DEPARTMENT OF PHYSICS

University of Strathclyde Science

PH450 Project Report 2018-19

Learning the Ising Model

Submitted in partial fulfillment for the degree of BSc Hons Mathematics and Physics

Author:
Alexis Drakopoulos

Supervisor:
Dr. Ben Hourahine

Registration Number:
201302555

SUPA Department of Physics, University of Strathclyde, Glasgow G4 0NG

Abstract

This project investigates the potential to model continuous behaviors of statistical systems using deep convolutional neural network architectures. This is done through a case study on using deep learning to extract temperature information from the two-dimensional isotropic Ising model.

Modern literature primarily concerns itself with classification based tasks, with regression problems being tackled using more classical methods. The need for an understanding of the behavior of deep learning models in regression problems is of interest to many fields, including statistical physics. Many of the models in the field involve two or three dimensional structures with high numbers of degrees of freedom. These structures are prime candidates for the application of image based techniques, which have recently been popularized in the deep learning community.

This project aims to implement and investigate several variations of modern convolutional network architectures. The purpose of these implementations is not only to evaluate the performance of deep learning regression tasks, but also to explore current claims and debates in the literature.

200,000 synthetic Ising systems are generated using an efficient Markov chain Monte Carlo algorithm implementation.

Using this data, eight variations of a core convolutional network architecture are trained and evaluated. These variations are used to study the effects of four fundamental aspects of deep convolutional networks, the effect of batch size, mode of down-sampling, regularization method and activation function.

The results indicate that larger batch sizes show significant improvements in the training and performance of models when using the Adam optimizer.

The mode of down-sampling and regularization method has no significant effect.

The use of more complex activation functions such as Leaky ReLU show promise at improving predictive performance of models.

Results indicate a high rate of accuracy, show the promise of regression based architectures being able to learn complex representations of physical systems, and make an argument that further research is required to test whether deep learning architectures perform similarly in regression and classification problems.

Preface

The subject of this project is the application of modern machine learning methods for extracting state information from statistical physics models. Several fundamental concepts are introduced to accomplish this, including machine learning concepts, fundamental statistical physics, Markov chain Monte Carlo methods and some numerical optimization techniques.

While most concepts are introduced in the early sections, some references to optimization techniques, computer science and statistical concepts made throughout, although not required for the understanding of the project. Each section includes references to recommended resources, with supplementary work moved to the appendices.

Sections 1 through 6 cover the theory and methods used in the project, with the later sections covering the work, results and discussion. Extra discussion, work and illustrations are available in the appendices.

Section 1 briefly covers the history and concepts of both machine learning and statistical physics. Velenik and Friedli offer elegant notation for introducing the Ising model in an appropriate manner for the context of this project [1, pages 80–90]. Their notation was used in equation 2.2, which helped define the Hamiltonian and other properties.

Section 2 expands on the two-dimensional isotropic Ising model, covering its fundamental properties and introducing constraints and simplifications that are key to later work. Some references to graph theory notation are used to help define neighbouring spins.

Section 3 provides a slightly more thorough overview of Markov chain Monte Carlo methods, with a focus on the Metropolis-Hastings algorithm which is later used for converging Ising systems.

Machine learning is briefly introduced in section 4, with a comprehensive overview of neural networks in section 5 and convolutional networks in section 6. Due to the brevity of the project, much of the mathematics are avoided, with a focus on introducing the concepts and theory required for understanding the later work.

Section 7 deals with the generation of the Ising systems required for training, with some discussions on handling and processing, as well as discussions on the implementation of the Metropolis-Hastings algorithm on large-scale computers.

Section 8 describes the choices made in designing a deep convolutional architecture and presents eight variations of the topology chosen to investigate four claims in the literature.

These models are then evaluated and discussed in section 9, with specific focus on the error distributions and behaviors of the models.

Finally, the conclusion makes a case for the importance of deep learning architectures in statistical physics, and offers several avenues of interest for further research in the field.

