SAPIENZA
UNIVERSITÀ DI ROMA

Dynamic Graph Neural Networks Applications to Glassy Systems

Facoltà di Scienze Matematiche Fisiche e Naturali

Corso di Laurea Magistrale in Fisica

Candidate

Massimo Ciacchi

ID number 1354565

Thesis Advisor

Prof. Stefano Giagu

Academic Year 2020/2021

Thesis defended on 19 Gennaio 2022
in front of a Board of Examiners composed by:

Prof. Antonello Davide Polosa (chairman)
Prof. Stefano Lupi
Prof. Stefano Giagu
Prof. Francesco Pannarale
Prof. Alessandro Paiella
Prof. Maria Chiara Angelini
Prof. Barbara Ruzicka

Dynamic Graph Neural Networks Applications to Glassy Systems
Master's thesis. Sapienza – University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email: ciacchi.1354565@studenti.uniroma1.it nihil39@gmail.com

A Roberta F.

Abstract

Some aspects of the nature of the glassy state are still puzzling after years of both theoretical and experimental studies, thus this field could be suitable to an application of an automatic learning system. Inspired by the success of Convolutional Neural Network on all kinds of image related tasks, in this work we modify a model of a Dynamic Graph Convolutional Neural Network (DGCNN) and apply it to a simulated glass forming system. This network is designed to represent particles as graph nodes and it is able to update itself at each iteration according to particles' features. The dataset is a standard 4096 particles 80:20 Kob-Andersen Lennard-Jones mixture simulated with the software LAMMPS. We achieve good results when the network is trained on a sample of four configurations with different pressures and temperatures starting values, while some predictions are not consistent and require further investigation.

Acknowledgments

I would like to thank Simone Scanzoni for his precious advice on Bash Scripting and for some interesting conversations on GNU/Linux in general.

Contents

Chapter 1

Glassy systems and some places to find them

In this chapter we provide a short bird's eye view of glass transition, focusing on the peculiar behavior of a few crucial characteristic parameters. We highlight some of the differences between the glassy transition and the ordinary phase transitions giving some glimpses on theoretical aspects and results from the Mode Coupling Theory.

1.1 Glass is different

1.1.1 A few words on phase transitions

There is not a unique way to classify state transitions, but maybe the traditional quantitative thermodynamical one is to imagine a substance stable at two different temperatures' levels. At the critical temperature the free energy F[1] of both phases has to be equal. It is not the case for its derivatives, for example in the everyday examples of transition from liquid to solid and liquid to gas we have that $S = -\frac{\partial F}{\partial T}$ is discontinuous. This means that the entropy "jumps" in those passages. The discontinuity could also happen in second derivatives of the free energy, but not in the first one. For example this happens to the specific heat:

$$C = -\frac{T}{V}\frac{\partial^2 F}{\partial V^2}$$

in a second order phase transition. The order of derivative at which discontinuity occurs defines the order of the transition.[2] In all the first order phase transitions, whose most famous example from everyday life is maybe the liquid-gas transition, not only the entropy is discontinuous but also all other quantities like internal energy, enthalpy, volume. Those transitions are coupled with a release or absorption of heat, known as latent heat. Following Landau let's try a phenomenological approach, we

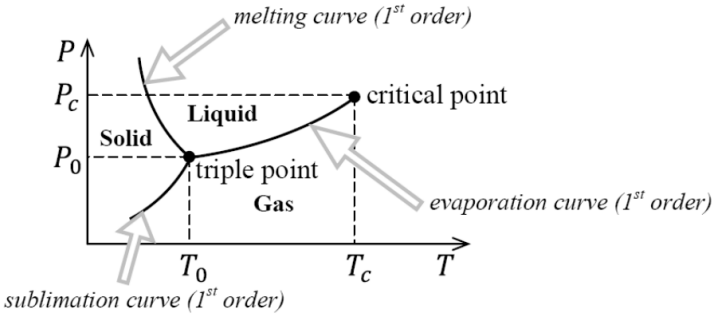


Figure 1.1. Pression-Temperature diagram for the first order phase transitions for water.

don't want to know anything about the detailed interactions at the atomic levels but let's focus on the system's symmetry. This way is also more useful to understand a little better the glass transition.[3]

[1] For a system in a state with entropy S, internal energy U at equilibrium with the environment at a temperature T: free energy is the amount of energy that can be used to do work in a system. $F = U - TS$.

[2] With a fractional derivative in theory we could also define fractional orders of phase change, see [8].

[3] Keep in mind that this is only a bird's-eye view of the subject, not an exhaustive explanation.

Let's take a substance and imagine a liquid-gas transition. The two phases are stable for a wide range of temperatures, but the free energy changes form in the two phases. It is dependent on temperatures in different ways but the values at the phase change temperature T_c are equal. At the same pressure levels a liquid with temperatures $< T_c$ is stable, a gas below that temperature would me metastable, undercooled in this case. Likewise above T_c the gas is stable, a liquid over that temperature would be overheated and so unstable. In this framework the second order phase transitions are different. The key idea to consider is symmetry: one phase is more symmetric than the other one. For example in the second order transition from ferromagnetic to paramagnetic, the magnetic phase, less symmetrical is the ferromagnetic one. There is a continuous increase in symmetry as $T \to T_c$. At the critical value the magnetization is zero, after that point the material becomes paramagnetic. In this case we were talking about a time-reversal symmetry. Another example could be the gauge invariance symmetry breaking in a superconductive material. So a second-order phase transition is a symmetry breaking.[4] The Landau theory of second order phase transitions stems from these ideas and defines a quantity, the order number η, whose value describes the symmetry of the systems. It is zero in the high symmetry region[5], has a finite value in the low symmetry phase and it is continuous. So its derivative must be discontinuous. The order parameter is a generic quantity and can represent different quantities for different phase transitions. It can also need not to be a scalar, for example for the ferromagnetic transition it is the magnetization which is a vector, instead for the superconductive one it is a complex scalar. A last glimpse at this theory, we said that η is zero at the critical point, so we can imagine to expand the quantity known as Landau free energy around the critical point until the fourth order.

$$F(\eta) = F_0 + F_1\eta + F_2\eta^2 + F_3\eta^3 + F_4\eta^4 + \cdots - \eta hV$$

In the ferromagnetic example, as written: η is the magnetization, h is the external magnetic field and V is the volume. From symmetry arguments it is found that the $F_1\eta$ term always vanishes and also the third order usually does.[6] This elegant theory predicts a lot of important physical quantities like latent heat but maybe it is more important because applying it we can group together systems with similar symmetries but very different constituent components.

An important parameter: the correlation length

Let's set ourselves in a general framework and let's take a scalar field. If we want to get information on the system we need to take the average over all the possible configurations. The probability of a given configuration ϕ is $e^{-\beta F[\phi]}$ where F is our free energy functional. The so called ensemble average of a quantity Q is

$$\langle Q \rangle = \frac{\int \mathcal{D}\phi Q e^{-\beta F[\phi]}}{\int \mathcal{D}\phi e^{-\beta F[\phi]}}$$

[4] If a symmetry group $\mathcal{L}$ describes the phase with low-symmetry and a group $\mathcal{H}$ the high-symmetry one, a second order phase transition is possible if $\mathcal{L}$ is a subgroup of $\mathcal{H}$.

[5] Note that a higher symmetry means more disorder.

[6] It does not if it is on the critical point of a first order phase transition.

where $\mathcal{D}\phi$ is a pseudo differential indicating all possible ϕ. $\langle Q \rangle$ is a weighted average over all the possible configurations, it is not important now how we calculate it.$\langle \phi(\mathbf{x}) \rangle$ is the average value of the field at the point $\mathbf{x}$. The quantity

$$C(\mathbf{x}, \mathbf{y}) = \langle \phi(\mathbf{x})\phi(\mathbf{y}) \rangle - \langle \phi(\mathbf{x}) \rangle \langle \phi(\mathbf{y}) \rangle$$

is the *correlation* between the fields at the point $\mathbf{x}$ and $\mathbf{y}$. Of course if the fields' values are completely independent[7] from the points at which they are measured than we have $\langle \phi(\mathbf{x})\phi(\mathbf{y}) \rangle = \langle \phi(\mathbf{x}) \rangle \langle \phi(\mathbf{y}) \rangle$ and so $C(\mathbf{x}, \mathbf{y}) = 0$. If $C(\mathbf{x}, \mathbf{y}) > 0$ both the fields tend to be or above average or below average, otherwise if $C(\mathbf{x}, \mathbf{y}) < 0$ the tend to have a reciprocal opposite behavior. With translational and rotational invariance this function tends to assume the asymptotic form

$$C(\mathbf{x}, \mathbf{y}) \approx \frac{e^{|\mathbf{x}-\mathbf{y}|/\xi}}{|\mathbf{x} - \mathbf{y}|^{\alpha}}$$

for some values of α. The *correlation length* is ξ. It measures the average size of fluctuations in the field: if $|\mathbf{x} - \mathbf{y}| \ll \xi$ then the correlation length is large, in the opposite case the correlation length decreases exponentially. Studying the dependency on the temperature we find that for a lot of systems typically $\lim_{T \to \infty} \xi(T) = 0$. This means that at high temperatures the random thermal effects dominate the field, even for very close points. As the temperature is lowered towards T_c, the temperature is not high enough to keep the field completely disordered, if $\xi \to \infty$ than there is no exponential decay of the correlation function and the fluctuations can be considered of arbitrarily large size. As a maybe counterintuitive result: it is common that $\lim_{T \to 0} \xi(T) = 0$. In this case the fluctuations are small obviously not because of the effect of the temperature. This behavior is explained by the "energy cost" of all the configurations besides the one at the lowest energy being too high because of the $e^{-\beta F[\phi]}$ factor. An example of a critical phenomenon modeled by this behavior is the magnetization of a 2D surface. If $\phi(\mathbf{x})$ represents the out of plane magnetization, then the correlation length ξ is the size of blocks which have the magnetization vector in the same direction. If $\xi \to \infty$ the fluctuations cover the whole system and the plane becomes ordered. A famous result for the Ising model is

$$\xi(T) \approx |T - T_c|^{-\nu}$$

for a two dimensional system we have $\nu = 1$, where ν is known as the *critical exponent*. For an interesting interactive demo of this phenomenon, see [7], for a figure see 1.2.

1.1.2 Some problems with glass

In a more general formulation the glass transition is the change of a system made of many components from a fluid-like equilibrium state to a non equilibrium disordered solid-like state [2]. This transition appears similar to the second order phase transition, but it is not easy to say that the two phenomena belong to the same class. Liquids and gas tend to relax, this means they have a tendency to lose

[7] In general independence is a stronger condition than absence of correlations.

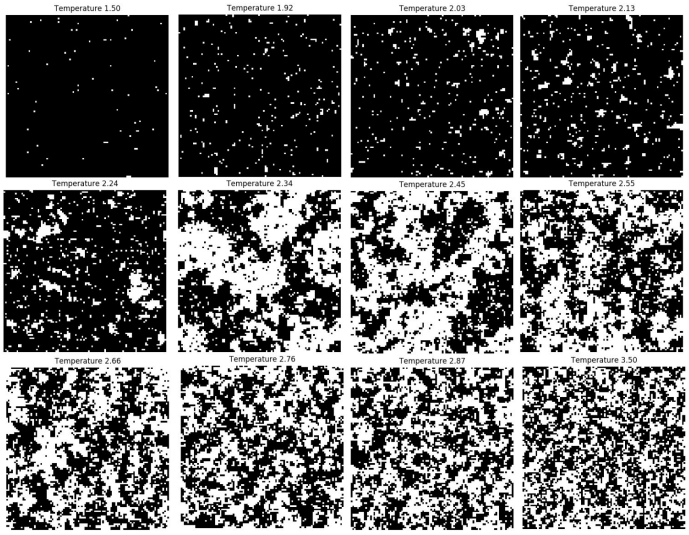


Figure 1.2. Spin clusters in a two dimensional 100×100 Ising model. $+1$ spins are black pixels, -1 are white pixels. The length scale of a cluster is the correlation length ξ. The correlation length grows increasing the temperature and diverges at T_c. Beyond T_c it starts decreasing and in the limit of infinite temperature it is zero.

correlation from their initial position, or configuration in general. A change in this behavior is usually due to a similar change in the internal structure of the substance, let's think about the water becoming ice crystals. Glass transitions are different: a fluid can be supercooled, stop moving and yet retain the liquid microscopical structure. There are a lot of theoretical questions still open in the glassy systems world and to treat them is of course beyond the scope of our study. Let's just make an example: in glasses there is no clear quantity to be defined as correlation length that diverges near the transition. Its existence, though, cannot be excluded with a theoretical justification, maybe we are just looking at the wrong correlation functions[8]. Another quantity whose trends baffles us is viscosity. It is defined in terms of autocorrelation function as:

$$\eta = \frac{V}{k_B T} \int_0^\infty \left\langle \sigma^{ab}(t_0)\sigma^{ab}(t_0+t) \right\rangle_{t_0} \mathrm{d}t$$

where V is the system volume, k_B is the Boltzmann constant, T is the temperature and σ^{ab} is the ab component of the stress tensor. The average is done over a starting equilibrated configuration, assuming that the ergodic hypothesis is valid[9]. In a glass the local fluctuations in the stress tensor do not lose correlation over time, and this is possible only if there is a small number of configurations explored by particles with appreciable probability, i.e. the system is not ergodic, see [2]. Viscosity is at least fifteen times higher in glasses than in liquids but if we probe the structure of a liquid and of an amorphous solid measuring the radial distribution function, we get

[8] To tell the truth there has been some success in this field [9].
[9] All accessible configuration have the same probability when $t \to \infty$.

only small structural changes, not apparently enough to justify such a big variation in a macroscopic quantity as viscosity. This behavior is different from the usual phase transitions which are characterized by the appearance of a structural order.

The radial distribution function

The radial distribution function $g(r)$ for a system with two kinds of particles (A and B), like the one we simulated in our study, is defined as:

$$g_{\alpha\alpha}(r) = \frac{V}{N_\alpha(N_\alpha - 1)} \left\langle \sum_{i=1}^{N_\alpha} \sum_{\substack{j=1 \\ i\neq j}}^{N_B} \delta(r - |\mathbf{r}_i - \mathbf{r}_j|) \right\rangle$$

$$g_{AB}(r) = \frac{V}{N_A N_B} \left\langle \sum_{i=1}^{N_A} \sum_{j=1}^{N_B} \delta(r - |\mathbf{r}_i - \mathbf{r}_j|) \right\rangle$$

where $r = r_{ij} = |\mathbf{r}_i - \mathbf{r}_j|$ and $\alpha \in \{A, B\}$. The Dirac delta $\delta(r)$ is the number density for a particle at $r = 0$ This quantity is basically a measure of the distribution of particles distances r from a reference particle i, see fig. 1.3. A maybe more intuitive interpretation is thinking of it as the probability to find a particle at position $\mathbf{r}$ at a time t knowing that there was a particle at the origin of the system at time $t = 0$ A generalization is the Van Hove correlation function whose definition is in [12]. Given a system of N particles where $\mathbf{r}_i(t)$ is the position vector of the i-th one at time t, the function is defined as:

$$G(\mathbf{r}, t) = \frac{1}{N} \left\langle \sum_{i=1}^{N} \sum_{j=1}^{N} \int \delta(\mathbf{r} + \mathbf{r}_j(0) - \mathbf{r}_i(t)) \right\rangle$$

Which can be rewritten as:

$$\begin{aligned} G(r, t) &= \frac{1}{N} \left\langle \sum_{i=1}^{N} \delta(\mathbf{r} + \mathbf{r}_i(0) - \mathbf{r}_i(t)) \right\rangle + \sum_{i\neq j}^{N} \delta(\mathbf{r} + \mathbf{r}_j(0) - \mathbf{r}_i(t)) \\ &\equiv G_s(r, t) + G_d(r, t). \end{aligned}$$

G_s, the self part, tracks the average motion of the particle that was chosen at the origin at starting time, G_d instead describes the behavior of the other, different, $N-1$ particles. It is worth reporting the asymptotic behavior of this function in the "thermodynamic limit". With this expression we mean that we take both the limits, $N \to \infty$ and $V \to \infty$ keeping $\frac{N}{V} = \rho$ finite. The results are:

$$\lim_{r\to\infty} G_s(r, t) = \lim_{t\to\infty} G_s(r, t) = 0,$$
$$\lim_{r\to\infty} G_d(r, t) = \lim_{t\to\infty} G_d(r, t) = \rho.$$

To study glasses and phase transitions in general, wouldn't it be great if there was a function which can somehow quantify how a configuration at a certain time is similar to another one at another time t at a scale length r? Luckily that function

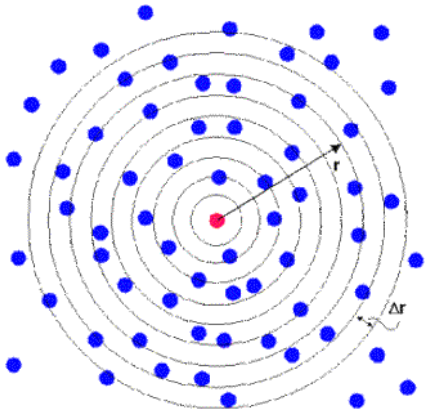


Figure 1.3. A graphical discretized representation of the meaning of the radial distribution function $g(r)$. The function counts the points between r and $r + \Delta r$. The area of this annulus, sometimes termed "shell", is $2\pi r \Delta r$. The red point is a central particle chosen as an example.

exists and is called the *intermediate scattering function*[10]. It is usually defined in terms of the wavenumber k, inverse of r. Choosing k we can probe the system at any level, the most interesting ones are the molecule level, $k = \frac{1}{d}$ where d is the particle diameter and the global when level $k \to 0$. It is usually defined as the spatial Fourier transform of the Van Hove correlation function (see box):

$$F(\mathbf{k}, t) = \int G(\mathbf{r}, t) e^{-i\mathbf{kr}} \, \mathrm{d}r$$

but can be explicitly rewritten as:

$$F(\mathbf{k}, t) = \frac{1}{N} \langle \rho_{-\mathbf{k}}(0) \rho_{\mathbf{k}}(t) \rangle = \frac{1}{N} \sum_{ij} \left\langle \mathrm{e}^{-\mathrm{i}\mathbf{k}\cdot\mathbf{r}_i(0)} \mathrm{e}^{\mathrm{i}\mathbf{k}\cdot\mathbf{r}_j(t)} \right\rangle$$

where

$$\rho(\mathbf{r}, t) = \sum_{i=1}^{N} \delta \left(\mathbf{r} - \mathbf{r}_i(t) \right)$$

is the particle density and

$$\rho_{\mathbf{k}}(t) = \sum_i \int \mathrm{e}^{\mathrm{i}\mathbf{k}\cdot\mathbf{r}} \delta \left(\mathbf{r} - \mathbf{r}_i(t) \right) \mathrm{d}\mathbf{r} = \sum_i \mathrm{e}^{\mathrm{i}\mathbf{k}\cdot\mathbf{r}_i(t)}$$

its Fourier transform. For a normal liquid the intermediate scattering function decays exponentially as intuitively expected. By lowering the temperature we get instead a more complex behavior, see fig. 1.4. At "intermediate" time scale glasses and supercooled fluids are in a plateau, this means that the particles are freezed. This is known as the β relaxation. Keep in mind that, for example a window glass stays in this regime for time longer of than the age of the universe, see [6]. For a supercooled fluid at longer times we see the onset of another form of decay of $F(k, t)$. This final process of decorrelation, termed the α regime has a trend

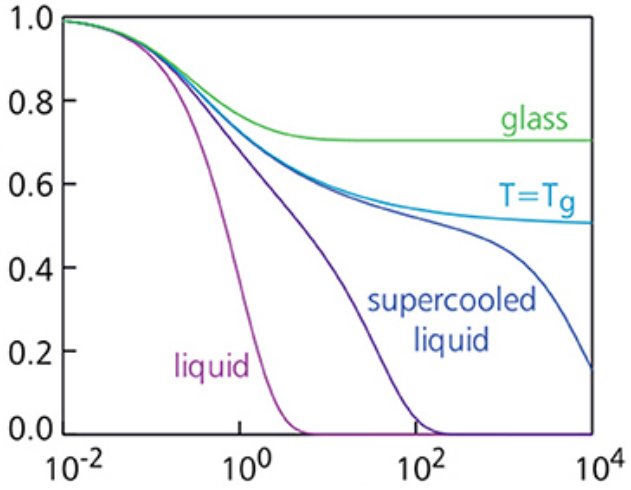


Figure 1.4. On the **x** axis time, on the **y** axis $F(k,t)$ fixed at $k \approx \frac{2\pi}{d}$ for different systems at different temperatures. d is the diameter of a molecule, so $F(k,t)$ is observed at the molecule scale length.

$\propto \left(e^{\frac{-t}{\tau}}\right)^{\beta}$ with $0 < \beta < 1$. The value of $f(k) = \lim_{t\to\infty} F(k,t)$ is called the non ergodicity parameter and can be used as the order parameter of the glass transition with the values 0 when the substance is in the liquid form and a finite positive value when in glass form. Before the β relaxation process begins, the particles move unimpeded in a "ballistic" way, see fig. 1.5 for a qualitative plot. Good results in predicting how the $F(k,t)$ evolves in time in a supercooling process have been achieved by the so called *Mode-Coupling Theory* [16]. Treating in detail this theory is beyond the scope of this work, but we report the main result, achieved with some reasonable approximations[11]:

$$\frac{d^2F(k,t)}{dt^2} + \frac{k_BTk^2}{mS(k)}F(k,t) + \int_0^t K_{MCT}(k,s)\frac{dF(k,t-s)}{dt}\,\mathrm{d}s = 0 \qquad (1.1)$$

where: k_B is the Boltzmann constant, m the mass of the particle and ρ the simple bulk density of the fluid $\frac{N}{V}$.

$$S(k) \equiv F(k,0)$$

is the *static structure factor*. This can indeed be considered a quantity that contains information on the structure of the material, before any decorrelation process has occurred.

$$K_{MCT}(k,t) = \frac{\rho k_BT}{16\pi^3 m}\int \left|V_{\mathbf{q},\mathbf{k}-\mathbf{q}}\right|^2 F(q,t)F(|\mathbf{k}-\mathbf{q}|,t)\,\mathrm{d}\mathbf{q}$$

$$V_{\mathbf{q},\mathbf{k}-\mathbf{q}} = \frac{1}{k}[(\mathbf{k}\cdot\mathbf{q})c(q) + \mathbf{k}\cdot(\mathbf{k}-\mathbf{q})c(|\mathbf{k}-\mathbf{q}|)]$$

[10] This quantity is usually measured with neutron or X-ray scattering experiments, that's why there is the adjective "scattering" in the name

[11] MCT starts from first principles, basically only the microscopic configuration at $t = 0$ of a liquid, but without approximations we get an exact but unsolvable model.

