Classifying Phases of Matter with Machine Learning

Daniel Long
School of Physics and Astronomy
University of Nottingham

12/04/2019

Abstract

Two different deep learning architectures are trained to classify the temperature of Ising model spin configurations in one and two dimensions. These architectures, a multi-layer perceptron (MLP) and a convolutional neural network (CNN), find different approaches for inferring temperature from the configurations, and subsequently the phase of the configuration. In this report it will be demonstrated that a configurations phase can be classified without prior knowledge of order parameters, the approaches that each network arrive at will also be analysed and understood. There will also be a comparison of the utility of the two architectures for the problem of classifying phase. The main finding from this report is that encoded in the weights of both networks lies the critical temperature of the 2D Ising model. It was also found that the convolutional neural network performs with a higher classification accuracy than the multi-layer perceptron, achieving a 98.5% phase classification accuracy compared to the 97.4% phase classification accuracy reached by the multi-layer perceptron.

Contents

1 Introduction **3**

2 Background Theory **3**
2.1 The Ising Model 3
2.2 The Two-Dimensional Ising Model 4
2.3 Monte-Carlo Simulation of Ising Model 5
2.4 Multi-Layer Perceptrons (MLP) 6
2.5 Convolutional Neural Networks 8

3 Method **9**
3.1 Development of 1D Simulation 10
3.2 Development of 2D Simulation 12
3.3 Preparation of Multi-Layer Perceptron 13
3.4 Preparation of Convolutional Neural Network 14

4 Results **15**
4.1 Analysis of Multi-Layer Perceptron 16
4.2 Analysis of Convolutional Neural Network 18

5 Discussion **20**

6 Summary **21**

7 References **22**

8 Appendix **24**
8.1 1D Monte-Carlo Simulation in C 24
8.2 2D Monte-Carlo Simulation in C 28
8.3 Multi-Layer Perceptron code in python 34
8.4 Convolutional Neural Network code in python 39

1 Introduction

The application of machine learning to scientific research is not new, however, in recent years the ease and potential of application has massively increased. Since the 1980s multi-layer perceptrons, a popular machine learning algorithm, have been used to detect subtle patterns in large volumes of data from the Large Hadron Collider [1]. The subtle patterns, detected with neural networks, were used to point researchers to fruitful avenues for investigation. However, these computations required lots of time and resources, hence their use was not widespread. More recently, with advances in hardware and an increase in the availability of large datasets, scientific research has become more geared towards these forms of analysis. In many different areas, machine learning methods have been used with considerable success; for example, in bioinformatics machine learning methods have been used with considerable success in analysing gene-gene interactions and detecting patterns between genomes and related phenotypes [2,3].

In the field of condensed-matter physics there has been a recent surge in the use of machine learning algorithms to analyse systems which cannot be solved analytically. One such system is the Ising model. Various approaches have been taken, including both unsupervised and supervised algorithms, on a range of different variants of the Ising model. A popular unsupervised algorithm is principle component analysis (PCA). Work done by Hu et al. has shown that PCA on 2D spin configurations yields the critical temperature, as well as eigenvalues corresponding to arrangements, starting from the most ordered to decreasing order [4]. Hu et al. also applied another unsupervised algorithm, an autoencoder with a convolutional layer as its hidden layer. This autoencoder detected phase transitions and therefore also located the critical temperature. This approach has strong similarities to the approach undertaken in this report. However, in this report a supervised convolutional neural network (CNN) will be used, instead of an unsupervised autoencoder. A more similar approach was undertaken by A. Tanaka and A. Tomiya, in their paper they used a CNN to predict the temperature of a configuration [5]. Their paper demonstrated the effectiveness of this approach as they were able to accurately determine the critical temperature of the 2D system.

In this report there will be an introduction to the Ising model, followed by an overview of the Monte-Carlo approach to simulating the model. The report will then go on to cover the principles behind MLPs and CNNs, and finally there will be an analysis of the results obtained. The main aims of the report are to train and understand networks for detecting the phase of a spin configuration. A more general aim of this report is to demonstrate machine learning algorithms as a tool for analysing complex systems.

2 Background Theory

2.1 The Ising Model

The Ising model was developed by Ernst Ising in his 1925 paper, "Beitrag zur Theorie des Ferromagnetismus" [6], which translates to, "Contribution to the theory of ferromagnetism". Ising's PhD supervisor Wilhelm Lenz proposed the model as a means of explaining the onset of ferromagnetism [7]. While at first the model was considered too simplified to be physically relevant, it grew in popularity in the 1960s when it was shown to fit empirical results [8]. The Ising model simulates an arrangement of atoms on a lattice by labelling each atom with a state corresponding to the spin of the atom (either $+1$ or -1). The model evolves following the Hamiltonian

$$H(\sigma) = -J \sum_{\langle i,j \rangle} s_i s_j - \mu h \sum_{i=1}^{N} s_i \tag{1}$$

where σ is the specific configuration, J is the interaction energy, $<i,j>$ refers to neighbouring atoms, s_i is the spin state of the i^{th} atom, μ is the magnetic moment of the atoms on the lattice and h is the external magnetic field strength. From equation 1 it is clear that, provided the interaction energy is positive, the system's energy will be minimised when neighbouring atoms are spin aligned and when they are aligned with the external magnetic field. Atomic arrangements sample the configuration space of possible microstates following a probability distribution given by

$$P(\sigma) = \frac{e^{\frac{-H(\sigma)}{k_B T}}}{Z} \tag{2}$$

where H is the Hamiltonian of the configuration σ, T is the temperature of the lattice and Z is the partition function. The partition function is given by

$$Z = \sum_{\sigma} e^{\frac{-H(\sigma)}{k_B T}}. \tag{3}$$

The Ising model takes different forms according to the chosen interaction energy. When the interaction energy is set to be positive the system is said to be ferromagnetic, when the interaction energy is negative the system is antiferromagnetic and when the interaction energy is zero the system is noninteracting. In this report only ferromagnetic models will be considered. Another caveat will be that the external magnetic field will be set to zero, leading to a Hamiltonian with only the interaction term active.

The temperature dependence of the Ising model is non-linear. Through applying the mean field approximation to the 1D Ising model it can be shown that the expectation energy of the 1D system follows

$$\langle E \rangle = -L \tanh\left(\frac{J}{k_B T}\right) \tag{4}$$

where $\langle E \rangle$ is the expectation energy and L is the length of the model. A derivation of this result can be found in the project diary on pages 20-22 and 27. For each model there exists a critical temperature at which entropic forces dominate over energetic forces. The location of the critical temperature varies with respect to the dimensionality of the model. For the 1D model the critical temperature is defined as

$$T_c = \frac{zJ}{k_B} \tag{5}$$

where z is the number of neighbouring atoms, 2 in the 1D case. A derivation of this result can be found on pages 39-40 of the project diary.

2.2 The Two-Dimensional Ising Model

When the Ising model is extended to two dimensions the lattice becomes a torus and more complex behaviour occurs. Unlike the 1D model, the 2D model exhibits a phase transition.

This development can be understood qualitatively by noting that unlike in 1D, simply flipping the state of a single atom will not break a region of aligned spin. This difference results in the 2D model behaving as a ferromagnet, having a non-zero magnetization at absolute zero, unlike the 1D model, which behaves as a paramagnet.

Analytical solutions for the 2D model are much more complex than the 1D equivalent, and rely on complex mathematical methods. In 1941 Kramers and Wannier developed the first analytical solution to the 2D model. Their work is laid out in detail in their paper "Statistics of the Two-Dimensional Ferromagnet. Part I" [9], the bulk of their approach lies on pages 256-259. A simple description of their approach was to construct a 2D lattice by winding a strip of the 1D lattice. This approach relied on the principle that the state of an additional atom to the chain depends only on the previous atom. While this approach largely failed they were able to find some exact results, such as the critical temperature which was found from the following equation

$$\sinh\left(\frac{2J}{k_B T_c}\right) = 1 \tag{6}$$

where T_c is found to be 2.269 if the interaction energy J and the Boltzmann constant k_B are set to one. In 1944 Lars Onsager advanced the previous work and found a full analytical solution for the 2D model [10]. On page 138 of his paper "Crystal Statistics. I. A Two-Dimensional Model with an Order-Disorder Transition", he arrives the following expression for the partition function, Z,

$$\frac{\ln(Z)}{N} = \ln\left(2\cosh\left(\frac{2J}{k_B T}\right)\right) + \frac{1}{2\pi}\int_0^{\pi} \ln\left[1 + \sqrt{1 - \left(\frac{2\sinh\left(\frac{2J}{k_B T}\right)\cos(\phi)}{\cosh^2\left(\frac{2J}{k_B T}\right)}\right)^2}\,\right] \mathrm{d}\phi \tag{7}$$

where N is the number of atoms on the lattice. It is convenient to keep the partition function in this form as it can be related to useful quantities such as the expectation energy as follows

$$\frac{\langle E\rangle}{N} = k_B T^2 \frac{\partial}{\partial T}\left(\frac{\ln(Z)}{N}\right) \tag{8}$$

The partition function can also be related to the variance of the expectation energy using

$$\frac{\langle(\Delta E)^2\rangle}{N} = k_B^2 T^3 \left[2\frac{\partial}{\partial T}\left(\frac{ln(Z)}{N}\right) + T\frac{\partial^2}{\partial T^2}\left(\frac{ln(Z)}{N}\right)\right] \tag{9}$$

Equation 7 can be solved numerically and therefore equations (8) and (9) will yield useful results. These results will be useful when assessing the performance of the 2D classifiers and their determination of the critical temperature. In figures 1 and 2 equations (8) and (9) are plotted. In figure 2 a singularity occurs at the critical temperature, T_c=2.269.

2.3 Monte-Carlo Simulation of Ising Model

In order to analyse the Ising model it is useful to simulate the model using a Monte-Carlo algorithm. In this report the popular Metropolis-Hastings algorithm is used. The algorithm works by randomly selecting an atom in the lattice and then determining the change in the systems energy due to flipping the selected atom. If the systems energy decreases, then the

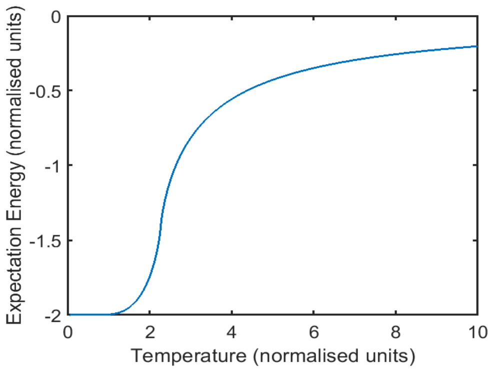


Figure 1: The expectation energy of the 2D Ising model against the temperature of the model. Numerically calculated using equations (7) and (8).

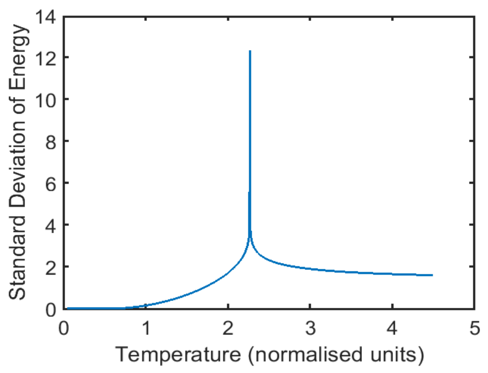


Figure 2: The standard deviation of the expectation energy of the 2D Ising model against the temperature of the model.

change is permanent as the flipped state is energetically favourable. On the other hand, if the flip causes an increase in the system energy, then the flip occurs with a probability given by

$$P = e^{\frac{-\left(H_f - H_i\right)}{k_B T}} \tag{10}$$

where H_f is the Hamiltonian of the flipped state and H_i is the Hamiltonian of the initial state. This process is iterated over until the system reaches equilibrium. The final state of the system will be microstate of the configuration space and each microstate occurs with their corresponding configuration probability.

