Super-Resolution with Convolutional Neural Networks for the 1D, 2D and 3D Ising Model

Fakultät für Physik und Geowissenschaften
Universität Leipzig

Master Thesis

Jan Zimbelmann

Prof. Dr. Wolfhard Janke
Universität Leipzig

Dr. Henrik Christiansen
Universität Leipzig

Leipzig, June 23, 2021

Contents

1 Introduction **1**
1.1 What Needs to be Done . . . 2

2 Theory **3**
2.1 Statistical Physics Preamble . . . 4
2.1.1 Equilibrium and Canonical Ensemble . . . 4
2.1.2 Fluctuations . . . 5
2.2 Ising Model . . . 6
2.2.1 Description . . . 6
2.2.2 Critical Exponents and Phase Transitions . . . 7
2.2.3 Analytic Solution for the Energy of the 1D Ising Model . . . 9
2.2.4 1D Renormalization by Decimation . . . 10
2.3 Monte Carlo Simulation . . . 11
2.3.1 Markov Chains . . . 12
2.3.2 Metropolis Algorithm . . . 13
2.3.3 Heat Bath Algorithm . . . 13
2.3.4 Monte Carlo Renormalization Group . . . 14
2.4 Neural Networks in Machine Learning . . . 15
2.4.1 Neural Network Layers . . . 16
2.4.2 Shallow vs Deep Neural Network . . . 16
2.4.3 Neural Network Connections . . . 17
2.4.4 Activation Function . . . 18
2.4.5 Dying ReLU Problem and ELU Function . . . 19
2.4.6 1D Convolutional Neural Networks . . . 20
2.4.7 2D Convolutional Neural Networks . . . 21
2.4.8 3D Convolutional Neural Networks . . . 22
2.4.9 Cost Function . . . 23
2.4.10 Backpropagation . . . 24
2.4.11 Adam Optimizer . . . 25
2.4.12 Overfitting . . . 25
2.5 Super-Resolution Procedure . . . 27
2.5.1 Method . . . 27
2.5.2 Decimation and Reconstruction of the Ising model . . . 29
2.5.3 Resolving to larger System Sizes . . . 29
2.5.4 Numerical Solution to Rescale the Temperature . . . 30
2.5.5 Regularization Term to the Loss Function . . . 31
2.5.6 Finite Size Considerations . . . 32
2.5.7 Alternative Blocking Method . . . 33

3 Simulations and Evaluations **34**
3.1 1D Simulation . . . 35
3.1.1 Data Preparation . . . 35
3.1.2 Results . . . 35
3.1.3 Discussion . . . 43
3.2 2D Simulation . . . 43
3.2.1 Temperature Transformation . . . 44
3.2.2 Data Preparation . . . 46
3.2.3 Visualisation of the Spin Configurations . . . 46
3.2.4 Visualisation of the Kernels . . . 49
3.2.5 Results . . . 51
3.2.6 Investigation of the unpolished Results . . . 56
3.2.7 Reconstruction from smaller System Sizes . . . 58
3.2.8 Discussion . . . 61
3.3 3D Simulation . . . 63
3.3.1 Temperature Transformation . . . 63
3.3.2 Data Preparation . . . 64
3.3.3 Results . . . 64
3.3.4 Discussion . . . 68

4 Conclusion **69**

5 Appendix **70**
5.1 Python Code for the 2D CNN Training . . . 70
5.2 Python Code for the CNN Super-Resolution . . . 79
5.3 C++ Code for simulating Spin Configurations . . . 88

References **99**

Abbreviations

MC	Monte Carlo
1D	One-Dimensional
2D	Two-Dimensional
3D	Three-Dimensional
$\mathcal{SR}$	Super-Resolution
PNAS	Proceedings of the National Academy of Sciences of the United States of America
MCMC	Markov chain Monte Carlo
RG	Renormalization Group
CNN	Convolutional Neural Network
pbc	Periodic Boundary Condition
FSS	Finite Size Scaling
HB	Heat Bath
MCRG	Monte Carlo Renormalization Group
DLSS	Deep Learning Super Sampling
ReLU	Rectified Linear Unit
ELU	Exponential Linear Unit
SGD	Stochastic Gradient Descent
Adam	Adaptive Momentum Estimator

1 Introduction

Machine learning has now become a very popular method of data analysis [1]. There are multiple reasons for machine learning approaches to currently succeed. It is an optimization technique to solve otherwise unknown problems which benefits a lot from parallelization. A subcategory of machine learning are neural networks. They very useful for categorizing patterns. A typical application for this is to recognize hand written digits or to super-resolve images. This work is especially inspired by super-resolving images since I will try to also resolve the system size of the Ising model with a neural network.

The Ising model is an approximation of a magnet with each spin being arranged in a binary state, typically on a lattice. Even though this is a very simple model, it does show a phase transition for dimension 2 or higher. The Ising model serves as a very important toy model in computational physics and it has many universal properties which also apply to more complex models. A problem for this model is the critical slowing down in computational simulations of the Ising model at the critical temperature. It is a problem which originates from a divergent correlation length [2, p. 214 ff.]. This is especially crucial when studying larger system sizes, since it would take very long to capture all important states in a more traditional computer simulation, like a Monte Carlo (MC) simulation with a Metropolis algorithm. Though it is worth mentioning, that there are other MC algorithms, which deal with this particular problem of the critical slowing down much better [3].

This is where the neural network enters the game. When using a renormalization procedure on a Ising spin configuration to decrease the system length to a half, the idea is then to train a neural network with supervised learning to revert the renormalization, bringing the system back. It would then be possible to run a computer simulation with a MC method of the Ising model at small system sizes and rescale their properties to a much larger size. It is not required for this super-resolution ($\mathcal{SR}$) to be an exact reversion to the problem, since only the probability distribution is what plays a role when calculating a statistical average of an observable. The idea comes from a paper from Efthymiou *et al.* [4]. The paper presents this method for the one dimensional (1D) and two dimensional (2D) case. However it does not go into every detail of training a neural network for this task and gives no information on the neural network itself. Moreover it does not discuss any problems concerning finite size problems.

This master's thesis will review all the necessary theory for this task to complete. This includes the physics parts as well as the theory behind the neural networks. It shows exactly of how to implement this method and gives further considerations by studying differently sized networks. Furthermore it documents and shows the results of the implementation of this method for the 1D, 2D and the three dimensional (3D) case. The method is also tested for very small system sizes in the 2D case. Finally there is a discussion about the results and a conclusion. The conclusion then identifies the strengths, the weaknesses as well as the limitations of this method.

The $\mathcal{SR}$ procedure for the Ising model is not the only case in physics for which a super-resolution is performed. In march 2021 a super-resolution for cosmological simulations with the use of neural networks is published in the PNAS journal [5]. Here a low resolution dark matter simulation is enhanced to reconstruct a corresponding high resolution simulation. By simulating a lower amount of particles and then enhancing them with the super-resolution procedure, it is resulting in faster computations of orders of magnitudes. It shows to be capable of a good reconstruction of the authentic statistical properties.

1.1 What Needs to be Done

This section gives a forecast of the simulations and methods involved for the $\mathcal{SR}$ procedure, to give a better understanding of why the broad width of different subjects in the theory are being discussed. The basic utility of this neural network method is that the $\mathcal{SR}$ procedure takes a set of spin configuration and resolves it to a spin configuration twice the original system length. This can then be repeated for the newly obtained spin configurations. However to accomplish this, different fields of theory have to be considered and multiple methods are in use.

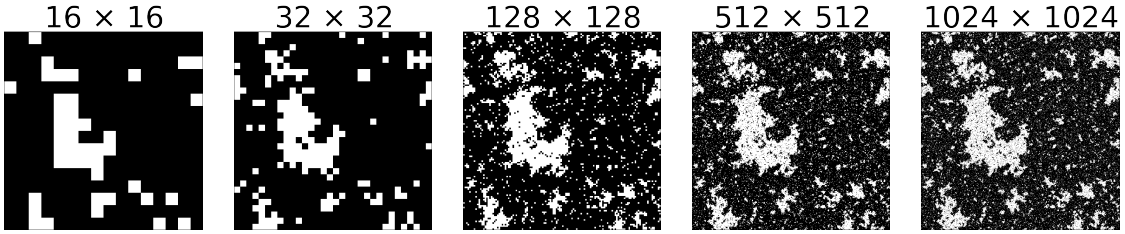


Fig. 1: Super-resolving a 2D Ising spin configuration simulated at critical temperature. Starting at a system size of 16×16. From left to right: original configuration, the result after: one, three, five and six $\mathcal{SR}$ steps.

The $\mathcal{SR}$ procedure starts by needing an initial set of spin configurations. This set of spin configurations is created with a statistical simulation called the Metropolis MC simulation. The Metropolis algorithm is a Markov chain Monte Carlo (MCMC) method for creating a series of spin configurations according to a certain probability distribution. This algorithm is also use for comparing the results at different sizes of the later obtained spin configurations with the super-resolution technique.

This method is a procedure for changing the system size of the Ising model. For an analysis of this, there is the renormalization group (RG) theory which gives insights to the changes of the Ising model under a length scale transformation. As a matter of fact, when resolving to larger system sizes, the newly created spin configuration corresponds to a different temperature. The RG theory is hereby useful to solve the temperature transformation from one system size to the other. This however is only known to be solvable analytically for the first dimension of the Ising model. For any larger dimension, a numeric solution is made instead. This numeric solution is also created with the use of Metropolis Monte Carlo simulations.

Before resolving to larger system sizes, the neural network needs to be trained to accomplish such a task. For this, a convolutional neural network (CNN) is chosen because it is independent of the initial system size and can therefore be used for any initial system sizes. For training the network, an initial set of spin configuration is taken and decimated to half its original system length, as it is known to be done within the RG theory. Afterwards the network is trained to revert this decimation for different temperatures. Once trained, the network can be used to resolve to larger system sizes, however for every $\mathcal{SR}$ step, there is a temperature transformation. So it is required to take a differently trained neural network for every further $\mathcal{SR}$ step. With this method it is possible to resolve a set of smaller Ising spin configurations to a set of larger Ising spin configurations.

2 Theory

This work deals with super-resolving the Ising model to a larger system size with the use of a trained neural network. Implying that the physical properties of the enlarged system still correspond accordingly to the physical properties of a system which is originally simulated on larger system sizes. This problem is an interdisciplinary field of research in which neural networks are applied to a computational physics problem. Computational physics is a third field in physics itself, next to experimental and theoretical physics, all of which are deeply interconnected. In the computational field the physical problems are solved with the means of computational simulations and algorithms. Since there are so many subjects to cover in this thesis, the focus will not be on covering every topic in its entirety but rather narrowed to the specific problem of super-resolving an Ising model with CNNs. All subjects are shown in the following 5 subsections:

2.1 A short introduction to the field of statistical physics itself. In it, statistical models in equilibrium are described with the use of the concept of canonical ensembles. Later on fluctuations of observables can be used to describe the specific heat and magnetic susceptibility.

2.2 The Ising model and its physical properties are introduced in this section. Basic observables and the critical exponents are defined, as well as the phase transition is discussed. Furthermore the analytic solution of the energy and the renormalization is shown for the 1D case.

2.3 MC methods are introduced. The Metropolis algorithm is a common method and is used here for investigating the Ising model. It is also used in the later part for creating a set of spin configurations to resolve to larger system sizes. Furthermore the heat bath algorithm is introduced and is later referred to in an experiment of having an alternative method to resolve to larger system sizes for the 1D case.

2.4 The section explains neural networks from scratch, starting with an example of a fully connected neural network. The method of choice however are CNNs, which behave similarly to a local filter. Afterwards it is described how a neural network learns with a method called gradient descent and how this learning is optimized by the Adam optimizer. Further problems when designing a network are also discussed, as well as some ways of dealing with them.

2.5 The $\mathcal{SR}$ procedure is supposed to resolve a set of Ising spin configurations to a larger system size. This is done with a neural network. The network is trained to reconstruct a decimated set of spin configurations to the original spin configurations. Once trained it is supposed to be able to indefinitely double the system size. A newly obtained spin configuration corresponds to a different temperature. A numerical method is introduced for calculating this temperature transformation.

2.1 Statistical Physics Preamble

This section introduces the statistical physics which is necessary to understand the methods used in this thesis. A system of many entities, such as a system of particles or spins, are often very difficult or even impossible to describe by solving equations from traditional mechanics or quantum mechanics. Such a system often has a large amount of degrees of freedom which makes it difficult to find an analytic solution to such a problem. This leads to a description of macroscopic observables by the examination of probabilities.

2.1.1 Equilibrium and Canonical Ensemble

Now a thermal equilibrium is considered, i.e., a system in contact with a heat bath at a temperature T with a fixed amount of particles. This is then called an canonical ensemble. The probability p of a state being occupied in equilibrium is described by the Boltzmann weights [6, p. 7 ff.]:

$$p(\{s\}) = \frac{1}{\mathcal{Z}} \cdot \mathrm{e}^{-\frac{\mathcal{H}(\{s\})}{k_B T}} \tag{1}$$

$\mathcal{H}$ is the total energy or Hamiltonian of the system, k_B is the Boltzmann constant. Since the total probability for all the possible system configurations $\{s\}$ must be equal to 1, the Partition function $\mathcal{Z}$ is defined as the following:

$$\mathcal{Z} = \sum_{\{s\}} \mathrm{e}^{-\frac{\mathcal{H}(\{s\})}{k_B T}} \tag{2}$$

Typically an inverse temperature is defined as $\beta = \frac{1}{k_B T}$. Now one might be interested in any expectation value, noted as: $\langle ... \rangle$, of an observable $\mathcal{O}$. It is possible to attain it when summing the observable and multiply it with the probability for every possible system configuration:

$$\langle \mathcal{O} \rangle = \sum_{\{s\}} \mathcal{O} \cdot p = \frac{1}{\mathcal{Z}} \sum_{\{s\}} \mathcal{O} \cdot \mathrm{e}^{-\beta \mathcal{H}(\{s\})} \tag{3}$$

A common observable to consider in this manner would be the expectation value of the energy $\langle E \rangle$. It is also known to be the internal energy U:

$$U = \langle E \rangle = \frac{1}{\mathcal{Z}} \sum_{\{s\}} E \cdot \mathrm{e}^{-\beta \mathcal{H}(\{s\})} \tag{4}$$

The expectation value of the energy can be rewritten according to a derivation of the partition function:

$$U = -\frac{1}{\mathcal{Z}} \frac{\partial \mathcal{Z}}{\partial \beta} = -\frac{\partial \log(\mathcal{Z})}{\partial \beta} \tag{5}$$

It is further interesting to investigate the specific heat. It is a measure for the energy amount required to raise a system by a certain temperature. With the equation for the inner energy, it is possible to relate the specific heat to the partition function.

$$C = \frac{1}{V}\frac{\partial U}{\partial T} = -k\beta^2\frac{1}{V}\frac{\partial U}{\partial \beta} = k\beta^2\frac{1}{V}\frac{\partial^2 \log(\mathcal{Z})}{\partial \beta^2} \tag{6}$$

2.1.2 Fluctuations

Often times it becomes interesting to observe the fluctuations of the observables which can be written as:

$$\langle(\mathcal{O} - \langle\mathcal{O}\rangle)^2\rangle = \langle\mathcal{O}^2\rangle - \langle\mathcal{O}\rangle^2 \tag{7}$$

Analogous to (5), the inner energy which is also the expectation of the energy, can be viewed in terms of its fluctuation [6, p. 11]:

$$\langle E^2\rangle - \langle E\rangle^2 = \frac{1}{\mathcal{Z}}\frac{\partial^2 \mathcal{Z}}{\partial \beta^2} - \left(\frac{1}{\mathcal{Z}}\frac{\partial \mathcal{Z}}{\partial \beta}\right)^2 = \frac{\partial^2 \log(\mathcal{Z})}{\partial \beta^2} \tag{8}$$

When considering a constant volume V, the equation of the specific heat C can be described accordingly by inserting the previous result into (6):

$$C = k_B\beta^2\frac{\langle E^2\rangle - \langle E\rangle^2}{V} = k_B\beta^2 V(\langle e^2\rangle - \langle e\rangle^2) \tag{9}$$

Here $e = \frac{E}{V}$ is the energy per site, which is one of the main observable for the latter described Ising model, to be examined throughout this thesis. Similarly to the specific heat, there is also the magnetic susceptibility χ, which is given here [6, p. 17]:

$$\chi = \left.\frac{\partial\langle M\rangle}{\partial h}\right|_{h=0} = \beta(\langle M^2\rangle - \langle|M|\rangle^2) = \beta V^2(\langle m^2\rangle - \langle|m|\rangle^2) \tag{10}$$

M is the magnetization and $m = \frac{M}{V}$ the magnetization per site. The external magnetic field h is considered to be 0 in any case throughout this thesis. With no external magnetic field, a net magnetization for a system is occurring for temperatures under the critical temperature $T < T_c$. However with no spin direction being energetically favorable, over an infinite period of time, those directions would balance each other out. Also there is no magnetization for temperatures over the critical temperature $T > T_c$. When given those conditions, the expected magnetization would vanish in either case $\langle m\rangle = 0$. This does not apply for the absolute of the magnetization or when squared.

2.2 Ising Model

The Ising model, named after Ernst Ising, is a simple spin model for the analysis of magnetism. It consists of N spins in a system with a binary value: $s_i \in \{1, -1\}$. The net amount of spins pointing in the same direction of the system relates to the magnetization of this system. The Ising Model can be formulated for different numbers of dimensions. The spins are typically arranged on a completely filled lattice and the distance between each adjacent lattice point is identical and normalized to 1. Throughout this thesis it is considered for every spin to only interact with its nearest neighboring spins. Therefore for the 2D Ising model case and considering a squared lattice, a spin then interacts with 4 spins in total. For a cubic lattice it would be with 6 spins. Therefore for any case with dimension d, it is $2 \cdot d$. If a spin is in alignment with the neighboring spin, this correspond to a lower energy. It results in a higher energy for the opposite case. The Ising model is capable of showing phase transitions for dimensions higher than 2. The high temperature case correlates to a state of disorder or high energy and the low temperature one to a state of order, which results in a lower energy.

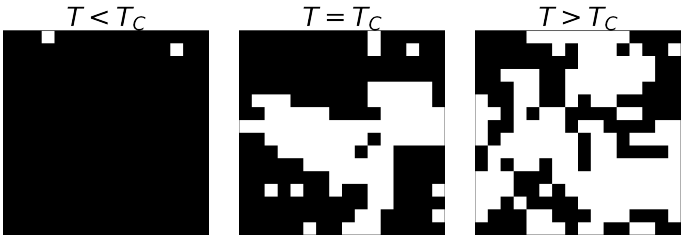


Fig. 2: Ising spin configurations in equilibrium simulated at different temperatures. From left to right: below the critical temperature, at the critical temperature, over the critical temperature.

2.2.1 Description

The spin configuration of a system is characterized by a Hamiltonian [7, p. 81], which is on the one hand described by a interaction term and on the other hand by a term from an external magnetic field:

$$\mathcal{H}(\{s\}) = -\sum_{\langle i,j \rangle} J_{i,j} s_i s_j - \mu \sum_i h_i s_i \tag{11}$$

The first sum describes the interaction term with $J_{i,j}$ being the interaction factor. A positive interaction factor $J_{i,j} > 0$ describes a ferromagnetic case, while the negative case $J_{i,j} < 0$ describes the anti-ferromagnetic case. The notation $\langle i, j \rangle$ indicates the summation over all nearest neighbor interaction pairs in the system. Periodic boundary conditions (pbc) are assumed throughout this thesis. Looking at the second sum, this corresponds to the interaction with an external magnetic field, where μ is the magnetic moment.

From now on the interaction parameter and the Boltzmann constant are set to a constant 1 and there is no external field assumed:

$$k_B = 1 \quad , \quad J_{i,j} = 1 \quad , \quad h_i = 0 \tag{12}$$

The simplifications in (12) for the constants do abbreviate the further observables which are going to be considered throughout this thesis. For one there is the energy E of a spin configuration and it is the interaction of the spins with the nearest neighbors. This is now equivalent to the Hamiltonian:

$$E = -\sum_{\langle i,j \rangle} s_i s_j \tag{13}$$

The magnetization M is the net orientation of the system and therefore the summation over every spin in a system:

$$M = \sum_i s_i \tag{14}$$

One more observable is the two point spin correlation $G(\boldsymbol{r})$. It signifies the correlation between two points on a lattice:

$$G(|\boldsymbol{r}|) = G(i,j) = \langle s_i s_j \rangle - \langle s_i \rangle \langle s_j \rangle \tag{15}$$

the distance between two spins is $\boldsymbol{r} = \boldsymbol{r}_i - \boldsymbol{r}_j$. For this observable, $\langle s \rangle$ corresponds to the expectation of the magnetization. As described in a previous chapter 2.1.2, this magnetization is 0 for symmetry reasons. Therefore the second term or subtrahend in this equation can be neglected. The first term would correspond to the energy for the case of a distance of 1. Therefore it is more useful to observe a distance larger than this.

2.2.2 Critical Exponents and Phase Transitions

Phase transitions are being talked about when changing from one phase to the other. In the realm of thermodynamics, this can be observed by a change of multiple parameters, such as temperature, pressure, volume, etc., beyond a certain ratio. Crossing this ratio results in a phase transition between one state (liquid, solid, gas) to another. This leads to a discontinuity according to one or more parameters.

For an equal or larger dimension than 2, a phase transition can be also observed for the Ising Model. The phase at low temperatures being an ordered phase (ferromagnetic), resulting in a total net magnetization and the other is the disordered phase (paramagnetic) at higher temperatures, showing no overall total magnetization. This transition happens at a critical temperature T_c and is of second order [6, p. 82 ff.]. The magnetization itself is continuous, but not differentiable at the critical temperature. A second order phase transition corresponds to a discontinuity in the first derivation to the temperature of the systems magnetization. Generally the order of the phase transition corresponds to the order of the derivation for the free energy to have a discontinuity.

When considering different temperatures of the Ising model, for low temperatures the spins tend very strongly to be aligned in the same direction. When coming closer but still staying below T_c, clusters of spins pointing in one direction can be observed. The length scale of those clusters is called the correlation length ξ. At the critical temperature this correlation length ξ diverges. And for temperatures above T_c, the system enters in a state of disorder and this correlation length is decreasing with larger temperatures. [6, p. 83-84]

For temperatures below the critical temperature ($T \leq T_c$), the correlation lengths scales as follows:

$$\xi \propto |t|^{-\nu} \tag{16}$$

Where the temperature is now given as the reduced temperature $t = \left|\frac{T-T_c}{T_c}\right|$ and ν is a critical exponent.

There are multiple of such relations to follow. Here it is only concentrated on a few of them:

$$C \propto |t|^{-\alpha} \quad , \quad M \propto t^{\beta} \quad , \quad \chi \propto |t|^{-\gamma} \tag{17}$$

The symbols α, β, γ correspond to more critical exponents.

For a finite sized system, however, the correlation length cannot diverge and is then limited by the system size. This also results in a minor shift of the divergences towards temperatures which are smaller than the critical temperature at an infinitely sized system. The smaller the system size is, the larger the shift [2, p. 214-216].

The correlation length is now bound to the system size and the previous correlation (16) can be written as:

$$|t| \propto \xi^{-1/\nu} \propto L^{-1/\nu} \tag{18}$$

The dependencies on the reduced temperature t in equation (17) can now be correlated to the system length at the critical temperature. This is called the finite-size scaling (FSS) and is given as:

$$C \propto L^{\alpha/\nu} \quad , \quad M \propto L^{-\beta/\nu} \quad , \quad \chi \propto L^{\gamma/\nu} \tag{19}$$

When now calculating the observables C, M or χ for different system sizes at the critical temperature, it is possible to obtain the critical exponents α, β and γ by examining the exponents. The ratio of one exponent towards ν can be calculated as the slope when taking the logarithms of the observable and the system size. This results in a graph with Y-axis: $\log(\mathcal{O})$ and X-axis: $\log(L)$. This should result in a linear graph where the slope corresponds to the inverse of the ratios which are given in (19).

The exact results for these critical exponents in the case of the 2D Ising model and the approximate results of the 3D case are shown in the following table [2, p. 215]:

Table 1: Critical exponents for the 2D and 3D Ising model.

Dimension	ν	α	β	γ
d = 2	1.000	0.000	0.125	1.75
d = 3	0.630	0.109	0.326	1.237

2.2.3 Analytic Solution for the Energy of the 1D Ising Model

The analytic solution for the energy can be easily derived by simplifying the energy when considering only the 1D case:

$$E = -\sum_i s_i s_{i+1} \tag{20}$$

Since pbc are applied, the following relation also applies: $s_{L+1} = s_0$.

When implementing this energy to the previously defined Hamiltonian (11) and the previously described setup in (12), the partition function from (2) is rewritten as:

$$\mathcal{Z} = \sum_{\{s_i\}} \prod_i \mathrm{e}^{\frac{1}{k_B T}(J_{i,i+1} s_i s_{i+1} + \mu h_i s_i)} \tag{21}$$

$$= \sum_{\{s_i = \pm 1\}} \prod_i \mathrm{e}^{\beta \cdot s_i s_{i+1}} \tag{22}$$

When using the expectation value (3) and once again the Hamiltonian with the previously chosen constants (12), one can derive this ensemble average for the energy:

$$\langle E \rangle = \frac{1}{\mathcal{Z}} \sum_{\{s\}} E \cdot \mathrm{e}^{-\beta \mathcal{H}} = -\frac{1}{\mathcal{Z}} \frac{\partial}{\partial \beta} \sum_{\{s\}} \mathrm{e}^{-\beta E} = -\frac{\partial}{\partial \beta} \log(\mathcal{Z}) \tag{23}$$

The exact solution [8, p. 260 ff.] for the partition function in the 1D Ising case for periodic boundary condition is:

$$\mathcal{Z} = 2^L \cdot (\cosh(\beta))^L + \sinh(\beta)^L) \tag{24}$$

And therefore the expected value for the energy can be solved as:

$$\langle E \rangle = -\frac{L \sinh(\beta) \cosh(\beta) \left(\sinh(\beta)^{L-2} + \cosh(\beta)^{L-2}\right)}{\sinh(\beta)^L + \cosh(\beta)^L} \tag{25}$$

2.2.4 1D Renormalization by Decimation

The RG theory allows to investigate the change of physical systems of different scales. In this work it will be used to reduce the system length by a factor of two for the purpose of two configurations with two system sizes to correspond to one another. The smaller system size is to be trained to match the original size. After the renormalization of the Ising model the temperature of the new system corresponds to a different temperature. The 1D Ising model is one of the few cases for which a renormalization by block spins decimation can be solved exactly [9, p. 703-705] [10].

For the renormalization procedure every second spin gets neglected and therefore the system size decreases by a factor of 2. This is pictured in Fig. 3 with the white circles disappearing.

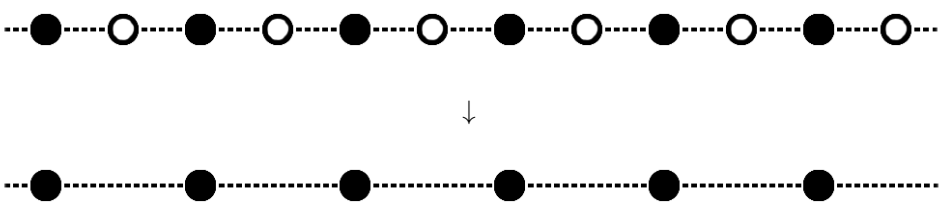


Fig. 3: Decimation of the 1D Ising model. The upper sketch shows the Ising configuration before the renormalization. Each circle stands for a spin +1 or −1, independent of its color. The white circles disappear after the renormalization in the lower sketch.

To derive an expression for the new partition function, the solution for the original partition equation (22) is solved for the even spins $(2i)$, pictured as the black circles in Fig. 3. The term for the odd spins $(2i+1)$, symbolizing the white circles, can be written-out and the entire term is then simplified to a hyperbolic cosine term. This is shown in the following conversion:

$$\sum_{\{s_i=\pm1\}} \prod_{i=1}^{N} \mathrm{e}^{\beta s_i s_{i+1}} = \sum_{\{s_i=\pm1\}} \prod_{i=1}^{N/2} \mathrm{e}^{\beta(s_{2i}s_{2i+1}+s_{2i+1}s_{2i+2})} \tag{26}$$

$$= \sum_{\{s_{2i}=\pm1\}} \prod_{2i=1}^{N/2} \mathrm{e}^{\beta(s_{2i}+s_{2i+2})} + \mathrm{e}^{-\beta(s_{2i}+s_{2i+2})} \tag{27}$$

$$= \sum_{\{s_{2i}=\pm1\}} \prod_{2i=1}^{N/2} 2\cosh(\beta(s_{2i}+s_{2i+2})) \tag{28}$$

$N = L^d$ is the number of sites in the lattice and $\{s_i = \pm1\}$ corresponds to every possible spin configuration. Analogous $\{s_{2i} = \pm1\}$ alternates only every second spin. In the next step the recursion relation for the inverse temperature β is shown. For this, the original partition function from equation (22) is taken into consideration once again. However this time it is applied to the renormalized system and therefore a renormalized inverse temperature $\tilde{\beta}$ is attained as well as the spin independent renormalization factor $z(\beta)$:

$$\mathcal{Z} = \sum_{\{s_{2i}=\pm1\}} \prod_{2i} z(\beta) \cdot \mathrm{e}^{\tilde{\beta} s_{2i} s_{2i+2}} \tag{29}$$

$$= [z(\beta)]^{N/2} \cdot \sum_{\{s_{2i}=\pm1\}} \prod_{2i} \mathrm{e}^{\tilde{\beta} s_{2i} s_{2i+2}} \tag{30}$$

For consistency reasons between equation (28) and (30), the following equation is required:

$$2 \cdot \cosh(\beta \cdot (s_{2i} + s_{2i+2})) = z(k) \cdot \mathrm{e}^{\tilde{\beta} s_{2i} s_{2i+2}} \tag{31}$$

When solving this equation, every spin is able to assume a value of +1 or −1 and therefore there are 4 equations to be solved. Since for the left side of the equation cosh is a symmetrical function and for the right side there are the both spins multiplied with each other, those 4 equations are resolved to 2 equations.

$$\left.\begin{array}{lll} s_{2i} = +1 & , & s_{2i+2} = +1 \\ s_{2i} = -1 & , & s_{2i+2} = -1 \end{array}\right\} \quad z(\beta)\mathrm{e}^{\tilde{\beta}} = 2 \cdot \cosh(2\beta) \tag{32}$$

$$\left.\begin{array}{lll} s_{2i} = +1 & , & s_{2i+2} = -1 \end{array}\right\} \quad z(\beta)\mathrm{e}^{-\tilde{\beta}} = 2 \tag{33}$$

This set of equations can be transformed to:

$$\mathrm{e}^{2\tilde{\beta}} = \cosh(2\beta) \tag{34}$$

$$z^2 = 4\cosh(2\beta) \tag{35}$$

Now when rearranging equation (34) there is the recursion relation for the inverse temperature $\tilde{\beta}$:

$$\tilde{\beta} = \frac{1}{2}\ln(\cosh(2\beta)) \tag{36}$$

It becomes possible to state the temperature under which the Hamiltonian is self-similar. The more interesting case for this thesis is when solving the previous equation for the original β:

$$\beta = \frac{1}{2}\mathrm{arccosh}(e^{2\tilde{\beta}}) \tag{37}$$

It is now shown that for when applying the future described $\mathcal{SR}$ method, how the temperature changes when resolving to larger system sizes. However this solution does only apply for the 1D case and for any larger dimension, it is not yet known how to find such a relation analytically. Once the methods for the statistical computer simulation for solving the Ising model is introduced, as well as the method for resolving to larger system sizes is explained, a numeric solution to attain an analogous temperature transformation will be presented in section 2.5.4.

2.3 Monte Carlo Simulation

MC simulations are probabilistic simulations used in various fields to obtain a numeric value based on probability analysis. It first was developed by a group led by Nicholas Metropolis within the Manhattan project [11] and named for the well known Monte Carlo casinos. In this sense MC models have been used to predict the risk when playing a gambling game [12]. The same application can be found in the finance world [13]. However the range of utilization are much broader with further examples being the mathematical calculation of integrals [14] or the biological study of proteins [15].

To calculate the observables of the Ising model, one could theoretically calculate the summation over the entire partition function. This however requires considering an amount of 2^N possible configurations with $N = L^d$ being the number of sites in a lattice [7, p. 7]. This problem for the exact solution gets targeted instead by generating spin configurations according to their own probability distribution and is called importance sampling. In this work the MC method will be used to create stochastic random states from an arbitrary initial state which are based on the concept of Markov chains.

There are many methods that count as valid MC simulations. However this is not the focus of this work and therefore the simple Metropolis algorithm is introduced. It is a method with local updates and functions very well in the realm of high temperatures. The Metropolis algorithm is used to generate spin configurations for the neural network to train the super-resolution procedure and also to create an initial set of spin configuration which gets resolved to larger system sizes. In addition the HB algorithm is also introduced, however for a different reason. It has the property to suggest a spin value according to its neighbors. Therefore this method can actually be used as a reference method for the super-resolution procedure in the 1D case.

2.3.1 Markov Chains

In theory, over a infinitely large amount of time, a system would occupy every single possible spin configuration for $T > 0$. This is due to the probability for such a configuration following the Boltzmann distribution shown in (1). Some of the configurations are not very likely to occur and the computer simulation of this problem considers only a finite amount of time. Therefore importance sampling is used to sample configurations according to the Boltzmann weight with the help of a Markov chain.

A Markov chain, here technically a Monte Carlo Markov Chain (MCMC), is a method in which a stream of configurations can be generated. Every next configuration in this stream is based on a probability that does only depend on the state from its previous configuration. To suggest a move from one configuration to the next, it is enough for the probability to fulfill the detailed balance condition [16, p. 146 ff.]:

$$W_{A\to B} P_A = W_{B\to A} P_B \tag{38}$$

With $W_{A\to B} \geq 0$ being the transition probability to enter from one state to the next state. All probabilities to enter any next state B from the current state A have to be normalized. Therefore the following condition also applies: $\sum_B W_{A\to B} = 1$. The Probability of the system P_A had already been defined in equation (1).
Now it is possible to measure a quantity N of a configuration $\{s_i\}$ created with the MCMC. Then calculate the desired observable $\mathcal{O}$ for each of them. The average for all of the calculated observables is then defined according to the following equation:

$$\overline{\mathcal{O}} = \frac{1}{N} \sum_{i=1}^{N} \mathcal{O}(\{s_i\}) \tag{39}$$

The average of the observable $\overline{\mathcal{O}}$ is an estimator which can be used to be equated to the expectation value $\langle \mathcal{O} \rangle$ which had been stated in equation (3). This enables to calculate the expectation of the observable numerically and therefore it is not required to be solved analytically. For the condition $\lim_{N\to\infty}$ this estimator would be an exact solution.