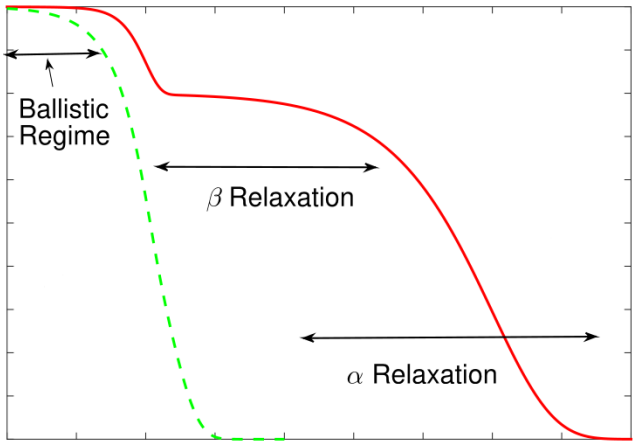


Figure 1.5. On the **x** axis time, on the **y** axis a qualitative curve of $F(k,t)$. The dashed green line represents a high temperature fluid, the solid red one a glass former one.

where $c(k) = \frac{1}{\rho}\left(1 - \frac{1}{S(k)}\right)$. $V_{\mathbf{q},\mathbf{k}-\mathbf{q}}$ indicates how strong is the coupling between different density modes at wave vectors q and $k-q$ [4] and overall K_{MCT}, known as *memory function*, can be thought as a four points correlation function for the density. Equation 1.1 looks like the one describing a damped harmonic oscillator and predicts fairly well the $F(k,t)$ evolution, see fig. 1.6.

1.2 The cage effect and the mean squared displacement

The spatial version of then $F(k,t)$ behavior described in the previous section is the so called "cage effect". In a supercooled liquid the particles are trapped in cages formed by their neighbors which are in turn trapped themselves by other particles. In this stage the particles experience only a vibrational motion, this is the β relaxation mode described in the previous part. After a sufficiently long time the particles manage to escape this "cage" and enter a different motion regime which is described in terms of $F(k,t)$ by the α relaxation mode. For a idealized picture see fig. 1.8. The quantity we tried to predict in our study can be useful to track this behavior, let's spend a few words on it.

1.2.1 The mean squared displacement

The mean squared displacement $\langle r^2(t)\rangle$, defined as

$$\left\langle r^2(t)\right\rangle = \left\langle |\mathbf{r}(t) - \mathbf{r}(0)|^2 \right\rangle \tag{1.2}$$

$$= \frac{1}{N}\sum_{i=1}^{N} |\mathbf{r}(t) - \mathbf{r}(0)|^2 \tag{1.3}$$

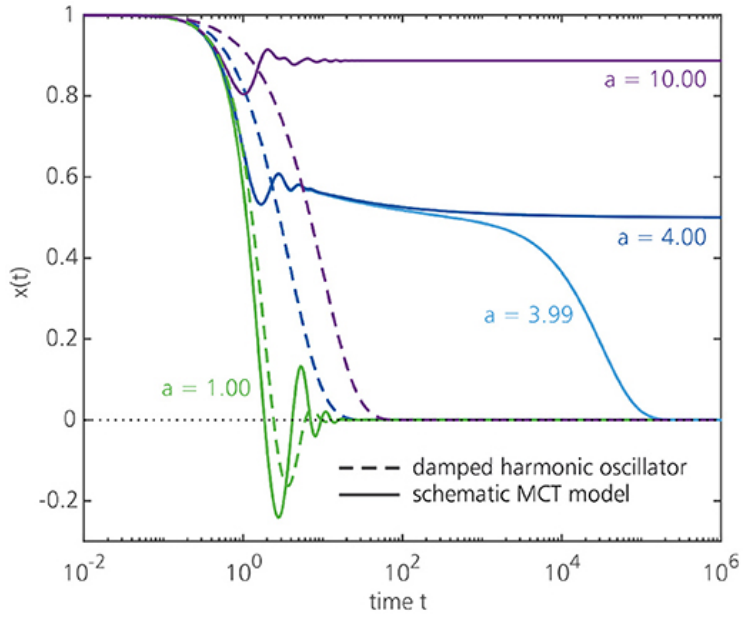


Figure 1.6. Dashed lines are the solutions for a simple damped harmonic oscillator whose equation is $\frac{\mathrm{d}^2x(t)}{\mathrm{d}t^2} + \omega^2 x(t) + \frac{\mathrm{d}x(t)}{\mathrm{d}t}$, while the solid ones are solution for a schematic MCT equation $\frac{\mathrm{d}^2x(t)}{\mathrm{d}t^2} + \omega^2 x(t) + a\int_0^t x^2(t-s)\frac{\mathrm{d}x(s)}{\mathrm{d}s}\,\mathrm{d}s$, $\omega^2 = 1$ for both equations. Different curves for different values of a are plotted. $a \geq 4$ is the critical value for which there is no return to the ergodic regime.

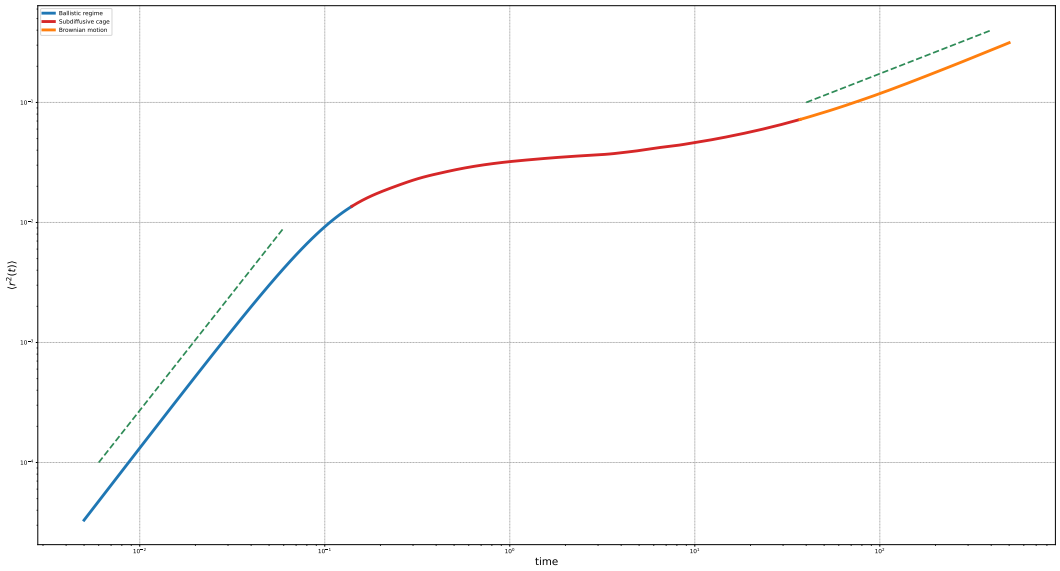
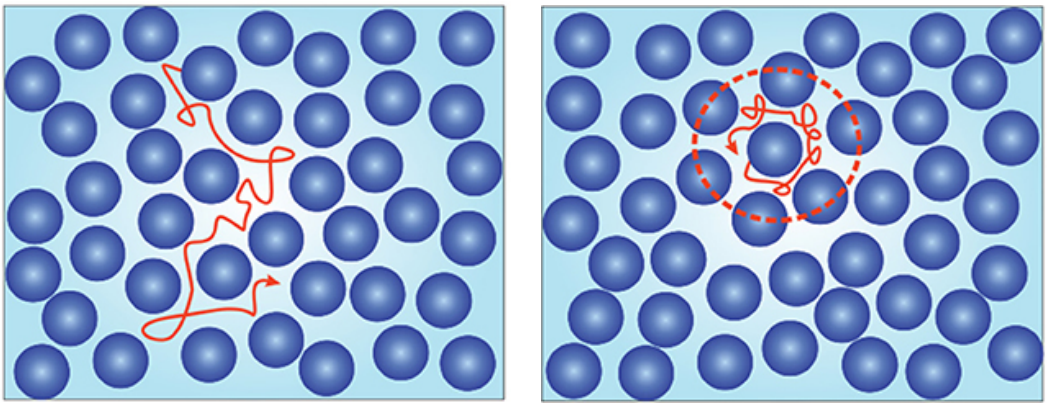


Figure 1.7. The three regimes of the Mean Squared Displacement. The blue one is ballistic, the red one is subdifussive and the orange one is Brownian. The two dashed green lines have slopes respectively of 2 and 1. The scale is double logarithmic and the times are in Lennard-Jones reduced units, more on this on chapter 2.



(a) Liquid phase **(b)** Glassy phase

Figure 1.8. A schematic representation of a particle motion during the two phases. In 1.8a we see the particle path in the liquid phase, while in 1.8b we see what happens in the glassy phase. The cage is highlighted as a red circle.

is a way to measure how far a particle has moved from the starting position to a final position at time t[12]. This quantity describes the dynamics of the system in a somewhat direct way Looking at fig. 1.7 three regions are clearly highlighted. The first one is known as the ballistic regime, and can be derived approximating $\mathbf{r}_(t) \approx \mathbf{r}(0) + \mathbf{v}(0)t$ and substituting in eq. 1.3. We get:

$$\left\langle r^2(t) \right\rangle = \frac{1}{N} \sum_{i=1}^{N} |\mathbf{v}(0)t|^2 = Ct^2$$

where C is a constant term. Taking the natural logarithm of both sides we obtain

$$\ln\left(\left\langle r^2(t) \right\rangle\right) = \ln(C) + 2\ln(t)$$

and that's the explanation of the line with slope 2 drawn in 1.7. We are basically saying that, for short times, no collision happens: the particles are moving with the same initial velocity $\mathbf{v}(0)$. After the glassy plateau, a regime with slope 1 starts. This is the result we get for a Random Walk, see box. This increase in the mean square displacement can be interpreted, besides the theoretical explanation provided by Mode Coupling Theory, as a particle escaping its cage and entering a diffusive regime, well modeled by a random walk.

MSD in a 1D Random Walk

In a one dimensional Random Walk the position of a particle after n steps is

$$x_n = \sum_{i=1}^{n} s_i$$

[12] In eq. 1.3 the average is calculated on the number of particles, but when the system is in equilibrium the mean squared displacement is independent of time and the square brackets could be also interpreted as a time average.

where s is the step size, suppose a fixed step size of ± 1. We have:

$$x_n^2 = \sum_{i=1}^{n} \left(\sum_{j=1}^{n} s_i s_j \right)$$

We are interested in averaging over time. All the terms with $i \neq j$ will have an equal probability of being $+1$ or -1, so the average value will be 0. The terms with $i = j$ will always be one, so we obtain:

$$\left\langle x_n^2 \right\rangle = \sum_{i=j=1}^{n} s_i s_j = \sum_{i=1}^{n} s_i^2 = \sum_{i=1}^{n} 1 = n$$

The result can be generalized for a random walk in d dimensions with a step size of l obtaining $\langle x_n^2 \rangle = dl^2 n$ as long as the probability of stepping in any of the $2d$ possible ways on a lattice of d dimensions is equal for each direction. The key outcome is that the MSD in a random walk is proportional to the number of steps n and so to the time t.

1.2.2 The Lennard-Jones potential

The classical potential used to model glass forming reaction is the Lennard-Jones one whose general form is:

$$V_{ij} = \frac{n}{n-m} \left(\frac{n}{m} \right)^{\frac{m}{n-m}} \epsilon \left[\left(\frac{\sigma}{r_{ij}} \right)^n - \left(\frac{\sigma}{r_{ij}} \right)^m \right] \tag{1.4}$$

These models have been studied extensively and found to be very good to model glass formers. Further details on the parameters choices (known as the Kob-Andersen-Lennard-Jones parameters) in chapter 2. The exponents usually chosen are $n = 12$ and $m = 6$,

$$V_{ij} = 4\epsilon \left[\left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^{6} \right] \tag{1.5}$$

where ϵ is the potential well depth and describes how strongly the particles interact, σ is the finite distance at which the potential is zero and represents the distance of the closest approach of two particles and thus defines a length scale. $r_{ij} = |\mathbf{r}_i - \mathbf{r}_j|$ is the distance between the particle i at the position $\mathbf{r}_i$ and the particle j at the position $\mathbf{r}_j$. The positive repulsive term $\propto (\frac{\sigma}{r})^{12}$ quickly vanishes beyond short ranges and should model the repulsion caused by overlapping orbitals. This term has no theoretical justification, it just seems to fit well with data gathered from experiments with diatomic gasses [17] and is extensively used in simulations. In contrast the attractive negative term $\propto (\frac{\sigma}{r})^6$ is calculated using perturbation theory and is basically the second-order correction to the dipole-dipole interaction between two atoms, see [21]. To be slightly more explicit on the parameters meaning: $V(\sqrt[6]{2}\sigma) = -\epsilon \implies$ that $V(\sqrt[6]{2}\sigma, -\epsilon)$ is the (only) potential minimum in $(0, \infty)$.

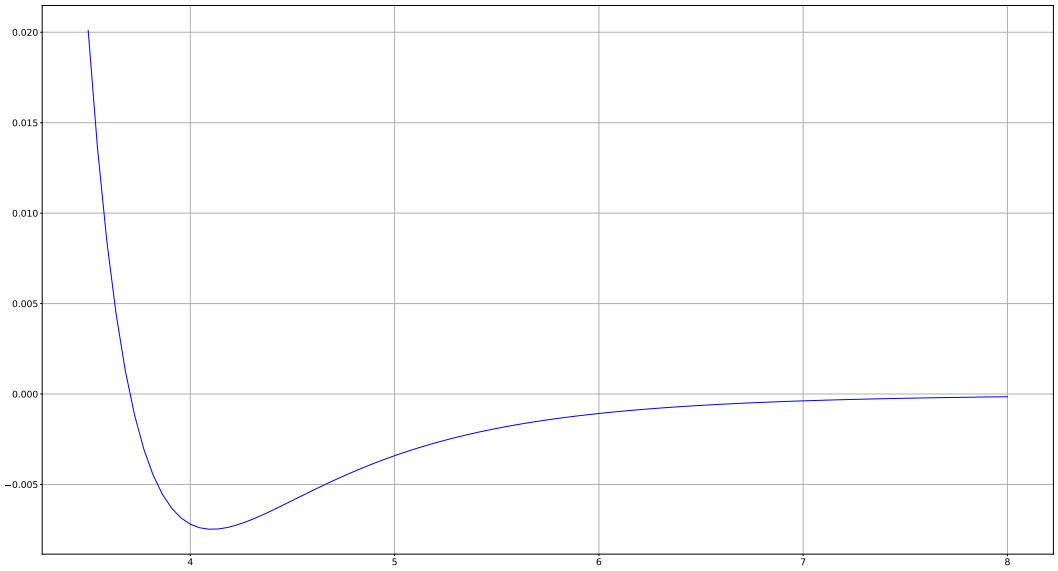


Figure 1.9. Lennard Jones potential vs distance

Chapter 2

Deep into simulation details

This chapter is dedicated to the details of the numerical simulation. We start by explaining some principles of molecular dynamics simulations. We then explain some methods that make the software we used, LAMMPS, so efficient. Finally we provide some details on how we generate the samples that will form our dataset, focusing on the two ensembles, NVE and NPT, we used.

2.1 Molecular Dynamics Simulations

Basic Principles of Molecular Dynamics

Suppose we have a constant number N of particles of mass m at random positions in a three-dimensional box. We know that their dynamics obey Newton's second law:

$$\mathbf{F}_i = m_i \mathbf{a}_i, \tag{2.1}$$

where $\mathbf{a}_i$ is the acceleration on particle i. The force in equation 2.1 is in its most general form, it can be caused by a lot of factors such as drag, an external field of any kind etc. In our simulation we are only taking into consideration conservative forces caused by the interaction of particles between them. To further simplify the model and speed up the computation (but without losing too much generality) we assume also a pairwise interaction between the particles. So a conservative Force calculated between couples of particles would have a potential of the following form.

$$V = \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} V_{ij} \tag{2.2}$$

The velocity Verlet algorithm

To make a simple example of a numerical integration procedure in a Molecular Dynamics simulation we have to introduce a basic algorithm. Let's set $m = 1$, and in the most general way possible, calculate the Taylor's expansion to second order of $\mathbf{r}_i(t + \Delta t)$ around $\mathbf{r}_i(t)$. We get:

$$\mathbf{r}_i(t + \Delta t) = \mathbf{r}_i(t) + \mathbf{v}_i(t)\Delta t + \frac{1}{2}\mathbf{a}_i(t)(\Delta t)^2 \tag{2.3}$$

Doing the same for $\mathbf{r}_i(t)$ expanded around the point $t + \Delta t$[1] we get:

$$\mathbf{r}_i(t) = \mathbf{r}_i(t + \Delta t) - \mathbf{v}_i(t + \Delta t)\Delta t + \frac{1}{2}\mathbf{a}_i(t + \Delta t)(\Delta t)^2$$

Calculating $\mathbf{v}_i(t + \Delta t)$ from the two previous equations in terms of $\mathbf{a}$ we finally get:

$$\mathbf{v}_i(t + \Delta t) = \mathbf{v}_i(t) + \frac{1}{2}\left[\mathbf{a}_i(t) + \mathbf{a}_i(t + \Delta t)\right]\Delta t \tag{2.4}$$

Eq. 2.3 and 2.4 are the positions and velocities update procedures of the *Velocity Verlet* algorithm. This is the standard one use by the software LAMMPS to perform simulations in the Microcanonical Ensemble. This ensemble is also known as

[1] Going back in time.

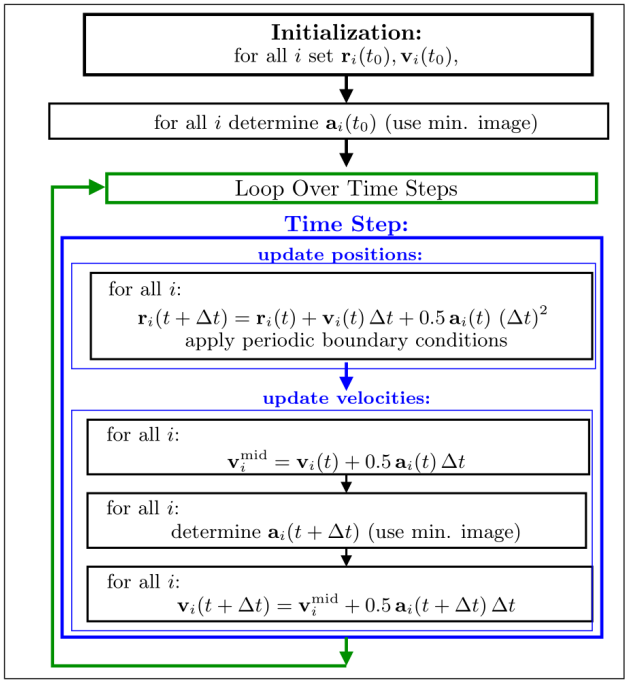


Figure 2.1. The schematic updating procedure for the positions and the velocities of the velocity Verlet algorithm.

NVE ensemble because it is modeling a system in which the number of particles N, the Volume V and the Energy E are constant. This ensemble and this integration technique was the one we used to calculate the mean squared displacement of our configuration once we got a stable starting point. The Velocity Verlet algorithm preserves time reversal symmetry and this implies energy conservation. For a detailed explanation see [24] but the main idea is that if a volume of the phase space that contains every configuration with a fixed energy E does not remain the same after a time reversal operation, this volume will change and the systems will access states with values of energy different from E. Remember that this simple updating algorithm can only be applied if $\mathbf{a}_i(t + \Delta t)$ does not depend on $\mathbf{v}_i(t + \Delta t)$, in other words accelerations have to depend only on the position of the particles, not on the velocities. We won't be so lucky integrating in the NPT ensemble. Usually calculating the accelerations takes most of the time in this algorithm. The updating scheme is well synthesized in the flowchart in fig. 2.1 taken from [18].

More on the Lennard Jones potential and the Kob-Andersen model

Let's give some more specific details on the Lennard-Jones potential introduced in chapter 1:

$$V_{ij} = 4\epsilon \left[\left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^{6} \right] \tag{2.5}$$

We notice that in this 12-6 Lennard-Jones potential the repulsive term is an integer power of the attractive one, and this can, depending on the implementation

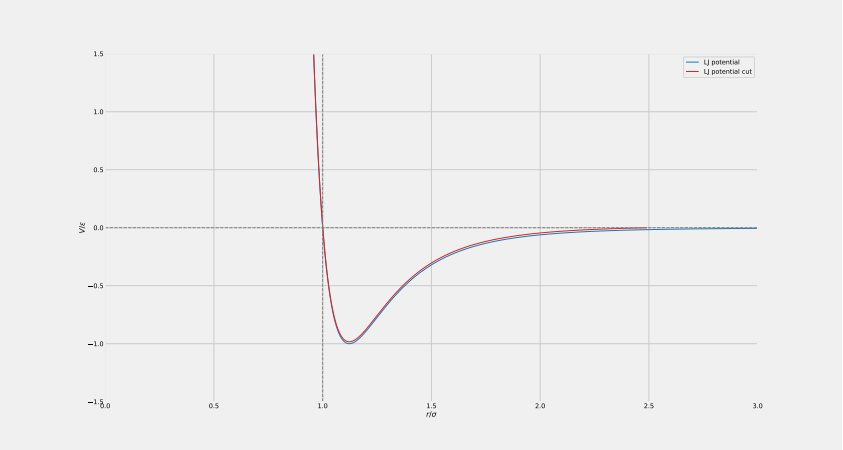


Figure 2.2. Lennard-Jones potential in its full long range form and in its shifted and cut one.

(compiler and libraries optimization) lead to faster calculation[2]. The result of the explicit calculation of the force acting on a particle is:

$$\mathbf{F} = -\nabla V = \frac{\partial V}{\partial r} = -4\epsilon\left(-12\frac{\sigma^{12}}{r^{13}} + 6\frac{\sigma^6}{r^7}\right) = 48\epsilon\left(\frac{\sigma^{12}}{r^{13}} - \frac{1}{2}\frac{\sigma^6}{r^7}\right) \tag{2.6}$$

In general this potential is appropriate for systems made with elements whose outer electron shell is complete, for example gases such as Argon or Krypton. However it proved to be useful in a variety of situation and is simple enough to make us focus on fundamental questions not related to a specific properties of a particular constituent of the system such as a molecule or a polymer, etc. It is clear from a straightforward calculation that $\lim_{r_{ij}\to\infty} V_{ij} = 0$, and this result has two consequences. The first is that our potential is not suitable to model inherently long-range interactions (such as Coulomb or gravitational) in such cases different ones are needed, see [22]. The second one is that it is practically useless to keep calculating the interaction of two particles beyond a certain cut threshold. So, to preserve precious computing time, it is normally used[3] a cut and shifted form of the Lennard-Jones potential, V_{ij}^c, eq. 2.8, that basically means that after a certain distance r^c the potential is zero. An upward shift is also applied in order to prevent the unrealistic jump in energy that would be caused by the abrupt cut. A simple plot of the Lennard-Jones potential (full and cut) is shown in fig. 2.2.

$$F_{i,x} = -48\sum_{j\neq i}\epsilon\left(\frac{\sigma^{12}}{r_{ij}^{14}} - \frac{1}{2}\frac{\sigma^6}{r_{ij}^8}\right)(x_i - x_j) \tag{2.7}$$

$$V_{ij}^c = \begin{cases} V_{ij}(r_{ij}) - V_{ij}(r^c) & \text{if } r_{ij} < r^c \\ 0 & \text{if } r_{ij} \geq r^c \end{cases} \tag{2.8}$$

[2] Another positive, maybe unintended, consequence of that choice of exponents.
[3] And we stick to that.