2.4 Multi-Layer Perceptrons (MLP)

Since the late 2000s there has been a rise in the application of MLPs and other similar machine learning models. In the article, "The rise of deep learning", there is detailed account of the rise in prominence and an explanation for their success in solving problems which cannot be tackled analytically [11]. The key reasons for first selecting an MLP are that: they can map complex non-linear functions, they are well suited to generalising the problem unlike other algorithms which can have greater bias, and as a first approach, they are flexible and relatively easy to implement. However, as will be highlighted later in the report, MLPs are not the optimal architecture for this problem. A schematic for a basic MLP is shown in figure 3. The perceptron takes an input of the spin configuration, in 2D the configuration is flattened to a 1D array. The inputted configuration is then carried into the first hidden layer through a series of weighted connections. The neurons in the first hidden layer have an activation corresponding to the weighted sum of the input, the weights are crucial to the process as they provide the means for the perceptron to "learn" an optimal representation. The simplest form of the activation of the j^{th} neuron can be written mathematically as,

$$a_j = \sum_i w_i a_i + b \tag{11}$$

where a_i is the activation of the i^{th} neuron which is connected to the j^{th} neuron in the previous layer by the weight w_i and b is the bias term.

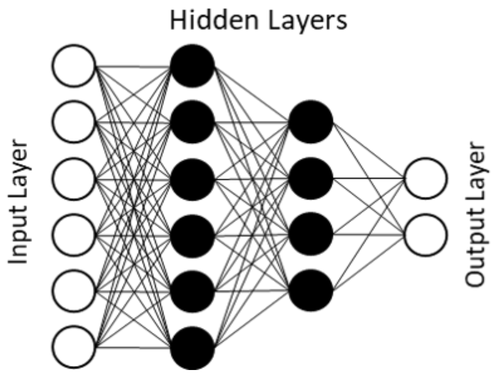


Figure 3: Schematic of multi-layer perceptron used for classification

The activations of the next hidden layer are calculated in the same way, as the weighted sum of the previous neurons. The output of the network is given by the activations of the final layer. Each neuron in the output layer signifies a temperature, the perceptron learns to maximise the activation of the neuron which corresponds to the temperature that the inputted configuration corresponds to, while minimising the activation of all other output neurons. This process is done using the loss function and the backpropagation algorithm.

The loss of the network can be defined in various ways, in this report the loss function used was categorical cross-entropy. This loss function is mathematically defined as

$$\text{Loss} = -\sum_i \log\left(f\left(a_i\right)\right) \tag{12}$$

where the summation is performed over the neurons in the output layer, y_i is the correct output for the inputted configuration. As the desired output will always be discrete, only one class should contain a one with the rest containing zeros, the loss function can be reduced to

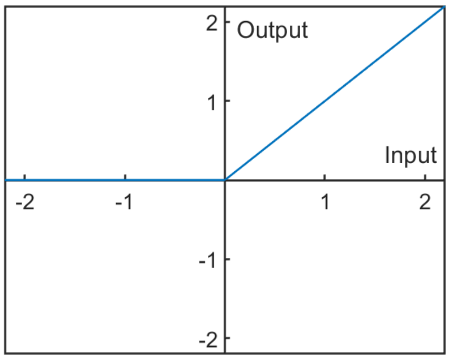
$$\text{Loss} = -\log\left(f\left(a\right)\right) \tag{13}$$

where a is the activation of the neuron corresponding to the correct temperature.

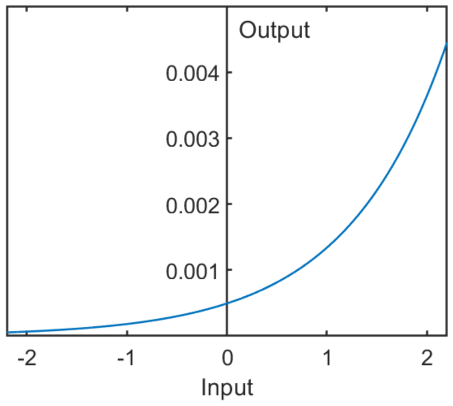
Through computing the loss function, it is possible to accurately determine the quality of the model with respect to the current weights in use. To minimise the loss of the network the backpropagation algorithm is applied. In principle the algorithm takes derivates of the loss function with respect to the weights and adjusts the weights in steps so as to move down the slope of the loss function, this process is commonly known as gradient descent. Applying gradient descent to MLPs leads to complications as each weight has a different impact on the output depending on which layer it is in. The backpropagation algorithm propagates the adjustments through the network, with each weight being adjusted relative to its effect on the output. The specifics of the algorithm are too complex to go into detail in this report, on page 734 of Russell and Norvig's textbook, "Artificial Intelligence: A Modern Approach", there is a pseudo-code example of the algorithm [12].

In this report two different activation functions are used, rectified linear units (ReLUs) and softmax activations. The simpler of the two is the ReLU function, shown in graph A of figure 4. The ReLU activation outputs a linear activation if the input is positive and outputs zero if the input is negative. ReLU's are popular for their simplicity and effectiveness.

The softmax activation is more complex, an example activation is shown in graph B of figure 4. In reality the activation is more complex than graph B might indicate. The activation



(a) ReLU activation function



(b) Softmax activation function

Figure 4: Activation Functions

of the j^{th} neuron is defined as,

$$\text{Activation}_j = \frac{e^{w_j x + b_j}}{\sum_{k=1}^{K} e^{w_k x + b_k}} \tag{14}$$

where $w_j x$ is the weighted sum of the previous layer with the j^{th} weights and the summation is performed over the K neurons in the layer, in which the j^{th} neuron also belongs. The softmax activation creates a probability distribution with each neuron having an activation corresponding to the probability of the neuron.

2.5 Convolutional Neural Networks

Convolutional neural networks are a specific type of deep learning architecture. Unlike perceptrons CNNs use and train a convolutional layer to extract information from an input. In recent years CNNs have proven to be well suited to computer vision problems. In the 2015 paper, "Multi-view Face Detection Using Deep Convolutional Neural Networks", CNNs were shown to outperform the state of the art, while also being less computationally expensive [16]. Instead of requiring separate classifiers to dissect an image, CNNs can locate the important features of an input independently, for example they can distinguish the edge of facial features such as eyes. This ability to detect edges will be especially relevant in this report, edges/boundaries in a spin configuration indicate entropic forces dominating energetic forces, allowing the phase to be inferred.

In this report the advantages of CNNs for this problem will be demonstrated through comparing the classification accuracies of the different models. There will also be different methods for extracting parameters of the Ising model such as the critical temperature for the different networks.

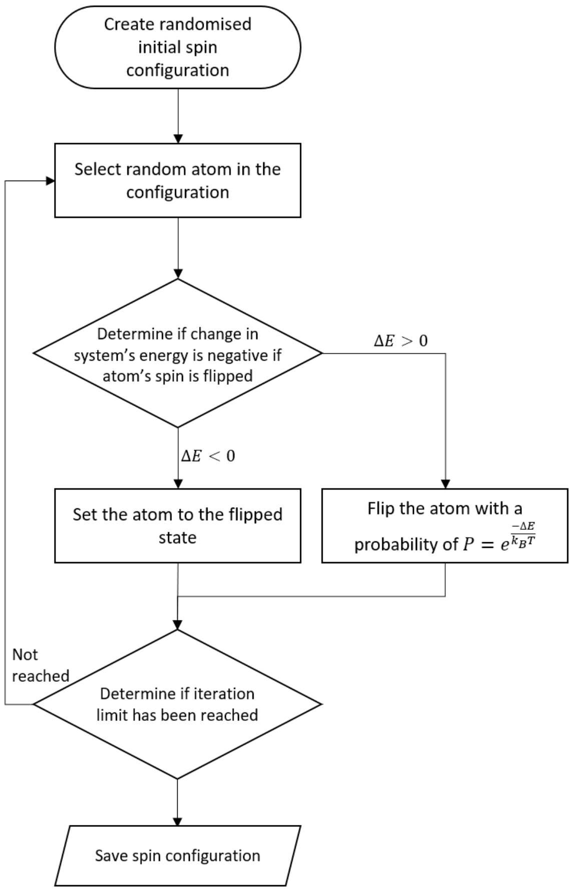

Create randomised initial spin configuration
Select random atom in the configuration
Determine if change in system's energy is negative if atom's spin is flipped
$\Delta E > 0$
$\Delta E < 0$
Set the atom to the flipped state
Flip the atom with a probability of $P = e^{\frac{-\Delta E}{k_B T}}$
Not reached
Determine if iteration limit has been reached
Save spin configuration

Figure 5: A flow chart of the Monte-Carlo algorithm for simulating the Ising model.

3 Method

In this project some of the raw data extracted from the networks could not be fitted to theoretical functions as there were no theoretical functions to fit to. Instead, an algorithm written by Rick Chartrand was used. The algorithm is presented in the paper "Numerical differentiation of noisy, non-smooth data" [13]. Unlike simpler approaches of finding the difference between neighbouring points, the algorithm uses regularization to prevent noise becoming amplified in the numerical derivative.

Initially 1D and 2D the Monte-Carlo simulation were encoded in Matlab. While this proved inefficient for large-scale data collection, it was useful for verifying the model and understanding the algorithm. Once these models were encoded and understood they were translated into C code. The algorithm in C was able to generate data far more efficiently, however it was also more difficult to debug. A schematic of the Monte-Carlo algorithm for simulating the Ising model is shown below in figure 5.

3.1 Development of 1D Simulation

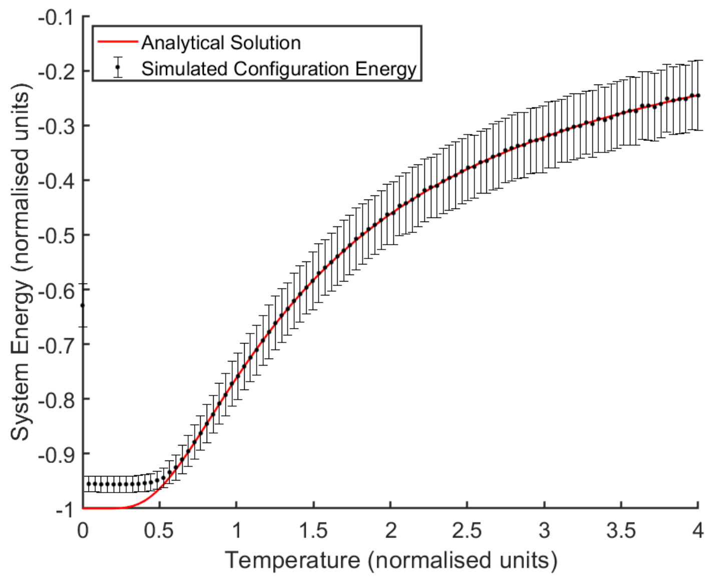


Figure 6: A plot of the mean system energy of a Monte-Carlo simulated 1D Ising model against the analytical expectation energy, with respect to temperature.

