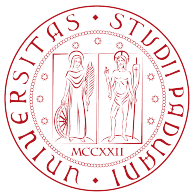

Università degli Studi di Padova

DIPARTIMENTO DI FISICA E ASTRONOMIA GALILEO GALILEI

Corso di Laurea magistrale in Astronomia

Master Thesis

Deep Convolutional Neural Networks in Astrophysics: a case study for gas turbulence

Candidate:
Piero Trevisan

Supervisor:
Prof. Michela Mapelli

Co-Supervisors:
Dr. Mario Pasquato
Dr. Alessandro Ballone

Anno Accademico 2018-2019

ABSTRACT

Machine learning has found many applications in astronomy, usually in the automation of tedious tasks. We show here how can we use machine learning algorithm, in particular deep convolutional neural networks (CNNs), as a tool of inference of physical information on a system. In particular, we translate the question of what is the powerspectrum of turbulent gas to a supervised regression problem that can be solved via deep learning. In order to do so, we trained a CNN on the output of gas simulations made with the hydrodynamic RAMSES. Then, we show the predictions on unseen gas. We compared with the predictions made with a standard computer vision library and show that the CNNs performs better. Finally, we discuss the possibility and the limits of this new approach.

CONTENTS

1 INTRODUCTION

Machine Learning (ML) is becoming increasingly popular in astronomy: for example in the characterization of exoplanets (e.g. Davies et al., 2015), the characterization of variable stars (e.g. Armstrong et al., 2015) or studying the cosmic ray propagation (e.g. Jóhannesson et al., 2016). In general, ML can be applied in all sort of tasks where a machine would perform quickly, and often better, than traditional methods in automating tasks on large datasets from surveys. Especially Deep Learning (DL) recently, has been successfully applied to astronomical problems using convolutional neural networks (Fukushima and Miyake, 1982, LeCun, Boser, et al., 1989, LeCun, Bottou, et al., 1998), for example, for real/bogus separation (Cabrera-Vives et al., 2016 2017), photometry computation for supernova classification(Kimura et al., 2017), calculation of an image comparable to the difference image (Sedaghat and Mahabal, 2017), gravitational wave detection (George et al., 2017) and exoplanet detection (Shallue and Vanderburg, 2018). An interesting, proactive approach with ML is taking form, e.g. in generating models for recover features in astrophysical images (e.g. galaxies in Schawinski, Zhang, et al., 2017), find black hole subsystems in globular clusters from mock observations (e.g. Askar et al., 2018) or to generate artificial data to test hypotheses for physical processes (e.g.in galaxy evolution Schawinski, Turp, et al., 2018).
In this work, the last approach is taken. We want to show here the first step to the process of inference of a physical parameter via DL. In particular, reducing the problem of finding the powerspectrum of a turbulent gas to a regression problem that we can solve with the training of a neural net. This training is done on the output of gas simulations. We can, then, apply our trained CNN to infere physical parameters on raw data.

Gas turbulence is interesting because spectroscopic observations of molecular lines in molecular clouds (MCs) show that their linewidth is much broader than the simple thermal linewidth. This effect is underdstood as the role of turbulence inside molecular clouds. The perturbation of hydrostatic equilibrium inside MCs can become gravitational unstable and eventually form stars.

We present now the layout of the thesis: in the second chapter, we describe the astrophysical background of molecular clouds and the turbulence. In the third chapter, we outline what is machine learning and describe in details neural networks, with particular regard to deep convolutional neural networks. In the fourth chapter, we present the gas simulations used and the convolutional neural network used to retrieve the turbulence index. The predictions of the CNN are shown in chapter five, followed by a discussion on the limits of this approach. Finally, in the sixth chapter we draw our conclusions.

2 ASTROPHYSICAL BACKGROUND

In this chapter we describe the general properties of turbulent molecular clouds (MCs) focusing on turbulence.

Stars form within MCs but our understanding of this fundamental process remains hampered by the complexity of the physics that drives their evolution. After a quick presentation of the general properties of MCs, we emphasize the relevant importance of turbulence in these objects.

2.1 MOLECULAR CLOUDS AND STAR FORMATION

There is no such thing as a predictive theory of star formation: given certain initial conditions, e.g., the density and temperature distributions inside an interstellar cloud, it is not possible to predict with certainty the star formation efficiency and the resulting initial mass function. Therefore, we rely mostly on observations to answer these doubts. Observations give us this picture: MCs are turbulent interstellar clouds almost entirely made of hydrogen with temperatures of $10 - 100$K, dimensions of ~ 10pc and densities of 10-100 molecules/cm^3. These values allow the formation of molecular hydrogen and a small quantity of other molecules, CO being among the most abundant and often the most visible. The small fraction of the cloud (about one percent) in the form of dust makes the MCs very opaque in the visual spectrum so we rely on infrared and radio observations.

Despite MCs are made of molecular hydrogen, direct H_2 observations are difficult. Electronic transitions occur in the ultraviolet, to which the Earth's atmosphere is opaque. Molecular vibrational and rotational transitions are faint because of their quadrupolar origin and the mid-infrared rotational lines occur at wavelengths where Earth's atmosphere is at best only partly transparent. Therefore, we measure H_2 indirectly from the carbon monoxide (CO), since there is a good correlation between the integrated intensity of the $J = 1 \rightarrow 0$ rotational transition line of CO and the H2 column density intensity (e.g. Glover and Low, 2011 and references therein). Figure 1 shows a map in this line for the Orion-Monoceros complex

The masses of MCs inferred from CO are of the order of $10^5 M_\odot$. If the mass of any cloud (or subclump) is larger than the Jeans mass:

$$M_{jeans} = 4 \times 10^4 M_\odot \left(\frac{T}{100K}\right)^{3/2} \left(\frac{n}{cm^{-3}}\right)^{-1/2}, \quad (1)$$

where n is the numerical molecular density of the cloud, this undergoes to gravitational instabilities. For typical values of T and n in MCs, M_{jeans} 10^3-10^4 $M_\odot$. Nearby supernovae explosions or collisions with other clouds create perturbations that can trigger these instabilities. These perturbations disturb the hydrostatic equilibrium, thus the MC start to collapse under its self-gravity. The collapse happens on a free-fall timescale that depends on the density of the gas ($t_{ff} \propto \rho^{-1/2}$). The density of a typical MC implies a timescale of the order of $\sim$ 1 Myr. At this stage, the collapsing MC is transparent to infrared radiation (far IR) and therefore the cooling is very

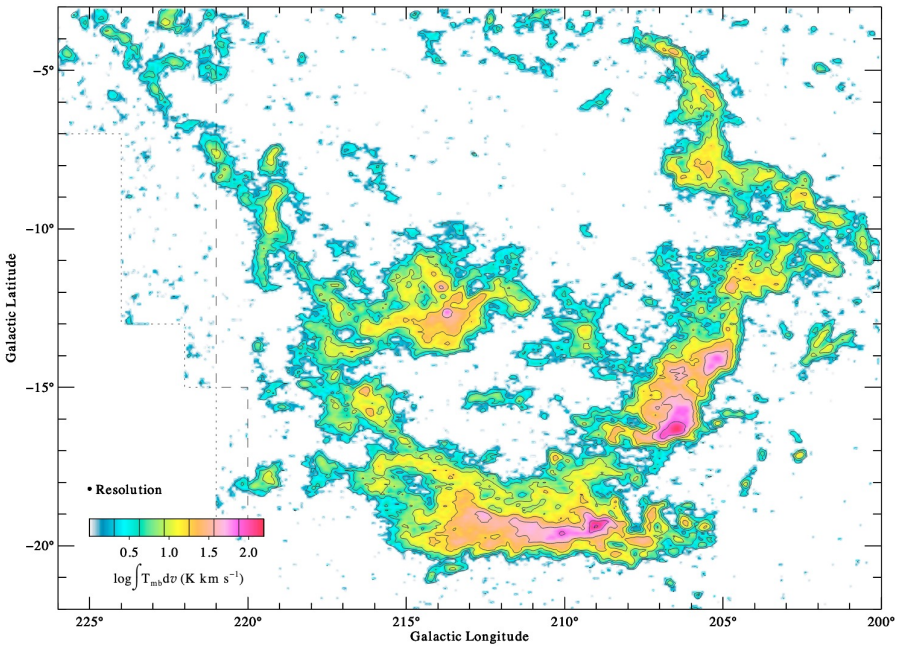

Figure 1: Integrated map of line J1 $\rightarrow$ 0 CO line for the Orion-Monoceros complex. All credits Wilson et al., 2005.

efficient. This is very important, since we can treat the gas in the cloud as an isothermal gas at this stage. As the collapse advances, the gas density increases, therefore the Jeans mass decreases and the Jeans instability is met on smaller regions, allowing local density peaks on smaller and smaller scales to collapse, in a process called fragmentation. Turbulence has the fundamental role of creating these local density perturbations and of "fragmenting" the cloud into smaller and smaller subclouds, that finally collapse into stars..

CO linewidths, from spectroscopic observations of MCs, are larger on average than what we would expected ones from the thermal motions of a gas at $T \sim 10-100K$. Larson (1981) found out the following relation between the observed linewidth σ and the size of the cloud (or of its sub-clumps) L:

$$\sigma = 0.5 \left(\frac{L}{1.0\mathbf{pc}} \right)^{0.5} \text{km/s}. \tag{2}$$

Figure 2 shows this relation from the original paper by Larson.

This broadening of the lines is understood as the active role of turbulence in MCs.
Star formation is far away to be solved, so it is interesting to further investigate the properties of turbulence in molecular clouds.

2.2 TURBULENCE: QUICK OVERVIEW

The mean free path of gas particles in MCs is smaller than the size L of the MC. Therefore, it is reasonable to approximate MCs as a fluid. The dynamic of a fluid is described by the Navier Stokes equations. Under the

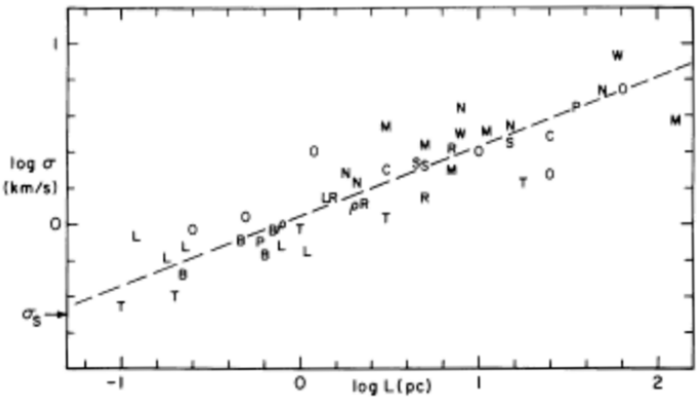

Figure 2: Three dimensional internal velocity dispersion σ against the size of the cloud L (or its sub-clumps) for different MCs (Larson, 1981)

assumption of a non viscous fluid without external forces, the Navier Stokes equations are reduced to the Euler equations:

$$\frac{\partial \rho}{\partial t} + \boldsymbol{\nabla} \cdot (\rho \mathbf{u}) = 0, \tag{3}$$

$$\frac{\partial \rho \mathbf{u}}{\partial t} + \boldsymbol{\nabla} \cdot (\rho \mathbf{u} \otimes \mathbf{u} + P\mathbb{1}) = 0, \tag{4}$$

$$\frac{\partial E}{\partial t} + \boldsymbol{\nabla} \cdot [\mathbf{u}\,(E + P)] = 0, \tag{5}$$

where ρ is the fluid density, $\mathbf{u}$ is the fluid velocity, P is the gas pressure and

$$E = \rho\epsilon + \frac{1}{2}\rho \mathbf{u} \cdot \mathbf{u} \tag{6}$$

is the total energy per unit volume, with ϵ being the internal energy. The Euler equations written in this form are called "conservative" Euler equations. The previous system of equations can be solved, at least with numerical methods, with the specification of an equation of state (EoS):

$$P = P(\rho, \epsilon). \tag{7}$$

Under the assumption of an ideal gas, whose EoS can be written as $P = (\gamma - 1)\rho\epsilon$, where γ is the fluid adiabatic index, the Euler equation can be written in the form:

$$\frac{\partial \rho}{\partial t} + \mathbf{u} \cdot \boldsymbol{\nabla}\rho + \rho \boldsymbol{\nabla} \cdot \mathbf{u} = 0, \tag{8}$$

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \boldsymbol{\nabla}\mathbf{u} + \frac{1}{\rho}\boldsymbol{\nabla}P = 0, \tag{9}$$

$$\frac{\partial P}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{P} + \gamma P \boldsymbol{\nabla} \cdot \mathbf{u} = 0. \tag{10}$$

This is the so-called quasi-linear form of the Euler equation. This system can be linearized, leading to the dispersion relation:

$$c_s^2 = \frac{\gamma P}{\rho} \tag{11}$$

of sound waves.

If we work with a non-ideal gas, physics is much more complex: Eq. 4, when the fluid viscosity is considered, becomes:

$$\frac{\partial \rho \mathbf{u}}{\partial t} + \boldsymbol{\nabla} \cdot (\rho \mathbf{u} \otimes \mathbf{u} + P\mathbb{1}) = \boldsymbol{\nabla} \cdot (\mu \mathbb{D}), \tag{12}$$

where $\mathbb{D}$ is a tensor describing the deformation of the fluid, μ is the dynamic viscosity coefficient. This coefficient can be expressed through the viscous term:

$$\nu_{visc} = \frac{\mu}{\rho}, \tag{13}$$

The importance of this nonideal effect on the system can be estimated through the viscous Reynolds number,

$$Re = \frac{U_L L}{\nu_{visc}}, \tag{14}$$

where U_L and L are typical velocity and length scales respectively. For $Re \gg 1$ viscosity can be ignored and the flow is likely to become turbulent. But what is turbulence? Reynolds number can be seen as a description of how fast the fluid is moving relative to how viscous it is: for small Reynolds number we are in the so-called laminar regime, i.e., the fluid flows in parallel layers, with no disruption between the layers. As Re increases, the viscous forces start to be negligible and the pattern of the fluid flows is characterized by chaotic changes in pressure and velocity, becoming turbulent. This can be used as an operative definition of turbulence. Turbulence causes the formation of unstable eddies of many different length scales. These eddies transfer energy to smaller scales which eventually become unstable too and transfer their kinetic energy to even smaller scales. This process can be described by the so-called Kolmogorov model.