2.3.2 Metropolis Algorithm

Metropolis is a common MC algorithm [6, p. 49 ff.] which realizes the detailed balance condition (38) within an update step. The algorithm picks a spin in the configuration at random and suggests to flip it with a probability which depends on the energy difference from the current state to the next state:

$$P(\Delta E) = \begin{cases} 1 & \text{if } \Delta E < 0 \\ \mathrm{e}^{-\beta \Delta E} & \text{if } \Delta E \geq 0 \end{cases} \tag{40}$$

$$\Delta E = E(\{s_B\}) - E(\{s_A\}) \tag{41}$$

For implementing this algorithm, one defines an arbitrary initial spin configuration. For example either by every spin being randomly up or down (hot start) or every spin pointing in the same direction (cold start). Up or down are categorized by either +1 or −1, optionally by 1 and 0. Then one chooses a random spin on the lattice to flip and calculates the probability for this to happen according to the energy difference of the two systems. To check whether this move is accepted or not, one draws a random number between 0 and 1 and compares this to the calculated probability. This is repeated according to an statistically sufficient amount or according to the capabilities of the computer. After a couple hundreds or thousands of sweeps, one starts to track the desired observables to the method in (39). A sweep correlates to suggest a amount of flip spins equal to the amount of spins N in the lattice.

2.3.3 Heat Bath Algorithm

The heat bath (HB) algorithm determines the direction of a chosen spin according of its neighbors. For the Ising model it allocates a spin according to the probability of both possible spin directions. It is simplified for this case, since only two cases need to be considered. For the probability of being in either direction, it is described as:

$$P(\{s_i\}_{old} \to \{s_i\}_{new}) = \frac{\mathrm{e}^{-\beta \Delta E/2}}{\mathrm{e}^{\beta \Delta E/2} + \mathrm{e}^{-\beta \Delta E/2}} \tag{42}$$

This also satisfies the detailed balance condition [7, p. 88].

The HB method does not depend on the targeted spin in the lattice to be either up or down, since it does only depend on the final result and the interaction with the neighboring spins. This is useful for a later alternative method to resolve to larger system sizes when looking at the 1D case and it shown to do a similar job to the neural network. However for larger dimensions than 1, the later presented method does not work. For the 1D case in the later described methodology of training the network, every unknown spin has two known neighbors. For a larger dimension there is a greater information loss and no exact information on any spins are given. The HB algorithm is therefore insufficient for the super-resolving methodology at dimensions larger than 1.

2.3.4 Monte Carlo Renormalization Group

The RG theory in this thesis is used in section 2.2.4 to obtain a temperature transformation for the rescaling of the system size in the 1D case. However in computational physics, the RG is also a very useful tool for the analysis of critical behavior with Monte Carlo simulations. This is now a brief introduction to this field of research, since the problems there help in the understanding of super-resolving to larger system sizes. However this work is not a direct link to this field of Monte Carlo renormalization group (MCRG) research.

The RG theory in combination with the MC simulations is used for the analysis of phase transitions. They prove to be very efficient and in good agreement with the calculations of other methods but it is difficult to judge the accuracy of such a calculation without a comparison with other known to work methods [17, p. 964].

Generally the MCRG method runs a MC simulation and transforms the spin configurations with a blocking method to a new system with fewer degrees of freedom. Each block contains b^d spins and therefore reducing the one dimensional system length by a factor b. A common way of blocking is the majority rule, this determines the resulting block spin to either +1 or -1, depending on the majority of its spins pointing in this direction. For the sum of spins in a block pointing in both directions for the same amount, this can then be resolved by either randomly choosing a spin direction or with a deterministic choice for choosing the first spin direction when iterating for the entire block.

A common use for this methodology of using the RG, is for the calculation of the critical temperature [6, p. 242-244]. It is known that the correlation length varies with temperature and the correlation length diverges at the critical temperature. For the down-scaled configuration it is therefore assumed to be typical for a Ising model at another temperature $\tilde{T}$. Meaning the blocked spin configuration corresponds to the same Boltzmann probabilities as if it was generated with the same method used to create the original configuration. This is for the appropriately transformed temperature. However for the critical point, the original temperature and the temperature for the blocked configuration is supposed to be the same. This comes from the fact that half of an infinitely large correlation length at the critical temperature, is still going to be infinite at the decimated system spin configuration. Half of infinity is still infinity. For calculating the critical temperature according to the RG methodology, an observable which is independent of the system size but dependent on the temperature, like the internal energy per spin u, is calculated. For the blocked system, the analogous observable $\tilde{u}$ is calculated as well. Then the temperature is reconstructed where $u \stackrel{!}{=} \tilde{u}$, which corresponds to the critical temperature.

Further methods show how to calculate the critical exponents [17, p. 967 ff.]. Also different transformations other than the majority rule are researched to give different and possibly better results for different cases [18, p. 3607 ff.]. Even up to this day there is research for different transformations, which is also presented in the latter chapter 2.5.7. The MCRG method is also applicable for longer than nearest neighbor range interactions [19] or other models than the Ising model, such as the XY Model [20]. As well as many more.

2.4 Neural Networks in Machine Learning

Neural networks are an optimization model which belong to the umbrella term of machine learning. It is inspired by the biological brain in which neurons are connected to each other to process information. In multiple layers of neural networks our brain is able to make sense of a picture which is projected on the photoreceptor cells of our eyes. In this sense it enables us humans to differentiate a picture of Picasso to a picture of Van Gogh. Or in a more simpler sense, a cat from a dog.

Inspired by this biological neural network, a neural network on the computer can also be trained to detect for example a cat or a dog in a given picture [21]. Multiple pictures are passed to the algorithm alongside with the correct categorization. Then the neural network is trained in such a way that when given a picture in the input layer, this picture will trigger multiple neurons throughout multiple layers in the hidden layers to come out in a final output layer with the desired interpretation. This process is called supervised learning since the network is trained on data which is already categorized. After this training phase, the neural network can be exposed to a picture which had not yet been classified and it is the task of such a neural network to do the classification of the picture. The neural network in this work is trained with a different purpose. It is more oriented on the enhancement of imagine resolution. With the difference being, instead of a classification into different categories, the output of this network would be an enhancement of the input itself. To archive this, a particular neural network setup called 'convolutional neural networks' is used. The inspiration for this thesis comes from the super-resolution of images with CNNs, seen here:

Fig. 4: [22] Left is a low resolution image. Right the same image super-resoluted with a CNN.

Research similar to the image super-resolution procedure is now having a great breakthrough in the video gaming industry. The company Nvidia is currently developing on the technology Deep learning super sampling (DLSS) [23]. In it, the model is trained offline on a game beforehand to reconstruct images generated with a lower quality to match those of higher quality. For example a game running at 1920×1080 could be efficiently upscaled to match the quality of a 3840×2160 resolution. DLSS therefore improves speed by rendering less pixels and cutting off the expensive effects such are ray tracing, lighting and more. The more expensive the rendering techniques that can be bypassed, the more benefit this method is giving to the performance. Since the version 2.0 of DLSS, it has become a generalized model, meaning it is supposed to be applicable for multiple games.

2.4.1 Neural Network Layers

A neural network consists of multiple layers of neurons where every neuron can be viewed as a place holder for a number which depends on a function. Here a layer is an array of these neurons [24, p. 5 ff.]. The first layer is the input layer. It contains the information of the data which needs to be interpreted. The information gets then passed onto the hidden layers. The hidden layers in such a neural network do not necessarily have an apparent meaning for a human interpretation [25, p. 55-57] but it is where the neural network takes some steps of transforming for the data. In the final step there is the output layer. It is just like the other layers, an array with numbers, but this time it is required to interpret this array. For a network which only needs to distinguish between a cat or a dog, a final layer with one neuron would be sufficient. It then could be interpreted as a 1 or a 0, which could be either cat or dog. In the case of this work, the final layer will have twice the length of the initial input, therefore resolving the initial spin configuration by a factor of 2^d with d being the dimension.

2.4.2 Shallow vs Deep Neural Network

Shallow neural networks are networks which consist of only one hidden layer, whereas deep neural networks consist of more than one. For a simple fully connected neural network, even for a shallow one, it had already been shown to be able to accurately approximate any Borel measurable function. This is given for a sufficiently large amount of neurons [26, p. 1 ff.] and called the *universal approximation theorem*. Generally it is unknown for any neural network problem to decide how many neurons are required for the appropriate learning. However for a shallow neural network this amount of required neurons can be exponentially growing with task complexity. Moreover a shallow network is sensitive to overfitting and a training algorithm might have difficulties to find the correct solution for a given problem. [24, p. 194-197]

Empirically it is easy to show that a deep neural network is easier to train than a shallow network and does not suffer as greatly from the problems previously described [27, p. 1 ff.].

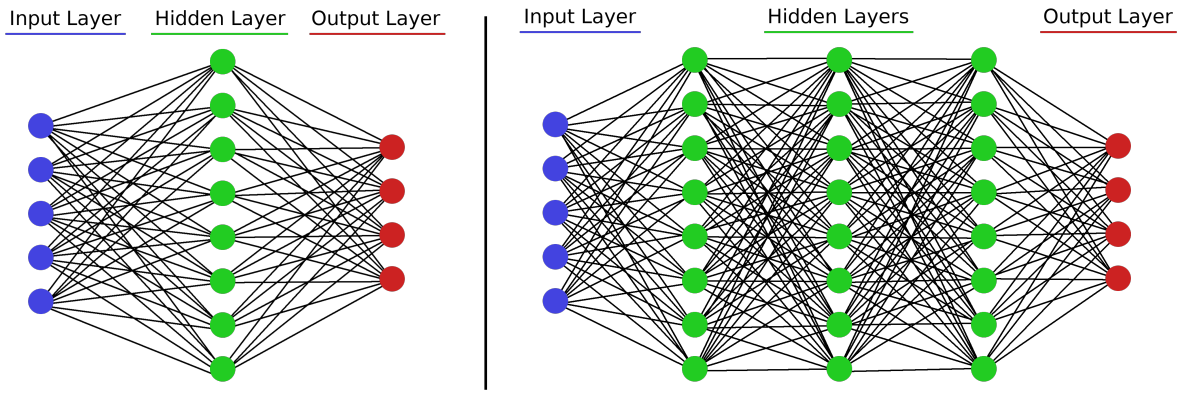


Fig. 5: Display of a shallow (left) and a deep (right) fully connected neural neural network. There are three types of layers, one input layer (blue), one or multiple hidden layers (green) and one output layer (red). Each neuron in each layer is connected with each neuron from the neighboring layers.

2.4.3 Neural Network Connections

There are many ways for the neurons to connect to each other. This work will be only using CNNs. However as an explanatory example it is useful to look at a fully connected neural network as well. Here every neuron in each layer is connected with all neurons in the next layer, shown as in Fig. 6. Speaking in mathematical terms, this is nothing more than a matrix multiplication, seen as in equation 43. With each of the layers being an array of functions. Each function is dependent on the previous layer and represents the value of a neuron. The connection between the neurons is the weighting matrix. [28, p. 581-583]

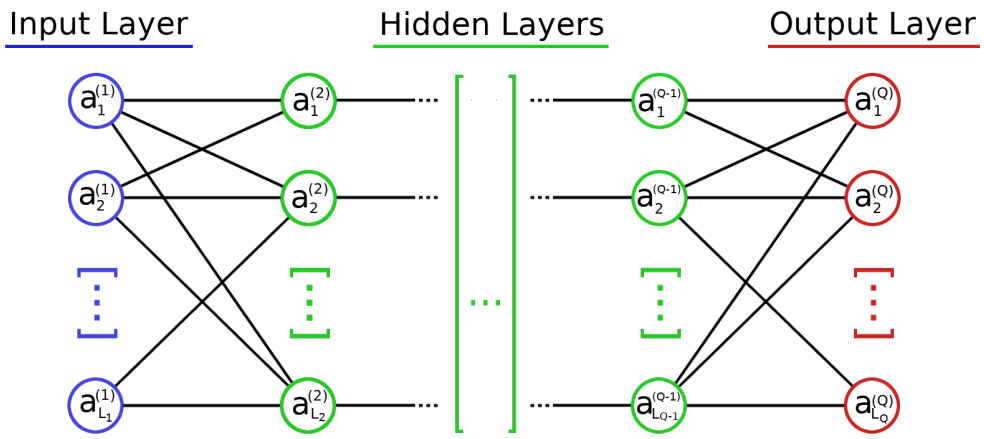


Fig. 6: Fully connected neural network. There is one input layer (blue), at least one but often more hidden layers (green) and finally one output layer (red). Every ith layer contains an amount L_i of neurons which are connected with every neuron in the next layer.

$$\begin{bmatrix} a_1^{(i)} \\ a_2^{(i)} \\ \vdots \\ a_{L_i^{(i)}} \end{bmatrix} = \sigma \left(\begin{bmatrix} w_{1,1}^{(i,i-1)} & w_{1,2}^{(i,i-1)} & \dots & w_{1,L_{i-1}}^{(i,i-1)} \\ w_{2,1}^{(i,i-1)} & w_{2,2}^{(i,i-1)} & \dots & w_{2,L_{i-1}}^{(i,i-1)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{L_i,1}^{(i,i-1)} & w_{L_i,2}^{(i,i-1)} & \dots & w_{L_i,L_{i-1}}^{(i,i-1)} \end{bmatrix} \begin{bmatrix} a_1^{(i-1)} \\ a_2^{(i-1)} \\ \vdots \\ a_{L_{i-1}}^{(i-1)} \end{bmatrix} + \begin{bmatrix} b_1^{(i,i-1)} \\ b_2^{(i,i-1)} \\ \vdots \\ b_{L_i}^{(i,i-1)} \end{bmatrix} \right) \tag{43}$$

$$\boldsymbol{a}^{(i)} = \sigma(\boldsymbol{w}^{(i,i-1)} \boldsymbol{a}^{(i-1)} + \boldsymbol{b}^{(i,i-1)}) \tag{44}$$

Here is the matrix multiplication in detail for any layer at position $i > 1$ in a fully connected neural network as shown in Fig. 6. Any layer after the first input layer can be calculated by the weighting matrix $\boldsymbol{w}^{(i,i-1)}$ multiplied with the previous layer. To this, a bias is added $\boldsymbol{b}^{(\boldsymbol{i,i-1})}$ which is useful for manipulating the activation function σ. This procedure repeats for every neighboring two layers until and including the final output layer. The first layer is for a 1D case the input of the data, but in most cases normalized between 0 and 1. For a higher dimension, like for a 2D picture, it is required to make a 1D projection.

2.4.4 Activation Function

There are multiple varieties of activation functions with different applications and for different reasons. The main purpose of such a activation function is to introduce non-linearity to the neural network. Without any activation function all the different hidden layers would be able to be compressible to a single layer and therefore not necessarily sufficient enough for solving non-linear problems. An important attribute for the activation function is the differentiability. Differentiation plays a role in the learning process when using a method called gradient descent.

The most common activation function for the hidden layers is the Rectified Linear Unit (ReLU) function:

$$\sigma_{ReLU}(x) = max(0, x) \tag{45}$$

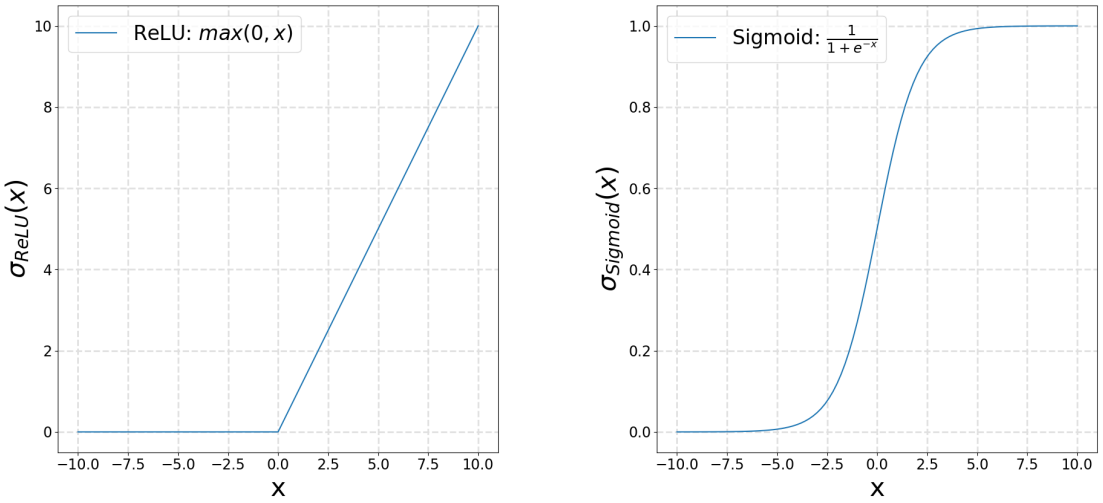


Fig. 7: Left a ReLU ist shown and right a sigmoid function. The variable x is from -10 to 10. For the Relu it takes values between 0 and 10 in this range and the sigmoid from 0 and 1.

The ReLU function is very simple since it is just a linear mapping of the same function for positive values and 0 for negative values. This function dominates the state of the art and had also been empirically shown [29, p. 3 ff.] to be very powerful and does not require any additional expensive operations to be calculated. Even though the function is not differentiable at $x = 0$, it is not a problem since it is not common to land on a value of exactly 0.

The output layer serves a different purpose and requires a number span which is defined in a range of $-\infty$ to $+\infty$ to be squished to a scale from 0 to 1. This is then interpreted as the probability for a certain spin in the configuration to be either up or down. The sigmoid function comes in handy for this case [30, p. 580]:

$$\sigma_{\text{sigmoid}}(x) = \frac{1}{1 + \mathrm{e}^{-x}} \tag{46}$$

2.4.5 Dying ReLU Problem and ELU Function

The ReLU function is now a default activation function for neural networks because of its effectiveness. However this function can be fragile and suffer under what is called the 'dying ReLU' problem. Since the activation function is zero for all non-negative arguments, the slope therefore is also zero. A neuron can get stuck in training in this position due to the wrong initialization of biases [31] or a large gradient flow through a ReLU neuron. This problem is less likely to occur for a low learning rate however it can also be addressed by implementing a small slope on the negative side. This is here going to be addressed by a modified ReLU function, called the Exponential Linear Unit (ELU) function. It uses an exponential function for negative values, shown in Fig. 8: .

$$\sigma_{ELU}(x) = \begin{cases} x & \text{if } x \geq 0 \\ e^x - 1 & \text{if } x < 0 \end{cases} \tag{47}$$

The ELU activation function can not only prevent the dying ReLU problem but it has also been shown to speed up the learning process and increase the accuracy. [32, p. 1 ff.]

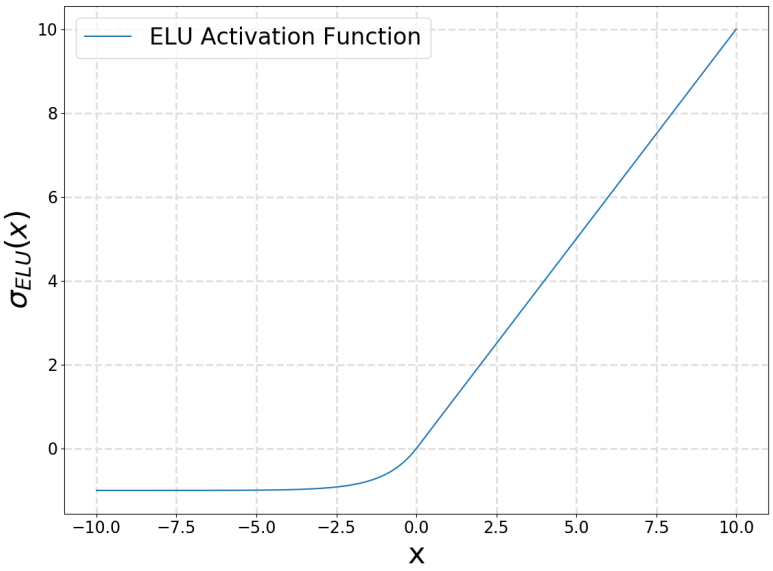


Fig. 8: Exponential Linear Unit activation function of a variable x shown in the range of −10 to +10.

2.4.6 1D Convolutional Neural Networks

A CNN is made up of one or more convolutional layers. A convolutional layer consists of a kernel which sweeps through the input of the layer. In this case the neurons are not fully connected with the next layer but are influenced by a filter that is shared amongst the neurons of this layer. This filter or kernel corresponds to the weight matrix in the fully connected layer which can be then trained similarly [33, p. 169 ff.]. The advantage of a fully CNN is that the kernel does not correspond to a fixed input size. In a fully CNN, the weights and biases are trained with a number of connections which is initiated by the size of the input. The weighted matrix in the convolutional layer is independent of such a initialization and can be used on any system size since it behaves like a filter [4, p. 3]. Since the goal is to increase the system size, this attribute is the essential reason for using a CNN.

The CNN used in this work consists only of convolutional operators. Such an operator for the 1D case consists of a $\mathrm{L}_x^{(i-1,i)}$ sized kernel between a layer on position i and the previous position $i-1$. The kernel sweeps over the entire lattice and here the lattice is extended by a padding to ensure a consistent size of the input and output layer. Most neural network libraries apply the padding on the end of the input, because the network starts with the very first entry of the input. The padding used in this thesis will apply the pbc, so it is consistent with the conditions of the Ising model. A pbc padding is however not typically implemented in most libraries and therefore in the implementation, the initial input itself is extended according to the pbc, before being passed to the neural network.

The size of the padding $\mathrm{L}_{\mathrm{padding}}$ is equal to the sum over all kernel lengths $\mathrm{N}_{\mathrm{kernels}}$ with the corresponding length of the kernel.

$$\mathrm{L}_{padding} = \sum_{i=1}^{\mathrm{N}_{\mathrm{kernels}}} \mathrm{L}_x^{(i-1,i)} - 1 \tag{48}$$

This equation also applies to larger dimensions, if the kernel has the same length for each dimension. This is the case throughout this thesis.

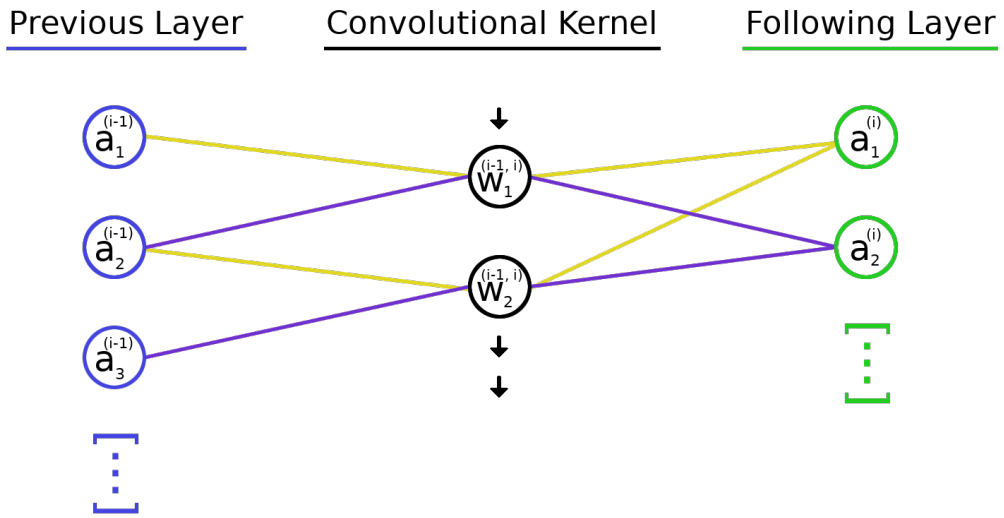


Fig. 9: Visualisation of the convolutional operation in the 1D case. A convolutional kernel of length 2 sweeps through a previous layer and reduces the system size by 1.

A layer $\boldsymbol{a}^{(i)} = [a_1^{(i)}, a_2^{(i)} \ldots, a_{L_k^{(i-1,i)}}^{(i)}]^T$ other than the input layer can be calculated by a convolutional operation of the previous layer $\boldsymbol{a}^{(i-1)}$:

$$a_x^{(i)} = \sigma \left[\left(\sum_{j=0}^{\mathrm{L}_x^{(i-1,i)}-1} w_{1+j}^{(i,i-1)} a_{x+j}^{(i-1)} \right) + b^{(i,i-1)} \right] \tag{49}$$

With $\boldsymbol{w}$ being weighting factors of the kernel. $\boldsymbol{b}$ is a bias and σ is the activation function. The latter is typically a sigmoid function for the final output and a ELU function for the other layers. Since the sigmoid function in the output layer serves the function of mapping a number range of $-\infty$ to $+\infty$ to a scale from 0 to 1. This is then treated as the probability P of a spin to be in a certain position, either up or down.

2.4.7 2D Convolutional Neural Networks

The 2D CNN is implemented analogous to the previous 1D case, just with an additional dimension. The kernel is therefore also in 2D and in the most cases quadratic. The padding is now applied to both dimensional directions.

An example of a 2D convolutional layer is seen in Fig. 10. There is a 2 dimensional 4×4 configuration with the pbc padding (in hatched), making it to a 6×6 configuration. There are two convolutional operations, both with kernel size 2×2, resulting in a padding of size 2 in both directions. The kernel sweeps with a step size 1 over the entire configuration. Every convolutional operation reduces the padding length by 1.

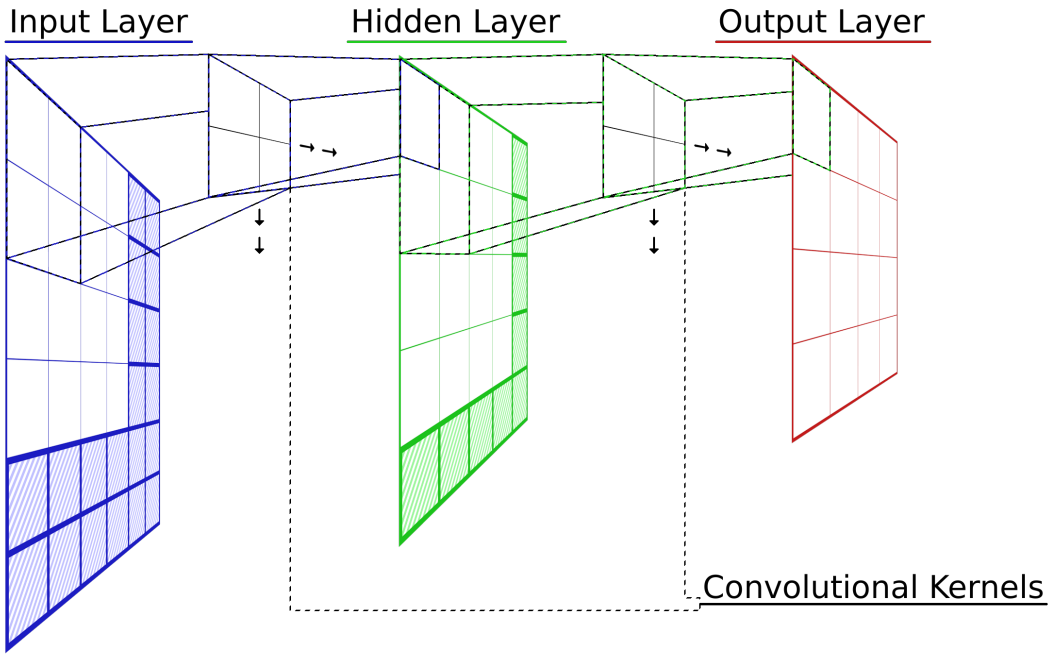


Fig. 10: Visualisation of two convolutional operations in 2D with one hidden layer. The kernel size is in both operation 2×2. The bottom and right edges are extended with a periodic boundary condition padding. Under such conditions this padding size is reduced by 1 for every operation.

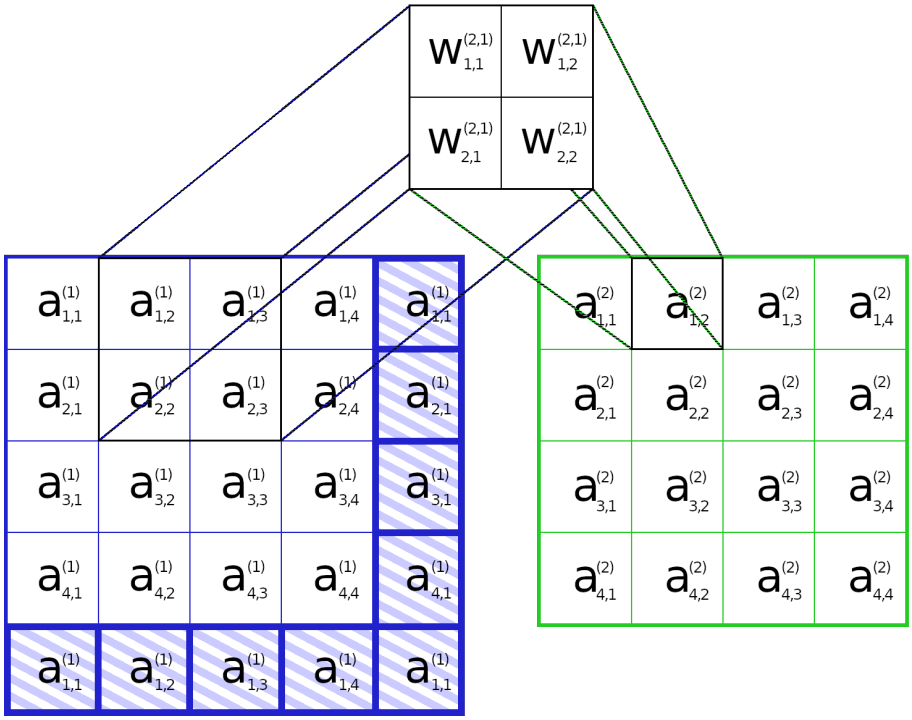


Fig. 11: Visualisation of a CNN operation in 2D. The kernel size is 2×2. The bottom and right edges of the left configuration is extended with a periodic boundary condition padding. This padding is consumed under such condition.

When assuming a kernel size of 2×2, the convolutional operation in Fig. 11 can be written as:

$$a^{(i)}_{y,x} = \sigma(w^{(i,i-1)}_{1,1} a^{(i-1)}_{y,x} + w^{(i,i-1)}_{1,2} a^{(i-1)}_{y,x+1} + w^{(i,i-1)}_{2,1} a^{(i-1)}_{y+1,x} + w^{(i,i-1)}_{2,2} a^{(i-1)}_{y+1,x+1} + b^{(i,i-1)}) \tag{50}$$

For periodic boundary conditions the case in Fig. (11), $a^{(1)}_{y,5}$ would correspond to $a^{(1)}_{y,1}$. Here x corresponds to the index of the axis in the first dimension of the data and y to the axis of the second dimension. The previous equation can be generalized to any size of kernel with:

$$a^{(i)}_{y,x} = \sigma\left[\left(\sum_{j=0}^{\mathrm{L}_y^{(i,i-1)}-1} \sum_{k=0}^{\mathrm{L}_x^{(i,i-1)}-1} w^{(i,i-1)}_{1+j,1+k} a^{(i-1)}_{y+j,x+k}\right) + b^{(i,i-1)}\right] \tag{51}$$

$\mathrm{L}_x^{(i,i-1)}$ is the kernel size in x-direction, $\mathrm{L}_y^{(i,i-1)}$ is the kernel size in y-direction. As previously mentioned they are going to be equally sized.

2.4.8 3D Convolutional Neural Networks

The three dimensional CNN is analogous to the 2D CNN, just with one additional dimension. The kernel is therefore extended with the length $\mathrm{L}_z^{(i,i-1)}$. The output of the neurons can therefore be computed similarly to (52):

$$a^{(i)}_{z,y,x} = \sigma\left[\left(\sum_{j=0}^{\mathrm{L}_z^{(i,i-1)}-1} \sum_{k=0}^{\mathrm{L}_y^{(i,i-1)}-1} \sum_{l=0}^{\mathrm{L}_x^{(i,i-1)}-1} w^{(i,i-1)}_{1+j,1+k,1+l} a^{(i-1)}_{z+j,y+k,x+l}\right) + b^{(i,i-1)}\right] \tag{52}$$

2.4.9 Cost Function

For a training algorithm to take place, there is a measurement required which distinguishes a good choice of parameters for the weights and biases from bad one. This measurement is called a cost function. Mathematically speaking it calculates the error as a function of the deviation from the prediction to the expectation. The loss function is then required to be minimized by the algorithm. A common minimization process is called gradient descent which requires the function to be differentiable. The cost function in this work is a cross-entropy loss [33, p. 62 ff.] function:

$$\mathcal{L}(\{\boldsymbol{s}\}, \{\boldsymbol{P}\}) = -\sum_{i=0}^{L} (\boldsymbol{s}_i \cdot ln(\boldsymbol{P}_i) + (1 - \boldsymbol{s}_i) \cdot ln(1 - \boldsymbol{P}_i)) \tag{53}$$

$\mathcal{L}(\{\boldsymbol{s}\})(\{\boldsymbol{P}\})$ is the cost, or loss. $\{\boldsymbol{s}\}$ is the set of expected spin configuration and $\{\boldsymbol{P}\}$ are the predicted probabilities from the output layer. In this case the spin s_i is a still a binary value but instead of taking the numbers -1 and +1 it is converted to 0 and 1. The probability is $0 \leq P_i \leq 1$. The Loss function diverges for a prediction far away from the expectation. One summand ($-(s_i \cdot ln(P_i) + (1 - s_i) \cdot ln(1 - P_i))$) of the sum is shown in Fig. 12.

