UNIVERSITÀ DEGLI STUDI DI MILANO - BICOCCA

FACOLTA' DI SCIENZE MATEMATICHE FISICHE E NATURALI
Corso di Laurea in Astrofisica e Fisica dello Spazio

Deep learning to speed up galaxies simulations

Master's Thesis

Relatore:
Prof. Dotti Massimo

Correlatori:
Prof. Landoni Marco
Dr. Rigamonti Fabio

Candidato:
Diana Alessandro
Matricola: 821425

2021-2022

Contents

1 Machine Learning **3**
1.1 Supervised Learning . . . 4
1.2 Regression problems . . . 4
1.3 Gradient Descent . . . 6
1.4 Neural Network . . . 7
1.5 Back Propagation . . . 9
1.6 Deep Learning . . . 10
1.7 Convolutional networks . . . 11
1.8 Residual networks . . . 14

2 Super-resolution **17**
2.1 CNN based . . . 18
2.1.1 FSRCNN . . . 20
2.2 Adversarial based . . . 22
2.2.1 SRGAN . . . 23
2.2.2 ESRGAN . . . 26
2.3 Transformer based . . . 27
2.3.1 ESRT . . . 29
2.4 Complexity and scalability . . . 30

3 Galaxies simulations **33**
3.1 Galaxy model . . . 34
3.2 Parameter estimation . . . 37
3.3 The sample . . . 38
3.4 Dataset preparation . . . 40

4 Architecture and training **41**
4.1 iSRCNN . . . 41
4.2 Ablation Study . . . 43

5 Hyperparameters Tuning **47**
5.1 Naive methods . . . 47
5.2 Bayesian optimization . . . 49
5.3 Hyperbands . . . 51
5.4 BOHB . . . 52
5.5 Optimizing iSRCNN . . . 54

6 Training **57**
6.1 Dataset pre-processing . . . 58
6.2 Optimizer and scheduler . . . 59
6.3 Weight initialization . . . 60
6.4 The impact of training size . . . 61
6.5 Outliers impact . . . 62

7 Results and conclusions **65**

Bibliography **69**

Alla mia Famiglia

For centuries *theory* and *experiments* were the fundamental tools that drove the scientific research. In the last decades, the dawn of the information age opened up the possibility of conducting our experiments in a different and complementary way using computer simulations, often faster and less expensive. However, the computational cost for many tasks is still prohibitive, especially in complex domains such as astrophysical or cosmological ones. A common workaround is to reduce the resolution of the numerical grid on which we intend to calculate the evolution of the physical system or its properties. However, this comes with a trade-off between the computational time and the amount of information contained in the simulation.
We propose a novel solution to get the best of the speed of a low resolution simulation while maintaining a very high quality of detail, using a machine learning (ML) algorithm.
Our framework is fully based on ML to learn an end-to-end map between the low-resolution space of the simulation to the high-resolution one, then we test with classical methods the physical accuracy *a posteriori* in the synthesized simulation to assess the performance of the algorithm.
The reconstruction of a high-resolution image given a low-resolution one is called super-resolution (SR), and is one of the problems successfully solved by deep learning in the last decade.
The algorithm I developed is indeed based on a deep learning model called fast super-resolution convolutional neural network (FSRCNN). This algorithm, together with its predecessor (SRCNN), were the pioneers in the field of super-resolution and demonstrated the effectiveness and simplicity of convolutional networks, already widely used in other computer vision tasks, in improving real-worlds images.
In a rapidly evolving field like deep learning, 5 years of progress have resulted in drastic increase in performance and, indeed, in the past few years several outstanding SR models were developed, ranging from the generative adversarial networks (GAN) which tries to reconstruct perceptually pleasing images by letting two complex adversarial models compete against each other, to the Transformer, a successful natural language model applied to the SR and capable of interpreting semantic information and their relationships within the image.
The problem with the new algorithms is that we have gone more and more towards deep and heavy architectures, not very suitable for modifications, increasingly opaque and computationally impractical if not with super-computers. In addition, some of those methods have such a large inference time that their application in trying to reduce the computational time of a simulation is futile.
I want to prove that even with a simple model it is possible to obtain outstanding results and a significant speed-up in the simulation pipeline. My model leverages the low number of parameters of the FSRCNN (tens of thousands instead of tens of millions, like in the enhanced version of the SRGAN) to deliver an algorithm that is trainable with a single GPU in one hour (instead of days using a computer cluster), and capable of up-scaling by a factor of 4 the resolution of thousands of simple simulations in milliseconds.
My main contribution is to modernize the FSRCNN with the most recent techniques developed in the field of deep learning without disregarding the initial objective of the model, i.e. being a fast and flexible algorithm, that's why I named this network improved SRCNN (iSRCNN).
To showcase the potential of our method we use a dataset containing hundreds of thousand of mock galaxies based on a physically informed model and used to estimate the parameters of the galaxies in the local Universe. Each mock galaxy is obtained fixing 18 parameters that produce four 2D unique maps for the surface

brightness, the velocity along the line of sight, the dispersion velocity and the mass-to-light ratio.

Preliminary results on a statistically significant sample show an improvement in the evaluation time of the galaxy's parameters by a factor of 100, when using a single CPU. This improvement is the same that one could obtain by calculating the fully optimized high-resolution simulation on a GPU. The parameter estimation error is below 10%, on average.

This is a very promising result. Given the extremely fast inference time, this opens up the possibility to perform this highly demanding task on a personal computer without a GPU. On the other hand, on can also leverage the computational power of a CPU cluster that, with respect to a GPU cluster, is much cheaper, accessible and with far more units, thus allowing a better parallelization and showing the potential of this technique.

Chapter 1

Machine Learning

What does *learning* mean? This is a question at the intersection of several (and somewhat related) disciplines, namely philosophy, cognitive sciences, neuroscience, psychology and artificial intelligence. Thus, there is not an unambiguous and agreed definition. Nevertheless, if we limit ourselves to the context of measurable tasks, we can define *learning* as the process of improving the performance on a task after making observations about the world. If the *learning agent* is a computer we can define this process as *machine learning*.
In practice, the computer observes past data to build a model based on the data and uses the model as both a representation of the acquired knowledge and as a tool to make predictions or decisions. This bottom-up approach is opposed to the top-down practice of programming the computer to perform a given task in a predefined way (i.e. *hard-coding*). The reasons to avoid hard-coding are manifold, ranging from the intrinsic unknowability of all the possible outcomes of certain problems (e.g. stock market forecasting) to the practical impossibility of hard-coding a solution for a nuanced task (e.g. face recognition). However, as a matter of fact, the designer of a machine learning algorithm is responsible to provide a model framework (or *architecture*) in order for the computer to learn effectively. The design choices (and, as we shall see, also the related parameters) embed a form of *a priori* knowledge in the architecture of the algorithm.

How does an agent *know* it is learning? A computer, just like a human, needs a means of testing the acquired knowledge or the forecasting capabilities. Based on the result of this reality check it can self-correct in order to improve its future performance. In other words, it needs a feedback mechanism. Depending on the type of feedback we can distinguish between different types of learning.
The first kind of feedback is realised through the use of labelled examples which represent a set of unique input-output pairs. The examples and the corresponding labels are elements of the input and the output spaces, respectively. The algorithm aim is to find the function that maps the input space in the output space. This type of learning is called *supervised learning*.
While the first kind of learning is achieved through examples, the second type is based on unlabelled data. The algorithm is led to build an internal representation of the input space. This representation is based on the relations between the elements within the input space and the algorithm learns to label similar elements with the same tag. Due to the absence of a a given output space this type of learning is called *unsupervised learning*.
The third and last kind is called *reinforcement learning*, due to the system of *rewards* and *punishments* underlying its feedback mechanism.

1.1 Supervised Learning

In this Thesis we are going to focus on supervised learning, therefore, it is necessary to lay the theoretical foundations of this branch of machine learning. As mentioned before, every supervised task is characterized by a set of input-output pairs. Defining the input examples as $(x_i) \in X$ and the output examples (or labels) as $(y_i) \in Y$ there must exist an unknown function $f(X) = Y$ that maps the input space to the output space.
In principle, this could be an extremely complex nonlinear function with thousands, millions or even billions of parameters. The task of the algorithm is to find an approximation $h \in H$ of this function. Depending on the context, we can also refer to the h function as the *hypothesis* that best describes the output data given the input, or as the *model* of the input data. Similarly, it is common practice in the literature to refer to the output space as the *ground truth* or *target value* (or simply *target*).

It is clear that the probability of finding by chance an approximation h which perfectly fits all the data, i.e. $h(X) = Y$, is increasingly smaller as the complexity of f grows. Conversely, the algorithm is expected to find a *best-fit* function such that $h(x_i) \approx y_i \ \forall (x_i, y_i)$. The proximity between $h(X)$ and Y can be made explicit in various ways depending on the case, but in general it can be parameterized by a loss function $\ell : Y \times Y \to \mathbb{R}^+$. Ideally, the aim of the algorithm is to choose the hypothesis that minimize the expected loss over all input-output pairs it will see in the future, such that the best hypothesis is:

$$\hat{h}(x) = \arg\min_{h \in H} \sum_i \ell(h(x_i), y_i) \tag{1.1}$$

The intuition behind this procedure is that the *learning* phase is iterated over a subset of input-output pairs, and the quality of the distilled hypothesis is assessed on pairs unknown to the agent.
The loss function, in practice, cannot be arbitrarily small, since some of the factors that determine the difference between $\hat{h}$ and f cannot be minimized. For example, the target function f may not be contained in the hypothesis space or not may not be easily represented within it. Alternatively, the target function may be nondeterministic or, more often, noisy. Lastly, the function may be too computationally expensive to approximate.

1.2 Regression problems

In the following pages, we will make extensive use of *regressions* algorithms, a subclass of supervised learning. The corresponding problems are characterized by linear functions and continuous-valued input and output. In general, the regression problem can be describe with a function:

$$h_{\mathbf{w}}(\mathbf{x}_i) = \mathbf{w}^T \cdot \mathbf{x}_i = \sum_j w_j x_{i,j} \tag{1.2}$$

Here, $\mathbf{w}$ is the vector of the real-valued coefficients to be learned, i.e. the *weights*, while $\mathbf{x}$ is the input vector[1]. In Figure 1.1 (left panel) is shown an example of univariate linear regression.

[1] We have defined $x_{i,0} \equiv 1$, such that $w_i^T x_{i,0} = w_0$.

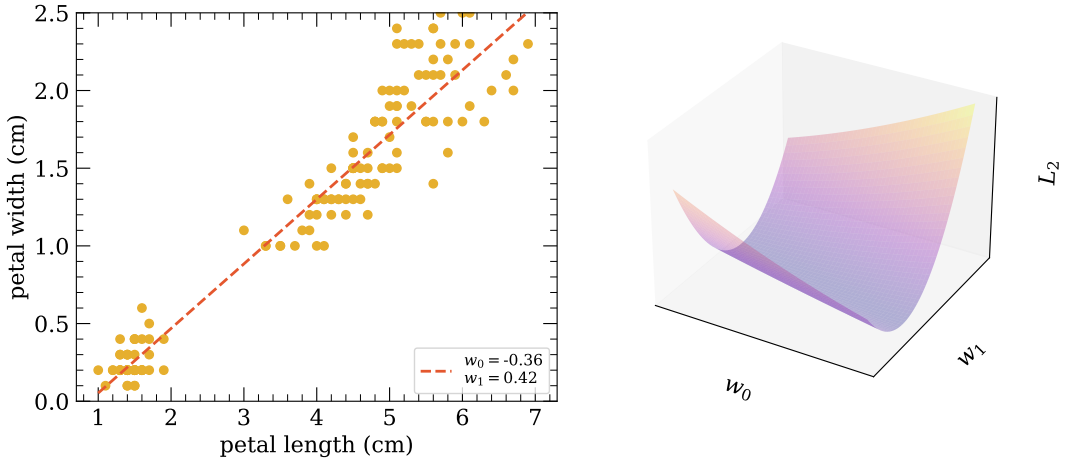

FIGURE 1.1: **Example of a univariate linear regression.** In the left panel, a scatter plot representing the petal widths and the petal lengths of different species of iris (Iris dataset; Anderson, 1936; Fisher, 1936). In the right panel, the *loss landscape*, representing the value of $\mathcal{L}_2$ as a function of (w_0, w_1), with the global minimum clearly visible.

In this simple case, the task of finding the hypothesis that best fits the data is achieved by minimizing the squared-error loss function (hereafter $\mathcal{L}_2$):

$$\mathcal{L}_2(\mathbf{w}) = \sum_i (y_i - h(\mathbf{x}_i))^2 = \sum_i (y_i - \sum_j w_j x_{i,j})^2 \tag{1.3}$$

The set of all possible values of $\mathcal{L}_2(\mathbf{w})$ is a N dimensional hyper-surface embedded in a $N+1$ dimensional space, where N is the number of *weights*. This hyper-surface is also called *loss landscape*, due to the similarities with an orographic map (see Figure 1.1, right panel). In this simple case the loss function is convex and has a global minimum that could be found analytically by solving the two equations:

$$\nabla_w \mathcal{L}_2 = 0 \tag{1.4}$$

where we have defined $\nabla_w \equiv (\partial/\partial w_0,\ \partial/\partial w_1, ..., \partial/\partial w_n)$. Every linear regression problem the $\mathcal{L}_2$ loss is a convex function for which there exists an analytical solution of the global minimum.
In general, however, it could not be possible to analytically minimize the loss function. Moreover, the number of weights to be learned could make the calculation computationally infeasible. It is necessary to use a tool that allow us to numerically solve all kind of problems, regardless of the type of loss function or the dimension of the parameter space.

A useful generalization of the linear regression can be achieved by considering linear combinations of fixed nonlinear functions of the input variables. In this case the problem is the hypothesis is:

$$h_{\mathbf{w}}(\mathbf{x}) = \mathbf{w}^T \cdot \phi(\mathbf{x}) = \sum_j w_j \phi(x_j) \tag{1.5}$$

where $\phi_j(\mathbf{x})$ are known as *basis functions*.
This function is still a linear combination of the weights, but it can represent a non-linear function, thus, greatly improving the generalization capabilities of the model. However, the assumption that the basis functions are fixed *a priori* greatly reduces

their practical applicability.
We will see in Section 1.4 how this problem may be solved.

1.3 Gradient Descent

The analogy between the loss function in the weights' space and the orographic map suggest the solution. The *gradient descent* is a technique generally attributed to Cauchy, expanded and formalized for non-linear optimization in Haskell (1944).
This method is a first-order iterative algorithm. Given a differentiable function $f(\mathbf{x})$ and a starting point $\mathbf{x} = \mathbf{x}_i$ the local direction of *fastest decrease* is $-\nabla f(\mathbf{x}_i)$. Therefore, the updated weights after an iteration are:

$$\mathbf{w}^{\tau+1} = \mathbf{w}^{\tau} - \lambda_R \nabla L(\mathbf{w}^{\tau}) \tag{1.6}$$

Where τ represents the iteration number, and λ_R is a real-valued coefficient that regulates the length of step. The initial point in the weights' space must be initialized $\mathbf{w} = \mathbf{w}_0$ (see Section 6.3, for details on initialization techniques).
The pseudo-code of the algorithm in its basic form is:

Algorithm 1 Gradient descent

repeat
for $x \in X, y \in Y$ **do**
    for $i \in (1, ..., p)$ where $p = dim(\mathbf{w})$ **do**
        $w_i \leftarrow w_i - \lambda_R \left(\frac{\partial L(\mathbf{w})}{\partial w_i} \right)$

The gradient descent algorithm, however, make use of all the input-output pairs at each iteration (or *epoch*). This is both computationally expensive and could lead to *overfit*, i.e. the algorithm may reproduce too closely the data it was trained on and fail to describe the unknown new data.
A faster version of this algorithm that is less prone to overfitting is the *stochastic gradient descent* (SGD; Robbins, 2007; Kiefer and Wolfowitz, 1952). In this variant, the algorithm firstly shuffle the dataset to avoid order-dependent patterns in the data and then it performs the gradient descent iteration over a small subset of the input-output pairs (i.e. a *a batch*). At each step the SGD selects a new batch until all the examples have been explored. This concludes the *epoch*.
The pseudo-code of the SGD is as follows:

Algorithm 2 Stochastic gradient descent

Randomly shuffle the dataset
Split the dataset D in N partitions: $(m_1, m_2, ..., m_n)$
repeat
for $m_i \in D$ **do**
    for $(x, y) \in m_i$ **do**
        for $i \in (1, ..., p)$ where $p = dim(\mathbf{w})$ **do**
            $w_i \leftarrow w_i - \lambda_R \left(\frac{\partial L(\mathbf{w})}{\partial w_i} \right)$

The algorithm has the advantage to reduce the computation time by a factor equal to the number of batches (N), while the standard error of the estimated mean gradient is proportional to the inverse of the square-root of the number of examples,

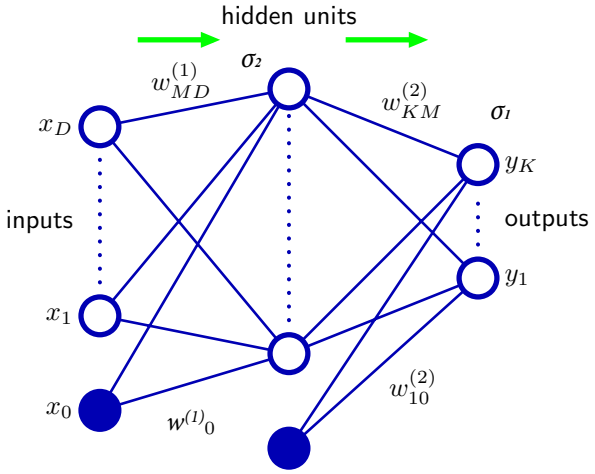

FIGURE 1.2: **Example of a simple neural network.** The figure shows a simple fully connected neural network with a single hidden layer. The information flows from left to right until it reaches the output layer.
Adapted from Bishop (2006).

therefore it is increased by $\sqrt{N}$. The computation time, adjusted for the standard error, is therefore reduced by a factor of $\sqrt{N}$ compared with the classical gradient descent.
In general, there is not a single best dimension of the batch. The batch size, as long as the learning rate, are considered *hyperparameters* of the model, and should be tuned accordingly to optimize the performance (see Section 5.5) for the discussion on hyperparameters tuning).

1.4 Neural Network

In previous Sections we saw that linear regression models are based on linear combinations of fixed *basis functions*. Although those models have useful analytical properties their lack of flexibility limits their practical applicability. One possible way around is to let the basis function *adapt* to the data. The most successful implementation of this type of algorithms was the *multilayer perceptron* (Werbos, 1970), an evolution of the simple perceptron (Rosenblatt, 1958). Those models are now considered the ancestors of the modern neural networks.

A neural network is a directed acyclic graph with fixed input and output nodes. Each node computes a function of its inputs and transmits the result to its output nodes. The information flows through the network from the input nodes to the output nodes (see Figure 1.2).
Mathematically, the neural network's neuron is a generalization of Equation 1.2:

$$h_{\mathbf{w}}(\mathbf{x}) = \sigma\left(\mathbf{w}^T \cdot \phi(\mathbf{x})\right) = \sigma\left(\sum_i w_i \phi_i(\mathbf{x})\right) \tag{1.7}$$

where $\sigma(\cdot)$ is a nonlinear function called *activation function*, by analogy with the activation of a biological neuron. In the case of linear regression the activation function is just the identity function.
This is the basic of a neural network model, which can be described as a series of

functional transformation. Each layer of neuron is fed with the elaborated information from the previous layer, and so on. The general structure of a fully connected neural network (in which every neuron is linked with all the neurons of the next layer) can be formalized as follows:

$$\mathbf{h_w}(\mathbf{x}) = ...\sigma^{(3)}\left(...\sigma^{(2)}\left(\sum_j w_{kj}^{(2)}\sigma^{(1)}\left(\sum_i w_{ji}^{(1)}\right)\right)\right) \tag{1.8}$$

where the superscript indicates the number of the layer. $\mathbf{h_w}(\mathbf{x})$ is now a vector, since every neuron in the final layer has a unique functional transformation of the input. The Equation 1.8 shows only the first two layers of a deeper neural network, the chain goes on until the last layer.
Each layer of neuron between the input layer and the output layer is called *hidden layer*.

The choice of the activation function is to be considered another hyper-parameter of the model and depends on the type of dataset. This can be as trivial as the identity function, but in practice it is always a nonlinear function. The reason for this choice is of algebraic nature. Given a linear function of a matrix $\phi(\mathbf{A})$ this can be expressed as another matrix $\mathbf{B} = \phi(\mathbf{A})$. In a neural network, if each downstream neuron multiplies its input vector with a linear function of its weights matrix, it has the effect of producing a chain of matrix multiplications $\mathbf{y} = \mathbf{ABC}...\mathbf{Yx}$. This matrix chain multiplication is equivalent to a single matrix $\mathbf{Z} = \mathbf{ABC}...\mathbf{Y}$. Therefore, $\mathbf{y} = \mathbf{Zx}$, which is the representation of a single layer neural network with linear activation.
The non linearity is what allows a sufficiently large neural network to represent arbitrary functions. Indeed, the *universal approximation* theorem states that a network with just two layers of neurons, the first nonlinear and the second linear, can approximate any continuous function to an arbitrary degree of accuracy (Hornik, Stinchcombe, and White, 1989).

Common choices for the activation function are:

- The logistic or *sigmoid* function: $\sigma(x) = 1/(1 + e^{-x})$
- The hyperbolic tangent function: $tanh(x) = (e^{2x} - 1)/(e^{2x} + 1)$
- The softplus function: $softplus = ln(1 + e^x)$
- The rectified linear unit (ReLU) function: $ReLU = max(0, x)$
- The gaussian error linear unit (GELU): $GELU = 1/2x(1 + erf(x/\sqrt{2}))$
- The sigmoid linear unit (SiLU): $SiLU = x/(1 + e^{-x})$

In Figure 1.3 are shown the six listed activation functions.
All these function are continuously differentiable, except for the ReLU function, which is not differentiable in $x = 0$. However, the ReLU is differentiable everywhere else and the value of the derivative at $x = 0$ can be arbitrarily chosen to be 0 or 1.
Although the ReLU has a discontinuous derivative, it is widely accepted as one of the best choice, especially for deep neural networks. This has several reasons, but they can be condensed into the computation efficient, scale invariant and highly-nonlinear nature of this activation function (Glorot, Bordes, and Bengio, 2010).

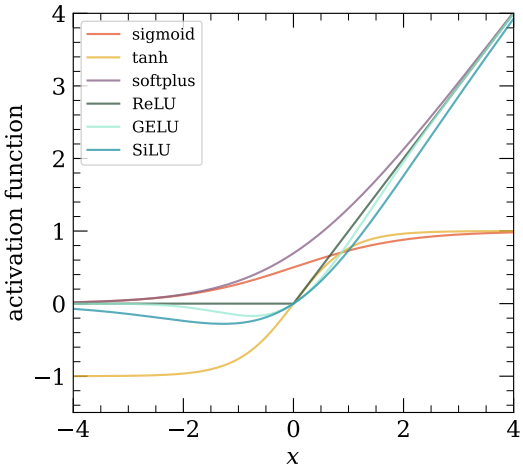

FIGURE 1.3: **Examples of activation functions.** The plot shows six of the most commonly used activation functions.

1.5 Back Propagation

We have already seen in Section 1.3 how the gradient descent and the SGD can be implemented to solve general supervised learning problems. We can apply the same algorithm to a neural network. In this context, however, it is useful to exploit the graph structure of the neural network, considering one layer at a time, connected to an input layer (its upstream layer) and an output layer (its downstream layer). Within this framework, we can calculate the gradient of the local information of that layer, and then we can go back along the graph to the true input layer. In this way, we compute the gradient for a small subset of the weights (the parameters of a single layer at a time), greatly improving the efficiency of the algorithm. The iterative evaluation of those derivatives is a process called *back-propagation*, as opposed to the *forward-propagation* which represents the natural flow of information (e.g. Figure 1.2). After the gradient has been calculated the weights are updated accordingly to the SGD (or other optimization algorithms, as we will see in Section 6.2).

Mathematically, the back-propagation can be formalized as follows. Let us consider a general feed-forward neural network with weights $\mathbf{w}$, with an arbitrarily almost-everywhere differentiable nonlinear activation function σ, and a generic loss function L. Usually, a loss function is a sum of the loss on each input-output pair, such that:

$$L(\mathbf{w}) = \sum_n L^n(\mathbf{w}) \tag{1.9}$$

Where the n superscript denotes the n-th example.
Therefore, we can consider the problem of calculating $\nabla_{\mathbf{w}} L^n(\mathbf{w})$ for a single term in the loss function evaluated on single example.
A single layer's output is:

$$z_j^n = \sigma(y_j^n) = \sigma\left(\sum_i w_{ji} z_i^n\right) \tag{1.10}$$

Here, z_i indicates the input of the neuron and z_j indicates the output, since an output is the input of the next downstream neuron.