Table 2.1. Basic units in the Lennard-Jones potential

Symbol	Quantity
ϵ	Energy [J]
σ	Length [m]
m	Mass [Kg]

Our cut-off distance was set at $r \geq 2.5\sigma_{AA}$ following [31]. In the classical article by Kob and Andersen [20] the authors develop a model to better describe the glassyfication of amorphous $Ni_{80}P_{20}$. This is achieved also by tweaking the Lennard-Jones potential. In the potential form given in eq. 1.5 there is no distinction between particle types: every particle interacts in the same way with the other ones. In [20] the model used is a mix of 80% of particle of a kind (type A) and 20% of another kind (type B). The mass of both particles is set to $m_A = m_B = 1$, but what differentiate them is their "size" (σ) and their mutual energy constant of interaction ϵ. To account for this difference, the potential is modified, as is easy to imagine, with different energy and distance constants for the different possible kinds of interactions. We define the Kob-Andersen potential as the following version of the Lennard-Jones one:

$$V_{ij} = V_{\alpha\beta}\left(r_{ij}\right) = 4\epsilon_{\alpha\beta}\left[\left(\frac{\sigma_{\alpha\beta}}{r_{ij}}\right)^{12} - \left(\frac{\sigma_{\alpha\beta}}{r_{ij}}\right)^{6}\right] \tag{2.9}$$

where $\alpha, \beta \in \{A, B\}$ and A and B are the particle types. In [20] the authors chose the following values for those parameters:

$$\begin{aligned} \epsilon_{AA} &= 1.0 \quad \sigma_{AA} = 1.0 \\ \epsilon_{BB} &= 0.5 \quad \sigma_{BB} = 0.88 \\ \epsilon_{AB} &= 1.5 \quad \sigma_{AB} = 0.8 \end{aligned}$$

Their choice is now a de-facto standard and is followed also in [31] and [18]. Kob and Andersen picked those values in order to get a model of the amorphous $Ni_{80}P_8$ less prone to crystallization, like the one proposed by Weber and Stillinger in [15].

Reduced units and parameters choice

Say that we want to express all physical quantities in terms of the ones that appears in the potential, see tab. 2.1. How can we do that? Taking for example the time, to get the so called reduced time τ, remembering that $[J] = \frac{[kg][m^2]}{[s^2]}$ we proceed as follows

$$\sigma^a \epsilon^b m^c = \tau \implies [m]^a \left(\frac{[kg][m]^2}{[s]^2}\right)^b [kg]^c = \tau \implies [m]^{a+2b}[kg]^{b+c}[s]^{-2b} = \tau \implies$$

$$\implies a + 2b = 0, b + c = 0, -2b = 1 \implies a = 1, b = -\frac{1}{2}, c = \frac{1}{2} \implies \tau = \sigma\sqrt{\frac{m}{\epsilon}}$$

Repeating this procedure to the other relevant quantities we get that the temperature is expressed in units of $\frac{\epsilon}{k_B}$ and pressure in units of $\frac{\epsilon_{AA}}{\sigma_{AA}^3}$. The Boltzmann constant k_B is usually set to 1. The reduced units give[4] us better physical insight of what is happening.

2.2 The LAMMPS Molecular Dynamics Simulator

The best and most efficient way to perform a molecular dynamics simulation today is using the software known as LAMMPS (Large-scale Atomic/Molecular Massively Parallel Simulator)[19]. Since 2004 this open source software has become the standard for simulating a wide range of systems on different length scales: molecules, polymers and atomic scale objects, too. It is by design highly optimized for parallel computation[5] and provides direct access to a lot of parameters and simulations settings like for example the microcanonical and isothermal-isobaric ensembles we used to generate our dataset[6]. In our simulation we exploited the parallelism by using the server CPU cores, but LAMMPS provides native support to CUDA and OpenCL to use the GPU, as well. In the following section we will shortly give a glimpse on some features that make this software so efficient, of course with no pretense of completeness.

Partitioning

The box in which the simulation is performed is divided into non-overlapping subdomains,[7] a two dimensional example in fig. 2.3. Every subdomain is assigned to a computing unit, a CPU core or a GPU CUDA core for example. For systems with known highly not uniform density[8] LAMMPS supports dynamic subdomain repartitioning to balance the load among computing units. The purpose of this feature is to avoid having cores with a lot of calculations to do while leaving others almost idle, slowing down the whole simulation. This is not the case in our system: typically simulations at the atomic scale do not involve large variations in density for a long time, a normal partitioning of the simulation box into equal size subdomains is enough. Each computing units is responsible for storing information[9] on its subdomain but also stores copies of that information for particles owned by other processors within a cut-off radius from the boundary of its subdomain. Those particles are known as *ghosts* and are essential to calculate short range interactions for particles at the boundaries. In the timesteps when a particle goes from a processor to another each computing unit creates a list of its own atoms which are ghosts for

[4] Or should give.

[5] Remember that eq. 2.1 is a short way to write $d \times N$ coupled equations for a system of N particles in d dimensions.

[6] All that glitters is not gold, though. Sometimes we had to resort to some good old bash shell scripts to put the LAMMPS data in a more usable form or to generate random seeds.

[7] Keep in mind that this has nothing to do with periodic boundary condition replicas of the system, this is just a technique to parallelize computation.

[8] A lot of particles in a subdomain, and nearly zero in another one.

[9] Positions and velocities in our study, but in other simulation with non point-like particles also information about particles' orientation can be stored for example.

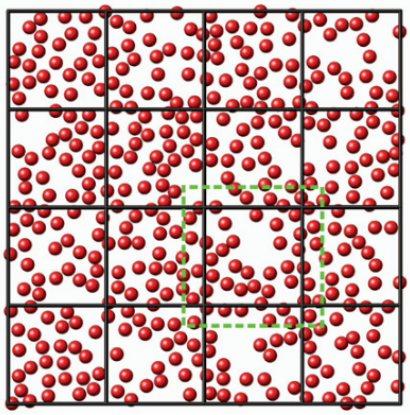


Figure 2.3. LAMMPS way of partitioning the simulation space. Every subdomain is assigned to a computing unit. The ghost particles for a subdomain are highlighted by the dashed green line.

neighboring processors, those lists are used to pass messages between processors in an efficient way.

Neighbor lists

Neighbor lists are a clever way to reduce the computational complexity of calculating the pair distances of all atoms.[10] That would be a $O(N^2)$ task for a system of N atoms: all distances have to be computed to check if an atom is within the cut-off distance r_c of the chosen potential. Instead of doing this, for every particle a list is created and in it are stored all its neighbors within a distance $r_n = r_c + s$ where s is known as the *skin distance*. The larger the skin distance, the more timesteps the list can be used, but of course at the expense of memory. This list is updated not at every timestep but, heuristically, only when any particle has moved half the skin distance. For a system of N particles with average number density of particles ρ_N, each particle will have:

$$N\frac{4}{3}\pi r_n^3\rho_N - 1$$

neighbors in its list, but the interaction between particles i and j need to be recorded only once so, we will have a total of

$$N\frac{2}{3}\pi r_n^3\rho_N - 1$$

particles' indexes to be stored. In our simulation with a force cutoff of 2.5σ the typical skin distance of 0.3σ was used. With these settings LAMMPS should recompute the neighbor list every 10 to 20 time steps. In LAMMPS these lists stored in a multiple-page data structure where each page is a contiguous block of memory. This allows pages to be incrementally allocated or deallocated as needed, neighbor lists typically consume the most memory of any data structure. An efficient way

[10] This trick is not used only by of LAMMPS but deserves to be explained because is an almost universal way of speeding up computations of molecular dynamics simulations, see [25] for an exhaustive explanation.

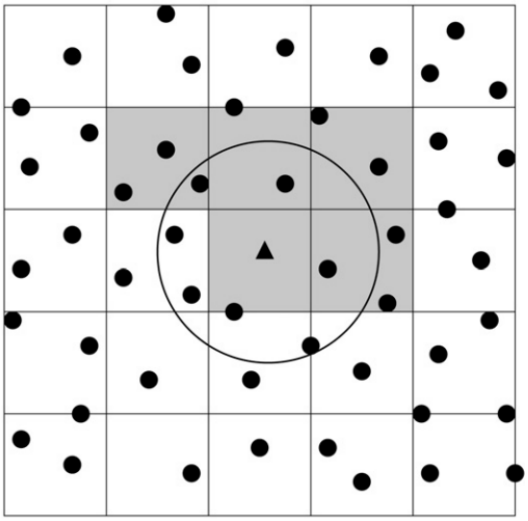


Figure 2.4. A possible binning for computing neighbor lists. The triangle represents an atom, the five shaded bins are the one to be checked. The radius of the circle is equal to the cut-off distance.

of constructing the neighbor list for a parallel computing system is to once again divide the box in bins and put each atom in one of them. Each of these bins can be assigned to a computing unit. To build the list for an atom i only the distance to atoms within the bins near to the one in which i lies must be checked. A typical choice is to make each bin edge $\geq r_n$. In 2 dimensions each atom should check only 9 bins see fig. 2.4, but if the distance between i and j is calculated only once for each pair, the system needs to check only 5 bins per atom.

2.3 How we got our sample

2.3.1 Starting configurations

We simulated the behavior of 4096 particles, 80% (3277) of the A type and 20% (819) of the B type. We created them at random position in a three-dimensional cubic box of L^3 volume. The length of each side of the box was $L = 9.4 \cdot 4.096 = 38.50$. This value was chosen to keep an initial density equal to that simulated in [20] and [18] but rescaled to account for our greater number of particles (in [20] $L = 9.4$ and $N = 1000$. One example of a starting configuration can be seen in fig. 2.5 and in fig. 2.6. Starting a simulation with particles randomly generated can be dangerous: if two particles are randomly created too close, the simulation can "blow-up". The force calculation can lead to very large velocities which the numerical integrator could not handle well. To mitigate this issue LAMMPS provides the command `minimize`. Minimization process basically consists in moving the atoms in an arrangement where the energy is at the nearest local minimum ideally, or just below a certain threshold. An idea to do that is to compute the new $r_{n+1} = r_n + \frac{F_n}{\max(|F_n|)}\eta_n$ where η_0 is a maximum displacement provided to the system, at each iteration the new force is recalculated and is accepted if it is lower than the

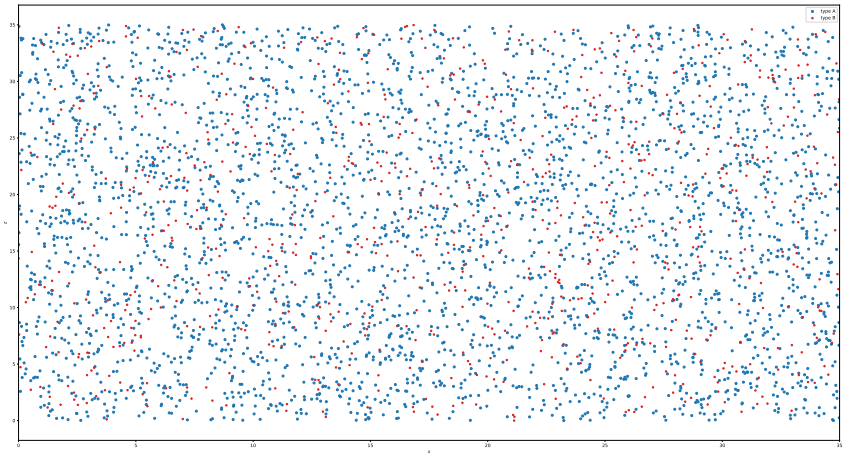


Figure 2.5. Two dimensional view on the XZ plane of a starting configuration. Type A particles are the bigger and blue ones, type B particles the red and smaller ones.

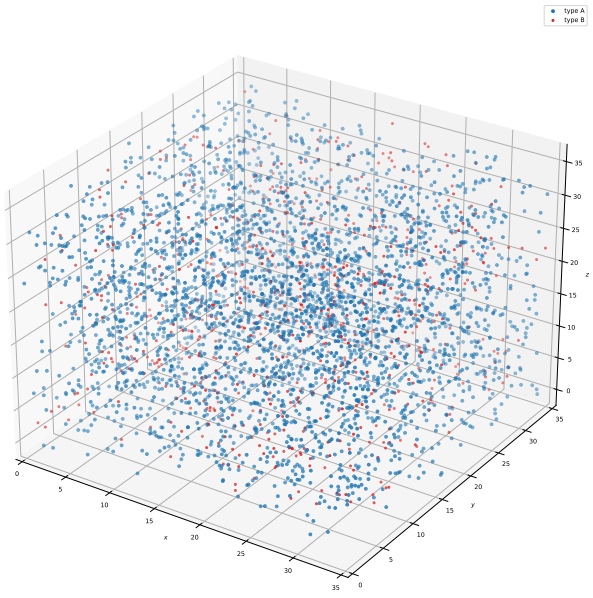


Figure 2.6. Three dimensional view of a starting configuration. Type A particles are the bigger and blue ones, type B particles the red and smaller ones.

one at the previous iteration. The process stops when some criteria are met. In the simulation we use the more sophisticated LAMMPS implementation of this procedure by running the

```
minimize 1.0e-4 1.0e-6 1000 1000
```

command. That means that the energy difference between two consecutive iterations divided by the total energy will be $\leq 10^{-4}$, and that all the three components of the force vector on any atom will be $\leq 10^{-6}$. All this process is tried for a maximum of 1000 iterations[11]. In our process at some point we will need, (see chapter 4, Dataset Generation) to discard the velocities and reset them randomly. At equilibrium the probability of a configuration[12] s $= P(s) \propto \exp\left(\frac{-E(s)}{k_B T}\right)$ where:

$$E(s) = \frac{1}{2}\sum_{i=1}^{N} m_i \mathbf{v}_i^2 + V(\mathbf{r}_i) \tag{2.10}$$

Combining the two previous equations with a bit of algebraic manipulations, it follows that the probability distribution for a component (for example x) of the velocity vector of the particle i is distributed according to the Maxwell-Boltzmann that is:

$$P(v_{i,x}) = \frac{1}{\sqrt{2\pi}\sigma_i} e^{\frac{-v_{i,x}^2}{2\sigma_i}} \tag{2.11}$$

where $\sigma_i = \sqrt{\frac{k_B T}{m_i}}$. As written before, in our simulation both $m = k_B = 1$ Obviously the y and z components have the same form. So, we have to choose a starting velocity in the form of 2.11, to do that LAMMPS provides the command

```
velocity all create 0.5495 1656 dist gaussian
```

The first number is the temperature in reduced units and the second one is a random seed.

Periodic boundary conditions Our goal was to simulate a realistic system of $N \approx 10^{23}$ without considering the boundary effect, but our systems deals with only 4096 particles. So it is somewhat necessary to replicate the original box to have more particles. Doing that obviously creates a problem at the boundaries: how to calculate the cut radius for a particle near the edge? We simply apply periodic boundary conditions with the minimum image convention, see fig. 2.7 taken from [18]. In this setting the original system is replicated and the distances between a particle i a particle j are calculated with the nearest form of j, whether is a particle in the original system or its replica. For example the particle 18 is considered within the cut-off radius even if the distance between its original form (bottom left corner of the original square) and the particle i is $> r^c$.

[11] To avoid this annoying energy minimization problem one can make the simulation start with the particles on a lattice, but that structure is not suitable for studying systems without long-range order such as glasses. However with a sufficiently long equilibration run the issue can be solved.

[12] The word "microstate" is often used in more general definitions, but for our system of classical particles with a Lennard-Jones-Kob-Andersen potential, a microstate is just a configuration of position and velocities $s = \{\mathbf{r}_i, \mathbf{v}_i\}$.

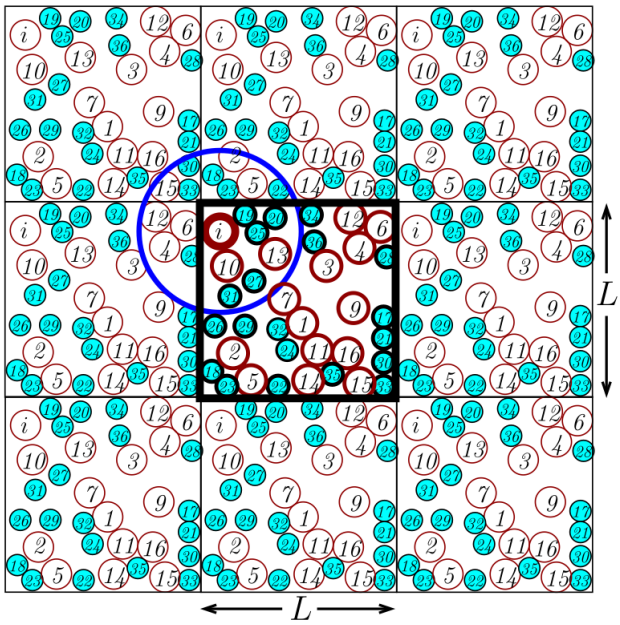


Figure 2.7. Periodic boundary conditions. The central square with bold boundaries is the original system, the other squares are its replicas. The blue circle is the cut off radius for the particle i.

2.3.2 NPT simulation

To study the glassy transition for a wide range of temperatures and pressures[13] we did the following. We start at a high temperature value ($T = 0.5495, P = 0.4888$) [14] and we equilibrate the system at that temperature using an isothermal-isobaric ensemble, we then reduce the temperature always keeping the NPT ensemble with various gradual steps towards the desired state point. After the values of temperature and pressure have been reached another extra NPT integration run is performed to enhance the configuration's stability. Each of these runs for temperature T lasted at least five relaxation times τ_g, empirically estimated from the Vogel-Fulcher-Tammann [26] equation:

$$\tau_g \approx e^{\frac{a}{T-T_0}}$$

where $a = 0.62 \pm 0.02$ and $T_0 = 0.387 \pm 0.001$. Another way to get an estimate of the relaxation time of a system is by studying the self intermediate scattering function, we didn't do that directly but used estimates taken from the plots found in [31]. We also verified the glassy transition temperature $T = 0.50$ estimated in [31] by checking the Temperature-Volume curve, see fig. 2.8. Finally we verified the glassification process by checking the Mean Squared displacement curves (see the NVE section) to evaluate if we got the expected trend for each temperature.

[13] Which can be found in chapter 4.

[14] Values picked according to the procedure followed by Bapst and others in [31]. There is only one configuration whose equilibrium temperature above this value ($T = 0.56$), it is the only one not involved in the glassification process, it was still included in the dataset

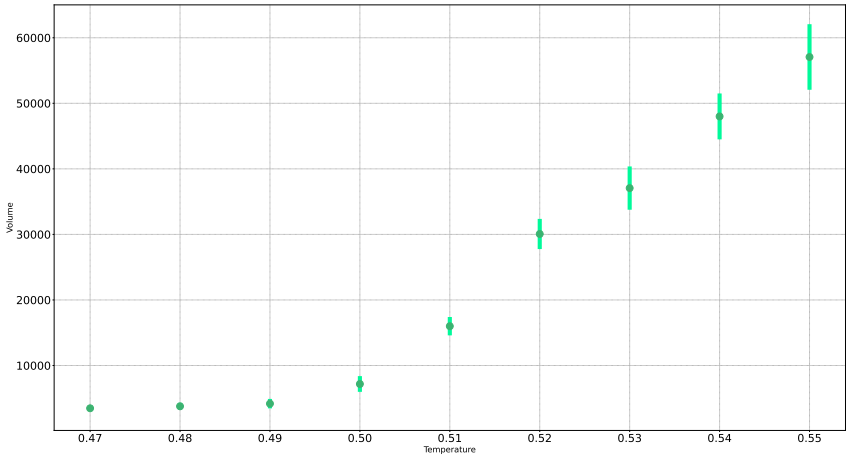


Figure 2.8. Temperature-Volume scatter plot for a configuration equilibrated at T = 0.47 during the cooling process. Only the values at T = 0.55, T = 0.54,..., T = 0.47 were saved. Error bars include 96% of values. There is a change of slope at the glassy transition temperature T = 0.50 as expected.

The LAMMPS command to perform an NPT simulation is:

```
fix 1 all npt temp T_start T_final iso T_damp P_start P_target P_damp
```

It should be clear from the syntax that we can set values for the initial and final temperatures and pressures. The damps parameters tell us how fast the temperature or pressure is modified towards the target value. The `Pdamp` value should be chosen wisely, if it is too small the pressure can have large fluctuations, if too large the system can take a lot of time to equilibrate. Luckily the rule of thumb provided by the LAMMPS manual seems to work right: the value was set at `100.0*dt` where `dt` is the length of a timestep.

Integration in the NPT ensemble

Integrating in the NPT ensemble has a crucial difference with doing the same in the microcanonical. The system is kept in thermal contact with a source of heat and its temperature stays the same during the simulation, not so its energy. The Hamiltonian is not conserved and is in the form $e^{-\frac{1}{k_B T}\mathcal{H}(x)}$. In order to implement such constant "temperature bath" we can no longer use the good old time reversible Newton's second law to describe the motion of the particles but we have to resort to extra parameters. The most famous procedure, and the one used by LAMMPS, to achieve this goal is the Nosé-Hoover [27] algorithm.[15] Newton's equation is replaced by this system:

$$\begin{aligned} \frac{\mathrm{d}^2\mathbf{r}_i}{\mathrm{d}t^2} = \ddot{\mathbf{r}}_i &= \frac{\mathbf{F}_i}{m_i} - \xi\dot{\mathbf{r}}_i \\ \frac{\mathrm{d}^2 \ln s}{\mathrm{d}t^2} = \dot{\xi} &= \frac{1}{Q}\left(\sum_{i=1}^{N} m_i\dot{\mathbf{r}}_i^2 - dNk_{\mathrm{B}}T\right) \end{aligned} \tag{2.12}$$

[15] The Nose-Hoover thermostat was originally designed for the NVT ensemble and was modified by Martyna and other in [28] to be adapted to the NPT. The basic principles remain the same.

where d is the number of system's dimensions, k_B the Boltzmann constant and Q the mass of the external temperature (T) source. The key idea is to introduce the variable ξ which models a fake friction whose purpose is to modify particles' acceleration until the temperature gets to the desired value. The eq. 2.12 are not fit to the velocity Verlet integration algorithm because $\ddot{\mathbf{r}}_i = \mathbf{a}_i$ depends on the velocity $\dot{\mathbf{r}}_i$ meaning that $\mathbf{a}_i(t + \Delta t)$ depends on $\mathbf{v}_i(t + \Delta t)$. This is not compatible with the velocity Verlet algorithm. Luckily for us in [29] Fox and Andersen devised the key ideas to apply a velocity-Verlet-like integration technique to a system in the following form:

$$\ddot{\mathbf{x}}(t) = f[\mathbf{x}(t), \dot{\mathbf{x}}(t), \mathbf{y}(t), \dot{\mathbf{y}}(t)]$$
$$\ddot{\mathbf{y}}(t) = g[\mathbf{x}(t), \dot{\mathbf{x}}(t), \mathbf{y}(t)].$$

which is exactly what we have in 2.12. Applying those ideas, the updating scheme becomes:

$$\mathbf{r}_i(t + \Delta t) = \mathbf{r}_i(t) + \mathbf{r}_i(t)\Delta t + \frac{1}{2}\left[\frac{\mathbf{F}_i(t)}{m_i} - \xi \mathbf{r}_i(t)\right](\Delta t)^2$$
$$\ln s(t + \Delta t) = \ln s(t) + \xi(t)\Delta t + \frac{1}{2Q}\left(\sum_{i=1}^{N} m_i \dot{\mathbf{r}}_i^2(t) - dNk_{\mathrm{B}}T\right)(\Delta t)^2$$

To get further details of these calculation refer to [24] and [30].

