2])
np.save('train_set_64.npy', train_set) # save
print("performing cuts for the test set...")
test_set=perform_cuts(x0s,galaxies_datasets[-2:],interp_rhos[-2:],
                      BoxSizes[-2:],False)
np.save('test_set_64.npy', test_set) # save
print("performing cuts for the training set...")
train_set=perform_cuts(x0s,galaxies_datasets[:-2],temp_models[:-2],
                        BoxSizes[:-2])
np.save('train_set_64_temp.npy', train_set) # save
print("performing cuts for the test set...")
test_set=perform_cuts(x0s,galaxies_datasets[-2:],temp_models[-2:],
                      BoxSizes[-2:],False)
np.save('test_set_64_temp.npy', test_set) # save
train_set=perform_cuts(x0s,galaxies_datasets[:-2],metal_models[:-2],
                        BoxSizes[:-2])
np.save('train_set_64_metal.npy', train_set) # save
print("performing cuts for the test set...")
test_set=perform_cuts(x0s,galaxies_datasets[-2:],metal_models[-2:],
                      BoxSizes[-2:],False)
np.save('test_set_64_metal.npy', test_set) # save

```

7.2 Normalization of the data sets

```

import numpy as np
train_set_1=np.load('data/train_set_64.npy')
test_set_1=np.load('data/test_set_64.npy')
train_set_2=np.load('data/train_set_64_metal.npy')
test_set_2=np.load('data/test_set_64_metal.npy')
train_set_3=np.load('data/train_set_64_temp.npy')
test_set_3=np.load('data/test_set_64_temp.npy')
def norm_data(train,test):
    data=np.concatenate((train,test),axis=1)
    galaxy_array=np.interp(data[0], (np.array(data[0]).min(), np.array(
        data[0]).max()), (0, +1))

    gas_array = np.log10(data[1])
    gas_array=np.interp(gas_array, (gas_array.min(), gas_array.max()), (
        0, +1))

    return np.array([galaxy_array[:-250],gas_array[:-250]]),np.array([
        galaxy_array[-250:],gas_array[-
        250:]])