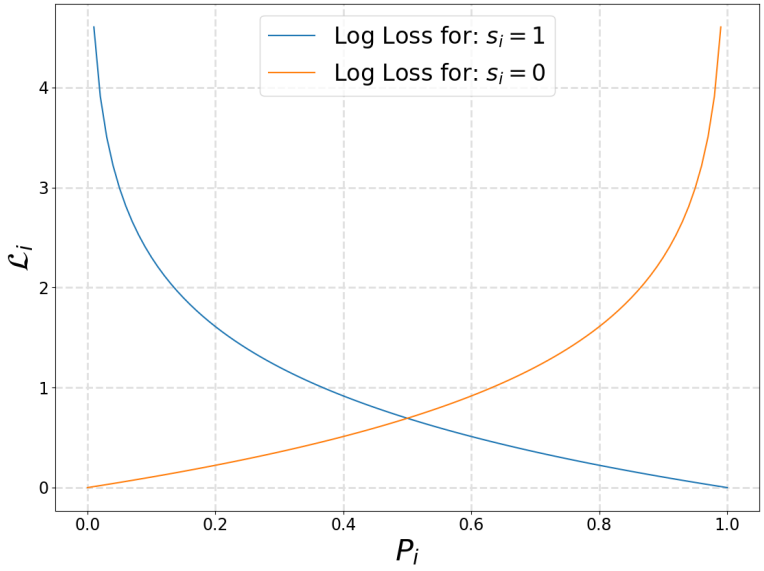


Fig. 12: One summand of the Loss function in equation (54) as a function of the probability. The blue curve is the case for s_i equals to 1 and the orange curve for s_i equals to 0 (analogous to -1 in terms of spin direction).

When training, the network is making a step towards minimizing the loss function. However it is training on multiple configurations and for efficiency purposes it is also useful to divide all the training configuration into a set of batches. The size of the batch is called N_{batch}. For such a batch the network is taking the average of the cost function and making a step to minimize this averaged cost function:

$$\mathcal{L}_{batch} = \frac{1}{N_{batch}} \sum_{i=1}^{N_{batch}} \mathcal{L}_i \tag{54}$$

2.4.10 Backpropagation

Backpropagation is a very common algorithm for supervised machine learning. The algorithm efficiently calculates the gradient of the loss function in terms of its weights and biases. Since all the layers are interconnected mathematically, it is possible to use the chain rule and compute each layer at the same time, starting with the last layer going to the first. This is where the term backpropagation is coming from. The computation cost is therefore about the same as with the forward pass of the data [33, p. 52 ff.].

$$\nabla\mathcal{L} = \begin{pmatrix} \frac{\partial\mathcal{L}}{\partial b^{(2,1)}} \\ \frac{\partial\mathcal{L}}{\partial w_1^{(2,1)}} \\ \frac{\partial\mathcal{L}}{\partial w_2^{(2,1)}} \\ \vdots \\ \frac{\partial\mathcal{L}}{\partial w_{V_k-1}^{(Q,Q-1)}} \\ \frac{\partial\mathcal{L}}{\partial w_{V_k}^{(Q,Q-1)}} \end{pmatrix} \tag{55}$$

With this gradient calculated it is then possible to make a step into the direction of the minimum of the loss function:

$$w_j'^{(i,i-1)} = w_j^{(i,i-1)} - \eta\frac{\partial\mathcal{L}}{\partial w_j^{(i,i-1)}} \tag{56}$$

$$b'^{(i,i-1)} = b^{(i,i-1)} - \eta\frac{\partial\mathcal{L}}{\partial b^{(i,i-1)}} \tag{57}$$

$w_j'^{(i,i-1)}$ is the weighted matrix at position j, between layer i and i-1 and $b'^{(i,i-1)}$ is the updated bias. η is called the step size, also called the learning rate. It controls how fast the update will take a step towards the gradient. A method to speed up the learning process and to evade to land in a local minimum is called the stochastic gradient descent (SGD) [34, 21 ff.]. It takes all of the training data and averages it to a stochastical approximation of the previous gradient descend method. This way it is a less accurate method to get to a minimum but the movement from each step fluctuates much more and faster. A better approach which is also referred to as SGD is to divide the training data sets into mini-batches and then do the approximation with the mini-batches instead of the entire data.

2.4.11 Adam Optimizer

The state of the art in neural network learning is to use an extension to the SGD which is called the Adaptive Momentum Estimator (Adam) optimizer. This optimization algorithm implements many different layers of complexity in optimization. It adapts the learning rate η individually for different parameters with a concept similar to momentum in physics. The Adam uses an estimation of the first and second moment for the learning rate to adapt [35]. For this it requires the gradient g of the function f at an iteration step, corresponding to a time t.

$$g_t = \Delta f_t(\theta_{t-1}) \tag{58}$$

The observable g_t corresponds here to the gradient of the cost function in a mini-batch. The first moment is then called the mean m and the second moment $n = 2$ is the uncentered variance v. To estimate the moments in equation (58), the algorithm calculates the exponentially moving average on the gradient of the loss function:

$$m_t = \beta_{1d} m_{t-1} + (1 - \beta_1) \cdot g_t \tag{59}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) \cdot g_t^2 \tag{60}$$

β_1 and β_2 are the exponential decay rates for the moment estimates, which oftentimes are also introduced as the forgetting factors for the first and second moment of the gradients. Most commonly they are set to 0.9 and 0.999. The time t refers to the iteration of the optimizer step. These estimators however do have a bias for small t, since for the time step $t = 0$ the momentum and variance estimator is initialized with 0. This leads to the bias correction:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{61}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{62}$$

Now it is possible to formulate the next update step for the weight:

$$w_t = w_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{63}$$

ϵ is a small scalar, typically 10^{-8} to prevent zero division. The update is analogous for the bias.

2.4.12 Overfitting

Overfitting is a modelling mistake which results by a model to incooperate too many details of a given dataset. The model is then incapable of generalizing and therefore less applicable for a dataset yet unseen by the model. The law of parsimony (also: Occam's razor) is often referred to when addressing overfitting [36, p. 1] . It calls upon to only use the most necessary means for solving a given problem. The fast growing complexity with the increase of the amount of variables is elucidated by the following famous quote by the physicist Enrico Fermi [37, p. 1]:

> *With four parameters I can fit an elephant and with five I can make him wiggle his trunk.*

The causes for overfitting are numerous. It can be because of a too small dataset which does not incoporate all there needs to be known about the problem according to the right proportion or there are more variables than required, leading to a overly complicated model. The opposite is also possible to happen, if the complexity of the model is not capable by any means to portray a given problem. This would be an example of what is called underfitting. [36, p. 1-2]

To showcase an example of such fits, a simple dataset is generated and three different models are used to differentiate a underfitting from a overfitting from a good fit, shown in Fig. 13. The dataset is a quadratic function y of x with an additional random noise term $r(x)$ between -10 and 10 for each data point:

$$y(x) = x^2 + r(x) \tag{64}$$

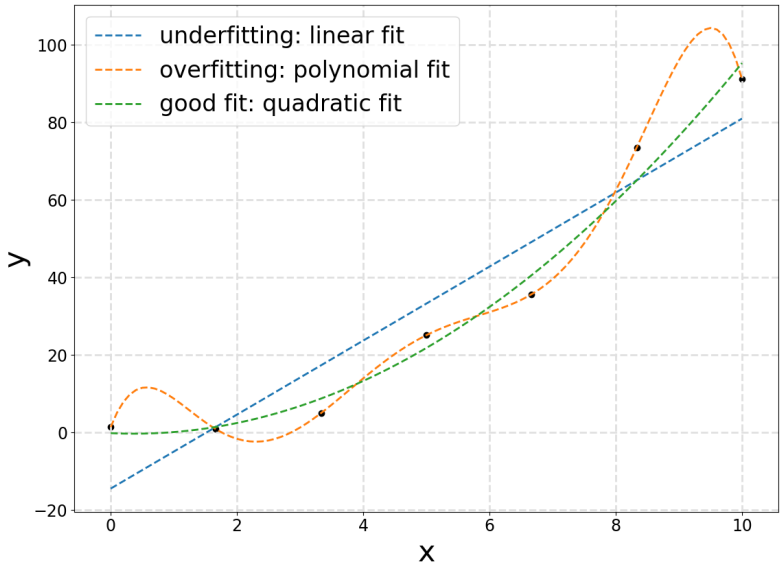


Fig. 13: Example of an overfit (orange, polynomial), underfit (blue, linear) and good fit (green, quadratic), on a quadratic dataset in x range from 0 to 10 with a random noise on each data point between −10 and +10.

When considering neural networks, it already results in a large amount of variables and possible connections, even for a few layers and with not too many neurons per layer. Therefore it is commonly thought that a network might generalize poorly [38, p. 381 ff.]. Restricting a neural network can therefore result in a better learning for the network. One way of doing so is to implement the early stopping method. This means to stop the learning procedure once the loss does not decrease by more then a certain threshold for a certain amount of training epochs.

Due to overfitting, it is common to divide a given data set into training and validation data set. Once a chosen method is trained, a good indicator for overfitting is when a machine learning method has a much greater accuracy for solving a problem when tested again on the training data set or on the validation data set. In this case it is likely that the model overfits the characteristics of the training data set and is therefore not as capable in solving the problem for a yet unseen dataset and therefore not able to properly generalize. [33, p. 73 - 78]

2.5 Super-Resolution Procedure

The goal of the $\mathcal{SR}$ procedure is to create larger system sized configurations of the Ising model from an initial data set of spin configurations [4, p. 1-3]. The entire procedure is described in its methodology in the next section 2.5.1. The following subsections go into more mathematical detail or further considerations.

2.5.1 Method

A set of spin configurations is created with the Metropolis algorithm and stored. This data set is then reduced by half its initial system length with the deterministic majority rule. This majority rule divides a spin configuration into blocks of 2^d spins. Every block is then reduced to the majority of the spins in this block. If there is an equal amount of spins, then the very first spin when iterating is corresponding to the entire block. This leads to a consistent rule, rather than randomly choosing a spin for the zero net spin case. For the 2D case, the first spin iterating is now said to be the top left spin of a block.

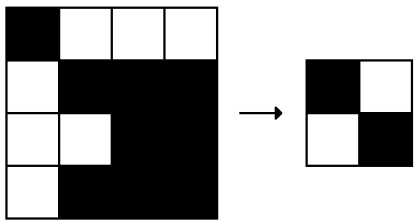


Fig. 14: A spin configuration of the Ising model is created by a MC simulation at a specific temperature. This is then divided into blocks of 2^d and each block is decimated by the deterministic majority rule.

This leads to two data sets in which every single configuration of one data set corresponds to one other configuration of the other data set. However this renormalization step leads to a change in temperature which needs to be taken into consideration.

A simple upscaling by duplication for the decimated system configuration leads to the same system size as the initial spin configuration. This is necessary for the neural network to learn, since the later built CNN itself does not change the system size. This upscaling is also required for scaling to larger system sizes.

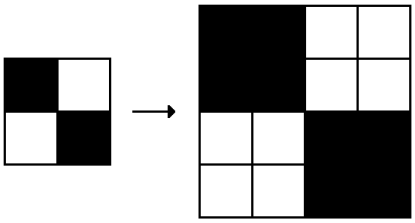


Fig. 15: Duplication of every spin to a block of size 2^d. This is applied to the previously decimated spin configuration and archives a lattice size of the original spin configuration. Also used to enlarge to larger system sizes.

A CNN is to be trained on the newly created Ising spin configuration which tries to reconstruct this initial spin configuration. Since the CNN is independent of the initial system size, the weights and biases are stored and can therefore be reused for resolving to larger system sizes. However for this problem the deflated spin configurations which also acts as the input layer for the CNN, is required to have a padding. This padding corresponds to the pbc. The length of this padding is given according to (48).

After the convolutional operation, every value on the lattice corresponds to a number between 0 and 1. This is the probability to be resolved to either 0 or 1.

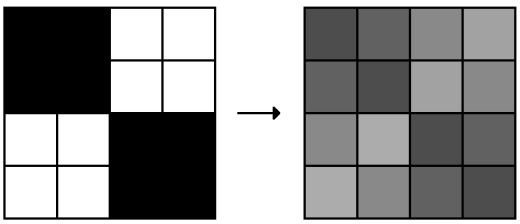


Fig. 16: To the left, a spin configuration after the upscaling procedure is shown. The operation after the CNN results in a probability distribution for each spin. This is shown to the right. The padding is not displayed in this diagram.

This probability distribution can be resolved to a binary spin configuration. For every spin, a random decimal between 0 and 1 is drawn and the output layer is set as the condition for the spin to be either +1 or −1. When the CNN had learned the proper super-resolution operation, this spin configuration ideally is corresponding to the original distribution.

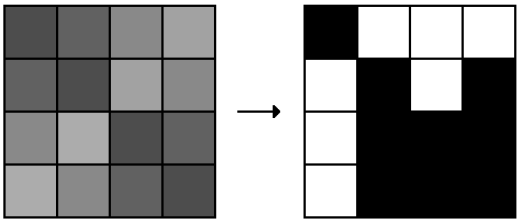


Fig. 17: The gray tiles in the left lattice correspond to a probability between 0 and 1 to be an actual spin of +1 or otherwise −1. In the right lattice this probability is allocated to a binary spin distribution.

This procedure thus far is required for training the CNN to reconstruct the initial spin configuration. Once the CNN is trained, all the steps without the decimation can be repeated and results in a enlarging of the spin configuration. This can be done indefinitely many times but will results in some error. Also every $\mathcal{SR}$ step does affect the temperature of the newly generated spin configurations according to the renormalization procedure. The following subsections consider further necessities, which those previously described transformation have already implied.

An alternative method is possible for the 1D case since after the upscaling in Fig 15, the information on every second spin is still preserved from the original spin configuration. Therefore their neighboring spins can be calculated with the HB algorithm as shown in section 2.3.3.

2.5.2 Decimation and Reconstruction of the Ising model

When in this chapter the decimation and reconstruction of the Ising is described, the same boundary conditions apply as given in section 2.2. Let $\boldsymbol{s} \in \mathbb{Z}_d^{L^d}$ be a d dimensional spin configuration, it follows a Boltzmann distribution [4, p. 2]:

$$P_\beta(\boldsymbol{s}) = \frac{\exp\left(\beta \sum_{<i,j>} s_i s_j\right)}{\mathcal{Z}} \tag{65}$$

The deterministic majority rule $\mathcal{MR}(\cdot)$ is applied as described in the previous chapter according to Fig. 14 to the spin configuration. This leads with the $\mathcal{SR}$ procedure $\mathcal{SR}(\cdot)$ to a new spins configuration $\tilde{\boldsymbol{s}} \equiv \mathcal{MR}(\mathcal{SR}(\boldsymbol{s})) = \mathbb{Z}_d^{(\frac{L}{2})^d}$, which is of half its initial system length. The newly obtained spin configuration can now be described by a marginalized distribution:

$$\tilde{P}_{\tilde{\beta}}(\tilde{\boldsymbol{s}}) = \sum_{\{\tilde{\boldsymbol{s}}\}} k(\tilde{\boldsymbol{s}}, \boldsymbol{s}) P_\beta(\boldsymbol{s}) \tag{66}$$

Here $k(\tilde{\boldsymbol{s}}, \boldsymbol{s})$ is the kernel of the transformation. This shows how the Boltzmann distributions of the original and the decimated spin configuration are related to each other.

When performing a reconstruction or resolving to larger system sizes, it is clear that any $\mathcal{SR}$ procedure should definitively satisfy the following relationship:

$$\mathcal{MR}(\mathcal{SR}(\tilde{\boldsymbol{s}})) = \tilde{\boldsymbol{s}} \tag{67}$$

The majority rule however results in a loss of information and it is not necessarily required for a reconstruction to follow the opposite order of the transformations. For every configuration generated with the Monte Carlo method, the $\mathcal{SR}(\mathcal{MR}(\boldsymbol{s}))$ does not have to be equal to its original spin configuration $\boldsymbol{s}$. However the goal of a properly trained neural network is for $\mathcal{SR}(\mathcal{MR}(\boldsymbol{s}))$ to still correlated to the proper Boltzmann distribution.

2.5.3 Resolving to larger System Sizes

Once the neural network has trained an appropriate set of weights of biases for the decimation and reconstruction of the Ising model, it becomes possible to resolve to larger system. Every spin is replaced by a block of 2^d equally oriented spins. Then the CNN is applied to the configuration with the pbc padding. Afterwards the probability is allocated to a binary value. This is the procedure which is described in section 2.5.1 just without the first decimation step. It is repeated for every $\mathcal{SR}$ step and leads with every step to a larger lattice size ($\mathbb{Z}_d^{L^d} \to \mathbb{Z}_d^{(2L)^d} \to \mathbb{Z}_d^{(4L)^d} \to \mathbb{Z}_d^{(8L)^d} \to ...$) [4, p. 3].

It is of crucial importance to consider that every $\mathcal{SR}$ step does actually change the temperature of the system. For a chain of multiple $\mathcal{SR}$ procedure steps it is required to train multiple neural networks. However it can be trained on its initial system length L with only the original and the decimated spin configuration. Concluding that the weights and biases of the neural network always correspond to a specific temperature transformation $\tilde{\beta} \to \beta$. For dimensions of equal or larger than 2 of the Ising model, the critical temperature would be a fix point in the system and therefore every $\mathcal{SR}$ procedure can be applied with the same weights and biases from the trained neural network, at this specific temperature transformation.

2.5.4 Numerical Solution to Rescale the Temperature

In the section 2.2.4 it had already been discussed of how to transform the temperature for the 1D case according to a decimation or $\mathcal{SR}$ step. This solution was shown analytically. However as of the time being, it is not know of how to solve this problem for the 2D or 3D case. This work instead uses an alternative numerical method as described in [4, p. 8-9] for finding the solution of transforming the temperature.

As described in chapter 2.5.2, when creating a set with size n of spin configurations $\boldsymbol{s}_i$ with $i \in \{1, 2, ..., n\}$ at size L, then one is able to convert it to its decimated form $\tilde{\boldsymbol{s}}_i$. The decimated spin configuration corresponds to half of the original size $\tilde{L} = L/2$. It is possible to calculate an observable at the decimated system length, such as the expectation value of the absolute of the magnetization. This can be done with the Metropolis algorithm.

One can calculate and compare the observables of the decimated spin configuration to the spin configuration at the same size which is calculated with the Metropolis algorithm. It is then required for those two curves to 'collapse' [4, p.9]. This will then create a transformation corresponding to the temperature transformation $T \to \tilde{T}$. The latter simulation will focus on the size $L = 16$, therefore this method is now shown for the decimated size at $\tilde{L} = 8$, for the 2D case.

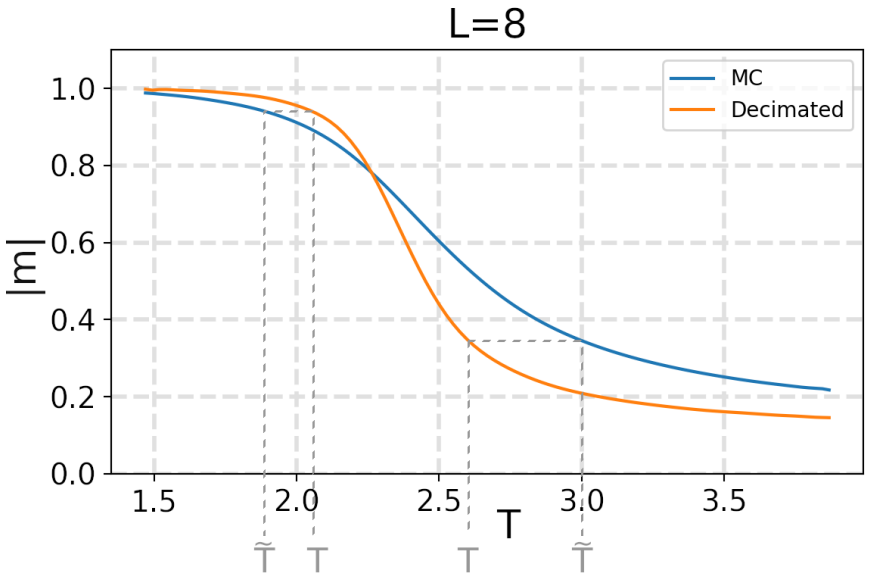


Fig. 18: Solution of the absolute magnetization of the decimated spin configurations and the Metropolis algorithm. Displaying the numerical method for the temperature transformation.

For creating a numerical transformation, one looks at one temperature which is calculated with the decimated spin configuration $\tilde{T}$ and goes vertically in the graph towards the intersection with the Monte Carlo Metropolis solution at the same system length $\tilde{L}$. In other words, one corresponds the temperatures of two points of the different graphs which have the same solution for the observable which is the absolute magnetization.

Once the temperature transformation is found out numerically, it is possible to transform the temperature to larger system sizes, analogous to section 2.5.3. The numerical solution enables the temperature transformation to larger system sizes, even though it is only calculated on one system size, which is $\tilde{L}$. However the temperature transformation is not required at the critical temperature, due to it being a fix point.

2.5.5 Regularization Term to the Loss Function

The cost function had been discussed in section 2.4.9. The cross-entropy loss function is chosen to be the measure of learning for the neural network. The cross-entropy function is comparing the output of the network, which is treated as a probability for a certain spin direction, to the original target spin direction. For the 1D case this is sufficient for the CNN to learn. However, when looking at the 2D Ising model, this is not the case anymore.

For better learning results, there is going to be a regularization term added to the cross-entropy loss function. This regularization term $\mathcal{R}$ is proportional to this energy difference term [4, p. 3]:

$$\mathcal{R} \propto |E(\boldsymbol{s}_i) - E(\boldsymbol{P}_i)|^2 \tag{68}$$

A proportionality factor p_f is to be implemented as a parameter before the energy term. The choice of this parameter is important and balances how much the energy difference term is weighted.

Now the implementation of the regularization term in the *Python* library *TensorFlow* is shown:

```
#importing libraries
import numpy as np
import tensorflow as tf
#implementing the regularization term
p_f = 2e-8 #setting the proportionality factor
L = 16 #system length
def regularization_term(y_true, y_pred):
    #spin input is set from between 0..1 to -1..+1
    y_true_new = (y_true*2)-1
    y_pred_new = (y_pred*2)-1
    #indexing for the nearest neighbor interaction of the energy
    idx = (np.array(list(range(L)))+1)%L
    #calculating the difference in the regularization term
    reg  = y_true_new * tf.gather(y_true_new, idx, axis=1)
    reg += y_true_new * tf.gather(y_true_new, idx, axis=2)
    reg -= y_pred_new * tf.gather(y_pred_new, idx, axis=1)
    reg -= y_pred_new * tf.gather(y_pred_new, idx, axis=2)
    reg = tf.reduce_sum(reg,axis=2)
    reg = tf.reduce_sum(reg,axis=1)
    #calculating the square in the regularization term
    reg = tf.square(reg)
    #summing over the entire mini-batch
    reg = tf.reduce_sum(reg)
    #returning the regularization term with a proportionality
        ↪ factor
    return p_f * reg
#adding the regularization term to the cross-entropy loss
def my_custom_loss(y_true, y_pred):
    return tf.keras.losses.BinaryCrossentropy()(y_true, y_pred) +
        ↪ regularization_term(y_true, y_pred)
```

The term $E(\boldsymbol{P}_i)$ correspond to the energy of the configuration which consists of a probability distribution. Therefore the energy is still calculated similarly to equation (13) but the spin value is not binary anymore. Moreover the probability value is mapped between -1 and +1. In the example code, this corresponds to the *y_pred*. The term $E(\boldsymbol{s}_i)$ is the energy of the original spin configuration, or *y_true* in the sample code.

Unlike the cross-entropy loss, the intended regularization term is not predefined as a loss function in any typical framework for neural networks. A way of implementing this is therefore presented for Python and the *TensorFlow* and *NumPy* library. This function needs to be differentiable according to the neural network framework, here *TensorFlow*. The regularization term is implemented according to (68). Finally a custom loss is defined as the summation of the regularization term and the cross-entropy loss function.

2.5.6 Finite Size Considerations

With the majority rule procedure in this work, there is the blocking of spins with each block containing 2^d spins. This procedure reduces the system length to half of its original system length. A procedure of downsampling which is described exemplarily in Fig. 14. It is shown now once again for larger Ising spin configurations.

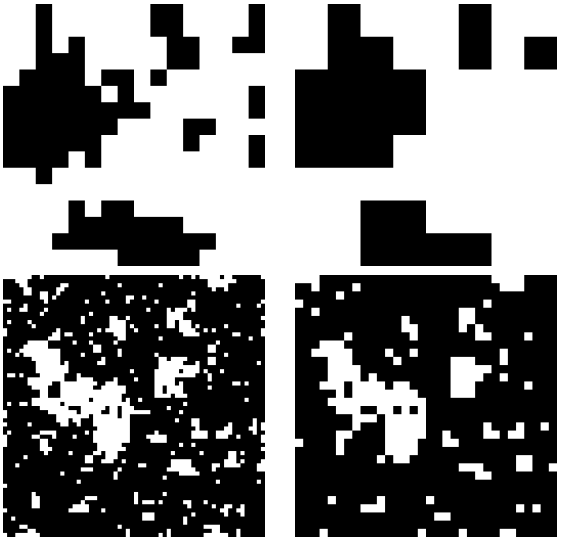


Fig. 19: Deterministic majority rule blocking for the Ising Model with size 16×16 (top) and 64×64 (bottom). The left images show the original Spin configuration at $T = T_c$. The right pictures show the same configuration with the blocking rule applied, resulting in half the original system length. Images inspired by [6, p. 241].

It is visible from this images that after the deterministic majority rule procedure is applied, on a larger scale the features of the configuration are still preserved. Therefore supposedly the correlation is still preserved as well. However when blocking the spins, these blocks form a new spin and the length between the originally located spins is as well reduced. The correlation length is therefore halved for the scaled-down configuration [6, p. 241].

Even the well established MCRG methods which had been discussed in 2.3.4 for calculating the critical temperature uses the assumption that the blocked configurations correspond to those configurations that would be generated at the reduced system size and having the appropriate temperature [6, p. 253]. It is not known a priori, whether it is going to be a good approximation. Moreover the results also depend on the way of the block transforming. The majority rule decimation works well for this kind of renormalization procedure, at least for the 2D case. But it might perform poorly for other models [6, p. 246].

This is a huge problem, since this methodology of super-resolving to larger system sizes isn't any better than the RG procedure. It is assumed that there are going to be systematic errors at play which will not be accounted throughout this thesis. It is not foreseeable how to grasp the possibility of those systematic errors. However it is possible a posteriori to analyze how well the distributions of a super-resolved set of spin configurations and original spin configuration at the corresponding system size do resemble each other.

2.5.7 Alternative Blocking Method

The 3D Ising model is known to be difficult to obtain accurate results with RG model. The reason for this being, since the fix point is approached rather slowly with this method, when looking for the critical exponentials. Moreover, the majority rule, which performs well for the 2D case, does not do so for the 3D case. Even though this issue is dealt with since the 70s, on the 12th of November 2020, there is a paper [39] by Swendsen et al., suggesting a better method to archive a faster convergence when calculating critical exponents. This method suggests a different method then the majority rule to block a cube of $2 \times 2 \times 2$ spins together.

The newly suggested probability $\mathcal{P}_{alt}$ for a block with the specific amount of l spins: $\sum_{i \in l} s_i$, to point in either direction of the resulting spin $\tilde{s}_l = \pm 1$ is described by:

$$\mathcal{P}_{alt}(\tilde{s}_l) = \frac{\exp(\mathrm{w}\tilde{s}_l \sum_{i \in l} s_j)}{\exp(\mathrm{w} \sum_{i \in l} s_i) + \exp(\text{-}\mathrm{w} \sum_{i \in l} s_i)} \tag{69}$$

With w being a weighting parameter to be optimized for certain circumstances. For $\mathrm{w} \to \infty$ the probability would correspond to the majority rule. The paper suggest different optimizations of the w, when considering different exponents. The value for w is then adjusted to make the calculated exponents converge as fast as possible, resulting in two different weighting parameters: $\mathrm{w}(y_T) = 0.4314$ and $\mathrm{w}(y_H) = 0.555$ [39, p. 2].

This alternative blocking method could be interesting when considering the 3D Ising case, in case of experiencing similar difficulties as the research for the traditional RG method has been experiencing. The method with the CNN is independent of the initial blocking methodology and is capable of interpreting any kind of input.

3 Simulations and Evaluations

The goal of this work is to train a CNN to upscale the system sizes of a set of Ising model spin configurations of different dimensions while retaining the physical properties according to the appropriate temperature. The neural network is trained with supervised learning and therefore needs two sets of spin configurations for training. One as the input and one as a true configuration. This is done with the Metropolis algorithm and its decimation by the deterministic majority rule. The procedure for this is shown in section 2.5. This method is implemented and evaluated for different dimensions. Here is a recapitulation of the implementation with the use of the theory which is described in the whole section 2:

- Define a temperature range for the upcoming simulations. Since the 1D case offers an analytic solution according to the renormalization group in 2.2.4, this can be done on the fly. For the 2D case it is required to obtain a numerical solution as shown in fig. (18). This applies iteratively for every upscaling step in the future.
- Run multiple MC simulations for the defined temperatures as well as the transformed temperatures at a system length L and store the spin configurations. This MC simulation can be the Metropolis simulation which is known from section 2.3.2
- Downscale the spin configurations according to the deterministic majority rule as shown in fig. 14. Afterwards perform a simple block duplication as shown in fig. 15. Store this configuration. This is now to as the deflated spin configuration.
- Define the setup conditions of the CNN. This referes to the amount of hidden layers and the amount of neurons for each of those layers. Then apply a padding according to those setups of the appropriate size calculated with (48) for the deflated spin configuration.
- Train a neural network with the previously defined setup to reconstruct the original spin configuration. Store the weights and biases after the training is completed with early stopping. This is to be done for the configurations at the defined temperature range as well as the renormalized temperature ranges of every future upscaling step.
- Use the super-resolution method according to section 2.5.3 to obtain larger system sizes. The newly obtained spin configurations correspond to a transformed temperature as mentioned in the first bullet point.
- Finally the results of this method can be compared to the authentic solutions on the appropriate system sizes and temperatures. Those solutions can be also obtained by the MC simulation.

The observables calculated for the simulations are going to be the expectation values of the absolute magnetization $|m|$, the energy e and the two point spin correlation G. The latter always considers a distance of $|r| = 3$. The systematic errors are unforeseeable as described in section 2.5.6. At large statistics therefore the statistical error is much smaller than the systematic errors. The errors are only shown according to the values which are obtained with an alternative methods which are known to be working properly (MC Metropolis).

3.1 1D Simulation

The 1D case of the $\mathcal{SR}$ procedure serves as a good testing ground for this method. The 1D case does show no critical phenomena which would lead to a divergent correlation length and can also be solved analytically. The latter is especially useful for the temperature transformation. Every $\mathcal{SR}$ step requires the temperature to be transformed as derived in (37). With a good solution in the more simple 1D case, it validates the foundational implementation, which then can be extended for the 2D case.

For the 1D case it is also possible to resolve the unknown data points with the HB algorithm, when also considering the appropriate temperature transformation. This is an alternative method which does only work for the 1D case and is here therefore also shown and compared to the trained CNN.

3.1.1 Data Preparation

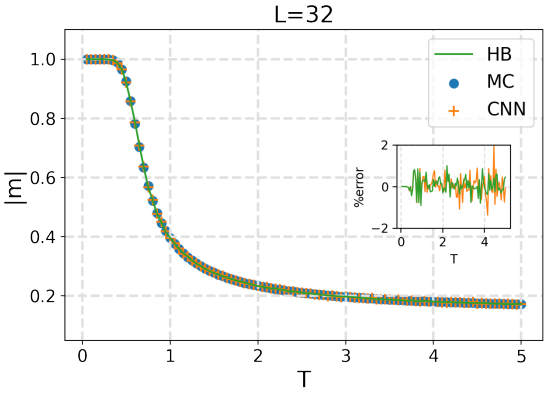
The Metropolis MC simulation generates $n = 2 \cdot 10^4$ spin configurations with an initial system size of $L_0 = 32$. This is then decimated to half its size and enlarged by simple block duplication back to the original system size. Additionally a padding is added. This procedure is also described in section 2.5.1. The two corresponding sets of spin configurations (original and deflated) which now have been created are divided into a training and a testing dataset with a ratio of $1 : 1$. This way, the neural network can be tested on data which is yet unknown to the network. The setup of the CNN is made up of 3 convolutional kernels, each with 15 neurons. The learning rate is set to $\eta = 10^{-3}$. The batch size is 10^3. This network is then trained to reconstruct the original spin configurations from the deflated spin configurations. The network stops learning with an early stopping procedure, once the loss is not decreased with further learning steps. The threshold for this early stopping is set to 0 and the patience to 15. It is possible that the neural network fails to obtain a good result, due to being stuck in a local minimum, therefore the learning procedure needs to be repeated to obtain a better reconstruction.

The original temperature range for the spin configurations generated at system size of $L_0 = 32$ is $0.05 \leq T \leq 5.00$. The step size is 0.05. There are 5 $\mathcal{SR}$ steps applied, resulting a system length increase of: $L_0 = 32 \rightarrow L_1 = 64 \rightarrow L_2 = 128 \rightarrow L_3 = 256 \rightarrow L_4 = 512 \rightarrow L_5 = 1024$. The results are shown for every $\mathcal{SR}$ step. This procedure is repeated on $5 \cdot 10^5$ spin configurations from a pool of the initially created spin configurations. To be able to compare the later results, a Metropolis MC simulation is run for each of the temperatures and sizes with $75 \cdot 10^6$ sweeps. One sweep corresponds to simulating spin configurations of the total system size of $N = L^d$. The super-resolution procedure is also executed with an alternative method using the HB algorithm. This HB super-resolution is repeated for 10^6 times.

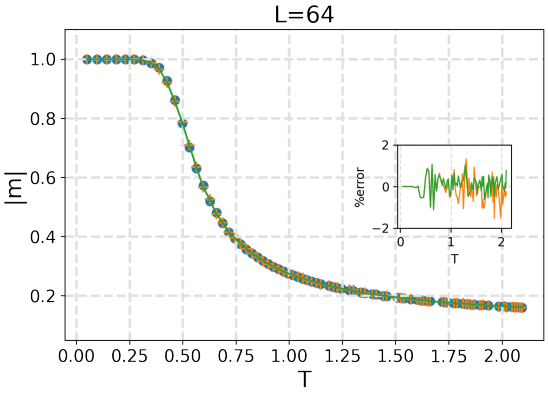
3.1.2 Results

The expectation values for the absolute magnetization, energy and two point spin correlation at distance 3 is calculated. Moreover the distribution of the observables are shown with a histogram plot. This is done for a hotter temperature case and for a colder case. For both cases, the original temperature is chosen at $L = 32$. However it is iteratively transformed with the according temperature transformation depending on the number of steps and results in the 1D case in a cooling of the temperature according to (37).