Using the chain rule, the gradient of the *local* loss function's term can be written as:

$$\frac{\partial L^n}{\partial w_{ji}} = \frac{\partial L^n}{\partial y_j^n} \frac{\partial y_j^n}{\partial w_{ji}} \tag{1.11}$$

Now, we can define the first term as the *error* of the n-th example and the j-th output: δ_j^n. We can also notice that the second term is just:

$$\frac{\partial y_j^n}{\partial w_{ji}} = z_i^n \tag{1.12}$$

Therefore, the Equation 1.11 becomes:

$$\frac{\partial L^n}{\partial w_{ji}} = \delta_j^n z_i^n \tag{1.13}$$

To calculate the *error* of the neuron j from all the downstream neurons k, we can write:

$$\delta_j = \sum_k \frac{\partial L^n}{\partial y_k^n} \frac{\partial y_k^n}{\partial y_j^n} \tag{1.14}$$

Substituting now the Equation 1.10 into the previous one, we finally obtain the recursive back-propagation formula:

$$\delta_j^n = \sigma'(y_j^n) \sum_k w_{kj} \delta_k^n \tag{1.15}$$

The back-propagation algorithm is computationally efficient compared to the simple gradient descent, but the chain multiplication of the local gradients may lead to what is called *vanish gradient*. In fact, one or more derivative can be very close to zero (especially in the case of some activation functions, such as the sigmoid, the softplus and the hyperbolic tangent), or exactly zero (in the case of ReLU). If one of the derivative linked to the error δ_j is zero or close to zero, this means that changing the downstream weights will have a negligible effect on the output of the j-th neuron. This effect can cascade upstream and can extinguish the signals of many neurons, especially in very deep neural networks.

1.6 Deep Learning

In Section 1.4 we mentioned the universal approximation theorem. The classical version of this theorem states that a one-hidden-layer neural network with an arbitrarily number of *hidden* neurons (i.e. an arbitrarily layer *width*), can approximate a continuous function $f : \mathbb{R}^n \to \mathbb{R}^m$ with arbitrarily precision.

This theorem can be extended to the width-fixed case with an arbitrarily deep neural network. In particular, defining n and m the input and the output size, respectively, the class of neural networks of arbitrary depth, width $n + m + 2$, and activation function σ with a continuous nonzero derivative at some point, is dense in $C(K; \mathbb{R}^m)$ for $K \subseteq \mathbb{R}^n$ with K (Kidger and Lyons, 2020).

We can immediately acknowledge that this theorem formally justifies the use of a non continuously differentiable activation function, such as the ReLU. Moreover, it ensures that a more precise approximation can be achieved by building a wider and/or a deeper neural network.

In recent years, we have witnessed a golden age of deep neural networks, supported by ever more high-performance hardware.

1.7 Convolutional networks

In many problems it is desirable to have a solution which is invariant under certain transformations of the input (i.e. translation, rotation and scale). For example, an object in a two-dimensional image is not transformed into another object if the image is rotated, even though the transformation introduces significant changes in the input data. If sufficiently large numbers of examples under several transformations are available, then an adaptive model such as a neural network can learn the invariance, at least approximately. This approach may be not feasible in real scenarios, however, since it is sometime impossible to find a sample with examples that represents a large amount of transformations.
Therefore, a first approach is to randomly transform replicas of the input data according to a set of predefined rules depending on the type of invariance you want to enforce in the network. This procedure is called *data augmentation* and it has been proved to be very effective in increasing the generalization capability and in reducing the overfitting in many algorithms (e.g. Shorten and Khoshgoftaar, 2019).

A second approach, that does not exclude the first one, is to build the invariance properties into the structure of a neural network. For example, the invariance under translations can be achieved by using a local *receptive field* and shared weights. This is the case of convolutional neural networks (CNN), successfully implemented for the first time in LeCun et al. (1989) in the context of handwritten digits recognition.
The task of digits recognition is a simple type of classification problem, in which the input is a grey-scale image, composed by a matrix of intensities values, and the output is a posterior probability distribution over ten different variables, that in this context are called classes (one for each digit). This task is invariant under translation and scale transformations as well as small rotations.
In principle, it is possible to use a large enough fully connected neural network. The network could learn the invariance just by virtue of the large number of examples. However, this approach neglects *in toto* the semantic information of the image, i.e. the locality of the features, considering each pixel as a separate entity.
Conversely, a convolutional network extracts the local features in early layers and then merges the information in later stages, in order to be sensitive to high-order features. Simultaneously, the local information can be useful in multiple regions of the image, thus the translational invariance.
Finally, a convolutional network is invariant under translational transformations.
These properties are achieved in practice through three mechanisms: *local receptive fields*, weight sharing and subsampling.

The basic component of a convolutional neural network is the *convolution kernel* or *filter*. Throughout this theses we will use the latter, since the operation being is used is not, strictly speaking, a convolution. Since images are generally in matrix form, it is useful to think of layers as matrices and neurons as elements of these matrices.
Each neuron is sensitive, i.e. is connected, to small region of the input matrix. This small region is the *local receptive fields* and it is defined by a corresponding filter. The filter is just another matrix with dimensions $m \times n$ which are relatively small compared to the input dimensions. The *perception*, i.e. the activation of a downstream neuron, is achieved by the component-wise inner product between the a region of

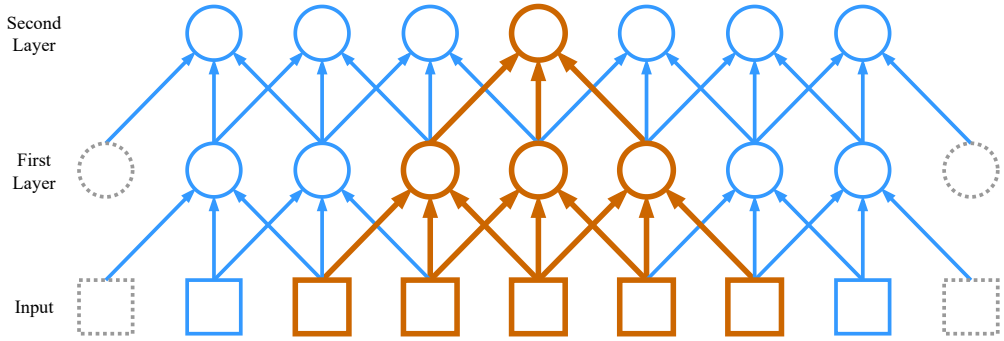

FIGURE 1.4: **Example of the receptive field of a CNN.** This picture depicts the first two layers of a CNN for a 1-dimensional input, with a kernel size $k = 3$ and a stride $s = 1$. A padding, represented with grey boxes, of $p = 1$ is added in order to keep the input size across the layers. In orange it is shown the receptive field of a single neuron in the second layer. We see here a general property of the CNNs: the deeper the neuron the larger the receptive field and, therefore, the feature scale.

the input with dimensions $m \times n$ and the filter ($\mathbf{k}$):

$$z = \sum_i \sum_j x_{ij} k_{ij} \tag{1.16}$$

By sliding the filter matrix over all the sub regions $m \times n$ of the input, we obtain a *feature map*. In this context, the elements of the filters are the weights shared between all the neurons of a layer.
In practice, it is common to use many filters for each layer, such that an hidden layer has $f \times m \times n$ learnable weights, where f is the number of feature maps in that layer. Since each neuron of a given feature map has a receptive field $m \times n$ and shares its weights with the other neurons, a single feature map is sensitive to a specific pattern in the input whose dimensions are smaller than the receptive field (see Figure 1.4).
After each convolutional layer, as usual, the nonlinear activation function of the output is calculated. If the input image is shifted, the activation of the feature map will be shifted by the same amount but will otherwise be unchanged. This provides the basis for the (approximate) invariance of the network outputs to translations of the input image.
In a practical architecture, there may be several convolutional layers. At each stage there is a larger degree of invariance to input transformations compared to the previous layer.
The whole network can be trained by error minimization using back-propagation to evaluate the gradient of the error function. This involves a slight modification of the usual back-propagation algorithm to ensure that the shared-weight constraints are satisfied.

Let us define now some useful technical vocabulary.
The dimension of the filters is called *kernel size*, generally denoted as k. Usually, the kernel size is symmetric, but in certain specific scenarios it is possible to use asymmetrical filters, i.e. with a rectangular shape.
The *stride* is the distance, or number of pixels, that the filter moves over the input matrix, it is usually denoted as s.
Finally, it is possible to add rows and/or columns of zeros on the border of the input

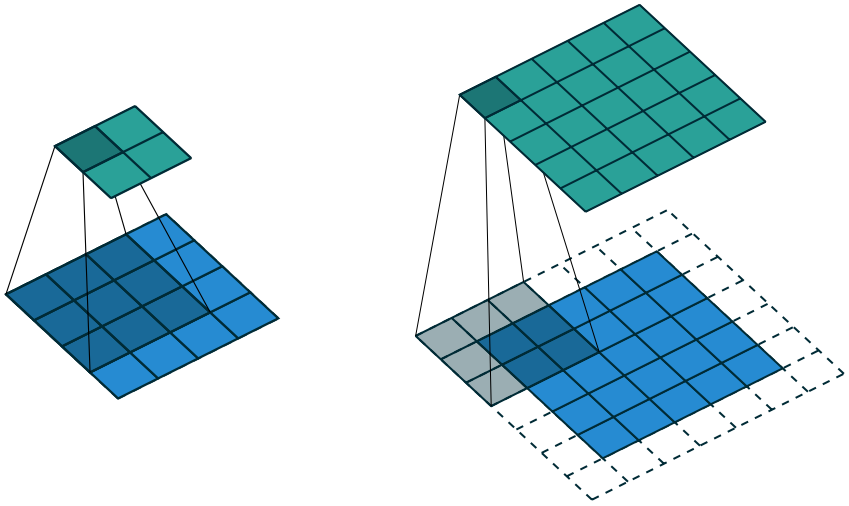

FIGURE 1.5: **Examples of filter application.** The blue maps are inputs, the cyan are outputs. The left figure shows a filter with a kernel size $k = 3 \times 3$, with no padding on the input matrix and a one-step stride in both the horizontal and vertical directions. In the right one, the same kernel size but with a padding $p = 1 \times 1$ and stride $s = 1 \times 1$. The output and the input in this case have the same dimension. Adapted from Dumoulin and Visin (2016).

matrix. This has the purpose to avoid under-sampling the external pixels of the matrix. The border is called *padding* and is denoted as p.
In Figure 1.5 it is depicted an example of these parameters and their impact on the dimension of the output layer.
A useful relation links the dimension of the output to the dimension of the input. Assuming a single dimension, denoted as n:

$$n_{out} = \frac{n_{in} - k + 2p}{s} + 1 \tag{1.17}$$

This can be generalized to the two-dimensional case.
We can also derive the equation for the $n_{in} = n_{out}$ case, assuming a common $s = 1$:

$$k = 2p + 1 \tag{1.18}$$

To conclude the description of convolutional networks we have to define the *channels*. We have already saw how it is common practice to use many filters in each convolutional layer. The number of filters is usually defined as *number of channels* or simply *channels*. This term is used for the conceptual similarity with the channels of a coloured image, for example an RGB image has 3 colour channels. Although the concepts are similar this should not be confusing in thinking that the channels of a convolutional layer represent the same "object" seen under different "coloured filters". Every channel represents a feature map of the input layer and it might not have real-world meaning, but it may be useful for the algorithm to process the input information.

Another ingredient that is not strictly a convolutional network but is somewhat related to this concept is the so-called *transposed convolution* or *reverse convolution* or *deconvolution layer*. In thesis we will use the first name, since the other two are

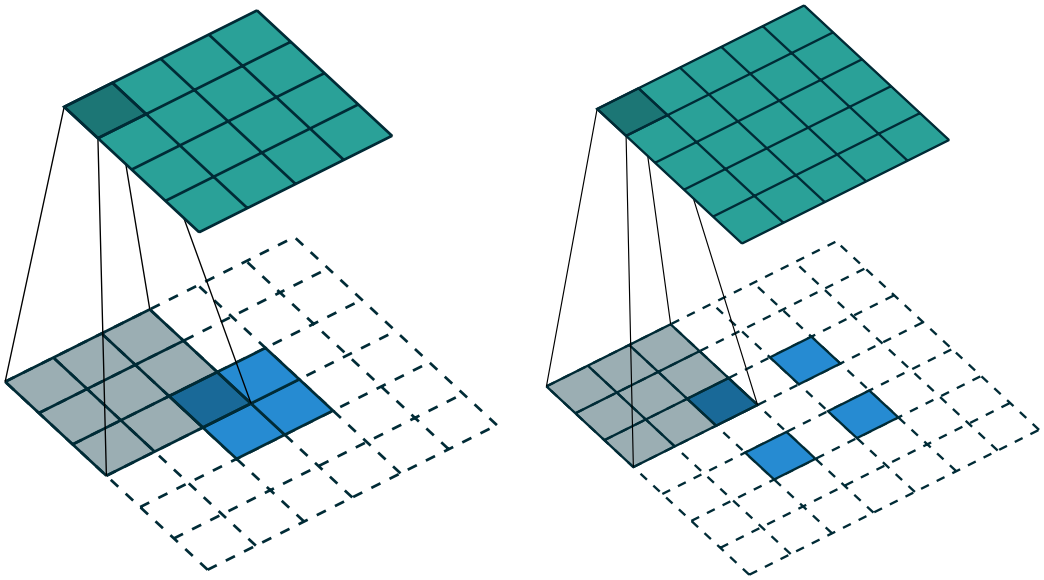

FIGURE 1.6: **Examples of transposed convolutions.** The blue maps are inputs, the cyan are outputs. The left figure shows a filter with a kernel size $k = 3 \times 3$, with no padding on the input matrix and no stride in both the horizontal and vertical directions. In the right one, the same kernel size but with a stride $s = 1 \times 1$. Notice that the output maps have larger size and that the stride, in the context of transposed convolution, acts like a spacing in-between the input map. Adapted from Dumoulin and Visin (2016).

quite imprecise. In fact, a convolution cannot be *reverted*, that is, one cannot retrieve the original values given the resulting output. Even the simplest of the filters, the identity filter, when convolved over an image, will result in some information being permanently lost. Convolution layer are usually employed to extract features with reduced size from the input. Although there is not anything we can do about this lost information, we can use transposed convolution to revert the size reduction (see Figure 1.6). In practice, if we feed into a convolutional layer an input X to obtain a $Y = f(X)$ a transposed convolution g with the same hyperparameters as f except for the number of output channels being the number of channels in X, then $g(Y)$ will have the same shape as X. Regarding the implementation of transposed layers, those can be viewed as back propagating convolutional layers in the feed-forward step and as feed-forward convolutional layers in the back propagation phase.
A useful formula to compute the output dimension of a transpose layer in 1D is the following:

$$n_{out} = (n_{in} - 1) \times s - 2 \times p + d \times (k - 1) + 1 \tag{1.19}$$

Where n_{out} and n_{in} are the output and input dimension, respectively; s is the stride; p is the padding; d is the dilation, which controls the spacing between kernel elements; k is the kernel size.

1.8 Residual networks

Another very popular and powerful approach to building deep networks, while avoiding the vanishing gradient problem, is the residual networks (He et al., 2016).

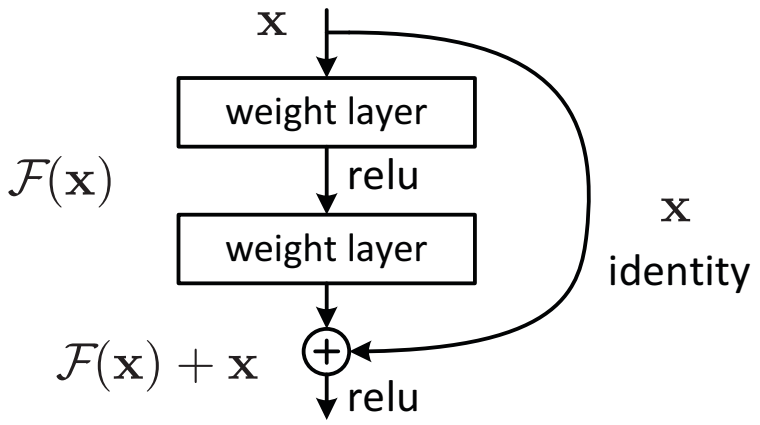

FIGURE 1.7: **Representation of a skip-connection.** (He et al., 2016)

The idea is very straight-forward. Typical feed-forward networks learn a new representation of the input space at each layer by using and "forgetting" all the information in the layer before. This can be easily seen in Equation 1.10 where the downstream neuron output z_j^n is completely oblivious of the previous neuron's activation value, which is now mixed with the weight matrix and the activation function. As a corollary, we can say that each layer must learn to do something useful.
The key idea behind residual networks is that a layer should *perturb* the representation from the previous layer without entirely replacing it. This is achieved by:

$$z_j^n = \sigma(y_j^n) = \sigma\left(z_i^n + \sum_i w_{ji} z_i^n\right) \tag{1.20}$$

To hold, this equation implies that i and j represent the same dimension. In other words, layers of these type must be the structurally equal, and they only differ by their parameters. It is easy to see how Equation 1.20 ensures that even if for some reason $W_{ij} = 0 \; \forall i, j$ we would have $z_j^n = z_i^n$. In residual networks a layer with zero weights simply passes its inputs through with no change.
In practice, the residual networks are just like classical neural networks except for the presence of what are called *skip-connections*. Those are simply shortcuts through which the input information can reach a deeper level of the network. Usually a connection is created when the input of a certain layer L^i is not only the output of the previous layer L^{i-1}, but also the output of another layer, not necessarily linked to the $i-th$ one.
For example, in a simple feed forward network:

$$z^i = L^i(z^{i-1} + z^{i-n}) \tag{1.21}$$

where z represents the output of the corresponding layer.
In this example the skip-connection is implemented with a element-wise sum, and the result is that the z^{i-n} output is not only fed to the downstream layer L^{i-n+1} but also to the $i-th$ layer.
See Figure 1.7 for an example of skip-connection.

Residual networks are often used with convolutional layers in vision applications, but they are in fact a general-purpose tool that makes deep networks more robust and allows researchers to experiment more freely with complex and heterogeneous network designs.

Chapter 2

Super-resolution

Image super-resolution (SR) is the process of recovering high-resolution (HR) images from low-resolution (LR) images. These methods can be broadly classified in two groups: the multi-image super-resolution that tries to combine multiple slightly different images of the same scene; and single image super-resolution (SISR) in which the algorithm tries to learn a correspondence map between the LR and HR images.
This type of algorithm is gaining more and more importance over the years due to its wide applications in different fields: medical imaging (see for example Kouame and Ploquin, 2009; Isaac and Kulkarni, 2015), surveillance (e.g. Zhang et al., 2010; Rasti et al., 2016, multimedia industry (e.g. Malczewski and Stasiński, 2009) and so on.

With this Thesis we want to prove that it is possible to reconstruct high-fidelity high-resolution physical simulations starting from a low-resolution version.
As we will detail in the following Chapter, the simulation we are trying to reconstruct compute the properties of simulated galaxies, given a set of input parameters. The output of the simulation are four maps for every galaxy, if the maps are stacked one on each other the output can be viewed as a 4-channels coloured image. Each one of these channels actually depicts the distribution of one physical parameter over space (ranging from the surface brightness of the galaxy, to the dispersion velocity, and so on).
Therefore, we are interested in SISR for this type of simulation up-scaling.

Specifically, building on the previous Chapter, the focus is a supervised deep learning algorithm that performs a regression over an image.
The problem can be formalized as follows. Given a certain *degradation* function $\mathcal{D}$, we can say in general that:

$$I^{LR} = \mathcal{D}(I^{HR}) + \eta \tag{2.1}$$

Where I^{LR} and I^{HR} are respectively the LR and the HR images, and η is an optional noise term independent from the degradation function $\mathcal{D}$[1]. The SISR tries to approximate the inverse degradation function:

$$I^{SR} = \hat{h}(I^{LR} + \eta) \approx \mathcal{D}^{-1}(I^{LR} + \eta) = I^{HR} \tag{2.2}$$

Where the $\hat{h}$ represents the learning algorithm and I^{SR} is the *super-resolution* image, which is in practice an approximation of the real HR image.
This is performed, as usual, by minimizing (or maximizing) a loss (or accuracy) function:

$$\hat{\theta} = \arg\max_{\theta} \mathcal{L}(I^{SR}, I^{HR}) \tag{2.3}$$

[1] This is a common name in the literature since the SR task is often aimed to restore compressed, or otherwise damaged, images. The meaning of this function is to be considered more generally as the *ideal* map between the HR image and the LR image.

Where $\hat{\theta}$ is the optimal set of model's parameters the minimizes a certain loss function $\mathcal{L}$.
In every practical scenario the *degradation* function is not only unknown, but also not unique. In fact, given any LR image there are many ways to obtain a mathematically compatible HR image. This is a fundamentally unsolvable problem of the SR tasks, since in a certain number of pixels we do not have enough information to discriminate between all the possible HR *parents* images that could have led to a certain LR image. In other words, SISR is ill-posed since the HR target, also called the *ground truth* is not the only solution (e.g. Wang, Chen, and Hoi, 2021). For example one can randomly sample pixels from patches of the original image, take the mean of the patches, select only the maximum (or minimum) value of the patches or any combination of those. Often, a LR is the result of a compression algorithm whose rules are generally very complex and hard to embed into a super-resolution algorithm.
In our case there is no set of rules that lead from the HR image to the LR one, since the two images represents the same physical object with a different grid resolution.
In recent years a great variety of models have been proposed, and in this chapter we will give a brief overview of the most relevant models of the last decade, during which there has been an explosion of variety and achievements in deep learning algorithms.

2.1 CNN based

CNN-based super-resolution algorithms use convolutional layers as their main building blocks, as the name implies. The progenitor of this family of algorithms is the super resolution convolutional neural networks (SRCNN; Dong et al., 2016). This was the first deep learning method for SISR which can directly learn an end-to-end mapping between the LR image and the HR image. This network has a very simple design and relatively few parameters.

It composed by three *convolutional blocks*, that indicate the composition of a convolutional layer followed by an activation function (see Figure 2.1). The input LR image get pre-up-sampled using a bicubic interpolation before being fed to the network. The first convolutional block mainly extracts patches and representations of the LR image. This is also known as *feature extraction*. The block is composed by a convolutional layer with 9×9 kernel, such that the resulting function on the input is:

$$f_1(X_{bi}) = \max(0, W_1 * X_{bi} + B_1) \tag{2.4}$$

Where X_{bi} is the bicubic-upsampled input of the layer, W_1 are the filters parameters and B_1 are the biases. When not explicit, the stride of the convolutional layer is always $s = 1$ and the padding is set so that the Equation 1.18 is satisfied.
The second block maps the 64 dimensional representations (or feature maps) into 32 *condensed* feature maps. Then the last layer realizes the reconstruction of high-resolution image, by combining all the information distilled in the previous layer, with a convolutional layer and no activation function. The activation functions for the whole networks are ReLU (see Chapter 1). The loss function is a simple L2 (see Equation 1.3).
The parameters of the network can be summarized as follows:

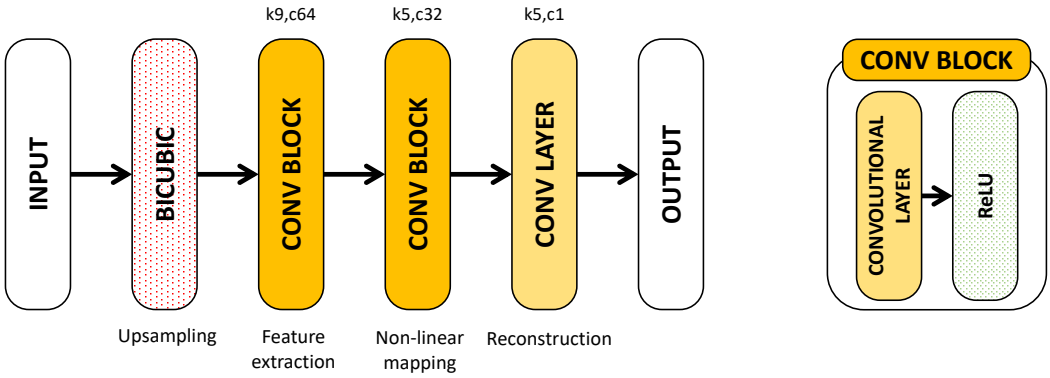

FIGURE 2.1: **SRCNN architecture.** The diagram schematically represents the building blocks of the super-resolution convolutional neural network (Dong et al., 2016). The solid filled blocks represent the layers with learnable parameters. Instead, the dot filled blocks depicts functions with fixed parameters. The arrows mark the information flow, from the input to the output. Above every block containing a convolutional layer there are noted the respective kernel size (e.g. "k9" stands for a 9×9 kernel) and the output channel (e.g. "c64" stands for 64 output channels). Using this notation the input channel of every block is implied by the output of the previous block. The initial input can be a grey-scale image (1 channel) or a RGB image (3 channels).