2.2.1 Kolmogorov's Model

A turbulent flow is composed by different eddies characterized by a scale l and a velocity U_l. Kolmogorov (1941) noted that, for high Reynolds number, dissipation is unimportant, meaning that the large scale eddies must transfer kinetic energy to smaller eddies since they are unstable. This process goes on and on until we arrive at scales where $Re \sim 1$ is reached and dissipation becomes significant. Furthermore, a turbulent flow must be sustained through a persistent source of energy in order to keep being turbulent, since it is unstable and the energy is dissipated in the smallest scales. This produces a distribution of kinetic energy over different scales. In order to understand and quantitatively study a complex, multiscale phenomenon we adopt a statistical approach. In a turbulent fluid the medium velocity fluctuates in time and space so we need some simple assumptions. First we assume that the turbulence is homogeneous. Second, we assume that the turbulence is isotropic. This is true at least for the intermediate and small scales as noted by Kolmogorov. The geometrical and directional information on the large energy injection scale is lost going to smaller scales, so that the statistics of the small scales has a universal character. Now, let $\mathbf{v}(\mathbf{x})$ be

the velocity at position x within some volume of interest V. Let's define the two-point autocorrelation function:

$$A(\mathbf{r}) = \frac{1}{V}\int_V \mathbf{v}(\mathbf{x})\cdot\mathbf{v}(\mathbf{x}+\mathbf{r})d\mathbf{x}. \tag{15}$$

Since we are working under the isotropic hypothesis, $r = |\mathbf{r}|$, thus $A(r)$ tells us how similar or different the velocities are at points separated by some distance r.

We then consider the Fourier transform of the velocity field:

$$\tilde{\mathbf{v}}(\mathbf{k}) = \frac{1}{(2\pi)^{3/2}}\int \mathbf{v}(\mathbf{x})e^{-i\mathbf{k}\cdot\mathbf{x}}d\mathbf{x}. \tag{16}$$

Then, we define the power spectrum as

$$\Psi(\mathbf{k}) = |\tilde{\mathbf{v}}(\mathbf{k})|^2. \tag{17}$$

For isotropic turbulence, the power spectrum depends only on the magnitude of the wave number, $k=|\mathbf{k}|$. It's common practice to use the power per unit radius in the k-space,

$$P(k) = 4\pi k^2\Psi(k). \tag{18}$$

$P(k)$ from now on will be called energy power spectrum. Using the Parseval's theorem, we have that:

$$\int P(k)dk = \int |\tilde{\mathbf{v}}(\mathbf{k})|^2 d\mathbf{k} = \int \mathbf{v}(\mathbf{x})^2 d\mathbf{x}, \tag{19}$$

i.e., the integral of the power spectral density over all wavenumbers is equal to the integral of the square of the velocity over all space. Why is this important? Because, for a flow with constant density the integral of the power spectrum tells us how much kinetic energy per unit mass there is in the flow. Moreover, thanks to the Wiener-Khinchin theorem we have:

$$\Psi(\mathbf{k}) = \frac{1}{(2\pi)^{3/2}}\int A(\mathbf{r})e^{-i\mathbf{k}\cdot\mathbf{r}}d\mathbf{r}. \tag{20}$$

This is crucial: the power spectrum at a wavenumber k tells us what fraction of the total power is in motions at that wavenumber, i.e., on the characteristics length scale l. A power spectrum that peaks at low k means that most of the turbulent power is in large-scale motions (small k means large l). Viceversa, a power spectrum that peaks at high k means that most of the power is in small-scale motions.

Kolmogorov speculated that the statistics of a turbulent flow, described through the energy power spectrum $P(k)$, for k ranging between the smaller scale, decided by ν_{visc} and the energy dissipation ψ, and the larger scale L of the system, are only determined by the scale k and ψ. The rate of energy dissipation ψ (and also the rate of energy injection) has units of energy per unit mass per unit time, so L^2/T^3. Instead, the energy power spectrum $P(k)$ has units of energy per unit mass per unit radius, so L^3/T^2, and is a function only of k and ψ (at least for $Re \gg 1$), thus we can write $P(k) = Ck^\alpha\psi^\beta$ with C being a dimensionless constant. Using simple dimensional analysis we have:

$$\frac{L^3}{T^2} \sim L^{-\alpha}\left(\frac{L^2}{T^3}\right)^{\beta}$$
$$L^3 \sim L^{-\alpha+2\beta}$$
$$T^{-2} \sim T^{-3\beta}$$
$$\beta = \frac{2}{3}$$
$$\alpha = 2\beta - 3 = -\frac{5}{3}$$

Therefore, $P(k) = C\psi^{2/3}k^{-5/3}$. Numerous experiment on Earth confirm this behaviour (see fig. 3). In general, if $P(k) \propto k^{-\alpha}$, $\Psi(k) \propto k^{-\alpha-2}$ therefore

$$\Psi(k) \propto k^{-n}. \tag{21}$$

Hence, in the case of the Kolmogorov's model, n equals to $n = 11/3 = 3.\bar{6}$ From now on we will call n *turbulence index*. This dependency is valid only in the so-called inertial range, i.e., at k between the dissipation range and the large scales where there is the energy injection. Kolmogorov's model is ideal and describes only subsonic and uncompressible fluids. For compressible fluids situation is more complex, like we will see in the next section.

2.2.2 Burgers Turbulence

In order to study dynamic of turbulent supersonic compressible flows we rely on numerical simulations. E.g. Federrath et al. (2010), predict an energy power spectrum closer to $P(k) \propto k^{-2}$ (see fig. 5) which is much steeper than the Kolmogorov prediction and exactly equal to Burgers turbulence (Burgers, 1948).

The latter consists of a network of discontinuities (shocks), which can only form in supersonic flows. In fact, in molecular clouds we do not only have $Re \gg 1$ but also Mach Number (ratio between velocity and c_s) $M \gg 1$for $Re \gg 1$. This means that, on large scales, the gas is moving at supersonic velocities without any viscosity in ballistic fashion. These fast moving volumes of fluid will overtake slower ones on smaller scales but at these small scales viscosity becomes increasingly important ($Re \sim 1$). The viscosity eventually stops the fluid to move ballistically. This leads to the formation of discontinuities (shocks). We model a shock as an Heavyside function, whose power spectrum is proportional to k^{-2}. In one dimension this can be shown easily: the Fourier transform of v is:

$$\tilde{v}(k) = \frac{1}{\sqrt{2\pi}}\int v(x)e^{-ikx}dx. \tag{22}$$

The integral of term e^{-ikx} vanish for all periods where v is constant and is non-zero only in the period that includes the shock. During that period the amplitude $\int v(x)e^{-ikx}dx$ is proportional to the length, i.e. $1/k$. Thus $\tilde{v}(k) \propto 1/k$. From equation 19, it follows that $P(k) \propto k^{-2}$ and $\Psi(k) \propto k^{-4}$.

As a final note it is interesting that we can link the energy power spectrum to the dispersion velocity. The latter is well correlated with the sizes of MCs (or its sub-clumps) through the Larson relation (see equation 2). This helps

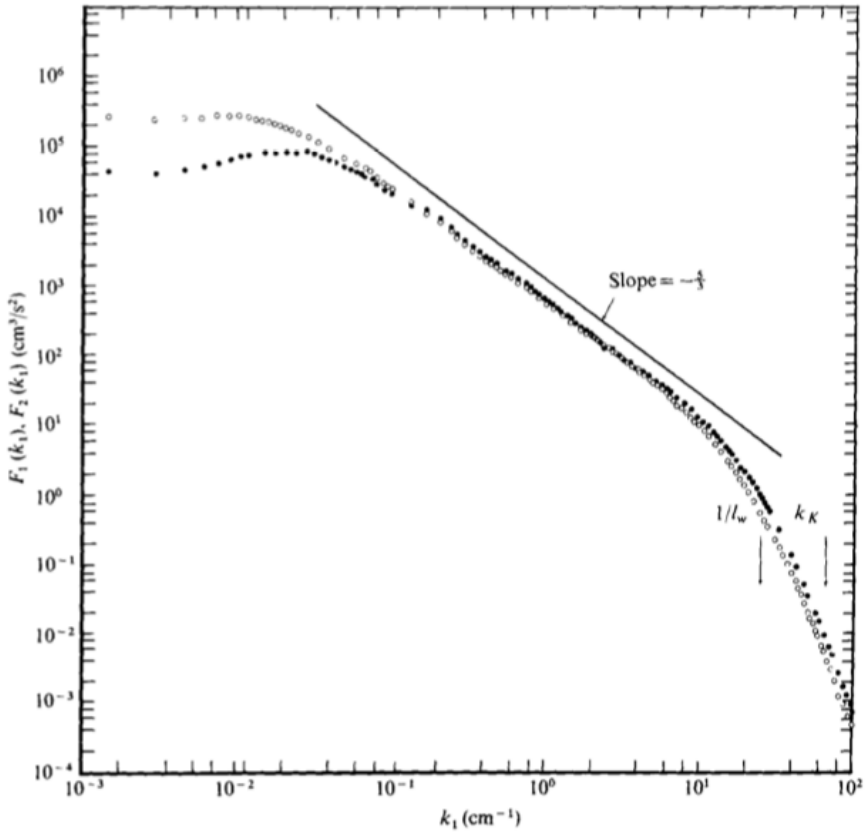

Figure 3: Power spectrum for turbulence generated by an air jet as a function of the wavenumber in logarithmic scale. The open and filled points show the velocity power spectrum for the velocity components parallel and transverse to the stream, respectively. Note the deviation from the inertial regime (slope = $-\frac{5}{3}$) for small and large log(k). Champagne, 1978

.

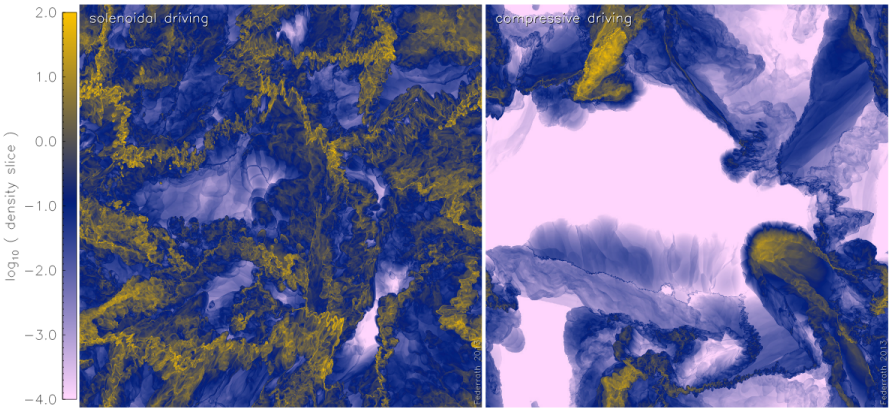

Figure 4: Slices through the three-dimensional gas density for highly compressed supersonic turbulence generated in two method: solenoidal driving (left panel) and compressive driving (right panel). The gas simulation has a resolution of 4096^3 cells. Reproduced from Christoph Federrath (2013).

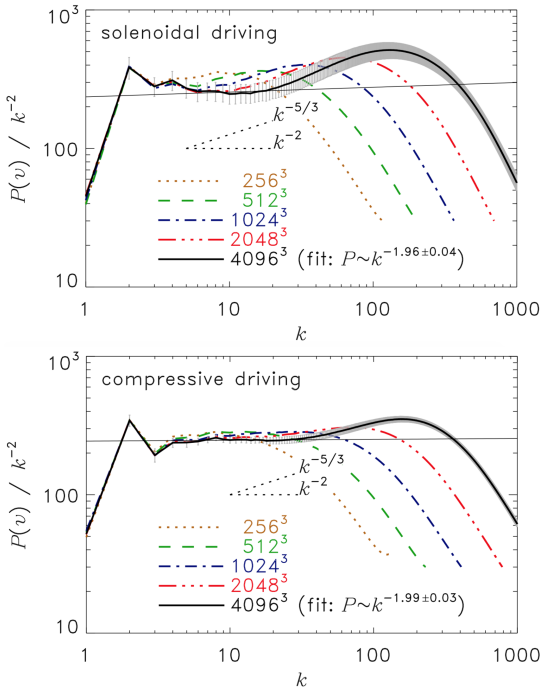

Figure 5: Power spectra $P(v)/k^{-2}$ for solenoidal driving (top panel) and compressive driving (bottom panel). Line styles shows the different resolutions. The fit in the inertial range is reported on the bottom right part of each panel: solenoidal driving yields $P(v) \propto k^{-1.96 \pm 0.04}$ and compressive driving yields $P(v) \propto k^{-1.99 \pm 0.03}$ (taken from Christoph Federrath, 2013).

to relate the observational properties of the MC (dispersion velocity) to the turbulence in it. In general, if we suppose to have an energy power spectrum in the form $P(k) \propto k^{-n}$, a volume of size L and a measure of its internal velocity dispersion $\sigma(L)$ within it, the total kinetic energy (per unit mass) will be:

$$E_k \sim \sigma(L)^2, \tag{23}$$

where we neglected factors of order unity. If we integrate over the modes with size $l < L$, i.e. $k > 2\pi/L$, we can write E_k in terms of the energy power spectrum:

$$E_k \sim \int_{2\pi/L}^{\infty} P(k)dk \propto k^{-n+1} \propto L^{n-1}. \tag{24}$$

It follows from the previous equations that:

$$\sigma \propto L^{(n-1)/2} \tag{25}$$

Therefore for a Kolmogorov law, we are left with $\sigma \propto L^{1/3}$ and for Burgers turbulence $\sigma \propto L^{1/2}$. Larson noted that the observed relation, for its group of MCs, $\sigma \propto L^{0.38}$ is different from more recent observations (e.g. Heyer Brunt, 2004) that found out the relation in equation 2. The dependance is $\sigma \propto L^{1/2}$, i.e. favouring Burgers turbulence. This might be explained if the observed motions in molecular clouds are actually due to subsonic or midly supersonic turbulence in a warmer medium. The question is still open.

3 DEEP LEARNING

In this chapter, after a brief introduction on what is ML, I describe Deep Neural Networks (DNNs) and especially the Convolutional Neural Networks (CNNs).

3.1 INTRODUCTION TO MACHINE LEARNING

The fundamental idea behind machine learning is the building of algorithms that "learn" a mathematical model of sample data, known as "training data", in order to make predictions or decisions without being explicitly programmed to perform the task. In order to do so, we have a training set of data, in which we know the relationship between input and output. Using this data we build a prediction model (such as a neural network), which will enable us to predict the outcome for new unseen objects (test set). This approach is the supervised learning one: we have the "labels" in the training set to drive the learning process. More formally (Mitchell et al., 1997): "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E."

3.1.1 The Task

ML tasks are usually described in terms of how the algorithm should process an example. The latter, in ML context, is a collection of features that have been measured or are known (pixels of an image, age of a population and so on). Since ML improved significantly with the number of examples, we want a lot of them. We represent the i-th example as a vector $\mathbf{x}^{(i)}$ where each entry $\mathbf{x}_j$ is the $j-th$ feature. The most common tasks that can be solved with machine learning are regression and classification (binary or multiclass). But ML applications are uncountable (transcription, machine translation, anomaly detection, sampling, imputation of missing values, denoising ...).

REGRESSION: The task T in a regression problem is the prediction of a numerical value given some input. Therefore, the ML algorithm is asked to output a function f from $\mathbb{R}^n$ to some real value. A completely disinterested example of regression is the prediction of the power spectrum index of a turbulent gas.

CLASSIFICATION: In this type of task, the algorithm has to find which of k categories some input belongs to. In order to do so, the ML algorithm is asked to produce a function f from $\mathbb{R}^n$ to some set $1, \ldots, k$. The input defined by vector $\mathbf{x}$ is mapped to a category defined by some value y.