Notation

Ising Model

- Images of Ising systems in this project such as Figure 11 are representations of the lattice where $\sigma = -1$ is white, $\sigma = +1$ is black. The axis of these images are spins against spins. Unless specified, any image of an Ising system has been run through 5×10^8 Metropolis steps.
- All units relating to the Ising model are normalized, with temperature T sometimes being referred to as $k_B T$ where $k_B \sim 1$ arbitrary unit. Thus β can be referred to as the inverse of T.

Machine Learning

- The dependent variable is referred to as the label, or ground truth.
- $\nabla_\theta F$ is the gradient of F, where $\nabla_\theta = \left(\frac{\delta}{\delta\theta_1}, \frac{\delta}{\delta\theta_2}, \cdots, \frac{\delta}{\delta\theta_n}\right)$.
- $\odot$ refers to element-wise multiplication of matrices or vectors.
- Unless otherwise specified, the term performance is used to indicate the subjective quality of a model, usually in terms of the predictive evaluation metrics being compared to other models.
- Unless otherwise specified, the terms batch and mini-batch are used interchangeably.
- The term parameters, when discussing deep learning, can refer to the hyper-parameters within algorithms such as learning rate, as well as internal learned parameters such as the weights and bias.
- The term significance is occasionally used to mean of importance, instead of its definition in statistics.

List of Figures

List of Tables

List of Algorithms

Contents

1 Introduction

1.1 Machine Learning

Machine learning, specifically neural networks were first theorized from work on creating a computational algorithmic model of neural logic by Warren McCulloch [2] in the 1940s. This work paved the way to understanding biological neural processes, and applying mathematical and algorithmic theory to artificial intelligence.

In the late 1950s, Rosenblatt published a paper describing an algorithmic model for pattern recognition based on the current theory of biological neurons [3] which he named the Perceptron.

Following this early work, teams at MIT believed that artificial intelligence could be applied on larger scales and invested significant time and resources into further research. This all came to an abrupt end in 1969 with a published book by Martin Minsky titled "Perceptron" [4], which discussed the work in the field of artificial intelligence, with specific focus on Rosenblatts work in the 1950s. Minsky wrote about several critical issues with artificial intelligence theory at the time, including the requirement for vastly more computational power than was available. Minsky was later often called responsible for the AI winter.

The AI winter was a time where much of the research in the field had been abandoned, it lasted for several decades, until modern computing provided the computational resources necessary to implement models developed in the 50s and 60s.

The modern era of distributed computing breathed life back into the field of artificial intelligence, with the past decade seeing exponential growth in research, and modern methods being used in a vast range of applications such as medical imaging [5], finance [6], fault prediction, and in STEM fields such as biology [7].

Machine learning has been widely used in the physical sciences, however several fundamental issues have slowed its adoption in physics.

By their very nature, neural network models lack interpret-ability, with their correctness often being ignored in favour of quantitative evaluation metrics [8]. They also require vast quantities of data [9] as they attempt to learn probabilistic or deterministic models, which is often hard or impossible to obtain. Most models also require an accurate understanding of the target system being available, which clashes with one of the goals of the physical sciences, which is to discover or gain understanding of new models.

Nevertheless there have been developments in attempting to reduce the amount of information required to train accurate models [10], and in finding new models through algorithms trained on simpler ones [11].

1.2 Statistical Mechanics

Statistical mechanics involves applying statistical approaches to developing an understanding of behaviors of large-scale systems that cannot be easily investigated analytically due to them usually having a high number of degrees of freedom.

There are various approaches to such a task, such as modelling systems using mathematical proofs and theorems or attempting to simplify complex models into simpler well understood models that exhibit similar properties.

One such model is the Ising model, which is used to model the ferromagnetic phase transition, which is of great interest to statistical physicists. It was first developed by Willhelm Lenz in 1920, it was then solved for the one dimensional case by Ernest Ising in 1924 [12], showing no phase transition. The two dimensional case was later solved by Lars Onsager in 1944 [13], and was the first to exhibit a phase transition. The three dimensional case has so far not been solved.

The analytical solutions are not the focus of this project and will not be covered, instead focusing on investigating into the potential for using modern machine learning methods, specifically convolutional neural networks, for retrieving information on the state of statistical systems.