First layer	$9 \times 9 \times 1 \times 64$
Second layer	$5 \times 5 \times 64 \times 32$
Third layer	$5 \times 5 \times 32 \times 1$
Total	~ 57k

These parameters are intended for the original version of the architecture, which involves the reconstruction of gray-scale (single-channel) images. However, in this thesis we are working on 4-channels images, therefore the number of parameters we are comparing to is actually $\sim 75k$.

SRCNN represented the first step into the realm of deep learning SISR, setting several records and surpassing all previous algorithms both in terms of accuracy and speed of inference. It showed that it is possible to build an end-to-end super-resolution algorithm without any *a priori* knowledge of the dataset and, thus, without the necessity to hand-craft tailored algorithms that works only for certain type of images. In addition, the nature of convolutional neural networks, which are based on matrix summation and multiplications, allows for a simple and scalable paralellization. This is even more advantageous in the era of high performance graphic processing units (GPUs), hardware specialized in parallel computing and particularly efficient in solving linear algebra problems.

However, SRCNN suffers from several problems. First among all, the depth of this network is strongly limited by the vanishing gradient problem. Thus, increasing the numbers of network layers cannot improve the performance of the SRCNN.

This in turn limits the level of expressiveness of the network. Nevertheless, given the extreme simplicity of this architecture, and the innate modularity of deep learning approaches, is easy to improve on this baseline.

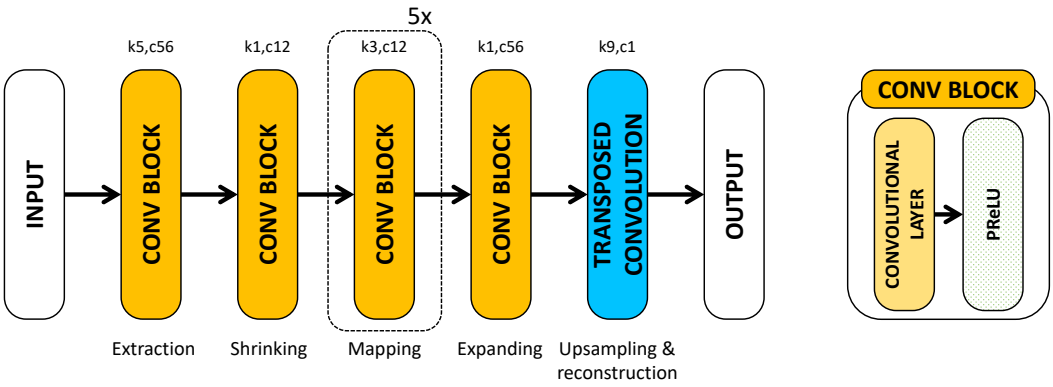

FIGURE 2.2: **FSRCNN architecture.** The diagram represents the networks proposed in that improves the SRCNN. The initial bicubic upsampling has been removed the upsampling is handled by the last layer. The feature extraction is similar to that seen in its predecessor. A significant reduction in the parameters number is obtained by shrinking the number of channels using a 1×1 kernel convolutional layer, before applying the non-linear mapping. The channels are then expanded again after the mapping. The final upsampling and the reconstruction is handled by a transposed convolutional layer (see text) with a kernel size 9×9.

2.1.1 FSRCNN

The first attempt to improve the performance of the SRCNN came a few months later by the same research team. The goal was to create a light-weight version of the predecessor while maintaining the same level of performance. For this reason the network has been named fast super-resolution convolutional network (FSRCNN; Dong, Loy, and Tang, 2016).
The first huge improvement was the substitution of the pre-up-sampling phase with a post up-sampling block. In this way the network receives a much smaller image as input and this greatly reduces the number of multiplications and sums during the whole feed forward and back propagation process. This also allows the network to be more scalable to higher resolution factors.
The structure of FSRCNN is a little more complicated and can be divided into five parts: feature extraction, shrinking, mapping, expanding and reconstruction (see Figure 2.2). The first block is similar to that of the original SRCNN, but with a smaller kernel size. This is due to the fact that, in order to extract features with the same size, the extraction on the LR image needs a smaller perceptive field than the same procedure performed on a pre-up-sampled image, as in the SRCNN network.
The second block uses a new technique which employs a 1×1 kernel convolutional layer to shrink the number of channels from 56 to 12. In this way the non-linear mapping performed in the third block acts on a much lower dimensional space, thus drastically reducing the number of parameters.
In the third block, they substituted the single 5×5 kernel mapping with four 3×3 layers. In this way the receptive field of the mapping is much larger but with few parameters (see Figure 1.4).
The fourth block then restore the channels number before the final stage.
The last block is a transposed convolutional layer (see Section 1.7), which replace the bicubic pre-up-sampling in the SRCNN with a learnable layer at the end of the

network to ease the computational burden.
Thus, unlike in the SRCNN in which the computation complexity grows quadratically with the spatial size of the HR image, the FSRCNN complexity grows quadratically in the LR size. It is worth noting that the transposed convolution layer does not simply substitute the conventional interpolation kernel. Instead, it consists of several learnable up-sampling kernels that work as one to generate the final HR output. Dong, Loy, and Tang (2016) showed that replacing this layer with uniform interpolation kernels results in a drastic performance drop. Since this type of layer can be interpreted as an inverse convolution, the authors imagined to extract the features from the output HR image. In the original SRCNN, the opted to adopt a 9×9 kernel convolutional network. Similarly, by reversing the information flow, the transposed filters should also have a spatial size 9×9.
However, this represents a major bottleneck in the reconstruction capacity of the network, since it limits the receptive field to a 9×9 pixels matrix. We will see how we can address this problem in Section 4.1 with our improved version of this architecture.

The parameters of the FSRCNN network can be summarized as follows:

Feature extraction	$5 \times 5 \times 1 \times 56$
Shrinking	$1 \times 1 \times 56 \times 12$
Non-linear mapping	$4 \times 3 \times 3 \times 12 \times 12$
Expanding	$1 \times 1 \times 12 \times 56$
Reconstruction	$9 \times 9 \times 56 \times 1$
Total	$\sim$ 12.5k

Similarly to the SRCNN, this architecture is intended to reconstruct gray-scale images, with a single channel. For a direct comparison with out architecture we scale this network to receive as input 4-channels images, as in our case. Therefore, the number of parameters we are comparing to is actually $\sim 30k$.
By virtue of the lower number of parameters experiments show that the FSRCNN achieves a speed-up of more than $40\times$ and superior performance with respect to the SRCNN (Dong, Loy, and Tang, 2016).
In addition to the great improvement in speed, the FSRCNN has another appealing property that could facilitate fast training and testing across different up-scaling factors. Specifically, in FSRCNN, all convolution layers (except the transposed convolutional layer) can be shared by networks of different up-scaling factors. During training it is only needed to fine-tune the transposed convolutional layer for another upscaling factor with almost no loss of mapping accuracy. During testing it is required to make convolution operations once, and up-sample an image to different scales using the corresponding transposed convolutional layer.

The last contribution to the state-of-the-art is the use of a modified version of the ReLU activation function: the parametric ReLU (PReLU). The only difference, as the name suggests, is that the negative part of the function has a slope that depends on a learnable parameter, such that:

$$PReLU(x) = \begin{cases} x & x \geq 0 \\ ax & x < 0 \end{cases} \tag{2.5}$$

Where a is the learnable parameter, which is $a = 0$ in the ReLU.
PReLU is useful to avoid the *dead features* caused by zero gradients in ReLU (see Zeiler and Fergus, 2014).

2.2 Adversarial based

During the initial exploration of the literature to select the most promising paths in search of the right algorithm to solve the super-resolution problem in an effective and accurate way, we could not ignore a new class of algorithms that has galvanized the SISR scene over the past 5 years. We are talking about the *adversarial* networks.
This type of algorithms have originated in the context of realistic images generation. The aim was to build images that can fool human eyes, and the generative adversarial networks (GAN; Goodfellow et al., 2014) has taken the first decisive step with a simple idea: training two parallel networks that compete against each other. The first one is a generative network, whose task is to create a realistic image from a noise input. The second one is its adversary, a discriminative model (see Figure 2.4) that learns to determine whether an image comes from the generative model or is indeed a real-world image.
The generative model $\mathcal{G}$ can be thought of as analogous to a team of counterfeiters, trying to produce fake currency and use it without detection, while the discriminative model $\mathcal{D}$ is analogous to the police, trying to detect the counterfeit currency.
Competition in this game drives both teams to improve their methods until the counterfeits are indistinguishable from the genuine articles (Goodfellow et al., 2014).

In this thesis we consider the two models composing the GAN as deep multilayer perceptron so that it is possible to train both models using only the back propagation and dropout algorithms and sample from the generative model using only forward propagation.
In this context, we define the noise input space as Z, the real image input space, or data space, as X, an the map between the noise and data space $\mathcal{G} : Z \to X$ which is a differentiable function parameterized by a set θ_g.
We also define a second differentiable function that maps the data space into a scalar $\in [0, 1]$ such that $\mathcal{D} : X \to [0, 1] \subset \mathbb{R}$. This function is parameterized by another set of parameters θ_d.
$\mathcal{D}$ is the discriminative model and $\mathcal{D}(x, \theta_d)$ represents the probability that an image $x \in X$ came from the data space rather than the generator model.
The training of adversarial model is aimed to maximize the probability of assigning the correct label to both training examples and samples from $\mathcal{G}$. Then the generative model is trained to minimize the quantity:

$$log(1 - \mathcal{D}(\mathcal{G}(z))) \tag{2.6}$$

This problem can be also formalized as a two-player minimax game with value function $V(\mathcal{G}, \mathcal{D})$:

$$\min_{\mathcal{G}} \max_{\mathcal{D}} V(\mathcal{G}, \mathcal{D}) = \mathbb{E}_{x \sim p_{data}(x)}[log\mathcal{D}(x)] + \mathbb{E}_{z \sim p_z(z)}[log(1 - \mathcal{D}(\mathcal{G}(z)))] \tag{2.7}$$

Where $p_{data}(x)$ is the data distribution function; $p_z(z)$ is the noise distribution function that is fed to the generator.

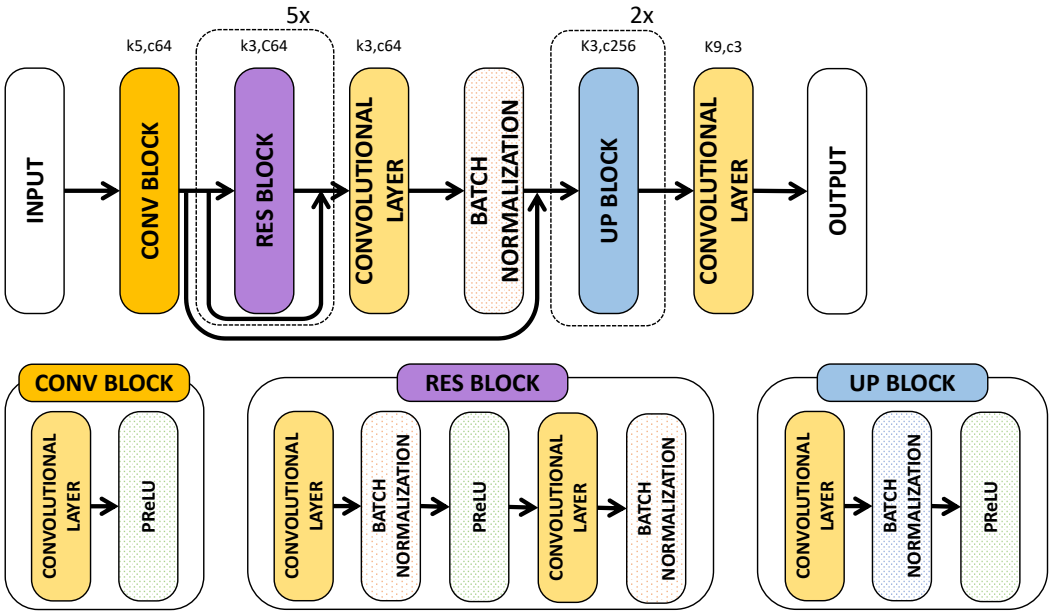

FIGURE 2.3: **SRGAN generator architecture.** The diagram represents the first model in the SRGAN algorithm, which is responsible for generating good quality SR images capable of deceiving the discriminator (see Figure 2.4) into believing they are HR images instead. The structure closely resembles a deep residual network (He et al., 2016) with the typical skip-connections clearly visible between each residual block (in purple) and between the first convolutional block and the first up-sampling block (in light-blue). Batch normalization layers, along with the activation function, are not learnable layer and therefore are represented with a dotted background.

In Goodfellow et al. (2014) it has been demonstrated that, in a non-parametric setting, that is, with a model that works exclusively in the space of probability density function (pdf), the Equation 2.7 has a global optimum for the pdf of the generator p_g that converges to the pdf of real data p_{data}.

2.2.1 SRGAN

The first successful attempt to use GANs to solve the SISR problem is the so-called super-resolution GAN, or SRGAN (Ledig et al., 2017). The goal that prompted the research team to use this new type of algorithm was to improve the subjective quality of SR images and increase their high-frequency details.
The main difference between classical models and the SRGAN is the use of new kind of loss functions. While in general it is common to minimize the mean squared error (or L_2 loss), this type of optimization naturally leads to an high peak signal-to-noise ratio (PSNR) but poor perceptual quality, since it is based on pixel-wise difference. The main contribution of the SRGAN paper is to introduce the so-called *perceptual loss* defined as the weighted sum of a *content loss* and an *adversarial loss*:

$$\mathcal{L} = \alpha \mathcal{L}_X + \beta \mathcal{L}_{gen} \tag{2.8}$$

Where in the original paper $\alpha = 0$ and $\beta = 0.001$.
In general, the content loss can be as simple as the MSE, but in this case it is proposed

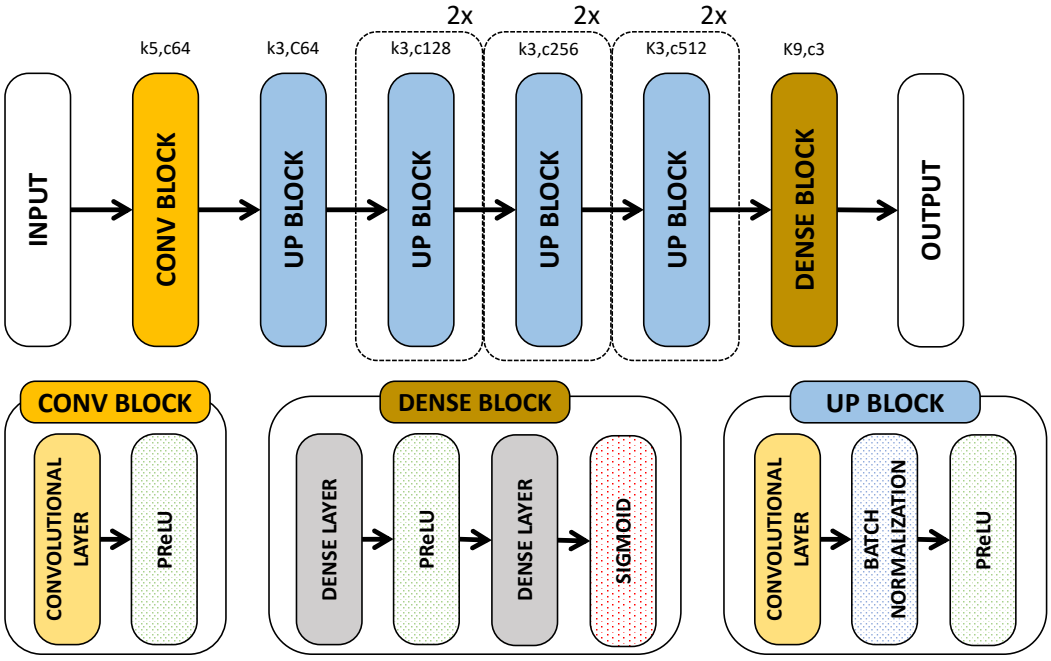

FIGURE 2.4: **SRGAN discriminator architecture.** The diagram represent the second model in the SRGAN algorithm, which is responsible for discriminating the true HR images from the SR generated ones. In the context of the GANs, this is the adversarial network that competes with the generating network in order to improve its performance. Differently from the generator (see Figure 2.3) this structure lacks skip-connections and it is much more similar to a standard deep convolutional network. As usual, the annotations over the blocks represent the kernel size of the convolutional layers inside the blocks and their output channels.

a type of loss based on another pre-trained neural network, namely the VGG-19 algorithm for image classification (Simonyan and Zisserman, 2014). This content loss exploits the pre-trained VGG-19, which is a very deep convolutional neural network, trained on large datasets of real-world images, to borrow some of the high-level feature maps of the network and to use them to recognise familiar high-level structures in the generated images.
Formally, defining ϕ_{ij} as the VGG feature map obtained by the j-th convolutional block and before the i-th maxpooling layer, we can define the content loss as follows:

$$\mathcal{L}_{\text{VGG}} = \frac{1}{WH} \sum_{x=1}^{W} \sum_{y=1}^{H} \left[\phi_{ij}(I^{HR}) - \phi_{ij} \left(\mathcal{G}(I^{LR}) \right) \right]^2 \tag{2.9}$$

Where W and H are the width and the height of the i, j feature map, respectively; $\mathcal{G}$ is the generator.
Although the notation may be heavy the meaning of the Equation 2.9 is straightforward: the SR image and the HR are fed into the pre-trained VGG-19 network, which is truncated at the i, j feature map. The difference of the two resulting feature maps is squared and then normalized by the pixel area of the feature map.
The second component of Equation 2.8 is the classical feedback provided by the discriminator (see Equation 2.6) declined in a slightly different way:

$$\min_{\theta_g} - log\mathcal{D}(\mathcal{G}(z)) \tag{2.10}$$

for better gradient behavior.

Another important ingredient that we see used in the SRGAN is the batch normalization layer (Ioffe and Szegedy, 2015). This component was proposed to solve the so-called *internal covariate shift* problem, which is the tendency to have a changing distribution for each layer's inputs during training. According to Ioffe and Szegedy (2015) this slows down the training by requiring lower learning rates and careful parameter initialization, and makes it hard to train models with saturating nonlinearities.
We will see however, how the presence of this component is not always necessary and can sometimes even be harmful.

The generator network architecture (see Figure 2.3) is based on a deep residual network (He et al., 2016) with skip-connections along with a perceptual loss and a content loss. On the other hand, the discriminator resembles a classical deep CNN, without skip-connections (see Figure 2.4).
Differently from the SRCNN and the FSRCNN, we will not enumerate all the parameters for each layer but just the total amount, due to the large number of layers. For the SRGAN the number of parameters is roughly 1.515 millions.
SRGAN, by its nature, is a powerful tool to produce perceptually pleasant images since it is train against an adversarial network that simulate a *critical* human eye. This automatically excludes some degree of attention to real details, preferring verisimilitude over mathematical precision in the process of reconstruction. This does not go well with the need to reconstruct physical simulations and is one of the reasons why we preferred other solutions (see Section 4.1).
In fact, SRGAN achieves poorly in the SR tasks when using the peak signal-to-noise ratio (PSNR) metric (see Figure 2.9), even with relatively simple datasets such as the Set5 (Bevilacqua et al., 2012).

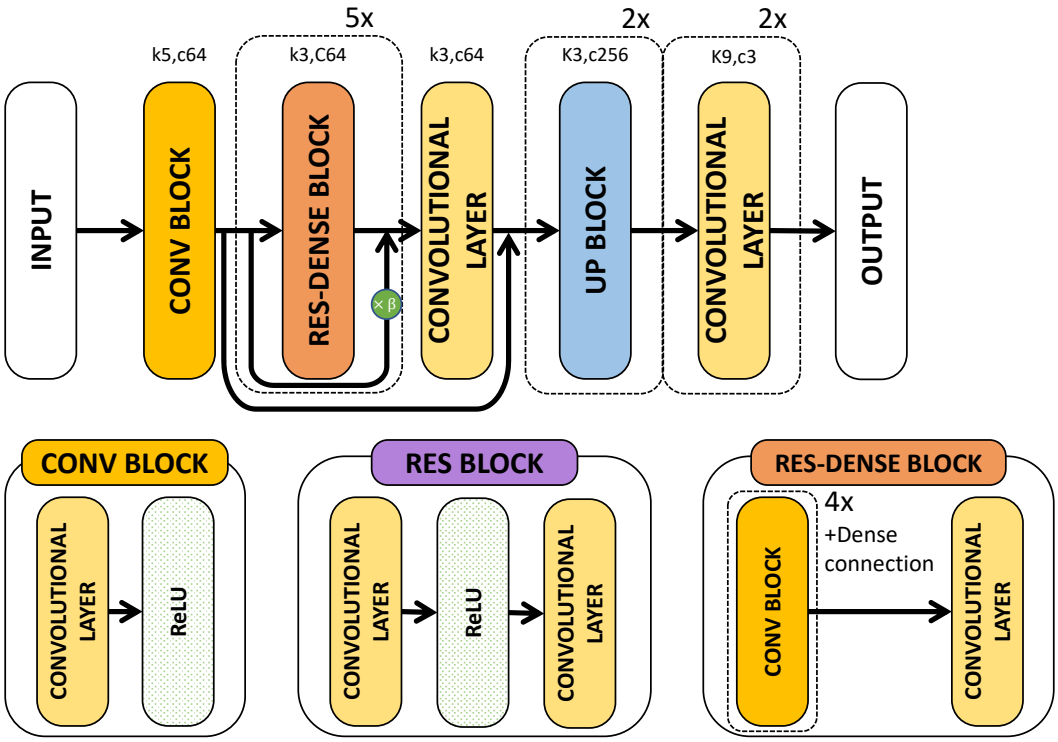

FIGURE 2.5: **ESRGAN generator architecture.** The diagram represents the modified generator in the ESRGAN model (Wang et al., 2018). The architecture closely resembles the SRGAN generator, with few fundamental differences (see text).

2.2.2 ESRGAN

The most successful successor of the SRGAN was the enhanced SRGAN (ESRGAN; Wang et al., 2018). They revisited the key components of SRGAN and improve the model in three aspects.

First, the main block of the network was substituted with a deeper *residual-in-residual dense block* (RDDB), which, according to the authors, should be easier to train (see *res-dense block* in Figure 2.5). Moreover, the batch normalization layers were removed as suggested in Lim et al. (2017), since batch normalization smooth out the features, and reduces the flexibility range from the networks by normalizing the features (see Figure 2.5). In addition, by removing batch normalization the performance greatly increases, and the training time is significantly reduced, since this type of operation consume a large amount of memory.

Second, the discriminator was improved using the relativistic average GAN (RaGAN; Jolicoeur-Martineau, 2019), which learns to judge "whether one image is more realistic than the other" rather than "whether one image is real or fake" (Wang et al., 2018).

Third, it has been proposed an improved perceptual loss by using the VGG features before activation instead of after activation as in SRGAN. ESRGAN empirically proved the adjusted perceptual loss provides sharper edges and more visually pleasing results.

For completeness, another small tweak in the SRGAN architecture is the addition of weighted skip-connections, parameterized with a scalar β (see Figure 2.5). Finally, they employed a dynamic learning rate, which is halved every fifty thousands iterations. This greatly help the convergence, but it is not the optimal approach, as we

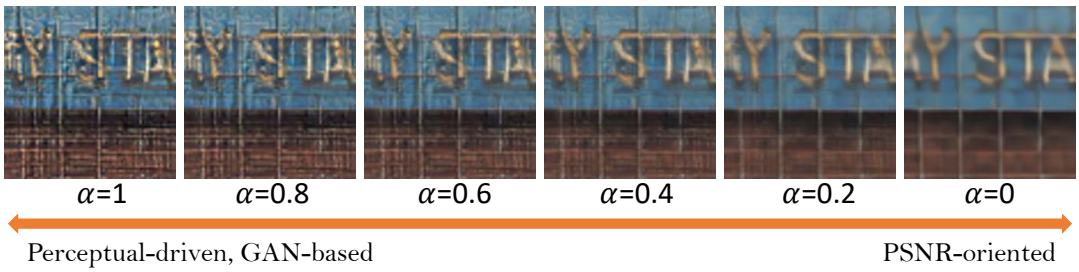

FIGURE 2.6: **Comparison between perceptual-oriented and pixel-wise loss in ESRGAN.** On the left the SR image produced by a network trained on a fully adversarial loss, focused on reconstructing perceptually pleasant image. On the right the same network trained on a pixel-wise loss, less prone to artifacts production. (Wang et al., 2018).

will see in the Chapter 6.