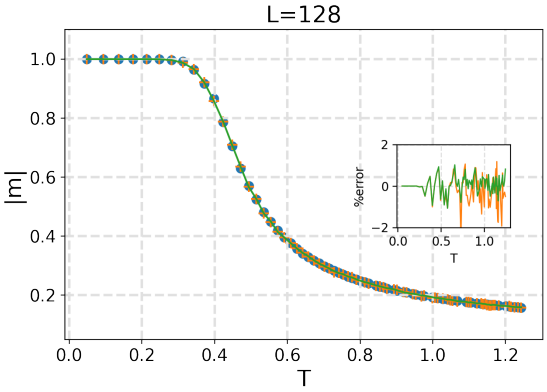
The expectation value of the absolute magnetization is shown. The results are done with the Metropolis Monte Carlo simulations and the two reconstruction methods. At first the reconstruction back to the original temperature is shown. Afterwards the super-resolution for the first 5 enlarging steps. The error is calculated as a deviation to the results of Metropolis MC simulations.



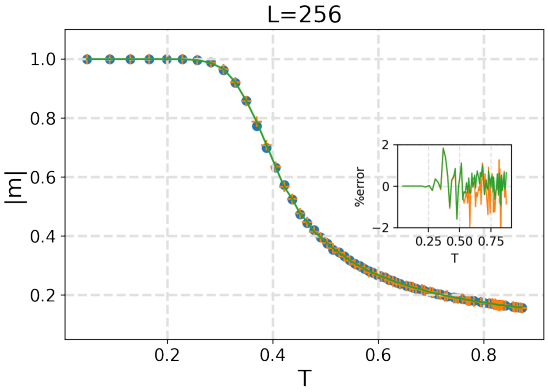
(a) Reconstructing to the original system size



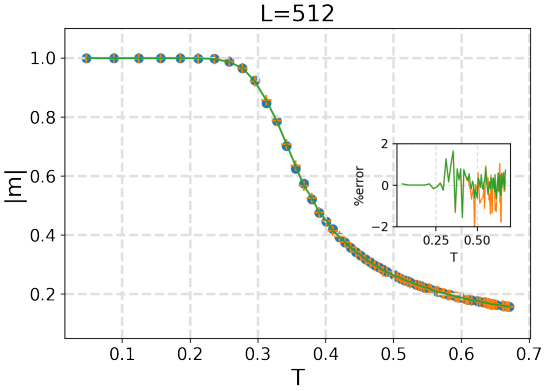
(b) 1st $\mathcal{SR}$ step



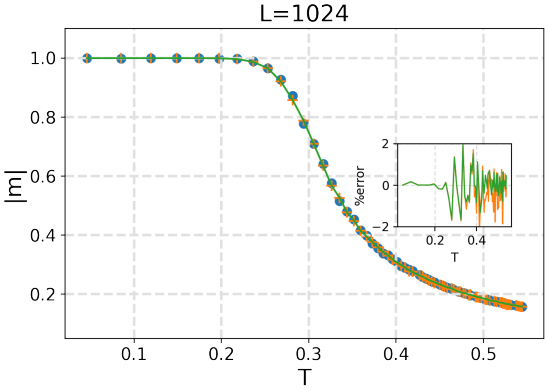
(c) 2nd $\mathcal{SR}$ step



(d) 3rd $\mathcal{SR}$ step



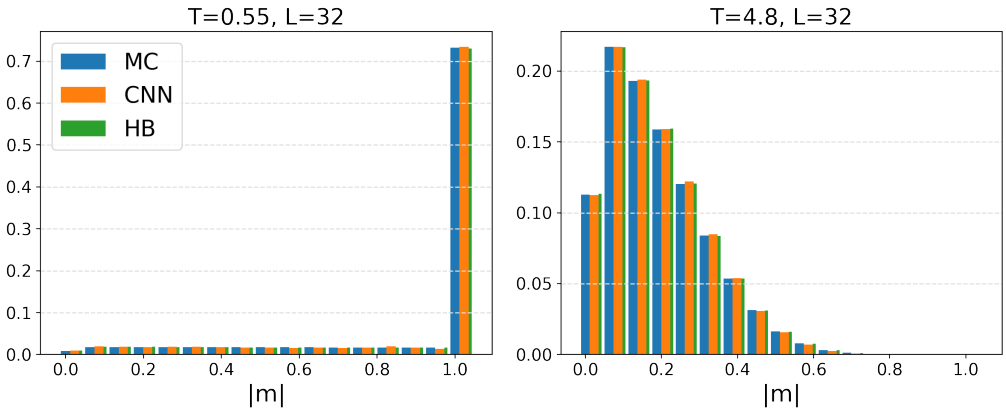
(e) 4th $\mathcal{SR}$ step



(f) 5th $\mathcal{SR}$ step

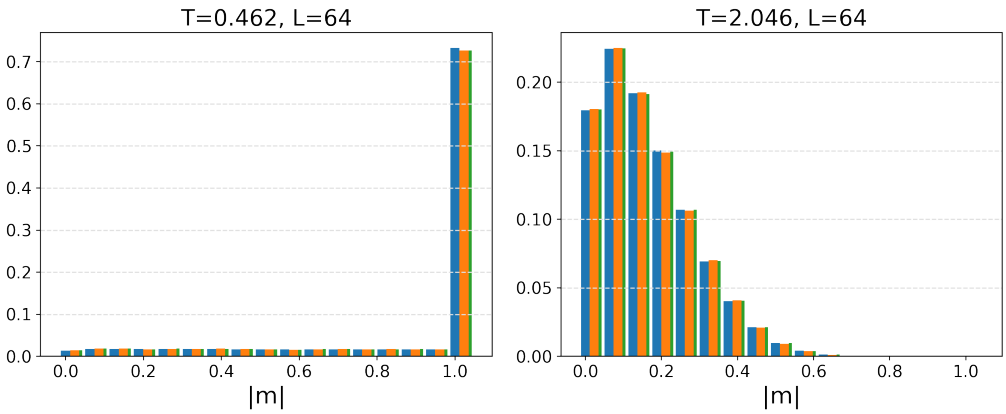
Fig. 20: Results for the expectation value of the absolute magnetization of the reconstructions with the CNN and the alternative HB method. Compared to the results of the Metropolis MC algorithm. Additionally the first 5 $\mathcal{SR}$ steps from $L_0 = 32$ up to $L_5 = 1024$.

The following histograms for the distribution of the absolute magnetization are shown for a lower and a higher temperature case. Those temperatures are appropriately transformed with each $\mathcal{SR}$ step.



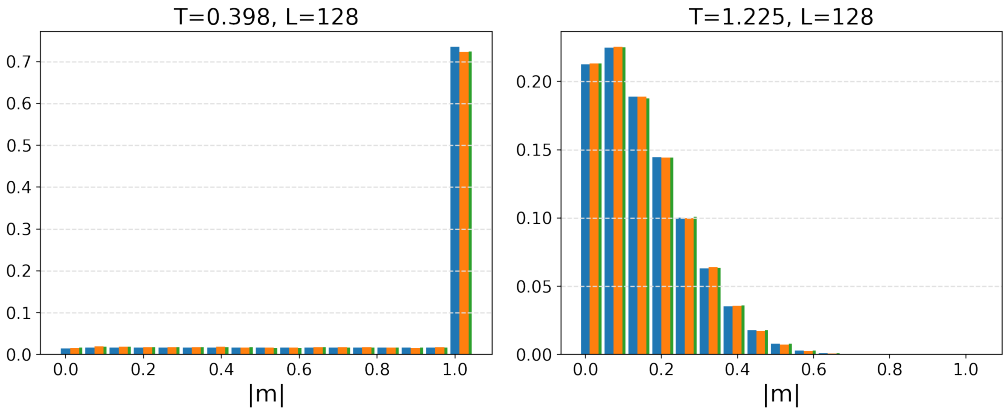
(a) Reconstructing, colder case

(b) Reconstruction, hotter case



(c) 1st $\mathcal{SR}$ step, colder case

(d) 1st $\mathcal{SR}$ step, hotter case

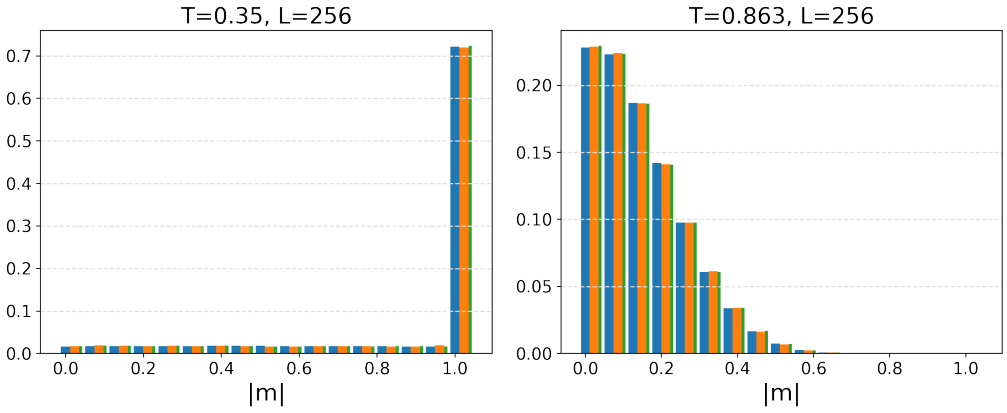


(e) 2nd $\mathcal{SR}$ step, colder case

(f) 2nd $\mathcal{SR}$ step, hotter case

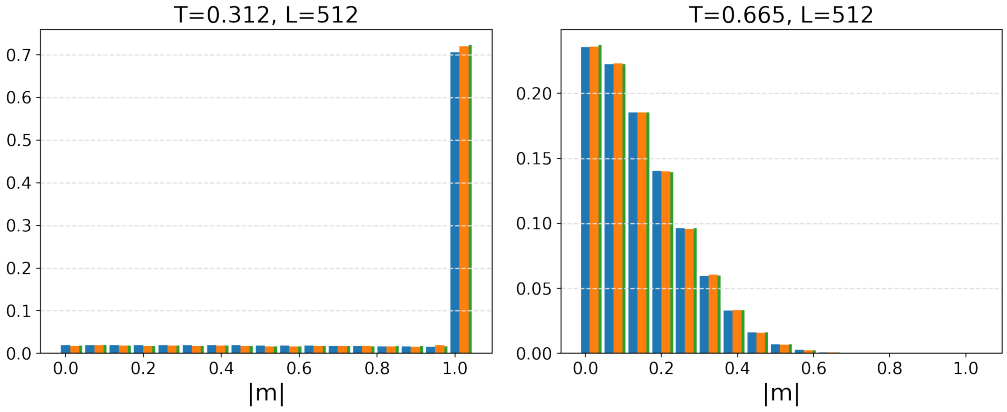
Fig. 21: Histogram plots of a colder (left) and a hotter (right) case for the expectation of the absolute magnetization. Showing the reconstruction to the original system size as well as the first two $\mathcal{SR}$ steps.

The following figures are a continuation to the previous ones. Now for the next three $\mathcal{SR}$ steps: 3 to 5.



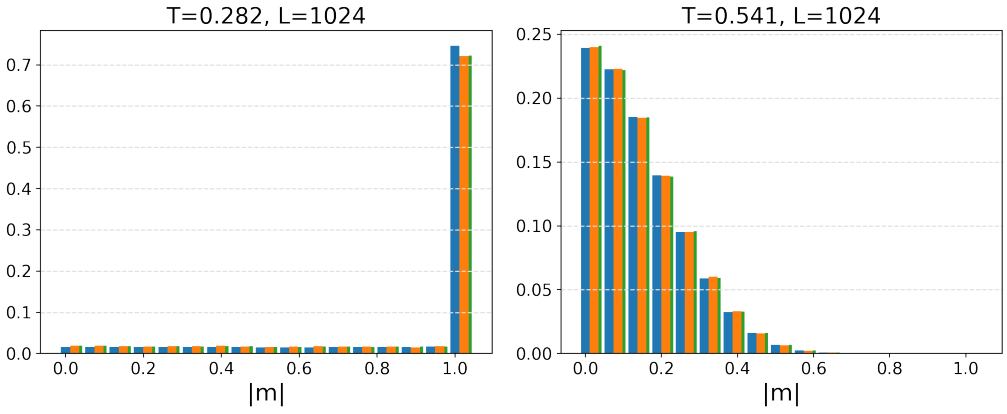
(a) 3rd $\mathcal{SR}$ step, colder case

(b) 3rd $\mathcal{SR}$ step, hotter case



(c) 4th $\mathcal{SR}$ step, colder case

(d) 4th $\mathcal{SR}$ step, hotter case

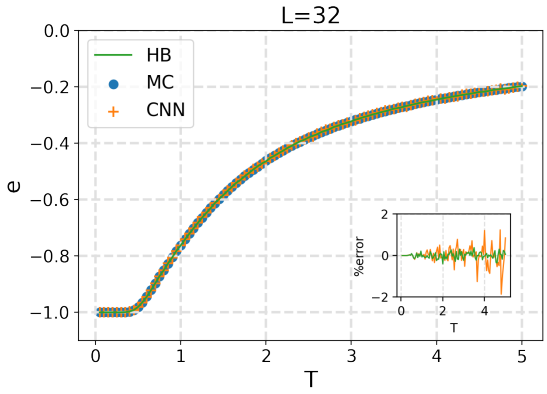


(e) 5th $\mathcal{SR}$ step, colder case

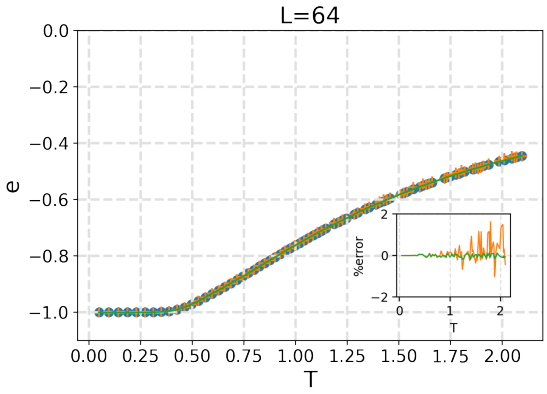
(f) 5th $\mathcal{SR}$ step, hotter case

Fig. 22: Histogram plots of a colder (left) and a hotter (right) case for the expectation of the absolute magnetization. Showing the $\mathcal{SR}$ steps 3 to 5.

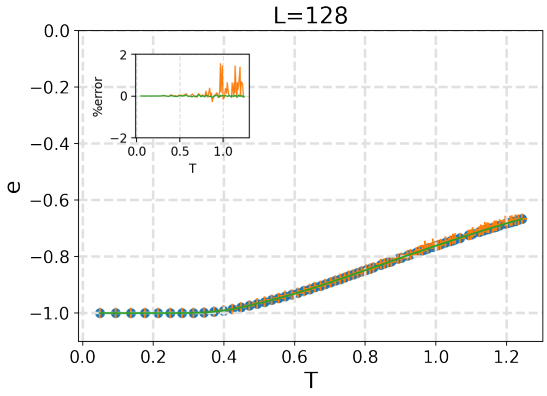
Analogous to the magnetization, here is the expectation value of the energy. First for the reconstruction to the original system size, afterwards for the first five $\mathcal{SR}$ steps.



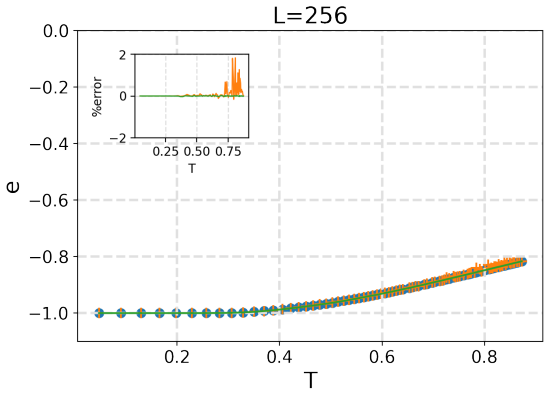
(a) Reconstructing to the original system size



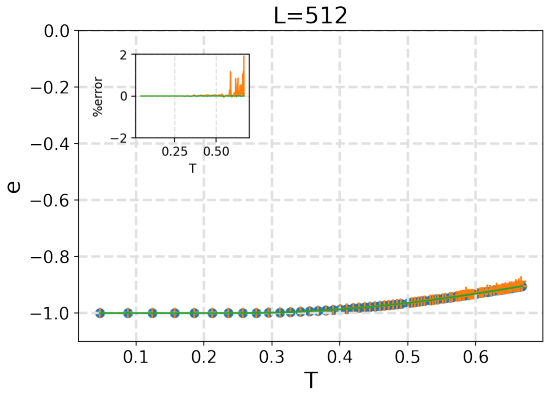
(b) 1st $\mathcal{SR}$ step



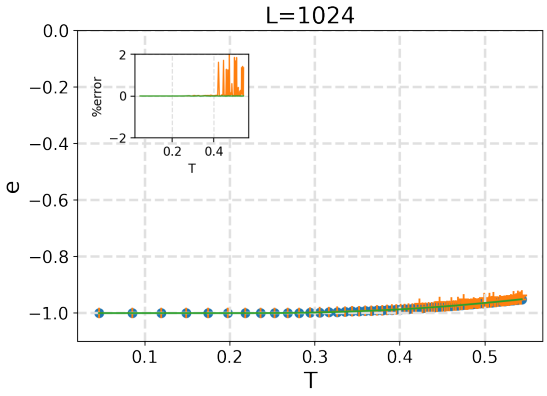
(c) 2nd $\mathcal{SR}$ step



(d) 3rd $\mathcal{SR}$ step



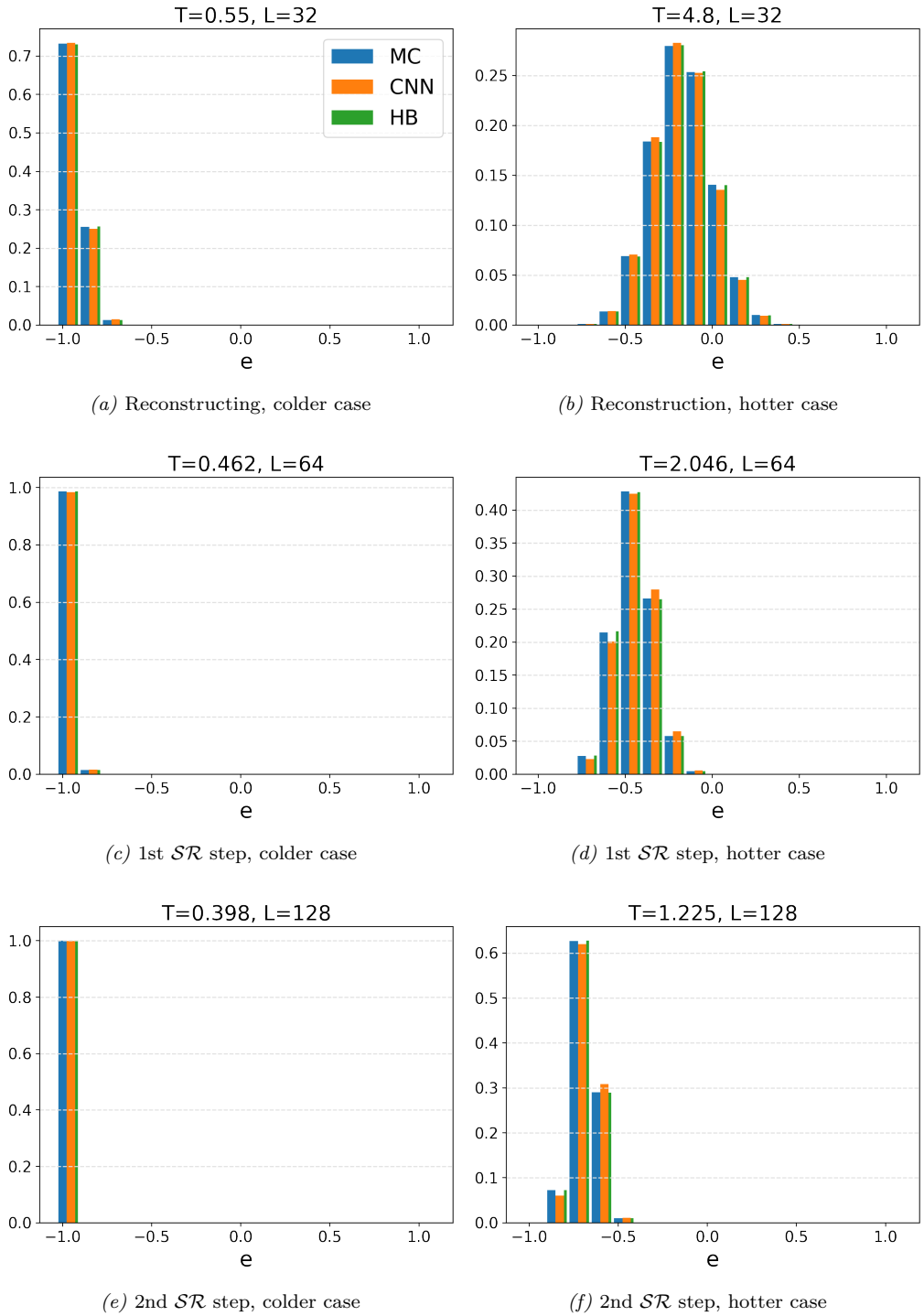
(e) 4th $\mathcal{SR}$ step



(f) 5th $\mathcal{SR}$ step

Fig. 23: Results for the expectation value of the energy of the reconstructions with the CNN. Additionally the first 5 $\mathcal{SR}$ steps from $L_0 = 32$ up to $L_5 = 1024$.

The following histograms for the distribution of the energy are shown for a lower and a higher temperature case. Those temperatures are appropriately transformed with each $\mathcal{SR}$ step.



(a) Reconstructing, colder case

(b) Reconstruction, hotter case

(c) 1st $\mathcal{SR}$ step, colder case

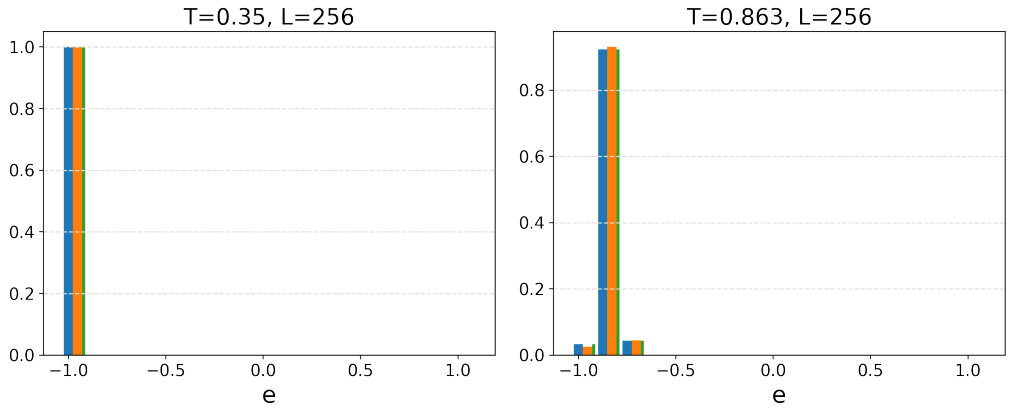
(d) 1st $\mathcal{SR}$ step, hotter case

(e) 2nd $\mathcal{SR}$ step, colder case

(f) 2nd $\mathcal{SR}$ step, hotter case

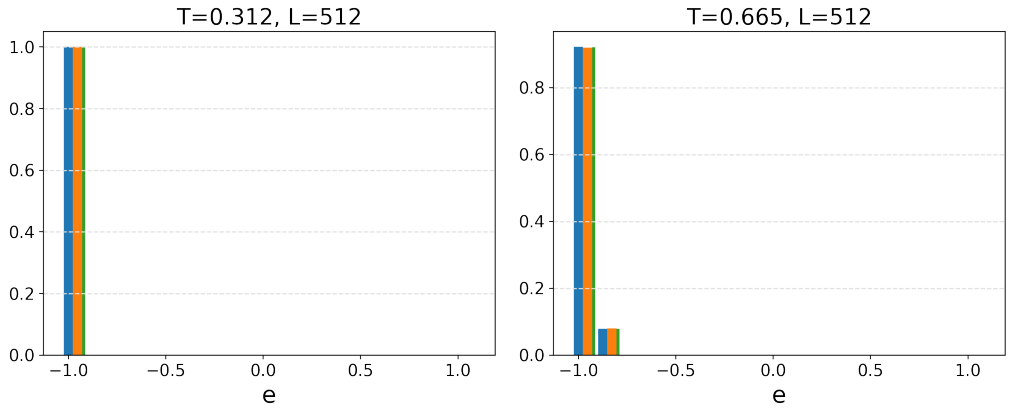
Fig. 24: Histogram plots of the colder (left) and the hotter (right) case for the expectation of the energy. Showing the reconstruction to the original system size as well as the first two $\mathcal{SR}$ steps.

Here is a continuation of the previous histograms for the next 3 $\mathcal{SR}$ steps: 3 to 5.



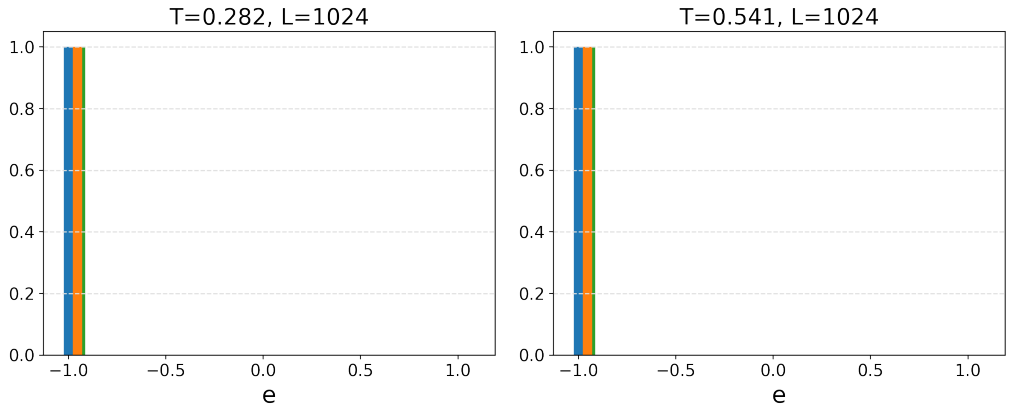
(a) 3rd $\mathcal{SR}$ step, colder case

(b) 3rd $\mathcal{SR}$ step, hotter case



(c) 4th $\mathcal{SR}$ step, colder case

(d) 4th $\mathcal{SR}$ step, hotter case

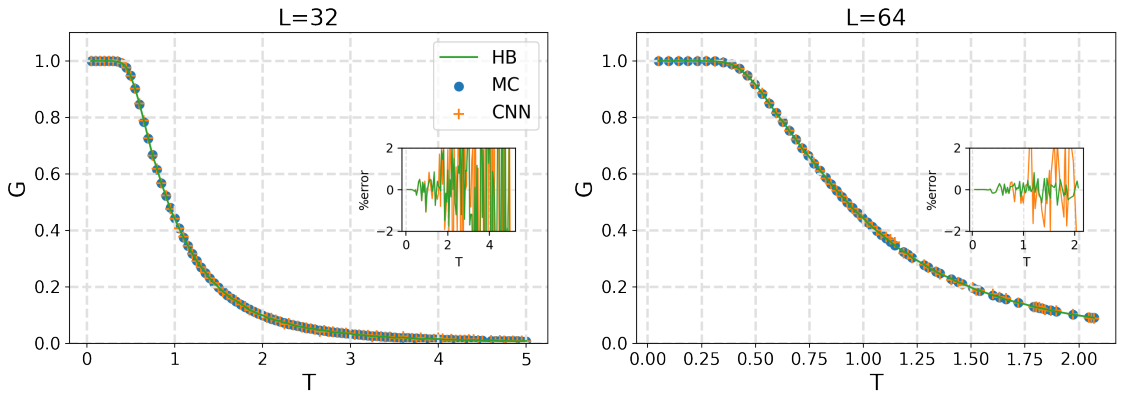


(e) 5th $\mathcal{SR}$ step, colder case

(f) 5th $\mathcal{SR}$ step, hotter case

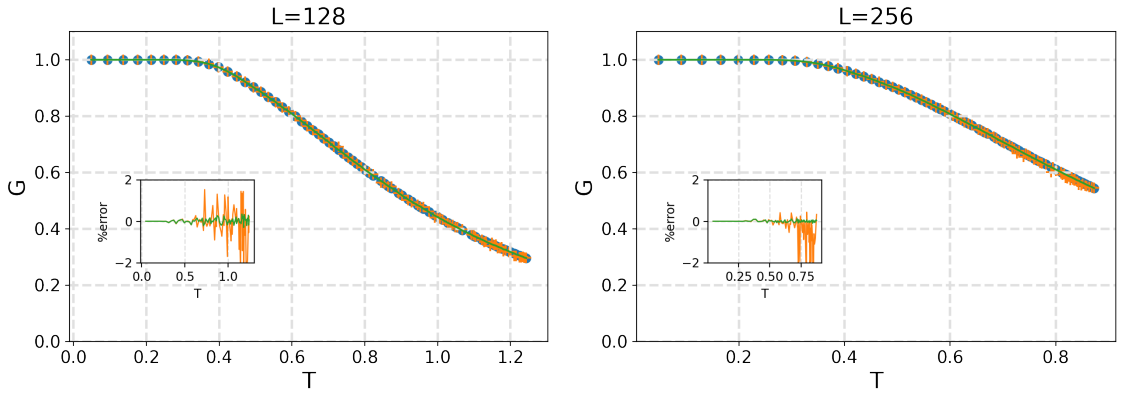
Fig. 25: Histogram plots of a colder (left) and a hotter (right) case for the expectation of the energy. Showing the $\mathcal{SR}$ steps 3 to 5.

Analogous to the expectation values of the magnetization and the energy, here is the expectation value of the two point spin correlation function at distance 3. First for the reconstruction to the original system size. Afterwards for the first five $\mathcal{SR}$ steps.



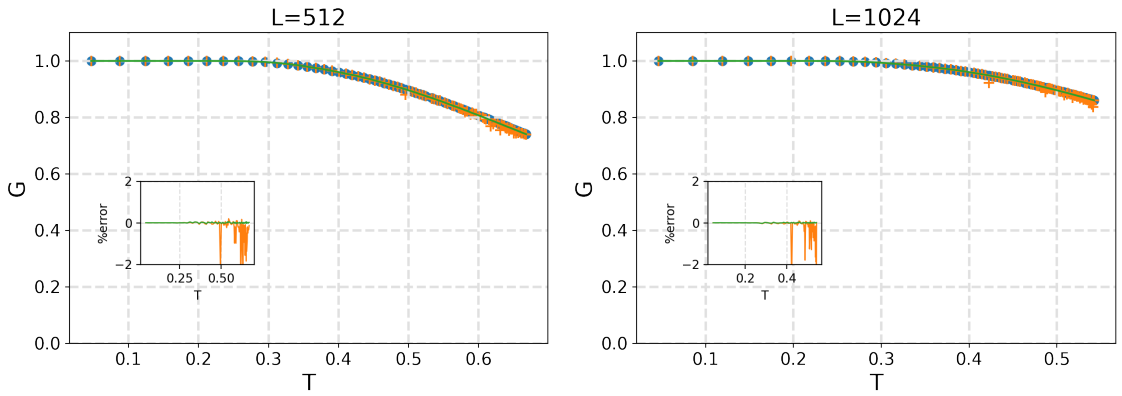
(a) Reconstructing to the original system size

(b) 1st $\mathcal{SR}$ step



(c) 2nd $\mathcal{SR}$ step

(d) 3rd $\mathcal{SR}$ step



(e) 4th $\mathcal{SR}$ step

(f) 5th $\mathcal{SR}$step

Fig. 26: Results for the expectation value of the two point spin correlation with distance $|r| = 3$ of the reconstructions with the CNN. Additionally the first 5 $\mathcal{SR}$ steps from $L_0 = 32$ up to $L_5 = 1024$.

3.1.3 Discussion

In the previous section the results for the reconstruction of the 1D Ising case, as well as the first five $\mathcal{SR}$ steps are shown. Starting at an initial configuration size of $L_0 = 32$ and resolving up to a size of $L_5 = 1024$. These results are compared and the error is calculated with the results of the Metropolis Monte Carlo simulations. Another comparison is shown to an alternative reconstruction with the HB algorithm. Moreover the histogram distributions at two initial temperatures are shown for all three methods. This is also shown after every temperature transformation step.

The $\mathcal{SR}$ with neural networks show a better reconstruction for cooler temperatures than for higher temperatures for every observable. This applies especially for larger super-resolution steps than for smaller ones. In general there is for most cases a small deviation around 0. A minority of data points stand out in this sense having a deviation of around $\pm 2\%$. This is likely due to the fact that the simulation is run for rather large amount of 100 data points.

The alternative HB method also shows a very similar reconstruction. However it shows to be significantly better for larger temperatures when looking at the energy and the two point correlation function starting at the 2nd super-resolution step and higher.

The histogram distribution for both the neural network as well as the HB reconstruction method show a good reconstruction of the histogram distribution. Only small deviations are seen. It is important to keep in mind that for larger system sizes the histogram bins are implying a much larger amount of possible magnetization or energy values. In 1D, for the first system size of $L_0 = 32$, there are a total of 17 possible energy or absolute magnetization values. For $L_5 = 1024$ there are 513 possible values for either observable.

3.2 2D Simulation

It is shown for the 1D case that the implementation for the general algorithm does work. However the 2D case captures additional challenges for the implementation of this algorithm as well as for the learning procedure.

At first the numerical solutions for the temperature transformation is found. The network is then trained according to the appropriate temperatures. Before the results of the observables are presented, the spins are also visualized for each of the steps as well as the probability distributions from the output layer of the neural network. For a further analysis of the CNN, different network layer setups are trained on the same data. The kernel of those trained network is also presented. Analogous to the previous chapter the performance of the CNN is shown for the reconstruction back to the original temperatures, but here only for the first two $\mathcal{SR}$ steps. The absolute magnetization, energy and two point spin correlation function is investigated. When studying the results, it is important to keep in mind that the training on the CNN does not always succeed and needs to be repeated. After the repetitions, the weights and biases resulting in the best reconstruction are chosen for the polished results. However the unpolished results for the first six repetitions are also investigated in a further section. To test the boundaries of this method, the $\mathcal{SR}$ procedure is also performed for much smaller initial system sizes.