2 The Ising Model

The Ising model is a simple model for describing a magnetic system that exhibits phase transitions from a paramagnetic state to a ferromagnetic phase. It can be represented as an n-dimensional lattice, in which the nodes of the lattice have binary values, usually $\sigma_i \in \{\pm 1\}$ representing spin-half notation. This project will be concerning itself with the two-dimensional case. There are several two-dimensional lattices such as the square, hexagonal, triangular, etc... This project will be focusing on the square lattice, as the others have not been shown to offer greater or more interesting state information for the problem at hand.

The two-dimensional square Ising model can be defined as a system $\mathbf{X}$ with dimensionality $n \times n$, where $n \in \mathbb{Z}^+$ and $\sigma_{i,j} \in \{\pm 1\}$, where $i, j = 1, 2, \cdots, n$ where i denotes the row and j the column of our matrix.

$$\mathbf{X} = \begin{bmatrix} \sigma_{11} & \sigma_{12} & \cdots & \sigma_{1n} \\ \sigma_{21} & \sigma_{22} & \cdots & \sigma_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{n1} & \sigma_{n2} & \cdots & \sigma_{nn} \end{bmatrix} \tag{2.1}$$

In the general case where the Ising model can be defined as a graph with configurations in some finite volume $\Lambda \Subset \mathbb{Z}^k$.

2.1 Restrictions

It is advantageous to impose several restrictions that are necessary or helpful in simplifying later work and the implementation of algorithms for working with the two-dimensional Ising model. It is also useful to occasionally use graph representations to define certain properties of the Ising model.

2.1.1 Spin Interactions

Modelling the interaction of spins becomes an exceptionally difficult computational problem, due in part to the long-distance interactions between spins. A powerful restriction key to the Ising model, is the local interaction restriction, where each vertex only interacts with its nearest neighbours. In the two dimensional case each vertex $\sigma_{i,j}$ interacts only with $\sigma_{i-1,j},\ \sigma_{i,j-1},\ \sigma_{i+1,j},\ \sigma_{i,j+1}$.

The spin interactions in a simplified model can be thought as a spin majority rule, where if neighbouring spins are of opposite sign to a central vertex, that vertex has a greater chance of changing sign, which is referred to as flipping. More discussion on the flip probability will follow in the Metropolis algorithm section.

The set of edges formed by neighbouring spins is denoted by

$$\mathscr{E}_\Lambda \triangleq \big\{\{i, j\} \subset \Lambda : i \sim j\big\}. \tag{2.2}$$

2.1.2 Boundary Conditions

In the free boundary regime the micro-states are in the set

$$\Omega_\Lambda \triangleq \{-1, +1\}^\Lambda \tag{2.3}$$

where some micro-state $\omega \in \Omega_\Lambda$ has the form $\omega \triangleq (\sigma_i(\omega))_{i \in \Lambda}$ with $\sigma_i : \Omega_\Lambda \to \{-1, +1\}$.

Non-finite lattice sizes, while required by the theory, are impossible to model numerically. It is critical to understand the effects of finite sized lattices on the behavior of the Ising model, as well as its effect on observables.

Finite systems do not feature mathematical singularities in the sense that any finite measurement, a sharp feature will be indistinguishable from a singularity.

Finite lattices also pose another problem, how does one represent the edge of such a lattice?

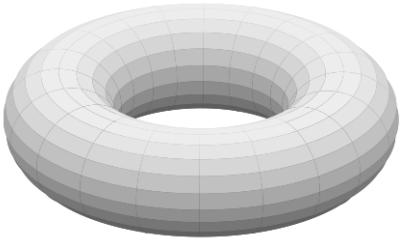


Figure 1: Illustration of torus representing topology of 2D Ising model using wrap around boundary conditions.

A common method of handling such a case is to allow for each edge to communicate with its opposite neighbour. This is topologically equivalent of mapping the two-dimensional square lattice onto a two dimensional torus.