The main problem with ESRGAN, even more than its predecessor, is well explained in the original paper itself. One of the experiments carried out was to compare the difference between a pixel-wise loss and a fully adversarial loss. This was performed varying a parameter $\alpha \in [0, 1]$ such that:

$$\mathcal{L}_{tot} = (\alpha - 1)\mathcal{L}_2 + \alpha\mathcal{L}_{gan} \tag{2.11}$$

The results are well summarized in Figure 2.6 from the ESRGAN paper, showing how for $\alpha \to 1$ the images are sharper but many unwanted artifacts arise. this problem is at best unpleasant in the case of real-world images, but deleterious and harmful in the case of a physical simulation, in which artifacts can be unphysical or utterly wrong.
On the other hand, the pixel-wise loss ($\mathcal{L}$) shows a smoother image (which is not desirable in real-world images) but it is much more faithful to the ground truth and the physical informations (see the scaffolding in foreground in Figure 2.6 with the bars the remains parallel).

Nevertheless, this network was used to solve a super-resolution task in the context of fluid-dynamics (Bode et al., 2021). This implementation is even more demanding, since it deals with 3D data, however we cannot make a direct comparison since the time performance and the hardware are not detailed in the paper.

As in SRGAN we do not report the parameters count for each layer, but just the total amount, which is roughly 16 millions parameters. Roughly ten times the number in the predecessor. The complexity-performance trade-off of the ESRGAN is depicted in Figure 2.9.

2.3 Transformer based

The Transformer (Vaswani et al., 2017) is a recent deep learning architecture aimed to solve sequence-to-sequence problems, usually the translation of sentences from one language to another.
This architecture revolutionized the natural language process field becoming its protagonist until today. It relies entirely on an attention mechanism to draw global dependencies between input and output, and employs an encoder-decoder structure. Let us analyse those two elements.

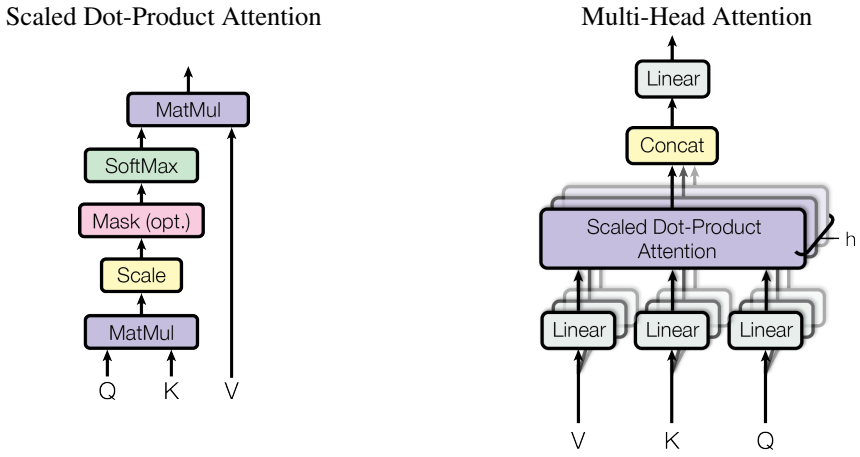

FIGURE 2.7: **The two main components of the Transformer.** On the left the attention module that calculate the attention function (see text). Q, K and V are the query, key and value matrices. This module is responsible for weighing the importance of the elements of the input sequence (value) according to the element requested in the output sequence (query). On the right, the multi-head attention which combines multiple attention functions (heads) in order to focus on different representations of the same input (due to the different learned linear mapping for each head). (Vaswani et al., 2017).

An *attention function* can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key[2].
The particular attention used in Transformer is called *Scaled Dot-Product Attention* (see Figure 2.7). The input consists of queries and keys of dimension d_k, and values of dimension d_v.
In practice, the attention function is computed simultaneously on a set of queries, packed together into a matrix Q. The keys and values are also packed together into matrices K and V. The attention function is:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{2.12}$$

Instead of performing a single attention function, the Transformer linearly project the Q, K and V matrices into other representations and then compute the attention function for every representation, in parallel (see Figure 2.7). This mechanism is called *multi-head attention* (MHA), in which every attention function and the relative input representation are a single *head*.

The encoder maps a variable-length source sequence (a phrase or even an image, for example) to a fixed-length vector, and the decoder maps the vector representation back to a variable-length target sequence (Cho et al., 2014). The two networks are trained jointly to maximize the conditional probability of the target sequence given a source sequence. In the Transformer, those two components are a combination of MHA modules with other classical modules, such as simple linear layers and activation functions.

[2]Although in this thesis we are not dealing with language process, the nomenclature is very similar.

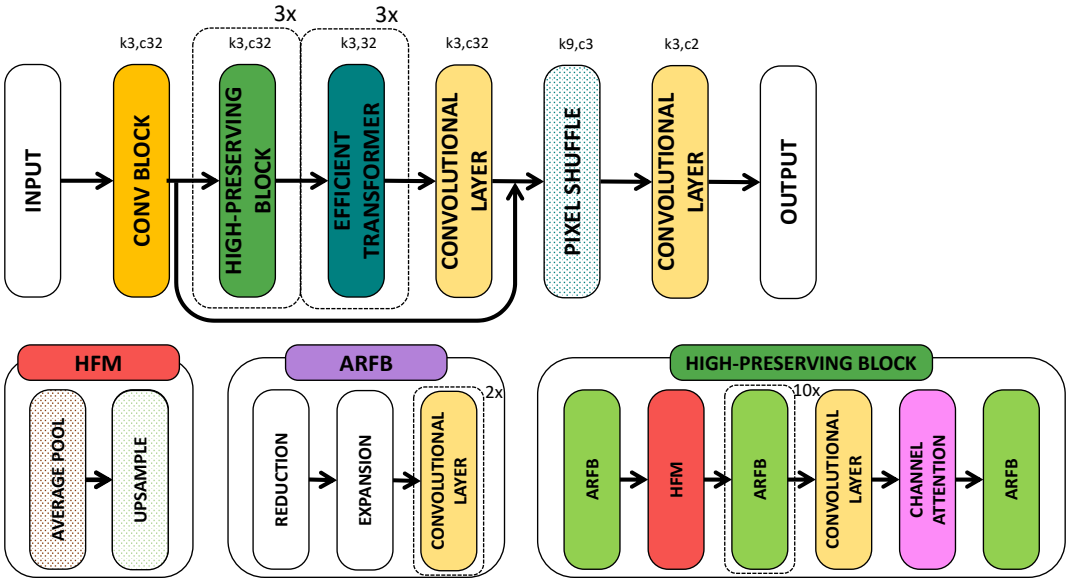

FIGURE 2.8: **ESRT architecture.** The smaller boxes on the bottom are the high-frequency module (HFM) and the adaptive residual feature block (ARFB) proposed in the original paper. This diagram represent a simplified version of the architecture in order to introduce and show the various novelties of the model, as well as for reasons of space. For the full version refer to Lu et al. (2021).

2.3.1 ESRT

The first attempt to apply the Transformer to a super resolution problem is the efficient super-resolution transformer (ESRT; Lu et al., 2021).
The key idea of Transformer is the *self-attention* mechanism, which can capture long-term information between sequence elements. By adapting Transformer to vision tasks, it showed very high-quality results in image recognition, object detection , and low-level image processing. Although in these tasks the Transformer achieved promising results, it always requires a lot of training data and needs very large memory to train, which is not suitable for practical applications. Hence, the attempt to find a more efficient vision-Transformer for SISR tasks.
The ESRT has by far the most complex architecture seen so far in this thesis (see Figure 2.8). We will try to briefly describe the main differences in this approach with respect to the Transformer, while ignoring the fine details of the model which are pedantic and, in our opinion, not necessary for the purpose of the thesis.

The first novelty is to adapt the Transformer to process images instead of words sequences. This is achieved with a pre and post-processing that avoid the classical method of turning a 2D image into a 1D sequence by sorting the pixels one by one. This approach is really simple but the resulting sequence would completely lose the local correlation of the image.
The workaround is to split the images into small non-overlapping patches that are then unfolded into 1D small sequences, as if they were the words of a sentence. The same procedure is reversed after the Transformer to obtain an image back.

The second contribution is the efficient transformer, which is a lighter version of the original Transformer, consisting only in the encoder module where the MHA modules are substituted with efficient MHA modules in which the attention function is calculated on sub-samples of channels of the input sequence.

By removing the decoder and the full channel attention mechanism this component is very light-weight if compared to the original Transformer.

The third and final main innovation is the focus on the reconstruction of the high-frequency details. This could be easily achievable with a Fourier transform but its implementation in a CNN is difficult and impractical, therefore it is proposed a differentiable high-frequency module. This is composed by an average pooling layer, which takes the input image and average its pixel values over patches of size $k \times k$, so that the resulting image has dimensions $(H/k, W/k)$. This is fed into an up-sample layer that reconstruct a smoother version of the original image, than this is subtracted, pixel-wise, from the original image, in order to obtain only the high-frequency details.

Although the ESRT claims to be mainly based on the Transformer, experiment showed that by completely removing this module the performance only drops from 32.18 dB to 31.96 dB using the PSNR as the metric, which, recalling the PSNR formula:

$$\text{PSNR} = 10 \cdot \log \left(\frac{(max_i)^2}{\mathcal{L}_2} \right) \tag{2.13}$$

Results in an absolute error ($\sqrt{\mathcal{L}_2}$) of roughly 31%. max_i in the above formula is the maximum of the pixel values in the image.
We argue that this shows that the efficient Transformer modules are not the main contributor to the ESRT performance which are mainly led by the deep CNN network.

The ESRT is the plastic example of how the complexity of this type of algorithms has exploded in recent years, without leading to an improvement in performance that justifies it.
Nevertheless, we want to explore the potentiality of such a novel architecture, especially in light of the reduced number of parameters, approximately between the FSRCNN and the ESRGAN.

The number of parameters in this model is roughly 750 thousands.

2.4 Complexity and scalability

In the previous Sections we saw some of the most promising algorithms to tackle the SISR task. Those three algorithms are the synthesis of three very different approaches.
The FSRCNN tries to keeps thing simple, optimizing the number of parameters to obtain the most efficient results.
The ESRGAN uses an enormous number of parameters, distributed in two competing networks which, through *guard-thief* game, improve the perceptual quality of the images at the expense of precision.
Finally, the ESRT employs the attention mechanism to model the long-term dependence between similar local regions in the input image and use this information to guide the reconstruction.

We can compare those three networks, along with other state-of-the-art models, according with their performance, their size (namely their parameters, which directly impact the memory usage) and the amount of operations (which is proxy for the training and inference time).
Since most of the operations in these CNN-based networks are multiplications and additions between matrices, we will use the multi-adds number to estimate their computational complexity.

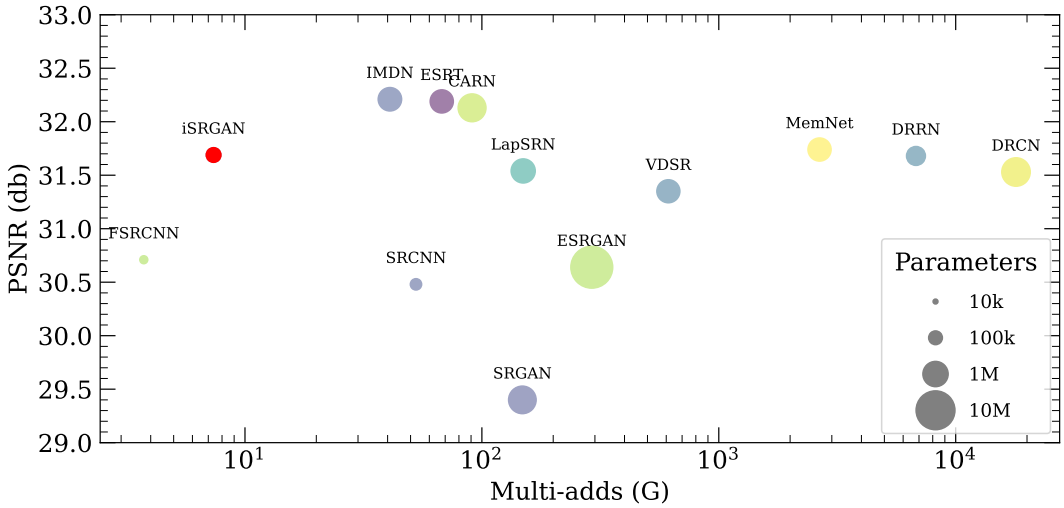

FIGURE 2.9: **Comparison of performance against complexity of several state-of-the-art SISR algorithms.** The bubble plot depicts the algorithms according to three different metrics: *i*) the performance, measured with the peak-signal-to-noise-ratio (y-axis); *ii*) the computational complexity, measured in number of billions of additions and multiplications between matrices (x-axis); *iii*) the dimension of the networks, i.e. the parameters count, depicted as the size of the bubbles. Our proposed model is depicted in red. The other colors are for decorative purposes. The number of Multi-adds is calculated assuming a 128×128 image input, for the sake of comparison.

As a measure of the performances we are going to use the peak-signal-to-noise ratio since it is the most objective and widespreadly used of all the measures commonly found in the literature.
In addition, ESRGAN in particular proved that this type of metric (enforced in the network by using a $\mathcal{L}_2$ loss) is to be preferred to obtain more reliable results, although less sharp and detailed. We will see in Chapter 3 that for our tests we do not need high-frequency details, since our images are far more homogeneous and continuous than the real-world images considered in all the previously considered models. Therefore, PSNR is the perfect choice for our case.
For similar reasons we choose one of simplest dataset available to compare the state-of-the-art architectures, the Set5 (Bevilacqua et al., 2012). Again, our simulated images are going to be very simple compared to real-world images (namely faces, animals, vehicles and so on). Thus, the best playground is the simplest dataset possible, without the need to investigate the performance of those algorithms on the hardest datasets.

We can see the result of our comparison in Figure 2.9.
At a glance, we can notice how the small and fast FSRCNN performs as good as the 16 millions parameters ESRGAN. This tells us that for particularly simple reconstruction tasks a complex loss function is not only unhelpful but can be harmful.

Readily, we can also see how the ESRT performs really well, with a modest amount of parameters and a relatively fast training/inference time. But we have already seen how intricate and unintuitive its architecture is.
Using this architecture as the baseline is impractical since small tweaks in its structure can lead to drastically different results.

In our opinion, the simplicity and the interpretability of the FSRCNN are added

values which moves the balance in favor of this model.
Moreover, the five-years difference in the deep learning know-how between the FSRCNN and the much more recent ESRT means that the former's room for improvement is very large.
Finally, given the extreme lightness of the network we are able to test several hundred different variations of this network in a reasonable time, to find the one that best suited our purpose (see Chapter 5).
This is why FSRCNN is the best baseline to improve on.

Chapter 3

Galaxies simulations

In order to test the effectiveness and applicability of our method in a real scenario we make use of a pipeline for the estimation of galaxy parameters presented in Rigamonti et al. (2022).
The authors have developed an algorithm that can retrieves the physical parameters of a target galaxy by applying a fully Bayesian estimation method on the global structure of the object.
The novelties of this method can be explained starting from the decomposition of the target galaxies in two (or more) components, the disc and the bulge.
This framework is not new and since the first structural studies of galaxies the *bulge+disk* decomposition was a common practice. Bulges were commonly thought to be spheroidal components with little net rotation. They could be well described by the de Vaucouleurs surface brightness profile (de Vaucouleurs, 1948).
The disk, on the other hand, is thought to be formed by a combination of angular momentum conservation and gas accretion and the cooling in dark matter (DM) halos which is typically described with a flat exponential profile (Freeman, 1970).
In recent years, however, new observations revealed that this decomposition is an oversimplification of the reality. In particular, bulges are now subdivided into *classical* bulges, indistinguishable from ellipticals galaxies, and *pseudo* bulges, which are the result of secular processes (Kormendy and Kennicutt, 2004). The *pseudo* bulges often show a disc + bars, thus greatly complicating the *disc+bulge* framework.
Moreover, in the last decade, the advent of integral field spectroscopy (IFS) units allowed the observation of the velocity field of galaxies, along with the surface brightness profile and the estimate of the mass-to-light ratio.
From the first information we can derive the velocity and dispersion velocity of the galaxy's component along the line of sight.
Inspired by the new IFS data a new structural framework was born, which is related to the dynamics of the galaxy. In this case, the decomposition of a galaxy is meant to reproduce both the measured line-of-sight velocity distribution at any projected two-dimensional resolution element and the measured surface brightness (which is related to the stellar mass surface density by mean of the stellar Mass to light ratio).
The drawback of these methods is that the calculation of the orbital families is computationally infeasible, especially due to the dependence on the assumed gravitational potential. A common workaround is to fix the potential of the visible component at the beginning of the procedure and then it is left unchanged.

In Rigamonti et al. (2022) it is proposed a novel approach which models simultaneously all the available data (photometry+kinematics). takes the best of the photometric approach and the dynamics-based one. As in the former, the method starts with the assumption that the target galaxy can be decomposed into simple components, i.e. the bulge plus one or more discs. These components are described by mean of standard analytical potential-density profiles and characterized by several

free parameters which are then optimized during the fitting procedure. However, their kinematics properties don't depend on a gravitational potential fixed a priori, but they are fitted to reproduce the kinematic and photometric data .

Given the large parameter space (18^{th} dimensional) the fitting procedure requires a fully Bayesian approach in order to assure optimal convergence. This has been accomplished through a nested sampling algorithm (Skilling, 2006) which iteratively explore the parameters space until a user-defined converges criterion is reached. The computational burden of the nested-sampling approach is lightened through the utilization of graphic processing units (GPUs) in the model construction, resulting in an overall speed up of a factor 100.

3.1 Galaxy model

Each galaxy in the framework presented in Rigamonti et al. (2022) is a superposition of a spherical bulge, two exponential razor-thin discs and a dark-matter halo. Each galaxy model has a well defined geometrical centre, that corresponds to both the photometric and dynamical centre, and it is axially symmetric. They are also assumed three independent mass-to-light ratios, which is the fraction of the mass and the luminosity of each of the visible components and its luminosity, one for the bulge and one for each of the two discs.

The two discs, which are axially symmetric, are associated with two other parameters that describe the inclination of the common axis. Each galaxy model, as anticipated before, relies on several simplifying assumption such as isotropic velocity dispersion, spherical symmetry (bulge) and negligible thickness (discs), which are mandatory in order to keep the computational cost of each model construction on reasonable timescales. The simplified nature of the model still catches all the relevant physics and it should be considered as a first step towards a complete description of galaxies.

More specifically the bulge is a spherically symmetric component with an isotropic velocity distribution. The density profile is assumed to be the Dehnen profile (Dehnen, 1993), for which the three-dimensional density is given by:

$$\rho_b(r) = \frac{(3-\gamma)M_b}{4\pi}\frac{R_b}{r^\gamma(r+R_b)^{4-\gamma}} \tag{3.1}$$

where M_b is the total mass of the bulge and R_b is the scale radius of the bulge; γ is a free parameter.

The corresponding potential can be calculated from the Poisson's equation:

$$\Phi_b(r) = -\frac{GM_b}{R_b(2-\gamma)}\left[1-\left(\frac{r}{r+R_b}\right)^{2-\gamma}\right] \tag{3.2}$$

For $\gamma \neq 2$. For $\gamma < 2$ the potential has a finite central value of $(2-\gamma)^{-1}GM_b/R_b$.

The circular velocity due to the gravitational potential generate by the Dehnen density profile is therefore:

$$v^2_{\mathrm{circ},b}(r) = \frac{GM_b r^{2-\gamma}}{(r+R_b)^{3-\gamma}} \tag{3.3}$$

The intrinsic radial velocity dispersion can be calculated from the Jeans equation as:

$$\bar{v_b^2}(r) = GMr^\gamma (r + R_b)^{4-\gamma} \int_r^\infty \frac{r'^{1-2\gamma} dr'}{(r' + R_b)^{7-2\gamma}} \tag{3.4}$$

which can be calculated analytically only if 4γ is an integer. Again, for $\gamma < 2$ the circular velocities and the velocity dispersion are finite in the centre.

The actual observables of the model are the projected two-dimensional quantities along the line-of-sight. Note that the bulge is assumed to be isolated [1] such that the projected quantities can be computed by solving analytically the Jeans equations. In particular we have that the surface density is:

$$\Sigma_b(R) = 2 \int_R^\infty \frac{\rho(r) r}{\sqrt{r^2 - R^2}} dr \tag{3.5}$$

Where R is the projected distance from the centre. This Equation must be solved numerically for a non-integer γ.

The projected velocity dispersion is instead:

$$\sigma_b^2(R) = \frac{(3-\gamma) G M_b^2 R_b^3}{2\pi \Sigma(R)} \int^\infty R \frac{r^{1-2\gamma}}{(r + R_b)^{7-2\gamma}} \sqrt{r^2 - R^2} dr \tag{3.6}$$

Finally, the projected line of sight can be computed as:

$$v_{b,\perp}(R) = 4\pi G M_b \int_R^{r_\parallel} f \left(\Psi(r) - \frac{v_\parallel^2}{2} \right) \frac{r^{1-\gamma} \sqrt{r^2 - R^2}}{(r + R_b)^{3-\gamma}} dr \tag{3.7}$$

where $r_\parallel$ is the radius where $2\Psi(r_\parallel) = v_\parallel^2; \Psi = -\Phi(r) R_b / (G M_b)$ is the adimensional positive potential

The dark matter halo is described by a Navarro-Frenk-White (NFW) profile (Navarro, Frenk, and White, 1996). Therefore, the three-dimensional DM halo density profile in the model can be computed as:

$$\rho_h(r) = \frac{\rho_0}{\frac{r}{R_h} \left(1 + \frac{r}{R_h}\right)^2} \tag{3.8}$$

where R_h is a scale radius, and ρ_0 is a normalization constant. The cumulative mass that can be inferred from this profile diverges, but it is common to truncate the integral at the virial radius:

$$R_{vir} = c R_h \tag{3.9}$$

where c is the so-called concentration parameter.

The circular velocity of the DM halo is calculated as follows:

$$\left(\frac{v_{\text{circ},h}}{v_{vir}} \right)^2 = \frac{1}{x} \frac{\ln(1 + cx) - cx/(1 + cx)}{\ln(1 + c) - c/(1 + c)} \tag{3.10}$$

where $x \equiv r / r_{vir}$ is the radius in units of the virial radius, similarly v_{vir} is the virial velocity.

[1] Note that this assumption is suited for compact systems.

Another defining parameter to describe the halo component is the halo-to-stars mass ratio M_h/M_*

The discs are described by an exponential profile. The assumption of negligible thickness facilitates both the computation of the circular velocity in the equatorial plane and the projection along the line-of-sight. The two discs are characterized by four parameters each: the total mass M_d, the scale radius R_d, the mass-to-light ratio $(M/L)_d$ and finally the fraction of kinetic energy involved in a circular rotation motion k. For example, $k = 1$ implies an orderly rotating disc and $k = 0$ means that the average velocity is zero and the only non-zero component is the velocity dispersion.

The projected surface density is computed as:

$$\Sigma_d(r) = \frac{M_d}{2\pi R_d^2 \cos i} exp(-\frac{r}{R_d}) \tag{3.11}$$

Where i is the inclination angle with respect to the line of sight.
The circular velocity is instead:

$$v_{\mathrm{circ},d}(r) = \sqrt{\frac{M_d}{2R_d^2} y^2 \left[I_0(y)K_0(y) - I_1(y)K_1(y)\right]} \tag{3.12}$$

Where $y = r/R_d$; $I_0,\ I_1,\ K_0,\ K_1$ are the modified Bessel functions of first and second kind.
Then, we can calculate the intrinsic bulk tangential velocity as:

$$v_{\mathrm{int},d}(r) = \sqrt{v^2_{\mathrm{circ},b}(r) + v^2_{\mathrm{circ},d}(r) + v^2_{\mathrm{circ},d}(r) + v^2_{\mathrm{circ},h}(r)} \tag{3.13}$$

Finally, the bulk line-of-sight velocity component at any location in the plane of the sky R is:

$$v_d(R) = -k \sin i \cos\phi v_{\mathrm{int},d}(r) \tag{3.14}$$

where ϕ is the azimuthal angle evaluated in the plane of the disc from the major axis of the projected disc.
And the line-of-sight velocity dispersion of the disc is instead:

$$\sigma^2_{d,j}(R) = (1 - k^2)\frac{v^2_{\mathrm{int},d}(r)}{3} \tag{3.15}$$

where the factor 3 at the denominator is due to the assumption of an isotropic velocity dispersion.