3.1.2 The Performance measure

To measure the ability of our algorithm to perform said task T, we must use some quantitative measure of its performance. This performance measure P will be specific to the task T. For classification tasks, usually we measure the accuracy of the model simply calculating the proportion of examples for which the model produces the correct output. Vice versa, we can obtain the proportion of examples for which the model produces an incorrect output. In general, it's not straightforward to choose a perfomance measure that corresponds well to the desired behaviour of the system. E.g. when performing a regression task, should we penalize the system more if it frequently makes medium-sized mistakes or if it rarely makes very large mistakes?

3.1.3 The experience

In the formal definition by Mitchell et al. the idea is that the machine learns from some experience E. Based on what kind of experience the machine is allowed to have during the learning process, we can categorize learning as unsupervised or supervised. In almost all cases the learning algorithms are allowed to experience an entire dataset containing examples with many features. The difference is that in supervised learning the machine experiences a dateset containing features, but for each example there is an associated label. For example, our simulations have a label corresponding to the index of the power spectrum Ψ (see section 2.2.1).

3.1.4 Training set

The simplest way to describe the dataset is with a matrix $\mathbf{X} \in \mathbb{R}^{n \times m}$ where m is the number of examples and n the number of features. So that, X^i_j is the j-th feature of the i-th example. Since not always, the examples have the same dimensions (e.g. images with different widths and heights), we describe the dataset as a set with m examples: $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}$. In supervised learning a dataset carries with it its vector of labels $\mathbf{y}$, with $y^{(i)}$ providing the label for example i.

3.1.5 Test set

Usually we are interested in how well the machine learning algorithm performs on data that it has not seen before, since this determines how well it will work when applied to the "real world". We therefore evaluate these performance measures using a test set of data that is separate from the data used for training the ML system. It is important that the ML algorithm never sees this dataset during the training.

3.1.6 Validation set

A fraction of the dataset composed of examples coming from the same distribution as the training set, can be used to estimate the generalization error of the ML algorithm, after the learning process has completed. It's important that the test examples are not used in any way to make choices about the model, including its hyperparameters. For this reason, no example from the test set can be used in the validation set. Therefore, we construct the

validation set from the training set: we can split the training set in two set and hold out one for testing the performance of our algorithm with a certain choice of hyperparameters during the training itself. Commonly, one uses about 80% of the training data for training and 20% for validation. One can also divide the dataset in k distinct subsets, leaving $k-1$ subsets for training and the left one as validation set. Then, we repeat this operation other $k-1$ times taking another subset as validation set. We estimate the error by taking the average validation error across the k runs.

3.2 NEURAL NETWORKS

Over the years, a lot of prediction models have been developed for different tasks but the state-of-the-art algorithms in ML are Neural Networks (NNs). The name derives from the fact that they were first developed as models for the human brain (McCulloch and Pitts, 1943). Each connection, like the synapses in a biological brain, can transmit a signal from one artificial neuron to another. An artificial neuron that receives a signal can process it and then signal additional artificial neurons connected to it. Such systems, especially in Deep (Feedforward) Neural Networks (DNNs), can decompose the input into more and more complex features and identify trends that exist across all the training data and classify unseen data.
More technically, a DNN defines a mapping $y = f_{\mathbf{w}}(\mathbf{x})$ and learns the value of the parameters $\mathbf{w}$ that result in the best approximation of some function f^*. These DNN are called feedforward NNs because inputs flows through the network f being evaluated from the inputs, through the inner layers used to define f and finally to the output $\mathbf{y}$. Typically, they are represented by composing together different functions (layers) that create concretely a network. In a vanilla feedforward DNN we have $(L-1)$ functions connected in a chain: $f(\mathbf{x}) = f^{(L)}(f^{(L-1)}(\dots(f^{(2)}(\mathbf{x}))))$. The $f^{(l)}$ function is called l-th layer of the network, with the convention that $l = L$ is the output layer and l=1 correspond to the input layer (x). L is a natural number that gives the depth of the model. The layers between the input layer and the output layer are called hidden layers. The dimensionality of these hidden layers determines the width of the NN. Even if we can think the layer l as a function $f^{(l)}\colon \mathbb{R}^p \to \mathbb{R}^q$ where p and q are the dimensions (widths) of the $(l-1)$-th and l-th layer respectively, we usually represent layer l as q units that act in parallel with each one of them being a vector-to-scalar function. These units are the artificial neurons of the network.

The simplest model of these fundamental units, that from now on we refer to as artificial neurons or simply neurons, is the *perceptron*.

3.2.1 Perceptron

Perceptrons were proposed by Rosenblatt (1958), inspired by earlier work by McCulloch and Pitts (1943) and successively analyzed and refined by Minsky and Papert (1969).

The perceptron model, takes a vector $\mathbf{x} \in \mathbb{R}^n$ and computes a linear combination of its components x_i:

$$z = \sum_{i=1}^{n} w_i x_i = \mathbf{w}^T \mathbf{x}, \tag{26}$$

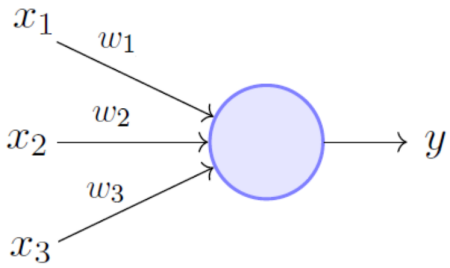

Figure 6: Layout of the perceptron model as presented by Minsky and Papert (1969). This scheme can be generalized with more than 3 features.

where $\mathbf{w}^T = [w_1, w_2, \ldots, w_n]$ are called *weights*. The classical perceptron then outputs y=1 if $z \geqslant 0$ and $y = 0$ otherwise. The weights can be learned in order to implement linearly separable function (OR, AND, and similar logical boolean function) but there is no solution for non-linearly separated data (XOR function) as shown by Minsky and Papert, 1969. Artificial neurons used today are slightly different from the original perceptron since we apply some activation function g, which is usualnon-linear (see sec. 3.2.3) like the sigmoid function. The output is no longer binary but a real value (between 0 and 1 in the case of the sigmoid function) that can be interpreted as a probability. The original perceptron was able only to answer yes/no decision, instead the upgraded version gives us the probability of "yes". In general, the output is a real number, depending on the choice of the activation function g and the task considered T, from 0 to 1, or from -1 to 1 or in general some real number.

Feedforward DNNs, also called multilayer perceptrons (MLPs), stack layers of these neurons where each successive layer uses the output of the previous layer as input, In the next section, I describe in detail the functioning of a feedforward DNN.

3.2.2 Architecture of Deep Neural Networks

In a DNN the output of certain neurons is connected to the input of other neurons forming a weighted graph (fig. 7). The weights are modified during the learning process driven by a process called backward propagation (3.3.3).

In general, if we denote with the apex l the layer l-th with $l \in [1, L]$, where l=1 is the input layer and l=L the output layer, the activation of the first neuron in the l-th layer will be:

$$a_1^l = g(z_1^l) = g(\sum_{i=0}^{p} w_{1i}^{(l-1)} a_i^{(l-1)}), \tag{27}$$

where p is the width of the l-th layer, g is a non-linear function and $z_1^l = \sum_{i=0}^{p} w_{1i}^{(l-1)} a_i^{(l-1)}$ is the feature vector. We used the convention that $a_0 = 1$ so that $w_0 \cdot 1$ is the bias unit. To get a better picture, let us describe explicitly the first hidden layer (l=2) of fig. 7:

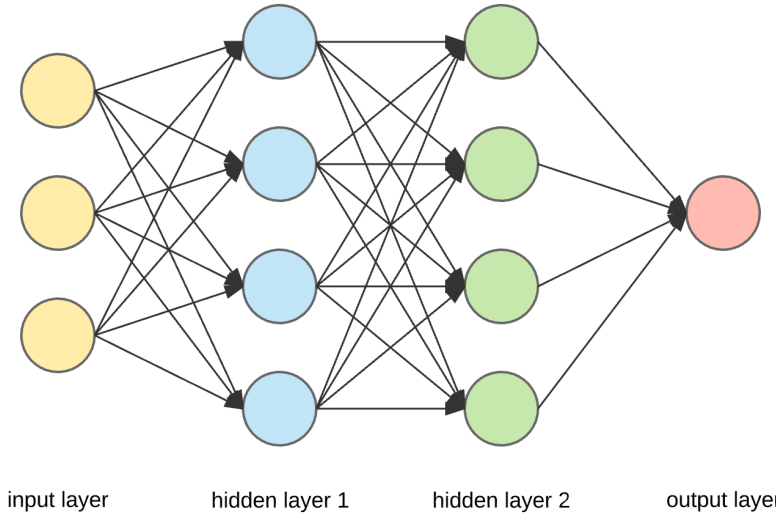

Figure 7: Layout of a fully connected DNN with two hidden layers. Every single hidden layer has four artificial neuron. The bias unit x_0 is not shown.

$$
\begin{aligned}
a_1^2 &= g(z_1^2) = g(w_{10}^1 x_0 + w_{11}^1 x_1 + w_{12}^1 x_2 + w_{13}^1 x_3) \\
a_2^2 &= g(z_2^2) = g(w_{20}^1 x_0 + w_{21}^1 x_1 + w_{22}^1 x_2 + w_{23}^1 x_3) \\
a_3^2 &= g(z_3^2) = g(w_{30}^1 x_0 + w_{31}^1 x_1 + w_{32}^1 x_2 + w_{33}^1 x_3) \\
a_4^2 &= g(z_4^2) = g(w_{40}^1 x_0 + w_{41}^1 x_1 + w_{42}^1 x_2 + w_{43}^1 x_3).
\end{aligned}
$$

More concisely, in vector notation the first hidden layer of activations can be represented as

$$\mathbf{a}^2 = \left[a_1^2, a_2^2, a_3^2, a_4^2\right]^T = g(\mathbf{z}^2) = g(\mathbf{w}^1 \cdot \mathbf{x}), \tag{28}$$

where

$\mathbf{x} = \left[x_0, x_1, x_2, x_3\right]^T = \left[1, x_1, x_2, x_3\right]^T$ is the input vector. Now, the activation of the first neuron of the third layer will be:

$$a_1^3 = g(z_1^3) = g(\mathbf{w}^3 \cdot \mathbf{a}^2) = g(w_{10}^2 a_0^2 + w_{11}^2 a_1^2 + w_{12}^2 a_2^2 + w_{13}^2 a_3^2 + w_{14}^2 a_4^2)$$

and so on for the others neurons. Note the addition of the bias unit a_0^2 for this layer. The output of this DNN and so its prediction will be:

$$h_{\mathbf{w}}(\mathbf{x}) = a^4 = g(z^4) = g(\mathbf{w}^3 \cdot \mathbf{a}^3) \tag{29}$$

In general $\mathbf{w}^l \in \mathbb{R}^{p \times q}$ where p is the dimension of the $(l-1)$-th layer and q the dimension of l-th layer.

Note that all features z^l in the hidden layers are learned features, meaning that a NN, instead of being constrained by the original input features, can learn its own features to feed into the successive layers. Therefore, it is essential to initialize all weights to small random values then then let the w weights change through iterative gradient-based optimization algorithm (see sec.3.3.3) letting the $\mathbf{w}$ weights change If we initialize all the weights with the same value (or even worse zero) all the weights, every hidden unit receives the same input, then all the the neurons of all the layers perform the same calculation, giving the same output. If we don't assign random values, or at least different values, the NN can not learn. The study of

the initialization of weights and its impact on the performance is still an open. For example, in Keras, the deep learning library used in this thesis, the default weights initializer is the Glorot uniform initializer, also called Xavier uniform initializer (Glorot and Bengio, 2010). The Glorot uniform initializer draws samples from a uniform distribution within $[-l, l]$, where $l = \sqrt{6/(p+q)}$ with p and q the number of input and output units of the layer, respectively. In any case, if the weights have initially different values, a NN has a lot of flexibility to learn whatever features it needs in order to improve its performance P, usually minimizing some cost function $J(\mathbf{w})$.

We kept hinting on this activation function. Let's now explore different activation functions.

3.2.3 Activation function

A hidden neuron computes an affine transformation $\mathbf{z} = \mathbf{W}^T\mathbf{x} + \mathbf{b}$, and then applies a nonlinear function $g(\mathbf{z})$.

The function g is called activation function. It defines the output of a neuron.

Nowadays, the most used activation function in the inner layers of DNNs is the Rectifier (ReLU) function.

$$g(z) = z^+ = max(0, z) \tag{30}$$

Also, these units are easy to optimize because they are so similar to linear units. The only difference between a linear unit and a ReLU unit is that a ReLU unit outputs zero across half its domain. This makes the derivatives through a rectified linear unit to remain large whenever the unit is active. The gradients are not only large but also consistent. The derivative of the rectifying operation is 1 wherever the unit is active. The second derivative of the rectifying operation is defined almost everywhere and it is 0 wherever it is defined. One drawback of ReLU units is that they cannot learn via gradient-based methods on examples for which their activation is zero. Various generalizations of ReLU units guarantee that the gradients are nonzero everywhere: when $z < 0$, $g(z) = max(0, z) + \alpha \cdot min(0, z)$. The most common is the leaky ReLU that fixes $\alpha = 0.01$ (Maas et al., 2013). The ReLU unit and its generalization works well on the principle that models are easier to optimize if their behavior is closer to linear (I. Goodfellow et al., 2016). Despite ReLU units are a good default choice other types of hidden units are available.

Prior to the introduction of ReLU, most NNs used the sigmoid function (logistic function):

$$g(z) = \sigma(z) = \frac{1}{1 + e^{-z}}, \tag{31}$$

or the hyperbolic tangent:

$$g(z) = tanh(z). \tag{32}$$

These functions where natural choices in binary classification, since they can be easily used to predict the probability that a binary variable is 1. As shown by Glorot, Bordes, et al. (2011), the ReLU outclasses the sigmoid in

the training of deeper NNs. Sigmoidal units, in fact, are strongly sensitive to their input when z is near 0 and moreover they saturate across most of their domain. This can slow down a lot the learning, and their use is not recommended in the inner layers (at least in Feedforward DNNs).

Other possible choices are:

- Linear: One possibility is not to have an activation g at all. We can use the identity function as the activation function. This can be useful in the output of a NN (like in the NN trained in chapter4)

- Softplus: smooth version of the ReLU $g(z) = \log(1 + e^z)$. Conterinuitively, Glorot, Bordes, et al. (2011) showed also that, despite its differentiation problems, ReLU beats its smooth counterpart: the Softplus is differentiable everywhere and saturate less, but empirically the ReLU performs better.

- Softmax: units that represent a probability distribution over a discrete variable with k possible values. These are the natural choice for multiple classes classification, but in more advanced architecture that learn to manipulate memory (Google Inception and GoogLeNet) they are used in the inner layers to also avoid vanishing gradients (I. Goodfellow et al., 2016).