def merge_sets(set1,set2,set3):
    return np.concatenate((set1[0],set1[1],set2[1],set3[1]),axis=2)
train_set_1,test_set_1=norm_data(train_set_1,test_set_1)
train_set_2,test_set_2=norm_data(train_set_2,test_set_2)
train_set_3,test_set_3=norm_data(train_set_3,test_set_3)
data=merge_sets(train_set_1,train_set_2,train_set_3)
print(data.shape)
np.save('train_set.npy', data) # save
data=merge_sets(test_set_1,test_set_2,test_set_3)
print(data.shape)
np.save('test_set.npy', data) # save

```

7.3 Primary architectures

Layer	Output shape
Input layer 1	(None,49152,1)
Input layer 2	(None,49152,1)
Concatenate	(None,49152,2)
Reshape	(None,49152,2)
Sphere Convolution	(None,49152,16)
Batch Normalization	(None,49152,16)
Leaky ReLu	(None,49152,16)
Graph Pooling	(None,12288,16)
Sphere Convolution	(None,12288,32)
Batch Normalization	(None,12288,32)
Leaky ReLu	(None,12288,32)
Graph Pooling	(None,3072,32)
Sphere Convolution	(None,3072,64)
Batch Normalization	(None,3072,64)
Leaky ReLu	(None,3072,64)
Graph Pooling	(None,768,64)
Sphere Convolution	(None,768,128)
Batch Normalization	(None,768,128)
Leaky ReLu	(None,768,128)
Graph Pooling	(None,192,128)
Sphere Convolution	(None,192,256)
Batch Normalization	(None,192,256)
Leaky ReLu	(None,192,256)
Graph Pooling	(None,48,256)
Sphere Convolution	(None,48,512)
Batch Normalization	(None,48,512)
Leaky ReLu	(None,48,512)
Sphere Convolution	(None,48,1)
Sigmoid Activation	(None,48,1)
Reduce Mean	(None,1)

Table 7.1: Discriminator's layers

Layer	Output shape
Input Layer	(None,49152,1))
Reshape	(None,49152,1)
Sphere Convolution	(None,49152,16)
Leaky ReLu	(None,49152,16)
Graph Pooling	(None,12288,16)
Sphere Convolution	(None,12288,32)
Leaky ReLu	(None,12288,32)
Graph Pooling	(None,3072,32)
Sphere Convolution	(None,3072,64)
Leaky ReLu	(None,3072,64)
Graph Pooling	(None,768,64)
Sphere Convolution	(None,768,128)
Leaky ReLu	(None,768,128)
Graph Pooling	(None,192,128)
Sphere Convolution	(None,192,256)
Leaky ReLu	(None,192,256)
Graph Pooling	(None,48,256)
Sphere Convolution	(None,48,512)
Leaky ReLu	(None,48,512)
Linear Combination	(None,48,512)
Add	(None,48,512)
UpSampling1D	(None,192,512)
ReLu	(None,192,512)
Sphere Convolution	(None,192,256)
Linear Combination	(None,192,256)
Concatenate	(None,192,512)
UpSampling1D	(None,768,512)
ReLu	(None,768,512)
Sphere Convolution	(None,768,128)
Linear Combination	(None,768,128)
Concatenate	(None,768,256)
UpSampling1D	(None,3072,256)
ReLu	(None,3072,256)
Sphere Convolution	(None,3072,64)
Linear Combination	(None,3072,64)
Concatenate	(None,3072,128)
UpSampling1D	(None,12288,128)
ReLu	(None,12288,128)
Sphere Convolution	(None,12288,32)
Linear Combination	(None,12288,32)
Concatenate	(None,12288,64)
UpSampling1D	(None,49152,64)
ReLu	(None,49152,64)
Sphere Convolution	(None,49152,16)
Linear Combination	(None,49152,16)
Concatenate	(None,49152,32)
ReLu	(None,49152,32)
Sphere Convolution	(None,49152,1)
Sigmoid activation	(None,49152,1)

Table 7.2: Generator's layers

7.4 Finding a model - best model

```
import numpy as np
import healpy as hp
import tqdm
import matplotlib.pyplot as plt
from tensorflow.keras import layers, models, optimizers, initializers,
                                losses
#from deepsphere import healpy_layers as hp_layer
import tensorflow as tf
import scnn.layers
import scnn.dropout
#%%
physical_devices=tf.config.list_physical_devices("GPU")
print("Num GPUs:", len(physical_devices))
#%%
print("load in the data")
train_set=np.load('../3_channels/train_set.npy').astype(np.float32)[...,
                                                                    [0,1]]
test_set=np.load('../3_channels/test_set.npy').astype(np.float32)[..., [0
                                                                    ,1]]

print(train_set.shape)
print(test_set.shape)
#%%
nside = 64
healpix_img = np.zeros(hp.nside2npix(nside))
npix = len(healpix_img)
#%%
LAMBDA = 100
loss_object = losses.BinaryCrossentropy()
def generator_loss(disc_generated_output, gen_output, target):
    gan_loss = loss_object(tf.ones_like(disc_generated_output),
                             disc_generated_output)

    # Mean absolute error
    l1_loss = tf.reduce_mean(tf.abs(target - gen_output))

    total_gen_loss = gan_loss + (LAMBDA * l1_loss)
    return total_gen_loss, l1_loss, gan_loss
#%%
def discriminator_loss(disc_real_output, disc_generated_output):
    real_loss = loss_object(tf.ones_like(disc_real_output),
                             disc_real_output)

    generated_loss = loss_object(tf.zeros_like(disc_generated_output),
                             disc_generated_output)