The observables of these simulations can be estimated combining the previous equations. The total surface brightness reads:

$$\begin{aligned} B_{\mathrm{tot}}(\mathbf{R}) &= B_b(\mathbf{R}) + B_{d,1}(\mathbf{R}) + B_{d,2}(\mathbf{R}) \\ &= \left(\frac{M}{L}^{-1}_{b}\right)\Sigma_b(\mathbf{R}) + \left(\frac{M}{L}^{-1}_{d,1}\right)\Sigma_{d,1}(\mathbf{R}) + \left(\frac{M}{L}^{-1}_{d,2}\right)\Sigma_{d,2}(\mathbf{R}) \end{aligned} \tag{3.16}$$

Where Σ represents the projected densities profile of each component, M/L is the constant mass-to-light ratios for each components and $\mathbf{R}$ is the projected distance from the galaxy center.
The line of sight velocity is the average of the velocities weighted on each surface

name	description
$(x_0,\ y_0)$	geometrical centre coordinates
ϕ	position angle
i	inclination angle
$(M_b,\ R_b)$	bulge's mass and scale radius
$(M_{d,1},\ R_{d,1})$	inner disc's mass and scale radius
$(M_{d,2},\ R_{d,2})$	outer disc's mass and scale radius
$(M_h,\ R_h)$	DM halo's mass and scale radius
$(M/L)_b$	bulge's mass-to-light ratio
$(M/L)_{d,1}$	inner disc's mass-to-light ratio
$(M/L)_{d,2}$	outer disc's mass-to-light ratio
k_1	inner disc's kinematical decomposition
k_2	outer disc's kinematical decomposition
γ	Dehnen parameter

TABLE 3.1: **Summary of the model parameters.**

brightness:

$$v(\mathbf{R}) = \frac{B_{d,1}(\mathbf{R})v_{d,1}(\mathbf{R}) + B_{d,2}(\mathbf{R})v_{d,2}(\mathbf{R})}{B_b(\mathbf{R}) + B_{d,1}(\mathbf{R}) + B_{d,2}(\mathbf{R})} \tag{3.17}$$

The bulge contribution is not included due to its isotropic velocity field.
Similarly, the projected velocity dispersion is computed as:

$$\sigma^2(R) = \frac{B_b(R)\sigma_b^2(R)B_{d,1}(\mathbf{R})\sigma_{d,1}^2(\mathbf{R}) + B_{d,2}(\mathbf{R})\sigma_{d,2}^2(\mathbf{R})}{B_b(\mathbf{R}) + B_{d,1}(\mathbf{R}) + B_{d,2}(\mathbf{R})} \tag{3.18}$$

The fourth component is instead related to the average mass-to-light ratio of the galaxy, which can be inferred from its inverse:

$$\langle L/M \rangle = \frac{\left(\frac{M}{L}_b^{-1}\right)\Sigma_b + \left(\frac{M}{L}_{d,1}^{-1}\right)\Sigma_{d,1} + \left(\frac{M}{L}_{d,2}^{-1}\right)\Sigma_{d,2}}{\Sigma_b + \Sigma_{d,1} + \Sigma_{d,2}} \tag{3.19}$$

During the parameter estimation those four observables are simultaneously compared to the data in order to estimate all the 18 parameters reported in table 3.1. These 4 quantities, which can be effectively represented as 4 channel images, are the main focus of our super-resolution task.

3.2 Parameter estimation

With such a large number of independent parameters any standard fitting routine is doomed to fail as a consequence of the presence of local minima and strong correlation between parameters.
The optimal choice for exploring the entirety of the parameters space in order to give an estimate of the n-dimensional probability distribution of the parameters is the Bayesian inference, more precisely the adopted algorithm is the nested sampling.

The likelihood function determined by the inference is composed by four components, one for each map:

$$\log\mathcal{L} = \log\mathcal{L}_B + \log\mathcal{L}_v + \log\mathcal{L}_\sigma + \log\mathcal{L}_{M/L} \tag{3.20}$$

such that:

$$p(\Omega|data) = \frac{\mathcal{L}(data|\Omega) \times p(\Omega)}{Z(data)} \tag{3.21}$$

where Ω is the parameter vector, $p(\Omega|data)$ is the posterior probability distribution, $\mathcal{L}$ is the likelihood function, $p(\Omega)$ is the prior distribution and $Z(data)$ is the *evidence*.

The prior distribution is usually assumed to be uniform or log-uniform for most of the parameters. The choice of the prior reflects the boundaries adopted in building our dataset as we will see in the following paragraph (see Table 3.1).

The terms in Equation 3.20 are directly related to the observables described in Equations 3.16, 3.17, 3.18 and 3.19.

At each iteration, a new set of parameters is extracted from the parameter space and the corresponding high-resolution simulation of a mock galaxy is generated. The simulation is then fitted to the real galaxy and the likelihood calculated to proceed for the next iteration (see an example of best fit model in Figure 3.1).
Our proposed method aims to speed-up every single iteration, by letting the model produce just a low-resolution simulation and then scale it up to the desired resolution using the machine learning algorithm.

3.3 The sample

The sample created to train and test our model is composed by 1.5 million mock galaxies. For each of them a complete simulation is calculated on two numerical grids with different resolutions: the low-resolution (LR) simulation 20×20 pixels, and the high-resolution (HR) one 80×80 pixels. Each simulation is therefore represented by an image 20×20 or 80×80 pixels with four channels each, representing the four components of the model.

In order to produce a representative sample the whole search space is used to reproduce the known galaxies in the current surveys.
The search space is detailed in Table 3.2. Note that the DM halo's mass and scale radius are given as fractions of the total mass and total scale radius, respectively.

The boundaries are chosen accordingly to the commonly used priors in nested sampling.
The parameter space is sample using the so-called latin hypercube sampling (LHS), a statistical method for generating a near-random sample of parameter values from a multidimensional distribution (McKay, Beckman, and Conover, 1979). The prior distributions are mainly uniform in linear scale, this also applies for the masses and the scale radii.

In order to produce simulations that are as similar to real observed galaxies as possible, the high resolution images are convoled with a point spread function (PSF). The chosen PSF is the average PSF in the Mapping Nearby Galaxies at APO survey (MaNGA; Bundy et al., 2015) that contains roughly $\sim$ 10000 galaxies with redshift around $z \sim 0.03$ and thanks to its integral field unit (IFU; Drory et al., 2015) can produce velocities maps of the observed objects. This is, to date, the largest sample of galaxies observed with an IFU and is the perfect survey to test this method.
In particular, the FWHM of the average PSF for the velocity maps is $psf = 2.5''$ and $psf = 1.3''$ for the photometric images.
Finally, the dataset is created with an average redshift of $z \sim 0.03$, in order to be consistent with the MaNGA survey, and assuming a flat ΛCDM cosmology with $H_0 = 70$ km s^{-1}Mpc^{-1}, $\Omega_\lambda = 0.7$ and $\Omega_M = 0.3$.

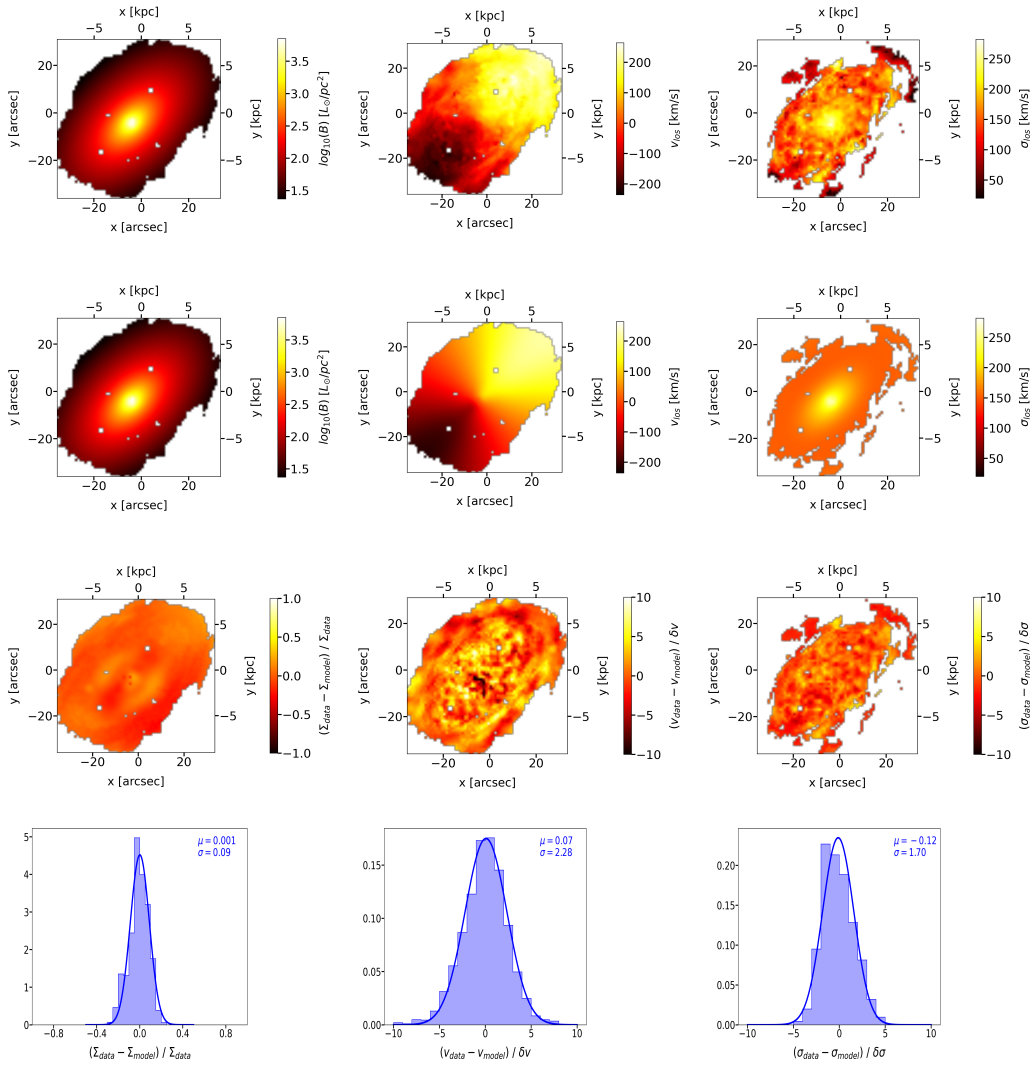

FIGURE 3.1: **Best fit model of the NGC 7683 galaxy.** The picture represent a successful fit of a real galaxy. The first, the second and the third columns represent the surface brightness, the line of sight velocity and the line of sight velocity dispersion, respectively. From top to bottom we can see the observational data, the best fit model, the residual maps and the histograms of the residuals, with a Gaussian fit over-imposed. (Image from Rigamonti et al., 2022)

Parameter	Range
$x_0\ (kpc)$	$[-2, 2]$
$y_0\ (kpc)$	$[-2, 2]$
ϕ	$[-\pi, \pi$
i	$[0.1, \pi/3]$
$\log_{10} M_b\ (M_\odot)$	$[8.5, 12]$
$\log_{10} R_b\ (kpc)$	$[-2, 1.5]$
$\log_{10} M_{d,1}\ (M_\odot)$	$[8.5, 12]$
$\log_{10} R_{d,1}\ (kpc)$	$[-2, 1.5]$
$\log_{10} M_{d,2}\ (M_\odot)$	$[8.5, 12]$
$\log_{10} R_{d,2}\ (kpc)$	$[-2, 1.5]$
$(M/L)_b$	$[0.1, 20]$
$(M/L)_{d,1}$	$[0.1, 20]$
$(M/L)_{d,2}$	$[0.1, 20]$
k_1	$[0.01, 1]$
k_2	$[0.01, 1]$
$\log_{10}(M_h/M_{tot})$	$[1, 2.7]$
R_h/R_{tot}	$[5, 15]$

TABLE 3.2: **Search space explored to build the dataset.**

3.4 Dataset preparation

In the dataset, some of the low resolution maps show some pixel outliers. This phenomenon is due to numerical instabilities, especially near the centre of the galaxy, where the density profile can sometime diverge. The workaround adopted in the construction of the dataset is to compute the properties of the central pixel as an average over the entire pixel area as opposed to all the other positions where the observables are computed point-wise.
We discovered that this procedure may underestimate some properties in the very center of each galaxy especially in cases of systems with flat inner slope ($\gamma < 1$). Nevertheless, we do not need to correct for this artifact since these pixels don't seem to affect the results.
As we will see in Section 6.5 our super resolution neural network is very robust to outlier leading to astonishing performances that could be furthermore enhanced by using a training set without outliers.

Chapter 4

Architecture and training

As previously detailed in Chapter 2 we decided to explore three different kind of architectures of increasing complexity and dimensions.
The first one, FSRCNN (Dong, Loy, and Tang, 2016), is the simplest. This deep convolutional network proved how even with a small amount of parameters and complexity it is possible to achieve good quality SR images. It pioneered the idea of an end-to-end learnable map between the LR image and the HR image by introducing a transposed convolutional layer (see Section 1.7) at the end of the network to both up-sample the image and reconstruct it. Finally, it showed that the non-linear mapping can be performed in a feature space with few dimensions, thus greatly reducing the number of parameters while preserving the performance.

The second one, ESRGAN (Wang et al., 2018) is the deepest and the heaviest of the three, but also the most groundbreaking. This architecture is built on the idea that has revolutionized the SR field by employing a new type of high-level loss function that focuses on the perceptual quality rather then the perfect accuracy. However, the huge number of parameters, which is one of the main characteristics of this architecture, is also why it fails when it comes training and inference velocity.

The last one is the ESRT (Lu et al., 2021), the most complex and creative. This architecture tries to make use of a successful natural language model (the Transformer; Vaswani et al., 2017) to solve the SR task with a similar approach. By treating the patches of the LR image as *words* in a sentence the algorithm should be able to *translate* it into another sequence of *words* of an output sentence of arbitrary length (and therefore, an image with an arbitrary resolution). This solution however has proven to under-perform not only the ESRGAN, which is expected, but also the small FSRCNN, by introducing some undesirable artifacts in the output image.

On the contrary FSRCNN and ESRGAN performed quite similarly on the specific simple task of reconstructing the LR simulated galaxies, but the former can be trained in a fraction of the time it takes for the latter, and, more importantly, its inference time is roughly 100 times better.

4.1 iSRCNN

The proposed architecture can be seen as the direct evolution of the FSRCNN. It preserves the five main stages: feature extraction, channels shrinkage, non-linear mapping, channels expansion and finally the up-sampling and the reconstruction (see Figure 4.1).
The first noticeable improvement is the presence of two transposed convolutional layers at the end of the network. In fact, during the design phase we noticed that, regardless of the number of layers, the network was unable to reconstruct the finest details of the HR image. We hypothesized that this could be a consequence of the fact

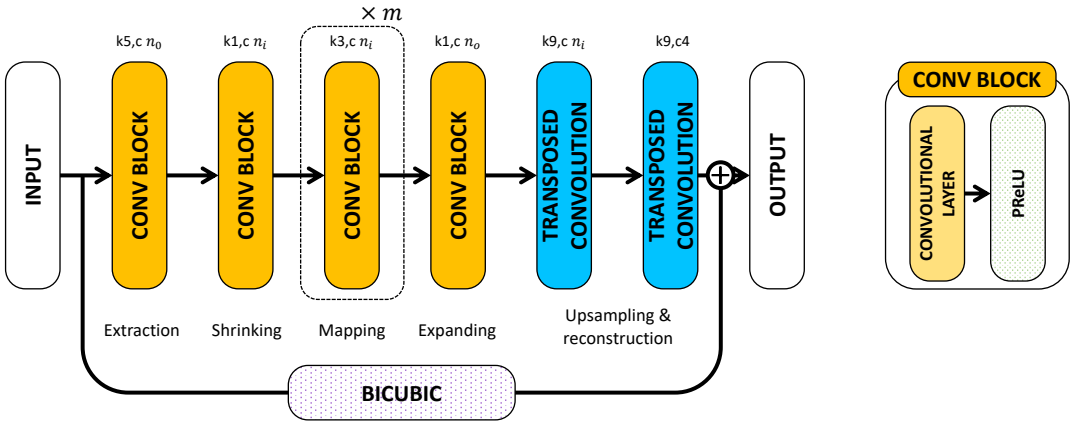

FIGURE 4.1: **Architecture of the proposed iSRCNN network.**

that each pixel of the feature maps, before the up-sampling, contains the prior information for the reconstruction of 16 pixels in the final image (for a 4× up-sampling). This extreme degeneracy can be mitigated by applying two sequential up-sampling (2× each) in such a way that for every four pixels in the downstream layer there is one pixel in upstream one. The results proved that this intuition was correct (see Section 4.2 for the ablation studies).

The second obvious evolution is the presence of a *skip-connection* (the path that partially protrudes from the network in Figure 4.1). This component allows the information to partially flow outside the network by skipping, in practice, some layers. This is the fundamental innovation brought by residual networks (see Section 1.8). Their computational footprint is negligible[1] (it is just a matrices summation) but the contribution of skip-connections to the overall stability and performance of the networks are invaluable and it is widely motivated in the literature (e.g Drozdzal et al., 2016; Orhan and Pitkow, 2017; Tong et al., 2017).
However, skip-connections are not the panacea and should be used with caution. In fact, we empirically proved that they can degrade performance if used across all layers of the network.

The third component is not actually an innovation as it was already present in the original version of the SRCNN, then removed in the FSRCNN. We are talking about the bicubic up-sampling.
The novelty is that this up-sampled image is not directly fed to the network. This in fact would result in a dizzying increase in the number of parameters which is the main reason why it was removed in the first place. In our case the pre-up-sampled image never enters the actual network and serves only as a basis for *residual learning*. In other words the network learns to reconstruct the pixel-wise difference between the bicubic up-sampled image and the LR one. This greatly helps a rapid convergence and, most importantly, it prevents the onset of artifacts by guiding the initial stage of reconstruction.

The last major contribution is the implementation of a systematic method for finding the best model named *architecture search*. This procedure is based on a search strategy that selects an architecture $a \in \mathcal{A}$ where $\mathcal{A}$ is a predefined search space (see Elsken, Metzen, and Hutter, 2019 for a recent review on the topic).

[1] Although this holds true in general, a huge amount of skip-connections may lead to a large memory consumption since every upstream layer must be saved in memory before the summation. This is especially true in *dense* networks in which every layer is connected to all the others (see Pelt and Sethian, 2018).

In the past, the choice of the hyperparameters that controls the architecture structure (e.g. the number of layers of a certain block or the number of channels in a convolutional layers) are chosen by hand, based on experience or heuristics. This was certainly necessary when computing resources were scarce but today it is possible to test various families of models and select the best ones based on certain metrics.
In particular, since our architecture is already very simple we use the loss function as a metric for the search.
We explore different values for the number of output channels in the Extraction and Expanding modules, the channels within the Shrinking and Mapping modules and finally the number of maps in the Mapping layer, respectively n_0, n_i and m in Figure 4.1.
The search space, the search strategy and the performance comparisons are detailed in Section 5.5.
The results of the hyperparameters search is $n_0 = 60$, $n_i = 18$ and $m = 8$. Thus, it seems that for obtaining a better performance a deeper structure is required. Although this increment in the number and dimension of layers resulted in a doubling of the number of parameters ($\sim 120k$ instead of $\sim 70k$ for a 4-channel input) compared to the baseline ($n_0 = 56$, $n_i = 12$ and $m = 4$, see FSRCNN), the performance improved by roughly 35% (see Table 4.1).

All of the aforementioned improvements are tested to assess their relative contribution to the final performance of the network in Section 4.2.

The computational complexity of the network can be summarized as follows:

$$O\{(4 \cdot 5 \cdot 5n_o + n_o n_i + 3 \cdot 3mn_i^2 + n_i n_o + 9 \cdot 9n_o n_i + 9 \cdot 9 \cdot 4n_i)S_{LR}\} \tag{4.1}$$

where S_{LR} is the size of the LR image in pixels.
In Equation 4.1 we have made explicit every single block and the kernel size within them. By simplifying the equation we obtain:

$$O\{(100n_0 + 83n_0 n_i + 9mn_i^2 + 324n_i)S_{LR}\} \tag{4.2}$$

4.2 Ablation Study

In this Section we calculate the impact of the various improvements presented in the previous Section:

- The bicubic up-sampling on the skip-connection.
- The second transpose convolution in the reconstruction block.
- The optimization of the channels and the number of maps within the network.

For the comparison we consider every combination of those components (six in total) in order to assess two main metrics: the validation loss and the reconstruction time for a single image. A good model is one that can minimize the validation loss while keeping the inference time low.
The test is conducted with the following hyperparameters (see Sections 5.5 and 6.2 for details):

Bi	Tr	GO	Val. Loss	Inference Time (ms)
			0.0057 (+338%)	0.60 (-76%)
		✓	0.0027 (+108%)	0.75 (-70%)
	✓		0.0031 (+238%)	1.70 (-32%)
	✓	✓	0.0029 (+223%)	1.90 (-24%)
✓			0.0022 (+69%)	0.78 (-69%)
✓		✓	0.0052 (+300%)	1.06 (-58%)
✓	✓		0.0018 (+38%)	1.50 (-40%)
✓	✓	✓	0.0013	2.50

TABLE 4.1: **Ablation studies results.** The table shows the results in terms of validation loss and inference time on a single image of eight different networks with different components combinations. Bi is the bicubic up-sampling on the skip-connection. Tr is the second transpose layer in the reconstruction block. GO is the global optimization of the hyperparameters (see Section 5.5). The percentages in parenthesis represent the relative changes with respect to the baseline with all the three components enabled (less is better).

Number of epochs	400
Learning rate	0.0035
Batch size	32
β_1	0.957
β_2	0.929
LR reduction patience	10
LR reduction factor	2

The results of this study are summarized in Table 4.1. The inference time reported in the Table is estimated using a single core of an Intel 11200K CPU.
For reference, the full high-resolution simulation takes $0.3s$ to calculate the high resolution maps on the same CPU.
The iSRCNN model is roughly 100 times faster.

It is noteworthy that the architecture without all improvements is basically the original FSRCNN, slightly adapted to receive four channels in inputs instead of one. In this way we can directly compare to the original model and acknowledge that the increase in performance is roughly 77% (the original FSRCNN performs 338% worse than our iSRCNN).
Although it is true that the original FSRCNN, due to its fewer parameters, has an inference time which is a quarter of our iSRCNN, the precision requirements for the reconstruction task of a physical simulation are just not met by the former architecture.

We can see from Table 4.1 how every single component that we introduced in the architecture contribute to the overall reconstruction precision. In particular, the most impactful addition seems to be the bicubic up-sampling on the skip-connection (reported as *Bi* in the Table), with an average absolute error that is 61% better than the architecture without it. Moreover, this addition has a very small impact on the inference time. We refer to this version of the network as the *faster* iSRCNN.

We see another possible trade-off between inference time and precision in the version of the iSRCNN without the bicubic up-sampling and with a single transposed layer (reported as *Bi* in the Table). This variation performs 3 times better than

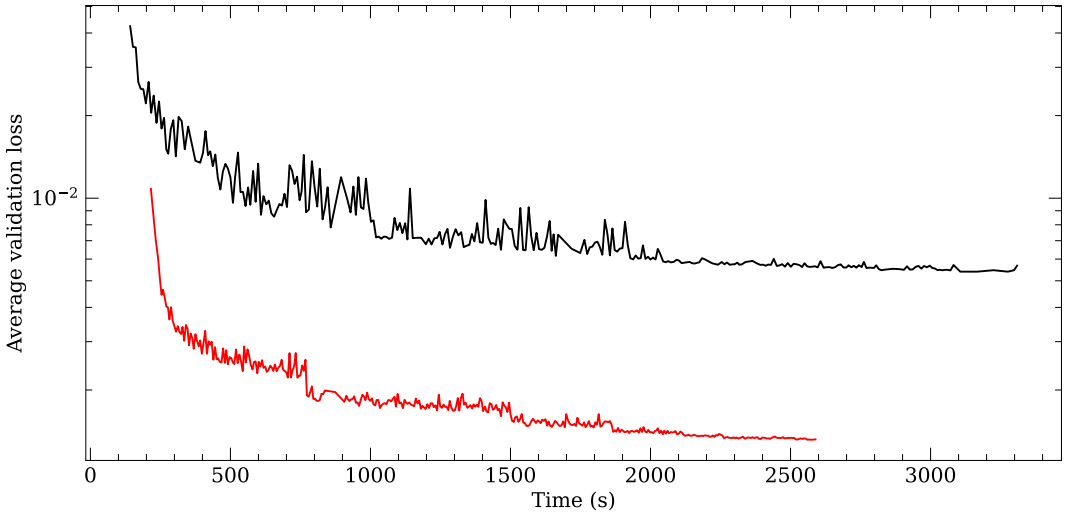

FIGURE 4.2: **Performance comparison between iSRCNN (our proposed model) and the original FSRCNN.** The plot shows the validation loss as a function of time (although the two training runs shared the same number of epochs). The red line represents our proposed model while the black one represents the originale FSRCNN (Dong, Loy, and Tang, 2016). Our algorithm not only performs much better than its predecessor, but it also trains faster, regardless of the number of parameters.

the original FSRCNN with a similar inference time. We argue that, for some very demanding simulations, this solution could be preferable. However, in this thesis we want to demonstrate the accuracy of the machine learning approach in reconstructing physical simulations, therefore, our baseline for iSRCNN is the architecture with all the new features enables (best accuracy) and that in any case respects the inference time requirements (few milliseconds).