It was possible to verify the 1D model by comparing the simulation with the analytical expectation energy given in equation 4. In figure 6 the average energy of 1000 1D Monte-Carlo simulations for 100 temperatures are plotted against the analytical expectation energy. From figure 6 it can be seen that the simulation is approximately accurate to the expectation for $T > 0.5$. The reason for the divergence at low temperatures is due to the fact that the number of iterations required to reach equilibrium increases exponentially as the temperature approaches zero. At $T = 0$ there will be no entropic forces, hence the system should be in a perfectly ordered state, with no boundaries. This requires the simulation to align all atoms, this is very time consuming to simulate. In this report only configurations at $T > 0.5$ are considered.

When selecting the system size there are two key criteria: the system should relatively accurately model the behaviour of an infinitely large system and the system must be feasible to simulate and to classify. To quantify this trade off the distribution of simulated energies is plotted for various system sizes in figure 8. As would be expected the distribution shrinks as the system size increases. This result is expected as larger systems can be thought of connected smaller systems, hence will average to reduce improbable behaviour. In theory an infinitely large system would have exactly the expectation energy.

A system size of 250 lattice points was used for the 1D simulation. This lattice size was computationally viable, while also being sufficiently physically accurate.

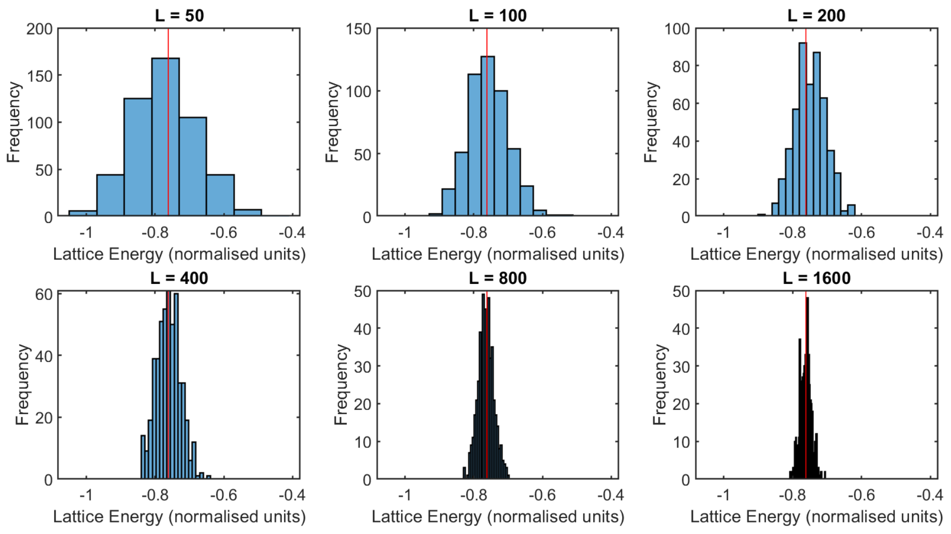


Figure 7: Histograms of the energies of 500 simulated configurations for six lattice sizes, in red the expectation energy of the system is plotted.

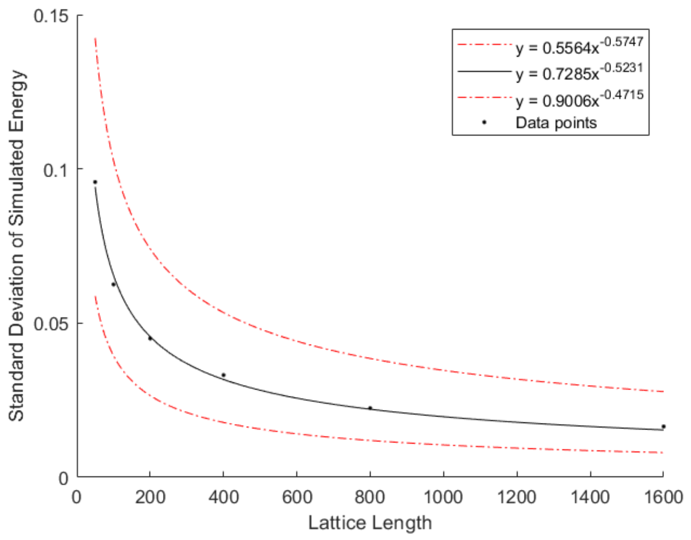


Figure 8: The standard deviation of the energy of the simulated system against the length of the simulated configuration. The data was shown to approximately fit the function y=$\frac{a}{\sqrt{L}}$ where a was found to be approximately 0.73.

3.2 Development of 2D Simulation

When simulating the Ising model in 2D the computational work was massively increased. Therefore, in order to generate enough data, it was important not to over iterate the simulation, while still iterating enough to obtain physically accurate results. To find the minimum number of iterations required the convergence of the model's energy was recorded. In figure 9 the evolution of the energy is plotted against the number of iterations performed, as expected systems of lower temperatures were seen to take longer to converge.

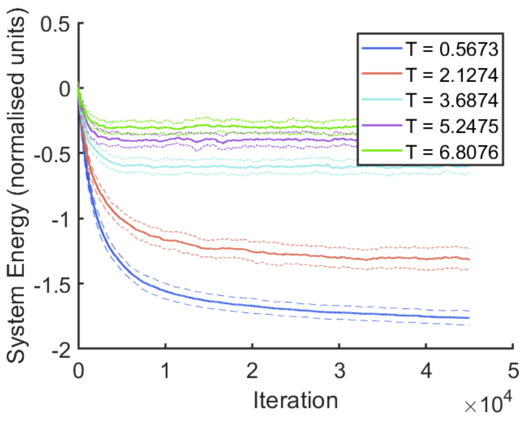


Figure 9: A plot of the system's energy against the number of iterations performed on the Monte-Carlo simulation. Each energy value is averaged from 50 examples and the standard deviation is plotted as dashed lines.

To determine the convergence of the system the energy with respect to iteration plot was divided into overlapping windows. The standard deviation of each window was then plotted in order, this plot is seen in figure 10.

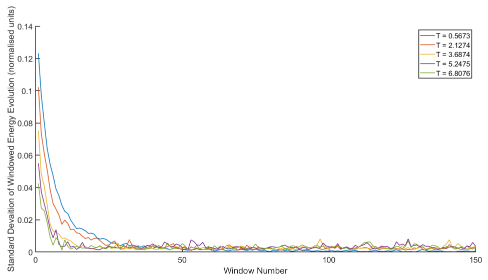


Figure 10: The convergence of the simulation measured from the standard deviation of the windowed system energy as it iterates.

There are two important features of figure 10. Firstly, as expected, the time taken for the simulation to converge increases as temperature decreases, hence when simulating these temperatures, it will be necessary to simulate for longer. Another feature is that the standard deviations of the converged high-temperature simulations are typically greater than the converged standard deviation of the low temperature simulations. This result is to be expected from figure 2 in which the analytical result for the standard deviation of the expectation energy of the 2D Ising model was plotted.

Through analysis of figure 10 it was found that a safe minimum number of Monte-Carlo iterations was $30N$, where N is the number of atoms in the 2D configuration. This number of iterations was sufficiently large to see convergence of the lowest temperature simulations, for the higher temperature simulations only $20N$ iterations were necessary.

When training networks for 30 and 100 temperature classification 5000 and 2000 examples per temperature respectively were used.

3.3 Preparation of Multi-Layer Perceptron

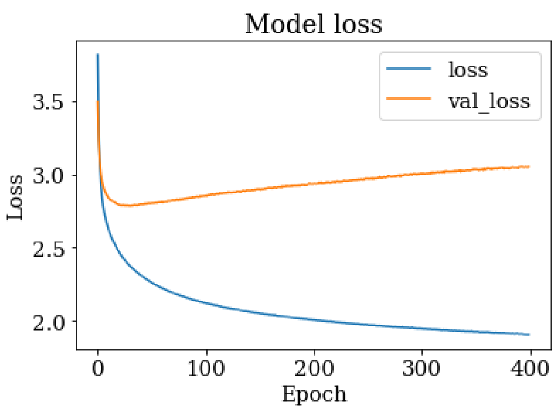
When configuring the networks there were many design decisions to make and parameters to tune. The first decision was to choose the depth of the network, there were two main considerations that guided this choice. Firstly, the network had to be able form a sufficiently complex structure in order to be able to classify phases from a raw spin configuration input. This was weighed with a desire to keep the network sufficiently simple in order to be able to understand the approach it chose for classification. A single layer was found to be too simple for temperature classification, while increasing the number of layers beyond two led to overfitting with minimal improvements in classification of unseen configurations, hence a two-layer system was chosen.

The number of neurons in the hidden layer of the network was chosen with the same considerations. A hidden layer of 80 neurons was chosen, as will be seen later the neurons in each layer generally operate in the same manner, and hence increasing the number of neurons in the hidden layer did not affect the network's performance, other than to increase the training time.

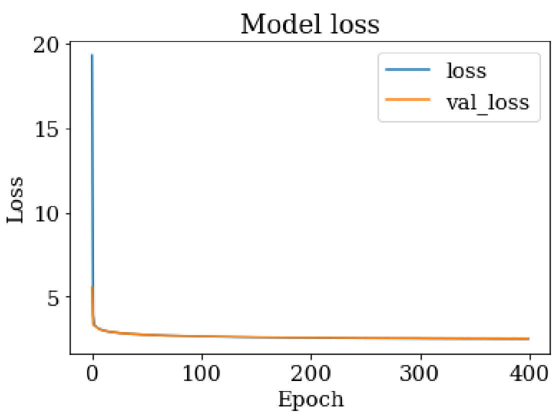
In both networks the Adam optimiser was used as it proved to converge more efficiently than other algorithms such as RMSprop. The Adam algorithm, short for "adaptive moment estimation", is a variant on the traditional back propagation algorithm [14]. In order to train optimally the algorithm varies the step size (learning rate) it takes when adjusting the parameters. For traditional algorithms the learning rate is fixed and therefore will not perform optimally throughout the descent.

The choice of activation functions to use in the perceptron was relatively simple. For the output layer it made sense to use a softmax activation, otherwise known as normalised exponential function. This activation behaves much like a probability distribution with the output being normalised. Unlike classification problems in which classes do not overlap, such as a cat or dog classification, the spin configurations inside temperature classes overlapped. Therefore, it makes sense to output the probability of a configuration belonging to a specific class. For the intermediate layer a rectified linear unit (ReLU) was used. This activation function has been shown to yield equal or better performance than a hyperbolic tangent activation, while also requiring less computational work to differentiate, which will speed up the training process [15].