3.2.4 Architecture Design

We can design NNs as the one described in 3.2 simply by stacking neurons layer after layer. In this case, the main considerations are the choice of the depth of the NN L and the width of each layer. Usually, deeper models tend to perform better I. J. Goodfellow et al., 2013 and with the use of far fewer units per layer can generalize better to the test set but also tend to be harder to optimize (I. Goodfellow et al., 2016). Therefore, the ideal NN architecture for a task must be found via experimentation by monitoring the validation set loss.

3.3 LOSS FUNCTION AND BACKWARD PROPAGATION

In supervised learning, we know the labels of our training data a priori; this is used to quantify the robustness of our network for predicting seen and unseen data. We want to measure the inconsistency between predicted value $\hat{y}$ and actual label y for all the examples in the dataset and find the values of $\mathbf{w}$ that minimize this inconsistency but at the same time are able to generalize to unseen data. This is represented as:

$$\begin{aligned}
\mathbf{w}^* &= \underset{\mathbf{w}}{\operatorname{argmin}}\, J(\mathbf{w}) + \lambda \cdot \Phi(\mathbf{w}) \\
&= \underset{\mathbf{w}}{\operatorname{argmin}}\, \frac{1}{m} \sum_{i=1}^{m} L(y^{(i)}, \hat{y}^{(i)}) + \lambda \cdot \Phi(\mathbf{w}) \\
&= \underset{\mathbf{w}}{\operatorname{argmin}}\, \frac{1}{m} \sum_{i=1}^{m} L(y^{(i)}, h_{\mathbf{w}}(\mathbf{x}^{(i)})) + \lambda \cdot \Phi(\mathbf{w}),
\end{aligned} \tag{33}$$

where $\Phi(\mathbf{w})$ is the regularization term, λ a regularization parameter, $\mathbf{w}$ the parameter to be learned and $J(\mathbf{w})$ the loss function. Leaving the regularization term for now, the output of a DNN will be some $h_{\mathbf{w}}(\mathbf{x})$ depending on the final activation function and $J(\mathbf{w})$ on how we want to measure the performance P on the task. Usually, in regression problems the standard performance measure is the mean squared error (MSE):

$$J(\mathbf{w}) = \frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)} - \hat{y}^{(i)}\right)^2, \tag{34}$$

where $\left(y^{(i)} - \hat{y}^{(i)}\right)$ is named residual.

It can be shown that minimizing the MSE yields the same estimate of maximizing likelihood estimation with respect to $\mathbf{w}$ (I. Goodfellow et al., 2016. This entails that the minimization of MSE has the same benefits of a maximum likelihood estimator: one of the most appealing is that as the number of examples $m \to \infty$, the rate of convergence of a maximum likelihood estimator increases and moreover the estimate of a parameter converges to the true value of the parameter (under some conditions).

The regularization term $\Phi(\mathbf{w})$ is a term added to the loss function in order to avoid overfitting. It is a modification to a ML algorithm that is intended to improve the ability of the algorithm to generalize to new data while not reducing its training error. Usually updates on the regularization term are done checking the performance of the ML on the validation set during each training step.

3.3.1 Linear Regression Model

The simplest ML algorithm we can think of is the linear regression model. The task T, as the name implies, is to solve a regression problem. Given a vector $\mathbf{x}$ as input we want to predict the value $y \in \mathbb{R}$ as its output. The output of the model will be:

$$\hat{y} = \mathbf{w}^T\mathbf{x}, \tag{35}$$

where $\mathbf{w} \in \mathbb{R}^n$ is the set of weights. As before, we included the bias in w_0 with $x_0 = 1$. This problem can be easily resolved with a single neuron without an activation function. In fact, the vector $\mathbf{z}$ of features is exactly the input of the linear model and without applying an activation function. So the $\hat{y} = h_{\mathbf{w}}(\mathbf{x}) = g(\mathbf{z}) = \mathbf{z} = \mathbf{w}^T\mathbf{x}$ is the prediction for the value y. As any ML algorithm we have to train on a same dataset $\mathbf{X}$ provided with a vector of labels $\mathbf{y}$ then improve $\mathbf{w}$ improve w by reducing the performance measure P, e.g., by minimizing the MSE. Even if the regression problem has an analytical solution (the system of normal equations) we show here an algorithm to improve the weights that can be generalized to more complex NNs called backward propagation.

3.3.2 Backward Propagation

When we use a feedforward NN to accept $\mathbf{x}$ and produce an output $\hat{y}$, information flows forward through the network in the so called forward propagation. During the training, forward propagation goes on until it produces an output from which we calculate the loss $J(\mathbf{w})$. The backpropagation algorithm (Rumelhart et al., 1986), or backward propagation or simply backprop,

allows the information of the cost function to flow backward through the network in order to compute the gradient of J with regards to $\mathbf{w}$. Backprop computes the gradient $\nabla_{\mathbf{w}} J(\mathbf{w})$, while another algorithm such as gradient descent is used to perform learning using this gradient.

The backprop simply computes the chain rule of calculus with a specific order of operations that is highly efficient. Suppose that $\mathbf{x} \in \mathbb{R}^m$, $\mathbf{y} \in \mathbb{R}^n$, g maps $\mathbb{R}^m$ to $\mathbb{R}^n$ and f $\mathbb{R}^m$ to $\mathbb{R}$. If $z = f(g(\mathbf{x}))$, then

$$\nabla_{\mathbf{x}} z = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}}\right)^T \nabla_{\mathbf{y}} z, \tag{36}$$

where $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$ is the $n \times m$ Jacobian matrix g. Since we usually work with tensors of arbitary dimensions in DNN we have to extend this concept to tensors. We can merely imagine to flatten each tensor into a vector before we run back-propagation then reshaping the gradient back into a tensor.

The backprop algorithm applies recursively the chain rule: in a fully connected MLP (like the one explained in section 3.2) this computation yields the gradient on the activations $\mathbf{a}^{(k)}$ for each layer k, starting from the output layer and going backwards to the first hidden layer. From these gradients, which can be interpreted as an indication of how each layer's output should change to reduce the loss, one can obtain the gradient on the parameters of each layer. The gradients on weights can be immediately used as part of a stochastic gradient descent or other optimization methods (see next section). This can be generalized with some precautions to any NN. For a simple example, in linear regression with two weights, the update rule will be:

$$w_j := w_j - \eta \frac{\partial}{\partial w_j} J(w_1, w_2), \tag{37}$$

for $j \in [1, 2]$ and η the learning rate.

Using the MSE as J(w) we get:

$$\begin{aligned}
\frac{\partial}{\partial w_j} J(w_0, w_1) &= \frac{\partial}{\partial w_j} \cdot \frac{1}{2m} \sum_{i=1}^{m} (h_w(x^{(i)}) - y^{(i)})^2 \\
&= \frac{\partial}{\partial w_j} \frac{1}{2m} \sum_{i=1}^{m} (w_0 + w_1 x^{(i)} - y^{(i)})^2 \\
j = 0 : \frac{\partial}{\partial w_0} J(w_0, w_1) &= \frac{\partial}{\partial w_0} J(w_0, w_1) = \frac{1}{m} \sum_{i=1}^{m} (h_w(x^{(i)}) - y^{(i)}) \\
j = 1 : \frac{\partial}{\partial w_1} J(w_0, w_1) &= \frac{1}{m} \sum_{i=1}^{m} (h_w(x^{(i)}) - y^{(i)}) \cdot x^{(i)}
\end{aligned} \tag{38}$$

Now, it's trivial to compute the update rule in 37. In linear regression $J(\mathbf{w})$ is always a convex function so it has always one global optima, therefore the convergence is guaranteed. Instead, in typical NNs and also in our work, the numbers of weights is $\sim 10^{6-7}$ meaning a high nonlinearity that causes complex and nonconvex loss functions. Therefore, iterative, gradient based method are always implemented. We drop the notation with single weights w_i preferring to group all the gradients. Henceforth, we will indicate the gradient of the loss function with respect to the weights as $\nabla_{\mathbf{w}} J(\mathbf{w})$.

3.3.3 Gradient descent

There are three variants of gradient descent, which differ in how much data is chosen to compute the gradient of the cost function. Depending on the amount of data, we make a trade-off between the accuracy of the parameter update and the time it takes to perform an update.

BATCH GRADIENT DESCENT Batch gradient descent (BGD), computes the gradient of the loss function with regard to the weights for the entire training dataset:

$$\mathbf{w} := \mathbf{w} - \eta \cdot \nabla_{\mathbf{w}} J(\mathbf{w}), \tag{39}$$

where η is called learning rate which is a hyperparameter that has to be tuned. BGD is very reliable but, as we need to compute all the gradients for the whole dataset only to perform one update, BGD can be very slow. We can upgrade BGD with the Scochastic Gradient Descent.

STOCHASTIC GRADIENT DESCENT Stochastic gradient descent (SGD) in contrast to BGD performs a parameter update for each training example $x^{(i)}$ and label $y^{(i)}$:

$$\mathbf{w} := \mathbf{w} - \eta \cdot \nabla_{\mathbf{w}} J(\mathbf{w}; \mathbf{x}^{(i)}; \mathbf{y}^{(i)}). \tag{40}$$

BGD is slow since it performs redundant computations for large datasets, as it recomputes gradients for similar examples before each parameter update. Instead, SGD performs one update at a time. It's also a lot faster than BGD but fluctuate heavily (Ruder, 2016). This can be a pro if the parameters are set to a bad local minimum: BGD converges to this minimum instead SGD can jump to a new and potentially better local minimum. The evident downside is the difficulty to converge to an exact minimum due to the continuous overshooting. However, Ruder (2016) has shown that when we slowly decrease the learning rate, SGD shows the same convergence as BGD, converging to a local or the global minimum for non-convex and convex optimization respectively.

MINI-BATCH GRADIENT DESCENT Mini-batch gradient descent performs an update for every mini-batch of n (with $n < m$) training examples:

$$\mathbf{w} := \mathbf{w} - \eta \cdot \nabla_{\mathbf{w}} J(\mathbf{w}; x^{(i:i+n)}; y^{(i:i+n)}). \tag{41}$$

The advantage of this gradient descent is double: firstly, it reduces the variance of the parameter updates, hence a more stable convergence. Secondly, usually deep learning libraries (KERAS in our case) have already implemented highly optimized matrix optimizations that make computing the gradient with regard to a mini-batch very efficient.

I'll explain in the next subsection the different optimizers used for this work.

3.3.4 Optimizers

Let's see first *momentum*.

MOMENTUM Momentum is a method that helps accelerate the SGD in the relevant direction (Qian, 1999) by adding a fraction γ of the update vector of the past step to the current update vector:

$$v_t = \gamma v_{t-1} + \eta \cdot \nabla_{\mathbf{w}} J(\mathbf{w}) \tag{42}$$

$$w_{t+1} := w_t - v_t. \tag{43}$$

The idea is that as we go down towards the minimum the momentum increases and so the the parameter updates. As a result, we gain faster convergence and reduced oscillation. Let's see now different optimizers that expand the concept of momentum and simple gradient descent.

ADAGRAD Adagrad (Duchi et al., 2011) is an algorithm for gradient-based optimization that adapts the learning rate to the parameters, performing smaller updates (i.e. low learning rates) for parameters associated with frequently occurring features, and larger updates (i.e. high learning rates) for parameters associated with infrequent features. We use different learning rate for every parameter w_i at every step t, so that the gradient g_t at step t will be:

$$g_{t,i} = \nabla_{\mathbf{w}} J(w_{t,i}). \tag{44}$$

The SGD update for every parameter w_i at each step t becomes:

$$w_{t+1,i} = w_{t,i} - \eta \cdot g_{t,i}. \tag{45}$$

Adagrad corrects the update rule by the diagonal matrix made of the sum of the squares of the gradients with regards to W_i up to the step t, i.e. :

$$w_{t+1,i} = w_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}, \tag{46}$$

where $G_t \in \mathbb{R}^{d \times d}$ is the diagonal matrix where each diagonal element $G_{t,ii}$ is s the sum of the squares of the gradients with regards to w_i and ϵ is a smoothing term that avoids division by zero. It is now straightforward to vectorize the implementation by performing the matrix vector product between G_t and g_t:

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{G_t + \epsilon}} g_t. \tag{47}$$

One of Adagrad's main benefits is that it eliminates the need to manually tune the learning rate. Most implementations use a default value of 0.01 and leave it at that. Adagrad's main weakness is its accumulation of the squared gradients in the denominator: Since every added term is positive, the accumulated sum keeps growing during training. This in turn causes the learning rate to shrink and eventually become infinitesimally small, at which point the algorithm is no longer able to acquire additional knowledge. The following algorithms aim to resolve this flaw.

ADADELTA Adadelta (Zeiler, 2012) is an extension of Adagrad that seeks to reduce its aggressive, monotonically decreasing learning rate. Instead of accumulating all past squared gradients, Adadelta restricts the window of accumulated past gradients to some fixed size W. Instead of inefficiently storing W previous squared gradients, the sum of gradients is recursively

defined as a decaying average of all past squared gradients. The running average $E[g^2]_t$ at step t then depends (as a fraction similarly to the Momentum term) only on the previous average and the current gradient:

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2. \tag{48}$$

We set γ to a similar value as the momentum term, around 0.9. Now, we can rewrite our SGD update in terms of the parameter update vector ΔW_t:

$$\begin{aligned} \Delta w_t &= -\eta \cdot g_{t,i} \\ w_{t+1} &:= w_t + \Delta w_t. \end{aligned}$$

The parameter update vector of Adagrad that we derived previously thus takes the form:

$$\Delta w_t = -\frac{\eta}{\sqrt{G_t + \epsilon}} g_t. \tag{49}$$

We now simply replace the diagonal matrix G_t with the decaying average over past squared gradients $E[g^2]_t$:

$$\Delta w_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t. \tag{50}$$

As the denominator is just the root mean squared (RMS) error criterion of the gradient, we can replace it with the criterion short-hand:

$$\Delta w_t = -\frac{\eta}{RMS[g]_t} g_t. \tag{51}$$

Zeiler (2012) note that the units in this update (as in SGD, Adagrad and momentum) do not match. In order to fix this, we can define another exponentially decaying average:

$$E[\Delta w^2]_t = \gamma E[\Delta w^2]_{t-1} + (1-\gamma) \Delta w_t^2. \tag{52}$$

The RMS of parameter updates is thus:

$$RMS[\Delta w]_t = \sqrt{E[\Delta w^2]_t + \epsilon}. \tag{53}$$

Since $RMS[\Delta w]_t$ is unknown, we approximate it with the RMS of parameter updates until the previous step. Replacing the learning rate η in the previous update rule $RMS[\Delta w]_{t-1}$ finally yields the Adadelta update rule:

$$\begin{aligned} \Delta wt &= -\frac{RMS[\Delta w]_{t-1}}{RMS[g]_t} g_t \\ w_{t+1} &= w_t + \Delta w_t \end{aligned}$$

With Adadelta, we do not even need to set a default learning rate, as it has been eliminated from the update rule.