    total_disc_loss = real_loss + generated_loss

    return total_disc_loss
#%%
def define_discriminator(image_shape, output_image):
    # weight initialization
    # source image input
    in_src_image = layers.Input(shape=image_shape)
    # target image input
    in_target_image = layers.Input(shape=output_image)
    # concatenate images channel-wise
    merged = layers.Concatenate()([in_src_image, in_target_image])

    downscale = []
    # Construct inputs, one per input frequency
```

```

# Construct coarsening / pooling layers, separate for each frequency
for i in range(num_downscales):
    #downscale.append([])
    if i != 0:
        # Pool all but first layer
        downscale.append(scnn.layers.GraphPool(4)(downscale[-1]))
    else:
        downscale.append(tf.keras.layers.Reshape((hp.nside2npix(
                                                    nside), 2))(merged))
    downscale.append(scnn.layers.SphereConvolution(
        filter_size=nsides_up[i], poly_k=poly_k[i], nside=nsides
        [i]
    )(downscale[-1]))
    downscale.append(layers.BatchNormalization()(downscale[-1]))
    downscale.append(layers.LeakyReLU(alpha=0.2)(downscale[-1]))
    downscale.append(scnn.layers.SphereConvolution(
        filter_size=nsides_up[i], poly_k=poly_k[i], nside=nsides
        [i]
    )(downscale[-1]))
    downscale.append(layers.BatchNormalization()(downscale[-1]))
    downscale.append(layers.LeakyReLU(alpha=0.2)(downscale[-1]))

d=scnn.layers.SphereConvolution(
    filter_size=1, poly_k=poly_k[-1], nside=2
)(downscale[-1])
d=layers.Activation('sigmoid')(d)
d=tf.math.reduce_mean(tf.math.reduce_mean(d,axis=2),axis=1)
print("d shape:{}".format(d.shape))

# define model
model = models.Model([in_src_image, in_target_image], d)
return model

#%%
# define the standalone generator model
def define_generator(image_shape):
    # weight initialization
    #init = initializers.RandomNormal(stddev=0.02)
    # image input
    in_image = layers.Input(shape=image_shape)
    downscale = []
    # Construct inputs, one per input frequency
    # Construct coarsening / pooling layers, separate for each frequency
    for i in range(num_downscales):
        #downscale.append([])
        if i != 0:
            # Pool all but first layer
            downscale.append(scnn.layers.GraphPool(4)(downscale[-1]))
        else:
            downscale.append(tf.keras.layers.Reshape((hp.nside2npix(
                                                        nside), 1))(in_image))
        downscale.append(scnn.layers.SphereConvolution(
            filter_size=nsides_up[i], poly_k=poly_k[i], nside=nsides
            [i]
        )(downscale[-1]))
        downscale.append(layers.LeakyReLU(alpha=0.2)(downscale[-1]))
        downscale.append(scnn.layers.SphereConvolution(
            filter_size=nsides_up[i], poly_k=poly_k[i], nside=nsides
            [i]
        )(downscale[-1]))
        downscale.append(layers.LeakyReLU(alpha=0.2)(downscale[-1]))
    # Construct scaling / biasing for adding to upscaled layers,
    # separate for each frequency
    for i in range(len(downscale)):

```

```

        downscale[i]=scnn.layers.LinearCombination()(downscale[i])
# Add separate fully-coarsened frequency layers into a single layer
# for upscaling
upscale = tf.keras.layers.add([downscale[-1],downscale[-1]])/2
#upscale = layers.LeakyReLU(alpha=0.2)(upscale)
upscale = tf.keras.layers.UpSampling1D(4)(upscale)
upscale = tf.keras.layers.Activation("relu")(upscale)
# Construct upscaling / skip connection layers
for i in range(1,num_downscales):
    upscale = scnn.layers.SphereConvolution(
        filter_size=nsides_up[num_downscales + i],
        poly_k=poly_k[num_downscales + i],
        nside=nsides[num_downscales + i],
    )(upscale)
    upscale = layers.Concatenate()([upscale, downscale[-1 - 5*i]])
    # Skip connection

    if i < num_downscales - 1:
        # Upsample all but last layer
        upscale = tf.keras.layers.UpSampling1D(4)(upscale)
        upscale = tf.keras.layers.Activation("relu")(upscale)