In addition to the features evaluated up to here, we have also tested the contribution of two additional methods: the dropout technique (e.g. Gal and Ghahramani, 2016) and the batch normalization (Ioffe and Szegedy, 2015). The former resulted in a much more stable loss function, but the performance are otherwise identical. Instead, the implementation of the latter always show a performance degradation, as already found in Wang et al. (2018).
For neither of them we considered it worthy to conduct further tests.

In Figure 4.2 we report the training run of our iSRCNN network compared to the run of the original FSRCNN.
For this comparison we have to adapt the original FSRCNN to accept 4-channels images. We also use the same global hyperparameters (i.e. learning rate, batch size and optimizer parameters) as long as the learning rate scheduler (see Section 6.2 for details) to ensure a fair comparison).
As we can clearly see from the plot, our network performs vastly better.
After only four minutes, the reconstruction performance of the iSRCNN is comparable to those of the original FSRCNN. After a full training run (400 epochs) the average absolute error between the super-resolution images and the ground truth images is more than 4 times lower, as we have already seen in Tabel 4.1.
In addition, the training of the iSRCNN seems to proceed faster, although the number of parameters to train is higher.

Chapter 5

Hyperparameters Tuning

We define *hyperparameters* the model's parameters that control the learning process. As opposed to the learnable parameters within the model, those special parameters are usually chosen by the model's designer using different heuristics developed during the decades. Some examples of hyperparameters can be simply the learning rate, the batch size, or the momentum, which are simple scalars that we could define as *global*[1] learning parameters. But hyperparameters can also involve only a part of the model structure, i.e. the number of neurons in a certain layer, the number of hidden layers of a certain type, the presence or absence of architectural features like skip connections, batch normalization, drop-out layers, and so on.
Usually, the choice of an optimal set of hyperparameters is model-dependent, but also dataset-dependent. This means that, for example, a certain batch size that works for a small dataset does not necessarily work well with a very large dataset. Or a certain learning rate may be optimal for a certain number of hidden layers but could cause severe instability in the learning process for deeper networks. Since hyperparameters also determine the structure of the architecture, this also implies that hyperparameters are strongly interdependent, or, in other words, underneath the optimization problem lies a complex non-linear function of the hyperparameters.
It goes without saying that a wise choice of the hyperparameters could make the difference between a successful training attempt or a failed one. However, wisdom and heuristics are not enough. As mentioned before, hyperparameter optimization (HPO) is commonly performed manually, via rules-of-thumb (e.g. Hsu, Chang, and Lin, 2003; Hinton, 2012). This approach is not robust, not reproducible and very impractical when the number of hyperparameters is large (e.g. Claesen et al., 2014). It would be nice to rely on an analytical, or at least systematic, procedure for the choice of these important *knobs*. Unfortunately, to date, we lack an underlying theory to justify the preference of a certain set of hyperparameters over other ones, the only thing to do is to carry out a more or less sophisticated systematic search within the space of possible hyperparameters. This allows a direct comparison between the performances of a sample of models backed by different hyperparameters sets and thus we can find the *best* set based on a certain metric, for the specific type of problem we are facing.

5.1 Naive methods

In this section we want to briefly describe what can be defined as *naive* hyperparameters search strategies: the grid search and the random search (RS). In the former

[1] In this context, this means that these scalars impact directly the whole learning process, for example by changing the scale of the step in gradient descent.

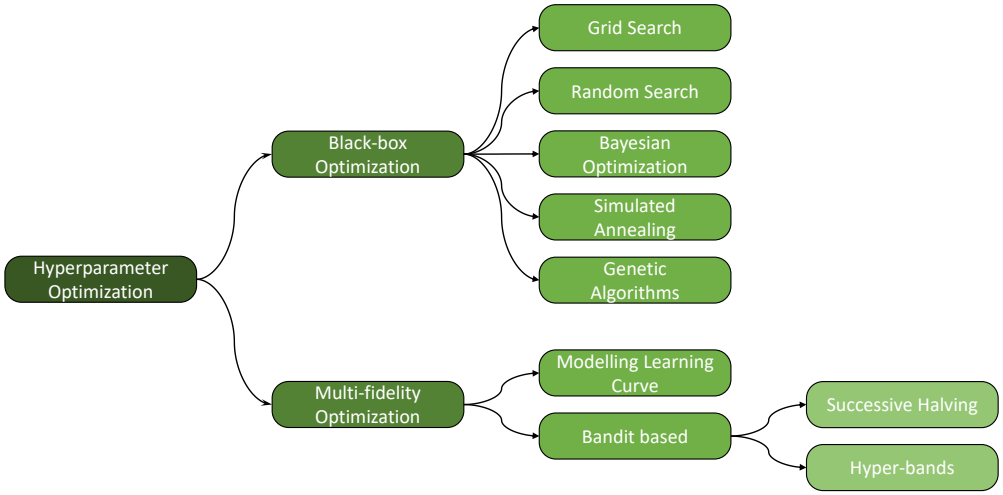

FIGURE 5.1: **Hyperparameters Optimization strategies.** In this diagram are represented some of the most popular hyperparameters optimization (HPO) methods. In this thesis we will focus on a combination of the Bayesian optimization and the hyper-bands methods. See text for details.

strategy, the designer of the HPO defines a search space as a bounded domain of hyperparameter values and the algorithm samples every points in that domain, with a certain grid resolution. The set of hyperparameters which gives highest accuracy is considered as best. This algorithm is guaranteed to find the best solution for every given search space. The latter, is similar to the first method but the algorithm samples random points in that domain. Grid search could be useful for checking combinations that are known to perform well. Random search is favorable for discovery new hyperparameter combinations. This approach is considered faster and more efficient although it may not converge to the best result.
RS algorithm represents the current baseline for HPO, but its downsides (shared with the grid search approach) are manifold:

- curse of dimensionality, leading to exponentially larger search spaces with increasing number of hyperparameters.
- extremely slow execution due to the brute-force approach
- Each objective function evaluation requires evaluating the performance of a model fully trained with one set of hyperparameters[2].
- completely ignores the relative importance of hyperparameters.

In order to overcome these problems a wide variety of optimization methods have been proposed (see Figure 5.1), from genetic algorithms (e.g Chen, Wang, and Lee, 2004; Tsai, Chou, and Liu, 2006; Loussaief and Abdelkrim, 2018) and simulated annealing (e.g Souza et al., 2009) to bandit based strategies (e.g.).
However, in the last years, the Bayesian optimization (BO; Shahriari et al., 2016) it is the algorithm that had the greatest traction in the literature.

[2]Depending on the architecture complexity this could require minutes, hours or even days.

5.2 Bayesian optimization

This family of algorithms is a state-of-the-art optimization framework for the global optimization of expensive *black-box*[3] functions, which has been successfully used for tuning deep neural networks for image classification (Snoek, Larochelle, and Adams, 2012; Snoek et al., 2015), speech recognition (Dahl, Sainath, and Hinton, 2013), and neural language modeling (Melis, Dyer, and Blunsom, 2017).
Bayesian optimization is an iterative algorithm with two key ingredients: a probabilistic surrogate model and an acquisition function to decide which point to evaluate next.
The function $f : \mathcal{H} \to \mathbb{R}$ that maps the hyperparameters of the model into the performance score of the model parameterized by the chosen set of hyperparameters. We can then say that, in each iteration i, BO uses the surrogate function, which is a probabilistic model $p(f|D)$ to model the objective function f based on a *prior*, the data points D in this case. The acquisition function $a : \mathcal{H} \to \mathbb{R}$ is based on the current state of the model (the surrogate function) and it trades off exploration and exploitation. The iteration proceeds as follows:

- Select the point to maximize a: $x_{new} = \arg\max_{x \in \mathcal{H}} a(\mathbf{x})$.
- Evaluate the objective function $y_{new} = f(x_{new}) + \epsilon$, where ϵ is normally distributed noise $\epsilon \sim \mathcal{N}(0, \sigma^2)$.
- augment the *prior* data with the new observation: $D \leftarrow D \cup (x_{new}, y_{new})$ and fit the model again.

This process is summarised in Figure 5.2.
A common choice for the acquisition function is the expected improvement over the currently observed value $\alpha = \min\{y_0, ..., y_n\}$:

$$a((x)) = \int \max(0, \alpha - f(\mathbf{x}))dp(f|D) \tag{5.1}$$

Compared to evaluating the expensive black-box function, the acquisition function is cheap to compute and can therefore be thoroughly optimized (Feurer and Hutter, 2019). Figure 5.2 illustrates BO optimizing a toy function.

BO was originally designed to optimize box-constrained, real-valued functions. However, for many machine learning hyperparameters, such as the learning rate in neural networks or regularization in support vector machines, it is common to optimize the exponent of an exponential term. Moreover, it is possible that certain hyperparameters are merely a truth condition (described with a boolean value), e.g. the presence of a certain hidden layer. Or again, a possible hyperparameter could be the activation function used in the model, which is in practice communicated to the algorithm as a string of characters. The classical gaussian-process-based BO algorithm fails to find the solution for these type of problems. To solve this aspect many solutions have been proposed (e.g. Nguyen et al., 2019), which usually involve an hybridization of the black-box Bayesian optimization with a bandit-based algorithm (more in the next section).
In addition, BO algorithms are poorly parallelizable, due to their intrinsic sequential nature.
The pros and cons of this algorithm are summarised in Table 5.1

[3]Here, *black-box* refers to the underlying model whose parameters the HPO tries to optimize.

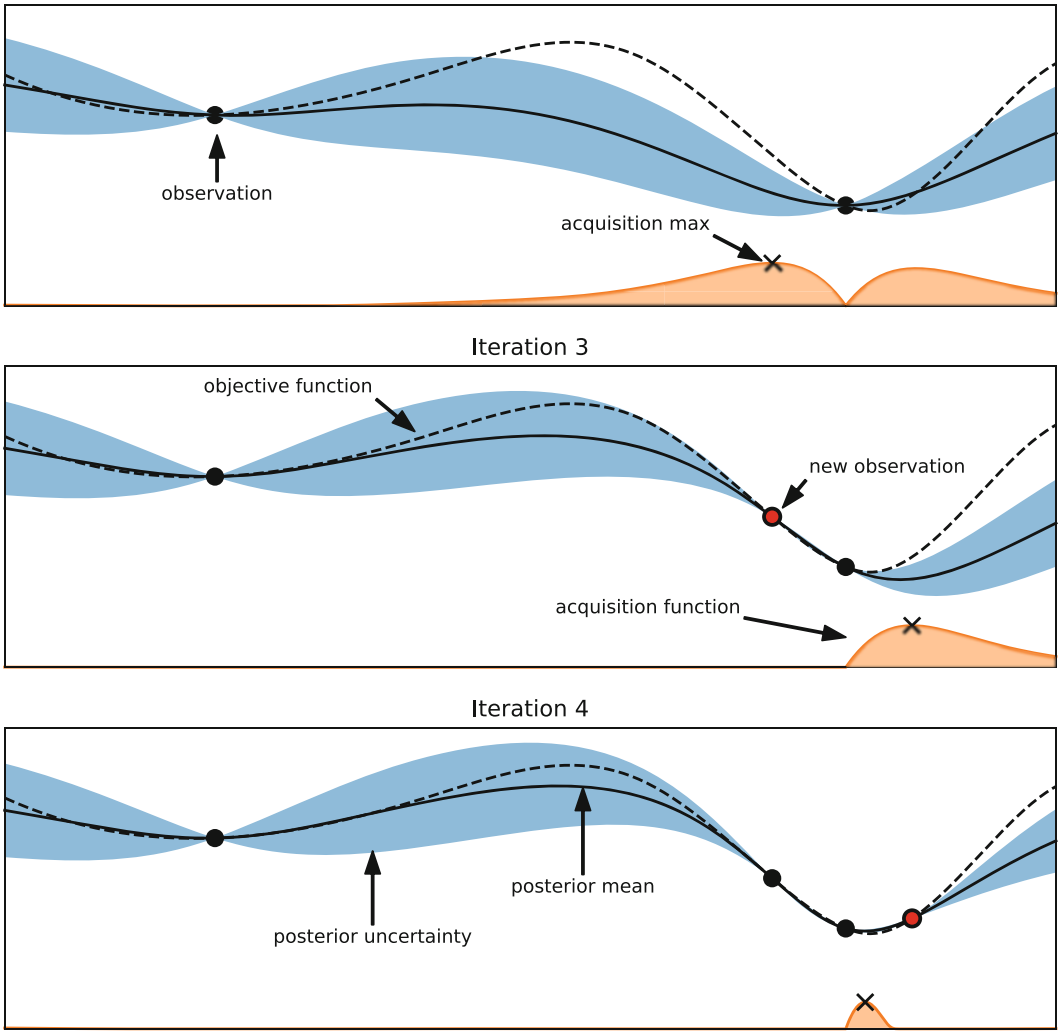

FIGURE 5.2: **Illustration of Bayesian optimization on a 1-d function.** The goal is to minimize the dashed line using a Gaussian process by maximizing the acquisition function represented by the lower orange curve. Predictions are shown as black line, with blue areas representing the uncertainty. In the top panel the acquisition value is low around observations, and the highest acquisition value is at a point where the predicted function value is low and the predictive uncertainty is relatively high. In the middle panel the predicted mean to the right is much lower and the next observation is conducted there. In the bottom panel the next evaluation is done there due to its expected improvement over the best point so far. (Feurer and Hutter, 2019)

Bayesian Optimization Pros & Cons	
Pros	Cons
Data efficient	Poorly parallelizable
Noise robust	Fail with categorical parameters
Acquisition function is relatively simple	Poor scalability to high dimensions

TABLE 5.1

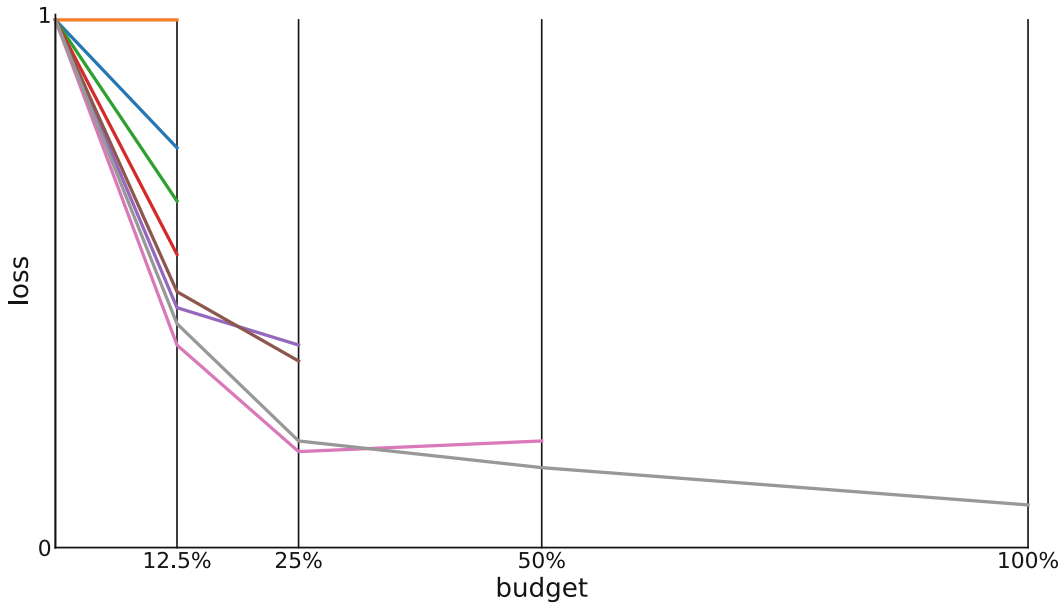

FIGURE 5.3: **Successive Halving example.** The plot represents the successive halving in action. It starts by randomly sampling n configurations. In each iteration it drops the worst half and doubles the *budget* for the remaining trials until it reaches the maximum budget allocated. Each line corresponds to a configuration. (Li et al., 2017).

5.3 Hyperbands

Hyperbands (HB) algorithm (Li et al., 2017) focuses on speeding up random search through adaptive resource allocation. This is a principled early-stoppping method that adaptively allocates a predefined resource, e.g., iterations, data samples or number of features, to randomly sampled configurations. In principle, the objective function $f : \mathcal{H} \rightarrow \mathbb{R}$ is expensive to evaluate but it is possible to define a cheaper approximation $\tilde{f}(x, b)$ parameterized by the *budget* $b \in [b_{min}, b_{max}]$, where $\tilde{f}(x, b_{max}) = f(x)$. In practice, the HB is a bandit strategy that takes advantage of these different budgets b with repeated *successive halving* (SH; Jamieson and Talwalkar, 2015) to identify the best out of n randomly-sampled configurations. SH evaluates these n configurations with a small budget, keeps the best half, according to the preferred metric, and doubles their budget (see Figure 5.3).
The main problems with a simple SH approach are:

- the significant budget required to evaluate a large number of sample. Since the initial budget get agnostically distributed among all the n samples, so that the budget per sample is B/n, by increasing the number of sample one must proportionally increase the overall budget in order to keep the same level of precision in the evaluation.

- the trade-off between the number of initial samples and the duration of the training phase is not known *a-priori*. In practice the algorithm cannot balance between exploration (the number of samples) and precision (the training steps).

HB solves this problems by balancing *aggressive* evaluations with many configurations on the smallest budget, and very conservative runs that are directly evaluated on b_{max}. The algorithm is shown in Algorithm 3.

Hyperbands Optimization Pros & Cons	
Pros	Cons
Highly scalable	Needs a significant total budget
Easily parallelizable	
High performance with few resources	with infinite resources performs as RS

TABLE 5.2

Following Li et al. (2017) we define b_{max} as the maximum amount of resources that could be allocated to a single configuration; η an integer that controls the proportion of configurations discarded in each round of SH. The algorithm is as follows:

Algorithm 3 Hyperband algorithm

Input: budgets b_{min} and b_{max}, η
Initialization: $s_{max} = \lfloor log_\eta \frac{b_max}{b_min} \rfloor$
1: **for** $s \in \{s_{max}, s_{max} - 1, ..., 0\}$ **do**
2: sample $n = \lceil \frac{s_{max}+1}{s+1} \cdot \eta^s \rceil configurations$
3: run SH on the configurations with $\eta^{-s} \cdot b_{max}$ as initial budget
Return: Configuration with the smallest lost seen so far.

In Line 1 it is computed the geometrically spaced budget $b \in [b_{min}, b_{max}]$. The number of configurations sampled in Line 3 is chosen such that every SH run requires the same total budget. SH internally evaluates configurations on a given budget, ranks them by their performance, and continues the top $\eta - 1$ on a budget η times larger. This is repeated until the maximum budget is reached.
We have summarised the pros and cons of the hyperbands algorithm in Table 5.2.

It is clear that HB approach to hyperparameter optimization can be considered orthogonal to the method used in the BO framework, since it tries to speed up configuration evaluation, instead of optimizing configuration selection, by dynamically allocating the resources where are most needed. In virtue of this orthogonality in Falkner, Klein, and Hutter (2018) it has been proposed a combination of these two very promising methods, with a model-based hyperband approach that will be detailed in the next section.

5.4 BOHB

The algorithm designed in Falkner, Klein, and Hutter (2018) is a combination of both Bayesian optimization and hyperbands approach, hence the name Bayesian Optimization HyperBands (BOHB). The aim of this algorithm is to deliver an optimization algorithm which has:

- Efficient use of resources, since the available budget of computational resources/time is usually not much larger than that necessary to fully train the model.
- Strong final performance, even compared with other methods with a larger budget.
- Effective use of parallelization, to fully exploit the current high-performance hardware.

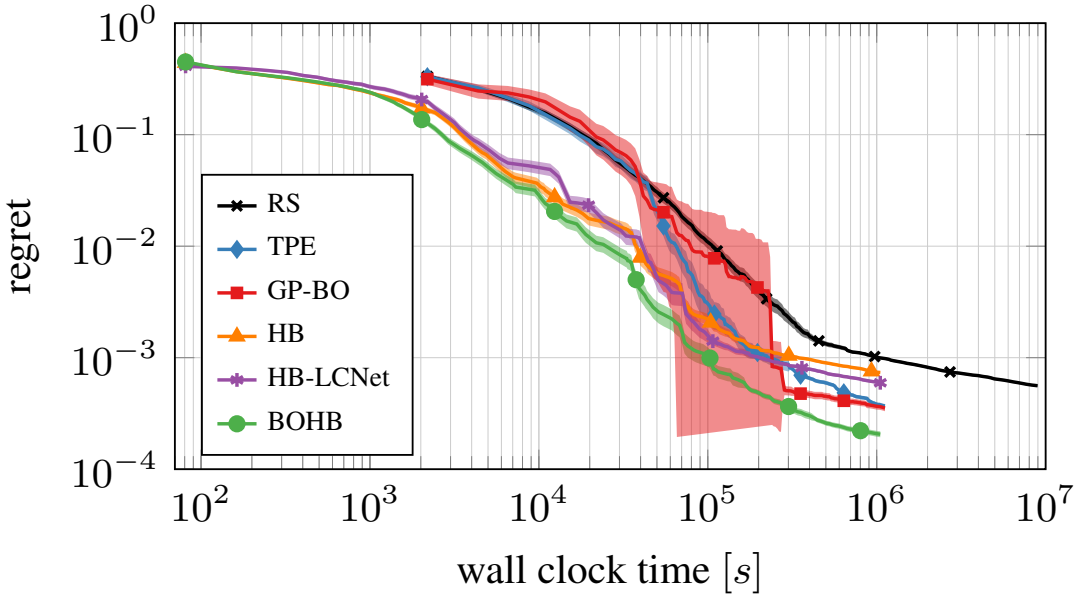

FIGURE 5.4: **HPOs performance comparison.** The plot depicts a comparison of several HPO algorithms (see the inset) used to optimize a simple neural network. On the y-axis it is shown a metric of performance called *immediate regret* (Shah and Ghahramani, 2015) which is defined, at a certain time-step t, as: $r_t = |f(\bar{x}_t) - f(x^*)|$, where x^* is the known optimizer of f and $\bar{x}_t$ is the recommendation of the algorithm after t steps. The BOHB algorithm shows consistently superior performance with respect to other state-of-the-art methods.

- Scalability, in order to consistently achieve the best results on small problems as well as large-scale problems.
- Flexibility in application scenarios. The algorithm must be able to handle every type of hyperparameter and machine learning paradigm (supervised, unsupervised and reinforcement learning).
- Simplicity with respect to gaussian process BO, for example.
- Computational efficiency with respect to BO.

Hyperbands already satisfies most of those requirements, therefore they first focus on improving BO before combining them.
In order to achieve simplicity, flexibility and computationally efficiency the BO is based on the so-called tree parzen estimator (TPE; Bergstra et al., 2011). The TPE is Bayesian optimization method that uses kernel density estimator (KDE) to model the densities

$$l(\mathbf{x}) = p(y < \alpha | x, D) \quad g(\mathbf{x}) = p(y > \alpha | x, D) \tag{5.2}$$

instead of modeling the objective function directly with $p(f|D)$. To choose the new point to evaluate (x_{new}) TPE maximizes the ratio $l(x)/g(x)$ (Falkner, Klein, and Hutter, 2018). This can be shown to be equivalent to the maximization of Equation 5.1 (Bergstra et al., 2011).
Due to the nature of kernel densities estimators TPE can handle complex hyperparameters spaces and the computation scales linearly in the number of evaluated data points.
Now we can again consider HB, that in this algorithm are used to determine how

Hyperparameters search space	
Local	Global
$n_0 \in \{40, ..., 60\}$	$\lambda \in [10^{-4}, 10^{-1}]$
$n_i \in \{6, ..., 20\}$	$B \in \{16, 32, 64, 128, 256\}$
$m \in \{1, ..., 10\}$	$\beta_1 \in [0.5, 1]$
	$\beta_2 \in [0.9, 1]$

TABLE 5.3

many configurations to evaluate and the relative budget. The difference with respect to classical HB optimization is that the random selection is replaced by a BO component to better guide the search. The full algorithm can be formalized as follows:

Algorithm 4 BOHB algorithm

Input: $\eta, D_b, N_s, N_{min}, b_w$
1: **if** $rand() < \eta$ **then return** random configuration
2: $b = \arg\max\{D_b : |D_b| \geq N_{min} + 2$
3: **if** $b = \{\emptyset\}$ **then return** random configuration
4: fit KDEs
5: draw N_s samples
return sample with the highest ratio $l(\mathbf{x})/g(\mathbf{x})$

Where N_{min} is the minimum number of data points required to constrain all the hyperparameters (in this case is set as the number of hyperparameters + 1); D_b the number of observations for a certain budget b; N_s number of sampled points from the KDE; b_w is the *bandwidth* factor, which in practice regulates the degree of *exploration* around promising configurations.
The performance of the BOHB algorithm, compared with other algorithms, is shown in Figure 5.4.