Returning to the system outlined in equation 2.1, the spin σ_{11} would interact with $\sigma_{21}, \sigma_{12}, \sigma_{1n}, \sigma_{n1}$. This condition is often referred to as wrap around boundary condition.

Periodic boundary conditions allow for simpler simulations of finite lattices with small dimensionality, as well as enabling more expressive behavior.

2.1.3 Hamiltonian

The Hamiltonian associated with each micro-state ω can be defined as

$$\mathscr{H}_{\Lambda,J,h}(\omega) \triangleq -\sum_{\{i,j\}\in\mathscr{E}_\Lambda} J_{ij}\sigma_i(\omega)\sigma_j(\omega) - \mu\sum_{j\in\Lambda} h_j\sigma_j(\omega) \tag{2.4}$$

where $J_{ij} \in \mathbb{R}$ is the coupling parameter, σ_i is the spin at vertex i, $h_j \in \mathbb{R}$ representing an external magnetic field with μ being the magnetic moment.

The model can be simplified by assuming no external magnetic field ($h_j = 0 \quad \forall j$), which reduces the Hamiltonian to the first sum

$$\mathscr{H}_{\Lambda,J}(\omega) = -\sum_{\{i,j\}\in\mathscr{E}_\Lambda} J_{ij}\sigma_i(\omega)\sigma_j(\omega). \tag{2.5}$$

A further straightforward, yet powerful restriction that simplifies computation greatly, is that of constant interaction strength J where $\forall i, j,\ J_{ij} = J$ where $\{i, j\} \in \mathscr{E}_\Lambda$. In this project, as well as in the majority of the literature, $J = 1$.

This further reduces the Hamiltonian to

$$\mathscr{H}_\Lambda(\omega) = -J\sum_{\{i,j\}\in\mathscr{E}_\Lambda} \sigma_i(\omega)\sigma_j(\omega). \tag{2.6}$$

This Hamiltonian is the one solved by Onsager analytically, and is one of the simplest models for magnetic systems that exhibits phase transition behavior.

2.2 Probability Distribution

The Boltzmann probability distribution defines the probability of being in a certain micro-state as a function of temperature and energy.

This probability is given by

$$\mathbb{P}_\beta(\omega) = \frac{e^{-\beta\mathscr{H}_\Lambda(\omega)}}{Z_\beta} \tag{2.7}$$

where $\beta \in \mathbb{R}_{\geq 0}$ with $\beta = 1/k_BT$ being the inverse temperature, k_B is Boltzmann's constant, with partition function Z_β for the canonical ensemble being

$$Z_\beta = \sum_\omega e^{-\beta\mathscr{H}_\Lambda(\omega)}. \tag{2.8}$$

This partition function for the two dimensional case is incredibly large, but surprisingly its transfer matrix is exactly solvable in the zero magnetic field case.

In this project, unless otherwise specified, units of $k_B = 1$ are used, meaning $\beta = T^{-1}$. However to retain the generality of the mathematics, k_B is kept in the core derivations and definitions.

2.3 Observables & Properties

Observables of interest for this project follow the magnetization and energy per spin, specifically the Helmholtz free energy, magnetization and magnetic susceptibility.

First recall the Helmholtz free energy $F = -k_B T \ln Z$, with magnetization per spin being the first derivative, and magnetic susceptibility being the second.

The magnetization per spin M of an Ising system $\mathbf{X}$ is defined as

$$M(\mathbf{X}) = \frac{1}{n^2} \sum_{i=1}^{n} \sum_{j=1}^{n} \sigma_{i,j} = \frac{1}{n^2 \beta} \frac{\delta}{\delta h} \ln Z \tag{2.9}$$

where $\mathbf{X}$ has dimension $n \times n$.

This limits magnetization to $0 \leq M \leq 1$, with M being the percentage of majority spin direction. At $T = 0$ the spins are perfectly aligned with $M = 0$, and at $T = \infty$ the spins are in complete disorder with $M = 1$.