# Final convolution layer
upscale = scnn.layers.SphereConvolution(
    filter_size=nsides_up[-1], poly_k=poly_k[-1], nside=nsides[-1]
)(upscale)
upscale = layers.Activation('sigmoid')(upscale)

return tf.keras.models.Model(inputs=in_image, outputs=upscale)
%%
# define the combined generator and discriminator model, for updating
# the generator
def define_gan(g_model, d_model, image_shape):
    # make weights in the discriminator not trainable
    for layer in d_model.layers:
        if not isinstance(layer, layers.BatchNormalization):
            layer.trainable = False
    # define the source image
    in_src = layers.Input(shape=image_shape)
    # connect the source image to the generator input
    gen_out = g_model(in_src)
    # connect the source input and generator output to the discriminator
    # input
    dis_out = d_model([in_src, gen_out])
    # src image as input, generated image and classification output
    model = models.Model(in_src, [dis_out, gen_out])
    return model
%%
print("optimizers")
generator_optimizer = tf.keras.optimizers.Adam(2e-4, beta_1=0.5)
discriminator_optimizer = tf.keras.optimizers.Adam(2e-4, beta_1=0.5)
@tf.function
#@njit(parallel=True)
def train_step(input_image, target):
    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        gen_output = g_model(input_image,training=True)
        disc_real_output = d_model([input_image, target],training=True)
        disc_generated_output = d_model([input_image, gen_output],
                                         training=True)

        gen_total_loss,l1_l,gan_l = generator_loss(disc_generated_output
                                                    , gen_output, target)
        disc_loss = discriminator_loss(disc_real_output,
                                       disc_generated_output)

```

```

generator_gradients = gen_tape.gradient(gen_total_loss,g_model.trainable_variables)
discriminator_gradients = disc_tape.gradient(disc_loss,d_model.trainable_variables)
generator_optimizer.apply_gradients(zip(generator_gradients,
                                         g_model.trainable_variables)

discriminator_optimizer.apply_gradients(zip(discriminator_gradients,
                                           d_model.

    return gen_total_loss, disc_loss
###
# train pix2pix model
#@tf.function
def train(d_model, g_model, gan_model, dataset,n_epochs=10, n_batch=1):
    # unpack dataset
    trainA, trainB = dataset[...,0][:,:,None],dataset[...,1:]
    # calculate the number of batches per training epoch
    bat_per_epo = int(len(trainA) / n_batch)
    # calculate the number of training iterations
    n_steps = bat_per_epo * n_epochs
    # manually enumerate epochs
    disc_loss=[]
    gen_loss=[]
    for i in tqdm.tqdm(range(n_steps)):
        ix = np.random.randint(0, trainA.shape[0], n_batch)
        # retrieve selected ima
        X1, X2 = trainA[ix], trainB[ix]
        g1,g2 = train_step(X1, X2)
        gen_loss.append(g1)
        disc_loss.append(g2)
        if i%(5*bat_per_epo)==0 and i!=0:
            g_model.save("model_1_epochs_{}.h5".format(i/bat_per_epo))
            np.save("g_loss_1.npy",gen_loss)
            np.save("d_loss_1.npy",disc_loss)
    return disc_loss,gen_loss
###
# define input shape based on the loaded dataset
image_shape = (npix,1)
output_image = (npix,1)
nsides = []
nsides_up = []
i = nside
j=16
while i >= 2:
    nsides.append(i)
    nsides_up.append(j)
    i = i // 2
    j=j*2
#nsides+=[1]
#nsides_up+=[j]
nsides += reversed(nsides)
nsides_up += reversed(nsides_up)
nsides_up+=[1]

print(nsides)
print(nsides_up)