5.5 Optimizing iSRCNN

In this section we will focus on the optimization of two types of hyperparameters, what we have defined as *global* hyperparameters, that is, the parameters that regulate the global learning behavior of the network, and *local* hyperparameters, which dictate the structure of the architecture.
In particular, the global hyperparameters are: the learning rate (λ), the batch size (B) and the betas (β_1, β_2) of the optimizer (see Section 6.2).
While the local hyperparameters are: the number of *outer* and *inner* channels n_o and n_i, respectively, and the number of layers in the non-linear mapping block (see Figure 4.1).

The hyperparameter search space $\mathcal{H}$ is defined in Table 5.3: The decision of this particular subspace of hyperparameters is largely arbitrary and is based partly on heuristics and partly on our experience with this architecture and the dataset. In particular, it has been widely proved that the batch size/learning rate ratio is fundamental in determining the accuracy of any model (He, Liu, and Tao, 2019). The learning rate is in principle unrestricted and could assume any value, but it has been shown that higher learning rates (in the order of 0.1 for deep network) achieve the

Hyperparameter	Importance	Correlation with L1
inner channels	0.619	-0.710
number of maps	0.205	-0.417
outer channels	0.188	-0.305
learning rate	0.720	0.759
batch size	0.167	-0.387
β_1	0.070	-0.200
β_2	0.043	-0.101

TABLE 5.4: **Relative impact of hyperparameters on the loss function.** The relative importance is a measure

best results (Smith, 2018), therefore we are including this values in our search space. On the other hand, the batch size is bounded above by the hardware (namely, the memory) and below by the extreme variance with batches of few examples. Moreover, we see that the best training speed is achieved with a batch size of 64. Since optimizing the training time is an objective of this thesis we decide to focus the search in the batch size space around this value (see Table 5.3).
This, in turns, also bounds the learning rate due to their ratio being such an important part of the learning process. Moreover, the batch size can only assume integer values, therefore for simplicity we choose to restrict the search to the powers of two.

The default values of β_1 and β_2 in the literature are 0.9 and 0.999 respectively, with a maximum possible value of 1. Therefore, we set up the search space for those two parameters to include a larger portion of the $[0, 1]$ interval.

Regarding global hyperparameters, all the three search spaces are chosen to be centered approximately around the values of the original FSRCNN, i.e. $m = 4$, $n_0 = 56$, $n_i = 12$.

For the optimization algorithm we use the BOHB described in the previous Section, with an hyper-band reduction factor of 2, a maximum budget of 300 epochs, and 128 samples both for the global and the local hyperparameters search.

Regarding the local optimization, we find that an increase in the inner channels number is the most impactful factor for reducing the training loss function, with a strong anti-correlation of -0.710. The relative importance of this hyperparameter can be immediately seen in the bottom panel of Figure 5.4 (red paths).
The other two hyperparameters show a weak anti-correlation with the loss function, even though they contribute to the overall improvement of the accuracy.
Note that the relative importance and the correlation are calculated on the runs that successfully reached the epochs budget (300 epochs). Since the time of training is strongly correlated with the training loss we have to remove its contribution to obtain a fair comparison. We have also discarded the runs the showed evident overfitting, in order to not reward hyperparameters that allowed this type of unwanted behavior.

Similarly, among the global hyperparameters only one showed a strong correlation with the loss function: the learning rate.
It seems that selecting a lower learning rate favors a more stable training and a fastest convergence toward a good optimum. Conversely, the other global hyperparameters seem to be relatively less important.

Summing up our optimization procedure we find that the best combination of hyperparameters is the following:

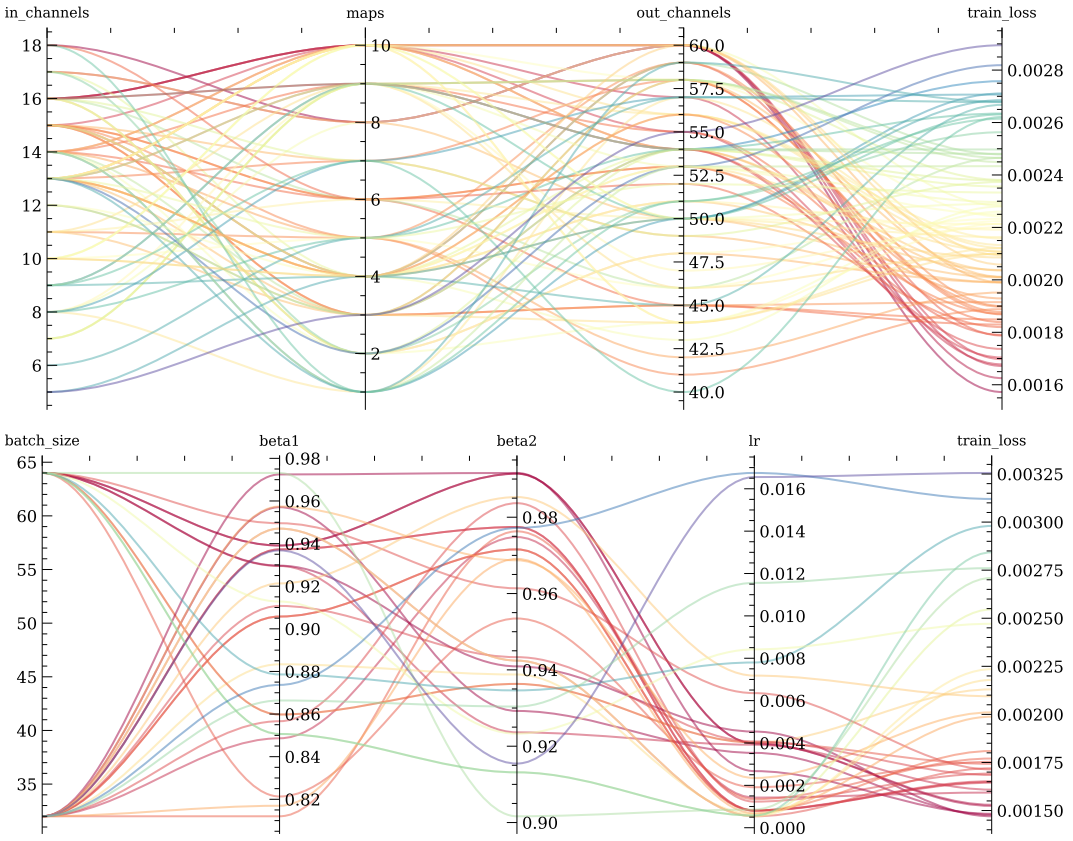

FIGURE 5.5: **Local and global hyperparameters optimization.** The plot shows the parallel coordinates for the two groups of hyperparameters. In the top panel are shown the local hyperparameters, inner channels, number of maps, outer channels and finally the training loss. Analogously, the bottom plot shows the global parameters, with the columns representing the batch size, the β_1 and β_2, the learning rate and the training loss. We can readily see from the plot how some hyperparameters consistently lead to lower training losses (red paths).

Best hyperparameters	
Local	Global
$n_0 = 60$	$\lambda = 0.0035$
$n_i = 16$	$B = 32$
$m = 8$	$\beta_1 = 0.957$
	$\beta_2 = 0.929$

We use these combination of hyperparameters to train our final version of the iSR-CNN (see next Section).

Chapter 6

Training

The training phase is the process through which the machine learning model *learns* how to map the input space X in the output space Y. In a supervised framework (see Section 1.1) we define the training dataset a subset of the whole data composed by couples (x_i, y_i) with $x \in X$ and $y \in Y$. The former is the input, the latter is the label. Each one of these examples represents a sampling of the objective function that we want to approximate $Y = f(X)$.
The input data is fed to the network which elaborates the input while passing through all the layers. The output is then compared to the label, and the difference measured with a metric. Then, as we saw in Section 1.3, the network uses this information to update its parameters, starting from the last layer and going backwards, to optimize the metric.
The so called *feed-forward* and the back-propagation are the two complementary phases that allow the convergence of the network towards the optimum of the metric.

For the super-resolution task, both X and Y are sets of images, with y_i being the HR images and the x_i the LR ones. The function that we want to approximate is the map between the LR and the HR space of the same image. The metric that we use is the $\mathcal{L}_1$ loss function:

$$\mathcal{L}_1 = \frac{1}{WH} \sum_{i}^{W} \sum_{j}^{H} |z_{ij} - y_{ij}| \tag{6.1}$$

Where W and H are the image dimensions; z is the output of the network given an input x. This loss function is a simplified version of the $\mathcal{L}_2$ that we have already seen in Section 1.2, which is basically the $W \times H$-dimensional Manhattan distance.
This is somewhat unusual in the SR field, where the preferred loss function is usually considered the $\mathcal{L}_2$ (e.g. Dong et al., 2016; Dong, Loy, and Tang, 2016). Nevertheless, we find that $\mathcal{L}_1$ leads to much more accurate images and since, as far as we know, there are no systematic studies in the literature that prove the superiority of one over the other, we choose to stick with this loss function.

In practice, the input data is fed to the network in chunks of many data points, also called *batches*. This method has several advantages:

- Since it is very common to train a network on a GPU this allows for an extreme parallelization, with a training time that is constant up to memory saturation, no matter how large the batch size is[1].
- Prevents overfitting (see Section 1.3).

[1] this is true in principle, but in practice for a very large batch size other factors outside of the GPU can contribute to significantly slowing down training time

- Greatly stabilizes the training, since it reduces the variance of the loss function proportionally to the batch size. This in turns allows for a better convergence.

Another fundamental aspect to consider in order to avoid overfitting is the variability of the training set. In general, overfitting is a relative measure that compares the performance of the network on the training set compared to the performance on data that the network has never *seen*. Therefore, is fundamental that the examples in the training dataset are as representative as possible of the entire population.
This is a somewhat *automatic* overfitting, but it is also possible for a human to overfit especially when trying to improve an algorithm on the same data.
The way to avoid both risks is to randomize the choice of the dataset to each new training attempt, thus we prevent us from overfitting and, given a large enough number of examples, we feed the network with representative data.

In order to test the degree of overfitting it is common to calculate a parallel loss for a subset of the dataset that is not used by the network for the training process. This subset is often called *validation set* and the corresponding metric is the *validation loss*.
If we notice that the training loss is way lower than the validation loss this is a clear sign of overfitting.

6.1 Dataset pre-processing

As already detailed in Chapter 3 our dataset is composed by roughly one million of simulated galaxies, each with 4-channels and two resolutions: a LR image 20×20, the input, and a HR image 80×80, the label. The training subset is composed by 8192 LR-HR couples (see Section 6.4 for the details on this number), the validation subset is composed by 4096 couples. The subsets, along with the batch size, are always a power of 2, this allows to always have an integer batch number for every training iteration.

Dataset normalization is a fundamental requirement in many areas of statistics, this is even more accentuated in machine learning where this technique has proved to be very impactful on the overall performance (Singh and Singh, 2020). In particular, it is necessary to avoid two common pitfalls of machine learning back-propagation: the vanishing gradient and the exploding gradient, which, among other reasons, can be caused by too low or too high levels of neurons activation. In other words, input values that are too close to zero or very large can strongly destabilize the convergence of the network.
Therefore, it is common to normalize the dataset between 0 and 1. In our case we used a simple minmax normalization:

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)} \tag{6.2}$$

In this way, we preserve the pixels distributions, but rescaled to the interval $[0, 1]$.
However, real application scenario, for the specific design of the problem we are trying to solve, the target high-resolution images that the network tries to reconstruct are not known. This implies that we cannot normalize for the high-resolution images in the pre-processing phase, since in the test phase the images are going to be only low-resolution ones. However, HR images show an higher range of values, since the LR counterparts, in a sense, average the value of more pixels in a single one. Therefore, as a coarse workaround, we consider the maximum and the minimum

pixel values of the entirety of the dataset, for each map, and we normalize the images accordingly.

Another needed pre-processing step is to divide each 4-channels galaxy into four different concatenated images. This step is not theoretically justified but it proved to greatly stabilize the training procedure and leads to overall better results.
However, this phenomenon can be explained intuitively by assuming two different galaxies

6.2 Optimizer and scheduler

As we have already seen (Section 1), stochastic gradient-based optimization is of core importance for optimizing a machine learning algorithm. In particular, SGD proved itself as an efficient and effective optimization method that was central in the recent deep learning success.
However, for noisy objective functions, which depend on many parameters, efficient stochastic optimization techniques are required. Particularly, to robustly solve the efficient optimization of stochastic objectives with high-dimensional parameters spaces. In these cases, higher-order optimization methods are ill-suited, and first-order methods are the only choice.

Let $f(\theta)$ be a noisy objective function (a loss function in our case): a stochastic scalar function that is differentiable with respect to the parameters θ. The stochasticity of our loss function is mainly due to the random subsampling of our data (batches). The objective is to minimize the expected value of this function.
Following the original paper, we define α as the gradient step size; $\beta_1, \beta_2 \in [0, 1)$ the exponential decay rates for the moment estimates; θ_0 the initial parameter vector; m and v the biased first and second moment of the gradient; ϵ the coefficient to improve numerical stability. The algorithm is formally summarized in Algorithm 5[2]. Note that all the operations on vectors are element-wise.

Algorithm 5 Adam algorithm

Require: $\beta_1, \beta_2 \in [0, 1)$
Require: θ_0
$m_0 \leftarrow 0$ Initialize 1^{st} moment vector
$v_0 \leftarrow 0$ Initialize 2^{nd} moment vector
$t \leftarrow 0$ Initialize time-step
while θ_t not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ Get gradients w.r.t. f at time-step t
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ Update biased first moment estimate
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ Update biased second raw moment estimate
    $\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ Compute bias-corrected first moment estimate
    $\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ Compute bias-corrected second raw moment estimate
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$ update parameters
return θ_t Resulting parameters

[2]The actual algorithm implementation is slightly different, due to efficiency reasons. In particular, the last three lines in the while loop can be written as $\alpha_t = \alpha \cdot \sqrt{1 - \beta_2^t} / (1 - \beta_1^t)$ and $\theta_t \leftarrow \theta_{t-1} - \alpha_t \cdot m_t / (\sqrt{v_t} + \epsilon)$.

An important property of Adam's update rule is its careful choice of stepsizes. Assuming $\epsilon = 0$, the effective step taken in parameter space at time-step t is:

$$\Delta_t = \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t}} \tag{6.3}$$

The effective step-size has two upper bounds:

$$\begin{cases} |\Delta_t| \leq \alpha \cdot \frac{(1-\beta_1)}{\sqrt{1-\beta_2}} & (1-\beta_1) > \sqrt{1-\beta_2} \\ |\Delta_t| \leq \alpha & \text{otherwise} \end{cases} \tag{6.4}$$

The first case only happens in the most severe case of sparsity: when a gradient has been zero at all time-steps except at the current time-step. For less sparse cases, the effective step-size will be smaller.
Since α sets (an upper bound of) the magnitude of steps in parameter space, we can often deduce the right order of magnitude of α such that optima can be reached from θ_0 within some number of iterations.

The local convergence of the algorithm has been proved in the case of a non-convex objective function that is at least twice continuously differentiable (Bock and Weiß, 2019).

Adam shows superior performance in many machine learning scenarios, in particular it has been proven to be very effective in conjunction with convolutional neural networks (Kingma and Ba, 2015), where Adam clearly outpaces other state-of-the-art optimizers. It combines a rapid progress in lowering the loss function in the initial stage of the training with a faster final convergence in the long run.
More interestingly, it has been found that in CNNs the second moment estimate (linked to the β_2 parameter) is a poor approximation to the geometry of the loss function. On the contrary, reducing the first batch variance, i.e. the variability in the input images within the same batch, through the first moment is more important and contributes to the observed speed-up (Kingma and Ba, 2015).

Alongside with the built-in adaptiveness of the Adam algorithm we empirically find that a dynamical scheduler for the learning rate is very beneficial for the overall convergence of the algorithm. The most effective technique is the reduction of the learning rate by a certain factor f when the loss function reaches a plateau at least N epochs long, also called *patience* (see (Chee and Toulis, 2018) for an theoretical description of this method).
In particular, a reduction factor of 2 is enough to have a visible effect on the learning rate but not too high to hinder the convergence. The patience factor is set to ten epochs instead.

6.3 Weight initialization

ReLU/PReLU-based networks are easier to train compared with traditional sigmoid-like activation networks. However, a bad initialization of the parameters of the network can still hamper the learning of a highly non-linear system. Usually, CNNs used to be initialized by random weights drawn from Gaussian distributions (e.g. Krizhevsky, Sutskever, and Hinton, 2012). With fixed standard deviations (e.g., 0.01 in Krizhevsky, Sutskever, and Hinton, 2012), very deep models have difficulties to converge, as also reported in Simonyan and Zisserman (2015). To address this issue, in Simonyan and Zisserman (2015) it is suggested that a pre-training phase on a

shallower model can help to initialize the weights of the target model. This strategy, however, requires more training time, and may also lead to a poorer local optimum. Glorot, Bordes, and Bengio (2010) proposed to adopt a properly scaled uniform distribution for initialization. This is commonly called *Xavier* initialization, in which layers' biases are initialized to zero and the weights at each layer are initialized as:

$$(\mathbf{w}_i)_j \sim U\left[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}\right] \tag{6.5}$$

where $(\mathbf{w}_i)_j$ is the $j-th$ parameter of the $i-th$ layer, U is the uniform distribution and n is the size of the weight vector of the previous layer $\mathbf{w}_{i-1}$.
This method is based on the assumption that the activations are linear. This assumption is not valid for ReLU and PReLU.
He et al. (2015) proposed a new initialization method to solve the non-linear problem, called Kaiming initialization. They found that a proper initialization method should avoid reducing or magnifying the magnitudes of input signals exponentially. The condition to prevent this exponential runaway is to set:

$$\mathrm{Var}(\mathbf{w}_i) = 2/n_i \tag{6.6}$$

Where $n_i = c_i k_i^2$, k_i and c_i are the kernel size and the number of channels in the $i-th$ layer, respectively. In practice, the initialization scheme implies that the parameter are drawn from a normal distribution:

$$\mathbf{w}_i \sim N\left(0, \frac{2}{n_i}\right) \tag{6.7}$$

where N is the normal distribution with mean 0 and standard deviation $\sqrt{2/n_i}$.
In the Kaiming initialization the biases are set to zero.

6.4 The impact of training size

In this section we explore the impact of the training dataset dimension on the performance of the algorithm. It is widely known that a large and various dataset can hugely improve the performance of any deep learning algorithm (e.g. Mukherjee et al., 2003; Tsangaratos and Ilia, 2016; Barbedo, 2018). This is especially true for deeper architecture with many millions parameters. Our starting dataset of mock galaxies, with its 1.5 millions samples, can be certainly classified as *large*. Moreover, since each galaxy depends on 18 parameters and that each parameter is very impactful on the final output, the dataset is also very varied and representative.
However, our model, iSRCNN, sits in between the shallow convolutional networks and the heaviest models that are the state-of-the-art at the moment (see Figure 2.9). Therefore, we are in the position to investigate what is the impact of training size on the performance, since we can potentially feed the algorithm with a number of examples that far exceed the number of parameters in the network ($\sim$ 120 thousands). For this test we consider different training sizes ranging from 512 up to 16384, in increments of power of two.
Our test metric is the validation loss, since overfitting on the training data is expected, especially for smaller training sizes. In particular, we use a common validation dataset size of 1024 examples.
The number of training epochs, i.e. the number of times the network *see* all the examples once, is set such that the number of epochs times the training size is a constant.

FIGURE 6.1: **Training size impact on performance.** The plot shows the validation loss of six runs of training that use six different training sizes. The number of epochs is set so that the number of epochs times the training sizes are constant (see text for further details). A lower validation loss indicates better performance.

The philosophy behind this choice is to have a standardized test, in which the absolute number of examples fed into the network is equal throughout each run, thus not penalizing too much the run with a smaller training size.
Finally, we use a common batch size equal to 32 and a learning rate of 0.001.

The validation losses of our runs are plotted in Figure 6.1 and the results are summarized in the following table:

number of epochs	training size	validation loss
4096	512	0.0084
2048	1024	0.0084
1024	2048	0.0065
512	4096	0.0040
256	8192	0.0033
128	16384	0.0045

From Figure 6.1 is clearly visible that smaller batch sizes are suboptimal and lead to low performance. On the other hand, increasing the training size above $\sim 16k$ starts to lower the performance as well. The best performance seems to be reached for a training size around 8192.
Interestingly, the performance starts to degrade when the number of examples ($66536 = 16384 \times 4$ maps for each galaxy) is comparable to the number of parameters of the network ($\sim 122k$). This seems to suggest a sort of *memory* mechanism of the network that, nevertheless, does not preclude the generalization over unseen examples.

6.5 Outliers impact

As already mentioned in Section 3.4, the dataset presents some problematic images. In particular, some of the low resolution maps show some pixel outliers.
This phenomenon is due to numerical instabilities, especially near the centre of the

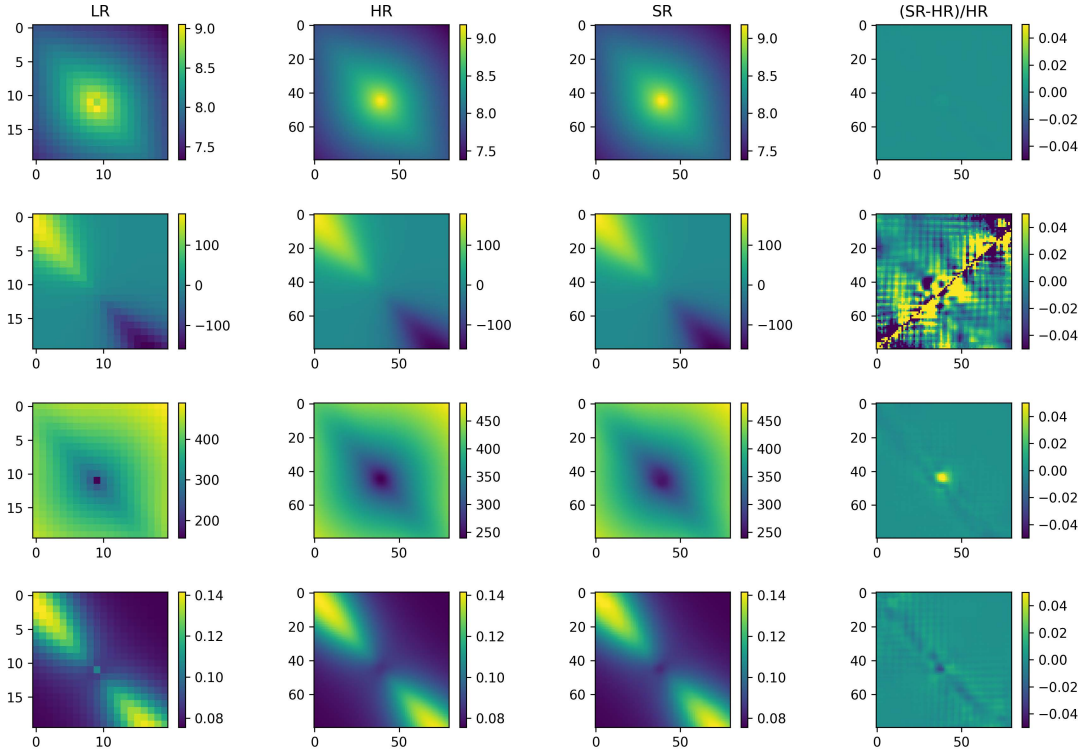

FIGURE 6.2: **Example of super-resolution on a galaxy simulation.** The plot shows an example of super-resolution resulted from the application of our network iSRCNN. The first 3 columns in the image represent the LR, HR and SR images, respectively, while in the last column it is calculated the residual between the SR and the HR images, on a scale of (-5%, +5%) difference, to showcase the accurate reconstruction. The 4 rows represent the surface brightness, the line-of-sight velocity and velocity dispersion, and the mass-to-light ratio, respectively. We chose this example to show the impact of an outlier pixel (see text) clearly visible in the mass-to-light ratio LR image. Since this artifact is due to the resolution alone, it is not present in the HR image. However, iSRCNN successfully correctly recognizes it as an artifact and does not reproduce it in the super-resolution image, proving the robustness of our model to outliers.
The relatively high residuals in the line-of-sight velocity reconstruction are due to a division close to zero, where numerical instabilities can cause greater variance. This is not, however, a symptom of a bad reconstruction.

galaxy, where the density profile can sometime diverge. The workaround adopted in the construction of the dataset is to compute the properties of the central pixel as an average over the entire pixel area as opposed to all the other positions where the observables are computed point-wise.
Nevertheless, we prove that those outliers don't particularly impact the performance of the network and, except for some noticeable cases, the outlier pixels are completely ignored by the algorithm.
This is probably due to the fact that, first of all, the statistical significance of the outliers is really negligible, and second, due to the relative scarcity of images that show this artifact.