3.2.1 Temperature Transformation

The rescaling function for the 1D case is shown to be solvable analytically with the help of the RG theory in section 2.2.4. However for the 2D case such an analytic solution is not available and therefore a numeric method is used as in section 2.5.4 for the purpose of temperature transformation.

The method which is presented here requires the solutions for the expectation value of the absolute magnetization for the decimated Ising model as well as the original. The decimated Ising model is simulated at a system size of $L = 16$ with Metropolis MC and then decimated to half its system length at $\tilde{L} = 8$. The original Ising Model is simulated as well with Metropolis MC but at a system length of 8. Both simulations consist of 60 data points each, which each are simulated with 10^6 sweeps. Afterwards a polynomial fit of order 20 for the original Ising model and the decimated Ising model is created. Both of those functions can be resolved together into a temperature transformation for the $\mathcal{SR}$ procedure. This is shown in the following figure for an initial temperature of $T_0 = 3.0$.

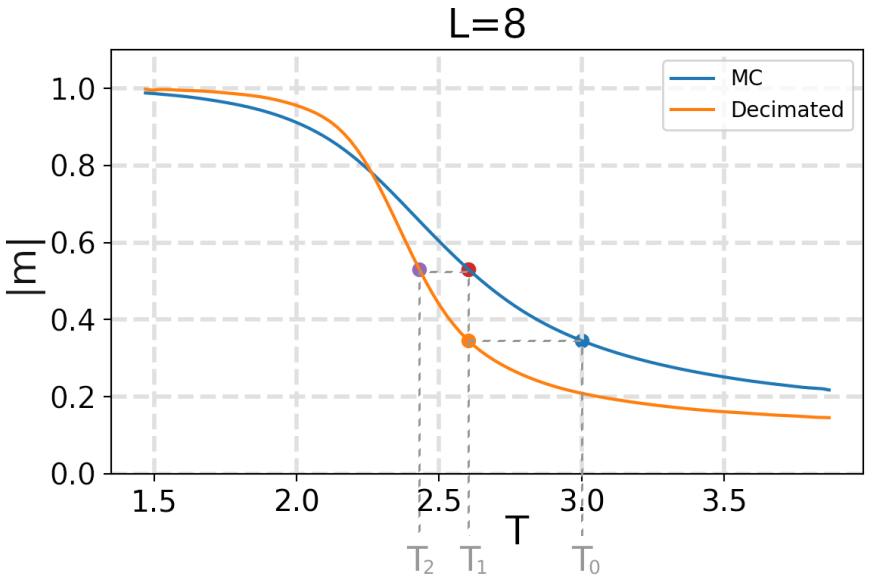


Fig. 27: Solution of the absolute magnetization of the decimated spin configurations and the Metropolis algorithm. Displaying the numerical method for the temperature transformation.

Starting on the graph at a temperature point of $T_0 = 3.0$ and the correspondent magnetization of the original Ising model, the transformation after one $\mathcal{SR}$ step corresponds to the intersection with the decimated Ising model graph. This is when going horizontally, which also correlates to looking for the corresponding temperature at the same magnetization. This results in a new temperature of $T_1 = \mathcal{SR}(T_0) = 2.60266$. For the second $\mathcal{SR}$ step, this procedure is repeated for the newly obtained temperature and results in the temperature $T_2 = \mathcal{SR}(\mathcal{SR}(T_0)) = 2.43032$.

The initial temperature range is chosen to be between 1.5 to 3.0 with a step size of 0.1. The intersection of both graphs in fig. 27 corresponds to a calculated critical temperature. The critical temperature for an infinite system would have been at $T_C^{(\infty)} = 2.26919$. The critical temperature calculated for the size of $\tilde{L} = 8$ is slightly shifted towards cooler temperatures and is given at the temperature $T_C^{(16\times16\to8\times8)} = 2.26311$. Both, the critical temperature at infinite system size and the calculated finite size critical temperature are added to the previously described temperature range. However only the latter is shown when presenting the results.

The numeric results for the temperature transformations at this range are shown in the following tables.

Table 2: Numerical solution for the temperature transformations for the initial temperatures between 1.5 and 2.0.

Original Temperature	1.5	1.6	1.7	1.8	1.9	2
$\mathcal{SR}$ step 1	1.7782	1.86029	1.93168	1.99621	2.06142	2.12206
$\mathcal{SR}$ step 2	1.98217	2.03561	2.08142	2.11989	2.15618	2.18872

Table 3: Numerical solution for the temperature transformations for the initial temperatures between 2.1 and 2.4, as well as the critical temperature at infinite system length and the one calculated for.

Original Temperature	2.1	2.2	2.26311	2.26919	2.3	2.4
$\mathcal{SR}$ step 1	2.17695	2.22997	2.26311	2.26628	2.28229	2.33319
$\mathcal{SR}$ step 2	2.21781	2.24575	2.26311	2.26476	2.27311	2.29939

Table 4: Numerical solution for the temperature transformations for the initial temperatures between 2.5 and 3.0.

Original Temperature	2.5	2.6	2.7	2.8	2.9	3
$\mathcal{SR}$ step 1	2.38209	2.42909	2.47451	2.51841	2.56091	2.60266
$\mathcal{SR}$ step 2	2.32421	2.34763	2.36982	2.39087	2.41092	2.43032

3.2.2 Data Preparation

The previous section transformed an initial 18 data points for according to the first two $\mathcal{SR}$ steps. Now the CNNs are to be trained on the data points provided in the previous chapter. Moreover it is crucial to implement a proper regularization term proportional to $|E(\boldsymbol{s}_i) - E(\boldsymbol{P}_i)|^2$ for the loss function. The implementation of this term is not a common loss function in any neural network library and therefore a custom implementation needs to be done. Here it is important to implement the loss in such a way, that the gradient is forwarded to the neural network to learn. This is shown to be implemented with *Tensorflow* for *Python* in section 2.5.5.

Three neural networks with a different amount of layers are implemented and tested. Each of the layers for all the different networks consists of 5 neurons. One CNN is constructed with three hidden layers [5,5,5], one with two [5,5] and the last with one hidden layer [5]. The arrays indicate the amount of neurons for each layer. This presentation is chosen within the results as well as in the *Python* implementation in the appendix. Hereby it is also analyzed how well a network can still learn and whether the amount of layers could be reduced. The networks are trained multiple times and the results with the best reconstructions are presented. The one-layered network is trained 25 times, the two-layered network is trained 10 times and the three-layered network 9 times. Now it is much more crucial to repeat the process of training, since for the 2D case it is much harder to not get stuck in a local minimum while searching for the correct weights and biases for the reconstruction. The initial system size is at $L_0 = 16$ which then gets increased with two $\mathcal{SR}$ steps to $L_1 = 32$ and $L_2 = 64$.

The other conditions are similar as for the 1D case. A dataset of $n = 20 \cdot 10^3$ not correlated spin configurations is created with the Metropolis MC algorithm. The dataset is split into a 1:1 ratio for testing and validation. The pbc padding is added accordingly. A proportionality factor, as a factor for the regularization term is chosen to be p_f $= 2 \cdot 10^{-8}$. The learning rate is once again $\eta = 10^{-3}$. The batch size is 10^3. The threshold for the early stopping is at 0 with a patience of 15 epochs. The $\mathcal{SR}$ process is repeated 10^5 times for a pool of spin configurations of the initially generated spin configurations.

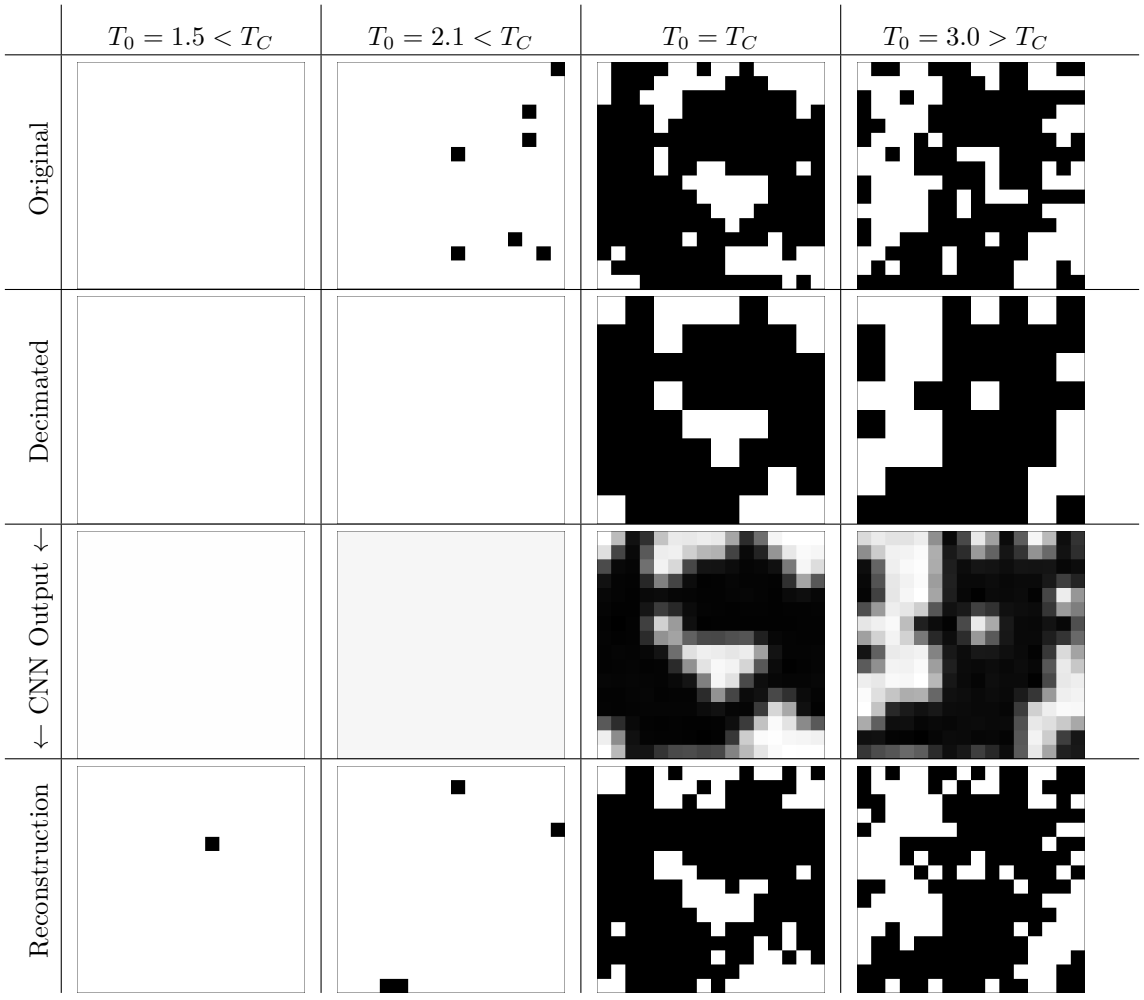
The Metropolis MC simulations are run for each data point and system size with 10^5 sweeps.

3.2.3 Visualisation of the Spin Configurations

Before presenting the results of the expectation values and histograms of the observables from the simulation, the resulting spin configurations from one of the neural networks are shown. The largest neural network with 3 layers, each 5 neurons, is chosen for creating the spin configuration pictures for this methodology. This is shown in two tables, one for the learning procedure and one for the $\mathcal{SR}$ procedure. The first shows the initial spin configuration before and after the decimation. The CNN output of the decimated spin configuration as input. Finally the reconstruction of a spin configuration after the decimation. The other table shows the $\mathcal{SR}$ procedure, by enlarging an initial spin configuration two times with the $\mathcal{SR}$ procedure.

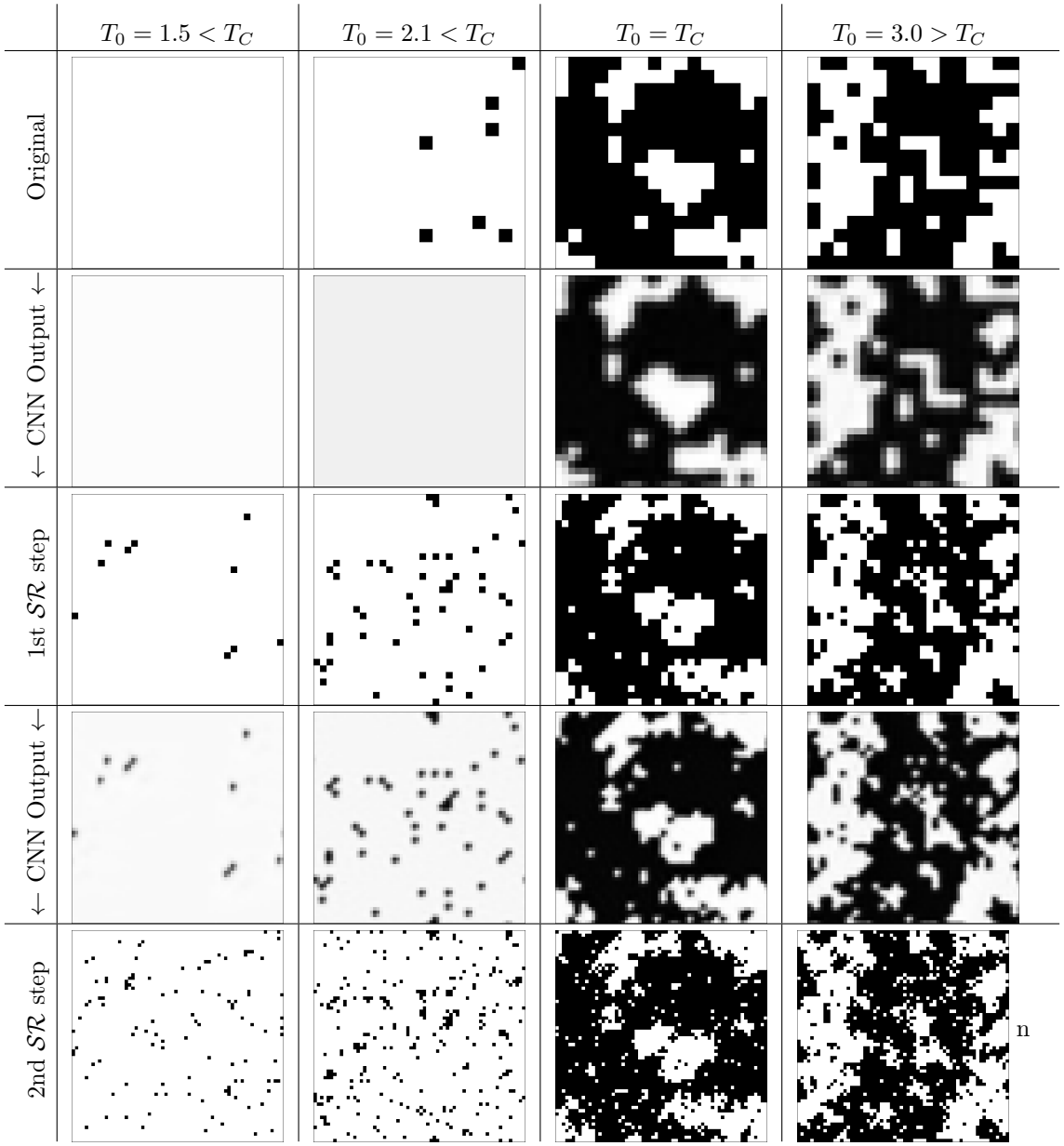
The original spin configuration is created with the Metropolis Monte Carlo algorithm at size 16×16. The decimated spin configuration with the deterministic decimation procedure, reducing its size to 8×8. For the pictures in the row of the CNN output, the decimated spin configuration is enlarged to a system size of 16×16 by simple block duplication. It is then treated as the input for the CNN and its output is then shown as a probability distribution. This distribution is not an integer but a continuous value between 0 and 1. Therefore the CNN output pictures are shown as a gray scale. Finally the reconstruction is shown, which is the realization of the probability from the CNN output.

Table 5: Visualisation of one spin configuration for four temperatures of the reconstruction procedure. First the initial spin configuration. Afterwards after the decimation. Then after the convolution from the CNN. Finally the reconstruction of the initial spin configuration.



Previously the result of the learning back to the original spin configuration was shown. Here it is the $\mathcal{SR}$ to larger system size, with the original spin configuration to be the same as in the previous table. The 1st CNN output in this table is the output layer of the CNN after simple block duplication of the original spin configuration, resulting at a system size of 32×32. The spin configurations at the 1st $\mathcal{SR}$ step, is the realization of the probability distribution of the CNN output. The last two steps are now repeated for a 2nd $\mathcal{SR}$ step. In the end the initial spin configuration at size 16×16 is resolved to a system size of 64×64.

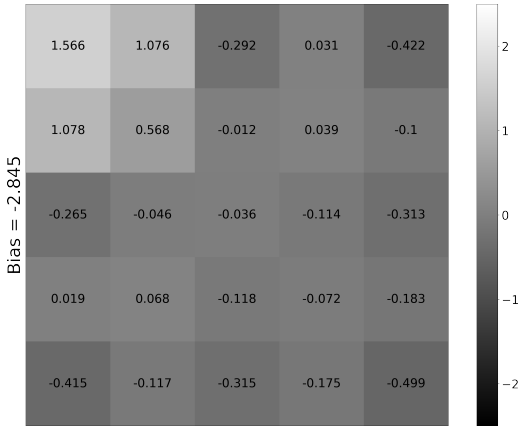
Table 6: Visualisation of one spin configuration for four temperatures for the $\mathcal{SR}$ procedure. First the initial spin configuration. Then the output after the convolution of the CNN for the simple block duplication of the initial spin configuration. Afterwards the realization of the probability distribution. The last two steps are repeated for a second $\mathcal{SR}$ step.



3.2.4 Visualisation of the Kernels

The trained kernel of the neural network is visualized and the numerical values of the weights and biases are shown for all three CNN configurations. The trained one, two and three layered networks each are shown for the calculated critical temperature of $T_C^{(16\times16\to8\times8)} = 2.26311$. In this case, the trained neural network can be used for any $\mathcal{SR}$ step, since a temperature transformation of this critical temperature results in the same temperature. It is noted here that the kernel is not centered around the input. According to the visualisation of the kernel, the kernel rather extends to the right and bottom of the input.

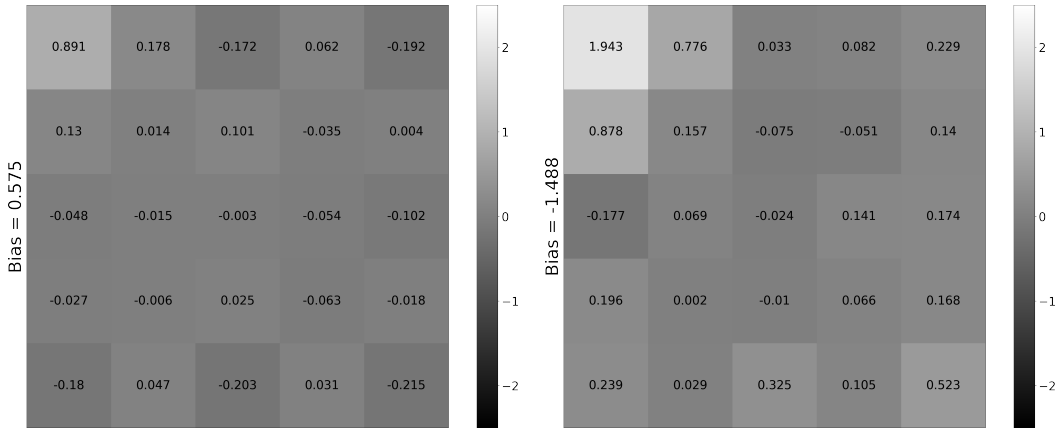
Here is the kernel of the one layered CNN with 26 of the kernel parameters in total. It is visualized in terms of its weights and the bias is shown to the left.



(a) 1st Kernel

Fig. 28: The one-layered kernel trained on the critical temperature.

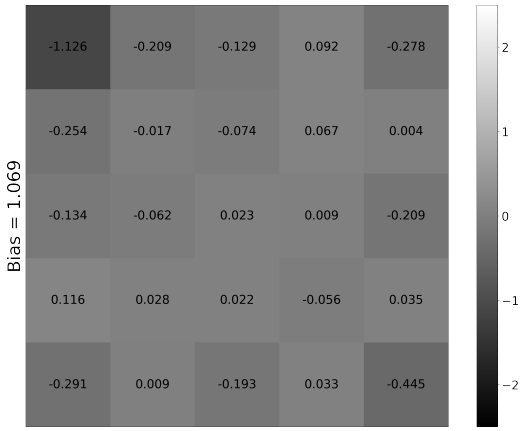
Now the kernel of the two layered CNN with 25 weights and 1 bias for every kernel is shown. Analogous to the one-layered case.



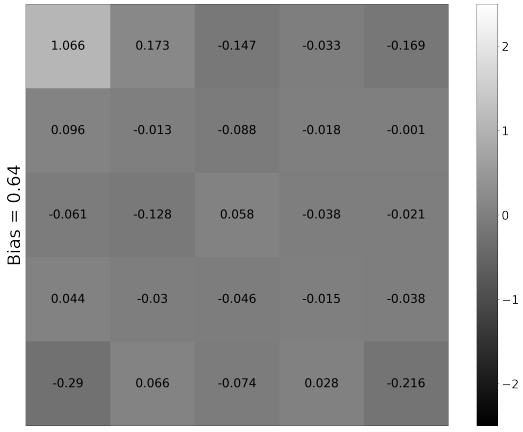
(a) 1st Kernel *(b)* 2nd Kernel

Fig. 29: The two-layered kernel trained on the critical temperature. Left is the first kernel and to the right is the second kernel.

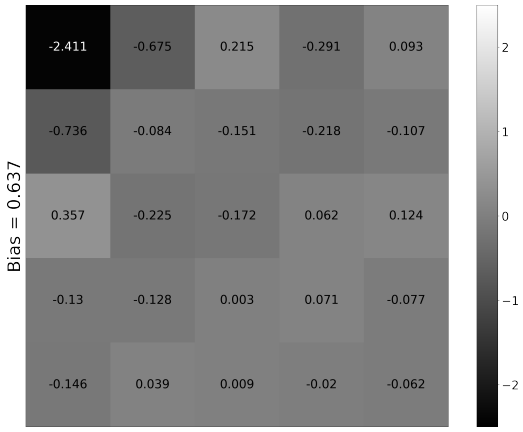
Finally the kernel of the three layered CNN with 25 weights and 1 bias in each of the kernels is visualized.



(a) 1st Kernel



(b) 2nd Kernel



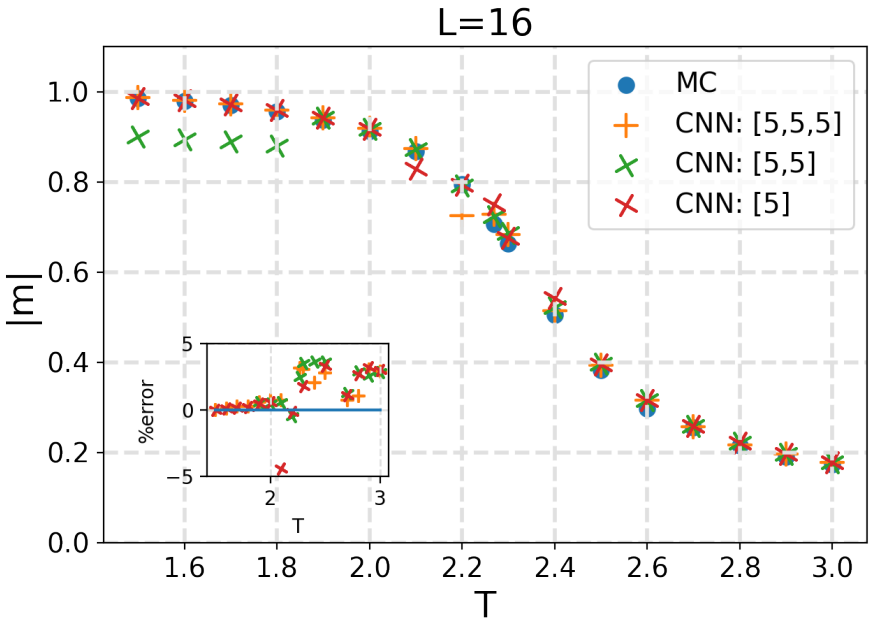
(c) 3rd Kernel

Fig. 30: The three-layered kernel trained on the critical temperature is shown in the correct order.

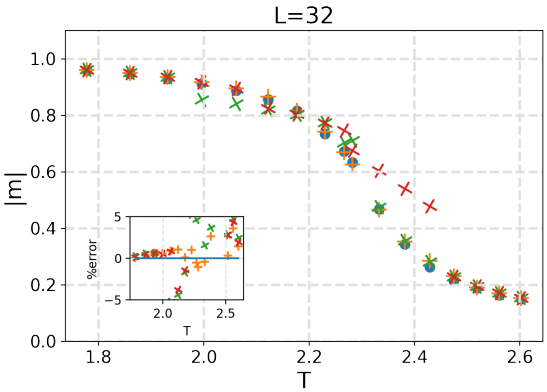
3.2.5 Results

The results for the 2D case are presented analogous to the ones for the 1D case in section 3.1.2. The original temperature points are chosen from section 3.2.1.

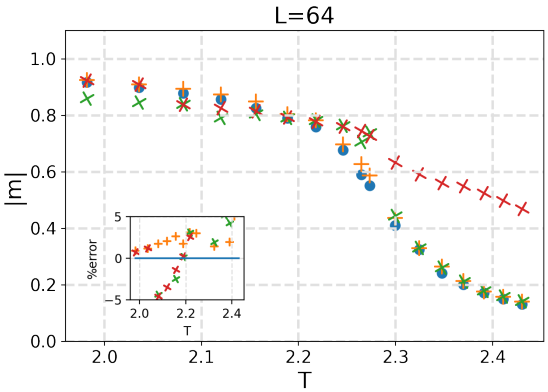
Here the expectation value of the absolute magnetization is shown. At first there is the reconstruction back to the original temperature. Afterwards the first two $\mathcal{SR}$ steps. The comparison is done according to the results of the Metropolis MC simulation results. The amount of layers and neurons are shown as an array for each of the CNN reconstruction simulations.



(a) Reconstructing to the original system size



(b) 1st $\mathcal{SR}$ step

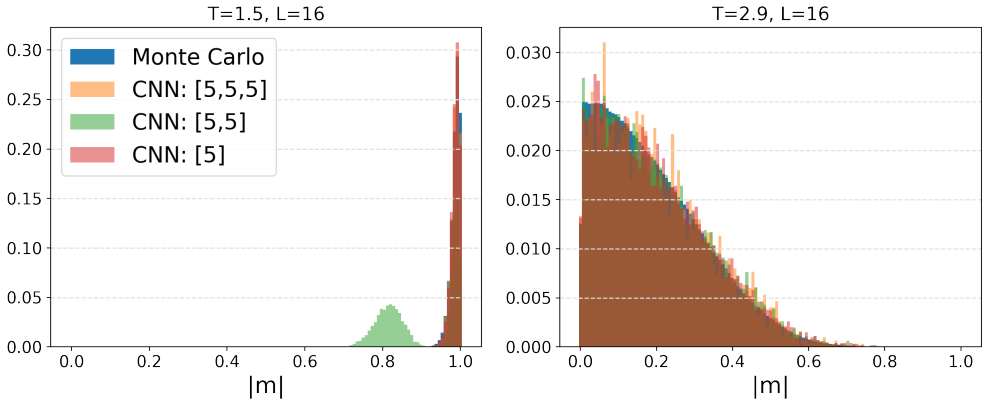


(c) 2nd $\mathcal{SR}$ step

Fig. 31: Results for the expectation value of the absolute magnetization of the reconstructions with three different CNNs. Additionally the first two super-resolutions from $L = 16$ up to $L = 64$.

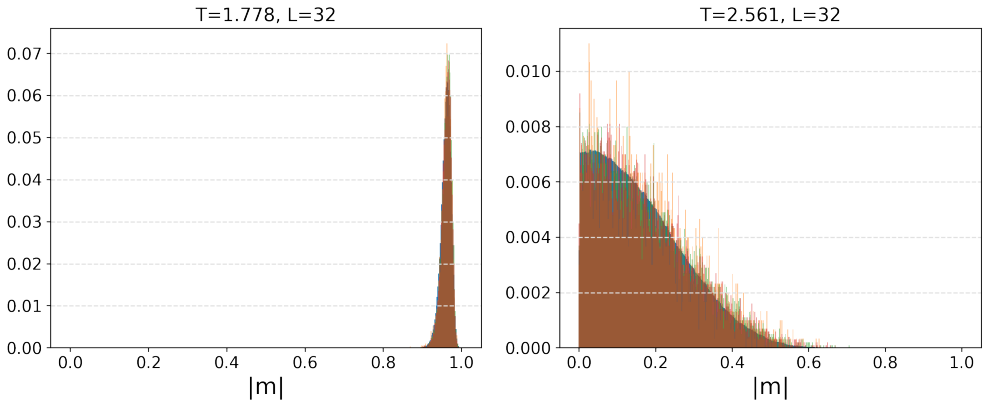
Histograms for the distribution of the magnetization for a cold and a hot temperature case.

It is noted here that for reasons of archiving a better overview with the histogram plots, they are now differently plotted that for the 1D case. Before they were plotted with a uniform bin size and the bins being next to each other. Now they are shown on top of each other with the bin sizes corresponding to every possible observable on that system size.



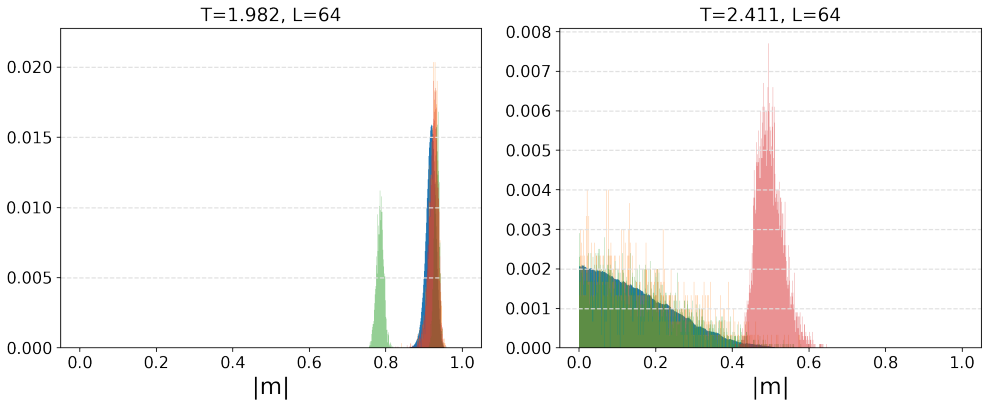
(a) Reconstruction, cold case

(b) Reconstruction, hot case



(c) 1st $\mathcal{SR}$ step, cold case

(d) 1st $\mathcal{SR}$ step, hot case

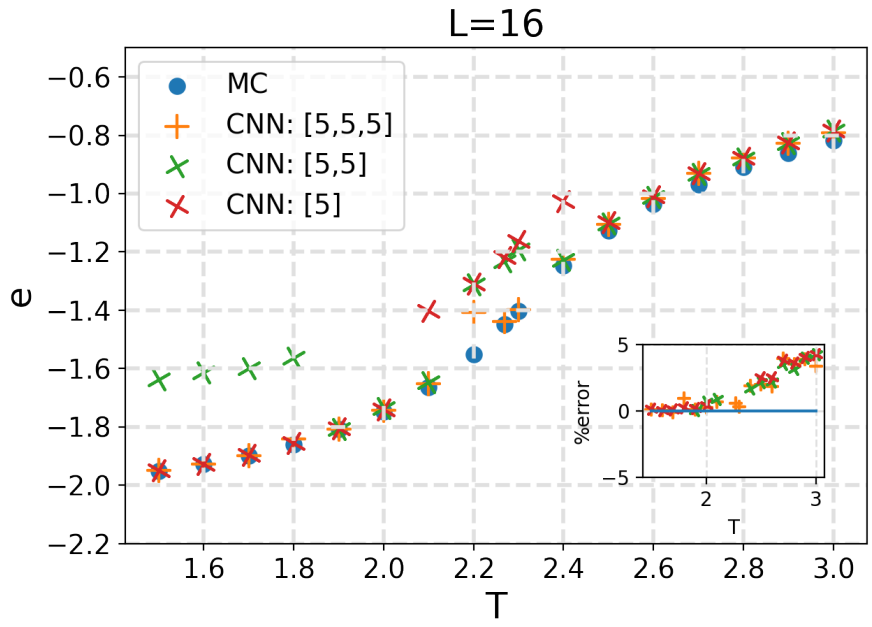


(e) 2nd $\mathcal{SR}$ step, cold case

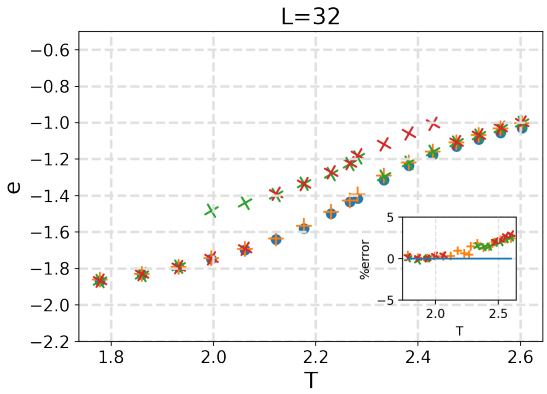
(f) 2nd $\mathcal{SR}$ step, hot case

Fig. 32: Histograms of a cold (left) and a hot (right) case for the expectation of the absolute magnetization in the 2D Ising case. Showing the reconstruction as well as the first two $\mathcal{SR}$ steps.

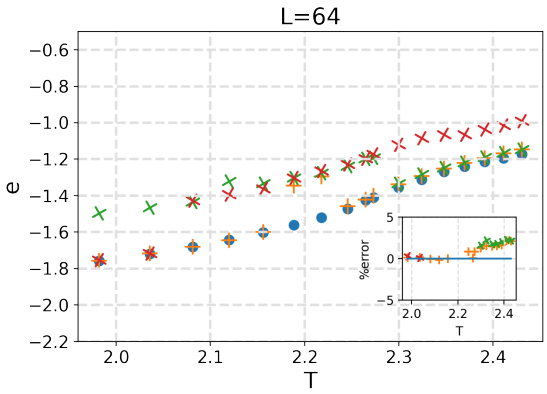
Analogous to the previous presentation of the magnetization, the expectation of the energy is now presented. First for the reconstruction to the original system size. After that for the first 2 $\mathcal{SR}$ steps.