FILTER_MAPS = 6

```

```

POLY_ORDER = 5
LENGTH_SCALE = 1e-4

# Construct filter maps and Chebyshev polynomial order lists
filters = [FILTER_MAPS] * len(nsides) + [1] # Result + uncertainty
                                              outputs
poly_k = [POLY_ORDER] * len(nsides)

n = train_set.shape[1]
wd = LENGTH_SCALE ** 2.0 / n
dd = 2.0 / n

num_downscales = len(nsides) // 2
input_channels=1
print(input_channels)
# define the model
print("discriminator")
d_model = define_discriminator(image_shape, output_image)
print("generator")
g_model = define_generator(image_shape)
print("Discriminator:")
print(d_model.summary())
print("Generator:")
print(g_model.summary())
gan_model = define_gan(g_model, d_model, image_shape)
# train model
print("training the model...")
n_epochs=200
print(train_set.shape)
print(test_set.shape)
d_loss, g_loss=train(d_model, g_model, gan_model, train_set, n_epochs)
#%%
# generate image from source
g_model.save("model_1_final.h5")
np.save("d_loss_1_final.npy", d_loss)
np.save("g_loss_1_final.npy", g_loss)

```

7.5 Plotting of results

```

import numpy as np
import healpy as hp
import tqdm
import matplotlib.pyplot as plt
import tensorflow as tf
import scnn.layers
import scnn.dropout
train_set=np.load('3_channels/train_set.npy').astype(np.float32)[..., [0,
1]]
test_set=np.load('3_channels/test_set.npy').astype(np.float32)[..., [0,1]
]
print(test_set.shape)
model_1= tf.keras.models.load_model('density/model_1_final.h5',
                                     custom_objects={'SphereConvolution':
scnn.layers.SphereConvolution, '
GraphPool':scnn.layers.GraphPool, '
LinearCombination':scnn.layers.
LinearCombination})
def plot_images(data, model, addition):
    gen_image = model.predict(data[..., 0])

```

```

sets=np.concatenate((data[... ,0][:,:,None],gen_image,data[... ,1:]),
                    axis=2)
titles=['Source','Generated - ','Real - ']
fig,axs=plt.subplots(5,3,figsize=(15,16))
fig.tight_layout()
for i in tqdm.tqdm(range(5)):
    idx=np.random.randint(low=0,high=gen_image.shape[0]-1)
    for j in range(3):
        plt.axes(axs[i%5,j])
        hp.mollview(sets[idx,:,j],cmap='turbo',hold=True,nest=True,
                    title=titles[j],min=0,
                    max=1)

        hp.graticule()
    plt.savefig("points_to_field_{}.pdf".format(addition))
plot_images(train_set,model_1,"train_1")
plot_images(test_set,model_1,"test_1")

```

7.6 Statistical analysis

```

import numpy as np
import healpy as hp
import tqdm
import seaborn as sns
import matplotlib.pyplot as plt
import tensorflow as tf
import scnn.layers
import scnn.dropout
train_set=np.load('3_channels/train_set.npy').astype(np.float32)[...,[0,
1]]
test_set=np.load('3_channels/test_set.npy').astype(np.float32)[...,[0,1]
]
model_1= tf.keras.models.load_model('density/model_1_final.h5',
                                     custom_objects={'SphereConvolution':
scnn.layers.SphereConvolution, '
GraphPool':scnn.layers.GraphPool, '
LinearCombination':scnn.layers.
LinearCombination})
titles=['Source','Generated','Real',"Angular power spectrum","Histogramm
"]
ll = np.arange(192)
def plot_images(data,model,name):
    n=5
    gen_image = model.predict(data[... ,0])
    sets=np.concatenate((data[... ,0][:,:,None],gen_image,data[... ,1:]),
                        axis=2)
    fig,axs=plt.subplots(n,5,figsize=(20,20))
    fig.tight_layout()
    for i in tqdm.tqdm(range(n)):
        idx=np.random.randint(low=0,high=data.shape[0]-1)
        for j in range(5):
            if j<3:
                plt.axes(axs[i%n,j])
                hp.mollview(sets[idx,:,j].flatten(),cmap='turbo',hold=
True,nest=True,title
=titles[j],min=0,max
=1)