An example of such a case is shown in Figure 6.2. Here we can clearly see in the mass-to-light ratio LR map an outlier pixel right in the centre of the galaxy. This is a spurious feature and this artifacts is only due to the low resolution grid, in fact it is not present in the HR image. Our network successfully recognises this pixel to be an outlier and ignores its contribution, returning an highly accurate high-resolution map, as one can tell from the residual map in Figure 6.2.

Chapter 7

Results and conclusions

Leveraging the recent advancements in the deep learning field we have developed a new architecture capable of up-scale low resolution images by a factor of 4 while reconstructing even the fine details of the original high-resolution image. This opens up interesting applications in the context of physical simulations, which are often time very time demanding and computationally expensive. To showcase the potential of this new approach we used an existing fitting procedure to estimate galaxies physical parameters starting from photometric and kinematics data. This pipeline iterates over a large 18-dimensional parameter space to produce mock galaxies that are progressively closer to the real one. The production of an high-resolution mock galaxy at each iteration takes only few milliseconds on a GPU, but given the huge amount of parameters to explore the pipeline needs to iterate over tens of millions of mock galaxies before obtaining a satisfactorily low uncertainty on the estimated parameters. This usually requires few hours on a GPU or few days on a CPU.
Our approach allows us to create mock galaxies at low resolution, which are much more cheap to produce (in particular, 16 times less, since the up-scaling factor of our architecture is 4 for each dimension). Therefore, the developed deep learning model needs to have a very fast inference time (in the order of a millisecond) along with a good accuracy in reconstruction the simulations.
We explored several different state-of-the-art models, but we found that the most recent architecture are over-parameterized and not suitable for very fast inference. We instead focused on improving a smaller network called FSRCNN (Dong, Loy, and Tang, 2016) with the most recent techniques in deep learning and the resulting network, iSRCNN, is more complex than its predecessor but still be able to super-resolve a low-resolution galaxy map in 2.5 milliseconds, both on CPU and GPU cores.
This would allow the galaxy fitting procedure to run on CPUs instead of GPUs, which are far more cheap and available than the latter. Therefore, using our approach the parameters estimation can be run on relatively accessible CPU clusters containing hundreds of cores, thus reducing the time required to few minutes.
In Figure 7.1, we have compared the time it takes for a single iteration of the algorithm presented in Rigamonti et al. (2022) when trying to fit a single galaxy with a full resolution galaxy simulation (BE in the Figure) with the shortcut that our network provides in up-scaling a cheaper low-resolution simulation to a full resolution one. In particular, we use our fully optimized iSRCNN and the faster but less accurate version that we described in Section 4.1, in Figure 7.1 BE+iSRCNN and BE+iSRCNN(fastest), respectively.

Regarding the accuracy of reconstruction, is difficult to make a direct comparison with the literature, since the attempts of leveraging machine learning in the field of physical simulations is a relatively new approach and common metric are still

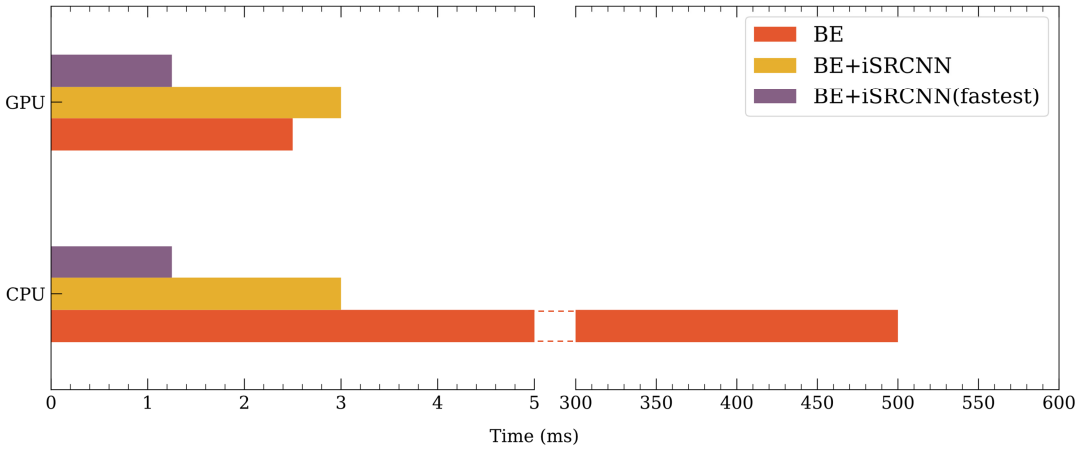

FIGURE 7.1: **Inference time comparison of a single parameters estimation with and without our network.** We compare the inference time of a single iteration of the parameters estimation presented in Rigamonti et al. (2022), on two different hardwares (CPU and GPU). In red, we indicate the full Bayesian estimate (BE) on the high-resolution simulations. In yellow, where our network up-scales a low-resolution simulation and the BE is

lacking. However, we can make use of the parameters estimation procedure to directly evaluate what are the uncertainties of the parameters inferred starting from a super-resolved simulation with respect to a known mock galaxy. This would give us the worst case scenario, since the mock galaxy will be fitted with infinite precision, since it is a synthetic simulation produced by the procedure itself.
Although we have some very encouraging results, in order to have a statistically significant proof of the systematic accuracy of our method we need tens of complete fitting on real galaxies, which are still ongoing.

In this thesis we have proved the accuracy and the efficiency of deep learning model when applied to physical simulations. Since our networks doesn't have any prior knowledge on the problem it tried to solve, we argue that this model can tackle any similar challenge involving the reconstruction of a 2D simulation with up-scaling factors up to x4.

Our aim was to deliver a machine learning framework tested on a real use-case that can be useful to anyone needs to perform demanding physical simulations but does not have the computational resources, since our network can be fully trained within few hours on a new dataset and can be deployed on every personal computer. We proved that this is indeed possible if one can trade some accuracy for a dramatic reduction in computation time.

Some possible improvements may be to try an full architecture search, using as a metric not only the reconstruction accuracy, but also the inference time, in order to find the most efficient algorithm, given a certain time requirement.
Another possible future step will be to try to completely avoid the low-resolution step and deliver a full resolution simulation from the input parameters using an end-to-end deep learning algorithm. This is, in a sense, an inverse classification problem, which has been successfully solved in recent years, and we think that there are excellent conditions for the success of these types of algorithms.

Acknowledgements

The implementation, the trainings and the tests were performed using Python 3 and the machine learning library PyTorch.
We used the Ray library to perform BOHB hyperparameter optimization.
Part of the training and the testing were performed on Marconi100 (M100) Cineca HPC.

Bibliography

Anderson, Edgar (1936). "The Species Problem in Iris". In: *Annals of the Missouri Botanical Garden* 23.3, pp. 457–509. ISSN: 00266493. URL: http://www.jstor.org/stable/2394164.

Barbedo, Jayme Garcia Arnal (2018). "Impact of dataset size and variety on the effectiveness of deep learning and transfer learning for plant disease classification". In: *Computers and electronics in agriculture* 153, pp. 46–53.

Bergstra, James et al. (2011). "Algorithms for Hyper-Parameter Optimization". In: *Proceedings of the 24th International Conference on Neural Information Processing Systems*. NIPS'11. Granada, Spain: Curran Associates Inc., 2546–2554. ISBN: 9781618395993.

Bevilacqua, Marco et al. (Sept. 2012). "Low-Complexity Single Image Super-Resolution Based on Nonnegative Neighbor Embedding". In: DOI: 10.5244/C.26.135.

Bishop, Christopher (2006). *Pattern Recognition and Machine Learning*. Springer. URL: https://www.microsoft.com/en-us/research/publication/pattern-recognition-machine-learning/.

Bock, Sebastian and Martin Weiß (2019). "A Proof of Local Convergence for the Adam Optimizer". In: *2019 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8. DOI: 10.1109/IJCNN.2019.8852239.

Bode, Mathis et al. (2021). "Using physics-informed enhanced super-resolution generative adversarial networks for subfilter modeling in turbulent reactive flows". In: *Proceedings of the Combustion Institute* 38.2, pp. 2617–2625. ISSN: 1540-7489. DOI: https://doi.org/10.1016/j.proci.2020.06.022. URL: https://www.sciencedirect.com/science/article/pii/S1540748920300481.

Bundy, Kevin et al. (Jan. 2015). "Overview of the SDSS-IV MaNGA Survey: Mapping nearby Galaxies at Apache Point Observatory". In: *ApJ* 798.1, 7, p. 7. DOI: 10.1088/0004-637X/798/1/7. arXiv: 1412.1482 [astro-ph.GA].

Chee, Jerry and Panos Toulis (2018). "Convergence diagnostics for stochastic gradient descent with constant learning rate". In: *International Conference on Artificial Intelligence and Statistics*. PMLR, pp. 1476–1485.

Chen, Peng-Wei, Jung-Ying Wang, and Hahn-Ming Lee (2004). "Model selection of SVMs using GA approach". In: *2004 IEEE International Joint Conference on Neural Networks (IEEE Cat. No.04CH37541)*. Vol. 3, 2035–2040 vol.3. DOI: 10.1109/IJCNN.2004.1380929.

Cho, Kyunghyun et al. (June 2014). "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation". In: DOI: 10.3115/v1/D14-1179.

Claesen, Marc et al. (Dec. 2014). "Easy Hyperparameter Search Using Optunity". In.

Dahl, George E., Tara N. Sainath, and Geoffrey E. Hinton (2013). "Improving deep neural networks for LVCSR using rectified linear units and dropout". In: *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 8609–8613. DOI: 10.1109/ICASSP.2013.6639346.

de Vaucouleurs, Gerard (Jan. 1948). "Recherches sur les Nebuleuses Extragalactiques". In: *Annales d'Astrophysique* 11, p. 247.

Dehnen, W. (Nov. 1993). "A Family of Potential-Density Pairs for Spherical Galaxies and Bulges". In: *MNRAS* 265, p. 250. DOI: 10.1093/mnras/265.1.250.

Dong, Chao, Chen Change Loy, and Xiaoou Tang (2016). "Accelerating the Super-Resolution Convolutional Neural Network". In: *Computer Vision – ECCV 2016*. Ed. by Bastian Leibe et al. Cham: Springer International Publishing, pp. 391–407. ISBN: 978-3-319-46475-6.

Dong, Chao et al. (2016). "Image Super-Resolution Using Deep Convolutional Networks". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 38.2, pp. 295–307. DOI: 10.1109/TPAMI.2015.2439281.

Drory, N. et al. (Feb. 2015). "The MaNGA Integral Field Unit Fiber Feed System for the Sloan 2.5 m Telescope". In: *ApJ* 149.2, 77, p. 77. DOI: 10.1088/0004-6256/149/2/77. arXiv: 1412.1535 [astro-ph.IM].

Drozdzal, Michal et al. (2016). "The importance of skip connections in biomedical image segmentation". In: *Deep learning and data labeling for medical applications*. Springer, pp. 179–187.

Dumoulin, Vincent and Francesco Visin (2016). "A guide to convolution arithmetic for deep learning". In: *ArXiv e-prints*. eprint: 1603.07285.

Elsken, Thomas, Jan Hendrik Metzen, and Frank Hutter (2019). "Neural architecture search: A survey". In: *The Journal of Machine Learning Research* 20.1, pp. 1997–2017.

Falkner, Stefan, Aaron Klein, and Frank Hutter (2018). *BOHB: Robust and Efficient Hyperparameter Optimization at Scale*. DOI: 10.48550/ARXIV.1807.01774. URL: https://arxiv.org/abs/1807.01774.

Feurer, Matthias and Frank Hutter (2019). "Hyperparameter optimization". In: *Automated machine learning*. Springer, Cham, pp. 3–33.

Fisher, R. A. (1936). "THE USE OF MULTIPLE MEASUREMENTS IN TAXONOMIC PROBLEMS". In: *Annals of Eugenics* 7.2, pp. 179–188. DOI: https://doi.org/10.1111/j.1469-1809.1936.tb02137.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1469-1809.1936.tb02137.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1469-1809.1936.tb02137.x.

Freeman, K. C. (June 1970). "On the Disks of Spiral and S0 Galaxies". In: *ApJ* 160, p. 811. DOI: 10.1086/150474.

Gal, Yarin and Zoubin Ghahramani (2016). "A theoretically grounded application of dropout in recurrent neural networks". In: *Advances in neural information processing systems* 29.

Glorot, Xavier, Antoine Bordes, and Y. Bengio (Jan. 2010). "Deep Sparse Rectifier Neural Networks". In: vol. 15.

Goodfellow, Ian J. et al. (2014). "Generative Adversarial Nets". In: *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*. NIPS'14. Montreal, Canada: MIT Press, 2672–2680.

Haskell, B. Curry (1944). In: *Quart. Appl. Math.* 2, pp. 258–261. DOI: https://doi.org/10.1090/qam/10667.

He, Fengxiang, Tongliang Liu, and Dacheng Tao (2019). "Control batch size and learning rate to generalize well: Theoretical and empirical evidence". In: *Advances in Neural Information Processing Systems* 32.

He, Kaiming et al. (2015). "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification". In: *2015 IEEE International Conference on Computer Vision (ICCV)*, pp. 1026–1034. DOI: 10.1109/ICCV.2015.123.

— (2016). "Deep Residual Learning for Image Recognition". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778. DOI: 10.1109/CVPR.2016.90.

Hinton, Geoffrey E. (2012). "A Practical Guide to Training Restricted Boltzmann Machines". In: *Neural Networks: Tricks of the Trade: Second Edition*. Ed. by Grégoire Montavon, Geneviève B. Orr, and Klaus-Robert Müller. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 599–619. ISBN: 978-3-642-35289-8. DOI: 10.1007/978-3-642-35289-8_32. URL: https://doi.org/10.1007/978-3-642-35289-8_32.

Hornik, Kurt, Maxwell Stinchcombe, and Halbert White (1989). "Multilayer feedforward networks are universal approximators". In: *Neural Networks* 2.5, pp. 359–366. ISSN: 0893-6080. DOI: https://doi.org/10.1016/0893-6080(89)90020-8. URL: https://www.sciencedirect.com/science/article/pii/0893608089900208.

Hsu, C., C. Chang, and Chih-Jen Lin (Jan. 2003). "A Practical Guide to Support Vector Classication". In: *Bioinformatics* 1.

Ioffe, Sergey and Christian Szegedy (2015). "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*. ICML'15. Lille, France: JMLR.org, 448–456.

Isaac, Jithin Saji and Ramesh Kulkarni (2015). "Super resolution techniques for medical image processing". In: *2015 International Conference on Technologies for Sustainable Development (ICTSD)*, pp. 1–6. DOI: 10.1109/ICTSD.2015.7095900.

Jamieson, Kevin and Ameet Talwalkar (Feb. 2015). "Non-stochastic Best Arm Identification and Hyperparameter Optimization". In.

Jolicoeur-Martineau, Alexia (2019). "The relativistic discriminator: a key element missing from standard GAN". In: *ArXiv* abs/1807.00734.

Kidger, Patrick and Terry Lyons (2020). *Universal Approximation with Deep Narrow Networks*. URL: https://openreview.net/forum?id=B1xGGTEtDH.

Kiefer, J. and J. Wolfowitz (1952). "Stochastic Estimation of the Maximum of a Regression Function". In: *The Annals of Mathematical Statistics* 23.3, pp. 462–466. ISSN: 00034851. URL: http://www.jstor.org/stable/2236690.

Kingma, Diederik P. and Jimmy Ba (2015). "Adam: A Method for Stochastic Optimization". In: *CoRR* abs/1412.6980.

Kormendy, John and Robert C. Kennicutt (2004). "Secular Evolution and the Formation of Pseudobulges in Disk Galaxies". In: *Annual Review of Astronomy and Astrophysics* 42.1, pp. 603–683. DOI: 10.1146/annurev.astro.42.053102.134024. URL: https://doi.org/10.1146/annurev.astro.42.053102.134024.

Kouame, D. and M. Ploquin (2009). "Super-resolution in medical imaging : An illustrative approach through ultrasound". In: *2009 IEEE International Symposium on Biomedical Imaging: From Nano to Macro*, pp. 249–252. DOI: 10.1109/ISBI.2009.5193030.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton (2012). "ImageNet classification with deep convolutional neural networks". In: *Communications of the ACM* 60, pp. 84 –90.

LeCun, Yann et al. (1989). "Handwritten Digit Recognition with a Back-Propagation Network". In: *NIPS*.

Ledig, Christian et al. (2017). "Photo-realistic single image super-resolution using a generative adversarial network". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4681–4690.

Li, Lisha et al. (2017). "Hyperband: Bandit-Based Configuration Evaluation for Hyperparameter Optimization". In: *ICLR*.

Lim, Bee et al. (2017). *Enhanced Deep Residual Networks for Single Image Super-Resolution*. DOI: 10.48550/ARXIV.1707.02921. URL: https://arxiv.org/abs/1707.02921.

Loussaief, Sehla and Afef Abdelkrim (2018). "Convolutional Neural Network Hyper-Parameters Optimization based on Genetic Algorithms". In: *International Journal of Advanced Computer Science and Applications* 9.10. DOI: 10.14569/IJACSA.2018.091031. URL: http://dx.doi.org/10.14569/IJACSA.2018.091031.

Lu, Zhisheng et al. (2021). "Efficient transformer for single image super-resolution". In: *arXiv preprint arXiv:2108.11084*.

Malczewski, K. and R. Stasiński (2009). "Super Resolution for Multimedia, Image, and Video Processing Applications". In: *Recent Advances in Multimedia Signal Processing and Communications*. Ed. by Mislav Grgic, Kresimir Delac, and Mohammed Ghanbari. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 171–208. ISBN: 978-3-642-02900-4. DOI: 10.1007/978-3-642-02900-4_8. URL: https://doi.org/10.1007/978-3-642-02900-4_8.

McKay, M. D., R. J. Beckman, and W. J. Conover (1979). "A Comparison of Three Methods for Selecting Values of Input Variables in the Analysis of Output from a Computer Code". In: *Technometrics* 21.2, pp. 239–245. ISSN: 00401706. URL: http://www.jstor.org/stable/1268522 (visited on 06/25/2022).

Melis, Gábor, Chris Dyer, and Phil Blunsom (2017). *On the State of the Art of Evaluation in Neural Language Models*. DOI: 10.48550/ARXIV.1707.05589. URL: https://arxiv.org/abs/1707.05589.

Mukherjee, Sayan et al. (2003). "Estimating dataset size requirements for classifying DNA microarray data". In: *Journal of computational biology* 10.2, pp. 119–142.

Navarro, Julio F., Carlos S. Frenk, and Simon D. M. White (1996). "The Structure of Cold Dark Matter Halos". In: *The Astrophysical Journal* 462, p. 563. DOI: 10.1086/177173. URL: https://doi.org/10.1086%2F177173.

Nguyen, Dang et al. (2019). *Bayesian Optimization for Categorical and Category-Specific Continuous Inputs*. DOI: 10.48550/ARXIV.1911.12473. URL: https://arxiv.org/abs/1911.12473.

Orhan, A Emin and Xaq Pitkow (2017). "Skip connections eliminate singularities". In: *arXiv preprint arXiv:1701.09175*.

Pelt, Daniël M and James A Sethian (2018). "A mixed-scale dense convolutional neural network for image analysis". In: *Proceedings of the National Academy of Sciences* 115.2, pp. 254–259.

Rasti, Pejman et al. (2016). "Convolutional Neural Network Super Resolution for Face Recognition in Surveillance Monitoring". In: *Articulated Motion and Deformable Objects*. Ed. by Francisco José Perales and Josef Kittler. Cham: Springer International Publishing, pp. 175–184. ISBN: 978-3-319-41778-3.

Rigamonti, Fabio et al. (July 2022). "Maximally informed Bayesian modelling of disc galaxies". In: *MNRAS* 513.4, pp. 6111–6124. DOI: 10.1093/mnras/stac1326. arXiv: 2205.06819 [astro-ph.GA].

Robbins, Herbert E. (2007). "A Stochastic Approximation Method". In: *Annals of Mathematical Statistics* 22, pp. 400–407.

Rosenblatt, Frank (1958). "The perceptron: a probabilistic model for information storage and organization in the brain." In: *Psychological review* 65 6, pp. 386–408.

Shah, Amar and Zoubin Ghahramani (2015). *Parallel Predictive Entropy Search for Batch Global Optimization of Expensive Objective Functions*. DOI: 10.48550/ARXIV.1511.07130. URL: https://arxiv.org/abs/1511.07130.

Shahriari, Bobak et al. (2016). "Taking the Human Out of the Loop: A Review of Bayesian Optimization". In: *Proceedings of the IEEE* 104.1, pp. 148–175. DOI: 10.1109/JPROC.2015.2494218.

Shorten, Connor and Taghi Khoshgoftaar (July 2019). "A survey on Image Data Augmentation for Deep Learning". In: *Journal of Big Data* 6. DOI: 10.1186/s40537-019-0197-0.

Simonyan, Karen and Andrew Zisserman (Sept. 2014). "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *arXiv 1409.1556*.

— (2015). "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *CoRR* abs/1409.1556.

Singh, Dalwinder and Birmohan Singh (2020). "Investigating the impact of data normalization on classification performance". In: *Applied Soft Computing* 97, p. 105524.

Skilling, John (2006). "Nested sampling for general Bayesian computation". In: *Bayesian analysis* 1.4, pp. 833–859.

Smith, Leslie N (2018). "A disciplined approach to neural network hyper-parameters: Part 1–learning rate, batch size, momentum, and weight decay". In: *arXiv preprint arXiv:1803.09820*.

Snoek, Jasper, Hugo Larochelle, and Ryan P. Adams (2012). "Practical Bayesian Optimization of Machine Learning Algorithms". In: *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 2*. NIPS'12. Red Hook, NY, USA: Curran Associates Inc., 2951–2959.

Snoek, Jasper et al. (2015). "Scalable Bayesian Optimization Using Deep Neural Networks". In: *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*. ICML'15. Lille, France: JMLR.org, 2171–2180.

Souza, Samuel Xavier-de et al. (Aug. 2009). "Coupled Simulated Annealing". In: *IEEE transactions on systems, man, and cybernetics. Part B, Cybernetics : a publication of the IEEE Systems, Man, and Cybernetics Society* 40, pp. 320–35. DOI: 10.1109/TSMCB.2009.2020435.

Tong, Tong et al. (2017). "Image super-resolution using dense skip connections". In: *Proceedings of the IEEE international conference on computer vision*, pp. 4799–4807.

Tsai, Jinn-Tsong, Jyh-Horng Chou, and Tung-Kuan Liu (2006). "Tuning the structure and parameters of a neural network by using hybrid Taguchi-genetic algorithm". In: *IEEE Transactions on Neural Networks* 17.1, pp. 69–80. DOI: 10.1109/TNN.2005.860885.

Tsangaratos, Paraskevas and Ioanna Ilia (2016). "Comparison of a logistic regression and Naïve Bayes classifier in landslide susceptibility assessments: The influence of models complexity and training dataset size". In: *Catena* 145, pp. 164–179.

Vaswani, Ashish et al. (2017). "Attention is all you need". In: *Advances in neural information processing systems* 30.

Wang, Xintao et al. (2018). "Esrgan: Enhanced super-resolution generative adversarial networks". In: *Proceedings of the European conference on computer vision (ECCV) workshops*, pp. 0–0.

Wang, Zhihao, Jian Chen, and Steven C. H. Hoi (2021). "Deep Learning for Image Super-Resolution: A Survey". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 43.10, pp. 3365–3387. DOI: 10.1109/TPAMI.2020.2982166.

Werbos, Paul (Jan. 1970). "Applications of advances in nonlinear sensitivity analysis". In: vol. 38, pp. 762–770. ISBN: 3-540-11691-5. DOI: 10.1007/BFb0006203.

Zeiler, Matthew D. and Rob Fergus (2014). "Visualizing and Understanding Convolutional Networks". In: *Computer Vision – ECCV 2014*. Ed. by David Fleet et al. Cham: Springer International Publishing, pp. 818–833. ISBN: 978-3-319-10590-1.

Zhang, Liangpei et al. (2010). "A super-resolution reconstruction algorithm for surveillance images". In: *Signal Processing* 90.3, pp. 848–859. ISSN: 0165-1684. DOI: https://doi.org/10.1016/j.sigpro.2009.09.002. URL: https://www.sciencedirect.com/science/article/pii/S0165168409003776.