2.3.3 NVE simulation

Starting from our stable configurations we run a NVE simulation to calculate the mean squared displacement of our systems, for more theoretical details refer to chapter 1. In our simulation we calculate separate averages for the two different particle types:

$$\left\langle r_\alpha^2(t) \right\rangle = \frac{1}{N_\alpha} \sum_{i=1}^{N_\alpha} |\mathbf{r}_i(t) - \mathbf{r}_i(0)|^2 \tag{2.13}$$

where $\alpha \in \{A, B\}$. An example of mean squared displacement is given in fig. 2.9. Looking at this figure it is clear that after a rapid increase $\langle r_\alpha^2 \rangle$ reaches a plateau. It is worth to notice that the smaller type B particles reach a higher plateau. To investigate this behavior at a longer time scale first of all we will have to change to double logarithmic scale and we will have to save a lot more values of the MSD. Given the dimension of the dataset we can be afraid for our hard drive space. But are all those values really necessary? We know that the plateau regime starts "soon" and lasts for quite some time and that it will end for longer times returning back to a growth regime, linear in log-log scale. So, by intuition, we need to save a lot of values for short times and progressively less values for longer times. How many? To do that is enough to save data not at every time but at times separated by a multiplicative ratio A. So every $t_k = t_0 \cdot A^k$. Thinking in terms of timesteps:

$$\frac{t_k}{\Delta t} = \frac{t_0}{\Delta t} \cdot A^k \tag{2.14}$$

We know how many values we want to save, say m total values. So we can solve eq. 2.14 to get A.

$$A = \left(\frac{n}{t_0/\Delta t}\right)^{\frac{1}{m}}$$

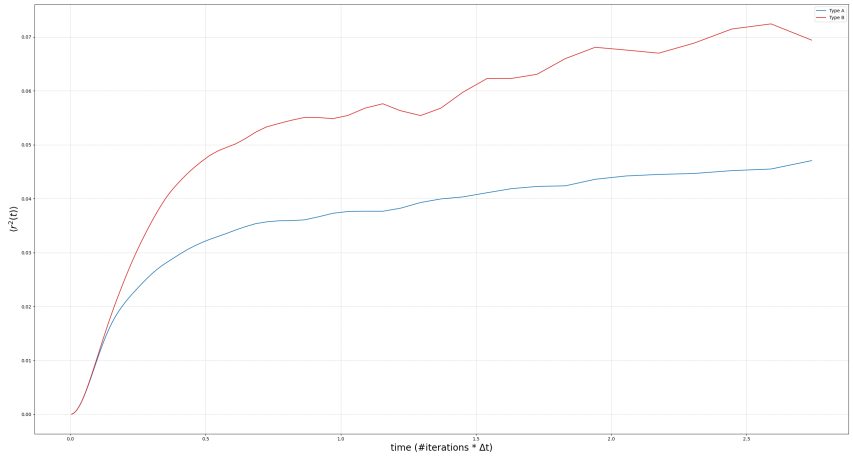


Figure 2.9. Mean squared displacement for a configuration starting at T = 0.47, P = 2.24. The time is expressed in Lennard-Jones units, scale is linear. The smaller B type particles reach a higher plateu.

where $n = \frac{t_m}{\Delta t}$. LAMMPS has a specific function to do just that

```
variable tmsd equal logfreq3(1,200,100000)
```

in our simulation we saved 200 values of the mean squared displacement from time 1 to time 100000 separated by the multiplicative ratio defined above and the result for longer times is shown in fig. 2.10. We performed 8000 simulations for seven starting configurations, see chapter 4 for more details on the dataset we used to train the Neural Network. For each one of them we calculated the Mean Squared Displacement in the NVE ensemble. In 2.11 we plotted the probability density function for three of them at the estimated relaxation time τ_g. A full plot with all the seven configurations can be seen in fig. 2.12. A somewhat raw animation of the probability distribution function evolution for four configurations during the whole experiment can be found at [32].

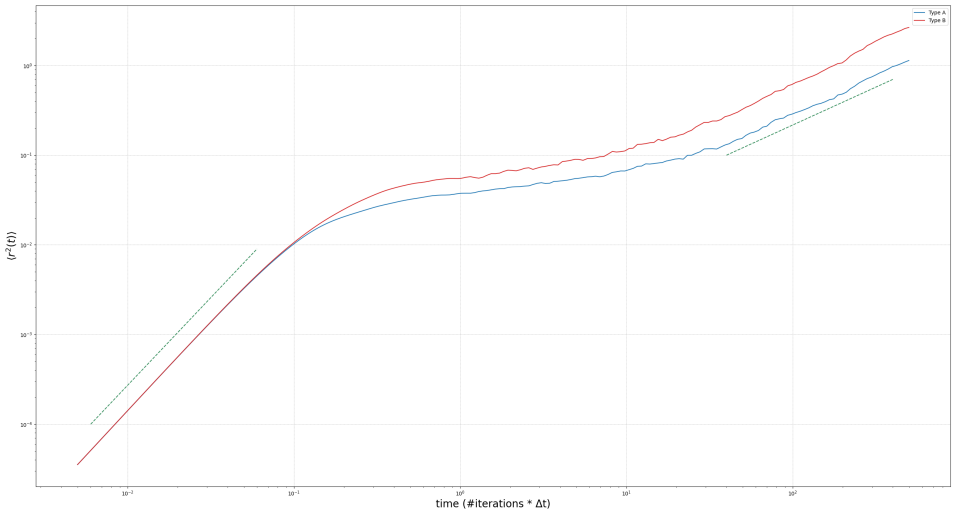


Figure 2.10. Mean squared displacement for a configuration starting at T = 0.47, P = 2.24. The time is expressed in Lennard-Jones units. Scale is double logarithmic. The two regimes with slopes 1 and 2 are highlighted by two green lines, see chapter 1 for more details.

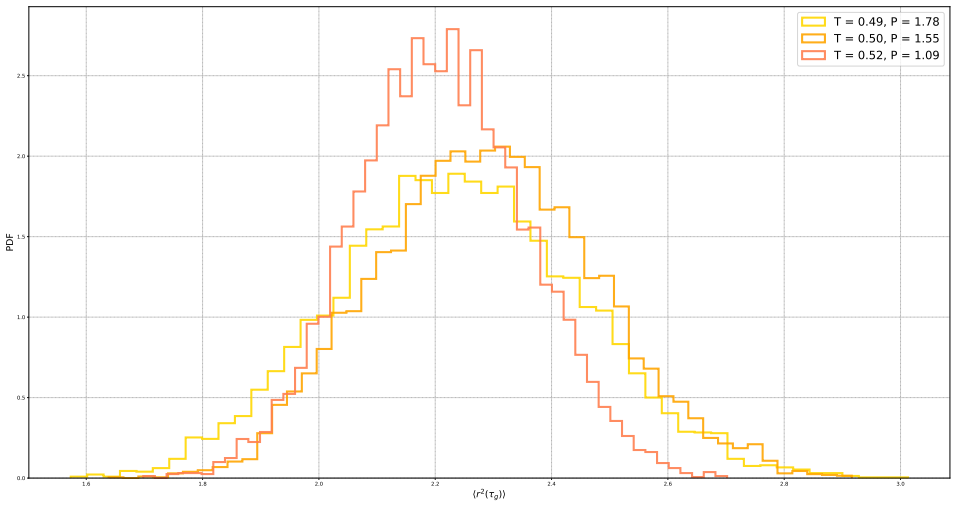


Figure 2.11. MSD probability density distributions for three configurations at temperatures $T = 0.49$, $T = 0.50$, $T = 0.52$ shown at the estimated value of τ_g.

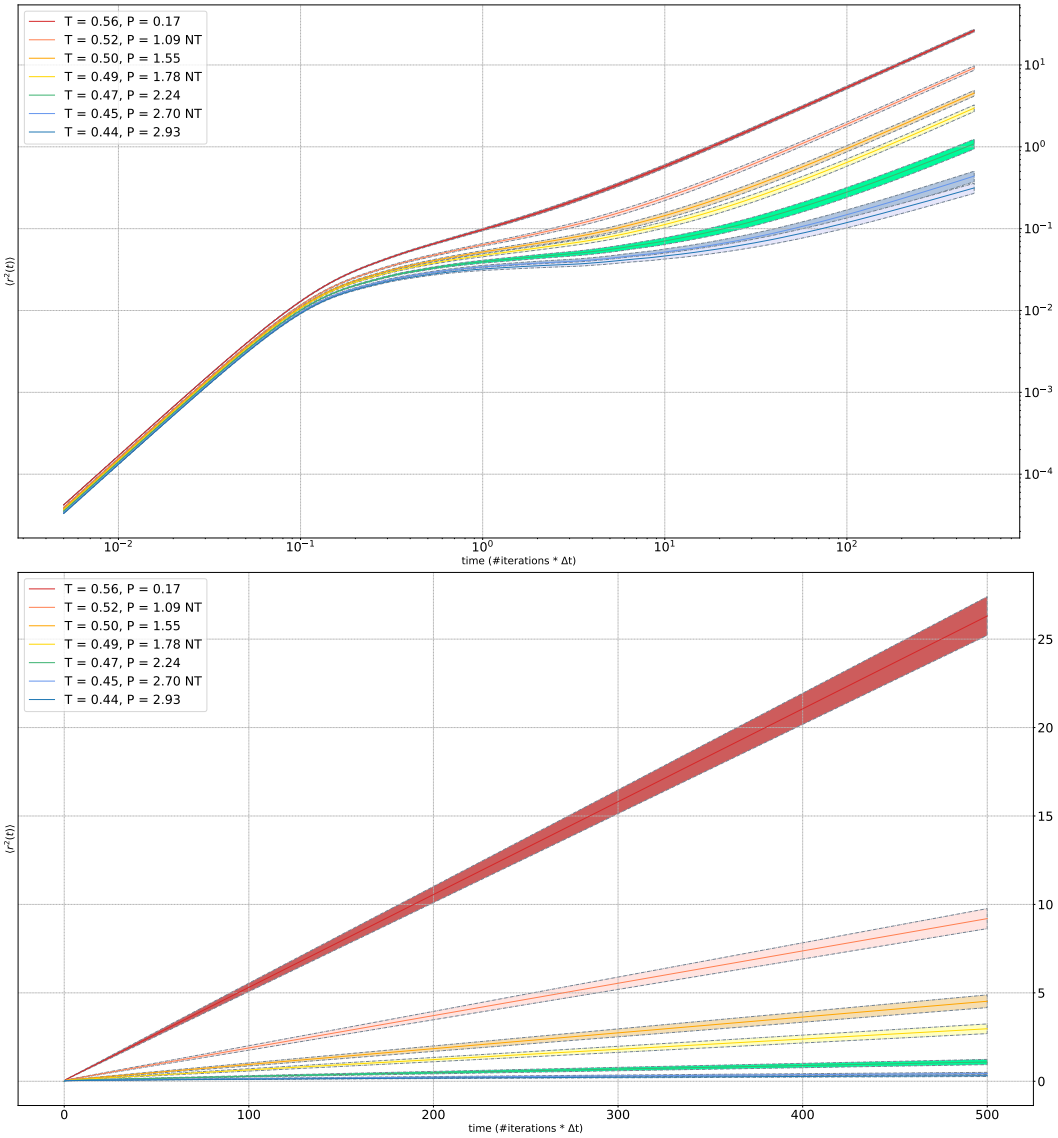


Figure 2.12. MSD curves for all seven configurations. The light-colored bands include ≈ 96% of values. Scale is double logarithmic for the top one and double linear for the bottom one. The plot in linear scale is useful to visualize the much higher thermal noise of the highest temperature configurations.

Chapter 3

A short introduction to Machine Learning and Graph Neural Networks

This chapter is a review of some of machine learning main themes. Some of the key concepts like perceptron, neural networks, backpropagation and the optimization algorithm known as Gradient Descent are outlined. In the second part we focus on Graph Neural Networks and their "Dynamic" version, the system we used in our study.

3.1 Machine Learning in a nutshell

Machine learning is an umbrella term that essentially refers to automatic adaptive models of data analysis whose performance "gets better" with "experience". As often happens in science, getting better means to make better predictions while, in this context, experience means being exposed to a lot of data. The key part is that all this process happens automatically, there should not be a specially crafted algorithm for the particular task we want to perform[1]. Having set a goal, there are often a lot of ways to try to achieve it and Machine Learning is no exception. The data on which the algorithm has to build experience on can present different levels of feedback to the system, we can make a broad division:

Supervised learning The simplest and widely adopt method. We want to match a set of input data with a set of preferred outputs. A large amount of examples, often called labeled training dataset, are provided by the human to the algorithm. With this provided dataset the system has to learn by fitting some internal parameters to correctly classify another, not labeled, dataset. Supervised learning can be used to simply classify an item (dog or not dog) or, using mostly standard regression analysis techniques, to understand and quantify the relationship between two or more variables.

Unsupervised learning Same as above, but the input data are not labeled. The system has to build a structure of the data on its own because it is difficult to provide labeled examples. Practical mundane examples are trying to infer patterns in stock data or consumer preferences etc.

After this rough classification one can be curios on some of the practical approaches used. Nowadays the machine learning field of study is vast and bustling with activity, so there are a lot of methods of implementing these ideas. Instead of a brief review of each one of them it is maybe better to examine in greater detail the method we used in our study, specifically *Neural Networks*.

3.1.1 What is a Neural Network?

The most general way we can think of a neural network is a system made of small components that map an input into an output of a desired form. Its being made of many simple parts (like neurons) and its association with learning, easily explains the adjective "neural", see fig. 3.1. They are generic structures that can be used in many different machine learning algorithms. Let's start by giving a brief outline of what a neuron is in this context. The simplest form of neuron is

[1] Not that there's anything wrong with that, it is just not machine learning.

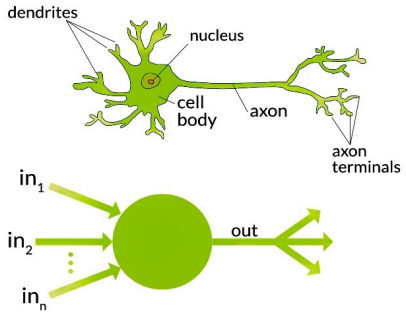


Figure 3.1. Neuron-Perceptron analogy, maybe more visual than functional.

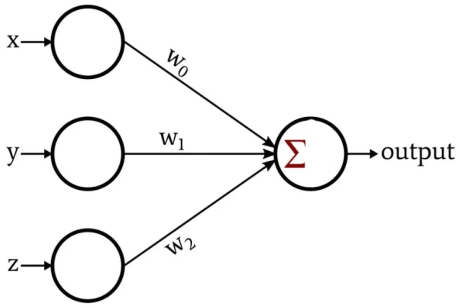


Figure 3.2. A perceptron with its three inputs, the corresponding weights and its output.

a *perceptron*. This structure, pioneered at least conceptually by Frank Rosenblatt, [38] is basically a binary evaluation of a linear combination of values, in our context input and weights. In its simplest form the inputs can only be 0 or 1. The output O of a perceptron is evaluated against a threshold value t as

$$O = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq t \\ 1 & \text{if } \sum_j w_j x_j > t \end{cases} \tag{3.1}$$

Given a set of inputs this system outputs a 0 or a 1[2] depending on the values of the weights and the threshold. By varying w_j and t we can define different models of decision to inputs x_j, see fig. 3.2. A larger value of the weight w_j means that its corresponding input x_j matters more than another input with a smaller associated weight. The threshold t is a synthetic indication on how strict the perceptron should be: a lower value means that the perceptron will be activated[3] more often. What happens when we combine a lot of these elemental structures, maybe stacked in *layers* that follow one another? We got a system like that in fig. 3.3, and this is the most basic form of a *Neural Network*. As we can see, every input is connected to every perceptron in the first layer[4] and every node of the first layer is connected to every node on the second one. To avoid ambiguity: a perceptron has always a single output, the multiple arrows coming out from the basic unit express the fact that the single output is used as input of more than one perceptron belonging to

[2] Firing or not firing, in a more biological context.
[3] Have 1 as output.
[4] The first column of circles.

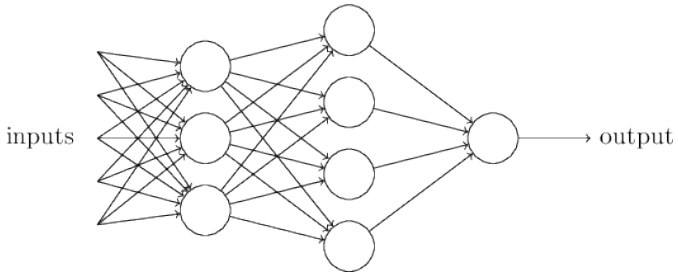


Figure 3.3. The simplest form of neural network made by two perceptrons layers.

Table 3.1. Truth table for the NAND operator.

Input 1	Input 2	Output
0	0	1
0	1	1
1	0	1
1	1	0

the next layer. The usual way of describing perceptrons' decision model is slightly different from the one used until now, so we rewrite 3.1 as:

$$O = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases} \tag{3.2}$$

where we switched to the implicit dot product notation $w \cdot x = \sum_j w_j x_j$ and introduced the bias term b. The larger the bias the easier is for the perceptron to output a 1[5]. These systems are versatile structures, let's take for example a two input (x_1 and x_2) perceptron with two equal weights $w_1 = w_2 = -1$ and a bias $b = +1.5$. The explicit calculation for input combination $x_1 = x_2 = 0$ produces 1 as output since $\underbrace{0}_{x_1} \times \underbrace{(-1)}_{w_1} + \underbrace{0}_{x_2} \times \underbrace{(-1)}_{w_2} + \underbrace{1.5}_{\text{bias}} = 1.5 > 0$. One can verify that for the other three combinations of binary inputs this perceptrons produces the same outputs as a NAND gate, tab. 3.1.

So a perceptron with those weights and bias is equivalent to a NAND gate. This is something because we know that the NAND port is universal: any boolean function can be made using only NAND gates[6], see [40]. But what's the point of using perceptrons instead of NAND gates? If we could change weights and biases in response to different inputs we could have a much more flexible network than a static circuit designed to solve a specific problem[7].

3.1.2 Too much linearity: the sigmoid neuron

If we want to train the network to do any not trivial learning task we probably have to be subtle, we need a larger "resolving power" and be able to fine tune the

[5] It is sometimes useful to remind that the bias can take negative values, too.
[6] We could theoretically build a CPU only with perceptrons.
[7] We could just flip this thing over and think of NAND gates as a subset of perceptrons.

Figure 3.4. A very small subset of the MNIST database.

parameters according to slightly different inputs. Nowadays the most used dataset for studying machine learning is probably the MNIST [41] database. Without going into details, it is a large collection of handwritten single digit numbers. Suppose we want to train a network to identify those digits, for example associate each of the handwritten numbers in fig. 3.4 with the characters 0, 1, 2, ..., 9 and suppose our perceptrons-based network was constantly identifying a handwritten eight as a nine. The characters for "8" and "9" are not that different, especially in their handwritten form, so it is easy to imagine that, for a different choice of parameters, our network could perform better.

The issue is with the binary nature of the perceptron: it can just output a 0 or a 1. A small change in the parameters in a small neighborhood of the decision threshold can drastically change the outcome: it can flip from 0 to 1 or vice-versa. Notice that this means that with the new set of parameters the 9 can be correctly classified, but other digits can be completely off focus. The network behavior of even a simplified network like the one in fig. 3.3 is hard to control when every constituent element can only output 0 or 1. We need a way to introduce small nuances in outputs according to small difference in inputs[8]. One way to do that is to introduce some nonlinear function in our perceptron decision model. Let's extend our perceptron in two ways:

- All the real values in the interval $[0, 1]$ are valid as inputs
- The output will be of this form $\sigma(w \cdot x + b)$

Where

$$\sigma(w_1, \ldots, w_n; x_1, \ldots, x_n; b) = \frac{1}{1 + e^{-\sum_j w_j x_j - b}}. \tag{3.3}$$

For a generic sigmoid function $\sigma(x)$ fig. 3.5, we have that $\lim_{x\to-\infty} \sigma(x) = 0$ and $\lim_{x\to+\infty} \sigma(x) = 1$, so for high input values this is basically a perceptron. Only for the values close to zero the behavior diverges[9]. To be more precise we have that

$$H(x) = \lim_{k\to\infty} \frac{1}{1 + e^{-2kx}} \tag{3.4}$$

[8] Perhaps is even more human not to want to deal with any sort of "chaotic" behavior.
[9] Notice that $\sigma(x) = \frac{1}{1+e^{-x}} = \frac{1}{2} + \frac{1}{2}\tanh\left(\frac{x}{2}\right)$.