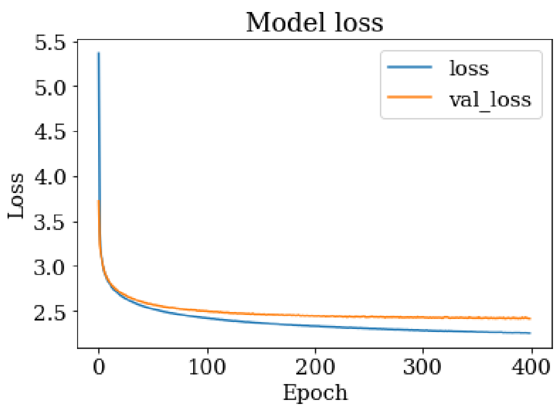
To prevent the perceptron from overfitting the training dataset the weights were regularised using $L2$ regularisation, otherwise known as "ridge regularisation". $L2$ regularisation works by adding the square of each weight's value to the loss, where the squared weight is weighted by a pre-defined coefficient. If the coefficient is large then the network will be heavily penalised for large weights, forcing it to use more of the network in classification, rather than becoming dependent on a few weights. This penalisation typically encourages the network to find a more generalised representation, often reducing potential overfitting of the seen data. However, too much regularisation can prevent the network from finding the optimal solution. For the MLP only weights connecting to the hidden layer were regularised with a coefficient of 0.03 as these were the only weights prone to overfitting. To determine for overfitting the evolution of the loss function on seen and unseen data was analysed. There are three main scenarios with regard to the regularisation, these scenarios are shown in figure 11.



(a) Overfitting with $\lambda = 0.003$



(b) Over-regularisation with $\lambda = 0.3$



(c) Optimal regularisation, with $\lambda = 0.03$

Figure 11: Training graphs for different regularisation parameters

The first scenario, seen in graph A of figure 11, is overfitting. Overfitting can be recognised from the fact that the unseen dataset's loss increases after initially falling with the seen dataset loss. The unseen data's loss increases as the network has learnt a classification approach which uses patterns in the seen dataset which are specific to the seen dataset, rather than the Ising model. Graph C of figure 11 is an example of over-regularisation. The accuracy of both models prematurely plateaus, this is because the optimal solution is unfavourable due to the loss associated to the weights. Graph B of figure 11 shows optimal regularisation. Both losses plateau without diverging or plateauing prematurely, indicating that an optimal solution has been found. To find an optimal regularisation coefficient a trial and error approach was used, more rigorous approaches exist but for this problem a near optimal solution was sufficient.

3.4 Preparation of Convolutional Neural Network

The convolutional network was found to perform sufficiently well with only one convolutional layer and one fully connected layer. This structure allowed for the network to represent the classification problem, as well as be understood. This meant that the convolutional network had two layers to tune, the convolutional layer and the output layer.

The convolutional layer had five parameters to choose: the size of the filter, the number of filters, the regularisation of the filter coefficients, step size in the convolution and the activation function of the convolutional weights. For this problem it made sense to use a $2\text{x}2$ filter, this filter size could achieve high classification accuracies while remaining easily explainable. Different filter sizes such as a $3\text{x}3$ filter were experimented with, however they did not show noticeable improvements in classification accuracy. The lack of increased accuracy with larger filter sizes will become clear later when the convolutional network's classification approach is analysed.

As with the increasing filter size, increasing the number of filters did not show improvements in classification accuracy. Hence, only one filter was used making the network both more explainable and faster to train.

The regularisation of the filter coefficients was set to zero. Unlike a fully connected layer the filter did not overfit the data. This was because the filter was kept small and hence could not encode any complexities of the seen dataset.

The step size used in the network was kept at one. This step size proved to work better than larger strides and therefore was kept simple. The activation function was chosen in a similar manner, a simple ReLU activation proved sufficient for achieving high accuracies and led to faster training. The final layer of the convolutional network was a fully connected layer, from a flattened convolution of the input to a set of output neurons, corresponding to temperature classifications. This layer used softmax activations for the same reasons as were cited in the MLP.

4 Results

The networks underwent supervised training to classify spin configurations to their corresponding temperature. At first the networks were trained to classify two and four different temperatures. The temperatures were distributed around the critical temperature with the four temperature at 0.5 T_c, T_c, 1.5 T_c and 2 T_c and the two temperatures at 0.5 T_c and 1.5 T_c. Through these initial networks it was possible to tune the hyper-parameters, such as regularization, for this classification problem. The networks were trained on 30,000 configurations per temperature, their respective test set accuracies are seen in table 1.

Model	1D		2D	
	Two temperature classification	Four temperature classification	Two temperature classification	Four temperature classification
MLP	99.9 %	80.4 %	87.5 %	84.7 %
CNN	100 %	91.3 %	99.9 %	95.4 %

Table 1: Classification accuracy for different architectures, dimensionality and number of target temperatures.

In table 1 it can be seen that the CNN consistently outperforms the MLP. This is no surprise as the convolutional network is much better suited to this classification problem. These accuracies demonstrate that the networks are capable of exploiting the underlying temperature dependence of the spin configurations in a sufficiently general manner, such that they can accurately classify unseen configurations. The next step is to analyse how the networks have learnt to estimate the temperature. Only the 2D classifiers will be analysed, the 1D classifiers used the same approaches in simpler forms.

4.1 Analysis of Multi-Layer Perceptron

The initial layer of the perceptron can be understood as a complex filter, designed to detect fluctuations in spin states. In figure 12 the weight values connecting to the hidden layer are shown.

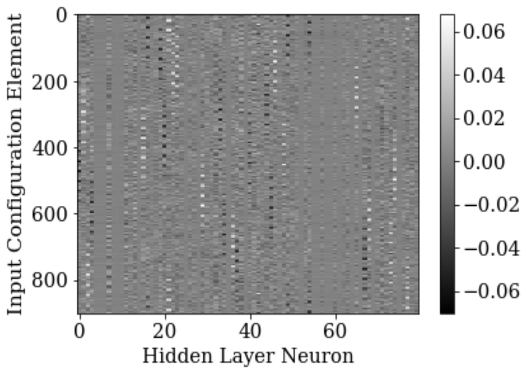


Figure 12: Value of weights connecting to neurons in the hidden layer.

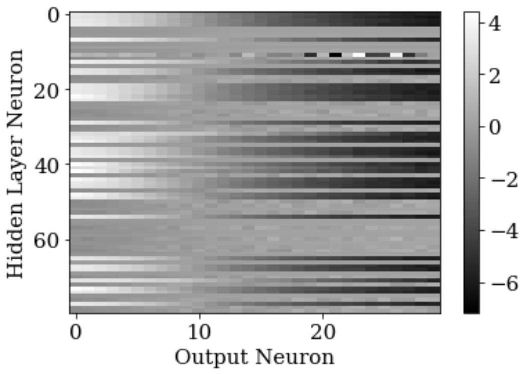


Figure 13: Value of weights connecting to final layer. The output neurons correspond to temperature classification for a 30-temperature classification problem. The neurons correspond to temperatures evenly distributed between 0.25 T_c and 3 T_c, with the 0^{th} neuron corresponding to 0.25 T_c.

In figure 13 there are horizontal strips of weights, such as the strip corresponding to the 0^{th} hidden layer neuron, which appear to decrease in weight value as the output neuron number increases. These strips correspond to hidden layer neurons connected to the vertical strips seen in figure 12. In figure 14 the average of the weights in these horizontal strips is plotted against the temperature corresponding to the output neuron.

Another crucial part of the network is the biases in the second layer. The bias for each output neuron is plotted in figure 15.

The structure of the MLP, as seen in figures 12-15, can be understood by noting that the magnitude of the output of the boundary detecting strips in the first layer will be greatest for low temperature configurations. This is because the weights are less likely to cancel out due to encountering boundaries, unlike for a high temperature configuration, in which many boundaries are present. When the hidden layer neurons have a large activation then the output neurons corresponding to low temperature classes will be greatest in activation. This is because the large hidden layer activations will be multiplied by positive coefficients for the low temperature neurons, as seen in figure 14, whereas the high temperature neurons are the product of large negative coefficients and the hidden layer activations. The biases seen in figure 15 have little effect when the product of the hidden layer activations and the output layer weights is large, meaning that the low temperature products will still be much larger than the high temperature products, despite bias. However, the biases dominate when the

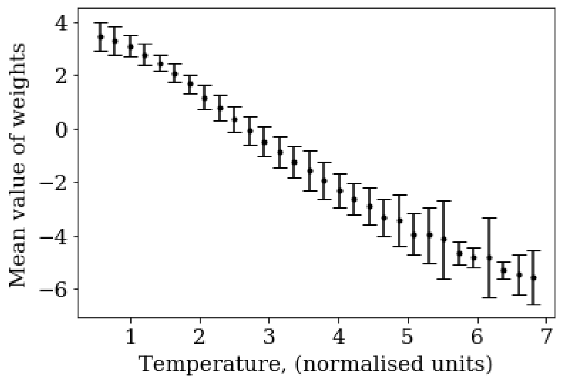


Figure 14: The mean of the weights connected to the output layer against the output neurons corresponding temperature.

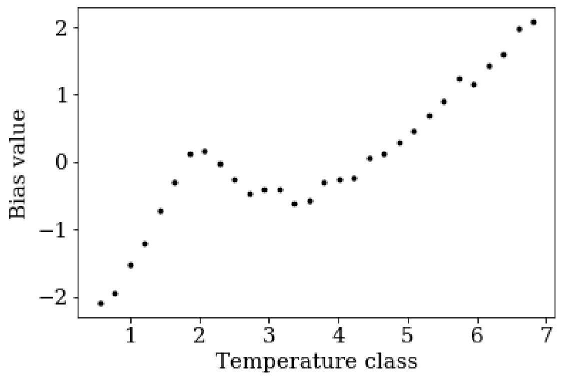


Figure 15: The bias value for each output neuron, plotted against the temperature corresponding to the output neuron.

product is small, in the case of a high temperature input, leading to the high temperature output neurons to dominate.

It is interesting to note that the biases have a local maximum at the neuron corresponding to a temperature of 2.07. This indicates that the MLP roughly locates the temperature region of phase transition.

During training the network's accuracy is determined to be the percentage of exactly correct classifications. However, as mentioned previously, the Monte-Carlo simulation of the Ising model samples the possible configurations from broad distributions, as seen in figure 2. This means that simulated models with similar temperatures will be hard to distinguish from just one instance of the configuration. The final softmax layer of the network effectively outputs the probability of the model belonging to a specific temperature. Therefore, a more realistic measure of classification accuracy of determining correct classifications was to measure the percentage of outputs which were within ± 2 classes of the correct temperature classification. For the MLP this accuracy was found to be 65.1%.

The temperature prediction can be converted to a prediction of the model being in an ordered or disordered phase. To convert the prediction the outputted probability distribution is divided around the critical temperature, the output of the network in neurons corresponding to greater than the critical temperature is defined as the probability of the system being in a disordered phase, and vice-versa. This output can be analysed in two manners, firstly the accuracy of this classification can be found and secondly the correlation in the confidence of the classification and the actual closeness to the critical temperature can be analysed. When the accuracy is defined as correct classification of phase then the test set accuracy is 97.4%. The high phase classification accuracy demonstrates that the two-layer perceptron can create a representation for solving the classification problem, without any prior knowledge of where the critical temperature occurs.

4.2 Analysis of Convolutional Neural Network

As seen in table 1 the convolutional network performed with a much higher accuracy than the MLP, therefore using this network it should be possible to more accurately find the critical temperature of the 2D Ising model. One approach to finding the critical temperature is to use the trained convolution parameters and find the greatest change in the convolved output of the configurations. The convolutional parameters that the network trains, seen in table 2, are designed to detect boundaries in the configuration. The presence of boundaries indicates entropic forces and therefore the phase of the system. The greatest change in this indicator of phase will occur at the phase boundary, this result is seen experimentally in figure 16, which shows the output of the convolutional layer after it has been trained to classify spin configurations into 100 different temperature categories. The reason for using 100 temperatures was to increase the accuracy in locating the temperature at which the gradient is maximum.