                hp.graticule()
            else:
                if j==3:

```

```

        axs[i%n,j].plot(l1, hp.anafast(sets[idx,:,1]), color="
                                red", label="
                                generated")
        axs[i%n,j].plot(l1, hp.anafast(sets[idx,:,2]), color="
                                blue", label="
                                real")
        axs[i%n,j].plot(l1, hp.anafast(sets[idx,:,1])/hp.
                                anafast(sets[idx
                                :,2]), color="
                                black", label="
                                generated vs.
                                real")

        axs[i%n,j].legend()
        axs[i%n,j].grid()
        axs[i%n,j].set_yscale("log")
        axs[i%n,j].set_title(titles[j])
    else:
        plt.axes(axs[i%n,j])
        sns.kdeplot(sets[idx,:,1], color="red", label="
                                generated", fill=
                                True)

        sns.kdeplot(sets[idx,:,2], color="blue", label="real",
                                fill=True)

        plt.legend()
        plt.grid()
        plt.xlim([0,1])
        plt.title(titles[j])
    plt.savefig("aps_analysis_{}_model_{}.pdf".format(name[0], name[1]))
plot_images(train_set, model_1, ["training_set", "1"])
plot_images(test_set, model_1, ["test_set", "1"])

```

7.7 Creation of noisy data

```

import numpy as np
import h5py
import healpy as hp
import tqdm
import pandas as pd
from numba import njit
from scipy.interpolate import NearestNDInterpolator
#%%
def file_to_pandas(name):
    data=[]
    with open(name, 'r') as f:
        my_list = f.read()
        for line in my_list.splitlines():
            data.append(line.split())
    return pd.DataFrame(np.array(data[64:]).astype(float), columns=data[0
    ])
#%%
PATH="../../../mnt/data/filipsabo"
n_files=16
BoxSizes=[]
pos_gs=[]
rho_gs=[]
interp_rhos=[]
galaxies_datasets=[]
print("importing files...")
for i in tqdm.tqdm(range(n_files)):
    with h5py.File(PATH+'snap/snap_033_{}.hdf5'.format(i)) as f:

```

```

        BoxSize = f['Header'].attrs[u'BoxSize']/1e3 # Mpc/h
        pos_g = f['PartType0/Coordinates'][:]/1e3 # Mpc/h
        pos_g = pos_g.astype(np.float32) # positions as float32

        pos_g = np.fmod(pos_g, BoxSize) # periodically wrap
        rho_g = f['PartType0/Density'][:]/1e10 # 1e10 h-1 Msun / ( h-1 ckpc)^3

    BoxSizes.append(BoxSize)
    pos_gs.append(pos_g)
    rho_gs.append(rho_g)
    interp_rhos.append(NearestNDInterpolator(pos_gs[-1], rho_gs[-1],
                                             tree_options={'boxsize': BoxSizes[-1]}))

    galaxies_datasets.append(file_to_pandas(PATH+'hlist/hlist_{i}_1.00000.list'.format(i)))