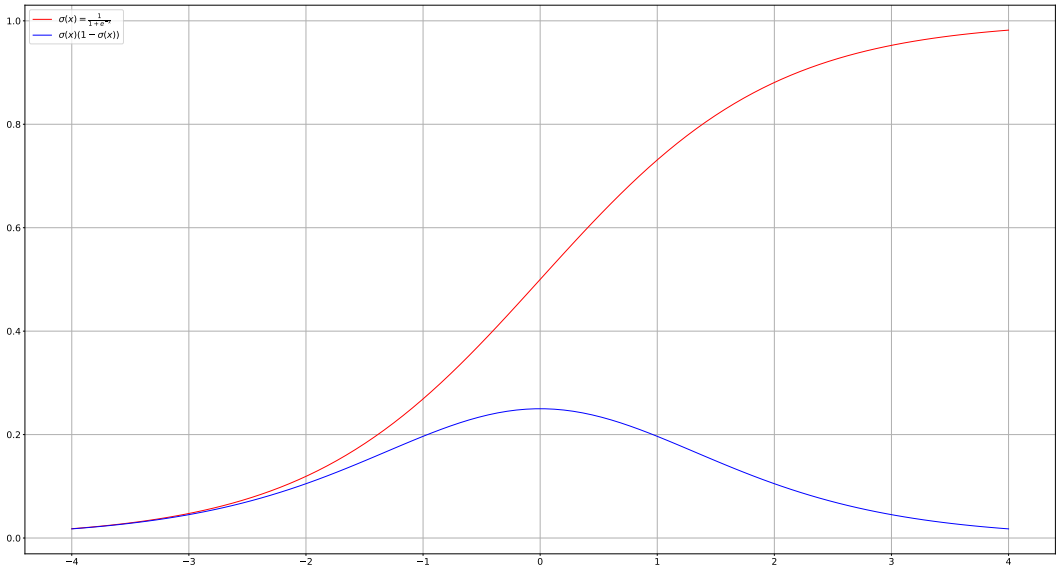


Figure 3.5. A sigmoid and its derivative.

where $H(x)$ is the Heaviside step function, the limit means pointwise convergence and the usual conventional choice of $H(0) = \frac{1}{2}$ is used. The main result of this changes is that now we got an output which depends on the weights and the bias via a smooth function, we can take derivatives and apply the usual approximations we always use in physics such as:

$$\Delta O(w_i, \dots w_j, b) \approx \sum_j \frac{\partial O}{\partial w_j} \Delta w_j + \frac{\partial O}{\partial b} \Delta b \tag{3.5}$$

This should make a bit easier to tune weights and biases to achieve a desired small output. The sigmoid function we used is known as an *activation function* and is not the only choice possible. But why should someone use a different function if the sigmoid maps the interval $[0, 1]$ so well and its derivatives are everywhere defined?[10] The first issue is a computational one: calculating a lot of exponential functions takes its toll on efficiency. Albeit it is true that $\frac{\mathrm{d}\sigma(x)}{\mathrm{d}x}$ is everywhere defined and is never zero, it is in exponential form too[11] so it could be difficult to compute as its primitive. But there is another problem, too: it quickly goes to zero for large values of x and this can cause problems with the backpropagation, more on this later.

So a sigmoid neuron can have as output any real number in $[0, 1]$ instead of just $\{0, 1\}$, how do we interpret this behavior? This is useful for meanings like brightness intensity of a pixel in image recognition tasks and is more flexible in general, but if we want a binary answer (dog or not dog) we must set a threshold value in output e.g. $O \geq 0.5$ is a dog.

[10] It is more than that: it is real analytic, to be precise.

[11] $\frac{\mathrm{d}\sigma(x)}{\mathrm{d}x} = \sigma(x)(1 - \sigma(x))$.

The softmax function

The sigmoid function is a scalar function, one possible generalization is the *softmax* function defined as:

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{N} e^{z_j}}$$

where $\mathbf{z}$ is a vector with N components. The softmax function applied to a vector $\mathbf{z}$ gives as output another vector whose components' values are all between 0 and 1 and sum up to 1. In that way they can be interpreted as probabilities. This function is very useful in mutually exclusive classifiers where every component is interpreted as category[12], and it is usually used in the last layer of a neural network to normalize the outputs. The sigmoid function can be retrieved from the softmax by giving as input a two dimensional vector whose second component is zero: $\mathbf{z} = [x, 0]$. Calculating the softmax output for this vector we get as first component:

$$\sigma\left(\mathbf{z}\right)_1 = \frac{e^x}{e^x + e^0} = \frac{e^x}{e^x + 1}$$

Dividing the numerator and denominator by e^x we get:

$$\sigma\left(\mathbf{z}\right)_1 = \frac{1}{1 + e^{-x}}$$

which is our sigmoid function.[13]

3.1.3 Architecture of a simple neural network

There are two other main ingredients necessary to understand a simple model of a Neural Network, the first one is the *backpropagation*.

Backpropagation

Let's set ourselves in an simple context, imagine we have to train a simple supervised network with two layers like the one in fig. 3.6. The process of training starts by providing the system a pair $(\mathbf{x}, \mathbf{y})$. $\mathbf{x}$ is the input vector and $\mathbf{y}$ is the corresponding label vector. The two layers and their outputs are represented by $\mathbf{h}^{(1)}$ and $\mathbf{h}^{(2)}$, $\hat{\mathbf{y}}$ is the final output of our network.[14] The C letter at the end represents the *loss function* which compares the output $\hat{\mathbf{y}}$ to the labeled data $\mathbf{y}$ and tells us how well the whole thing is working, so it should be explicitly written as $C(\mathbf{y}, \hat{\mathbf{y}})$. The main goal of the training is to reduce this function so that, after the process, if we feed the network a training set it should result in a small loss function. We have to slowly adjust the weights and the bias in order to minimize this function

[12] For example in an image classifier the results could be stored in a vector and interpreted as 65% dog, 20% cat, etc.

[13] The second component is simply $1 - \sigma\left(\mathbf{z}\right)_1$.

[14] Notice that the information flow is only forward-directed, for example there is no arrow going back from the neuron N to the neuron $N - 1$: this is called, without great creativity a *feedforward* network.

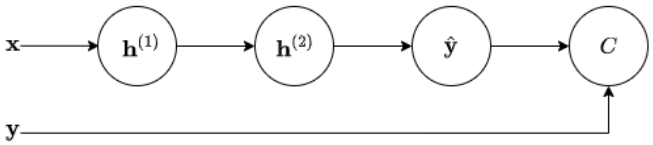


Figure 3.6. Schematic representation of a neural network with two layers of neurons.

and minimize rhymes with derivatives.[15] At the start of the training C will be large because our parameters will have, let's suppose, random values. We must calculate

$$\frac{\partial C}{\partial W_{jk}^{(i)}} \tag{3.6}$$

where $W_{jk}^{(i)}$ is the network weight from the node j in the layer $i-1$ to node k in the layer i to achieve our goal. There are many ways to choose a loss function, by exploiting different properties of mathematical objects[16] we can determine how similar are two vectors, but our choice for this example will be the good simple mean squared error.

$$C(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{2}(\mathbf{y} - \hat{\mathbf{y}})^2 \tag{3.7}$$

According to what we have already written in the previous section, the output of a neuron h of a feedforward network will be in this form

$$h = \sigma(w \cdot x + b) \tag{3.8}$$

Where

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

is our sigmoid function. Also remember this interesting property of its derivatives:

$$\frac{\mathrm{d}\sigma(x)}{\mathrm{d}x} = \sigma(x)(1 - \sigma(x)).$$

Let's get back to fig. 3.6 and for the sake of simplicity let's set the bias vector to zero and imagine we have not layers of neurons but just neurons. [17] We have

$$h^{(1)} = \sigma(W^{(1)}x)$$
$$h^{(2)} = \sigma(W^{(2)}h^{(1)})$$

As the outputs of the first and second neurons, and

$$\hat{\mathrm{y}} = W^{(3)}h^{(2)}.$$

[15] Not actually.

[16] Cross entropy, cosine similarity functions, etc.

[17] To deal with scalars and not vectors and to make us focus on the main idea of backpropagation. Sometimes we will maybe refer to layers but the context should not create any ambiguity.

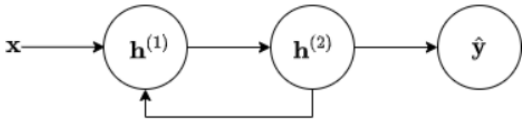


Figure 3.7. The simplest form of Recurrent Neural Network.

as the final output of the network. Putting this last result in 3.7 we get:

$$\begin{aligned} C(y, \hat{y}) &= C(y, W^{(3)} h^{(2)}) \\ &= C(y, W^{(3)} \sigma(W^{(2)} h^{(1)})) \\ &= C(y, W^{(3)} \sigma(W^{(2)} \sigma(W^{(1)} x))) \end{aligned}$$

Let's calculate some gradient going backwards from the last neuron to the first, apply the chain rule when possible and see what happens. For the last layer we have:

$$\begin{aligned} \frac{\partial C}{\partial W^{(3)}} &= \frac{\partial C}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial W^{(3)}} \\ &= (\hat{y} - y) h^{(2)} \end{aligned}$$

For the one in the middle:

$$\begin{aligned} \frac{\partial C}{\partial W^{(2)}} &= \frac{\partial C}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h^{(2)}} \frac{\partial h^{(2)}}{\partial W^{(2)}} \\ &= (\hat{y} - y) h^{(2)} W^{(3)} h^{(1)} \sigma(W^{(2)} h^{(1)})(1 - \sigma(W^{(2)} h^{(1)})) \end{aligned}$$

And finally we can derivate with respect to the weight in layer 1:

$$\frac{\partial C}{\partial W^{(1)}} = \frac{\partial C}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h^{(2)}} \frac{\partial h^{(2)}}{\partial h^{(1)}} \frac{\partial h^{(1)}}{\partial W^{(1)}}$$

As highlighted by the colors, there are some terms in common between those expressions. What does this mean? Simply that it is not necessary to calculate the entire expressions at every layer: we can recycle the common terms. With this nice trick[18] probably devised by Seppo Linnainmaa[43], we are able to compute all the partial derivatives in linear time of the graph size. If instead we decided naively to calculate all those terms layer by layer, the number of operations would scale exponentially with the depth, and these networks are deep. Let's also notice that the backpropagation works only for feedforward networks: we have written every layer as a function of the previous ones, we have essentially to work with networks that can be written as direct acyclic graphs. For example other kinds of structures, known as *Recurrent Neural Network*, fig. 3.7, where the input of a neuron is fed back to a previous one at a following time, are harder to adapt to our trick.[19]

[18] In fact a clever implementation of the chain rule.

[19] A Recurrent Neural Network is maybe more similar to a human brain structure and can be rewritten as a series of feedforward networks.

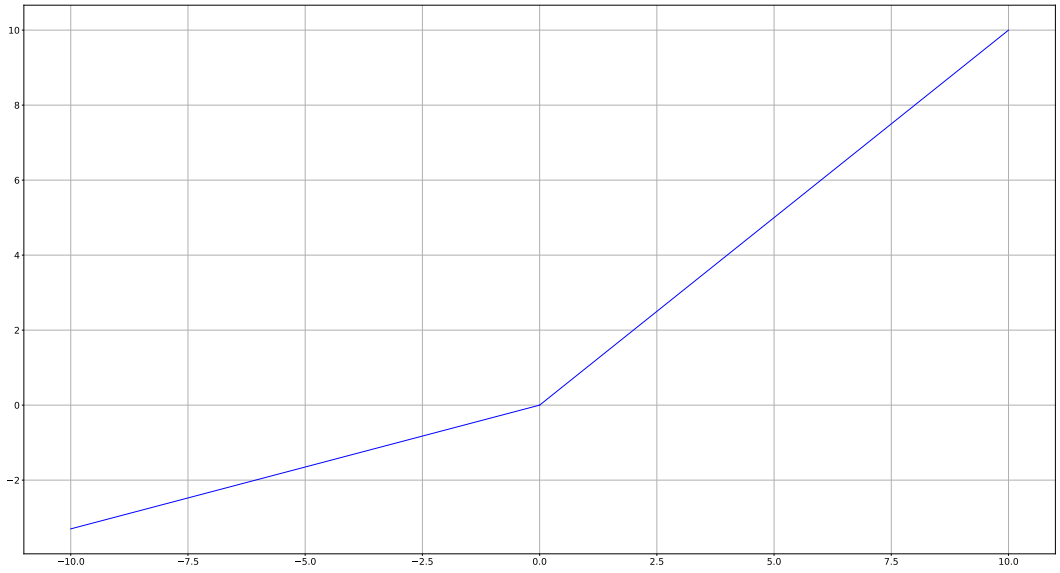


Figure 3.8. Parametric ReLU with $\alpha = 0.3$.

Vanishing gradient problem: What happens if the gradient of the loss function gets very small? In our example we had a shallow network with only three layers, but real networks are deeper. If n small derivatives are multiplied, the result is even smaller, this implies a slow updating of the weights and the biases spoiling the performance of the whole network. This phenomenon is known as the *Vanishing gradient problem* and its severity depends on the chosen activation function.

One way to mitigate the problem is indeed to choose as activation function the rectified linear unit, in its general form called the *Parametric rectified linear unit*, PReLU:

$$f(\alpha, x) = \begin{cases} x, & \text{if } x \geq 0 \\ \alpha x, & \text{if } x < 0 \end{cases} \tag{3.9}$$

where $\alpha \in (0, 1)$. The critical case is when $\alpha = 0$[20] because the derivative would not be defined in zero, this can be avoided by choosing different values. This function is not bounded as the sigmoid, it does not squeeze the input space into the small output interval $(0, 1)$ hence it doesn't suffer of the small derivative problem. Backpropagation is not a learning algorithm, it is just an ingenious trick. After having calculated all the derivatives we need, how do we update the weights, and the parameters in general, in order to reduce the loss function?

Learning algorithms

Let's start saying that "learning" in this context basically means optimizing the loss function, looking for a (local) minimum. There is a pretty simple old [44] procedure to try to follow. Let $f\colon A \subseteq \mathbb{R}^N \to \mathbb{R}$ be a function defined and differentiable in neighborhood of a point $\theta \in A$. If one wants to look for the fastest way of decreasing f starting from θ he or she has to move in the negative gradient

[20] This is the standard simpler form of ReLU.

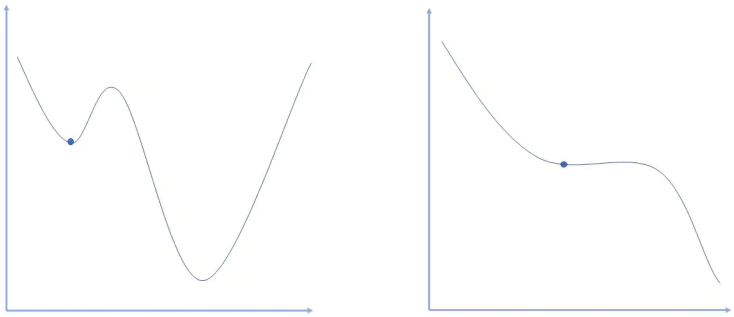


Figure 3.9. **Left**: the blue point reached a local minimum but there is a lower one on the right. **Right**: the blue point is on a saddle point with a local minimum on the left and a local maximum on the right.

direction: $-\nabla f(\theta)$. This is a fact of life, remember that the gradient is a vector whose components are the directional derivatives of the function along each axis.

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \dots, \frac{\partial f}{\partial x_n} \right).$$

The main idea behind the gradient descent procedure is to update our position from θ_n to θ_{n+1} like that

$$\theta_{n+1} \leftarrow \theta_n - \lambda f(\theta_n) \tag{3.10}$$

hoping that $f(\theta_{n+1}) \leq f(\theta_n)$. The quantity λ is known as the *learning rate* and is an example of hyperparameter: an external parameter that we can set at the beginning of every training iteration and affects the whole process. The learning rate has to be chosen heuristically.[21]. It sets the size of the steps taken to go towards the minimum at each iteration. Picking a value too low will impact the performance of the network: we are making too small changes. But picking a value too large can quickly converge to a local minimum which could be higher than the global one, see fig. 3.9. Or it can just simply miss it, see 3.10.[22]

Keep also in mind that to fully compute the gradient we need to calculate it for every component of the whole input vector and this can be another factor that slows the process.

We can try to mitigate this issue by applying a procedure known as *Stochastic Gradient Descent*. This is a way of both trying to find the global minimum and to speed up the computation of the gradient with an estimation. The idea is simple: instead of computing the gradient for the whole input vector $\mathbf{x}$, let's divide it in m parts and choose one of them. Then we calculate an approximate version $\widetilde{\nabla}$ for this small subset of $\mathbf{x}$. Let's take for example $n = 3$ elements.

$$\widetilde{\nabla} f = \nabla f_\theta(x_1) + f_\theta(x_2) + f_\theta(x_3)$$

and use this approximation in 3.10. We get

$$\theta_{n+1} \leftarrow \theta_n - \lambda \widetilde{\nabla}(\theta_n)$$

[21] It is usually in the interval $(0, 1]$.

[22] Under certain conditions which includes f convex and ∇f Lipschitz and some others on the learning rate itself the convergence to a local minimum is certain.

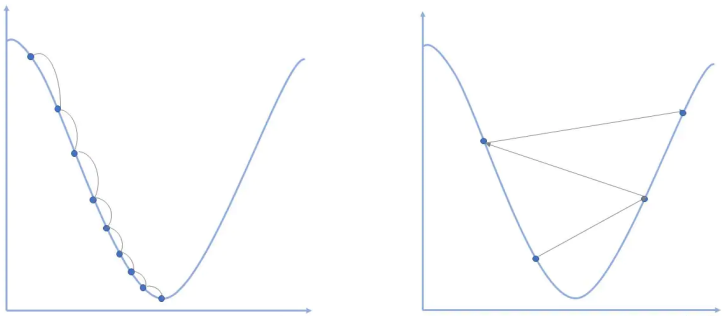


Figure 3.10. On the **x** axis the weight values, on the **y** axis the loss function. **Left**: an example of low learning rate slowly reaching the minimum. **Right**: a high learning rate completely missing the minimum.

With this approach the CPU time used to compute the gradient is greatly reduced, so we can do a lot more iterations and maybe surprisingly have better results than the standard gradient descent. One reason may be that the extra noise introduced by the reduced sample can help escaping saddle points and local minimums [49]. Another explanation of the unreasonable efficacy of the Stochastic Gradient descent can lie in the following calculations:

$$\theta_{n+1} = \theta_n - \lambda \nabla f(\theta_n) - \underbrace{\lambda \epsilon_t}_{noise}$$
$$y_n = \theta_n - \lambda \nabla f(\theta_n)$$

Where y_n is just a change of variables, it follows that:

$$y_{n+1} = y_n - \lambda \epsilon_t - \lambda \nabla f(\underbrace{y_n - \lambda \epsilon_t}_{\theta_n})$$

Taking the expected value on both sides with respect to the noise,[23] we get:

$$\mathbb{E}[y_{n+1}] = y_n - \lambda \nabla \mathbb{E}\left[f(y_n - \lambda \epsilon_t)\right]$$

This equation looks like 3.10 and means that Stochastic Gradient Descent can be interpreted as standard gradient descent with respect to a version of f smoothed by the mean.

3.1.4 Some complications and how to soften them

Before we start examining the network architecture we used in our study, it is necessary to write a few words about two issues that are inherently present in the Neural Network, but luckily we have ways to try to mitigate them. These are not the only one, but are the ones directly addressed by the ParticleNet architecture, which we will explain better in the next section.

[23] Whose expected value is zero.

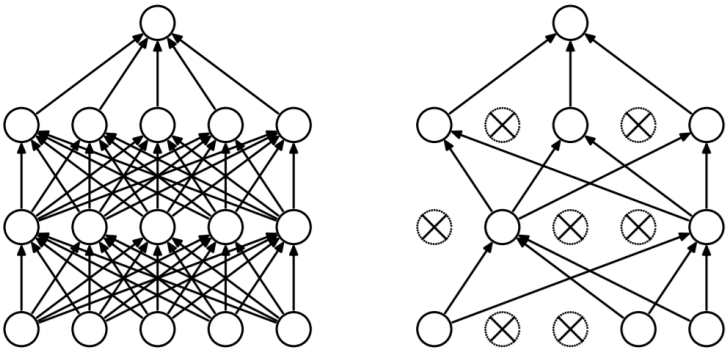


Figure 3.11. Left: a normal neural network. **Right**: the same network after applying the dropout technique.

Overfitting: Large Networks are always at risk of overfitting, by overfitting we mean that the net has learned noise in the training data and that noise has been treated as a property of the system which is obviously not by definition. This can happen when there are a lot of parameters compared to the sample size features and can lead to poor generalization performance.[24] In a neural network a sign of this issue is when the error on the training sample steadily decreases iteration after iteration, but not so the error on new samples on which the network has not trained, the *validation sample*. A way to ease this problem was proposed in [45]: the *dropout* method. During the training phase, some outputs are randomly discarded, see fig. 3.11, each update is performed with a different view of the layers so the neurons do not develop a co-dependency between them which could reduce the predictive power of the single unit. The drop out method is a clever way to use randomness in a different place to prevent a problem caused by the treatment of random noise in input. Applying the dropout technique increases the number of iterations for a convergence of the loss function to a stable value but each iteration requires less time because of the lowered number of neurons. Just for curiosity: if we have n neurons and each of them can be discarded or not at each iteration we have 2^n possible networks' structures.

Batch regularization: When training a deep neural network the process of updating proceeds backwards from the output to the input traversing each layer. The problem is that this process of updating a layer via gradient descent assumes that the weights in other layers are constant. In other words: the weights of a layer are updated according to an expectation that the previous layer outputs value with a certain distribution, but that distribution has probably changed after the weights updating process of the prior layers. This change of inputs' distribution during the training has been introduced in [48] and has been called by the authors Internal Covariate Shift. What to do? A way to regularize the input across the neurons is to apply a good old normalization-to-Gaussian procedure. A Batch regularization step applied on a layer would first calculate the mean and the variance of the signal

[24] Quoting Von Neumann "With four parameters I can fit an elephant and with five I can make him wiggle his trunk". Someone has interpreted him to the letter, see [42].

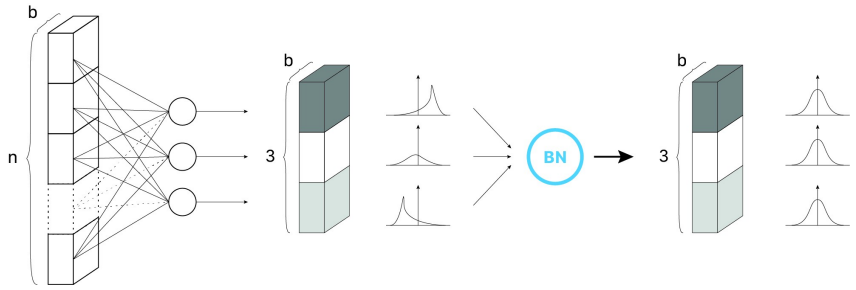


Figure 3.12. A visual schematic of the batch normalization process.

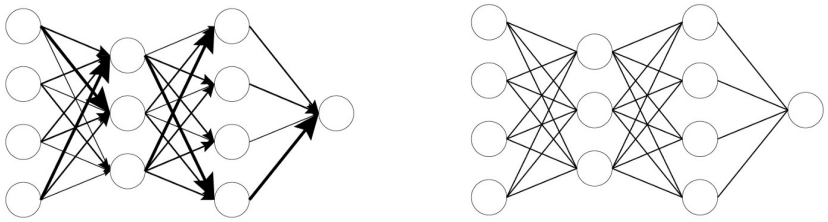


Figure 3.13. Left: a not normalized neural network, the high level of codependency between neurons is highlighted by the wider arrows. **Right:** a normalized network, no interdependence.

for each neuron S^i:

$$\mu = \frac{1}{n}\sum_i S^{(i)}, \qquad \sigma^2 = \frac{1}{n}\sum_i \left(S^{(i)} - \mu\right)^2$$

than normalize the output vector with

$$S_n^{(i)} = \frac{S^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

where ϵ is a small constant used to avoid numerical divergence issues. The result is visible in fig. 3.12.