RMSPROP RMSprop is an unpublished, adaptive learning rate method proposed by Geoff Hinton in Tieleman and G. Hinton, 2012. Both RMSprop and Adadelta were developed in order to resolve Adagrad's radically diminish-

ing learning rates. RMSprop in fact is identical to the first update vector of Adadelta that we derived in the previous paragraph:

$$E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1g_t^2$$
$$w_{t+1} := w_t - \frac{\eta}{\sqrt{E[g^2}_t + \epsilon} g_t.$$

RMSprop as well divides the learning rate by an exponentially decaying average of squared gradients. Hinton in [Tieleman and G. Hinton, 2012] suggests γ to be set to 0.9, while a good default value for the learning rate η is 0.001.

ADAM Adaptive Moment Estimation (Adam) (Kingma and Ba, 2014) is another method that computes adaptive learning rates for each parameter. Like Adadelta and RMSprop, it stores an exponentially decaying average of past squared gradients v_t, but in addition Adam also keeps an exponentially decaying average of past gradients m_t, similar to momentum. Whereas momentum can be seen as a ball running down a slope, Adam behaves like a heavy ball with friction, which thus prefers flat minima in the error surface. We compute the decaying averages of past and past squared gradients m_t and v_t respectively as follows:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2,$$

where m_t and v_t are estimates of the first momentum and the second momentym of the gradients respectively. The authors of Adam observe that, since m_t and v_t are initialized as vectors of zeros, they are are biased towards zero, especially during the initial steps, and especially when the decay rates are small (i.e. β_1 and β_2 are close to 1). A way to counter these biases is by computing bias-corrected first and second moment estimates:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

They then use these to update the parameters just as we have seen in Adadelta and RMSprop, which yields the Adam update rule:

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t. \tag{54}$$

The values proposed by the authors are 0.9 for β_1, 0.999 for β_2 and 10^{-8} for ϵ.
In the next chapter, we apply these various optimizers to our training set and tested the impact on the performance of our neural network (see section4.2.4).

3.3.5 Other NNs parameters

In the next chapters, we will refer sometimes to these words:

EPOCHS An epoch describes the number of times the algorithm sees the entire dataset. So, each time the algorithm has seen all examples in the dataset, an epoch has been completed. If the global minimum of the cost function is not found, usually increasing the number of epochs (with the right optimizer) can help minimizing the loss function.

BATCH SIZE The number of training examples in one forward/backward pass. The higher the batch size, the more memory space you'll need. When the batch size is equal to the number of examples in the training dataset we have usual batch gradient descent.

DROPOUT A regularization technique consisting in setting a random fraction ϵ of neurons inactive during the training. The neurons which are "dropped out" in this way do not contribute to the forward pass and do not participate in back- propagation. So every time an input is presented, the neural network samples a different architecture, but all these architectures share weights. This technique reduces complex co-adaptations of neurons, since a neuron cannot rely on the presence of particular other neurons. In fact, during the training the gradient received by each weight tells it how it should change so the final loss function is reduced, given what all other units are doing. In this way, units may change in a way that they fix up the mistakes of the other units. This in turn leads to overfitting because these co-adaptations do not generalize to unseen data. If we leave some units dropped during the training a hidden unit, therefore, is forced to learn more robust features since it cannot rely on other specific units to correct its mistakes. (G. E. Hinton et al., 2012).

3.4 CONVOLUTIONAL NEURAL NETWORK

Modern Convolutional Neural Networks (LeCun, Boser, et al., 1989), or CNNs are a specialized kind of NN for processing data that has a known grid-like topology (arrays of pixels, time series data). In all aspect they are very similar to ordinary NNs. They are composed of weights and biases, each neuron takes some input, perform a dot product and follows optionally with some non linearity. We can express the performance of our network with a loss function on the last layer and all the tricks and statement made for regular NNs still apply here. So why we use them? And why they work?

Since the winning of 2012 ImageNet competion by the Deep Convolutional Neural Network AlexNet (Krizhevsky et al., 2012), CNNs have been successfully applied to a larger variety of computer vision tasks, e.g. object detection (Girshick et al., 2014), video classification (Karpathy et al., 2014), object tracking (Wang and Yeung, 2013). It was a natural step to have also the astrophysics community to make use of CNNs. They provide the best ML algorithm in image processing.

Regular NNs usually receive as input a single vector and transform through some series of hidden layers to an output (a number). These NNs are still amazing, working with numbers, words and more generally input vector with a "small" size. But these don't scale well with images. Let's make some examples: the classical *MNIST* dataset is composed of 28x28 pixels greyscale images. So a single fully-connected neuron in a first hidden layer of a regular NN would have 28x28=784 weights. This is a small amount for todays standard, but if we want to work (and we do) with bigger images,

maybe 3-channel RGB, a single neuron will have a huge amount of weights. The simulations descripted in 4.1 are 512x512 pixels big. This would lead to 262144 weights for a single neuron. And obviously we want more than a single neuron, this will lead quickly to a huge number of parameter and overfitting.

3.4.1 Convolution Operation

The convolution is usually denoted as:

$$s(t) = (x * w)(t) \int x(a)w(t-a)da, \tag{55}$$

where $w(a)$ is a weighting function and $x(t)$ a function of some variable t. In CNN terminology the first argument x is referred to as input and the second one w as kernel or filter. The output s is referred sometimes as feature map. In realistic application, we work with discretized data on a computer so the t can take only integer values. Therefore, the convolution operation becomes:

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a). \tag{56}$$

In ML applications, the input is a multidimensional array of data (e.g. a set of images in RGB channels). Since each element of the input and kernel must be stored separetely, we can assume that these functions are zero everywhere but in the finite set of points for which we store the values. In practice, we can implement the infinite summation as a sum over finite number of array elements.

If the input is a 2D image (one channel) the convolution:

$$S(i,j) = (I * K)(i,j) = \sum_{m} \sum_{n} I(m,n)K(i-m, j-n). \tag{57}$$

Many Deep Learning libraries implement a related function called cross-correlation:

$$S(i,j) = (K * I)(i,j) = \sum_{m} \sum_{n} I(i+m, j+n)K(m,n), \tag{58}$$

calling it wrongly convolution.

The convolution operation applied to NNs uses three important ideas that improve tremendously their performance: sparse interactions, parameter sharing and equivariant representations (I. Goodfellow et al., 2016). Furthermore, from a computational point-of-view convolution reduces the size of the inputs to a workable size.

In the NNs described in section 3.2 every output unit interacts with every input unit. CNNs, instead, have sparse interactions (also referred to sparse weights). This is accomplished by making the kernel smaller than the input. For example, in processing an image, the input image can be 256×256 pixels, but we can detect features such as edges with kernels of size 3×3. It follows that to find meaningful features storing fewer parameters, which both reduces the memory requirements of the model and improves its statistical efficiency. Moreover, computing the output of a convolutional layer requires fewer operations: $\mathcal{O}(n \times k)$ instead of $\mathcal{O}(n \times n)$ runtime per exam-

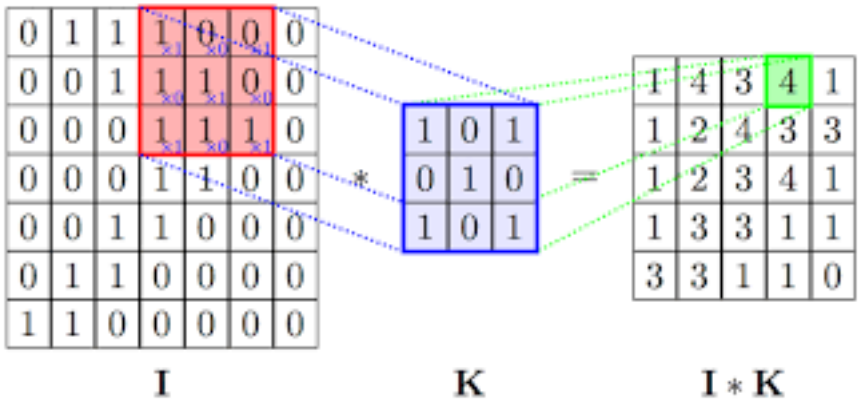

Figure 8: Effect of convolution with one kernel of size $k = 3$ on a 7×7 pixels image. Note the size of the output is 5×5 pixels big.

ple in the matrix multiplication. Usually k is of several orders of magnitude smaller than n, reducing the runtime drastically.

Furthermore, CNNs share parameter: when fixing the weights of the kernel in a convolutional layer, each element in the weight matrix is used exactly once when computing the ouptut of the layer. The parameter sharing used by the convoltuion means that rather than finding and learning the optimal set of weights for every location, the algorithm learns only one set. At the same time, this does not affect the runtime of forward propagation (still $\mathcal{O}(n \times k)$)

3.4.2 Convolution

In practice, when we talk of convolution in a CNN, we mean many applications of the convolution operation in parallel. Since, one kernel can extract only one kind of feature, we run multiple convolutions in the same layers to extract many features at many locations. Additionally, we use "deep" CNN: we stack convolutional layer after convolutional layer, meaning that the input in the inner layers is not just a grid of real values (pixels of an image). For example, working with images, we usually think of the input of the convolution as a 3D tensor (width, lenght, channel) $\mathbf{V}$ with $V_{i,j,k}$ giving the value of the input unit within channel i at row j and column k. The kernel tensor will be $\mathbf{K}$ with element $k_{i,j,k,l}$ giving the connection "strength" between a unit in channel i of the output and a unit in channel j of the input with an offset of k rows and l columns between the output unit and the input unit. The output of the convolution will be a tensor $\mathbf{Z}$:

$$Z_{i,j,k} = \sum_{l,m,n} V_{l,j+m-1,k+n-1} K_{i,l,m,n}.$$

The important result is that the result of a convolution of an image is a new image with shrinked size $(n-k+1) \times (n-k+1)$ where k is the size of the kernel and n the width of the input image. The depth, or number of "channels", is equal to the number of convolutional kernels we decided to use. If we want to reduce even more the computational cost, we may

want to skip the filter every s pixels of the input image. This downsampled convolution function will be:

$$Z_{i,j,k} = c(\mathbf{K}, \mathbf{V}, s)_{i,j,k} = \sum_{l,m,n} \left[V_{l,(j-1)\times s+m,(k-1)\times s+n} K_{i,l,m,n} \right]. \quad (59)$$

We refer to s as the stride. Effectively, the stride reduces the size of the output image to:

$$\frac{n-k}{s} + 1 \times \frac{n-k}{s} + 1.$$

Not only, any CNN implementation has the ability to implicitly pad the input $\mathbf{V}$ to make it wider by adding p layers of constant pixels on the border during the convolution. This is done because during the convolution the outer pixels are considered fewer times than the central ones. With padding p the size of the output image will be:

$$\frac{n+2p-k}{s} + 1 \times \frac{n+2p-k}{s} + 1. \quad (60)$$

Padding let us control the kernel width and the size of the output independently, without being forced to choose between shrinking the spatial representation or using small kernels. In software implementation (Keras in our work) we give as input also all the images in the batch, so that the input is effectively a 4D tensor. For simplicity, we omitted the batch "axis" in the previous description.

3.4.3 Pooling layers

It's common practice to insert a pooling layer in between successive convolutional layers in a CNN. Typically a layer of a CNN consist in the several convolutions in parallel producing a set of linear activations, then each linear activation is run through a nonlinear function (typically a ReLU) then we use a pooling function to modify the output even further. For example the max pooling (Zhou and Chellappa, 1988) operation takes the maximum value over a $m \times m$ neighbourhood. Like the convolutional layers, we can fix a value s for stride and f for the filter size. Typical values are $f = 2$ and $s = 2$, so that we are left out with an output image with exactly half the size of the input image (fig. 9). This not only reduces the number of parameters but also progressively reduces the spatial size of the representation, and hence reduce overfitting. Moreover, pooling helps to make the representation approximately invariant to small translations of the input. This is very important if we care more about whether some feature is present than exactly where it is.

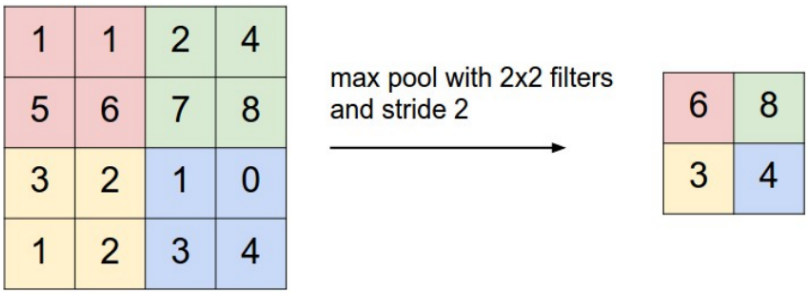

Figure 9: Max pooling layer with stride 2 and filter size 2.

4 METHODS

In this chapter we explain the gas simulations and the convolutional neural network that make use of them for training. The simulations were carried out with the hydrodynamic code RAMSES, instead the training and implementation of the CNN with the neural network library Keras.
In the first section we explain briefly the characteristics of the gas simulation. In the second section we describe the preprocessing of the dataset, the hyperparameter optimization and the training of the CNN.
The predictions of the CNN are discussed in the next chapter.

4.1 SIMULATIONS

In the framework of my thesis, we decided not to simulate realistic MCs. The main ingredient of the simulation is gas with a turbolent velocity field without self-gravity. The simplicity of our simulations is needed for running a lot of different simulation in a considerable amount of time. Furthermore, as a proof-of-concept, the simplicity of the simulations is good to prove that we can infere physics with DL. In future, self-gravity and other physical ingredients can be added.
To simulate the turbulent gas, we used RAMSES[1] (Teyssier, 2002), an Adaptive Mesh Refinement code for self-gravitating magnetized fluid flows.

4.1.1 RAMSES

RAMSES was developed first to study large scale structure and galaxy formation. It is written in Fortran 90 with extensive use of the MPI library. This code is a grid-based hydro solver with adaptive mesh refinement. It is now used for general purpose simulations in self-gravitating fluid dynamics.

4.1.2 Initial conditions and physical setup

Initially, the simulation box (side of size 10 pc) is completely filled with gas with uniform density (6.77×10^{-22} g/cm^3) corresponding to a total mass of $10^4 M_\odot$ in the box. The gas is at the same temperature $T = 10K$. We then injected a divergence free ($\nabla \cdot \mathbf{v} = 0$) turbulent, mildly supersonic (Mach number $M = \sqrt{2}$) velocity field with different values of n from 3.0 to 4.5, where n is the dependance of the power spectrum $\Psi(k)$ on the scale k as defined in equation 21.

Furthermore, $E_k/E_{th} = (v/c_s)^2 = 2$, i.e. the gas is supersonic. Then, we let the system evolve for 0.5 Myr with an isothermal EoS with $\gamma = 1$ (see eq 7), solving the Euler's equation described in section 2.2 with a Lax-Friedrichs Riemann Solver and no self-gravity. For each simulation, we take snapshots of the density from three different perpendicular directions. An example of these snapshot for $n = 3.7650$ can be seen in fig. 10 and 11.