(a) Reconstructing to the original system size



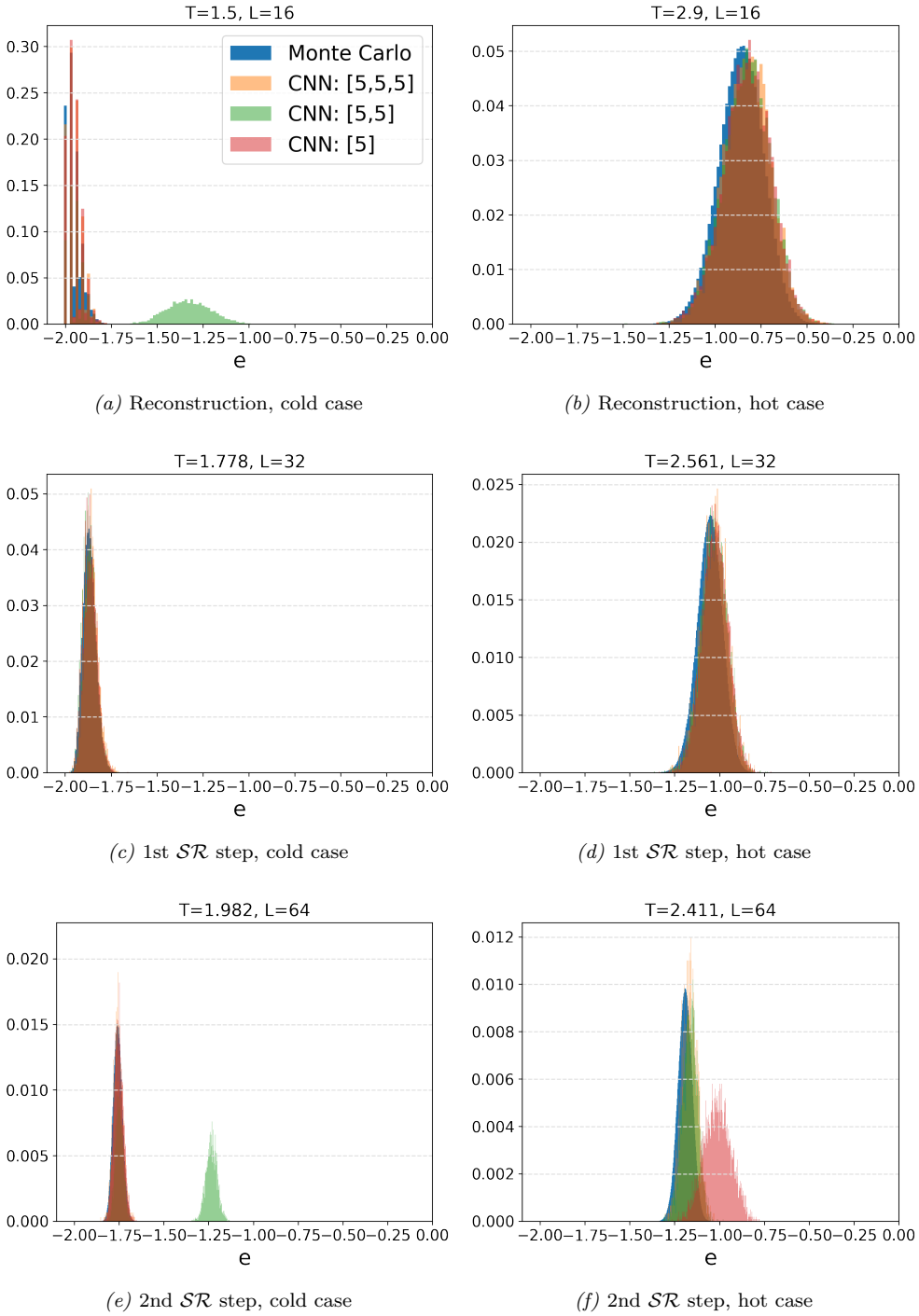
(b) 1st $\mathcal{SR}$ step



(c) 2nd $\mathcal{SR}$step

Fig. 33: Results for the expectation value of the energy of the reconstructions with the CNN. Additionally the first two super-resolutions from $L = 16$ up to $L = 64$.

Here is a plot of histograms for the distribution of the energy for a cold and a hot temperature case. Those temperatures are appropriately transformed with each $\mathcal{SR}$ step.



(a) Reconstruction, cold case

(b) Reconstruction, hot case

(c) 1st $\mathcal{SR}$ step, cold case

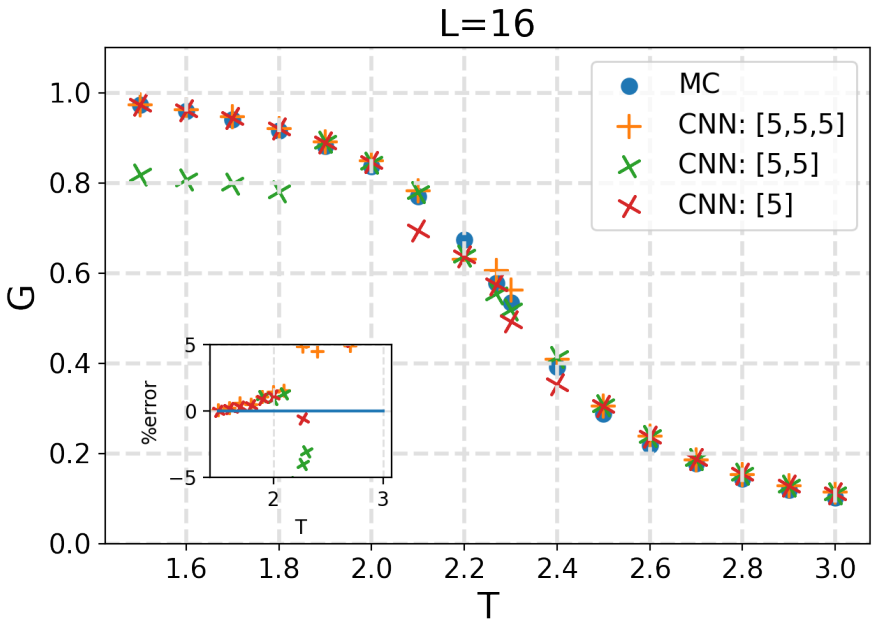
(d) 1st $\mathcal{SR}$ step, hot case

(e) 2nd $\mathcal{SR}$ step, cold case

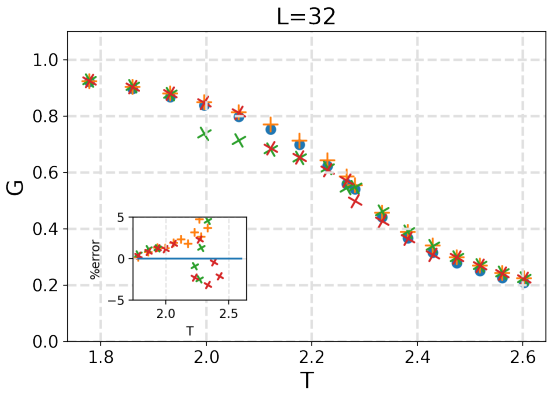
(f) 2nd $\mathcal{SR}$ step, hot case

Fig. 34: Histogram plots of the cold (left) and hot (right) case for the expectation of the energy in the 2D Ising case. Showing the reconstruction to the original system size as well as the first two $\mathcal{SR}$ steps.

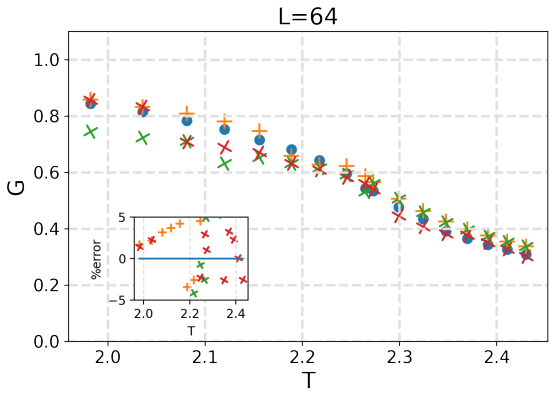
Analogous to the expectation values of the absolute magnetization and the energy, here the expectation value for the two point spin correlation at distance 3 is shown. First for the reconstruction to the original system size. After this for the first 2 $\mathcal{SR}$ steps.



(a) Reconstructing to the original system size



(b) 1st $\mathcal{SR}$ step



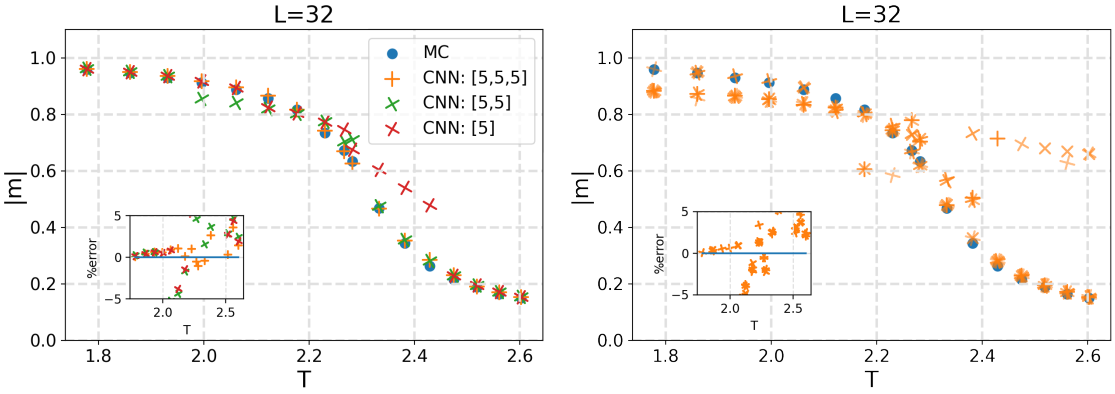
(c) 2nd $\mathcal{SR}$ step

Fig. 35: Results for the expectation value of the two point spin correlation with distance $|r| = 3$ of the reconstructions with the CNN. Additionally the first two $\mathcal{SR}$ from $L = 16$ up to $L = 64$.

3.2.6 Investigation of the unpolished Results

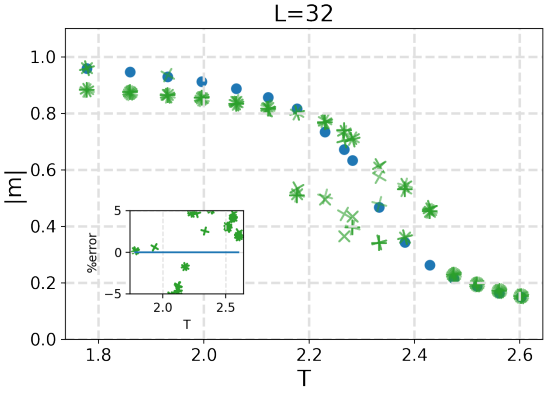
As previously mentioned, it is suggested for the neural network to train with multiple iterations at different initializations of the weights and biases for each of the data points. Then the best reconstruction of the data point is chosen for the final results. However some data points show a larger deviation than expected, this can be also seen in the previous results section. For the CNN configurations investigated, this happens more often for the smaller ones than for the larger neural network.

In this section, the first six training iterations for every CNN setup is shown separately for the first $\mathcal{SR}$ step. The observables in showcase are the absolute magnetization and and the energy. For a better comparison, the previous polished results are also shown. At first here for the magnetization.

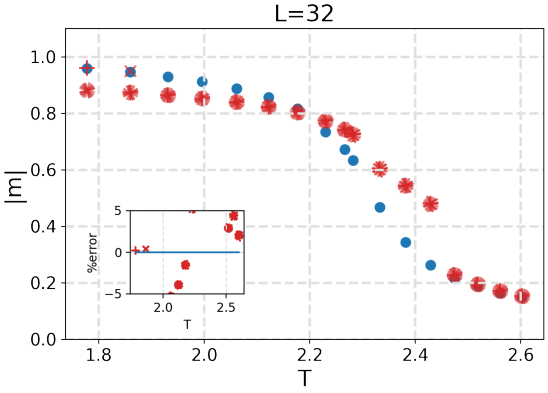


(a) Polished results

(b) CNN: [5,5,5], first six training iterations



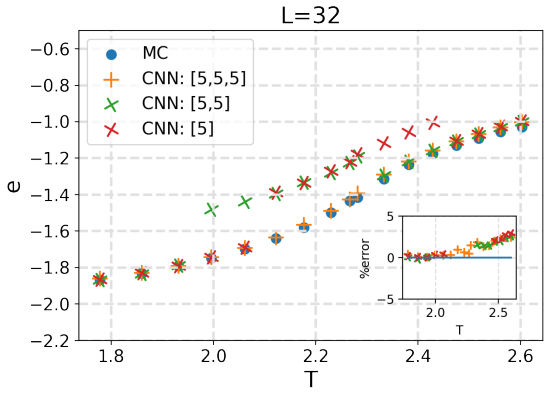
(c) CNN: [5,5], first six training iterations



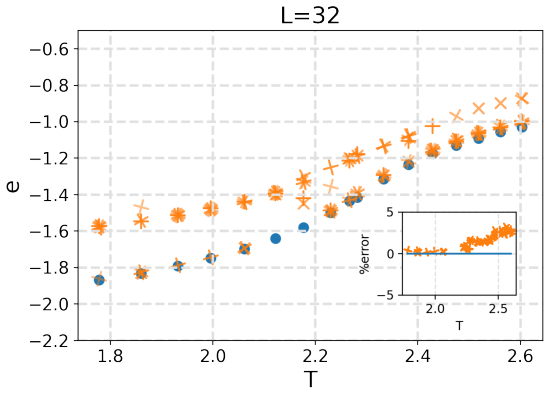
(d) CNN: [5], first six training iterations

Fig. 36: Results for the expectation value of the absolute magnetization with the first $\mathcal{SR}$ step. Showing the first six training iterations as well as the polished results for each neural network configuration.

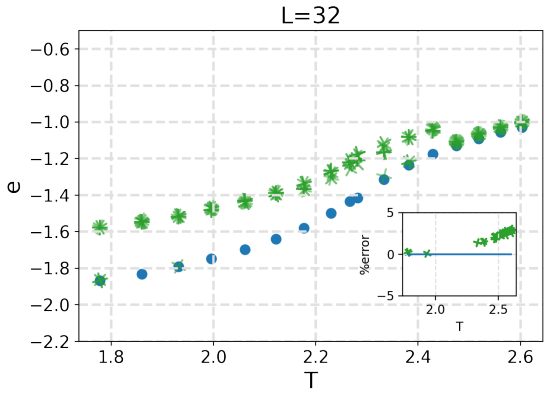
Now the same presentation is shown for the energy case.



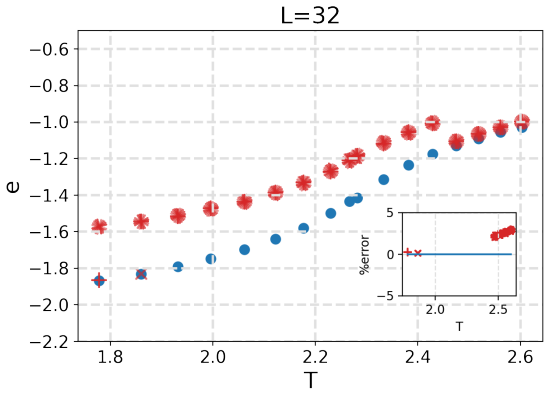
(a) polished results



(b) CNN: [5,5,5], first six training iterations



(c) CNN: [5,5], first six training iterations



(d) CNN: [5], first six training iterations

Fig. 37: Results for the expectation value of the energy with the first $\mathcal{SR}$ step. Showing the first six training iterations as well as the polished results for each neural network configuration.

It is noted that the final polished results which are shown here are created according to more than six training iterations. This is described in section 3.2.2. Therefore it might be that for some points where the CNN does not manage to reconstruct the data point in the first six training but does so in the polished results, it could have managed to do so in the further iterations.

It is also noted that for choosing the best weights and biases from the trained CNNs on every data point, it is not required to actually resolve to larger system sizes. The network is always trained on the original spin configurations, here at system size $L_0 = 16$. It is selected accordingly on how well it performs on the validation dataset of this training. The weights and biases which perform the best reconstruction on the validation data set are then chosen for the polished data in the final $\mathcal{SR}$ procedure.

3.2.7 Reconstruction from smaller System Sizes

For further investigation of the functionality of this method, neural networks are trained from much smaller system sizes, to see if the network can still reconstruct the physical properties of a larger system. For this the initial spin configuration is set to $L = 4$ and the decimated spin configuration is therefore at size $\tilde{L} = 2$. The network then reconstructs the spin configuration at $L_0 = 4$ as well as performs the first $\mathcal{SR}$ step to $L_1 = 8$.

Analogous to the section 3.2.1 the temperature transformation is now performed but according to the decimated system size of $\tilde{L} = 2$. For a better comparison of the differences for the two temperature transformations, the previous case of the decimated system of length $\tilde{L} = 8$ is also presented.

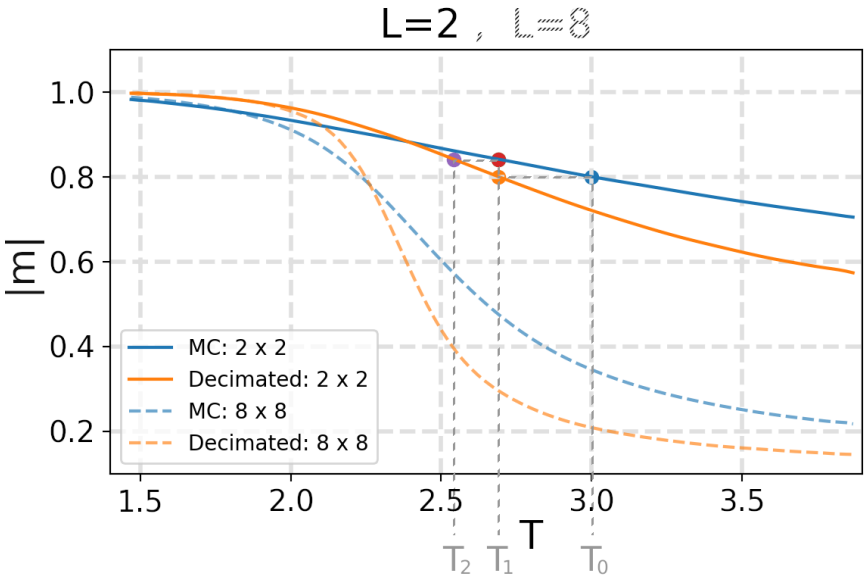


Fig. 38: Solution of the absolute magnetization of the decimated spin configurations and the Metropolis algorithm. Displaying the numerical method for the temperature transformation. A previous solution, shown in Fig 18 is shown on top as a comparison.

The initial temperature is chosen between 1.5 and 3.0 with a step size of 0.1. Same as before. The critical temperature is now calculated to be $T_C^{(4\times4\to2\times2)} = 2.38977$. The numerical results for the first temperature transformation are shown as well as the difference towards the previous numeric result in section 3.2.1 for the first temperature transformation. The positive difference is colored in green and the negative difference is colored in red. However there are only positive differences towards the previous results. The difference for the critical temperature is not shown.

Table 7: Numerical solution for the temperature transformations for the initial temperatures between 1.5 and 2.0.

Original Temperature	1.5	1.6	1.7	1.8	1.9	2
$\mathcal{SR}$ step 1	1.84285	1.9122	1.98005	2.04697	2.11016	2.17016
Difference to Table 2	↳+0.0647	↳+0.0519	↳+0.0484	↳+0.0508	↳+0.0487	↳+0.048

Table 8: Numerical solution for the temperature transformations for the initial temperatures between 2.1 and 2.4. Including the calculated critical temperature and the critical temperature of an infinitely sized system.

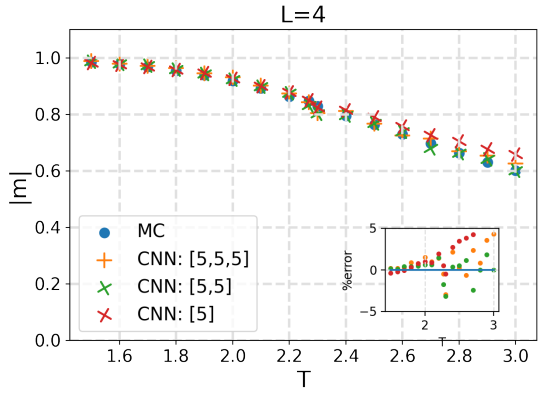
Original Temperature	2.1	2.2	2.26919	2.3	2.38977	2.4
$\mathcal{SR}$ step 1	2.22944	2.28627	2.32414	2.34089	2.38977	2.39534
Difference to Table 3	↳+0.0525	↳+0.056	-	↳+0.059	-	↳+0.0622

Table 9: Numerical solution for the temperature transformations for the initial temperatures between 2.5 and 3.0.

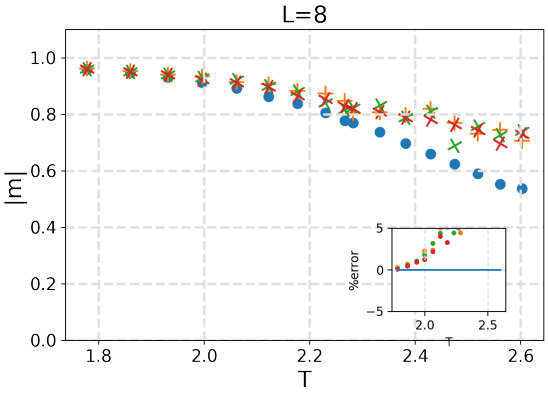
Original Temperature	2.5	2.6	2.7	2.8	2.9	3
$\mathcal{SR}$ step 1	2.44871	2.49878	2.54627	2.59419	2.64329	2.69101
Difference to Table 4	↳+0.0666	↳+0.0697	↳+0.0718	↳+0.0758	↳+0.0824	↳+0.0884

Here is a quick revision on data preparation. The simulations are performed with mostly the same conditions as stated in section 3.2.2. The simulations are therefore also performed with three different setup of CNNs with either three, two or one layer(s), each with a kernel size of 5×5. Now the differences are stated. Once again the networks are trained multiple times for every data point on every system size and with every CNN layer configuration and the best reconstruction is chosen. The training with the three layered CNN is repeated 25 times, 30 times with the two layered CNN and 50 times with the one layered one. The super-resolution is repeated for 10^5 spin configurations.

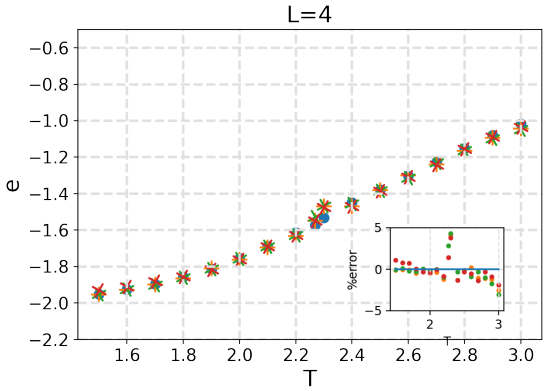
The results for the trained neural network at a decimated system length of $\tilde{L} = 2$ are presented similar to the previous results. However here showing all three observables on one page for the reconstruction as well as the first $\mathcal{SR}$ step.



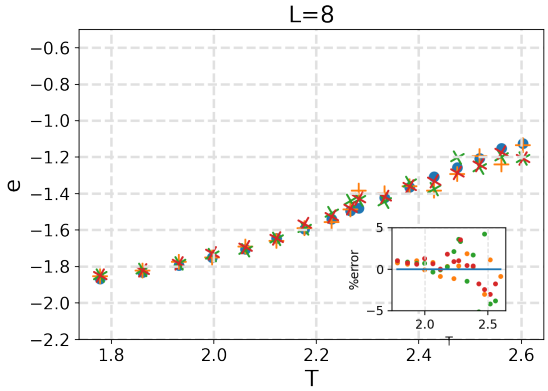
(a) Reconstruction, magnetization



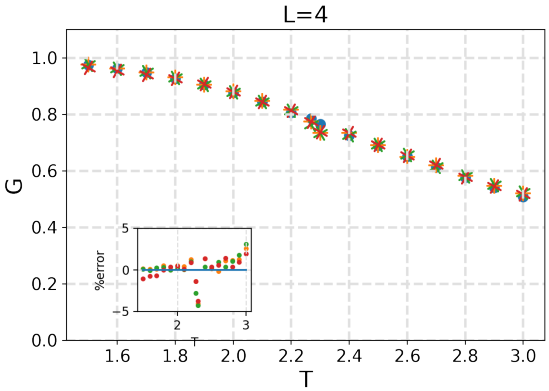
(b) 1st $\mathcal{SR}$ step, magnetization



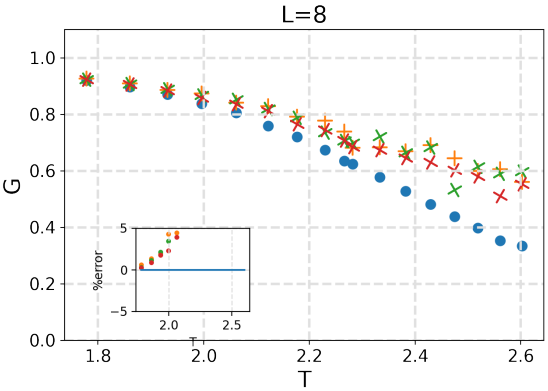
(c) Reconstruction, energy



(d) 1st $\mathcal{SR}$ step, energy



(e) Reconstruction, two point spin correlation



(f) 1st $\mathcal{SR}$ step, two point spin correlation

Fig. 39: Plots of the results for the expectation of the absolute magnetization, the energy and the two point spin correlation at distance $|r| = 3$ for the reconstruction and the first $\mathcal{SR}$ step.

3.2.8 Discussion

The previous sections cover the numeric solution for the temperature transformation for the $\mathcal{SR}$ steps, multiple visualisations for this methodology of the spin configurations and the kernel, moreover it covers the results for different system size settings of the 2D case. As well as the unpolished results for one system size setting. Three networks are trained to reconstruct a decimated 8×8 spin configuration to its original 16×16 system size. The networks have either one, two or three layers each with kernels of size 5×5. This is presented in the plots according to the different arrays. The results of the three layered network are visualized for the reconstruction and the $\mathcal{SR}$ as well as for the output layer of the CNN for different temperatures. The kernel of all networks are visualized for the critical temperature. Then the results of the reconstruction and two $\mathcal{SR}$ steps are shown. The same network constellations are also trained to reconstruct a set of 2×2 spin configurations to its original 4×4 set of spin configurations. Here the results are shown for the reconstruction but only the first $\mathcal{SR}$ step.

The numeric solutions for the temperature transformation is shown in section 3.2.1 for 18 initial data points and 2 $\mathcal{SR}$ steps. A critical temperature is calculated to be $T_C^{(16\times16\to8\times8)} = 2.26311$ which is slightly below the critical temperature at infinite system size. All other temperatures are transformed to temperatures closer to this critical temperature for every further $\mathcal{SR}$ step.

The visualisation of the spin configurations in section 3.2.3 for the reconstructions of two temperatures below the critical temperature show no capture of any initial features. This is since the decimation removed all spins pointing in another direction than the majority. Therefore the reconstruction results in a couple random spins for a few sites to point in another direction than the majority. This has no apparent order. A similar behavior is shown for resolving to larger configurations. This is to be expected, since here the temperature is too cold for spins forming a correlated cluster. When looking at the two higher temperatures, then there are actually spin domains to be observed. The neural network captures this spin domains and reconstructs them roughly back to the original. When resolving to larger system sizes, the original spin configuration is used as in input and enlarged. Here it is shown to capture the regions in a very similar fashion but fills them with greater details.

When looking at the visualisation of the kernels in section 3.2.4, it shows the trained kernels for the different neural networks at critical temperature. The very top left value is for any kernel the most significant value, since its the furthest apart from 0. For the one and two layered network, its neighboring weights also play a more significant role. However all other weights seem more and more randomly varying for some small values around 0. In the three layered case, the latter behavior is also seen but the neighboring weights play much less of a significant role. It is clear for all cases, that the top left value plays a significant role, since it shows that the spin direction at the initial site plays the most important role for the reconstructed spin after the convolution. Prior to presenting this results, experiments to center the convolution have been conducted. But this did not influence any results and therefore was not presented in this thesis.

The network was capable of roughly reconstructing and resolving the appropriate spin configurations according to section 3.2.5. The network was trained multiple times but still some single data points show to deviate significantly from the Metropolis MC solutions. This is especially the case for the smaller sized neural networks. Generally the networks are capable of capturing configurations at lower temperatures but for larger ones it becomes much more difficult. Up until the critical temperature, the largest network was capable to reconstruct with under a 1% error. At the critical temperature, the energy was reconstructed still quite well, but the magnetization is much more problematic. Here the choice of the proportionality factor becomes quite crucial. For a larger proportionality factor, the network learns the reconstruction of the energy better and vice versa at smaller proportionality factors for the magnetization. The proportionality factor of $2e{-}8$ is chosen via trial and error as a good compromise between the both. But there was a great tolerance in prior testing for choosing a different proportionality factor.

Generally for the data points which managed to reconstruct the observables accurately to a certain degree, it can be compared to the results of the paper of Efthymiou *et al.* [4, p. 5]. There for larger temperatures, the CNN also turned out a bit above the graph and around the critical temperature, the error is shown to be the largest. However the paper did not present the error for the 2D case for every data point and therefore it cannot be compared numerically.

Since the network is sometimes able to reconstruct the observables and sometimes not, the simulations are run multiple times. For a better understanding, the unpolished results are also presented for the first six iterations. Generally the smaller CNN require more epochs for training until the early stopping condition but the training procedure is also much quicker. This is because a lot less variables need to be derived. However the small neural networks also fail to reconstruct the observables more often. Larger neural networks are more consistent in reconstructing the data points but still suffer from this problem. It is imaginable that for the data points in which the CNN actually didn't manage to reconstruct the observables at all, it is still possible for the neural network to find a proper solution. This however cannot be said with complete certainty. The reason for a false reconstruction is thought to be due to the different initialization of weights and biases. This is the only difference between the training iterations.

To investigate the boundaries of this of this methodology, it is also tested in section 3.2.7 on a smaller system size. The same three setups of CNNs are trained to reconstruct a set of spin configuration of size 2×2 to its original 4×4 spin configuration. Before that, the solution for the numeric temperature transformation is also calculated. It now deviates from the temperature transformation calculated at larger system sizes. It would be supposed to have a similar transformation, if the $\mathcal{SR}$ would be applicable universally, independent of system size. Therefore this corresponds necessarily to a systematic error when resolving to larger system sizes. The results show the reconstruction and the first $\mathcal{SR}$ step. After training, the network is able to reconstruct the energy and two point spin correlation function with an accuracy of 3% on the temperatures other than around the critical temperature. The magnetization is reconstructed with less than 2% until around the critical temperature. The error for the reconstruction of the magnetization at critical temperature and above is much higher. The network is capable of the first $\mathcal{SR}$ step at small temperatures but not for larger temperatures. This applies to all of the three observables. Therefore it indicates that for very small system sizes, a trained neural network is not capable of super-resolving the Ising model. The reason for this is indicated to be due to the finite size effects of the Ising model. It is conceivable that the neural network made an error in training procedure, however, since it resolved back to the original system size much better than for super-resolving to larger system sizes, where this becomes less of an option.

3.3 3D Simulation

The results for CNN on the 1D as well as the 2D case have shown to be able to both reconstruct the physical properties and resolve to larger system sizes. Now the method is implemented as well for the 3D case.

The 3D case is very similar to the 2D case in terms of the implementation since the procedure for the temperature transformation works very analogous. The regularization term is also implemented similarly, just extended in the third dimension. However it is much more difficult to find a proper proportionality factor for the neural network to adapt a proper learning. For this reason instead of showing different neural network configuration, the results are shown for different proportionality factors.

3.3.1 Temperature Transformation

At first it is required to find a proper temperature transformation. This is done in the same fashion as in the 2D case in section 3.2.1. Once again there are two simulations, here one is at $L = 8$ which then is decimated to $\tilde{L} = 4$ and the other originally simulated at the reduced system length. 80 data points are simulated with $30 \cdot 10^3$ sweeps each. For both graphs a polynomial fit is created of order 15. Both of those functions can be resolved to a temperature transformation for the $\mathcal{SR}$ procedure. This is shown in the following figure for a temperature at $T_0 = 3.65$.

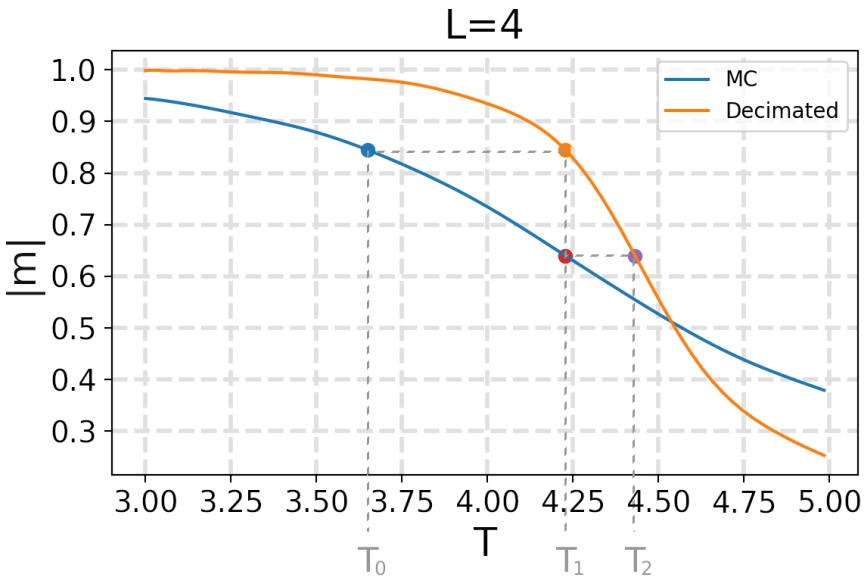


Fig. 40: Solution of the absolute magnetization of the decimated spin configurations and the Metropolis algorithm at 3D. Displaying the numerical method for the temperature transformation.

The initial temperature range is chosen between 3.5 and 5.0, with a step size of 0.15. The critical temperature is also considered as a data point for the 3D simulation. Resulting in a total of 12 data points. The critical temperature is calculated to be 4.54389. The critical temperature for the 3D case at an infinitely large system is at 4.51080.