Note that this batch normalization process can be thought as a way to reduce interdependence between neurons, too, see fig. 3.13.

3.2 [Dynamic Graph]Convolutional Neural Networks

3.2.1 Convolutional Neural Networks

A particular kind of neural network that has achieved a lot of success in computer vision field is the *Convolutional Neural Network*, CNN. Our intention is surely not to write an exhaustive description of them, we just want to provide a glimpse of the main idea behind them. In our study we applied a convolution-like operation on a graph, so it is appropriate to summarize shortly some of their features in their most famous application. In functional analysis convolution between two functions

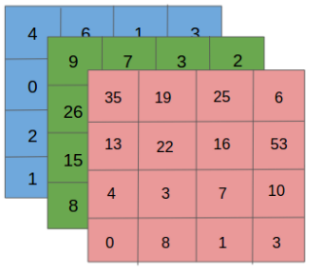


Figure 3.14. A representation of an image with three color channels, it would be stored as a $4 \times 4 \times 3$ array.

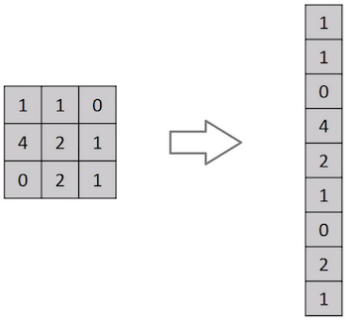


Figure 3.15. A 3x3 matrix reshaped into a 9x1 vector.

f and g is defined as:

$$(f \star g)(x) = \int_{\mathbb{R}} f(y)g(x-y)dy \tag{3.11}$$

Now consider the translation operator defined as $T_y g(x) = g(x-y)$. We can image to write a convolution operator with f like that:

$$C_f = \int_{\mathbb{R}} f(y)T_y dy \tag{3.12}$$

Maybe it is not so rigorous but it makes us grasp the main idea: convolution is an infinite linear combination of translation operators, a finite version would have a sum instead of an integral. This relation between translation and convolution is useful to explain the name of the system we are talking about. Let's imagine we want to analyze a picture to recognize objects in it, or just to differentiate the background from the foreground. For a computer an image is just a tensor[25] whose values represents color intensities for the three different channels, see fig. 3.14. We could think to reshape the matrix into a vector and feed every value to a Neural Network neuron like the one described in the previous section. This could be done, we could also try to take the average value of the intensities between the three color channels, but there would be two main problems:

- The standard Neural Network like the one in fig. 3.16 is fully connected: every neuron is connected to every other in the layers adjacent to the one it belongs.

[25] In this context a multidimensional matrix, also called an array.

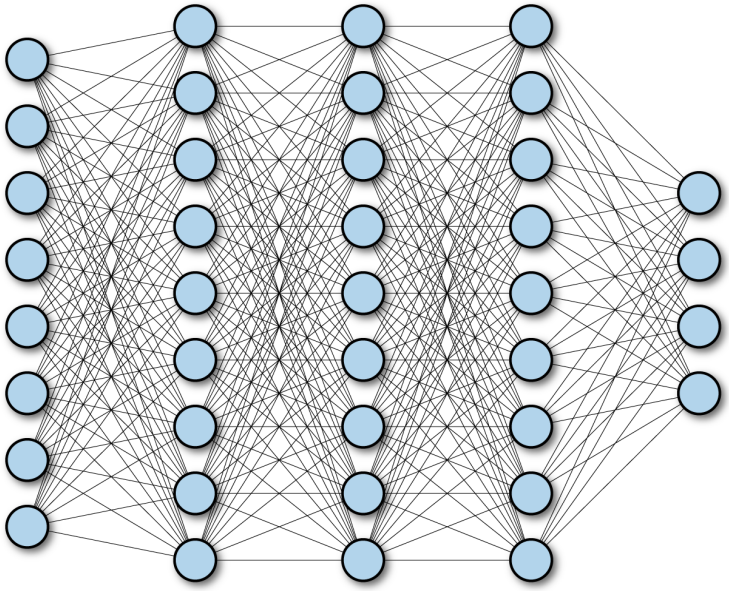


Figure 3.16. A fully connected neural network.

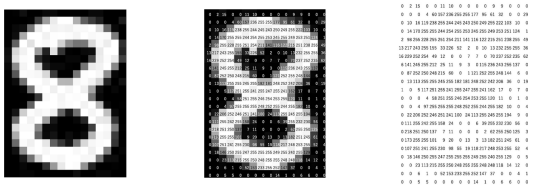


Figure 3.17. **Left**: a $16 \times 22 \times 1$ greyscale image of the eight digit from the MNIST database. **Center**: same as right but with superimposed the stored intensities values. **Right**: the intensities values.

This means that every pixel in the image is treated the same, there would be no spatial relationship between them: this is not a smart move to do if we want to extract shapes and more general features. In a picture of a cat and a tree, two pixels belonging to the cat must have a stronger relationship between them than with pixels belonging to the tree[26].

- There are simply too many pixel: a 4K resolution image has roughly 8.3 millions pixels, if every one is composed by 3 color channels we clearly see the scale of the problem.

Ok, what to do then? Let's look at fig. 3.17 for a simple example. We can identify the number eight because, at the edges, there is a sharp change in intensity, but intensities are just integers. To understand if a pixel is on the edge or not we could just subtract its intensity values from the ones of its neighbors[27], maybe defining a transition threshold. Do we have to apply this procedure to every pixel? No we just have to convolve a matrix known as *kernel* or *filter* over the whole image

[26] At least that is what our brains suggests.

[27] Note that we are making use of a spatial relation between pixels.

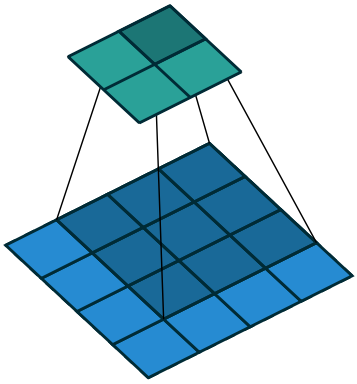


Figure 3.18. A 3×3 convolution kernel over a 4×4 input matrix, the result is a 2×2 matrix.

in the attempt of getting some useful information. This would occur for each $n \times n$ group of pixels. By convolving we mean an operation like the one in figure 3.18.

We are sliding over the whole image a small matrix in order to extract spatial features. In practical terms we are multiplying element by element the input matrix with the kernel[28] and then summing up the result. An example is in eq. 3.13.

$$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix} \odot \underbrace{\begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix}}_{Kernel} = \begin{bmatrix} 19 & 25 \\ 37 & 43 \end{bmatrix} \tag{3.13}$$

At the first iteration every red element is multiplied to the corresponding blue kernel element and summed. The result is the element in green in the matrix on the right[29].Then the kernel will slide over one column to the right, then one row down etc. until it will have covered the whole input matrix. Notice that there is some overlapping in this process. What kernels to use to extract what features? An example of a simple one is the Prewitt kernel [52]:

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

Applying this kernel to an image could detect borders in the x component, it is basically performing a difference between the adjacent elements of a pixels along the x axis.

Convolutional neural networks would not be that effective if they were limited to one layer, there are usually many layers responsible of extracting specific features. The deeper the network, the more sophisticated these filters become: rather than edges or simple shapes, they may detect objects at different scales like faces, ears,

[28] An operation sometimes known as Hadamard product indicated with $\odot$.
[29] To be more explicit: $0 \cdot 0 + 1 \cdot 1 + 3 \cdot 2 + 4 \cdot 3 = 19$.

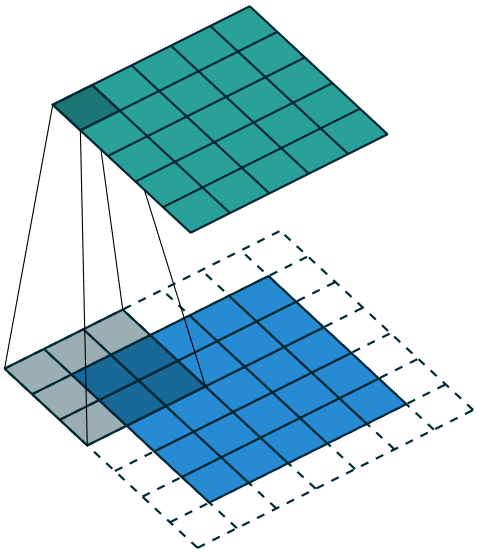


Figure 3.19. A 3×3 convolution kernel over a 5×5 input matrix padded with a white extra row and column. The result is a 5×5 matrix: the same dimensions as the one in input.

hair, etc.[30] Note that each kernel, associated with a set of weights, will slide over the whole image so the system can use all the pixels' locations to learn. Also note that in 3.13 the result is a 2×2 matrix while the input was a 3×3 one. Even in fig. 3.18 we get a similar downsampling in the output. If the desired results is an output matrix whose dimensions are the same as the input one after the convolution, we can pad the original matrix. One way of padding is framing the matrix with zeros in order to get an object of the same dimensions as the one in input, see fig. 3.19.

But there could also be other stages, called *pooling layers* whose goal is to explicitly downsample the input in order to create a "summarized" version of it. An example of that is the average pooling filter, see fig. 3.20. For every $n \times n$ region of the input matrix, the arithmetical mean is calculated and the region is replaced by the result. This achieves two goals:

- A smaller output means less computational power required, maybe less neurons in a fully connected Neural Net at a following stage.
- This procedure provides a bit of translational invariance: moving the image by a small amount of pixels does not significantly affect the values of the pooling operation.

The other commonly used pooling method is max pooling, it works like the average pooling but it takes the maximum value instead of the average one. Choosing between the two requires a bit of wisdom and heuristics. What is certain is that with a 2×2 kernel, max pooling discards $\frac{3}{4}$ of the total data and keeps $\frac{1}{4}$. Average pooling on the other hand does not discard any data and retains more information.

[30] This progress to hierarchical levels of understanding is maybe closer to how humans think.

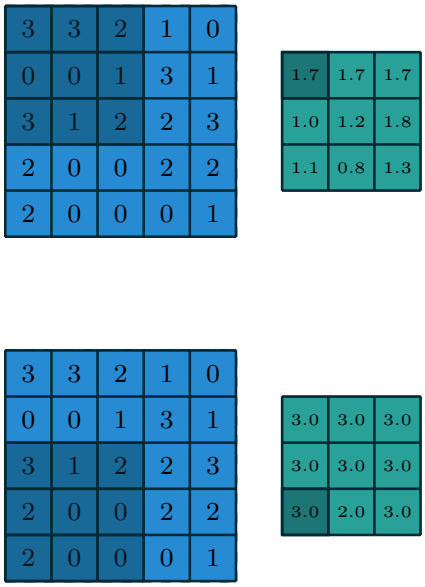


Figure 3.20. Average and Max pooling with a 3×3 filter applied on different regions of the input matrix.

Max pooling is better to highlight a feature in a block. For the MNIST dataset, with digits represented with bright pixels over black backgrounds, it works better than average pooling but in general both choices are possible.

Convolutional Neural Networks had a tremendous success in the past ten years in image classification and contributed to drawing a lot of attention on the AI field. The most famous example is AlexNet[31] [54] which won the 2012 Imagenet Large Scale Visual Recognition Challenge 2012 [55] with an error rate of 15.3%, in 2016 the error rate achieved by the winner was $\approx 5\%$. The effectiveness of these systems inspired the scientific community to extend their methods beyond their original scope and to other structures, different from images.

3.2.2 Dynamic Graph Convolutional Neural Networks

Classifying objects in images is[32] a hard task but images themselves are somehow simpler objects: they are of fixed size and the relative position of their constituent pixels is constant, like in a solid. What to do if we wanted to treat inputs that are not as well structured? One way to attack this problem is to represents those inputs like graphs. Same caveat made in the previous section: this is not a comprehensive description of the subject, we will give some basic elements necessary to understand the application of these ideas in our study.

[31] A CNN with 650.000 neurons, 60 million parameters and eight layers: five convolutional and three subsequent fully connected ones. The activation function used was mostly a ReLu and the network was trained on two GPUs for two weeks.

[32] Or was, before CNNs.

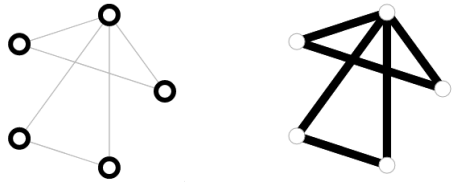


Figure 3.21. Respectively from left to right: a graph with its node highlighted and one with its edges highlighted.

Figure 3.22. Two edges of a graph, one undirected and one directed. An undirected edge is equivalent to have a directed one in both directions.

A *graph* $\mathcal{G}$ is a graphical[33] way of representing a set with two kinds of elements: a finite number n of entities (*nodes* or *vertices* $\mathcal{V}$) and relations between them (*edges* $\mathcal{E}$), see fig. 3.21. In symbols:

$$\begin{aligned}\mathcal{G} &= (\mathcal{V}, \mathcal{E}) \\ \mathcal{V} &= \{1, \dots, n\} \\ \mathcal{E} &\subseteq \mathcal{V} \times \mathcal{V}\end{aligned}$$

Should the need of establishing a source and a destination in these relations arise, we can define a directed graph, see fig. 3.22. If one looks carefully, they can be ubiquitous: a lot of systems we are already familiar with can be represented as graphs. One example in the scientific community is maybe the citations network graph: each paper is a node and each directed edge represents a citation. Other examples are molecules' structures and social networks interactions.

That said, it is not very clear how to generalize to graphs the idea after which the Convolutional Neural Networks are named. As written before, in a picture, convolving depends on the position of the pixels, while in a graph the neighbors are different from one node to another[34]. This is especially true in graphs whose nodes represents gas particles like the one used in our application.

An interesting convolution-like operator has been proposed in [33] to work on Point Clouds[35]. We follow an application of that approach described in [34]. As written above, let's start by representing the particles as nodes of a graph. Every node is connected with its k nearest neighbors and every point x_i is associated with a F-dimensional vector $\mathbf{x}_i \in \mathbb{R}^F$ describing its features. Those features usually are spatial coordinates and at least one property of a particle. Both the features and the number of nearest neighbors to consider are like hyper-parameters. The local region for a point on which apply our new form of convolution will be those k points.

[33] Typically, but not necessarily.

[34] Someone [56] tried to keep the consistency in an evolving graph but in our study we chose a different path.

[35] A *Point Cloud* is a collection of points in $\mathbb{R}^3$, usually the data necessary to make them are gathered with LiDARs or other environment scanning devices.

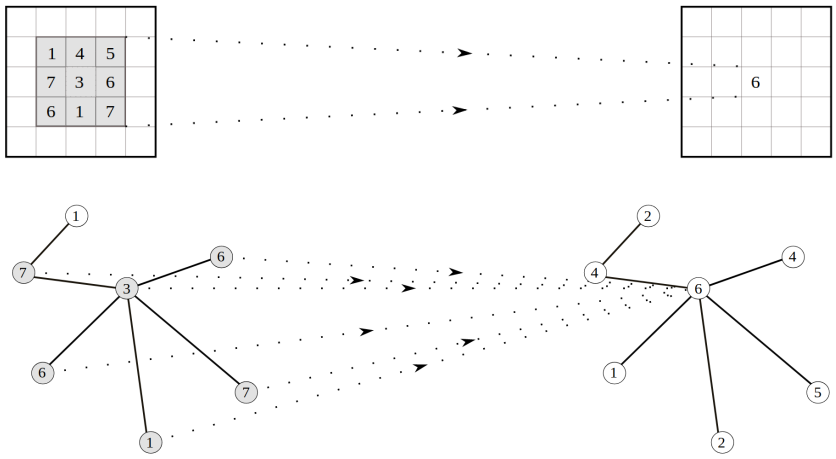


Figure 3.23. **Top**: an operation of convolution with a 3×3 kernel on the central element of a matrix. **Bottom**: A possible localized (first neighbors) convolution on a graph. The convolution of the element with the highest number of neighbor is highlighted by arrows.

Like the images of the previous section we have a point x_i which is like a central pixel and a patch of neighbors surrounding it. Our new operation, *EdgeConv*, for a point x_i is defined as

$$\mathbf{x}'_i = \square_{j=1}^{k} \mathbf{h}_{\mathbf{\Theta}}(\mathbf{x}_i, \mathbf{x}_{i_j}) \tag{3.14}$$

where $\{i_1, \ldots, i_k\}$ are the k neighbors of x_i and $\mathbf{h}_{\mathbf{\Theta}} \colon \mathbb{R}^F \times \mathbb{R}^F \to \mathbb{R}^{F'}$ is a function of the set of learnable parameters $\mathbf{\Theta}$. Note that this operation defines features specific to edges sometimes indicated as $\mathbf{e}_{ij} = h_{\mathbf{\Theta}}(\mathbf{x}_i, \mathbf{x}_j)$. $\square$ is a operation invariant to permutation of its arguments like max, sum or a kind of mean. The fact that the parameters $\mathbf{\Theta}$ are shared for all points and the symmetry of $\square$ makes the whole operation symmetric on the point cloud.[36] As is easy to image, the choice of the edge function h and the aggregation operator $\square$ is crucial. We follow [34] and set:

$$h_{\mathbf{\Theta}}(\mathbf{x}_i, \mathbf{x}_j) = \bar{h}_{\mathbf{\Theta}}(\mathbf{x}_i, \mathbf{x}_{i_j} - \mathbf{x}_i)$$

This takes into account the global structure of our point cloud by taking into consideration the centers $\mathbf{x}_i$ but also local information with $\mathbf{x}_{i_j} - \mathbf{x}_i$. In practical terms this means that the features of the neighbors $\mathbf{x}_{i_j}$ are replaced by the difference with the features of the central node. $\bar{h}_{\mathbf{\Theta}}$ can be implemented with a multilayer network. For the aggregator $\square$ in our application we chose the max function, one experiment was made with the average but with a similar result, more on in the following chapter. Maybe the most important advantage of EdgeConv, shared with the key operation of the simple Convolutional Neural Network operation, is that it

[36] To tell the truth the spatial coordinates require special care. It is true that the output of EdgeConv is independent from the order of the input but the k nearest neighbors of a particle are calculated after a coordinate system has been set, so there is some sort of "canonical ordering".

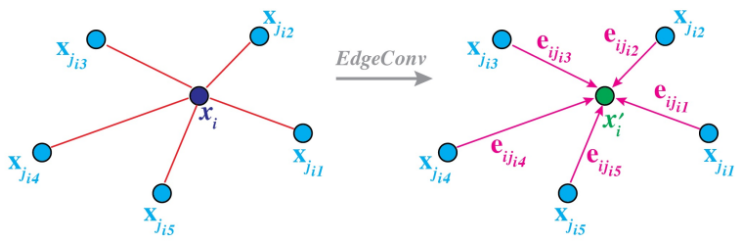


Figure 3.24. The EdgeConv operation in action. x_i is updated into x_i' and all the edge features e_{ij} are included into the computation.

is stackable. Every EdgeConv operation is a map from a point cloud to another one with the same number of points[37], so they can be applied in sequence building a feature-aware hierarchical structure like an image classifying CNN.

One final point: what does the adjective "dynamic" mean in this context? This is a clever idea explained in [33] and an important distinction of this method from other ones with fixed graph CNNs. The features learned by the system can be treated as new coordinates stored in a sort of latent space used to compute k nearest neighbors of a point x_i. To be more clear: even the proximity of points can be learned in a dynamical way, so the graph is recomputed at each layer to take into account the change in the neighbors of the edges, that is of the neighbors.

ParticleNet

In our study on gas particles we applied the ParticleNet architecture described in [34], see fig. 3.25. Analyzing the EdgeConv block: the initial input are the particle coordinates, with them the distance are computed and the k nearest neighbors are found for each particle. Then the features chosen are fed to the block and as described in the previous paragraph the edge features are calculated knowing what the k nearest neighbors are. The real EdgeConv operation is implemented as a three stage operation. First a linear stage, than a Batch normalization one and finally a ReLU block. The curved arrow indicates that a copy of the features can pass through a parallel channel, this apparently improves performances [57]. Each of the three linear transformation layers in an EdgeConv block is governed by a hyperparameter known as channel, $C = (C_1, C_2, C_3)$, this sets the number of units to be used in each layer. The system is made by three EdgeConv blocks with different values of C[38] followed by a global pooling phase (average in the original architecture but mostly max in our study) and finally two fully connected layers. The first EdgeConv block takes the spatial coordinates of the particles to calculate the distances, while the subsequent blocks use the learned feature vectors as coordinates to make the whole graph dynamic as written above. After that there is a global average[39] pooling stage to aggregate the feature learned for all the particles in the cloud. Then a

[37] The feature vector for each node can change but it is not our case.

[38] $(64, 64, 64)$ for the first block, $(128, 128, 128)$ for the second and $(256, 256, 256)$ for the final third.

[39] Or max, in our application.

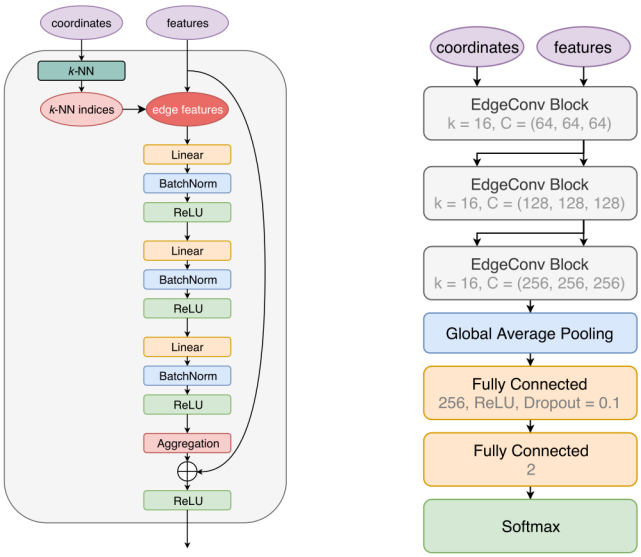


Figure 3.25. Left: inside the EdgeConv block of the ParticleNet implementation. **Right**: a schematic representation of the ParticleNet architecture.

fully connected Neural Network with 256 neurons activated by the ReLU function coupled with a dropout probability of 10% to prevent overfitting and the final stage is a softmax block.

Chapter 4

Application of our Network and Results

We devote this chapter to the analysis of results. Our aim is to predict the values of the mean squared displacement at two different points. The first point is in the middle of the glassy plateau; the second is the last point of the simulation, after the particles have been out of the glassy "cage" for a long time. We trained the Network first with one configuration at the time, then with four main configurations together. We probe the system in interpolation on three other configurations and in extrapolation.