1 https://www.ics.uzh.ch/~teyssier/ramses/RAMSES.html

Table 1: Simulation parameters

Parameters	Values
Mass ($M_\odot$)	10^4
Box size (pc)	10
Density (g/cm^3)	6.77×10^{-22}
Temperature (K)	10
Sound speed (km/s)	0.1
Equation of State	Isothermal

The black in the images corresponds to a columns density of $5 \times 10^{-4} g/cm^2$ and the white to a $3 \times 10^{-2} g/cm^2$. Pixels intensity values are normalized to 1 in order to work with small values to speed up the training. The density to pixel intensity transformation is linear, meaning that a pixel of intensity I corresponds to a column density $5 \times 10^{-4} + I \cdot (3 \times 10^{-2} - 5 \times 10^{-4}) g/cm^2$.

4.1.3 Refinement strategy

RAMSES is an Adaptive Mesh Refinement (AMR) code. Mesh refinement enables us to have more resolution on the regions that are more physically meaningful while leaving the lowest resolution in the less important regions. Therefore, we can have a good resolution saving computational resources and time. We chose the refinement criteria based on gradients of the flow variables q (pressure, density, Mach number ...). Especially, for each cell i, the gradient of the $i-th$ variable q is computed using the 2 x dim neighboring cells. If this gradient, times the local mesh spacing, exceeds a fraction of the central cell variable:

$$\nabla q_i \geqslant (\nabla q)^l_{max} = C_q \frac{q_i}{\Delta x^l} \tag{61}$$

then the cell is refined (Teyssier, 2002). C_q is a free parameter. As refinement criterion, we choose the velocity gradient with $C_q = 1.35$. We adopted this fixed value for all simulations, after testing that this choice allowed to resolve effectively the turbulence for all values of n with a high enough number of cells. We set the minimum and maximum refinement levels as 5 and 8 respectively. This means that the spacial resolution of $1/(2^5) = 1/32$ of the box side (0.3 pc) for the least resolved cells and a resolution of $1/(2^8) = 1/256$ of the box side (0.04 pc) for the most resolved ones.

4.2 NEURAL NETWORK TRAINING

In section 3.4, we provided reasons why CNNs are one of the best ML algorithm for image processing. Now, we want to build a CNN to predict the power spectrum index n.

4.2.1 Preprocessing Data

The performance of a NN improves increasing the number m of examples. Furthermore, a CNN has fewer parameters than a MLP but still of the order of 10^{6-7}. If we don't show enough examples, the model will inevitably overfit the dataset. Since, running hydrodynamic simulations can be very

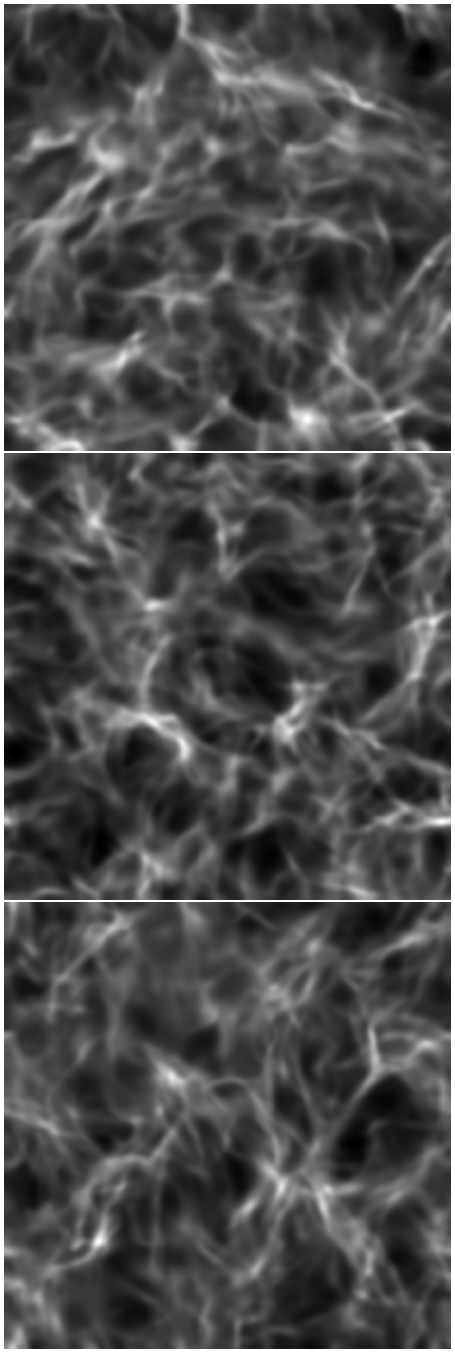

Figure 10: Top, central and bottom panels show density snapshot for $n = 3.7650$ seen from 3 different direction (x, y, z axes respectively).

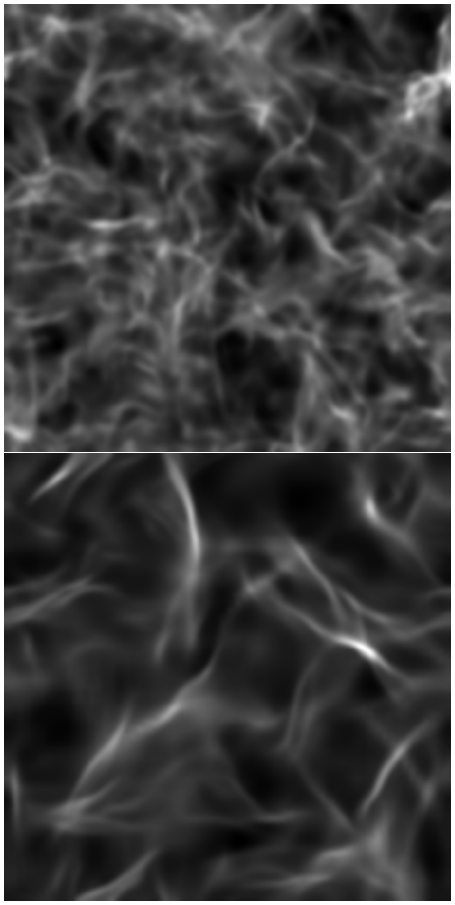

Figure 11: Examples of a projected density map from a RAMSES simulation for n equal to 3.6399 (*top panel*) and 4.4553 (*bottom panel*).

time and computational expensive, we can augment m synthetically with some minor transformations. Before the augmentation, we exploited the fact that the gas simulations have periodic boundary conditions: the properties on one side of the box are the same on the opposite side for each side of the box. The gas on the side is interacting with the one on the opposite side. This means that we can craft an image made with 4 copies of the same simulation that has a bigger dimension (see 12). Cropping a portion of this image with same size of the original simulation (1000×1000 pixels) yields a "new" simulation, i.e. the gas structures have a different spatial orientation from the original simulation.

4.2.2 Augmentor

To perform the data augmentation we used the python library Augmentor[2]. Augmentor samples every time a different cropped section of the collage large 1000X1000 pixels. Then we set a probability of 0.5 of horizontal reflections, 0.5 of vertical reflections, 0.5 of 90-degrees rotations on this crop section. With this process we artificially enlarge our dataset from 240 (corresponding to 80 different indexes) images to 20000. The number combinations of cropping plus reflections and rotations, guarantes that we do not have two identical examples in our training dataset.

4.2.3 Choice of the architecture

We chose to work with a CNN for all the reasons stated in section 3.4.

To implent the CNN we used Keras, a NN library written in Python. It runs on top of TensorFlow, an open-source software library developed by Google. Keras allows use of distributed training of deep learning models on clusters of GPUs.

Our CNN is composed by: two convolutional layers with 32 filters each then a MaxPool layer followed by other two convolutional layers then another MaxPool then a fully connected layer then the output. The convolutional filters are 5×5 pixels big with "same" padding (that results in padding the input such that the output has the same length as the original input). The exact structure is as follows:

- Two 5×5 convolutional layers with 32 filters and same padding resulting in a $128 \times 128 \times 32$ output.
- A max pooling layer with 2×2 filter size and stride 2, resulting in an $64 \times 64 \times 32$) output.
- A dropout layer with 33% ratio of dropped units.
- Two 5×5 convolutional layers with 64 filters and same padding resulting in a $64 \times 64 \times 64$ output.
- A max pooling layer with 2×2 filter size and stride 2, resulting in an $32 \times 32 \times 64$) output.
- A dropout layer with 33% ratio of dropped units.
- A fully connected layer with 1024 neurons and ReLU activation.
- A single neuron with a linear activation with RSE loss as cost function.

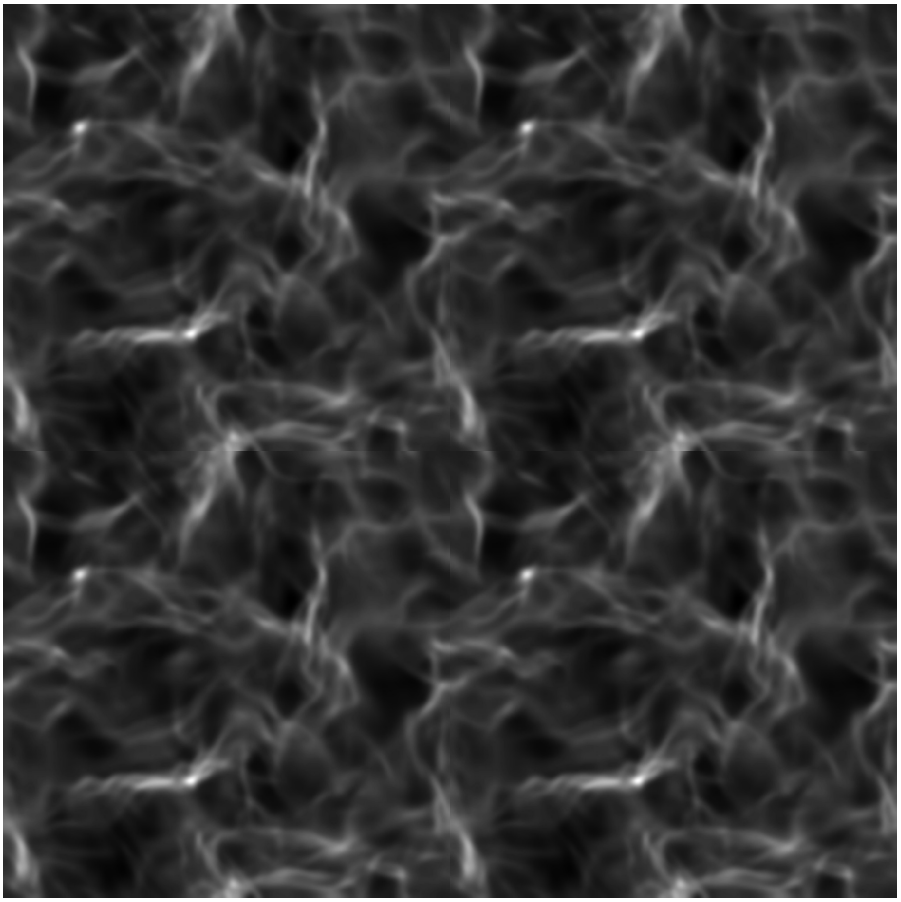

Figure 12: Example of a collage of the same simulation ($n = 4.0189$). Note that the border of the collage can be see only with a careful analysis.

Table 2: CNN architecture: the filter size of the convolutional and max pooling layers is displayed also with the output of the layer. The final column correspond to the number of parameters of that specific layer.

Layer name	Filter size	Output size	Parameters
Convolution	5×5	$128 \times 128 \times 32$	832
Convolution	5×5	$128 \times 128 \times 32$	25632
MaxPool	2×2	$64 \times 64 \times 32$	
Convolution	5×5	$64 \times 64 \times 64$	51264
Convolution	5×5	$64 \times 64 \times 64$	102464
MaxPool	2×2	$32 \times 32 \times 64$	
Flatten		$1 \times 1 \times 65536$	
Dense 1	1024	$1 \times 1 \times 1024$	67109888
Dense 2	1	1	1025

Table 3: Impact of batch size and Optimizer on training and validation final loss measured by MSE times 10^{-2} (respectively left and right value in the brackets)

Optimizers	Batch size			
	32	64	128	256
AdaDelta	[0.87 2.50]	[1.40 0.70]	[2.79 0.97]	[6.03 6.07]
AdaGrad	[2.23 18.6]	[2.63 3.54]	[9.20 70.7]	[5.16 270.8]
RMSProp	[1.13 2.24]	[1.57 0.79]	[3.28 3.61]	[5.27 3.72]
Adam	[0.48 0.58]	[0.54 0.33]	[0.72 0.60]	[1.89 1.66]

We chose ReLU units, because CNN with ReLUs train several times faster than their equivalents with `tanh` or sigmoid units (Krizhevsky et al., 2012).

The final number of parameters to be trained is 67,291,105. Note that almost all of parameters are in the first fully connected layer. Optimization on the width of this layer can lead to a faster training.

4.2.4 Optimizers and batch size impact

We trained our NN on the augmented dataset (see 4.2.2) with a NVIDIA Titan V GPU. We used a validation split on the dataset of 0.2 meaning that every epoch 1/5 of the training set is left for validation (4000 images). The validation set is picked at random every epoch, in this way we are not introducing any bias on the validation indexes. For the training and validation set split we used a built-in option of Keras for splitting the training set. Before seeing its performance on the test set we compare four optimizers: AdaDelta, AdaGrad, RMSprop and Adam (see section 3.3.4 for a complete discussion). For each optimizers, we train the CNN for a 100 epochs with different batch size (32, 64, 128, 256). The loss evolution is displayed in fig. 13. The differences are not extreme but we can notice how RMSProp and AdaGrad underperform with respect to AdaDelta and Adam for almost all batch size. As expected (I. Goodfellow et al., 2016), almost all optimizers have a better performance with small batch size. This high accuracy is obtained at the cost of computational speed since we have to compute more gradients, (note that larger batch size implies fewer gradients but is more "expensive" in terms of memory space). The final training and validation losses are shown in table 3. Note also that the Adam optimizer is the most robust to the batch size change, yielding comparable results with different sizes.

Due to this preliminary test, we decided to utilize the Adam optimizer with a batch size of 64 since it is a good trade-off between small loss and reasonable computational speed.

4.2.5 Training

For the final training of our CNN, we trained it for 1000 epochs with a 0.2 validation split and using the mean squared error as loss function. The evolution of the training and validation loss is shown in figure 14. The final training and validation loss is 2.15 $\times 10^{-3}$ and 3.60$\times 10^{-2}$.