The magnetic susceptibility can then be defined to be

$$\chi = \frac{\delta M}{\delta T} = \frac{1}{n^2 \beta} \frac{\delta^2}{\delta h^2} \ln Z \tag{2.10}$$

where χ is the first derivative of magnetization with respect to temperature, which is the second derivative of the free energy with respect to magnetic field h.

One can also define the heat capacity

$$C_v = \frac{\delta \langle E \rangle}{\delta T} = -\frac{1}{k_B T^2} \frac{\delta \langle E \rangle}{\delta \beta} \tag{2.11}$$

where $\langle E \rangle$ is the mean energy $\langle E \rangle = \frac{\delta(\beta F)}{\delta \beta} = -\frac{\delta \ln Z}{\delta \beta}$.

It is also useful to derive formulae that can easily be used in finite system simulations for investigating the behavior of observables.

This can be done by re-writing the observables as variations of energy and magnetism,

$$\begin{aligned}
C_v &= \frac{\delta \langle E \rangle}{\delta T} \\
&= -\frac{1}{k_B T^2} \frac{\delta \langle E \rangle}{\delta \beta} \\
&= \frac{1}{k_B T^2} \frac{\delta^2 Z}{\delta \beta^2} \\
&= \frac{1}{k_B T^2} \frac{\delta}{\delta \beta} \left(\frac{1}{Z} \frac{\delta Z}{\delta \beta} \right) \\
&= \frac{1}{k_B T^2} \left[\frac{1}{Z} \frac{\beta^2 Z}{\delta \beta^2} - \frac{1}{Z^2} \left(\frac{\delta Z}{\delta \beta} \right)^2 \right] \\
&= \frac{1}{k_B T^2} \left[\langle E^2 \rangle - \langle E \rangle^2 \right].
\end{aligned} \tag{2.12}$$

This can be done similarly for magnetic susceptibility,

$$\chi = \frac{\delta M(\mathbf{X})}{\delta T} = \frac{\langle M(\mathbf{X})^2 \rangle - \langle M(\mathbf{X}) \rangle^2}{k_B T}. \tag{2.13}$$

Equations 2.13 and 2.12 allow for simpler computation of heat capacity and susceptibility in the finite regime, which is later used in simulations of the Ising model.

2.4 Phase Transition

Phase transitions are characterized by changes in behavior due to some change in parameter. This occurs when derivatives of the free energy $F = -k_B T \ln Z$ become discontinuous in the thermodynamic limit.

Finite systems are commonly said to be unable to exhibit phase transitions, due to having finite derivatives and sharp features, which are indistinguishable from singularities due to the nature of finite sampling. However the behavior can still be investigated in the finite regime and simulations can be compared to analytical solutions, providing adequate approximations of the behaviors of such statistical systems.

Surprisingly the two-dimensional Ising model exhibits a phase transition in the finite regime, from a paramagnetic to a ferromagnetic phase. At temperatures below the critical temperature, the Ising model experiences an ordered phase with a large majority of spins pointing in one direction, whereas above the critical temperature areas of disorder form, where the disorder becomes noise as $T \to \infty$. Of note is that the energy remains constant no matter whether the spin direction when aligned.

The Ising model as defined above has a critical temperature derived in [14] that can be found exactly as

$$T_c = \frac{2J}{k \ln(1 + \sqrt{2})} \tag{2.14}$$

where T_c is the referred to as the Curie temperature, where $T_c \approx 2.269$ when $J = 1$.

McCoy and Wu offer a more thorough overview of the analytical properties and their derivations for the two-dimensional Ising model [14].

3 Markov Chain Monte Carlo

Markov chain Monte Carlo methods (MCMC) are statistical algorithms that are often used for computing numerical approximations to systems with high dimensional probability distributions by stochastically sampling the distributions through pseudo-random number generation.

MCMC methods are of interest to statistical physicists and in this project are used to numerically approximate the two-dimensional square Ising system in a time efficient manner. The analytical solutions to many systems of interest are unknown, and although Onsager has developed one for the two dimensional case it would require an inordinate amount of computational time to generate the number of required systems.