4.1 Training details

Hardware and software

The training was performed on a NVIDIA V100 GPU with 32 GB of RAM memory. This GPU is powered by 640 Tensor Cores and 5120 CUDA Cores and runs at the frequency of 1245 MHz [58]. A CUDA Core is able to perform a single multiplication-addition operation like:

$$d = y \cdot z + a$$

where x, a, y and z are all single column vectors. A Tensor Core can perform a similar operation but with 4×4 matrices per clock cycle[1]. The Neural Network was

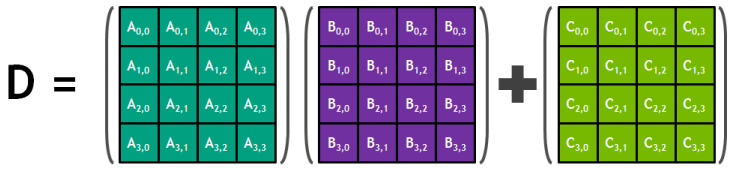


Figure 4.1. An example of a matrix operation performed by a Tensor Core in a clock cycle.

implemented using Google's TensorFlow API with the Keras Library.

Parameters' choice

We chose as loss function the Mean Squared Error:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \widehat{y}_i)^2$$

where y is the predicted value, $\widehat{y}$ the labeled, true, value and of course N is the number of samples taken into consideration. MSE is very strict with deviations from the ground truth value: the square "magnifies" errors[2]. This function is the one that will be minimized by the optimization algorithm, it is differentiable at all

[1] With somewhat reduced precision compared to CUDA cores, to tell the truth, but it is negligible for our purposes.

[2] It is just a quadratic polynomial after all

points, so no problem on that front. We monitored a different metric, too: the Mean Absolute Error. It is defined as:

$$MAE = \frac{1}{N}\sum_{i=1}^{N} |y_i - \widehat{y}_i| \,.$$

This function weighs all points at the same way, no extra weight on the "outliers" as instead occurs with the Mean Squared Error. MAE was used to judge the performance of the system and only the best model according to this metric was saved.

Dataset generation

To build the samples, taking inspiration from [31], we first generated 400 independent equilibrated configurations, we kept the positions and discarded the velocities. After that, for each of the 400 configurations, we run 20 independent simulations with different random initial velocities drawn from a Maxwell-Boltzmann distribution at the equilibrium temperature for a total of 8000 NVE simulations on which to calculate the Mean Square Displacement. To learn the details of the equilibration process, the ensembles used and further aspects refer to chapter 2, we just remind here that every simulation was performed with two kinds of particles: 80% A type and 20% B type for a total of $N = 4096$. This process was repeated for each of the seven state points spanning a wide range of pressures and temperatures, we used four of them as training configurations, see tab. 4.1. We trained the network to predict two values of the Mean Squared Displacement for particles of type A. The first one was chosen in the "glassy plateau" at time $t = 1.63$, see fig. 1.7, to test the network in the region where the curves diverge. The second one is the last point of the simulation at $t = 500$. For a full plot of the Mean Squared Displacement for all configuration, see fig. 2.12. The whole sample was split in 80% as training set and 20% as test set. The standard training was conducted with the following hyperparameters setting

- Batch size: 16 examples
- Pooling algorithm: Max pooling
- Number of first neighbors to be considered by the Graph Neural Network: 20

The training was performed for a total of 30 epochs[3], the learning rate was chosen as 10^{-3} for the first 10 epochs and reduced to 10^{-4} for the last 20.

4.2 Results

4.2.1 Training with one configuration at a time

Our first attempt was to train the model with one configuration at a time and to test it with a statistically independent dataset produced with the same configuration it was trained. Let's examine what happened with the lowest temperature

[3] An *epoch* ends when the entire dataset has been passed to the Neural Network forwards and backwards once. To do this it has to be split into smaller chunks known as *batches*. The *batch size* is the number of training examples included in a single batch.

Table 4.1. Starting values of Temperature, Pressure and Density. The four state points used for training have a gray background, sometimes they are referred as the "main" configurations.

T	P	$\frac{N}{V}$
0.44	2.93	1.203
0.45	2.70	1.196
0.47	2.24	1.178
0.49	1.78	1.158
0.50	1.55	1.148
0.52	1.09	1.127
0.56	1.17	1.073

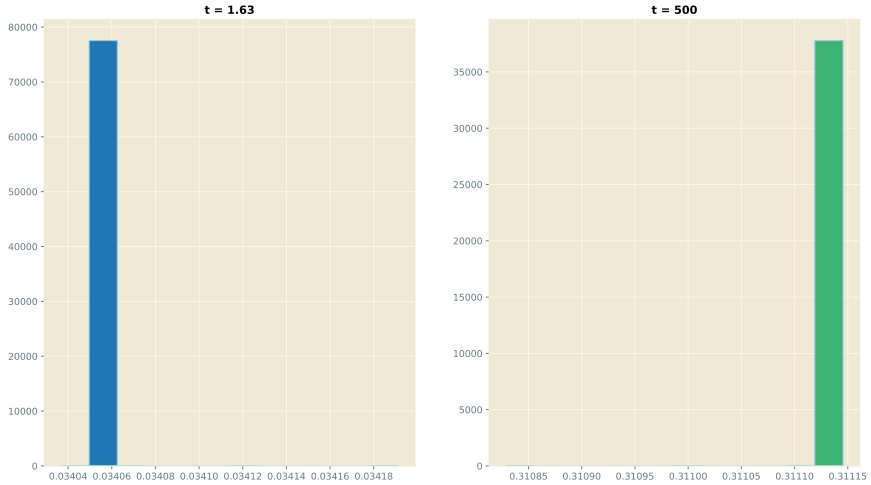


Figure 4.2. Prediction for the Network trained with the T = 0.44 state point, prediction on the same configuration.

configuration. This test seems to work well, see fig. 4.2: more than 99% of the predictions fit in a single bin[4]. Keeping on training the Neural Network with a single configuration at a time we get similar results prediction-wise, we get more Gaussian distribution with the highest temperature configurations, probably due to the increased randomization due to the thermal noise present in the starting configuration, see fig. 4.3. The complete plot of predictions we got from the system trained with a single configuration at a time, predicting the same configuration are in fig. 4.5. They are superimposed on the Mean Squared Displacement curves for the different configuration to get a graphical check of Network's performance in

[4]The number n of histograms' bins was chosen as $n = \max(FD, ST)$ where FD is the Freeman Diaconis method and ST is the Sturges' method. Without going into too many details, we can say that the first method takes into account data variability and data size, the second one takes into account only data size and is optimal for normally distributed data. The strategy of taking the maximum value between them guarantees good performances in almost all cases.

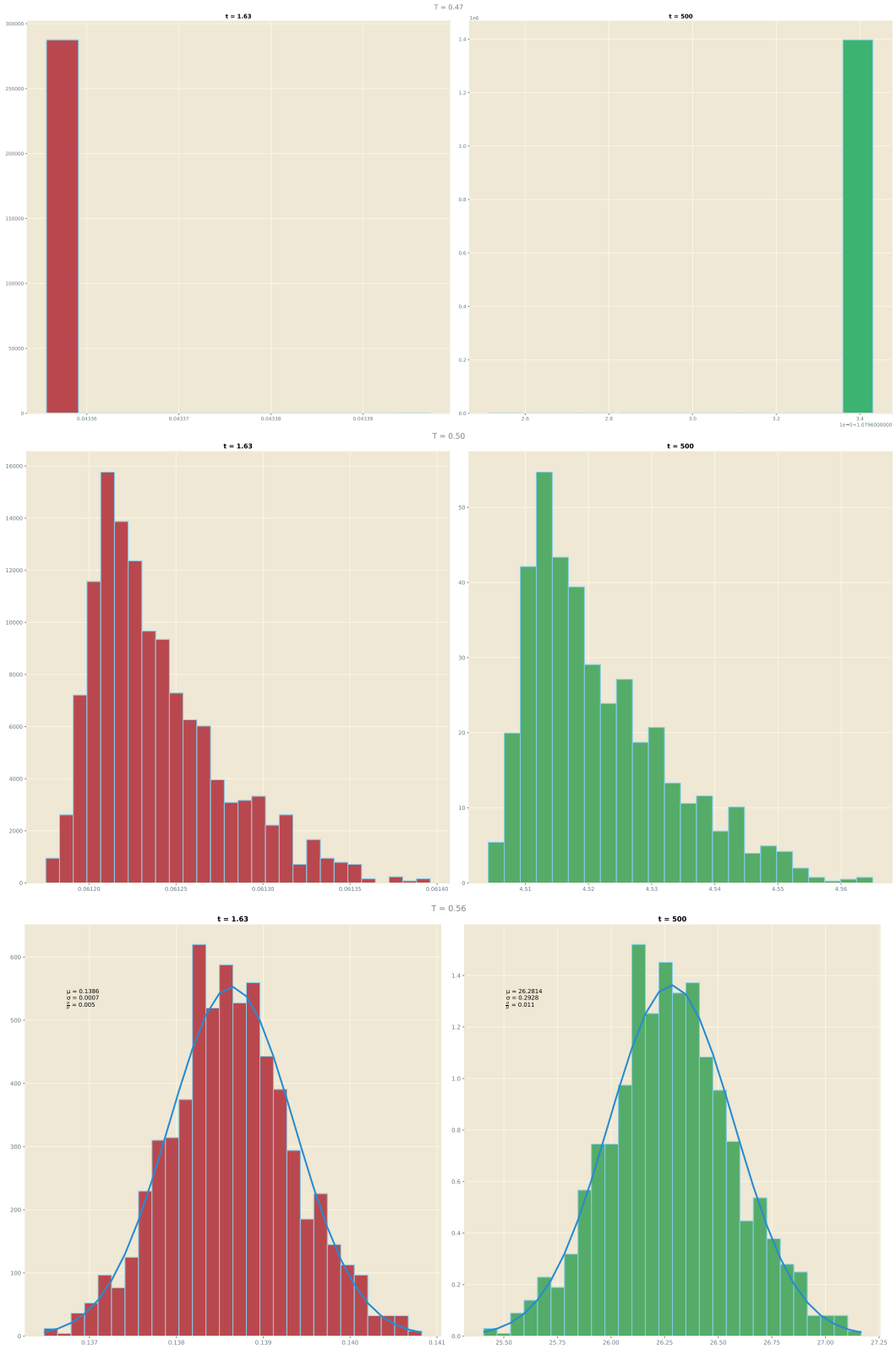


Figure 4.3. Predictions distribution for the Network trained with the T = 0.47, T = 0.50, T = 0.56 state point one at a time, prediction on the same configurations. On the left column the values at time = 1.63, on the right column values at time = 500. Mean and standard deviation of a Gaussian fit are shown where meaningful.

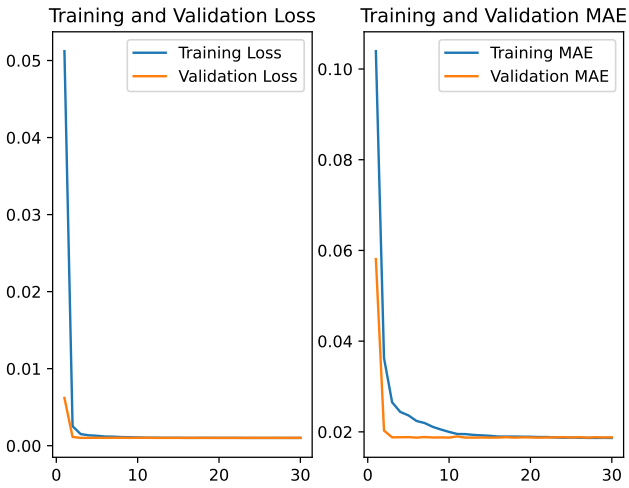


Figure 4.4. A plot of loss and MAE functions during the training with the single configuration at $T = 0.44$, epochs on the **x** axis.

this training scenario. A plot of the loss function and the MAE can be found at 4.4, we notice a quick convergence in the first ten epoch for both functions, and a monotonic decreasing trend.

4.2.2 Training with four configurations together

Following an intuitive path, as a second training step we tested our network with all the four configurations together making the system predict MSD values for them at the same time. The prediction distributions are plotted in fig. 4.6. They are all well fitted by a normal distribution whose parameters are reported on the graph. We found a somewhat longer left tail only on the $T = 0.50$ configuration, the one at the glass transition temperature, this is maybe due to an instability in the data on the critical point but further investigation is needed. To compare the distributions between them it is useful to look at the ratio between the standard deviation and the mean known as *coefficient of variation* $= \frac{\sigma}{\mu}$. This ratio is in the order of 10^{-2} for all of them except for the prediction for the lowest temperature configuration at time $= 500$ for which it is ≈ 0.3. This result is a real outlier, maybe due to a somewhat failed equilibration process for that configuration confusing the network, but there seems to be no sound theoretical explanation. To further investigate in the future we could repeat the training process substituting the configuration at $T = 0.44$ with the one at the close temperature of $T = 0.45$ to check if the phenomenon is still occurring. The predictions on the MSD plot are shown in fig. 4.7 and, besides the outlier point we wrote about above, are satisfying. The loss function and the MAE have a different trend in this training scenario, see fig. 4.8. We no longer see a monotonic drop but instead a steep increase and decrease for the first 11 epochs. After some investigation this phenomenon seems to be related to the configuration at T $= 0.56$ in the training set, the probable cause is the high thermal noise of this configuration, the only one above the glass transition. When the system is trained

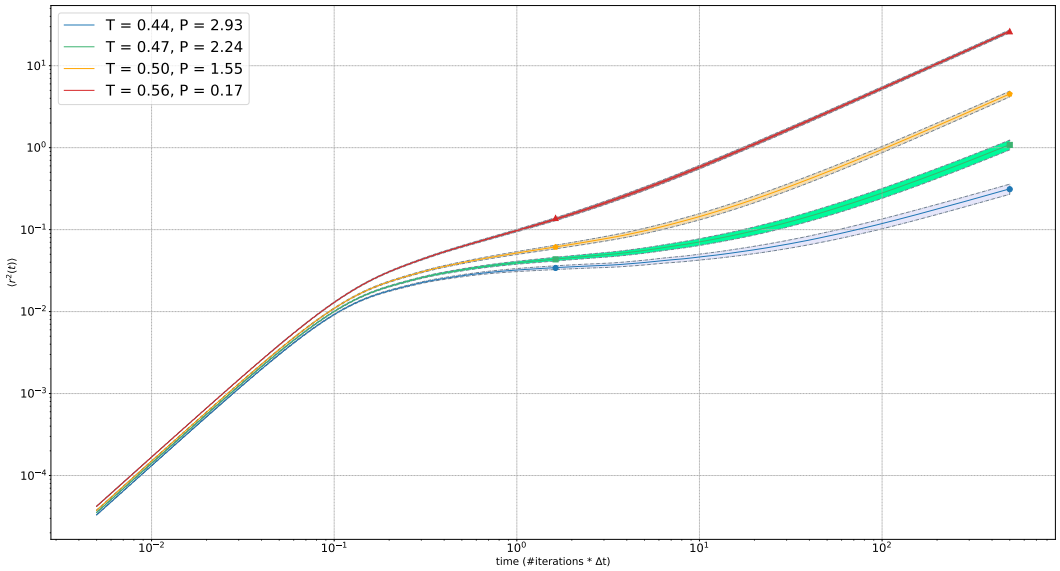


Figure 4.5. MSD graph and prediction for the Network trained with the four main configurations one at a time and predictions on the same configurations for each one of them. The light colored error bands in the graph represent ± one standard deviation, the thick line at the center is the mean value. The prediction points are reported with their error bands comprising ± one standard deviation or 68.27% of values when not normally distributed. Error bands on predicted points are not visible because of their small size.

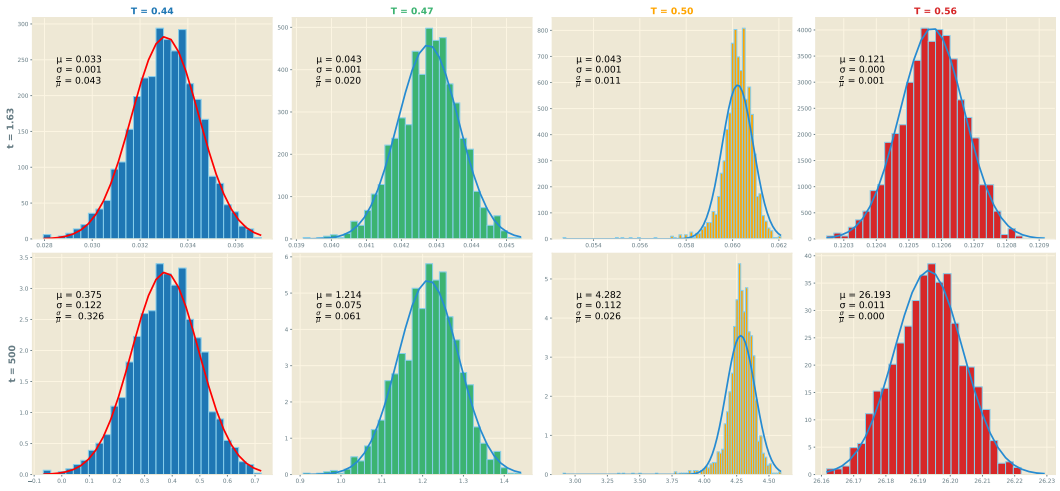


Figure 4.6. Predictions distribution for the Network trained on all four configurations together set to predict the same configurations at the same time.

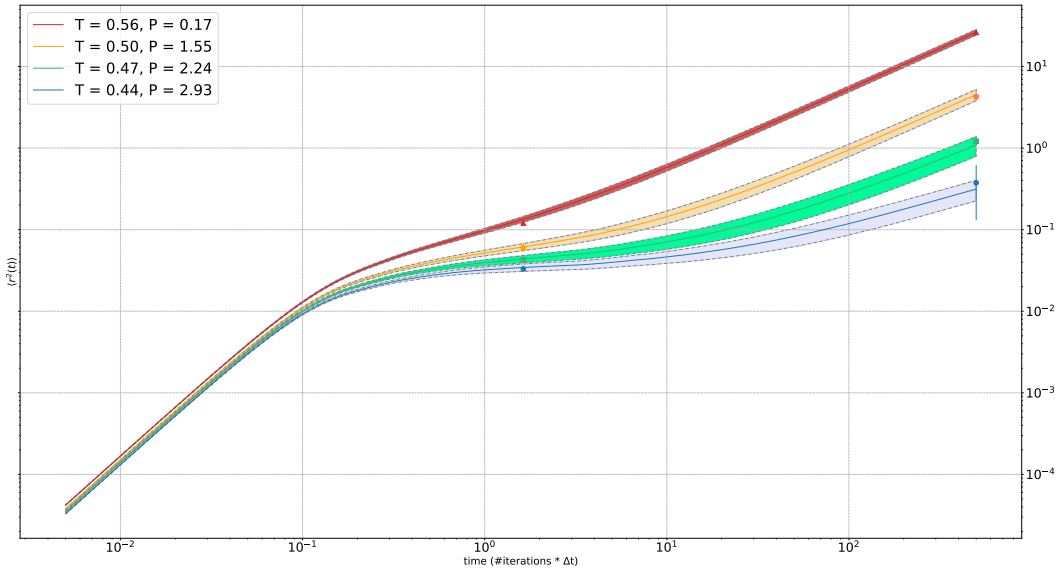


Figure 4.7. NN trained on all four configurations, prediction on the same configurations together. The error bands both for the predicted points and for the mean squared displacement curves include 96% of values.

only with the T = 0.56 configuration there is a similar trend of those functions not present in other training settings. In this section of the study we tried to tweak the parameter we judged most important in a graph neural network: the number of first neighbors to be considered by the graph. Modifying this value could in theory introduce a lot of noise: the graph could be constructed and updated with particles that should not, after all the starting value of 20 was chosen heuristically after some tries. We tried the two values of $k = 5$ and $k = 10$ but we found not significant differences with the results we have already exposed.

4.2.3 Interpolation Test

The next step in our experiment was to test the system in interpolation mode: will the Network trained on all four configurations together be able to predict values for other (three) configurations whose equilibrium starting temperatures are between the aforementioned four ones? To see the configurations distribution of the starting values of temperatures and pressure look again at table 4.1. Predictions distributions are plotted in fig. 4.9. Their Gaussian shape seems to be lost going towards the critical temperature of T = 0.50, and the distributions are more skewed and kurtotic[5]. This effect maybe due to some configuration pattern that confounds the neural network near the glass transition temperature. As previously written, a somewhat similar behavior was found when the system was set to predict two points of the MSD on the T = 0.50 configuration even when it was included in the training set. The plot of the predictions on the MSD is in fig. 4.10, besides

[5]Skewness and Kurtosis are the 3rd and 4th moment of a random variable X defined as $\mathbb{E}\left[\left(\frac{X-\mu}{\sigma}\right)\right]^{\alpha}$ where $\alpha \in \{3, 4\}$ respectively, $\mathbb{E}$ is the expected value. They give information on the how well the "tails" balance themselves and on their "weight".

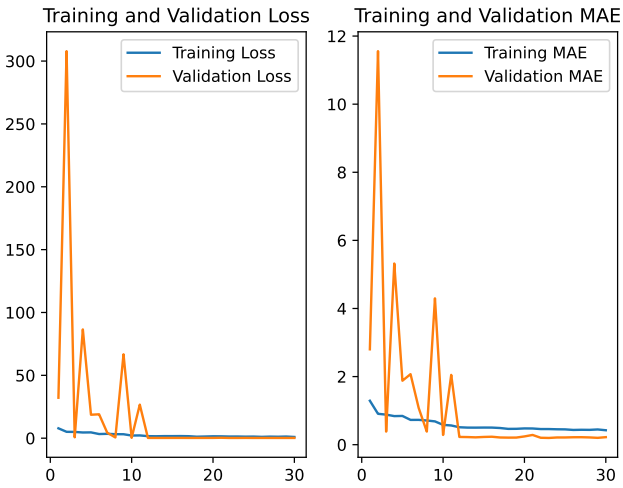


Figure 4.8. A plot of loss and MAE functions during the training with all the configuration together, epochs on the **x** axis.

the considerations on predictions distributions the quality of them seems to drop for the second point for all the configurations.

4.2.4 Extrapolation Test

Last question, will the network trained on a single configuration be able to generalize and predict values of the Mean Squared Displacement for others? Quick answer: no. Let's examine some examples. For the system trained with the configuration at T = 0.44, see fig. 4.11 and fig. 4.12 we can clearly see a pattern. Moving away from the training temperature, the prediction distributions gets more Gaussian, the ratio $\frac{\sigma}{\mu}$ stays of the same, low (10^{-3}), magnitude order for all the points, but the mean value is always very close to the MSD value predicted for T = 0.44, in other words the network is not able to extrapolate values outside of the dataset it has been trained with. The results are clear from fig. 4.13: the Network is predicting MSD points on the training configuration, not the one belonging to others as it should. We get similar results in extrapolation for the network trained on the highest temperature configuration, $T = 0.56$.