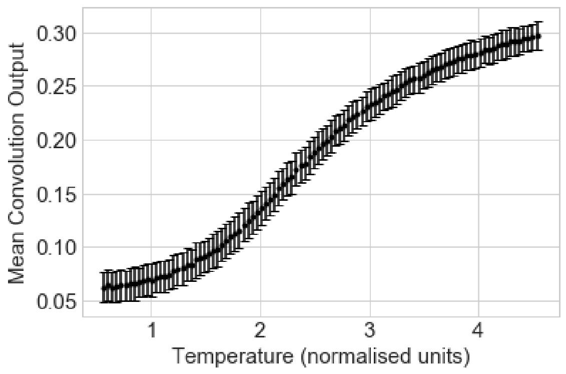


Figure 16: A plot of the mean convolved output of test set spin configurations against the temperature of the configuration. The gradient of the output is greatest at T=2.2675±0.0869, which is 0.02 standard deviations from the analytical critical temperature of the 2D Ising model, T_c=2.269.

The success of the CNN can be best understood by comparing figure 16 to figure 1, a plot of the expectation energy of the 2D Ising model against the model's temperature. The similarities in these plots led directly to a new approach for extracting the critical temperature of the 2D Ising model from the location of the maximum gradient in the mean convolution output, with respect to temperature. Ideally to find the maximum gradient of figure 16 it would be possible to fit a theoretically motivated function to the data; this is not possible in this case as there is no known equation to fit. Instead an algorithm written by Rick Chartrand was used to differentiate the raw data[13]. Using this approach, the maximum gradient in the mean convolution output was found to occur at T=2.2675±0.0869. The error in the

-0.787810	0.798473
0.789885	-0.799807

Table 2: Convolutional parameters of the trained CNN

maximum gradient location was determined by applying the algorithm 1000 times to the data with noise added to each data point and finding the standard deviation of the temperatures found in each iteration. The noise was randomly sampled from a gaussian distribution with a standard deviation equal to the standard deviation of each data point. The analytical critical temperature, T_c=2.269, lies within 0.02 standard deviations of the experimentally determined point. This is strong evidence of the CNN independently learning features of the 2D Ising model.

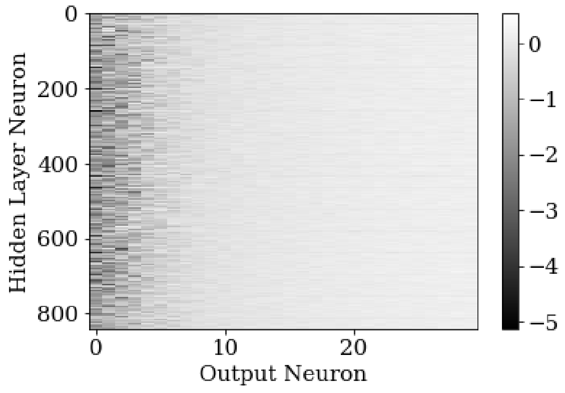


Figure 17: The weights of the final layer, connecting the convolution to the output neurons. The neurons are linearly distributed in temperature, with the 0^{th} neuron corresponds to 0.25 T_c and the 99^{th} neuron corresponds to 3 T_c.

A second aspect of the CNN to analyse is the distribution of the weights in the final layer. In figure 17 the weights in the final layer are shown, the mean of these weights is plotted against the temperatures they correspond to in figure 18.

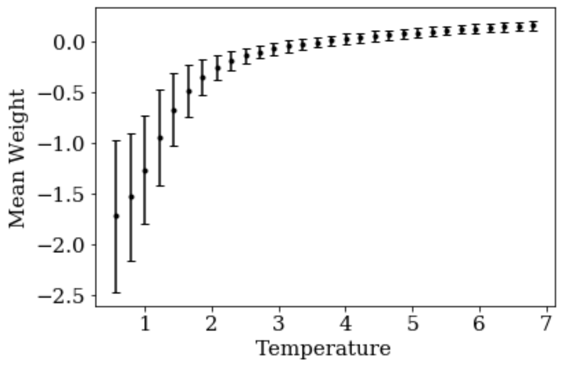


Figure 18: The mean weight value connecting to each output neuron, plotted against the temperature corresponding to the output neuron.

There is a clear trend in the weight values with respect to temperature, with different behaviour before and after the critical temperature. However, as for the MLP, the final layer weights in the CNN do not show the full picture of the classification.

To fully understand the logic of the CNN approach the bias terms have to be appreciated. The bias terms in the final layer of the CNN are plotted in figure 19.

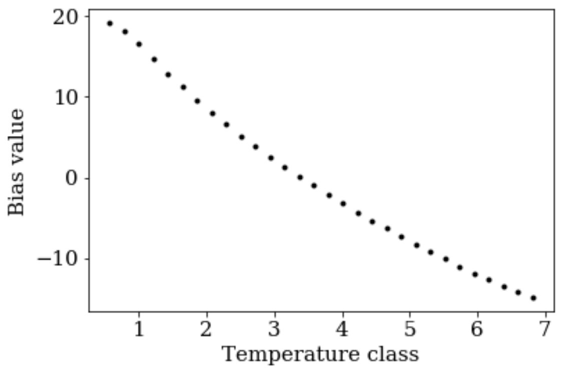


Figure 19: The output neuron bias value for the CNN.

The bias terms, seen in figure 19, follow an approximately linear trend with respect to temperature. This behaviour can be understood by following the logic of the network's approach. When the network is given a spin configuration formed at a low temperature, the convolution will output an array with a relatively small average value. The convolution outputs this array because the filter will not encounter as many boundaries. When this convolved input is passed to the final layer the product of the negative weights of neurons, corresponding to low temperatures, and the convolved input will not be very large. Therefore, the bias term will dominate and lead to large activations in the low temperature output neurons. On the other hand, the high temperature output neurons will not have large activations as their coefficients are much smaller in magnitude than the low temperature neurons, as seen in figure 18. As the bias terms for the high temperature neurons are very negative, the activations of the high temperature neurons will be much less than the low temperature neurons, leading to a correct classification. For high temperature spin configurations, the approach also holds. The weighted convolved output cancels with the bias terms for the low temperature output neurons and does not cancel for the high temperature output neurons as the bias term for these neurons is relatively small compared to the weighted sum of the convolution.

As for the MLP the percentage of CNN outputs within ± 2 classes of the correct temperature classification were determined. For the CNN this accuracy was found to be 74.1%, higher than the MLP's 65.1%.

To compare the accuracy of the CNN with the MLP the phase classification accuracy of the CNN has been calculated. This accuracy was once again computed through determining the binary phase classification of the network, which was trained to predict the temperature on the same 30 temperature dataset. The CNN achieved a 98.5% phase classification accuracy on the unseen test dataset.

5 Discussion

The 98.5% and 97.4% phase classification accuracies achieved by the CNN and MLP respectively demonstrate that both deep learning architectures are capable of accurately detecting the phase of spin configurations. The networks both proved to be able to predict the temperature of an unseen spin configuration. However, as would be expected this accuracy was limited by the inherent overlapping of configurations between temperature classes. The networks were found to be able to accurately localise the temperatures to ±2 temperature classes of the max output, with 74.1% for the CNN and 65.1% for the MLP. Analysis of the networks also shows that the networks use similar approaches to predict the temperature of a configuration.

The MLP uses the first layer to detect the degree of disorder in the input. The degree of disorder is then fed forward to the second layer where it is used to estimate temperature. The MLP approach is much cruder than the CNN as the first layer does not perform as well as a convolutional filter. Unlike the perceptron the CNN can model the energy of the system with a high degree of accuracy from the first layer. This results in the second layer of the system being more precise in classification.

An interesting extension to this work would be to apply the methods to the 3D Ising model. The 3D Ising model has so far not been analytically solved, although recent advances have been made using a conformal bootstrap approach [18]. The critical point of the 3D Ising model is thought to belong to the same universality class as liquid-vapour transitions as well as other physical systems [19]. Therefore, applying the approaches used in this report could yield new results which can be compared to real systems.

6 Summary

Both networks proved capable of learning approaches to classify spin configurations according to the temperature they were formed at. It was then shown that the temperature classification could be extended to allow the networks to classify the phase of a spin configuration. When the overlap between temperature classes was small the networks achieved much lower accuracies, as would be expected. However, increasing the density of temperature classes allowed the approaches taken by the networks to be analysed. It was found that both networks effectively measured the number of boundaries in the configurations and used this frequency to predict the temperature. The CNN was better suited to this approach, leading to its greater performance. From the graph of the average final layer weight of the CNN per output neuron, the critical temperature of the 2D Ising model was found to be 2.2675$\pm$0.0869. This result was approximately 0.02 standard deviations away from the analytical result.

7 References

References

[1] A. Cho, AI in Action: AI's early proving ground: the hunt for new particles, Science, 2017, Vol. 357, Issue 6346, pp. 20, DOI: 10.1126/science.357.6346.20

[2] Koo CL, Liew MJ, Mohamad MS, Salleh AH. A review for detecting gene-gene interactions using machine learning methods in genetic epidemiology. Biomed. Res. Int. 2013; 2013:432375. doi: 10.1155/2013/432375.

[3] Upstill-Goddard R, Eccles D, Fliege J, Collins A. Machine learning approaches for the discovery of gene-gene interactions in disease data. Brief. Bioinform. 2013; 14:251–260. doi: 10.1093/bib/bbs024.

[4] W. Hu, R. R. P. Singh, R. T. Scalettar, Discovering phases, phase transitions, and crossovers through unsupervised machine learning: A critical examination, PHYSICAL REVIEW E 95, 062122 (2017), DOI: 10.1103/PhysRevE.95.062122

[5] Tanaka, A. and Tomiya, A. Detection of phase transition via convolutional neural network. J. Phys. Soc. Jpn. 86, 063001 (2017)

[6] Ising, E. (1925), "Beitrag zur Theorie des Ferromagnetismus", Z. Phys., 31 (1): 253–258, Bibcode:1925ZPhy...31..253I, doi:10.1007/BF02980577

[7] Ising, T, Folk, R, Kenna, R, Berche, B and Holovatch, Y 2017, 'The Fate of Ernst Ising and the Fate of his Model' Journal of Physical Studies, vol 21, no.3, 3002.

[8] M. Niss, History of the Lenz–Ising Model 1950–1965: from irrelevance to relevance, Arch. Hist. Exact Sci. (2009) 63:243–287 DOI 10.1007 s00407-008-0039-5

[9] H. A. Kramers and G. H. Nannier, "Statistics of the Two-Dimensional Ferromagnet. Part I", Phys. Rev. 60, pgs. 252-262 (1941)

[10] L. Onsager, Crystal Statistics. I. A Two-Dimensional Model with an Order-Disorder Transition, Phys. Rev. 65 (1944) 117–149.

[11] S.Gleyzer, The rise of deep learning, Cern Courier, (2018)

[12] S. Russell, P. Norvig, Artificial Intelligence: A Modern Approach, 3rd ed. New Jersey, Pearson 2010

[13] Rick Chartrand, "Numerical differentiation of noisy, nonsmooth data", ISRN Applied Mathematics, Vol. 2011, Article ID 164564, 2011

[14] D. Kingma and J. Ba. Adam: a method for stochastic optimization. Arxiv, 1412.6980, 2014.