2 https://github.com/mdbloice/Augmentor

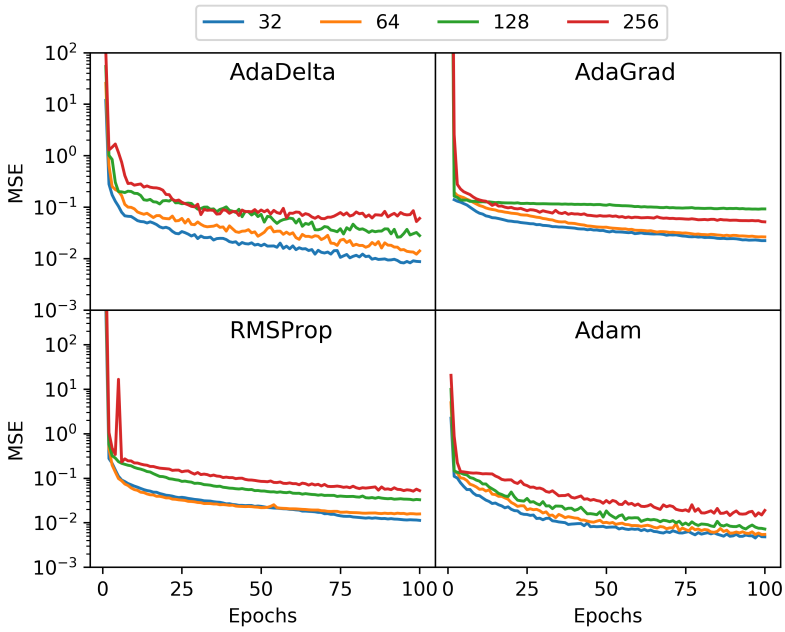

Figure 13: Comparison of the training loss evolution with different optimizers for batch size of 32, 64, 128, 256 images (blue, orange, green and red line respectively).

At this point of our work, the CNN never saw the test set composed of 60 simulations corresponding to 20 different turbulence indexes. The predictions on the test set are shown in the next chapter.

4.3 A COMPARISON BASELINE

As a baseline for comparison for our results we used a linear model trained on standard computer vision features known as Histograms of Oriented Gradients [HOG; Freeman and Roth, 1994]. HOG is a feature descriptor widely used in image processing for object detection (e.g. pedestrian detection for self driving cars, face detection, etc...). This method divides an image into rectangular cells and counts occurrences of intensity gradient orientation along specified directions. The number of directions and cells used determines the number of features produced, each feature being a numeric value used as a descriptor of the image. We used the R language implementation of HOG in the *OpenImageR* library with 5×5 cells and 6 orientations, resulting in 150 features per image. On these features corresponding to the training set images we trained a linear model to predict the turbulence index. This model was then used to make predictions on the features corresponding to the test set image. The predictions are shown in the next chapter.

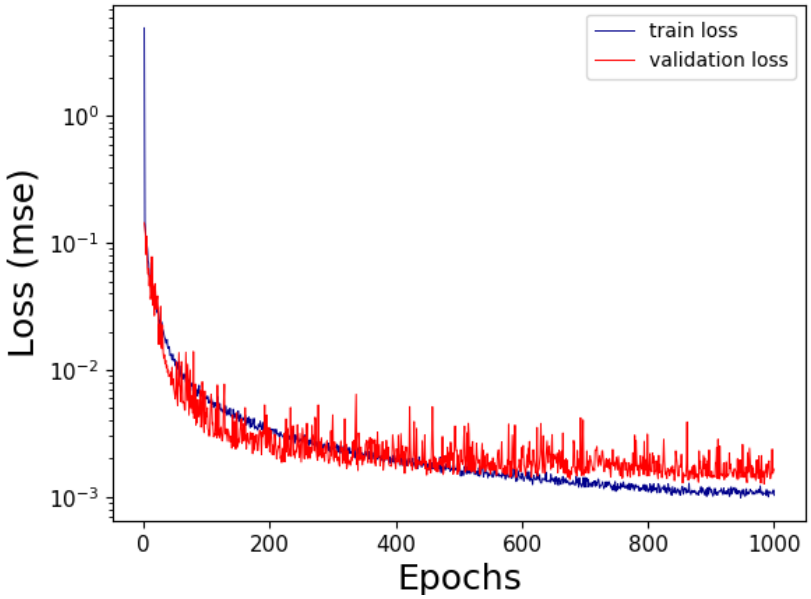

Figure 14: Evolution of the training loss (solid blue line) and the validation loss (solid red line) for 1000 epochs. Note the more noisy nature of the validation loss. A big gap from the two lines means that the network is not able to generalize its prediction on the validation data.

5 RESULTS AND DISCUSSION

The central challenge of our work and in general in any ML problem is that our algorithm must perform well on new, unseen data. This ability to perform well on previously unobserved inputs is called generalization. In the first section, we show the predictions of the CNN described in the previous chapter on unseen simulations and then compare its predictions with the baseline prediction made with an HOG. We show that the CNN outperforms the HOG.
In the second section, we discuss the limits of this method.

5.1 PREDICTIONS ON UNSEEN SIMULATIONS

We fed the CNN, described in section 4.2, 60 simulations made with the same ingredients of the training simulations (see section 4.1) then we compared the output with the prediction of the HOG on the same simulations. The 60 images correspond to 20 simulations with different turbulence index seen from three different perpendicular directions. On this set, we did not perform any augmentation process (crop, flip, rotation). The predictions of the CNN are shown in fig.15. The predictions of the HOG are shown in fig.16. We chose as a performance measure the MSE: we obtained a MSE of 0.029 for the CNN and 0.106 for the HOG. The performance of the CNN is about one order of magnitude better than the HOG. It is interesting that we did not perform a complete hyperparameter optimization, but we only chose to experiment with the optimizers and the batch size. We can imagine that a proper hyperparameter optimization can lead to even better results.

We see that points spread more towards bigger indexes. A reason could be that simulations with bigger indexes have more big and sparse structures (see bottom panel of figure 11). Convolutional filters of size 5×5 maybe can not extract the best features to acknowledge these structures. In any case, large indexes (n>4) are not meaningful in MCs. The "hot" range is equal and below the Burgers index, $n = 4$, where we can find the Kolmogorov value, $n = 3.66667$.

5.2 DISCUSSION AND LIMITS

5.2.1 Discussion

The DL approach used in this thesis performs better than a linear model trained on standard computer vision features. The features extracted with the DL approach generalized better on unseen simulations. This confirms the CNNs as powerful tool for image processing and feature extraction. But more importantly, we can retrieve a physical parameter (the turbulence index) only from raw data without any data reduction. This opens a new way to the very process of scientific inference. Usually, a theorist, in order to make prediction from some physical theory, produces some quantitative

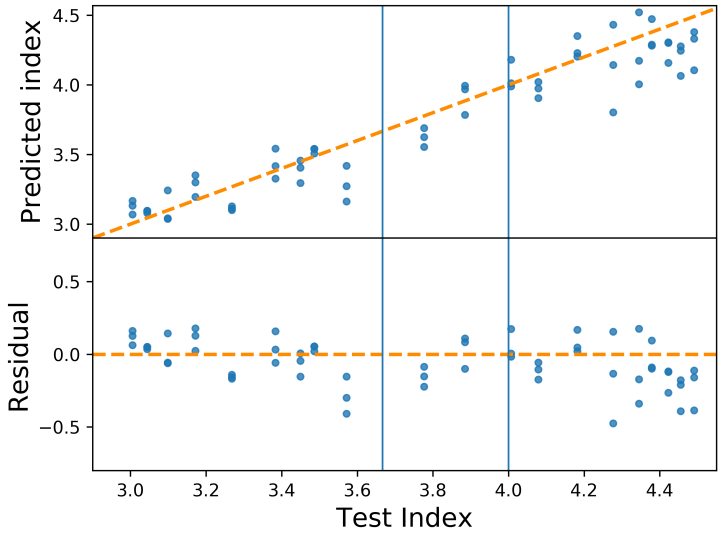

Figure 15: Prediction of the CNN on the 60 test images. Top panel: power spectrum indexes predicted from the CNN plotted versus the actual indexes labeled as test indexes. Bottom panel: the residual are plotted versus the test indexes. The vertical line at $n = 3.6667$ and $n = 4.0$ corresponds to Kolmogorov and Burger index respectively.

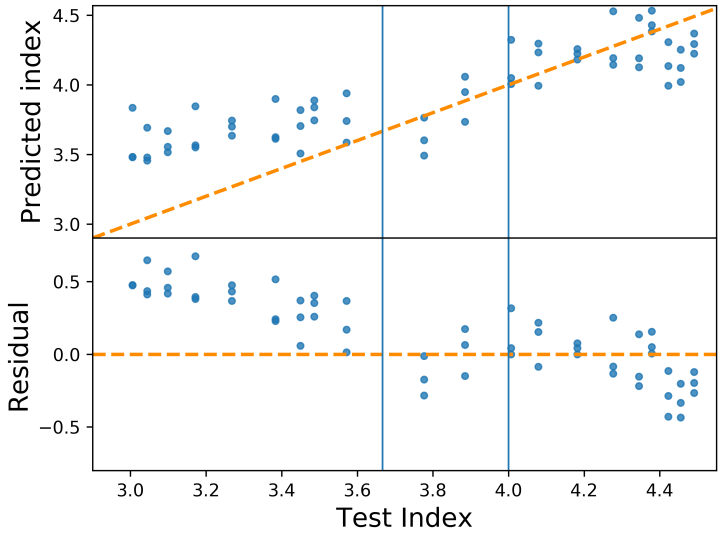

Figure 16: Prediction of the HOG on the 60 test images. Top panel: power spectrum indexes predicted from the CNN plotted versus the actual indexes labeled as test indexes. Bottom panel: the residual are plotted versus the test indexes. The vertical line at $n = 3.6667$ and $n = 4.0$ corresponds to Kolmogorov and Burger index respectively.

model via calculation but more often with a simulation. To test the goodness of the model, we will be able to compare its predictions to the results of experiments and observations. We showed here that the process of physical inference on data, for the moment being only simulations output, can be done with a ML algorithm, specifically a CNN trained on the output of gas simulations. Nonetheless, this approach, for now, has some limitations.

5.2.2 Limits

UNREALISTIC SIMULATIONS: It is true that our gas simulations lack of many physical ingredient, such as self-gravity, in order to be a realistic model of MCs. We must stress out that this work is a proof of concept. Our network can not produce reasonable predictions in the "real world". We do not claim that the turbulence index predicted with this CNN on a real MC is a reliable one. But in the next future, we can train a CNN with more realistic simulations made with more physical ingredients and retrieve, in theory, more parameters only from raw data. In any case, at the state-of-the-art it's not legitimate to train on simulations and use real observations as test set. In the next future, we can train an autoencoder to quantify this "distance" from realistic simulations or observations. An autoencoder (Ballard, 1987) is a ML algorithm whose aim is to learn a representation (encoding) for a dataset (simulations) by reducing its dimension then decoding the learnt representation in order to produce an output as close as possible to the input. If we train an autoencoder on our simulations and then feed a more realistic simulations, the reconstruction error will be a measure on how far away the training dataset is from for being realistic.

USE OF MACHINE LEARNING: NNs and in general models with more parameters than examples have the problem that tend to overfit, i.e.m to learn too well the data on which they are trained. This can lead to a poor generalization on unseen inputs. Monitoring the validation error and/or adding a regularization term (e.g.m dropout) usually takes care of this aspect. In any case, we introduce a bias due to how the simulations are carried on (and in general to the dataset used). But that is true for every method that relies on simulations to make predictions. As a matter of fact, also predictions of experiments are biased on how the experiment is made. As with any scientific observation, there is possibility for confusion between real physical effects and deficiencies and biases in the training data and the network architecture. Not only, prediction made from raw data via DL skip the data reduction and cleaning process that can introduce a human bias. But predictions on the output of simulations are of course different from predictions on observations. Thus, the importance of the autoencoder.
More problems related in general of any ML algorithm and therefore also with our CNN are the reproducibility of the results: can another scientist find our results? Change in the software, in the GPU drivers and dataset can lead to a different model with the same architecture. Also, weights are initialized randomly at the start of the training and the dropped hidden units are chosen at random. This can lead to different results. A practical way to address this problem is to run the network many times and use statistics to evaluate the performance of the model. This can be difficult, due to the very long training times of the bigger models. A simple alternative is to set a seed for the generation of (pseudo)random numbers in the model. Usually, current time in milliseconds is used as seed, that ensure that different

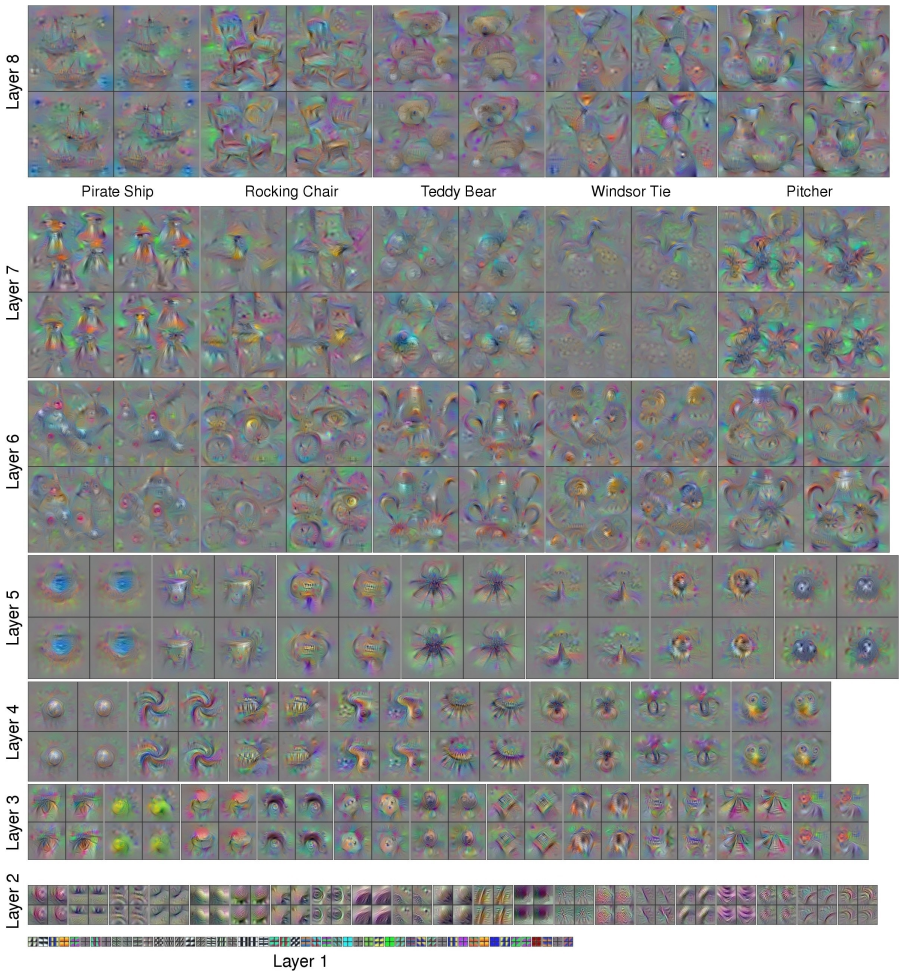

Figure 17: Images that show example features of all eight layers on a network similar to AlexNet (Krizhevsky et al., 2012). The complexity of the learned features increases in higher-layers as they combine simpler features from lower layers. Furthermore, a variation of patterns can be seen going deeper, revealing that increasingly invariant, abstract representations are learned. All credits to [Yosinski et al., 2015]

sequences of numbers are generated everytime the training start. Fixing a seed takes away this unpredictability, thus making possible to replicate the exact same training and obtaining the same result.