4.2.5 Conclusions and further developments

Neural Networks are known to work well when predicting values of the configurations belonging to the dataset used for training, they are also in theory able to generalize in an interval included in the training dataset (interpolation). Outside of that interval (extrapolation) the performances are not that great in general. Overall this is what we observed in our study, the suboptimal results in interpolation are probably due to the the reduced dataset size partly due to computational time limitations. The most effective way to improve the prediction in interpolation is just to have more data, the sample used in [31], a study similar but with a different

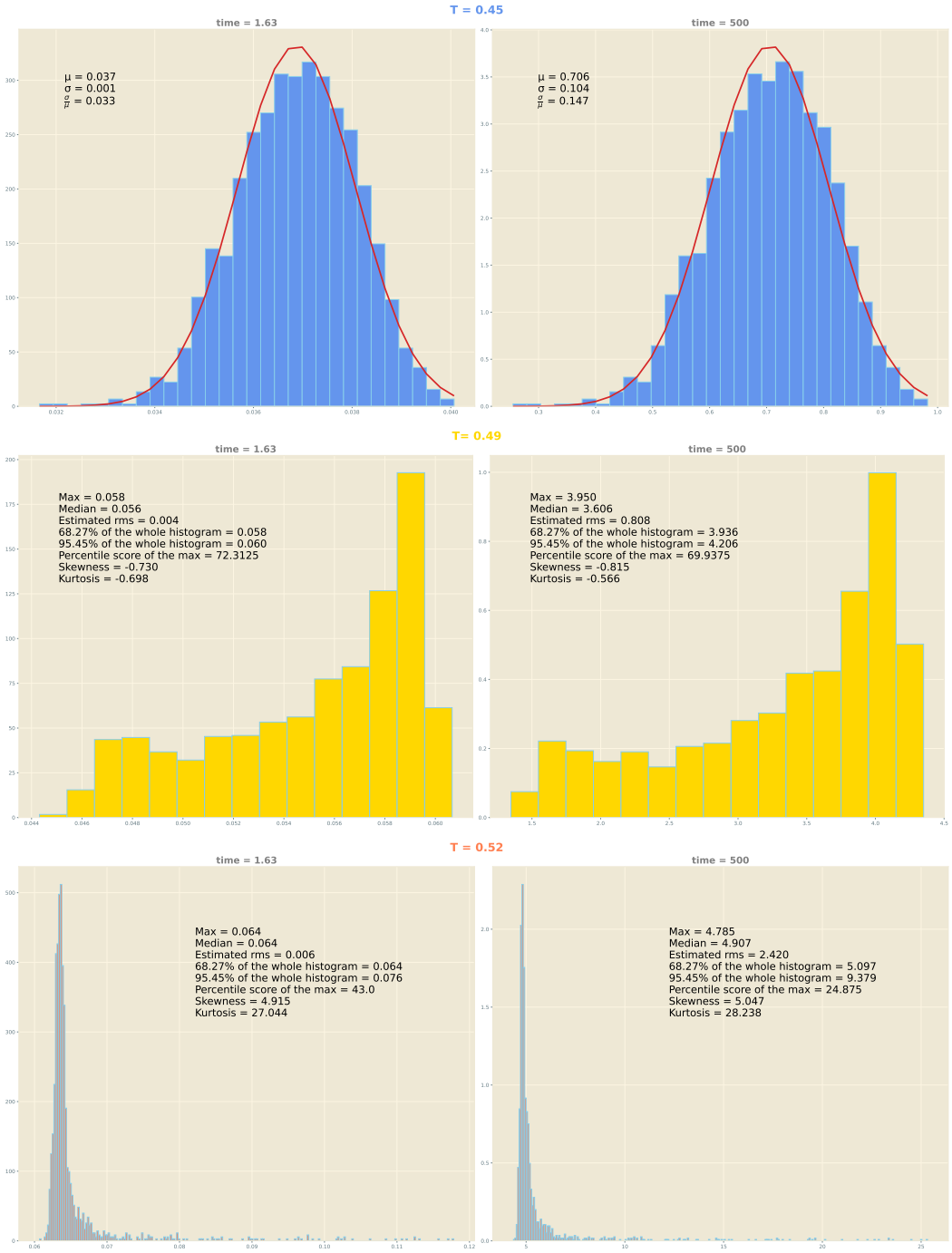


Figure 4.9. NN trained on all four main configurations, prediction distributions for the other three not included in the training set.

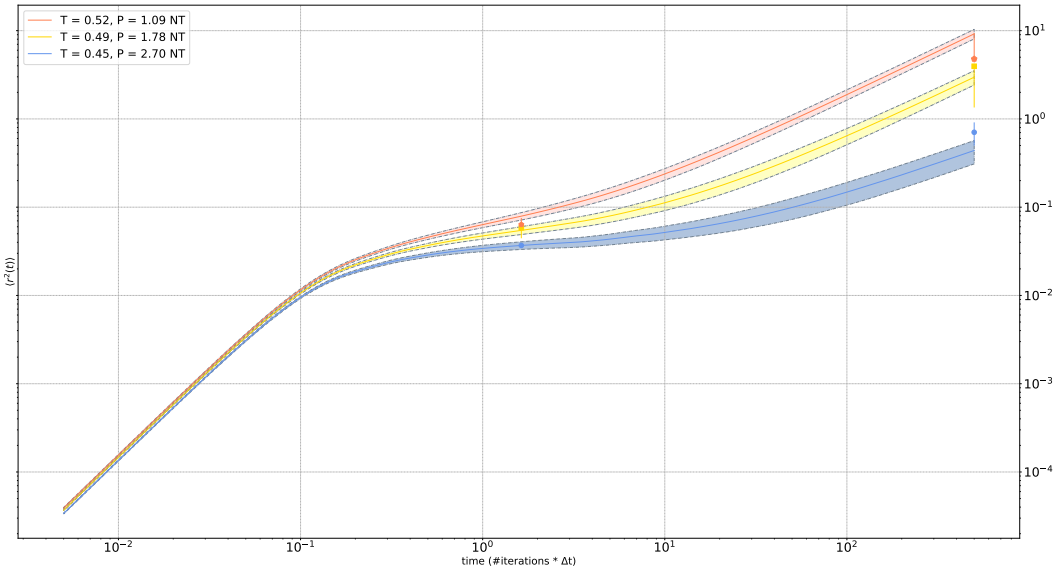


Figure 4.10. NN trained on all four configurations, predictions for the other three not included in the training set.

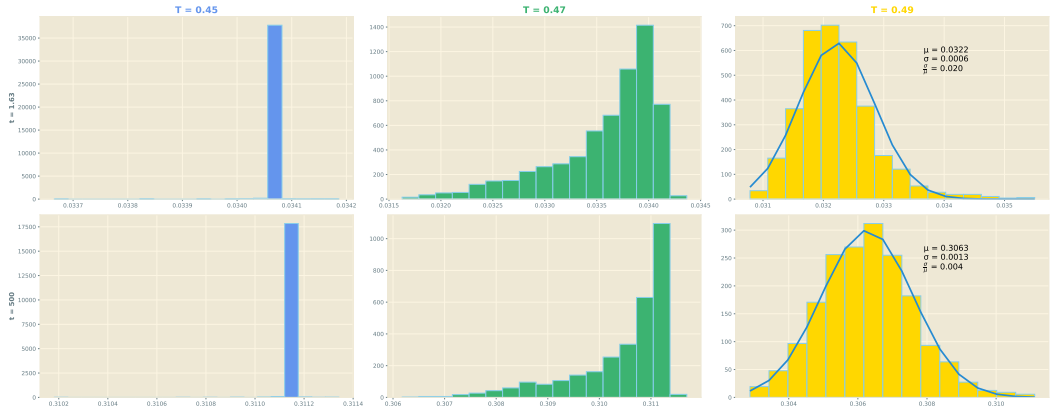


Figure 4.11. Prediction distribution on the first group of configurations for the neural network trained on the lowest temperature configuration, T = 0.44.

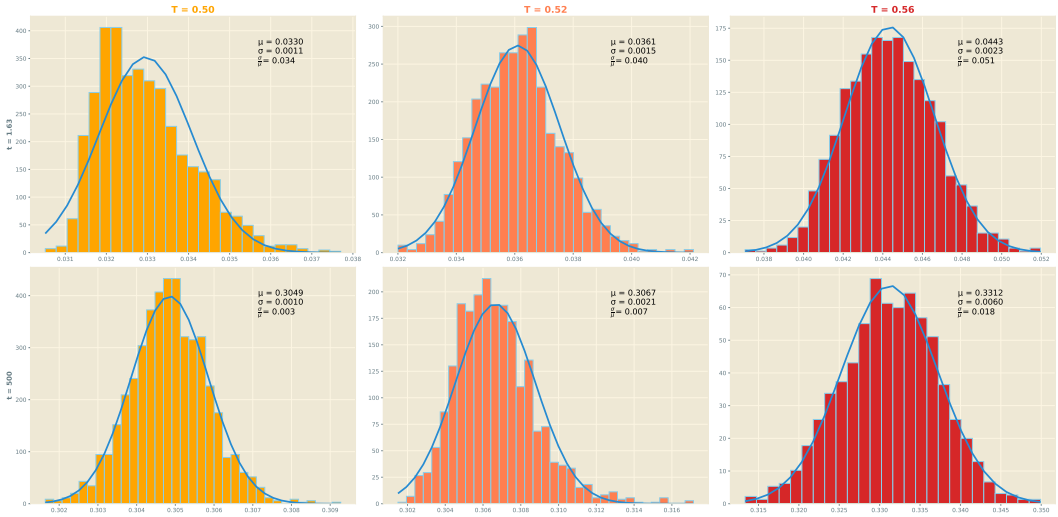


Figure 4.12. Prediction distribution on the second group of configurations for the neural network trained on the lowest temperature configuration, $T = 0.44$.

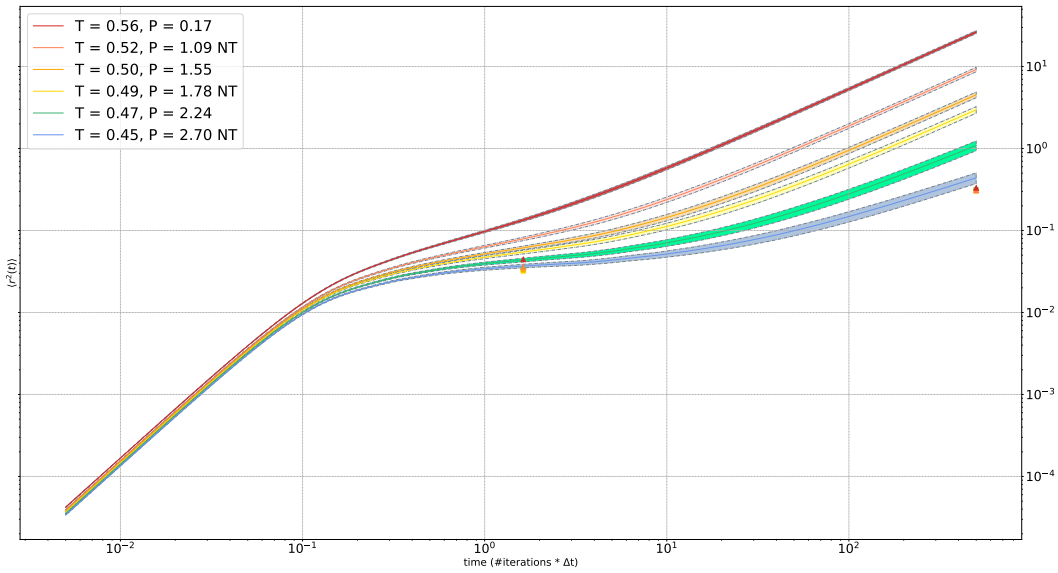


Figure 4.13. NN trained with the configuration at $T = 0.44$. MSD curves with the prediction for other configurations. The MSD curve for $T = 0.44$ is not plotted.

kind of network, was at least 50% bigger than the one we used. A way to try to get more data could be to exploit the symmetries of the particles configuration: one of the 24 cubic symmetries could be applied at random on the dataset in a process known as *data augmentation*. Hyperparameters can be further tweaked to get better results, for example the learning learning rate which was kept constant in this simulation. After having increased the dataset it could be also important to understand why reducing the number of nearest neighbors seems to give the same performances. The dataset was generated using CPU only, and that took a lot of time. Another aspect to consider is the known not so good performances in expressivity, what function can or cannot be learned by the system, of the Graph Neural Network with message passing as the one we used in this study. Better results could be achieved for example leveraging the recently proposed subgraph aggregation method [59], or the combination of GNN with transformers [60], both resulting in Deep Neural Networks models that are much more expressive than message passing GNN. An important remark is the Dynamic Graph Neural Network we used had no knowledge of the physical system to be predicted, they could be tested in another scenario and compare the results with ours.

Bibliography

[1] P. G. Debenedetti, F. H. Stillinger, *Supercooled liquids and the glass transition*, Nature 410, 259 (2001).

[2] G. Biroli, J.P. Garrahan, *Perspective: The glass transition*, J. Chem. Phys. 138, 12A301 (2013); doi: 10.1063/1.4795539 https://aip.scitation.org/doi/10.1063/1.4795539

[3] Eric R. Weeks, D.A. Weitz, *Subdiffusion and the cage effect studied near the colloidal glass transition*, https://www.sciencedirect.com/science/article/abs/pii/S0301010402006675

[4] Liesbeth M. C. Janssen, *Mode-Coupling Theory of the Glass Transition: A Primer* https://www.frontiersin.org/articles/10.3389/fphy.2018.00097/full

[5] Second order phase transitions, https://ocw.tudelft.nl/wp-content/uploads/lecture6_reading.pdf

[6] E.D. Zanotto, *Do cathedral glasses flow?* American Journal of Physics, 66, 392-395 (1998) https://aapt.scitation.org/doi/10.1119/1.19026

[7] Daniel V. Schroeder *Ising Model (JavaScript/HTML5 version)* https://physics.weber.edu/thermal/isingHTML5.html

[8] A.A. Kilbas, H.M. Srivastava, J.J. Trujillo, *Theory and Applications of Fractional Differential Equations*, Elsevier, 2006 https://www.elsevier.com/books/theory-and-applications-of-fractional-differential-equations/kilbas/978-0-444-51832-3

[9] Alex Malins, Jens Eggers, C. Patrick Royall, Stephen R. Williams, Hajime Tanaka *Identification of long-lived clusters and their link to slow dynamics in a model glass former*, The Journal of Chemical Physics, 2013 https://aip.scitation.org/doi/10.1063/1.4790515

[10] Smarajit Karmakar *An Overview on Short and Long Time Relaxations in Glass-forming Supercooled Liquids* Journal of Physics Conference Series 759(1):012008 https://iopscience.iop.org/article/10.1088/1742-6596/759/1/012008

[11] Jean-Philippe Bouchaud, Giulio Biroli, *On the Adam-Gibbs-Kirkpatrick-Thirumalai-Wolynes scenario for the viscosity increase in glasses*, The Journal of Chemical Physics, 2004 https://aip.scitation.org/doi/10.1063/1.1796231

[12] Jean-Pierre Hansen, Ian R. McDonald *Theory of Simple Liquids with applications to Soft Matter* Academic Press, 2013

[13] Eric R. Weeks, D. A. Weitz, *Subdiffusion and the cage effect studied near the colloidal glass transition* https://www.researchgate.net/publication/222673779_Subdiffusion_and_the_cage_effect_studied_near_the_colloidal_glass_transition

[14] W. van Megen, S. M. Underwood, P. N. Pusey, *Nonergodicity parameters of colloidal glasses*, https://journals.aps.org/prl/abstract/10.1103/PhysRevLett.67.1586

[15] Weber T.A, Stillinger F.H., *Local order and structural transitions in amorphous metal-metalloid alloys* https://journals.aps.org/prb/abstract/10.1103/PhysRevB.31.1954 (1985)

[16] David R. Reichman, Patrick Charbonneau, *Mode-coupling theory*, Journal of Statistical Mechanics: Theory and Experiment, 2005 https://iopscience.iop.org/article/10.1088/1742-5468/2005/05/P05013

[17] Lennard-Jones, John Edward (1931), *Cohesion*, Proceedings of the Physical Society, https://iopscience.iop.org/article/10.1088/0959-5309/43/5/301.

[18] Vollmayr-Lee, Katharina (2020), *Introduction to molecular dynamics simulations*, https://aapt.scitation.org/doi/10.1119/10.0000654, https://arxiv.org/abs/2001.07089

[19] *Large-scale Atomic/Molecular Massively Parallel Simulator*, https://www.lammps.org/.

[20] Kob Walter, Andersen Hans, *Testing mode-coupling theory for a supercooled binary Lennard-Jones mixture I: the van Hove correlation function*, http://arXiv.org/abs/cond-mat/9501102v1.

[21] Cohen-Tannoudji C., Diu B., Laloe F., *Quantum mechanics, Vol. 1*, John Wiley & Sons, New York, (1977)

[22] Allen M.P. and Tildesley D.J., *Computer Simulation of Liquids*, Oxford University Press, (1990)

[23] Gould Harvey, Tobochnik Jan, *Statistical and thermal physics with computer applications*, Princeton University Press (2010)

[24] Mark Tuckerman, *Statistical Mechanics: Theory and Molecular Simulation*, Oxford Graduate Press, 2010

[25] Pieter J. in 't Veld, Steven J. Plimpton, Gary S. Grest, *Accurate and efficient methods for modeling colloidal mixtures in an explicit solvent using molecular dynamics* Computer Physics Communications, Volume 179, Issue 5, 1 September 2008, Pages 320-329 https://doi.org/10.1016/j.cpc.2008.03.005

[26] Angell C. A., Ngai K. L., McKenna G. B., McMillan P. F., Martin S. W., *Relaxation in glassforming liquids and amorphous solids.* Journal of Applied Physics 88, 3113–3157 (2000)

[27] W. G. Hoover, *Canonical dynamics: Equilibrium phase-space distributions* Phys. Rev. A 31, 1695–1697 (1985) https://doi.org/10.1103/PhysRevA.31.1695

[28] Glenn J. Martyna, Douglas J. Tobias, Michael L. Klein, *Constant pressure molecular dynamics algorithms.*

[29] Jeffrey R. Fox, Hans C. Andersen, *Molecular dynamics simulations of a supercooled monatomic liquid and glass* J. Phys. Chem. 1984, 88, 18, 4019–4027 https://doi.org/10.1021/j150662a032

[30] D. Frenkel, B. Smit, *Understanding Molecular Simulation: From Algorithms to Applications* Academic Press, San Diego, 2002

[31] Bapst V., Keck T. et al. *Unveiling the predictive power of static structure in glassy systems*, https://www.nature.com/articles/s41567-020-0842-8.

[32] https://raw.githubusercontent.com/nihil39/Thesis_test/main/Grafici/prova_64_colors.gif

[33] Wang Y., Sun Y., Liu Z., Sarma S. E., Bronstein M. M., Solomon J. M., *Dynamic graph CNN for learning on point clouds*, https://arxiv.org/abs/1801.07829

[34] Qu Huilin, Gouskos Loukas, *ParticleNet: Jet Tagging via Particle Clouds* https://arxiv.org/abs/1902.08570

[35] Dumoulin Vincent, Visin Francesco, *A guide to convolution arithmetic for deep learning* https://arxiv.org/abs/1603.07285

[36] Nielsen Michael A., *Neural network and deep learning*, Determination Press, 2015

[37] Goodfellow Ian, Bengio Yoshua, Courville Aaron, *Deep Learning*, MIT Press, 2016 http://www.deeplearningbook.org

[38] Tappert Charles C., *Who is the father of deep learning?* https://ieeexplore.ieee.org/abstract/document/9070967

[39] Sanchez-Lengeling, et al., *A gentle introduction to graph neural networks*, Distill, 2021.

[40] Sheffer, Henry Maurice *A set of five independent postulates for Boolean algebras, with applications to logical constants*, Transactions of the American Mathematical Society, volume 14, 1913.

[41] LeCun Yann, Cortes Corinna, Burges Christopher J.C, *The MNIST database of handwritten digits*, http://yann.lecun.com/exdb/mnist/

[42] Mayer Jürgen, *Drawing an elephant with four complex parameters* https://aapt.scitation.org/doi/10.1119/1.3254017

[43] Schmidhuber Jürgen, *Who invented backpropagation?* https://people.idsia.ch/~juergen/who-invented-backpropagation.html

[44] Cauchy A., *Méthode générale pour la résolution des systémes d'équations simultanées.* C. R. Acad. Sci. Paris, 25:536–538, 1847

[45] Srivastava Nitish et al, *Dropout: A Simple Way to Prevent Neural Networks from Overfitting* https://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf

[46] https://www.wikiwand.com/en/Gradient_descent

[47] Huber Johann, *Batch normalization in 3 levels of understanding* https://towardsdatascience.com/batch-normalization-in-3-levels-of-understanding-14c2da90a338

[48] Ioffe Sergey, Szegedy Christian, *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*, https://arxiv.org/abs/1502.03167

[49] Kleinberg Robert, Li Yuanzhi, Yuan Yang, *An alternative view: when does SGD escape local minima?* https://arxiv.org/abs/1802.06175

[50] Hardt Moritz, Recht Benjamin, Singer Yoram, *Train faster, generalize better: stability of stochastic gradient descent* https://arxiv.org/abs/1509.01240

[51] Gradient Descent https://www.ibm.com/cloud/learn/gradient-descent

[52] J. M. S. Prewitt, Object enhancement and extraction," Picture Processing and Psychopictorics, B. Lipkin and A. Rosenfeld, Eds., New York: Academic Press, 1970, pp. 75-149.

[53] Chollet François, *Deep Learning with Python*, Manning, 2017

[54] Krizhevsky Alex, Sutskever Ilya, Hinton Geoffrey E., *ImageNet Classification with Deep Convolutional Neural Networks* https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

[55] *ImageNet Large Scale Visual Recognition Challenge* https://image-net.org/challenges/LSVRC/index.php

[56] Niepert Mathias, Ahmed Mohamed, Kutzkov Konstantin, *Learning Convolutional Neural Networks for Graphs* https://proceedings.mlr.press/v48/niepert16.html

[57] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun, *Deep Residual Learning for Image Recognition* https://ieeexplore.ieee.org/document/7780459

[58] https://www.nvidia.com/en-us/data-center/v100/

[59] Beatrice Bevilacqua, Fabrizio Frasca, Derek Lim, Balasubramaniam Srinivasan, Chen Cai, Gopinath Balamurugan, Michael M. Bronstein, Haggai Maron, *Equivariant Subgraph Aggregation Networks* https://arxiv.org/abs/2110.02910v2

[60] Zhanghao Wu, Paras Jain, Matthew A. Wright, Azalia Mirhoseini, Joseph E. Gonzalez, Ion Stoica, *Representing Long-Range Context for Graph Neural Networks with Global Attention* Proceedings NeurIPS 2021 https://proceedings.neurips.cc/paper/2021/hash/6e67691b60ed3e4a55935261314dd534-Abstract.html