[15] X. Glorot, A. Bordes, Y. Bengio, Deep sparse rectifier neural networks. In Proceedings of the Fourteenth International Conference on Arificial Intelligence and Statistics, Fort Lauderdale, FL, USA, 11–13 April 2011, pp. 315–323.

[16] S. S. Farfade, M. J. Saberian, and L. J. Li, "Multi-view face detection using deep convolutional neural networks", in ACM Int. Conf. Multimedia Retrieval, 2015, pp. 643-650.

[17] K. Kashiwa, Y. Kikuchi, and A. Tomiya, "Phase transition encoded in neural network", arXiv:1812.01522.

[18] Kos, Filip, David Poland, and David Simmons-Duffin, 2014a, "Bootstrapping Mixed Correlators in the 3D Ising Model", J. High Energy Phys. 11, 109.

[19] D. Poland, D.S. Duffin, "The conformal Bootstrap", Nature Physics Volume 12, pages 535–539 (2016)

8 Appendix

8.1 1D Monte-Carlo Simulation in C

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

// Configure basic parameters of simulation
//L = length of chain
#define L 250
//numtemps = number of temperatures simulated
#define numtemps 2
//m = number of examples per temperature
#define m 30000
//J = interaction energy
#define J 1
//Set boltzmann constant to one
#define kB 1
//Set external magentic field to zero
#define H 0

// This function calculates the energy of the system by
   determining the energy associated with each neighbour
   interation
// It receives the configuration as an input and outputs the
   energy of the configuration
int calculateE1D(int Atoms[L])
{
    // Create arrays for shifting one to the left
    // Only left neighbours is counted to prvent double counting
        of neighbouring interactions
    int AtomsR[L];
    int j;
    for(j = 0; j < (L-1); j++){
        AtomsR[j] = Atoms[j+1];
    }
    AtomsR[L-1] = Atoms[0];
    // Sum the energy of each interaction
    int E = 0;
    for(j = 0; j < L; j++){
        E = -J*Atoms[j]*AtomsR[j] + E;
    }
    // align determines the sum of the alignments, ie sum of
        spins
    int align = 0;
    for(j = 0; j < L; j++){
        align = Atoms[j] + align;
    }
    // Add external magnetic field energy term
    E = E - H*align;
    return E;
}

// This function determines the change in the system energy due
   to flipping the element defined by co-ordinate i
int calculateE1D_change(int Atoms[L], int i)
{
    int Ei, Ef, deltaE;
    // Calculate initial energy of system, Ei
    Ei = -J*Atoms[i]*Atoms[(i+1)%L] + -J*Atoms[i]*Atoms[(i-1)%L
        ];
    // Calculate final energy of system, Ef
    Ef = J*Atoms[i]*Atoms[(i+1)%L] + J*Atoms[i]*Atoms[(i-1)%L];
    // Find change in energy
    deltaE = Ef - Ei;
    return deltaE;
}

// Determine magnetisation of configuration
int calculateM1D(int Atoms[])
{
    int i;
    int M = 0;
    // Sum spins in configuration
    for(i = 0; i < L; i++){
        M += Atoms[i];
    }
    return M;
}

int main()
{
    // Initialise random seed from timestamp
    time_t t;
    srand((unsigned) time(&t));

    // Create randomised spin configuration and temperorary
        array for storing spins
    int i;
    int Atoms[L];
    for(i = 0; i < L; i++){
        Atoms[i] = 2*(rand()%2)-1;
```

```
}
// Configure array for storing temperatures
int T0 = J/kB;
int deltaE;
float Tl = 0.5;
float Th = 1.5;
float coefs[numtemps];
for(i = 0; i < numtemps; i++){
    coefs[i] = (Th-Tl)*((float)(i)/(float)(numtemps-1)) + (
       float)(Tl);
}
// Create arrays for storing simulated data
int **Data;
Data = (int **) malloc(numtemps*m*sizeof(int *));
for(i=0; i<(numtemps*m); i++){
    Data[i] = (int *) malloc(L*sizeof(int));
}
// Initialise temperature variable
float T;
// Magnetic field measurememt disabled for speed
int M = 0;
float **yData;
yData = (float **) malloc(numtemps*m*sizeof(float *));
for(i=0; i<(numtemps*m); i++){
    yData[i] = (float *) malloc(3*sizeof(float));
}
int temp, iter, numiter, j, mi, E, Efinal;
// Configure plotting data array
float **plotData;
plotData = (float **) malloc(numtemps*m*sizeof(float *));
for(i=0;i<(numtemps*m);i++){
    plotData[i] = (float *) malloc(4*sizeof(float));
}
// Run Monte-Carlo Simulation
// Loop over examples
for(mi = 0; mi < m; mi++){
    // Loop over temperatures
    for(temp = 0; temp < numtemps; temp++){
        // Set temperature per loop
        T = coefs[temp]*T0;
        // Set number of iteratations per loop
        numiter = 100;
        // Re-randomise atomic array
        for(i = 0; i < L; i++){
            Atoms[i] = 2*(rand()%2)-1;
        }
        // Iterate simulation by selecting random horizontal
```

```
                and vertical co-ordinate and determining state
              through MC algorithm
            for(iter = 0; iter < numiter; iter++){
                for(i = 0; i < L; i++){
                    i = rand()%L;
                    deltaE = calculateE1D_change(Atoms, i);
                    // If energy of new state is less than
                        previous state then maintain flip
                    if(deltaE <= 0){
                        Atoms[i] = -1*Atoms[i];
                    }
                    // If energy of new state is greater than
                        previous state then flip specific
                        probability
                    else{
                        if(exp(-(double)deltaE/((double)(kB*T)))
                            > (double)rand()/(double)RAND_MAX){
                            Atoms[i] = -1*Atoms[i];
                        }
                    }
                }
            }
            // Magnetisation can be recorded but has been
              removed for speed.
            //M = calculateM1D(Atoms);
            // Determine final energy of system
            Efinal = calculateE1D(Atoms);
            // Store final spin configration
            for(j = 0; j < L; j++){
                Data[temp*m + mi][j] = Atoms[j];
            }
            // Store final energy, temperature and magnetisation
                of final configuration
            yData[temp*m + mi][0] = (float)(Efinal)/L;
            yData[temp*m + mi][1] = T;
            yData[temp*m + mi][2] = (float)M;
            // Store plotting data, temperature, energy,
              expectation energy and number of iterations of
              the final configuration
            plotData[temp*m + mi][0] = T;
            plotData[temp*m + mi][1] = (float)(Efinal)/L;
            plotData[temp*m + mi][2] = -tanh((double)(J)/(double
              )(kB*T));
            plotData[temp*m + mi][3] = (float)numiter;
        }
    }
    // Save data into .csv files
    FILE *f1 = fopen("CspinData1DM2T_0.4.csv", "w");
    for(j = 0; j < numtemps*m; j++){
        for(i = 0; i < L; i++){
            fprintf(f1, "%i,", Data[j][i]);
        }
        fprintf(f1,"␣␣␣␣\n");
    }
    fclose(f1);