The numeric results for the temperature transformations at this range are shown in the following table. Here for the original temperature and the first $\mathcal{SR}$ step.

Table 10: Numerical solution for the temperature transformations for the initial temperatures between 3.5 and 4.25.

Original Temperature	3.5	3.65	3.8	3.95	4.1	4.25
$\mathcal{SR}$ step 1	4.17019	4.22865	4.28285	4.3342	4.38687	4.43961

Table 11: Numerical solution for the temperature transformations for the initial temperatures between 4.4 and 5.0. Also includes the calculated critical temperature.

Original Temperature	4.4	4.53831	4.55	4.7	4.85	5
$\mathcal{SR}$ step 1	4.49095	4.53831	4.54233	4.59307	4.64	4.68724

3.3.2 Data Preparation

The previous section transformed an initial 12 data points according to the first $\mathcal{SR}$ step. For all of these data points, the now constructed neural network is to be trained. A big challenge for the 3D case to find the correct proportionality factor for the regularization term. There are different CNN setups constructed, however they only differ with their proportionality factors.

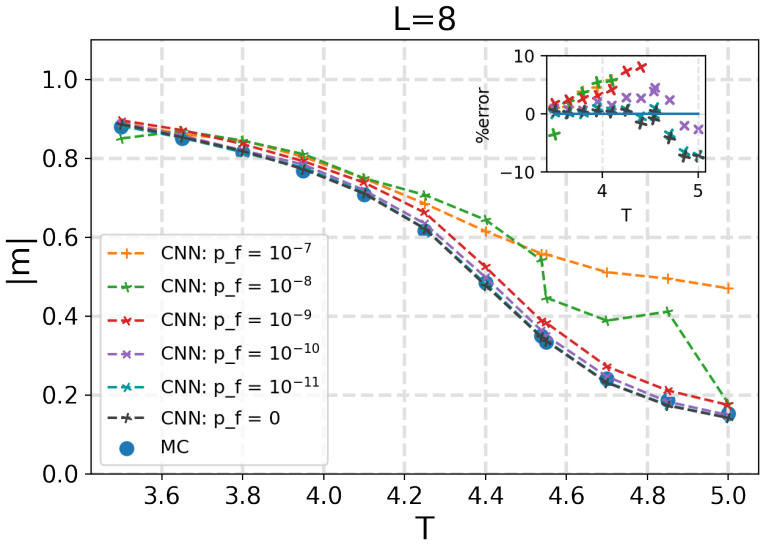
Six different neural network setups are constructed with the following similarities. Each of the networks consists of three 3D convolutional kernels. Every kernel is of length 3. A dataset of $n = 10^4$ not correlated spins is created with the Metropolis MC algorithm. The dataset is split into a 1:1 ratio for testing and training. The pbc is added accordingly. The learning rate is $\eta = 10^{-3}$. The batch size is 10^3. The threshold for the early stopping is set to 0, with a patience of 15 epochs.

The difference of the six networks is the proportionality factor p_f. One CNN is trained with p_f = 10^{-7}, one with p_f = 10^{-8}, one with p_f = 10^{-9}, p_f = 10^{-10}, p_f = 10^{-11} and one with p_f = 0. For every data point at every temperature at every system size and for every constructed CNN setup with different proportionality factor, six networks are trained and the best one is chosen. The reconstruction procedure is repeated for 10^5 times on the pool of initial spin configurations.

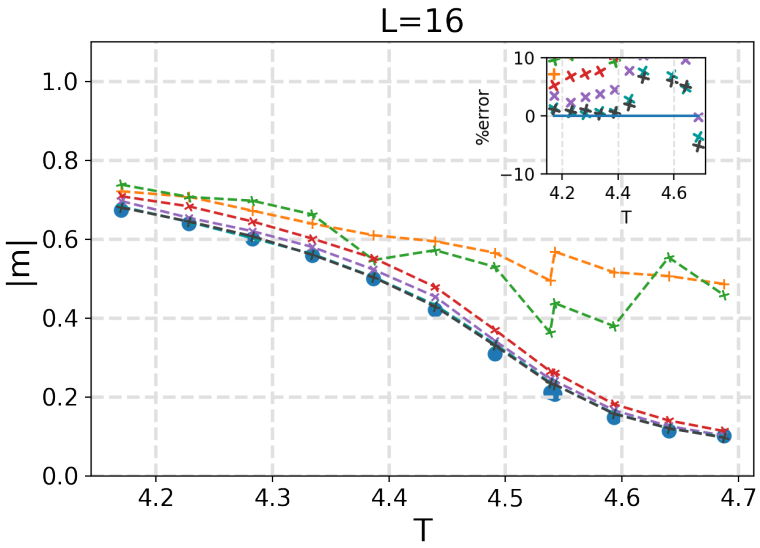
3.3.3 Results

The results for the 3D case are presented analogous to the ones for the 2D case in section 3.2.5 and the 1D case in section 3.1.2. The original temperature points are taken from section 3.3.1.

Here the expectation value of the absolute magnetization is shown. At first the reconstruction back to the original temperature. Afterwards the first $\mathcal{SR}$ step. The comparison is done according to the results of the Metropolis MC simulation results.



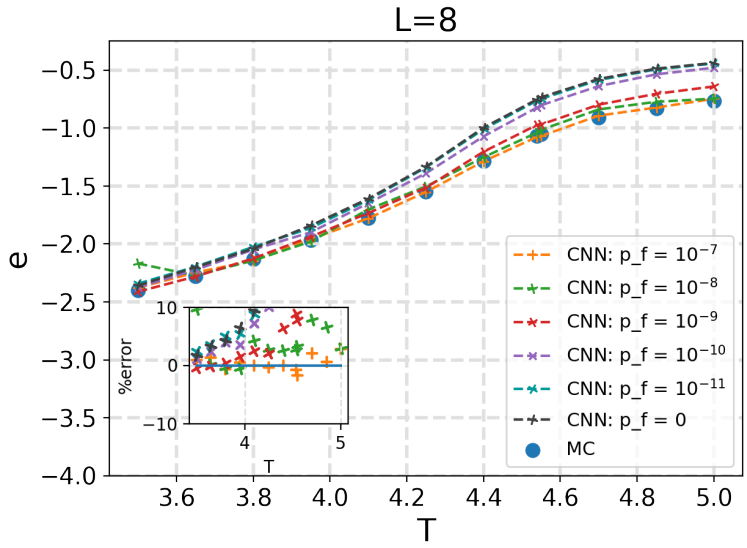
(a) Reconstructing to the original system size



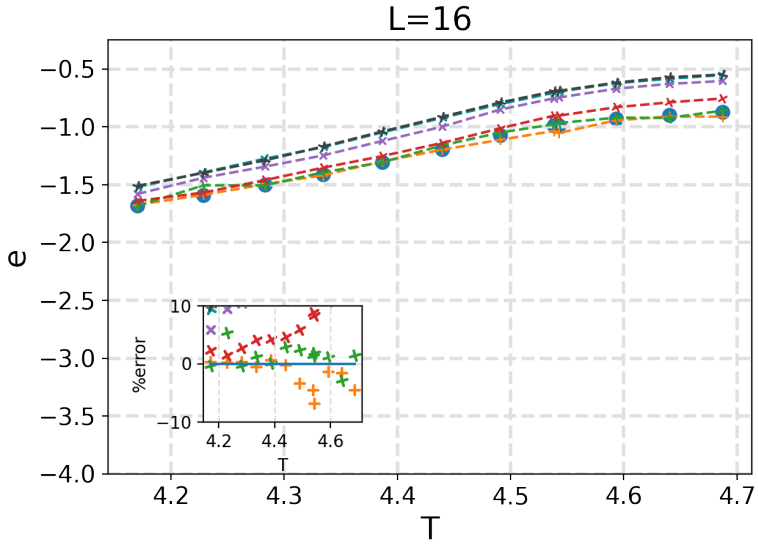
(b) 1st $\mathcal{SR}$ step

Fig. 41: Results for the expectation value of the absolute magnetization of the reconstructions with the CNN. Additionally the first $\mathcal{SR}$ from $L = 8$ up to $L = 16$. The results are shown for different proportionality factors p_f.

Analogous to the previous presentation of the magnetization, the expectation of the energy is now presented. First the reconstruction to the original system size. After that for the first $\mathcal{SR}$ step.



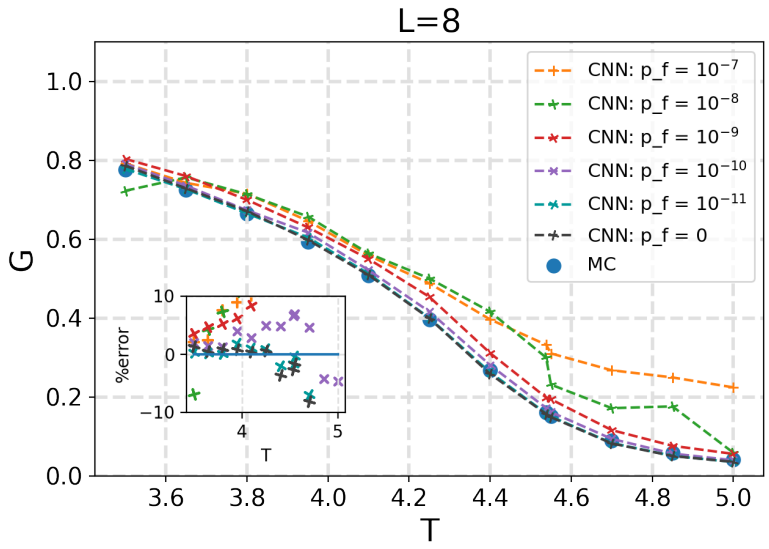
(a) Reconstructing to the original system size



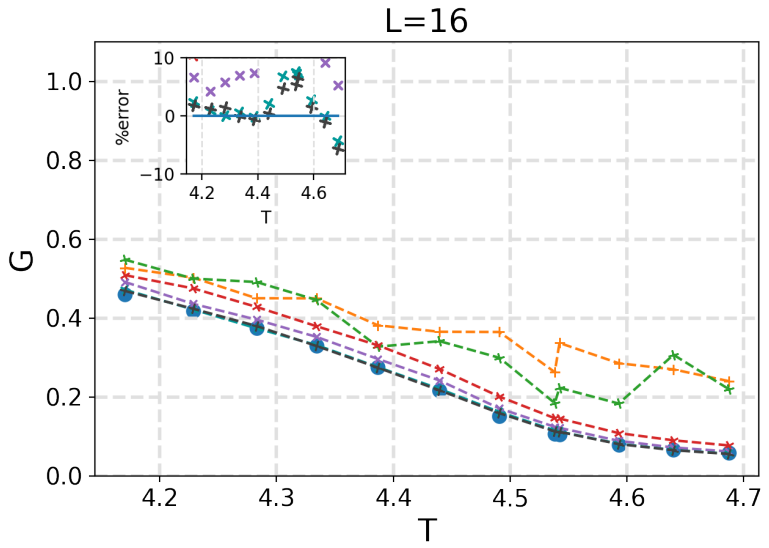
(b) 1st $\mathcal{SR}$ step

Fig. 42: Results for the expectation value of the energy of the reconstructions with the CNN. Additionally the first $\mathcal{SR}$ from $L = 8$ up to $L = 16$. The results are shown for different proportionality factors p_f.

Analogous to the previous two observables, here the expectation value for the two point spin correlation function at distance 3 is shown. First the reconstruction to the original system size. After this for the first $\mathcal{SR}$ step.



(a) Reconstructing to the original system size



(b) 1st $\mathcal{SR}$ step

Fig. 43: Results for the expectation value of the two point spin correlation function at distance $|r| = 3$ of the reconstructions with the CNN. Additionally the first $\mathcal{SR}$ from $L = 8$ up to $L = 16$. The results are shown for different proportionality factors p_f.

3.3.4 Discussion

The section for the 3D case covers the solution for the numeric temperature transformation on 12 data points. Here it is shown for multiple neural networks to be trained on a set of $4 \times 4 \times 4$ Ising spins configuration to reconstruct its original counterpart of size $8 \times 8 \times 8$. After training, the results are shown for the reconstruction as well as the first $\mathcal{SR}$ procedure. A total of seven CNN setups are constructed. Every CNN consists of 3 layers with each kernel of size $3 \times 3 \times 3$. They are all however trained with a different proportionality factor. A larger proportionality factor gives a greater influence to the regularization term presented in section 2.5.5. The regularization term specifically trains the network to minimize the energy difference between the prediction and the original spin configuration. Generally resulting in a better reconstruction of the energy.

The critical temperature is calculated to be 4.54389. This is slightly shifted to a larger temperature than the critical temperature at infinite system size. The temperature transformation is shown for the first $\mathcal{SR}$ step of the 12 data points. For any $\mathcal{SR}$ step, the new temperature is transformed towards the direction of the critical temperature.

The only difference of the seven different neural networks is in their training. The 3D turns out to be much more sensitive to the proportionality factor than the 2D case and therefore the focus in the presentation of the results is to showcase this dependency, rather than different neural network layer setups. It is visible that a larger proportionality factor results in a better learning of the energy and a smaller one results in a better learning of the magnetization and two point spin correlation function at distance 3. For too large or too small proportionality factors, the CNN becomes very accurate in reconstructing the corresponding observables but incapable of learning the opposite observables. A factor of 10^{-9} seems to be the best tested compromise for all observables. However this compromise is systematical for all data points above the MC results. For data points above the critical temperature, the error of all observables is also above 10%.

Prior to this presentation, the alternative blocking method presented in 2.5.7 had been also tested. However since it did not seem to have any effect on the results, those results are not presented.

Summed up, for the 3D model the $\mathcal{SR}$ method gives worse results and is much more sensitive to the proportionality factor than for the 2D case. It can be questioned whether the reason for this could be the training on a relatively small initial system length. In the earlier 2D section 3.2.7, this seemed to be the case. However here the model is trained in 3D and therefore still adds up to a larger amount of total spins, even for smaller system lengths. Another reason might be the choice of a relatively small neural network. Initializing with more neurons for each of the layers could result in a better training results. But the training time also increases much faster for larger networks, which is especially the case for larger dimensions. But after all the 3D case Ising model is generally known to be more difficult for getting good results with various methods and the implementation might also just hit a limitation in its performance capabilities.

4 Conclusion

Neural networks are very popular in the current times. They extend a vast range of utility and manage to solve otherwise unknown problems. This work shows how neural networks can be used to super-resolve the system size of an Ising model for up to 3D and also gives a some insights into the field of research for neural networks itself.

Part of this work is reproducing similar results of the inspirational source by Efthymiou *et al.* [4]. It extends this work by showing the theory of all the intersecting fields of research, giving additional insights to the implementation, as well as an analysis of different CNN setups. Moreover this work implements the $\mathcal{SR}$ procedure for the 3D Ising model case and analyses the problem of finding a good proportionality factor of the regularization term. In the 2D case, the method is tested with three different CNN setups for two different initial system sizes. To give more insights into the functionality of the network setups, the unpolished results are shown for different learning iterations. To push the boundaries of this methodology, one of the system sizes is chosen to be very small. Additionally for the 1D case, the method is compared to an alternative HB reconstruction method and the results for a total of 5 $\mathcal{SR}$ steps are showcased.

A working implementation of the method is presented, although a few problems remain. The accuracy of the method does not prove to be yet on par with the traditional MC simulation methods. The 1D case shows a reconstruction on 100 data points mostly with an error of under 1%, and just a few points exceeding this. For the 2D case the error is also below 1% for cold temperatures and the first $\mathcal{SR}$ step. However for high temperatures the errors are much larger and even for a good reconstruction at around 4%. For the 3D case, the errors are more difficult to describe and the reconstruction is strongly dependent on the choice of the proportionality factor.

This method is introduced to be a solution for the critical slowing down. On the plus side, the principle of the shown CNN methodology is to create much less correlated data from smaller system sizes. This allows to get good results even for small statistics because a vaster variety of spin configurations are simulated. However its accuracy also suffers the most for the critical temperature.

Neural networks require a lot of time for training. In addition the here constructed neural networks require a lot of repetitions for its training since it seems to get stuck in local minimum due to a unlucky initialization of the weights and biases. Furthermore the CNN in this use case needs to be trained for every temperature point multiplied by the amount desired $\mathcal{SR}$ steps. To manage to run the computer simulations in a reasonable time frame, every single data point was calculated parallel on a computer cluster for multiple times. It is however imaginable to restrict the chosen data points in a smart way that it is possible to save a couple of neural network training iterations. For saving time it is also worth experimenting with a quicker early stopping condition.

When taking a look at the application of neural networks in super-resolving the graphics quality with DLSS, the network is trained for a universal case and to capture the mechanics of much more expensive rendering techniques. This is where the network really shines. In the application of the simple Ising model, it is the opposite case. Here a trained network is only useful in a very particular temperature pair transformation and captures the physics of the otherwise rather simple Ising model. The question of the usability of such neural networks is therefore answered especially for larger scale sizes, if the use case is more generalizable and for more difficult or otherwise hard to calculate problems.

5 Appendix

This section covers only some of the code used in this thesis. It is focused to show the 2D case and also the part of the code used for executing the $\mathcal{SR}$ procedure. The code for the numerical solutions of the temperature transformation is not shown, as well as the code for plotting the results. Also the Metropolis MC code for creating the results at all later super-resolved system sizes is not shown. Moreover no special case investigations are shown, like for example the HB algorithm.

Primarily the *Python* code is shown which is used for training the CNN and storing the weights and biases. Afterwards the code for the $\mathcal{SR}$ procedure is presented which is also written in *Python*. It loads the trained weights and biases of the previous code. The library used for the neural network method is *TensorFlow*. At last the *C++* implementation of creating the original and decimated spin configurations is shown. The code is not presented in the order of execution.

5.1 Python Code for the 2D CNN Training

In this section the *Python* code used in this thesis is presented for making a CNN learn the weights and biases. Prior to running this code it is required to run a simulation to create a numerical solution for the temperature transformation and a simulation to generate uncorrelated original and decimated spin configurations. The latter is documented in a later section 5.3.

This code trains and saves the weights and biases by reconstructing the original spin configuration from a decimated spin configuration. The weights and biases are used in the following section for the actual super-resolution procedure. It does only train on one temperature point and one super-resolution step. This information is given to the code with a parser. However the code trains multiple CNNs with different initializations according to the same conditions from the parser. Afterwards it saves the weights and biases of all reconstructions. Then it calculates the best reconstruction and saves a pointer to it in a additional file.

```
1  #code written by Jan Zimbelmann
2  #it is required to have the following files:
3  #1. a file for the numeric results of the temperature
       ↪ transformation
4  #with the name './transformT.csv'
5  #2. a set of original and decimated spin configurations
6  #   and the observables of the original spin configurations
7  #original name: './configurations/z<step>Mc<it>L<L>.csv'
8  #decimated name: './configurations/z<step>Rg<it>L<L>.csv'
9  #observables name: './configurations/z<step>McL<L>.csv'
10 #the terms in the angle brackets refere to numeric variables
11 #'step' referes to the super resolution step
12 #'it' referes to the iteration index pointing to the transformT.csv
       ↪  temperature
13 #'L' is the system length, here 16
14 #spin configurations are stored as 0 and 1, not -1 and 1
15 #this file is originally saved as './learn.py'
16 ######
```

```
17
18 #importing libraries
19 ######
20 import os
21 import os.path
22 import tensorflow as tf
23 from tensorflow.keras import datasets, layers, models
24 import numpy as np
25 from tensorflow.python.framework import ops
26 import random
27 from tensorflow.keras.callbacks import EarlyStopping
28 tf.keras.backend.set_floatx('float64')
29 import time
30 import argparse
31
32 #initializing variables
33 ######
34 verbose = 0 #chosing a verbose or silent neural network output
35
36 #parsing the super resultion step and the iteration index pointing
    ↪ in the transformT.csv
37 parser = argparse.ArgumentParser(description='Specifying␣on␣which␣
    ↪ datapoints␣to␣learn.')
38 parser.add_argument('step', type=int, nargs='?', default = 0)
39 parser.add_argument('iteration', type=int, nargs='?', default = 0)
40 step = (parser.parse_args()).step
41 it = (parser.parse_args()).iteration
42
43 L=16 #system length
44 N=L*L #number of spins
45 dist = 3 #distance for the two point spin correlation function
46 T = np.array(np.loadtxt("transformT.csv", delimiter =',')) #
    ↪ temperature points
47 data_points = len(T[0]) #amount of all temperature points
48
49 #CNN setup conditions
50 EPOCHS = 3000 #maximum epoch size, will later be stopped with early
    ↪  stopping
51 BATCH_SIZE = 1000 #mini batch size
52 best_of = 9 #how often the training is to be repeated
53 lr = 1e-3 #learning rate
54 p_f = 2e-8 #proportionality factor
55
56 padding_sizes = [5,5,5] #initialize kernel amount (array size) and
    ↪ kernel length (array entries)
57 padding_size = len(padding_sizes) #storing kernel amount as a
```

```
    ↪ variable
padding_sum = sum(padding_sizes)-padding_size #calculating the
    ↪ padding length

es_patience = 15 #early stopping patience
es_thresh = 0 #early stopping threshold

#folders
folder_cp = 'checkpoints/' #weights and biases saving folder
folder_mc = 'configurations/' #configurations folder
folder_nn = 'reconstructions/' #folder for the CNN reconstruction
    ↪ data

#paths for the saved weights and biases
checkpoint_path = [[ None for y in range(best_of) ] for x in range(
    ↪ data_points)]

for i in range(data_points):
    for j in range(best_of):
        checkpoint_path[i][j] = folder_cp + "z" + str(step) + "T"+
            ↪ str(i) +"cp" + str(j) + ".ckpt"

#printing some information on the variables
print("super-resolution counter.",step)
print("iteration counter:", it)
print("total amount of data points:", len(T[0]))
print("current T:", T[0][it])
print("padding length:",padding_sum)

#defining functions
######
#implementing the regularization term
def regularization_term(y_true, y_pred):
    #spin input is set from 0..1 to -1..+1
    y_true_new = (y_true*2)-1
    y_pred_new = (y_pred*2)-1
    #indexing for the nearest neighbor interaction of the energy
    idx = (np.array(list(range(L)))+1)%L
    #calculating the differences in the regularization term
    reg  = y_true_new * tf.gather(y_true_new, idx, axis=1)
    reg += y_true_new * tf.gather(y_true_new, idx, axis=2)
    reg -= y_pred_new * tf.gather(y_pred_new, idx, axis=1)
    reg -= y_pred_new * tf.gather(y_pred_new, idx, axis=2)
    reg = tf.reduce_sum(reg,axis=2)
    reg = tf.reduce_sum(reg,axis=1)
    #calculating the square in the regularization term
```

```
99      reg = tf.square(reg)
100     #summing over the entire mini-batch
101     reg = tf.reduce_sum(reg)
102     #returning the regularization term with a proportionality
            ↪ factor
103     return p_f * reg
104
105 #adding the regularization term to the cross-entropy loss
106 def my_custom_loss(y_true, y_pred):
107     return tf.keras.losses.BinaryCrossentropy()(y_true, y_pred) +
            ↪ regularization_term(y_true, y_pred)
108
109 #initiate the neural network model
110 def create_model(inp):
111     #initialize a sequential model
112     model = models.Sequential()
113     #setting the elu activation function for all layers other than
            ↪ the last
114     total_padding = padding_sum
115     for a in range(padding_size-1):
116         model.add(layers.Conv2D(1, padding_sizes[a], activation=
                ↪ lambda x : tf.nn.elu(x), input_shape=[inp+
                ↪ total_padding,inp+total_padding,1]))
117         total_padding -= padding_sizes[a]-1
118     #setting the sigmoid activation function for the final layer
119     model.add(layers.Conv2D(1, padding_sizes[-1], activation='
            ↪ sigmoid', input_shape=[inp+total_padding,inp+
            ↪ total_padding,1]))
120     #compile and return
121     model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate
            ↪ = lr),loss=my_custom_loss)
122     return model
123
124 #pbc padding construction
125 #a pbc cut condition required for the next function
126 def cut(number,length):
127     if(number<length):
128         return number
129     else:
130         return length
131
132 #adding the pbc padding on a set of decimated spin configurations
133 #according to the CNN setup
134 def pbc_padding(array, p_size):
135     #conditios in place for if the pbc padding is larger than
136     #the spin configurations and applying them appropriately
```

```
137     output = np.array(array)
138     #in the first dimension
139     new_size=p_size
140     [cfgs, lng, lng] = output.shape
141     column = np.empty([cfgs,lng,0])
142     for i in range(int((new_size/lng)-1e-50)+1):
143         column = np.append(column,output[:,:,:cut(new_size,lng)],
            ↪ axis=2)
144         new_size -= lng
145     output = np.append(output, column, axis=2)
146     #in the second dimension
147     new_size= p_size
148     row = np.empty([cfgs,0,lng+new_size])
149     for i in range(int((new_size/lng)-1e-50)+1):
150         row = np.append(row,output[:,:cut(new_size,lng),:],axis=1)
151         new_size -= lng
152     output = np.append(output, row, axis=1)
153     return output
154 
155 #calculating observables
156 def getM(array): #absolute magnetization
157     m = 0
158     for i in array:
159         for j in i:
160             m += (j*2)-1
161     return abs(m)
162 
163 def getE(array): #energy
164     e = 0
165     size = len(array)
166     for i in range(size):
167         for j in range(size):
168             e += ((array[i][j]*2)-1) * ((array[(i+1)%size][j]*2)-1)
169             e += ((array[i][j]*2)-1) * ((array[i][(j+1)%size]*2)-1)
170     return -e
171 
172 def getG(array, dist): #two point spin correlation function
173     g = 0
174     amount = 0
175     size = len(array)
176     for i in range(size):
177         for j in range(size):
178             g += (array[i][j]*2-1)*(array[(i+dist)%size][j]*2-1)
179             g += (array[i][j]*2-1)*(array[i][(j+dist)%size]*2-1)
180     return g/(2*size*size)
181 
```

```
182 #loading a csv file and saving the file at a specific condition of
    ↪ this file
183 #if condition is not fulfilled, create a new data point with this
    ↪ conditions
184 def savingData(array, condition, name):
185     #check if file exists
186     #if yes, load file as array
187     #otherwise create new array stored later as file
188     if(os.path.isfile(name)):
189         old = np.loadtxt(name, delimiter =',')
190         if(len(old.shape)==1):
191             old = old.reshape(1,len(array))
192         try: # replace the data point with condition if it already
            ↪ exists
193             index = np.where(old[:,0]==condition)[0][0]
194             print(index)
195             old = np.delete(old,index,0)
196         except:
197             None
198         new = np.append(old, [array],axis=0)
199         new = new[new[:,0].argsort()]
200     else:
201         print('write')
202         new = [array]
203     #replace the old file with the new, resulting in only 1 line
        ↪ change
204     np.savetxt(name, new, delimiter = ',')
205     time.sleep(random.random()*3)
206     np.savetxt(name, new, delimiter = ',')
207
208 #diversify data set by multiplying spin configurations randomly
    ↪ with -1
209 def variation(array, nums):
210     new_array = []
211     for i, s in enumerate(array):
212         if(nums[i]==0):
213             new_array.append(1-s)
214         else:
215             new_array.append(s)
216     return np.array(new_array)
217
218 #load observables and original/decimated spin configurations + data
    ↪  preperation
219 ######
220 #original spin configurations
221 mcS, train_mcS, test_mcS = [],[],[] #load monte carlo
```

```
    ↪ configurations
mcName = 'z'+str(step)+'Mc'+str(it)+'L'+str(L)+'.csv'
mcSpin = np.array(np.loadtxt(folder_mc + mcName, delimiter =','))

#store some data poin size variables
train_size = int(len(mcSpin)/2)
test_size = int(len(mcSpin)-train_size)
size = int(len(mcSpin))

#proper 2D reshaping
mcSpin = mcSpin.reshape(size,L,L)
mcS = mcSpin
#split into training and testing/validation dataset
train_mcS = mcS[:train_size]
test_mcS = mcS[train_size:]

#decimated spin configurations
rgS, train_rgS, test_rgS = [],[],[] #load decimated MC
    ↪ configurations
rgName = 'z'+str(step)+'Rg'+str(it)+'L'+str(L)+'.csv'
rgSpin = np.array(np.loadtxt(folder_mc + rgName, delimiter =','))
#propper 2D shaping
rgSpin = rgSpin.reshape(size,L,L)
rgS = rgSpin
#appling the pbc padding
rgS = pbc_padding(rgS,padding_sum)
#split into training and testing/validation dataset
train_rgS = rgS[:train_size]
test_rgS = rgS[test_size:]

#create array for configurations for testing the reconstruction
#is only to be validated on the test part of the data set
test_mdS = np.empty([best_of,test_size,L,L])

#loading calculated observables of the original spin configurations
name_mc = 'z'+str(step)+'McL'+str(L)+'.csv'
MC = np.loadtxt(folder_mc + name_mc, delimiter =',')
mcK = MC[:,0]
mcM = MC[:,1]
mcE = MC[:,2]
mcG = MC[:,3]

#create and train the CNN model
######
#best_of amounts of models are to be trained
model_train = [ 0 for y in range(best_of) ]
```

```
266 mdE = np.empty([best_of])
267 mdM = np.empty([best_of])
268 #difference of the observables towards the original observables
269 model_dif = np.empty([best_of])
270
271 #training
272 tf.get_logger().setLevel('INFO')
273
274 #input data, the decimated spin configuration with padding
275 initial = np.array(train_rgS)
276 test = np.array(test_rgS)
277 #true data, the original spin configurations
278 target = np.array(train_mcS)
279
280 #diversifying the spin configurations
281 nums = np.random.choice([0, 1], size=size, p=[.5, .5])
282 initial = variation(initial,nums)
283 target = variation(target,nums)
284 nums = np.random.choice([0, 1], size=size, p=[.5, .5])
285 test = variation(test,nums)
286
287 #iterate the training procedure for a best_of amount
288 for j in range(best_of):
289     now = time.time() #start timer
290     model_train[j] = create_model(L)
291     cb = tf.keras.callbacks.ModelCheckpoint(filepath=
        ↪ checkpoint_path[it][j],save_weights_only=True,verbose=0)
        ↪ #condition for saving the weights & biases
292     es = EarlyStopping(monitor='loss', mode='min',verbose = 1,
        ↪ patience = es_patience, min_delta = es_thresh) #condition
        ↪  for early stopping
293     #start training
294     model_train[j].fit(initial.reshape(train_size,L+padding_sum,L+
        ↪ padding_sum,1),target.reshape(train_size,L,L,1),
        ↪ batch_size = BATCH_SIZE, epochs=EPOCHS, verbose=verbose,
        ↪ callbacks=[cb,es])
295     #stop timer & print
296     later = time.time()
297     print(later - now)
298     #testing the trained network on the validation data
299     output = model_train[j](test.reshape(test_size,L+padding_sum,L+
        ↪ padding_sum,1)).numpy().reshape(test_size,L,L)
300     #setting the output probability condition to a binary value
301     for a, spin in enumerate(output):
302         for b in range(L):
303             for c in range(L):
```

```
304                 r = random.uniform(0,1) #random number
305                 condition = spin[b][c] #condition for spin to point
                        ↪  in either direction
306                 if(r>condition):
307                     test_mdS[j][a][b][c] = 0
308                 else:
309                     test_mdS[j][a][b][c] = 1
310     #calculate observable of the testing/validation data
311     #later used to decide which model is the best according to
            ↪ best_of
312     partM = 0
313     partE = 0
314     for a in test_mdS[j]:
315         partM += getM(a)
316         partE += getE(a)
317     partM /= test_size
318     partE /= test_size
319     print(step, it, j, partM, partE)
320     print(step, it, j, mcM[it], mcE[it])
321     mdM[j]= partM
322     mdE[j] = partE
323     model_dif[j] = abs(mcE[it] - mdE[j]) + abs(mcM[it] - mdM[j])
324
325 #identifying the best choice of all the stored weights and biases
326 best_choice = np.argmin(model_dif)
327 print('best␣choice␣is␣%s␣\n'%(best_choice))
328 #creating or appending the best choice
329 name = folder_cp+'z'+str(step)+'best_choice.csv'
330 savingData([int(str(it)),best_choice], int(str(it)), name )
```

5.2 Python Code for the CNN Super-Resolution

In this section the *Python* code which was used in this thesis is presented for the super-resolution of a set of 2D spin configurations to larger system size. Prior to running this code it is required to run a simulation to create a numerical solution for the temperature transformation and a simulation to generate uncorrelated spin configurations. The latter is documented in the later presented section 5.3. Moreover it is required to run the previous *Python* code from section 5.1 to train the weights and biases for a CNN.