%%
nside = 64
npix = hp.nside2npix(nside)
indices=np.arange(hp.nside2npix(nside))
%%
theta, phi = hp.pix2ang(nside, np.arange(npix), nest=True)
theta_phi = np.stack((theta, phi), -1)
pi_minus_theta = np.pi - theta
pi_minus_theta_phi = np.stack((pi_minus_theta, phi), -1)
two_pi_minus_phi = (2*np.pi - phi)%(2*np.pi)
two_pi_minus_theta_phi = np.stack((theta, two_pi_minus_phi), -1)
%%
@njit
def build_inds(minus_array, th_ph):
    inds_i = []
    inds_j = []
    for i, p1 in enumerate(minus_array):
        for j, p2 in enumerate(th_ph):
            if np.sum(np.abs(p1 - p2)) < 1e-10:
                inds_i.append(i)
                inds_j.append(j)
    return inds_i, inds_j
%%
print("theta")
inds_i_1, inds_j_1 = build_inds(pi_minus_theta_phi, theta_phi)
print("phi")
inds_i_2, inds_j_2 = build_inds(two_pi_minus_theta_phi, theta_phi)
sigma=5*10**(-2)
%%
def gen_query_points( x0, r, npix ):
    theta, phi = hp.pix2ang(nside, range(npix))
    pos_healpy = np.zeros((npix,3))
    pos_healpy[:,0] = x0[0] + r * np.sin(theta) * np.cos(phi)
    pos_healpy[:,1] = x0[1] + r * np.sin(theta) * np.sin(phi)
    pos_healpy[:,2] = x0[2] + r * np.cos(theta)
    return pos_healpy
%%
def gen_avg_maps(x0, rmin, rmax, nsteps, interp):
    rr = np.linspace(rmin, rmax, nsteps)
    rho_img = np.zeros(hp.nside2npix(nside))
    for r in rr:
        pos_healpy = gen_query_points( x0=x0, r=r, npix=npix)
        rho_slice = interp( pos_healpy )
        rho_img += rho_slice
    return rho_img/nsteps
%%
def get_spherical(galaxy, x0, size):
    gal_pos=galaxy[["x(17)", "y(18)", "z(19)"]].to_numpy()

```

```

rgal_pos = gal_pos - x0
rgal_pos[rgal_pos<=-size/2] += size
rgal_pos[rgal_pos>size/2] -= size

r_gal = np.sqrt((rgal_pos**2).sum(axis=1))
n1 = np.random.normal(0,sigma,r_gal.shape[0])
n2 = np.random.normal(0,sigma,r_gal.shape[0])

phi_gal = np.arctan2(rgal_pos[:,1],rgal_pos[:,0])
theta_gal = np.arccos(rgal_pos[:,2]/r_gal)

phi_gal += n1
theta_gal += n2 * np.sin(theta_gal)

iflip = theta_gal < 0
theta_gal[iflip] = -theta_gal[iflip]
#phi_gal[iflip] += np.pi

iflip2 = theta_gal > np.pi
theta_gal[iflip2] = 2*np.pi - theta_gal[iflip2]
#phi_gal[iflip2] += np.pi

return np.c_[r_gal,theta_gal,phi_gal]
#%%
def get_counts(theta,phi):
    # convert to HEALPix indices
    indices = hp.ang2pix(nside,theta,phi)

    idx, counts = np.unique(indices, return_counts=True)

    # fill the fullsky map
    hpx_map = np.zeros(npix, dtype=int)
    hpx_map[idx] = counts
    return hpx_map
#%%
def perform_cuts(x0,galaxy,gas,sizes,augmentation=True):
    galaxy_array=[]
    gas_array=[]
    for i in tqdm.tqdm(range(len(galaxy))):
        for j in range(5):
            galaxy_positions_spherical=get_spherical(galaxy[i],x0[j,:],
                                                    sizes[i])

            ii = (galaxy_positions_spherical[:,0]>=r_min)&(
                    galaxy_positions_spherical
                   [:,0]<=r_max)
            galaxy_array.append(hp.pixelfunc.reorder(get_counts(
                    galaxy_positions_spherical
                    [ii,1],
                    galaxy_positions_spherical
                    [ii,2]),r2n=True))

            gas_array.append(hp.pixelfunc.reorder(gen_avg_maps(x0[j],
                    r_min, r_max, 20, gas[i]
                    ),r2n=True))
    galaxy_array=np.interp(galaxy_array, (np.array(galaxy_array).min(),
                    np.array(galaxy_array).max()), (
                    0, +1))