    FILE *f2 = fopen("CspinYData1DM2T_0.4.csv", "w");
    for(j = 0; j < numtemps*m; j++){
        for(i = 0; i < 3; i++){
            fprintf(f2, "%f,", yData[j][i]);
        }
        fprintf(f2,"\n");
    }
    fclose(f2);
    /*
    FILE *f3 = fopen("CplotData1D_20T.csv", "w");
    for(j = 0; j < numtemps*m; j++){
        for(i = 0; i < 4; i++){
            fprintf(f3, "%f,", plotData[j][i]);
        }
        fprintf(f3,"\n");
    }
    fclose(f3);
    */
    return 0;
}
```

8.2 2D Monte-Carlo Simulation in C

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

// Configure basic parameters of simulation
//L = side length
#define L 30
//numtemps = number of temperatures simulated
#define numtemps 100
//m = number of examples per temperature
#define m 2000
//J = interaction energy
#define J 1
//Set boltzmann constant to one
#define kB 1
```

```
//Set external magentic field to zero
#define H 0

// This function calculates the energy of the system by
   determining the energy associated with each neighbour
   interation
// It receives the configuration as an input and outputs the
   energy of the configuration
int calculateE2D(int Atoms[L][L])
{
    // Create arrays for shifting one to the right and up one
    // Only up and right neighbours are counted to prvent double
        counting of neighbouring interactions
    int AtomsR[L][L];
    int AtomsU[L][L];
    int i,j;
    for(i = 0; i < L; i++){
        for(j = 0; j < (L-1); j++){
            AtomsR[i][j] = Atoms[i][j+1];
        }
        AtomsR[i][L-1] = Atoms[i][0];
    }
    for(j = 0; j < L; j++){
        for(i = 0; i < (L-1); i++){
            AtomsU[i][j] = Atoms[i+1][j];
        }
        AtomsU[L-1][j] = Atoms[0][j];
    }
    // align determines the sum of the alignments, ie sum of
       spins
    int align = 0;
    for(i = 0; i < L; i++){
        for(j = 0; j < L; j++){
            align = Atoms[i][j] + align;
        }
    }
    // Sum the energy of each interaction
    int E = 0;
    for(i = 0; i < L; i++){
        for(j = 0; j < L; j++){
            E = -J*Atoms[i][j]*AtomsR[i][j] + E;
            E = -J*Atoms[i][j]*AtomsU[i][j] + E;
        }
    }
    // Add external magnetic field energy term
    E = E - H*align;
    return E;
```

```
}

// This function determines the change in the system energy due
   to flipping the element defined by co-ordinates i,j
int calculateE2D_change(int Atoms[L][L], int i, int j)
{
    int Ei, Ef, deltaE;
    // Calculate initial energy of system, Ei
    Ei = -J*Atoms[i][j]*Atoms[(i+1)%L][j] + -J*Atoms[i][j]*Atoms
        [(i-1)%L][j]
    + -J*Atoms[i][j]*Atoms[i][(j+1)%L] + -J*Atoms[i][j]*Atoms[i
        ][(j-1)%L];
    // Calculate final energy of system, Ef
    Ef = J*Atoms[i][j]*Atoms[(i+1)%L][j] + J*Atoms[i][j]*Atoms[(
        i-1)%L][j]
    + J*Atoms[i][j]*Atoms[i][(j+1)%L] + J*Atoms[i][j]*Atoms[i][(
        j-1)%L];
    // Find change in energy
    deltaE = Ef - Ei;
    return deltaE;
}

// Determine magnetisation of configuration
int calculateM2D(int Atoms[L][L])
{
    int i,j;
    int M = 0;
    // Sum spins in configuration
    for(i = 0; i < L; i++){
        for(j = 0; j < L; j++){
            M += Atoms[i][j];
        }
    }
    return M;
}

int main()
{
    // Initialise random seed from timestamp
    time_t t;
    srand((unsigned) time(&t));

    // Create randomised spin configuration and temperorary
        array for storing spins
    int i,j,k,ii,jj;
    int Atoms[L][L];
    int Atomst[L][L];
    for(i = 0; i < L; i++){
        for(j = 0; j < L; j++){
            Atoms[i][j] = 2*(rand()%2)-1;
            Atomst[i][j] = Atoms[i][j];
        }
    }
    // Configure array for storing temperatures
    float T0;
    T0 = 2/log(1 + sqrt(2));
    int deltaE;
    float Tl = 0.25;
    float Th = 3;
    float coefs[numtemps];
    for(i = 0; i < numtemps; i++){
        coefs[i] = (Th-Tl)*((float)(i)/(float)(numtemps-1)) + (
           float)(Tl);
    }
    // Create arrays for storing simulated data
    int **Data;
    Data = (int **) malloc(numtemps*m*sizeof(int *));
    for(i=0; i<(numtemps*m); i++)
        Data[i] = (int *) malloc(L*L*sizeof(int));
    float Efinal;
    float **yData;
    yData = (float **) malloc(numtemps*m*sizeof(float *));
    for(i = 0; i<(numtemps*m); i++)
        yData[i] = (float *)malloc(3*sizeof(float));
    // Initialise useful variables
    int temp,iter,mi;
    int numiter = 30;
    // Magnetic field measurememt disabled for speed
    float M = 0;
    float T;
    float Tplot[numtemps*m];
    // Configure plotting data array
    int **Eplot;
    Eplot = (int **) malloc(numtemps*m*sizeof(int *));
    for(i = 0; i<(numtemps*m); i++)
        Eplot[i] = (int *)malloc((numiter*L*L + 1)*sizeof(int));
    // Run Monte-Carlo Simulation
    // Loop over examples
    for(mi = 0; mi < m; mi++){
        // Loop over temperatures
        for(temp = 0; temp < numtemps; temp++){
            // Set temperature per loop
            T = coefs[temp]*T0;
            // Re-randomise atomic array
```

```
for(i = 0; i < L; i++){
    for(j = 0; j < L; j++){
        Atoms[i][j] = 2*(rand()%2)-1;
    }
}
// Set number of iterations according to simulation
   temperature
numiter = ceil(30 - 2*T);
// Store values for plotting
Tplot[temp*m + mi] = T;
Eplot[temp*m + mi][0] = calculateE2D(Atoms);
// Iterate simulation by selecting random horizontal
    and vertical co-ordinate and determining state
   through MC algorithm
for(iter = 0; iter < numiter; iter++){
    for(i = 0; i < L; i++){
        ii = rand()%L;
        for(j = 0; j < L; j++){
            jj = rand()%L;
            Atomst[ii][jj] = Atoms[ii][jj]*-1;
            deltaE = calculateE2D_change(Atoms, ii,
               jj);
            // If energy of new state is less than
               previous state then maintain flip
            if(deltaE <= 0){
                Atoms[ii][jj] = -1*Atoms[ii][jj];
            }
            // If energy of new state is greater
               than previous state then flip
               specific probability
            else{
                if(exp(-(double)deltaE/((double)kB*(
                   double)T)) > ((double)rand()/(
                   double)RAND_MAX)){
                    Atoms[ii][jj] = -1*Atoms[ii][jj
                       ];
                }
            }
            Eplot[temp*m + mi][i*L + j + iter*L*L +
               1] = calculateE2D(Atoms);
        }
    }
}
// Magnetisation can be recorded but has been
   removed for speed.
//M = calculateM2D(Atoms);
// Determine final energy of system
```

```
            Efinal = (float)calculateE2D(Atoms)/(float)(L*L);
            // Store final spin configration
            for(j = 0; j < L; j++){
                for(i = 0; i < L; i++){
                    Data[temp*m + mi][j*L + i] = Atoms[i][j];
                }
            }
            // Store final energy, temperature and magnetisation
                of final configuration
            yData[temp*m + mi][0] = Efinal;
            yData[temp*m + mi][1] = T;
            yData[temp*m + mi][2] = M;
        }
    }

    // Save data into .csv files
    FILE *f1 = fopen("CspinData2DM_30T.csv", "a");
    for(j = 0; j < numtemps*m; j++){
        for(i = 0; i < L*L; i++){
            fprintf(f1, "%i,", Data[j][i]);
        }
        fprintf(f1,"\n");
    }
    fclose(f1);

    FILE *f2 = fopen("CspinYData2DM_30T.csv", "a");
    for(j = 0; j < numtemps*m; j++){
        for(i = 0; i < 3; i++){
            fprintf(f2, "%f,", yData[j][i]);
        }
        fprintf(f2,"\n");
    }
    fclose(f2);

    /*
    FILE *f3 = fopen("Eplot.csv", "w");
    for(i = 0; i < numtemps*m; i++){
        for(j = 0; j < (numiter*L*L); j++){
            fprintf(f3, "%i,", Eplot[i][j]);
        }
        fprintf(f3,"\n");
    }
    fclose(f3);

    FILE *f4 = fopen("Tplot.csv", "w");
    for(i = 0; i < numtemps*m; i++){
        fprintf(f4, "%f,", Tplot[i]);
    }
    fclose(f4);
    */
    return 0;
}
```

8.3 Multi-Layer Perceptron code in python

```
# -*- coding: utf-8 -*-
"""
Created␣on␣Fri␣Mar␣22␣10:54:19␣2019

@author:␣Danie
"""
# Import relevant packages
from matplotlib import pyplot as plt
import numpy as np
import pandas as pd
from tensorflow import keras
from keras.models import Sequential
from keras.layers import Dense
from keras import optimizers
from keras import regularizers
from sklearn.model_selection import train_test_split
from keras import backend as K

# Load data from C generator
Data = pd.read_csv('largeData\CspinData2DM_30T.csv', sep=',',
   header=None)
# Convert data into a usable format
L = int(np.sqrt(np.size(Data,1) - 1))
#spinData stores in the spin configurations
spinData = np.array(Data)
spinData = spinData[:,:(L*L)]
spinData = np.array(spinData, int)

# Load data from C generator
Data = pd.read_csv('largeData\CspinYData2DM_30T.csv', sep=',',
   header=None)
# Convert data into a usable format
Data = np.array(Data, float)
Data = Data[:,:L]
# Store energy, temperature, normalised temperature and
   magnetisation data for each configuration
energyData = Data[:,0]
tempData = Data[:,1]
tempDatanorm = tempData/np.max(tempData)
magData = Data[:,2]
```

```
# Separate data into training (80%), test sets (20%)
xTrain, xTest, yTrain, yTest = train_test_split(spinData,
    tempDatanorm, test_size=0.2, random_state=42)

# Store useful details of the dataset
# m = number of examples, n = number of elements in
    configuration
# n_class = number of temperature classes simulated, m_tot =
    total number of examples
# m_perT = examples per temperature class
m = np.size(yTrain)
n = np.size(xTrain,1)
n_class = np.size(np.unique(yTrain))
m_tot = np.size(Data,0)
m_perT = int(m_tot/n_class)

# Convert output into categorical arrays, data is converted into
     corresponding data indexes,
# i.e. 0.2 => 0, 0.9 => 1, 1.6 => 2, 2.3 => 3.
temps = np.unique(tempData)
tempsnorm = np.unique(tempDatanorm)
idx = np.flip(np.arange(0,n_class), 0)
for i in idx:
    yTrain[yTrain==tempsnorm[i]] = i
    yTest[yTest==tempsnorm[i]] = i

yTrain = keras.utils.to_categorical(yTrain, n_class)
yTest = keras.utils.to_categorical(yTest, n_class)

# Construct NN
# Set Adam as optimiser
Adam = optimizers.Adam
nn = Sequential()
# Create dense layer with 80 hidden layer neurons
nn.add(Dense(80, activation='relu', input_shape=(n,),
    kernel_regularizer=regularizers.l2(0.03)))
# Ad final dense layer with output neurons
nn.add(Dense(n_class, activation='softmax'))
#nn.add(Dropout(0.1))
# Comile network
nn.compile(loss='categorical_crossentropy',optimizer=Adam(),
    metrics=['accuracy','top_k_categorical_accuracy'])
# Output summary of network
nn.summary()

# Set batch size and number of training epochs
```

```
batch_size = 2000
epochs = 400

# Fit the network to the data
history = nn.fit(xTrain, yTrain, batch_size=batch_size, epochs=
    epochs, validation_data=(xTest, yTest))
# Store loss and accuracy scores of network
score = nn.evaluate(xTest, yTest, verbose=0)

# Plot training & validation loss values
hist = pd.DataFrame(history.history)
plt.rc('font', family='serif', size=15)
hist.plot(y=['loss','val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(loc='upper right')
plt.show()

# Plot training & validation accuracy values
hist = pd.DataFrame(history.history)
fig = plt.figure(figsize=(8, 6))
plt.rc('font', family='serif', size=15)
ax = fig.add_subplot(1, 1, 1)
hist.plot(y=['acc','val_acc'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(loc='lower right')
plt.show()

# Plot training & validation top-k accuracy values
hist = pd.DataFrame(history.history)
fig = plt.figure(figsize=(8, 6))
plt.rc('font', family='serif', size=15)
ax = fig.add_subplot(1, 1, 1)
hist.plot(y=['top_k_categorical_accuracy','
    val_top_k_categorical_accuracy'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(loc='lower right')
plt.show()

# Output the test set accuracy of network
print('Test Accuracy: ', np.round(score[1],4)*100,'%')
```

```
# Display first layer weights
weights = nn.layers[0].get_weights()
weights = weights[0]
plt.rc('font', family='serif', size=15)
plt.imshow(weights, aspect='auto', cmap='gray')
plt.ylabel('Input Configuration Element')
plt.xlabel('Hidden Layer Neuron')
plt.colorbar()
plt.show()

# Display second layer weights
weights = nn.layers[1].get_weights()
weights = weights[0]
plt.rc('font', family='serif', size=15)
plt.imshow(weights, aspect='auto', cmap='gray')
plt.colorbar()
plt.ylabel('Hidden Layer Neuron')
plt.xlabel('Output Neuron')
plt.show()

# Display average of weights in second layer
weights = nn.layers[1].get_weights()
weights = weights[0]
Wsum = np.mean(weights,0)
Wstd = np.std(weights,0)
plt.rc('font', family='serif', size=15)
plt.xlabel('Temperature, (normalised units)')
plt.ylabel('Mean value of weights')
plt.errorbar(temps, Wsum, Wstd, fmt='.k', capsize=5)
plt.show()

# Find first layer weights with high variance, ie weights which
    are discerning patterns in configuration
weights = nn.layers[0].get_weights()
weights = weights[0]
filtW = np.std(weights,0)
# Define pattern once as having a high standard deviation
idxs = (filtW>np.mean(filtW))
weights = nn.layers[1].get_weights()
weights = weights[0]
weights = weights[idxs,:]
weights_mean = np.mean(weights,0)
weights_err = np.std(weights,0)
plt.rc('font', family='serif', size=15)
plt.xlabel('Temperature, (normalised units)')
plt.ylabel('Mean value of weights')
plt.errorbar(temps, weights_mean, weights_err, fmt='.k', capsize
    =5)
plt.show()

# Find output of netork for test set configurations
nn_output = K.function([nn.layers[0].input], [nn.layers[1].
    output])
zz = nn_output([xTest])[0]
# Convert yData into output neuron number
yTest_form = np.sum(yTest*np.arange(0,n_class), axis=1)
# Define ordered phase as below Tc
ordPhase = (temps<=(2/np.log(1+np.sqrt(2))))*np.ones((n_class,))
# Define disordered phase as above Tc
disPhase = (temps>(2/np.log(1+np.sqrt(2))))*np.ones((n_class,))
# Find outputted probability of configuration being in ordered
    or disordered phases
ordProb = zz*ordPhase
disProb = zz*disPhase
ordProb = np.sum(ordProb,1)
disProb = np.sum(disProb,1)
# Determine absolute probability of >50% => 1 and <50% => 0
ordProbAbs = (ordProb>=0.5)*1
disProbAbs = (disProb<0.5)*1
# Determine neurons corresponding to output temperatures which
    are above and below Tc
cTempIdx = np.argmin(abs(temps-(2/np.log(1+np.sqrt(2)))))
yordProb = (yTest_form<cTempIdx)*1
# Determine accuracy of phase classification
phaseAcc = (1-(np.sum(abs(yordProb-ordProbAbs))/np.size(yTest,0)
    ))*100
# Determine error in preduction
Prederr = abs(yordProb-ordProbAbs)*(yTest_form+1)
Prederr = Prederr[Prederr!=0]
Prederr = Prederr-1
plt.rc('font', family='serif', size=15)
fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(1, 1, 1)
ax.set_xlabel('Temperature')
ax.set_ylabel('Frequency␣of␣incorrect␣phase␣classification')
# pLot historgram of error occurences wrt temperature
plt.hist(Prederr, bins=np.size(np.unique(Prederr)))
xticks = np.unique(Prederr)[np.unique(Prederr)%5==0]
ax.set_xticks(xticks)
ax.set_xticklabels(np.round(temps[xticks.astype(int)],2))
print('Phase␣classification␣accuracy␣:', phaseAcc)
plt.show()

# Find accuracy of temperature prediction to +- 2 classes
```

```
yPred = np.argmax(zz,1)
predDiff = abs(yPred-yTest_form)
Pred_2 = np.sum(predDiff<3)/np.size(predDiff,0)
print('Temp class +-2 accuracy : ', Pred_2*100, '%')

# Print biases of second layer
weights = nn.layers[1].get_weights() # Get CNN paramaters
weights = weights[1]
plt.rc('font', family='serif', size=15)
plt.plot(temps,weights, '.k')
plt.ylabel('Bias value')
plt.xlabel('Temperature class')
plt.show()
```