BLACK BOX CRITIC: DNNs extract the features they need in order to minimize the loss function. This of course can lead to amazing performance but at the same time, we do not really know what computations DNNs perform in the hidden layers. Furthermore, DNN is a non-identifiable model, i.e., given a dataset and NN architecture, there can be two NNs with different weights yielding the same result. This makes hard to understand the function we are trying to approximate with our model. Nonetheless, progresses on visualize and understanding the learning process in the inner layers are carried on. A way can be to visualizing and understand the activations and layer weights in the inner layers (see figure 17).

6 CONCLUSIONS AND FUTURE PROSPECTS

We have shown a proof-of-concept of reducing a scientific question (what is the turbulence index of molecular clouds) to a supervised regression problem, which we tackled with deep learning. What singles out our work with respect to other applications of machine learning to astronomy is the fact that we trained our neural network models on the output of simulations. This is a necessary step to bring the full power of machine learning techniques to fruition in the astronomical field, where they are mostly confined to the automation of tedious tasks on large datasets. Our works instead sets the stage for automating the very process of scientific inference, by training a neural network to recover physical parameters of interest directly from (simulated) raw data.

Future improvements and developments could be: hyperparameter optimization of the network in order to achieve better performance. We carried out an optimization only on the optimizer and the batch size. A proper hyperparameter optimization or even a change of architecture can lead us to better performance. DL is in constant evolution: new architectures and strategies in this field are found out almost everyday.

Secondly, training of a classifier on the two interesting turbulence indexes, Kolmogorov and Burgers. More simulations can be carried out for these two indexes, and a classifier can be trained to recover what kind of turbulence there is simply from the simulations. This step is straightforward from our results.

Finally and the most important, the training of an autoencoder on simulations in order to quantify the error we are making using this neural network on realistic simulation or observations: as discussed in the previous chapter our prediction are not reliable prediction of turbulence in molecular clouds, because they are trained on simple simulations that lack self-gravity in particular. How much our simulation differs from realistic simulation can be investigated with an autoencoder trained on the set of simulation. If the reconstruction error is small on a realistic simualation, the prediction made with the neural network will not differ much from a neural network trained with realistic simulations. This also can save a lot of computational time.

As a final remark, the approach shown here is a data-driven way to investigate a physical question (the turbulence of the gas). The advantage of this approach is exactly its data-driven nature: the predictions made with the neural network make no assumptions on the physics of the phenomenon. The physical assumptions are in the making of the simulations but not in the proper data inference.

BIBLIOGRAPHY

Armstrong, DJ, J Kirk, KWF Lam, J McCormac, HP Osborn, J Spake, S Walker, DJA Brown, MH Kristiansen, D Pollacco, et al.

2015 "K2 variable catalogue–II. Machine learning classification of variable stars and eclipsing binaries in K2 fields 0–4," *Monthly Notices of the Royal Astronomical Society*, 456, 2, pp. 2260-2272.

Askar, Ammar, Abbas Askar, Mario Pasquato, and Mirek Giersz

2018 "Finding Black Holes with Black Boxes–Using Machine Learning to Identify Globular Clusters with Black Hole Subsystems," *arXiv preprint arXiv:1811.06473*.

Ballard, Dana H

1987 "Modular Learning in Neural Networks.," in *AAAI*, pp. 279-284.

Burgers, Johannes Martinus

1948 "A mathematical model illustrating the theory of turbulence," in *Advances in applied mechanics*, Elsevier, vol. 1, pp. 171-199.

Cabrera-Vives, Guillermo, Ignacio Reyes, Francisco Förster, Pablo A Estévez, and Juan-Carlos Maureira

2016 "Supernovae detection by using convolutional neural networks," in *Neural Networks (IJCNN), 2016 International Joint Conference on*, IEEE, pp. 251-258.

2017 "Deep-HiTS: Rotation invariant convolutional neural network for transient detection," *The Astrophysical Journal*, 836, 1, p. 97.

Champagne, Frank H

1978 "The fine-scale structure of the turbulent velocity field," *Journal of Fluid Mechanics*, 86, 1, pp. 67-108.

Davies, GR, V Silva Aguirre, TR Bedding, R Handberg, MN Lund, WJ Chaplin, D Huber, TR White, O Benomar, S Hekker, et al.

2015 "Oscillation frequencies for 35 Kepler solar-type planet-hosting stars using Bayesian techniques and machine learning," *Monthly Notices of the Royal Astronomical Society*, 456, 2, pp. 2183-2195.

Duchi, John, Elad Hazan, and Yoram Singer

2011 "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, 12, Jul, pp. 2121-2159.

Federrath, C, J Roman-Duval, RS Klessen, W Schmidt, and M-M Mac Low

2010 "Comparing the statistics of interstellar turbulence in simulations and observations-Solenoidal versus compressive turbulence forcing," *Astronomy & Astrophysics*, 512, A81.

Federrath, Christoph

2013 "On the universality of supersonic turbulence," *Monthly Notices of the Royal Astronomical Society*, 436, 2, pp. 1245-1257.

Freeman, William T. and Michal Roth

1994 *Orientation Histograms for Hand Gesture Recognition*, tech. rep. TR94-03, MERL - Mitsubishi Electric Research Laboratories, Cambridge, MA 02139, http://www.merl.com/publications/TR94-03/.

Fukushima, Kunihiko and Sei Miyake

1982 "Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition," in *Competition and cooperation in neural nets*, Springer, pp. 267-285.

George, Daniel, Hongyu Shen, and EA Huerta

2017 "Deep Transfer Learning: A new deep learning glitch classification method for advanced LIGO," *arXiv preprint arXiv:1706.07446*.

Girshick, Ross, Jeff Donahue, Trevor Darrell, and Jitendra Malik

2014 "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 580-587.

Glorot, Xavier and Yoshua Bengio

2010 "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pp. 249-256.

Glorot, Xavier, Antoine Bordes, and Yoshua Bengio

2011 "Deep sparse rectifier neural networks," in *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pp. 315-323.

Glover, SCO and M-M Mac Low

2011 "On the relationship between molecular hydrogen and carbon monoxide abundances in molecular clouds," *Monthly Notices of the Royal Astronomical Society*, 412, 1, pp. 337-350.

Goodfellow, Ian J, Yaroslav Bulatov, Julian Ibarz, Sacha Arnoud, and Vinay Shet

2013 "Multi-digit number recognition from street view imagery using deep convolutional neural networks," *arXiv preprint arXiv:1312.6082*.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville

2016 *Deep Learning*, http://www.deeplearningbook.org, MIT Press.

Habart, Emilie, Malcolm Walmsley, Laurent Verstraete, Stephanie Cazaux, Roberto Maiolino, Pierre Cox, Francois Boulanger, and Guillaume Pineau Des Forêts

2005 "Molecular hydrogen," in *ISO Science Legacy*, Springer, pp. 71-91.

Hinton, Geoffrey E, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov

2012 "Improving neural networks by preventing co-adaptation of feature detectors," *arXiv preprint arXiv:1207.0580*.

Imara, Nia, Frank Bigiel, and Leo Blitz

2011 "Angular momentum in giant molecular clouds. II. M33," *The Astrophysical Journal*, 732, 2, p. 79.

Jóhannesson, G, R Ruiz de Austri, AC Vincent, IV Moskalenko, E Orlando, TA Porter, AW Strong, R Trotta, F Feroz, P Graff, et al.
2016 "Bayesian analysis of cosmic ray propagation: Evidence against homogeneous diffusion," *The Astrophysical Journal*, 824, 1, p. 16.

Karpathy, Andrej, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar, and Li Fei-Fei
2014 "Large-scale video classification with convolutional neural networks," in *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pp. 1725-1732.

Kimura, Akisato, Ichiro Takahashi, Masaomi Tanaka, Naoki Yasuda, Naonori Ueda, and Naoki Yoshida
2017 "Single-epoch supernova classification with deep convolutional neural networks," *arXiv preprint arXiv:1711.11526*.

Kingma, Diederik P and Jimmy Ba
2014 "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*.

Kolmogorov, Andrey Nikolaevich
1941 "The local structure of turbulence in incompressible viscous fluid for very large Reynolds numbers," in *Dokl. Akad. Nauk SSSR*, 4, vol. 30, pp. 299-303.

Kritsuk, Alexei G, Michael L Norman, Paolo Padoan, and Rick Wagner
2007 "The statistics of supersonic isothermal turbulence," *The Astrophysical Journal*, 665, 1, p. 416.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E Hinton
2012 "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, pp. 1097-1105.

Krumholz, Mark R
2015 "Notes on Star Formation," *arXiv preprint arXiv:1511.03457*.

Larson, Richard B
1981 "Turbulence and star formation in molecular clouds," *Monthly Notices of the Royal Astronomical Society*, 194, 4, pp. 809-826.

Lathuilière, Stéphane, Pablo Mesejo, Xavier Alameda-Pineda, and Radu Horaud
2018 "A Comprehensive Analysis of Deep Regression," *arXiv preprint arXiv:1803.08450*.

LeCun, Yann, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel
1989 "Backpropagation applied to handwritten zip code recognition," *Neural computation*, 1, 4, pp. 541-551.

LeCun, Yann, Léon Bottou, Yoshua Bengio, and Patrick Haffner
1998 "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, 86, 11, pp. 2278-2324.

Maas, Andrew L, Awni Y Hannun, and Andrew Y Ng
2013 "Rectifier nonlinearities improve neural network acoustic models," in *Proc. icml*, 1, vol. 30, p. 3.

McCulloch, Warren S and Walter Pitts
1943 "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, 5, 4, pp. 115-133.

Minsky, Marvin and Seymour Papert
1969 *Perceptron Expanded Edition*.

Mitchell, Tom M et al.
1997 "Machine learning. 1997," *Burr Ridge, IL: McGraw Hill*, 45, 37, pp. 870-877.

Murray, N.
2011 "Star Formation Efficiencies and Lifetimes of Giant Molecular Clouds in the Milky Way," 729, 133 (Mar. 2011), p. 133, DOI: 10.1088/0004-637X/729/2/133, arXiv: 1007.3270.

Qian, Ning
1999 "On the momentum term in gradient descent learning algorithms," *Neural networks*, 12, 1, pp. 145-151.

R Core Team
n.d. *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, https://www.R-project.org.

Rosenblatt, Frank
1958 "The perceptron: a probabilistic model for information storage and organization in the brain." *Psychological review*, 65, 6, p. 386.

Ruder, Sebastian
2016 "An overview of gradient descent optimization algorithms," *arXiv preprint arXiv:1609.04747*.

Rumelhart, David E, Geoffrey E Hinton, and Ronald J Williams
1986 "Learning representations by back-propagating errors," *nature*, 323, 6088, p. 533.

Schawinski, Kevin, M Dennis Turp, and Ce Zhang
2018 "Exploring galaxy evolution with generative models," *arXiv preprint arXiv:1812.01114*.

Schawinski, Kevin, Ce Zhang, Hantian Zhang, Lucas Fowler, and Gokula Krishnan Santhanam
2017 "Generative adversarial networks recover features in astrophysical images of galaxies beyond the deconvolution limit," *Monthly Notices of the Royal Astronomical Society: Letters*, 467, 1, pp. L110-L114.

Schinnerer, Eva, Sharon E Meidt, Jérôme Pety, Annie Hughes, Dario Colombo, Santiago Garcıa-Burillo, Karl F Schuster, Gaëlle Dumas, Clare L Dobbs, Adam K Leroy, et al.
2013 "The PdBI arcsecond whirlpool survey (PAWS). I. A cloud-scale/multi-wavelength view of the interstellar medium in a grand-design spiral galaxy," *The Astrophysical Journal*, 779, 1, p. 42.

Sedaghat, Nima and Ashish Mahabal
2017 "Effective Image Differencing with ConvNets for Real-time Transient Hunting," *arXiv preprint arXiv:1710.01422*.

Shallue, Christopher J and Andrew Vanderburg
2018 "Identifying Exoplanets with Deep Learning: A Five-planet Resonant Chain around Kepler-80 and an Eighth Planet around Kepler-90," *The Astronomical Journal*, 155, 2, p. 94.

Szegedy, Christian, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich

2015 "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1-9.

Teyssier, R.

2002 "Cosmological hydrodynamics with adaptive mesh refinement. A new high resolution code called RAMSES," 385 (Apr. 2002), pp. 337-364, DOI: 10.1051/0004-6361:20011817, eprint: astro-ph/0111367.

Tieleman, Tijmen and Geoffrey Hinton

2012 "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude," *COURSERA: Neural networks for machine learning*, 4, 2, pp. 26-31.

Wang, Naiyan and Dit-Yan Yeung

2013 "Learning a deep compact image representation for visual tracking," in *Advances in neural information processing systems*, pp. 809-817.

Wilson, BA, TM Dame, MRW Masheder, and P Thaddeus

2005 "A uniform CO survey of the molecular clouds in Orion and Monoceros," *Astronomy & Astrophysics*, 430, 2, pp. 523-539.

Yosinski, Jason, Jeff Clune, Anh Nguyen, Thomas Fuchs, and Hod Lipson

2015 "Understanding neural networks through deep visualization," *arXiv preprint arXiv:1506.06579*.

Zeiler, Matthew D

2012 "ADADELTA: an adaptive learning rate method," *arXiv preprint arXiv:1212.5701*.

Zhou, Yi-Tong and Rama Chellappa

1988 "Computation of optical flow using a neural network," in *IEEE International Conference on Neural Networks*, vol. 1998, pp. 71-78.

ACKNOWLEDGEMENTS

Vorrei ringraziare Michela Mapelli, per essere stata disponibile e sorridente nonostante i mille impegni.
Grazie con tutto il cuore a Mario e Alessandro che sono stati sempre presenti e gentilissimi. Mi avete accompagnato serenamente in questi mesi.
Grazie a mia sorella, a mio padre e Simba, ma sopratutto ringrazio i miei amici più cari che sono stati vicini nell' anno più buio della mia vita. Non ce l'avrei mai fatta senza di voi. Ringrazio chi ora naviga, con me, nella rinascita, in questo veliero. In particolare una certa bussola meravigliosa che mi sta conducendo in un futuro con continui dolori alle guance. C'era una presa da corrente vicino al letto comunque.

Ringrazio il pecorino romano per le best ricette, i narvali zenzerosi, il master of caseifici che ha inventato il gorgonzola e lo staff di Iginio Massari per le rapide risposte. Ma soprattutto l'aggettivo lapalissiano.

Saluto ora queste stanze, questo dipartimento che è stato non solo una casa ma un mondo pieno di avventure. So che Alberto, essenzialmente, sarà, là in alto, sempre a guardarmi (aglio a parte).

Hold your memory for a moment with a blind hand
Write some stories for tomorrow
From the bottle of amnesia
Find instructions, to salvation, to oblivion, supreme.