    gas_array = np.log10(gas_array)
    gas_array=np.interp(gas_array, (gas_array.min(), gas_array.max()), (
                    0, +1))

    if augmentation:

```

```

        for i in tqdm.tqdm(range(galaxy_array.shape[0])):
            img = galaxy_array[i,:]
            galaxy_array = np.r_[galaxy_array, img[inds_j_1][None,:]]
            galaxy_array = np.r_[galaxy_array, img[inds_j_2][None,:]]
            galaxy_array = np.r_[galaxy_array, img[inds_j_1][inds_j_2][
                None,:]]

            img_1 = gas_array[i,:]
            gas_array = np.r_[gas_array, img_1[inds_j_1][None,:]]
            gas_array = np.r_[gas_array, img_1[inds_j_2][None,:]]
            gas_array = np.r_[gas_array, img_1[inds_j_1][inds_j_2][None,:]]

        return galaxy_array[:, :, None].astype(np.float32), gas_array[:, :, None]
            .astype(np.float32)

#%%
x_grid=np.arange(0,25,5)
x0s=np.array(np.meshgrid(x_grid,x_grid,x_grid)).T.reshape(-1,3)
r_min=8
r_max=10

print("performing cuts for the test set...")
N=100
test_set=[]
for i in tqdm.tqdm(range(N)):
    test_set.append(perform_cuts(x0s,galaxies_datasets[-2:],interp_rhos[
        -2:],BoxSizes[-2:],False))
np.save('test_set_64_noise_multiple.npy', test_set) # save

```

7.8 Statistical analysis of noisy data

```

import numpy as np
import healpy as hp
import tqdm
import seaborn as sns
import matplotlib.pyplot as plt
import tensorflow as tf
import scnn.layers
import scnn.dropout
test_set=np.load('3_channels/test_set.npy').astype(np.float32)[..., [0,1]]

test_noise=np.load('test_set_64_noise_multiple.npy')
print(test_noise.shape)
print(test_set.shape)
model_1= tf.keras.models.load_model('density/model_1_final.h5',
                                     custom_objects={'SphereConvolution':
                                         scnn.layers.SphereConvolution, '
                                         GraphPool':scnn.layers.GraphPool, '
                                         LinearCombination':scnn.layers.
                                         LinearCombination})
titles=['Source', 'Real', 'Avg. of generated with noise', 'Std. of
        generated with noise', "Relative
        variance"]

def plot_images(data, noise, model, name):
    fig, axes=plt.subplots(5,5,figsize=(15,10))
    fig.tight_layout()
    for i in tqdm.tqdm(range(len(idxs))):
        image=[]
        image_generated=[]
        for ii in tqdm.tqdm(range(noise.shape[0])):

```

```

        image_generated.append(model.predict(noise[ii,0,idxs_noise[i
                                                    ]][None,:]).flatten())
image_generated=np.array(image_generated)
print(image_generated.shape)
for j in range(5):
    if j==0:
        image.append(data[idxs[i],:,0])
    else:
        if j==1:
            image.append(data[idxs[i],:,1])
        else:
            if j==2:
                image.append(np.mean(image_generated,axis=0))
            else:
                if j==3:
                    image.append(np.std(image_generated,axis=0))
                else:
                    image.append(np.std(image_generated,axis=0)/
                                np.mean(
                                    image_generated
                                    ,axis=0)
                                )

plt.axes(axes[i%5,j])
hp.mollview(image[-1].flatten(),cmap='turbo',hold=True,nest=
            True,title=titles[j],min
            =0,max=upper[j])

hp.graticule()
plt.savefig("noise_aps_analysis_{}_model_{}.pdf".format(name[0],name
[1]))

idxs=[0,3,126,128,129]
idxs_noise=[0,3,6,8,9]
upper=[1,1,1,0.15,0.6]

plot_images(test_set,test_noise,model_1,["test_set","1"])

```