8.4 Convolutional Neural Network code in python

```
# -*- coding: utf-8 -*-
"""
Created on Sat Mar 16 12:57:59 2019

@author: Danie
"""
# Import relevant packages
from matplotlib import pyplot as plt
import numpy as np
import pandas as pd
from tensorflow import keras
from keras.models import Sequential
from keras.layers import Dense, Conv2D, Flatten
from keras import optimizers
from sklearn.model_selection import train_test_split
from keras import backend as K
import NoisyNumDiff

# Load data from C generator
Data = pd.read_csv('largeData\CspinData2DM_30T.csv', sep=',',
    header=None)
# Convert data into a usable format
L = int(np.sqrt(np.size(Data,1) - 1))
#spinData stores in the spin configurations
spinData = np.array(Data)
spinData = spinData[:,:(L*L)]
spinData = np.array(spinData, int)

# Load data from C generator
Data = pd.read_csv('largeData\CspinYData2DM_30T.csv', sep=',',
    header=None)
# Convert data into a usable format
```

```
Data = np.array(Data, float)
Data = Data[:,:L]
# Store energy, temperature, normalised temperature and
   magnetisation data for each configuration
energyData = Data[:,0]
tempData = Data[:,1]
tempDatanorm = tempData/np.max(tempData)
magData = Data[:,2]

# Separate data into training (80%), test sets (20%)
xTrain, xTest, yTrain, yTest = train_test_split(spinData,
   tempDatanorm, test_size=0.2, random_state=42)

# Store useful details of the dataset
# m = number of examples, n = number of elements in
   configuration
# n_class = number of temperature classes simulated, m_tot =
   total number of examples
# m_perT = examples per temperature class
m = np.size(yTrain)
n = np.size(xTrain,1)
n_class = np.size(np.unique(yTrain))
m_tot = np.size(Data,0)
m_perT = int(m_tot/n_class)

# Convert output into categorical arrays, data is converted into
    corresponding data indexes,
# i.e. 0.2 => 0, 0.9 => 1, 1.6 => 2, 2.3 => 3.
temps = np.unique(tempData)
tempsnorm = np.unique(tempDatanorm)
idx = np.flip(np.arange(0,n_class), 0)
for i in idx:
    yTrain[yTrain==tempsnorm[i]] = i
    yTest[yTest==tempsnorm[i]] = i

yTrain = keras.utils.to_categorical(yTrain, n_class)
yTest = keras.utils.to_categorical(yTest, n_class)

# Reshape input for network
xTrain = xTrain.reshape(-1, int(np.sqrt(n)), int(np.sqrt(n)), 1)
xTest = xTest.reshape(-1, int(np.sqrt(n)), int(np.sqrt(n)), 1)

# Construct CNN
# set number of filters to one
nfilts = 1
# Set filter size to 2x2
ksize = 2
```

```
# Set Adam as optimiser
Adam = optimizers.Adam
cnn = Sequential()
# Configure convolution layer
cnn.add(Conv2D(filters=nfilts, kernel_size=ksize, input_shape=(
    int(np.sqrt(n)), int(np.sqrt(n)), 1), activation='relu'))
# Flatten convolution
cnn.add(Flatten())
# Create final dense layer to output neurons
cnn.add(Dense(n_class, activation='softmax'))
# Compile network
cnn.compile(loss='categorical_crossentropy',optimizer=Adam(),
    metrics=['categorical_accuracy'])
cnn.summary()

# Train CNN
# Set batch size and number of training epochs
batch_size = 2000
epochs = 3000

# Fit the network to the data
history = cnn.fit(xTrain, yTrain, batch_size=batch_size, epochs=
    epochs, validation_data=(xTest, yTest))
# Store loss and accuracy scores of network
score = cnn.evaluate(xTest, yTest, verbose=0)

# Plot training & validation loss values
hist = pd.DataFrame(history.history)
hist.plot(y=['loss','val_loss'])
plt.title('Model␣loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(loc='upper␣right')
plt.show()

# Plot training & validation accuracy values
hist = pd.DataFrame(history.history)
hist.plot(y=['categorical_accuracy','val_categorical_accuracy'])
plt.title('Categorical␣␣accuracy')
plt.ylabel('accuracy')
plt.xlabel('Epoch')
plt.legend(loc='lower␣right')
plt.show()

#Print test accuracy
print('Categorical␣Accuracy:␣', np.round(score[1]*100,2),'%')
```

```
# Print convolution filter/s
weights = cnn.layers[0].get_weights()
weights = weights[0]
for i in np.arange(0, nfilts):
    filtweights = np.array(weights[:,:,:,i]).reshape(ksize,ksize
        )
    print('Filter', i+1,'convolution Parameters:\n', filtweights
        )


# Calculate confusion matrix
#yTestPred = cnn.predict(xTest, batch_size=None, verbose=0,
    steps=None)
#yTestPred = (yTestPred == yTestPred.max(axis=1)[:,None]).astype
    (int)
#yTestPred = np.matmul(yTestPred, np.arange(1, n_class+1))
#yTestconf = np.matmul(yTest, np.arange(1, n_class+1))
#conmat = confusion_matrix(yTestconf, yTestPred, labels=None,
    sample_weight=None)
#df_cm = pd.DataFrame(conmat, range(n_class), range(n_class))
#sn.set(font_scale=1.4) # Set label size
#sn.heatmap(df_cm, annot=True, cmap="YlGnBu", annot_kws={"size":
     8}) # With set font size

# Get conv output
conv_output = K.function([cnn.layers[0].input],
                                  [cnn.layers[1].output])
# Determine output of convolutional layer on test set data
x = conv_output([xTest])[0]
x = np.mean(x, axis=1)
# Convert yData into output neuron number
yTest_form = np.sum(yTest*temps, axis=1)

# Plot the convolved input against temperature of configurtion
y = np.zeros(n_class)
ystd = np.zeros(n_class)
for i in np.arange(0,n_class):
    extract = (yTest_form==temps[i])*1
    y[i] = np.sum(x*extract)/np.sum(extract)
    ystd[i] = np.std(x[(x*extract)!=0])
plt.errorbar(temps, y, ystd, fmt='.k', capsize=3)
plt.xlabel('Temperature (normalised units)')
plt.ylabel('Mean Convolution Output')
plt.show()

# Find derivative of convolved configurations wrt temperature
n = 1000
```

```
Tc = np.zeros(n)
for j in range(n):
    noise = np.zeros(np.size(y))
    for i in range(np.size(ystd)):
        noise[i] = np.random.normal(0,ystd[i]**2,1)
    dy = NoisyNumDiff.TVRegDiff(y+noise, itern=5, alph=1e-7);
    dy = dy[0:100]
    Tc[j] = temps[np.argmax(dy)]

Tc_mean = np.mean(Tc)
Tc_std = np.std(Tc)
print("Tc is found to be :", Tc_mean)
print("The uncertainty in Tc was :", Tc_std)

# Extract weights of final layer
weights = cnn.layers[2].get_weights() # Get CNN paramaters
weights = weights[0]
plt.rc('font', family='serif', size=15)
plt.imshow(weights, aspect='auto', cmap='gray')
plt.ylabel('Hidden Layer Neuron')
plt.xlabel('Output Neuron')
plt.colorbar()
plt.show()

# Plot mean weight of hidden layer against temperarure of
   corresponding output neuron
Wmean = np.mean(weights,0)
Wstd = np.std(weights,0)
plt.rc('font', family='serif', size=15)
plt.ylabel('Mean Weight')
plt.xlabel('Temperature')
plt.errorbar(temps, Wmean, Wstd, fmt='.k', capsize=3)
plt.show()

# Find output of netork for test set configurations
cnn_output = K.function([cnn.layers[0].input], [cnn.layers[2].
   output])
zz = cnn_output([xTest])[0]
yTest_form = np.sum(yTest*np.arange(0,n_class), axis=1)
# Define ordered phase as below Tc
ordPhase = (temps<=(2/np.log(1+np.sqrt(2))))*np.ones((n_class,))
# Define disordered phase as above Tc
disPhase = (temps>(2/np.log(1+np.sqrt(2))))*np.ones((n_class,))
# Find outputted probability of configuration being in ordered
   or disordered phases
ordProb = zz*ordPhase
disProb = zz*disPhase
```

```
ordProb = np.sum(ordProb,1)
disProb = np.sum(disProb,1)
# Determine absolute probability of >50% => 1 and <50% => 0
ordProbAbs = (ordProb>=0.5)*1
disProbAbs = (disProb<0.5)*1
# Determine neurons corresponding to output temperatures which
   are above and below Tc
cTempIdx = np.argmin(abs(temps-(2/np.log(1+np.sqrt(2)))))
yordProb = (yTest_form<cTempIdx)*1
# Determine accuracy of phase classification
phaseAcc = (1-(np.sum(abs(yordProb-ordProbAbs))/np.size(yTest,0)
   ))*100
print("Phase␣classification␣accuracy␣:", phaseAcc)

# Find accuracy of temperature prediction to +- 2 classes
yPred = np.argmax(zz,1)
predDiff = abs(yPred-yTest_form)
Pred_2 = np.sum(predDiff<3)/np.size(predDiff,0)
print('Temp␣class␣+-2␣accuracy␣:', Pred_2*100, '%')

# Plot bias terms of final layer
weights = cnn.layers[2].get_weights() # Get CNN paramaters
weights = weights[1]
plt.rc('font', family='serif', size=15)
plt.plot(temps,weights, '.k')
plt.ylabel('Bias␣value')
plt.xlabel('Temperature␣class')
plt.show()
```