This code here super-resolves the original spin configurations to a larger system size according to the previously trained weights and biases and calculates their observables. It does only resolve the spin configurations for one temperature point. The targeted temperature point can be defined before running the code with a parser. However it does calculate the results for all intended $\mathcal{SR}$ steps. It is mandatory to have the CNNs trained on all temperature pairs before running this code.

```
#code written by Jan Zimbelmann
#it is required to have the following files:
#1. a file for the numeric results of the temperature
    ↪ transformation
#with the name './transformT.csv'
#2. a set of original and decimated spin configurations
#original name: './configurations/z<step>Mc<it>L<L>.csv'
#the terms in the angle brackets refere to numeric variables
#'step' referes to the super resolution step
#'it' referes to the iteration index pointing to the transformT.csv
    ↪  temperature
#'L' is the system length, here 16
#spin configurations are stored as 0 and 1, not -1 and 1
#3. a set of stored weights and biases
#checkpoint path: './checkpoints/z<step>T<it>cp<num>.ckpt'
#'num' is the repitition index for running multiple CNNs
#additionally a file is required pointing at the best
#'num' checkpoint data file
#this file is originally saved as './recon.py'
######

#importing libraries
######
import os
import os.path
import tensorflow as tf
from tensorflow.keras import datasets, layers, models
import numpy as np
from tensorflow.python.framework import ops
import random
from tensorflow.keras.callbacks import EarlyStopping
tf.keras.backend.set_floatx('float64')
```

```
31 import time
32 import argparse
33
34 #initializing variables
35 ######
36 #parsing the iteration index pointing in the transformT.csv
37 parser = argparse.ArgumentParser(description='Specifying␣which␣data
   ↪ ␣point␣to␣super␣resolve.')
38 parser.add_argument('iteration', type=int, nargs='?', default = 0)
39 it = (parser.parse_args()).iteration
40
41 repeat_output = 100000 #the amount of statistics used for
   ↪ calculating the observable
42 L=16 #system length
43 N=L*L #number of spins
44 dist = 3 #distance for two point spin correlation function
45 T = np.array(np.loadtxt("transformT.csv", delimiter =',')) #
   ↪ temperature points
46 data_points = len(T[0]) #amount of all temperature points
47
48 steps=2 #how many super resolution steps are to be calculated
49 best_of = 9 #how often the training had been repeated in the learn.
   ↪ py file
50 #padding configurations
51
52 #the following variables are not actively used but are kept for
   ↪ recycling
53 #the CNN models from the previous learn.py code
54 lr = 1e-3 #learning rate
55 loss_factor=2e-8
56
57 #CNN setup conditions
58 padding_sizes = [5,5,5]
59 padding_size = len(padding_sizes)
60 padding_sum = sum(padding_sizes)-padding_size
61 print(padding_sum)
62
63 #folders
64 folder_cp = 'checkpoints/' #weights and biases saving folder
65 folder_mc = 'configurations/' #configurations folder
66 folder_nn = 'reconstructions/' #folder for the CNN reconstruction
   ↪ data
67 os.makedirs(folder_nn[:-1], exist_ok=True)
68 folder_hi = 'nn_histogram/' #solutions for higher steps
69 os.makedirs(folder_hi[:-1], exist_ok=True)
70
```

```
71 #path for the saved weights and biases
72 checkpoint_path = [[ [None for z  in range(best_of)] for y in range
       ↪ (data_points) ] for x in range(steps+1)]
73 for step in range(steps+1):
74     for i in range(data_points):
75         for j in range(best_of):
76             checkpoint_path[step][i][j] = folder_cp + "z" + str(
                   ↪ step) + "T"+ str(i) +"cp" + str(j) + ".ckpt"
77
78 #printing some information on the variables
79 print("the␣amount␣of␣different␣spin␣configurations␣super␣resolved␣
       ↪ is:", repeat_output)
80 print("best_of␣variable␣is␣set␣to:", best_of)
81
82 #defining functions, no additional functions to 'learn.py' is used
83 ######
84 #implementing the regularization term
85 def regularization_term(y_true, y_pred):
86     #spin input is set from 0..1 to -1..+1
87     y_true_new = (y_true*2)-1
88     y_pred_new = (y_pred*2)-1
89     #indexing for the nearest neighbor interaction of the energy
90     idx = (np.array(list(range(L)))+1)%L
91     #calculating the differences in the regularization term
92     reg  = y_true_new * tf.gather(y_true_new, idx, axis=1)
93     reg += y_true_new * tf.gather(y_true_new, idx, axis=2)
94     reg -= y_pred_new * tf.gather(y_pred_new, idx, axis=1)
95     reg -= y_pred_new * tf.gather(y_pred_new, idx, axis=2)
96     reg = tf.reduce_sum(reg,axis=2)
97     reg = tf.reduce_sum(reg,axis=1)
98     #calculating the square in the regularization term
99     reg = tf.square(reg)
100    #summing over the entire mini-batch
101    reg = tf.reduce_sum(reg)
102    #returning the regularization term with a proportionality
           ↪ factor
103    return p_f * reg
104
105 #adding the regularization term to the cross-entropy loss
106 def my_custom_loss(y_true, y_pred):
107    return tf.keras.losses.BinaryCrossentropy()(y_true, y_pred) +
           ↪ regularization_term(y_true, y_pred)
108
109 #initiate the neural network model
110 def create_model(inp):
111    #initialize a sequential model
```

```
112     model = models.Sequential()
113     #setting the elu activation function for all layers other than
            ↪ the last
114     total_padding = padding_sum
115     for a in range(padding_size-1):
116         model.add(layers.Conv2D(1, padding_sizes[a], activation=
                ↪ lambda x : tf.nn.elu(x), input_shape=[inp+
                ↪ total_padding,inp+total_padding,1]))
117         total_padding -= padding_sizes[a]-1
118     #setting the sigmoid activation function for the final layer
119     model.add(layers.Conv2D(1, padding_sizes[-1], activation='
            ↪ sigmoid', input_shape=[inp+total_padding,inp+
            ↪ total_padding,1]))
120     #compile and return
121     model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate
            ↪ = lr),loss=my_custom_loss)
122     return model
123
124 #pbc padding construction
125 #a pbc cut condition required for the next function
126 def cut(number,length):
127     if(number<length):
128         return number
129     else:
130         return length
131
132 def pbc_padding(array, p_size):
133     #conditios in place for if the pbc padding is larger than
134     #the spin configurations and applying them appropriately
135     output = np.array(array)
136     #in the first dimension
137     new_size=p_size
138     [cfgs, lng, lng] = output.shape
139     column = np.empty([cfgs,lng,0])
140     for i in range(int((new_size/lng)-1e-50)+1):
141         column = np.append(column,output[:,:,:cut(new_size,lng)],
                ↪ axis=2)
142         new_size -= lng
143     output = np.append(output, column, axis=2)
144     #in the second dimension
145     new_size= p_size
146     row = np.empty([cfgs,0,lng+new_size])
147     for i in range(int((new_size/lng)-1e-50)+1):
148         row = np.append(row,output[:,:cut(new_size,lng),:],axis=1)
149         new_size -= lng
150     output = np.append(output, row, axis=1)
```

```
151     return output
152 
153 #calculating observables
154 def getM(array): #absolute magnetization
155     m = 0
156     for i in array:
157         for j in i:
158             m += (j*2)-1
159     return abs(m)
160 
161 def getE(array): #energy
162     e = 0
163     size = len(array)
164     for i in range(size):
165         for j in range(size):
166             e += ((array[i][j]*2)-1) * ((array[(i+1)%size][j]*2)-1)
167             e += ((array[i][j]*2)-1) * ((array[i][(j+1)%size]*2)-1)
168     return -e
169 
170 def getG(array, dist): #two point spin correlation function
171     g = 0
172     amount = 0
173     size = len(array)
174     for i in range(size):
175         for j in range(size):
176             g += (array[i][j]*2-1)*(array[(i+dist)%size][j]*2-1)
177             g += (array[i][j]*2-1)*(array[i][(j+dist)%size]*2-1)
178     return g/(2*size*size)
179 
180 #loading a csv file and saving the file at a specific condition of
    ↪ this file
181 #if condition is not fulfilled, create a new data point with this
    ↪ conditions
182 def savingData(array, condition, name):
183     #check if file exists
184     #if yes, load file as array
185     #otherwise create new array stored later as file
186     if(os.path.isfile(name)):
187         old = np.loadtxt(name, delimiter =',')
188         if(len(old.shape)==1):
189             old = old.reshape(1,len(array))
190         try: # replace the data point with condition if it already
            ↪ exists
191             index = np.where(old[:,0]==condition)[0][0]
192             print(index)
193             old = np.delete(old,index,0)
```

```
194         except:
195             None
196         new = np.append(old, [array],axis=0)
197         new = new[new[:,0].argsort()]
198     else:
199         print('write')
200         new = [array]
201     #replace the old file with the new, resulting in only 1 line
        ↪ change
202     np.savetxt(name, new, delimiter = ',')
203     time.sleep(random.random()*3)
204     np.savetxt(name, new, delimiter = ',')
205
206 #load observables and original/decimated spin configurations + data
    ↪  preperation
207 ######
208 #original spin configurations
209 mcName = 'z'+str(0)+'Mc'+str(it)+'L'+str(L)+'.csv'
210 mcS = np.array(np.loadtxt(folder_mc + mcName, delimiter =','))
211 size = len(mcS)
212 mcS = mcS.reshape(size, L, L)
213
214 #decimated spin configurations
215 rgName = 'z'+str(0)+'Rg'+str(it)+'L'+str(L)+'.csv'
216 rgS = np.array(np.loadtxt(folder_mc + rgName, delimiter =','))
217 rgS = rgS.reshape(size,L,L)
218
219 #reconstructed/super-resolved spin configurations
220 nnS = [None for i in range(steps+1)]
221
222 #preparing arrays to save the observables for every super
    ↪ resolution
223 nnM = np.zeros(steps+1) #absolute magnetization array
224 nnE = np.zeros(steps+1) #energy array
225 nnG = np.zeros(steps+1) #two point spin correlation function array
226 histM = [[] for i in range(steps+1)] #magnetization histograms
227 histE = [[] for i in range(steps+1)] #energy histograms
228
229 #preparing arrays for the CNN super resolution procedure
230 best_choices = np.zeros(steps+1) #best choice of weights and biases
231 forward_model = [None for i in range(steps+1)] #loading the CNN
    ↪ models
232 #the following output array is created in this way for
    ↪ visualization reasons
233 output = [0 for i in range(steps+1)] #array of outputs
234
```

```
235 #further step dependent initializations
236 for step in range(0,steps+1):
237     newL = L * (2 ** step) #system size after super-resolutions
238     nnS[step] = np.zeros([1,newL,newL]) #store a empty spin
        ↪ configuration
239     name_bc = folder_cp+'z'+str(step)+'best_choice.csv' #name of
        ↪ the best choice file
240     best_choice_load = np.loadtxt(name_bc, delimiter =',') #loading
        ↪  the best choice
241     best_choices[step] = int(best_choice_load[np.where(
        ↪ best_choice_load[:,0]==int(it))[0][0]][1])
242     forward_model[step] = create_model(newL) #create a CNN model
243     load_path = checkpoint_path[step][it][int(best_choices[step])]
        ↪ #path to weights and biases
244     forward_model[step].load_weights(load_path).expect_partial() #
        ↪ load the weights and biases
245
246     #initialize the possible observable values for the histograms
247     for a in range(int((newL*newL)/2)+1):
248         histM[step].append([a*2,0])
249     for a in range(int(newL*newL)+1):
250         histE[step].append([(a*4)-(newL*newL*2),0])
251
252 #start the reconstruction/super-resolution procedure
253 start = time.time() #start a timer
254
255 #iteration over every step, including the reconstruction to the
    ↪ original configuration
256 for dp_ in range(repeat_output):
257     dp = dp_%size #repeat data point for all original spin
        ↪ configurations
258     for step in range(0,steps+1):
259         newL = L * (2 ** step) #system size after super-resolutions
260
261         #load the input spin configuration for the reconstruction/
            ↪ super resolution
262         if(step==0): #for the reconstruction
263             inputS = np.array([rgS[dp]])
264         elif(step==1): #for the first super-resolution step
265             inputS = np.array([mcS[dp]])
266             inputS = inputS.repeat(2,axis=1).repeat(2,axis=2)
267         else: #for any super-resolution steps > 1
268             inputS = np.array([nnS[step-1][0]]) #nnS is always size
                ↪  1
269             inputS = inputS.repeat(2,axis=1).repeat(2,axis=2)
270
```

```
271         #applying the pbc padding
272         inputS = pbc_padding(inputS,padding_sum)
273 
274         #diversify the spin configurations
275         if(random.random()>0.5):
276             inputS = -(inputS-1)
277 
278         inputSize = 1 #every spin configuration is super resolved 1
                ↪  by 1
279         output[step] = forward_model[step](inputS.reshape(inputSize
                ↪ ,newL+padding_sum,newL+padding_sum,1)).numpy().
                ↪ reshape(inputSize,newL,newL) #output of the CNN
280 
281         #setting the output probability condition to a binary value
282         for a, spin in enumerate(output[step]):
283             for b in range(newL):
284                 for c in range(newL):
285                     r = random.uniform(0,1) #random number
286                     condition = spin[b][c] #condition for spin to
                            ↪ point in either direction
287                     if(r>condition):
288                         nnS[step][a][b][c] = 0
289                     else:
290                         nnS[step][a][b][c] = 1
291 
292         #calculate observables
293         for a in nnS[step]:
294             E = getE(a)
295             M = getM(a)
296             G = getG(a,dist)
297             nnE[step] += E
298             nnM[step] += M
299             nnG[step] += G
300             #calculate histograms
301             intM = M/2 #index of magnetization
302             intE = int(E+(newL*newL*2))/4 #index of energy
303             histM[step][int(intM)][1] += 1 #set counter +1 for
                    ↪ magetization
304             histE[step][int(intE)][1] += 1 #set counter +1 for
                    ↪ energy
305 
306 #expectation of the observables
307 nnE = nnE / repeat_output
308 nnM = nnM / repeat_output
309 nnG = nnG / repeat_output
310 
```

```
311 #storing observables in .csv files
312 ######
313 for step in range(steps+1):
314     #saving histograms
315     np.savetxt(folder_hi+"z"+str(a)+"nnHistE"+str(it)+"L"+str(L)+".
        ↪ csv",histE[a],fmt='%g',delimiter=',')
316     np.savetxt(folder_hi+"z"+str(a)+"nnHistM"+str(it)+"L"+str(L)+".
        ↪ csv",histM[a],fmt='%g',delimiter=',')
317     #saving observables
318     nn_sol = [it,nnM[a],nnE[a],nnG[a]] #constructing the line of
        ↪ the histogram
319     savingData(nn_sol,int(it),folder_nn+"z"+str(a)+"NnL"+str(L)+".
        ↪ csv")
320
321     #printing the previously saved data
322     print("M:", nnM[a])
323     print("E:", nnE[a])
324     print("G:", nnG[a])
325
326 #printing the total run time
327 end = time.time() #end the timer
328 print(end - start)
```

5.3 C++ Code for simulating Spin Configurations

In this section the *C++* code used for this thesis is presented for creating a set of original and decimated Ising spin configurations. Also the observables for the original spin configurations are calculated and stored. Prior to running this code it is required to create a numerical solution for the temperature transformation. The original and decimated spin configurations which are generated with this code are used in section 5.1 for the CNN learning procedure. Only the original configurations are used in section 5.2 for the super-resolution procedure.

The original spin configurations are simulated at a system length L_0 and not for any larger system sizes. This simulation creates the original and decimated spin configurations for all the temperature points from the numerical simulation. Therefore it needs to be run only once. It is noted that the created and stored spin configurations are all uncorrelated.

```
1  //code written by Jan Zimbelmann
2  //it is required to have the following file:
3  //a file for the numeric results of the temperature transformation
4  //with the name './transformT.csv'
5  //the spin configurations are stored as 0 and 1
6  //this files is originally saved as './configurations_ising.cpp'
7  //////
8
9  //libraries
10 //////
11 #include <iostream>
12 #include <math.h>
13 #include <vector>
14 #include <fstream>
15 #include <ctime>
16 #include <sys/stat.h>
17 #include <sstream>
18
19 using namespace std;
20
21 //initializing variables
22 //////
23
24 const int L0 = 16; //1D system length
25 const int steps = 2; //enlarging steps
26 double J = 1; //coupling constant
27
28 int iteration; //number of total spin flip iterations
29 int start; //starting iteration to count the observable
30 int outSize = 20000; //amount of the configurations stored
31 int mult = 30; //every N*mult, one iteration is stored
32 int counter; //security counter, later used
33
```

```
34 //information on the spin lattice
35 int x; // random position in first dimension
36 int y; // random position in second dimension
37 double r; // uniform random number
38 double condition;
39 int blockSpin; //for the majority rule decimation
40
41 //observables
42 int beforeE = 0; //local energy before spinflip
43 int afterE = 0; //local energy after spinflip
44 int difE = 0;//energy differene of previous two energies
45 int E = 0; //total energy
46 int M = 0; //total magnetization
47 double G = 0; //total two point spin correlation function
48 double beforeG = 0; //local G before spinflip
49 double savedE = 0; //summed energy over all iterations
50 double savedM = 0; //summed magnetization "
51 double savedG = 0; //summed G "
52 int distG = 3; //distance of G
53 int data_points; //size of the loaded temperature array
54
55
56 //functions
57 //////
58
59 int add1(int target, int L){ //custom +1 ; addition function
60   if(target<(L-1)){return target + 1;}
61   else{return 0;}
62 }
63 int addX(int target, int L, int distance){ //custom +X ;addition
    ↪ function
64   int newTarget = target;
65   for(int i = 0; i<distance; ++i){
66     newTarget = add1(newTarget,L);
67   }
68   return newTarget;
69 }
70 int sub1(int target, int L){ //custom -1 ; subtraction function
71   if(target>0){return target - 1;}
72   else{return L-1;}
73 }
74 int subX(int target, int L, int distance){ //custom -X ;
    ↪ subtraction function
75   int newTarget = target;
76   for(int i = 0; i<distance; ++i){
77     newTarget = sub1(newTarget,L);
```

```
78    }
79    return newTarget;
80  }
81
82  template <typename spinT> //multidimensional template
83  void printDisplay(spinT& spin, int L){ //printing a colored spin
      ↪ configuration
84    for(int i = 0; i < L; ++i){
85      for(int j = 0; j < L; ++j){
86        int s = spin[i][j];
87        if(s==1){
88          //color green
89          cout << "\033[1;32m"<< "␣" << s << "\033[0m";
90        }
91        else{
92          //color red
93          cout << "\33[;31m" << s << "\033[0m";
94        }
95      }
96      cout << endl;
97    }
98    cout << endl;
99  }
100
101 template <typename totET> //multidimensional template
102 //functions for calculating the observables
103 double totE(totET& spin, int L){ //calculate the total energy
104   double energ = 0;
105   for(int i = 0; i<L; ++i){
106     for(int j = 0; j<L; ++j){
107       energ += spin[i][j]*spin[add1(i,L)][j];
108       energ += spin[i][j]*spin[i][add1(j,L)];
109     }
110   }
111   energ *= -J;
112   return energ;
113 }
114
115 template <typename locET> //multidimensional template
116 int localE(locET& spin, int x, int y, int L){ //calculated the
      ↪ local energy
117   int dif = 0;
118   dif += spin[y][x]*spin[y][add1(x,L)];
119   dif += spin[y][x]*spin[y][sub1(x,L)];
120   dif += spin[y][x]*spin[add1(y,L)][x];
121   dif += spin[y][x]*spin[sub1(y,L)][x];
```

```
122   dif = -dif * J;
123   return dif;
124 }
125 
126 template <typename totMT> //multidimensional template
127 int totM(totMT& spin, int L){ //calculate the total magnetization
128   int m = 0;
129   for(int i = 0; i<L; ++i){
130     for(int j = 0; j<L; ++j){
131       m += spin[i][j];
132     }
133   }
134   return m;
135 }
136 
137 template <typename locGT> //multidimensional template
138 double localG(locGT& spin, int x, int y, int L, int distance){ //
    ↪ calculate the local G
139   double dif = 0;
140   dif += spin[y][x]*spin[y][addX(x,L,distance)];
141   dif += spin[y][x]*spin[y][subX(x,L,distance)];
142   dif += spin[y][x]*spin[addX(y,L,distance)][x];
143   dif += spin[y][x]*spin[subX(y,L,distance)][x];
144   return dif/(2*L*L);
145 }
146 
147 template <typename totGT> //multidimensional template
148 double totG(totGT& spin, int L, int distance){ //calculate the
    ↪ total G
149   double g = 0;
150   for(int i = 0; i<L; ++i){
151     for(int j = 0; j<L; ++j){
152       g+= spin[j][i] * spin[j][addX(i,L,distance)];
153       g+= spin[j][i] * spin[addX(j,L,distance)][i];
154     }
155   }
156   return g/(2*L*L);
157 }
158 
159 //start the Metropolis MC algorithm for creating the Ising spin
    ↪ configurations
160 //////
161 
162 int main(){
163   //begin timer
164   clock_t begin = clock();
```

```
165   srand(time(NULL));
166   //load the numeric solutions for the temperatures
167         ifstream f;
168         f.open("transformT.csv");
169         vector<vector<double>> temp;
170         string line, val;
171         while(getline(f, line)){
172     cout << "test" << endl;
173                 vector<double> v;
174                 stringstream s (line);
175                 while(getline(s, val, ','))
176                         v.push_back(stod(val));
177       cout << val << endl;
178                 temp.push_back(v);
179         }
180   data_points = temp[0].size(); //calculate the amount of initial
        ↪ temperatures
181   cout << "total amount of data points " << data_points << endl;
182   f.close();
183   //create a folder under linux
184   string folder = "configurations/"; //folder for storing the spin
        ↪ configurations
185   mkdir("configurations", S_IRWXU | S_IRWXG | S_IROTH | S_IXOTH);
186   //iterate over every super resolution step, as well as the
        ↪ initial system size
187   for(int step=0; step <= steps; step+=1){
188     cout << "super-resolution step is currently at: " << step <<
          ↪ endl;
189     //step dependent boundary conditions
190     const int L = L0; //system size is gonna only be calculated at
          ↪ the original system size
191     int N = L*L; //numer of lattice size
192     int spin[L][L]; //create a original spin lattice
193     int rgSpin[L][L]; //create the decimated spin lattice
194     int period = N * mult; //period of storing a new spin
          ↪ configuration
195     start = N * 3000; //starting the procedure after a set amount
          ↪ of iterations
196     iteration = (outSize * period)+start; //total amount of
          ↪ iterations
197     //randomize spins, plotting and calculating observables
198     //for(int i=0; i<L; ++i){for(int j=0; j<L; ++j){spin[i][j]=((
          ↪ round((float)rand()/((float)RAND_MAX)))-0.5)*2;}} //hot
          ↪ start
199     for(int i=0; i<L; ++i){for(int j=0; j<L; ++j){spin[i][j]=1;}}
          ↪ //cold start, all spins pointing at +1
```

```
200     printDisplay(spin,L); //plot the spins
201
202     //initiate observable/configuration files
203     ofstream mcConfFile; //file for the original spin configuration
204     ofstream rgConfFile; //file for the decimated spin
            ↪ configuration
205     ofstream mcFile; //file for the observables of the original
            ↪ spin configurations
206     string name = string("z") + to_string(step) + string("Mc") +
            ↪ string("L") + to_string(L) + string(".csv");
207     mcFile.open(folder+name);
208     //initiate observables
209     E = totE(spin,L); //calculate total energy
210     M = totM(spin,L); //calculate total magnetization
211     G = totG(spin,L,distG); //calculate total two point spin
            ↪ correlation function
212     //loop over all temperatures of interest
213     for(int t = 0; t<data_points; ++t){
214       double T = temp[step][t]; //set the current Temperature
215       cout << "T: " << T <<endl;
216       //initiate spin configuration files
217       string name = string("z") + to_string(step) + string("Mc") +
              ↪ to_string(t) + string("L") + to_string(L0) + string(".
              ↪ csv");
218       mcConfFile.open(folder + name); //original spin configuration
              ↪  file
219       name = string("z") + to_string(step) + string("Rg") +
              ↪ to_string(t) + string("L") + to_string(L0) + string(".
              ↪ csv");
220       rgConfFile.open(folder + name); //decimated spin
              ↪ configuration file
221       //surveilance of the configuration size with a counter
222       counter = 0;
223       //for loop over the iteration amount and and store the spin
              ↪ configurations & observables
224       for(int i = 0; i < iteration; ++i){
225         //random spin position
226         x=rand()%L; //random x value
227         y=rand()%L; //random y value
228         //calculate the local energy difference before the flip
                ↪ spin
229         beforeE = localE(spin, x, y, L);
230         beforeG = localG(spin, x, y, L, distG); //also the two
                ↪ point spin correlation function
231         spin[y][x] *= -1; //execute the spin flip
232         //calculate Energy difference/two point correlation before
```

```
   ↪ the flip spin
233         afterE = localE(spin, x, y, L);
234         difE = afterE - beforeE; // calculate the energy difference
               ↪ from before and after
235         //check spin acceptance condition
236         r = ((double) rand() / (RAND_MAX)); //random double number
               ↪ between 0 and 1
237         condition = exp(-difE * (1/T)); //acceptance/rejection rate
               ↪ of the spin flip
238         if(difE < 0||r <= condition){ //accept the spin flip
239           E += difE;
240           M += 2*spin[y][x];
241           G += localG(spin,x,y,L,distG) - beforeG;
242         }
243         else{ //reject the spin flip
244           spin[y][x] *= -1;
245         }
246         //summing observables for every iteration
247         if(i>=start){
248           savedE += E;
249           savedM += abs(M);
250           savedG += G;
251         }
252
253         //decimating the spin configurations and storing both
               ↪ configurations
254         if(i>= start && i%period ==0 && counter < outSize){//
               ↪ calculated only with after every certain period
255           counter += 1; //counter for security reasons
256           //decimation of the spin configurations
257           //now iterating over every block
258           for(int p = 0; p<L; p = p+2){
259             for(int q=0; q<L; q = q+2){
260               //majority rule of the block decimation
261               blockSpin = 0;
262               blockSpin += spin[p][q];
263               blockSpin += spin[p][q+1];
264               blockSpin += spin[p+1][q];
265               blockSpin += spin[p+1][q+1];
266               blockSpin = (blockSpin > 0) ? 1 : ((blockSpin < 0) ?
                     ↪ -1 : spin[p][q]);
267               //after the majority rule & now the simple upscaling
                     ↪ is applied
268               //resulting in a deflated spin configuration
269               rgSpin[p][q] = blockSpin;
270               rgSpin[p][q+1] = blockSpin;
```

```
271                 rgSpin[p+1][q] = blockSpin;
272                 rgSpin[p+1][q+1] = blockSpin;
273               }
274             }
275             //storing the original and deflated spin configuration
276             //the 2 dimensional array is converted into a 1
                  ↪ dimensional string of 0s and 1s
277             for(int p = 0; p<L; ++p){
278               for(int q=0; q<L; ++q)
279                 if(not(p == L-1 and q == L-1)){
280                   mcConfFile << (spin[p][q]+1)/2 << ",";
281                   rgConfFile << (rgSpin[p][q]+1)/2 << ",";
282                 }
283                 else{
284                   mcConfFile << (spin[p][q]+1)/2 << endl;
285                   rgConfFile << (rgSpin[p][q]+1)/2 << endl;
286                 }
287             }
288           }
289         }
290         //for security reasons printing out the calculated energy,
              ↪ magnetization and two point correlation function
291         //if they actually do not equate to each other, there is a
              ↪ bug in the code
292         cout <<
293         M << " = " << totM(spin, L) << " , " <<
294         E << " = " << totE(spin, L) << " , " <<
295         G << " = " << totG(spin, L, distG)
296         << endl;
297         //store the expectation values of the observables
298         savedM /= iteration-start; //magnetization
299         savedE /= iteration-start; //energy
300         savedG /= iteration-start; //two point spin correlation
              ↪ function
301         //saving observables
302         mcFile << T << "," << savedM << "," << savedE << "," <<
              ↪ savedG << endl;
303         //printing the calculated expectations of the observables
304         cout << "T : " << T << endl << "E : " << savedE << endl << "M
              ↪ : " << savedM << endl;
305         //closing spin configuration configuration files
306         mcConfFile.close();
307         rgConfFile.close();
308       }
309       //plotting the final spin configuration
310       printDisplay(spin,L);
```

```
311     //checking the counter for security reasons
312     cout << "Counter␣has␣reached:␣" << counter << ".␣It␣was␣
            ↪ expected␣to␣reach:␣" << outSize << "." << endl;
313     //closing the observables file
314     mcFile.close();
315   }
316   //end timer
317   clock_t end = clock();
318   double elapsed_secs = (double)(end - begin) / CLOCKS_PER_SEC;
319   cout << elapsed_secs << "␣seconds." << endl; //print the run time
320   return 0;
321 }
```

References

[1] Valentina Borghesani and Fabian Pedregosa. `https://jmlr.csail.mit.edu/stats.html`. *Journal of Machine Learning Research*, accessed: 2021-06-19.

[2] Wolfhard Janke. Introduction to Simulation Techniques. *Ageing and the Glass Transition, Lect. Notes Phys. 716*, 1997.

[3] Wolfhard Janke. Nonlocal Monte Carlo Algorithms for Statistical Physics Applications. *Mathematics and Computers in Simulations 47, 329 (1998); invited review lecture at the IMACS Workshop on Monte Carlo Methods , Brussels*, 1997.

[4] Stavros Efthymiou, Matthew J. S. Beach, and Roger G. Melko. Super-Resolving the Ising Model with Convolutional Neural Networks. *Physical Review B*, 99(7), Feb 2019.

[5] Yin Li, Yueying Ni, Rupert A. C. Croft, Tiziana Di Matteo, Simeon Bird, and Yu Feng. AI-assisted Superresolution Cosmological Simulations. *Proceedings of the National Academy of Sciences*, 118(19):e2022038118, May 2021.

[6] Mark E. J. Newman and Gerard T. Barkema. *Monte Carlo Methods in Statistical Physics*. Clarendon Press, Oxford, 1999.

[7] Wolfhard Janke. Monte Carlo Methods in Classical Statistical Physics. *Computational Many-Particle Physics, Wilhelm & Else Heraeus Summerschool, Greifswald, edited by H. Fehske, R. Schneider, and A. Weiße, Lect. Notes Phys. 739 (Springer, Berlin, 2008), pp. 79-140*, 2008.

[8] Silvio Salinas. *Introduction to Statistical Physics*. Springer-Verlag New York, 01 2001.

[9] Michael Nauenberg. Renormalization Group Solution of the One Dimensional Ising Model. *Journal of Mathematical Physics 16*, 1975.

[10] Subhajit Paul. Lecture notes on Computational Physics 2, Winter semester 2019-2020.

[11] Avneet Sood, Arthur Forster, Billy J. Archer, and Robert C. Little. Neutronics Calculation Advances at Los Alamos: Manhattan Project to Monte Carlo, 2021.

[12] Graham Barr and Ian Durbach. A Monte Carlo Analysis of Hypothetical MultiLine Slot Machine Play. *International Gambling Studies*, 8:265–280, 12 2008.

[13] Paul Glasserman. *Monte Carlo Methods in Financial Engineering*. Springer, New York, 2004.

[14] Haifeng Ma and Yuxi Liu. Research Based on the Monte-Carlo Method to Calculate the Definite Integral. In *2011 Third Pacific-Asia Conference on Circuits, Communications and System*, pages 1–3, 2011.

[15] Andrzej Kolinski and Jeffrey Skolnick. Monte Carlo Simulations of Protein Folding. II. Application to Protein A, ROP, and Crambin. *Proteins: Structure, Function, and Bioinformatics*, 18(4):353–366, 1994.

[16] Bernd A. Berg. *Markov Chain Monte Carlo Simulations and Their Statistical Analysis: With Web-Based Fortran Code*. World Scientific Publishing Company, 2004.

[17] Robert H. Swendsen. Monte Carlo Renormalization Group. *Journal of Statistical Physics*, 34(5):963–973, Mar 1984.

[18] Rajan Gupta. Open Problems in Monte Carlo Renormalization Group: Application to Critical Phenomena. In *31st Annual Conference on Magnetism and Magnetic Materials - MMM*, 11 1986.

[19] Raphael Goll and Peter Kopietz. Renormalization Group for the phi4 Theory with Long-Range Interaction and the Critical Exponent eta of the Ising Model. *Physical Review E*, 98(2), Aug 2018.

[20] Mitsuhiro Itakura. Monte Carlo Renormalization Group Study of the Heisenberg and the XY Antiferromagnet on the Stacked Triangular Lattice and the Chiral phi4 Model. *Journal of the Physical Society of Japan*, 72(1):74–82, Jan 2003.

[21] Bang Liu, Yan Liu, and Kai Zhou. Image Classification for Dogs and Cats, 2014.

[22] Christopher Thomas. Deep learning based Super-Resolution, without using a GAN. `https://towardsdatascience.com/deep-learning-based-super-resolution-without-using-a-gan-11c9bb5b6cd53`, 2019.

[23] Alexander Watson. Deep Learning Techniques for Super-Resolution in Video Games. *Computing Research Repository*, abs/2012.09810, 2020.

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

[25] Roman Zimbelmann. Variant calling using Neural Networks. Master's thesis, Gottfried Wilhelm Leibniz Universität Hannover, Institut für Informationsverarbeitung, 2018. accessed 2021-06-21 at `https://hut.pm/data/VariantCallingUsingNeuralNetworks.pdf`.

[26] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer Feedforward Networks are Universal Approximators. *Neural Networks*, 2(5):359 – 366, 1989.

[27] Lei Jimmy Ba and Rich Caruana. Do Deep Nets Really Need to be Deep? *Computing Research Repository*, abs/1312.6184, 2013.

[28] Marius-Constantin Popescu, Valentina E. Balas, Liliana Perescu-Popescu, and Nikos Mastorakis. Multilayer Perceptron and Neural Networks. *WSEAS Transactions on Circuits and Systems*, 8(7):579–588, July 2009.

[29] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[30] Popescu Marius, Valentina Balas, Liliana Perescu-Popescu, and Nikos Mastorakis. Multilayer Perceptron and Neural Networks. *WSEAS Transactions on Circuits and Systems*, 8, 07 2009.

[31] Lu Lu, Yeonjong Shin, Yanhui Su, and George Em Karniadakis. Dying ReLU and Initialization: Theory and Numerical Examples. *Communications in Computational Physics*, 28(5):1671–1706, Jun 2020.

[32] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs). *arXiv: Learning*, 2016.

[33] Michael A. Nielsen. Neural Networks and Deep Learning. `http://neuralnetworksanddeeplearning.com/`, 2018. accessed 2021-06-20 at `https://static.latexstudio.net/article/2018/0912/neuralnetworksanddeeplearning.pdf`.

[34] Léon Bottou, Frank E. Curtis, and Jorge Nocedal. Optimization Methods for Large-Scale Machine Learning, 2018.

[35] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization, 2014. Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015.

[36] Douglas M. Hawkins. The Problem of Overfitting. *Journal of Chemical Information and Computer Sciences*, 44(1):1–12, 2004. PMID: 14741005.

[37] Jürgen Mayer, Khaled Khairy, and Jonathon Howard. Drawing an Elephant with Four Complex Parameters. *American Journal of Physics*, 78(6):648–649, June 2010.

[38] Rich Caruana, Steve Lawrence, and Lee Giles. Overfitting in Neural Nets: Backpropagation, Conjugate Gradient, and Early Stopping. In *Proceedings of the 13th International Conference on Neural Information Processing Systems*, NIPS'00, page 381–387, Cambridge, MA, USA, 2000. MIT Press.

[39] Dorit Ron, Achi Brandt, and Robert H. Swendsen. Monte Carlo Renormalization Group Calculation for the d=3 Ising Model using a modified Transformation, 2020.

Statement of Authorship

I hereby declare that I completed this thesis on my own, that all the information which has been directly or indirectly taken from other sources has been noted as such and that I have used no other means than the ones indicated. This work has not been submitted to another examination committee.

Date: **Signature** (Jan Zimbelmann)**:**

Leipzig, June 23, 2021 .....