The motivation for representing the Ising model as a Markovian model will now be discussed, along with an elementary overview of importance sampling techniques and the feasibility of using Monte Carlo simulations. Binder et al. offers a thorough overview of Monte Carlo algorithms in statistical physics [15].

3.1 Markov Chains

To understand MCMC algorithms we must first define Markov chains. A common explanation, though lacking in mathematical rigour, is to define a Markov chain as a probabilistic decision of the next step in a system, using only knowledge of the prior state, specifically with no other historical data.

Suppose we have a stochastic system $\mathbf{X}_t$, taking values from a countable set S and evolving in discrete integer time steps $t \in \mathbb{N}$. We can define the Markov property as,

$$\mathbb{P}(\mathbf{X}_{t+1} = j \mid \mathbf{X}_t = i, \mathbf{X}_{t-1} = i_{t-1}, \cdots, \mathbf{X}_0 = i_0) = \mathbb{P}(\mathbf{X}_{t+1} = j \mid \mathbf{X}_t = i) \tag{3.1}$$

where $i_0, \cdots, i_{t-1}, i, j \in S$.

The probability $\mathbb{P}(\mathbf{X}_{t+1} = j \mid \mathbf{X}_t = i)$ is called the transition probability from state i to j, often simply stated as p_{ij}.

Meyn and Tweedie offer a more thorough definition and theoretical introduction to the Markov property [16, chapter 2–3].

Since S is countable, one can write a transition matrix comprised of the transition probabilities for all micro-state evolutions from i to j. For systems with a high number of degrees of freedom this would present with an enormous amount of possible micro-states on the order of 2^N for a system with N lattice sites of binary values.

Many systems can be modelled via Markov processes, such as the Ising model where local spin interactions can be modelled in this way, due to having no temporal dependence on the flip calculation, and thus not requiring prior state information to compute the next state. The local spin interaction restriction introduced earlier also allows each micro-state evolution to be modelled by only considering a local lattice of the system containing only one spin with its immediate neighbours.

The question remains, in order to find the next step, in this case the flip probability, in this Markov process one must be able to compute the transition probability.

3.2 Monte Carlo Methods

Thermodynamic averages of observables $\mathcal{O}(\omega)$ in the canonical ensemble are given by

$$\langle \mathcal{O} \rangle = \frac{\int_\omega \mathcal{O}(\omega) e^{-\beta \mathscr{H}(\omega)} d\omega}{\int_\omega e^{-\beta \mathscr{H}(\omega)} d\omega} \tag{3.2}$$

where $\mathscr{H}$ is the system Hamiltonian.

Monte Carlo methods can be used to solve the integrals in 3.2 numerically, where classical approximations through sum with finite terms are unfeasible due to the high degree of freedom in many statistical systems.

Classical stochastic sampling techniques also pose problems due to $\int e^{-\beta \mathscr{H}}$ often experiencing sharp fluctuations near temperatures of interest, thus benefiting from importance sampling schemes such as the one proposed by Metropolis et al.[17] where the sampling is performed in accordance with a chosen probability $\mathbb{P}(\omega)$ that is proportional to the probability density of sampling state ω. Using a finite number of samples c approximates equation 3.2 to

$$\langle \mathcal{O} \rangle \approx \overline{\mathcal{O}} = \frac{c \sum_\omega \mathcal{O}(\omega) \mathbb{P}^{-1}(\omega) e^{-\beta \mathscr{H}(\omega)}}{c \sum_\omega \mathbb{P}^{-1}(\omega) e^{-\beta \mathscr{H}(\omega)}} = \frac{\sum_\omega \mathcal{O}(\omega) \mathbb{P}^{-1}(\omega) e^{-\beta \mathscr{H}(\omega)}}{\sum_\omega \mathbb{P}^{-1}(\omega) e^{-\beta \mathscr{H}(\omega)}} \tag{3.3}$$

where $\overline{\mathcal{O}}$ is the arithmetic mean. The probability can be chosen in the simplest case to be $\mathbb{P}(\omega) \propto e^{-\beta \mathscr{H}(\omega)}$ which for a finite system of N spins reduces equation 3.3 to

$$\overline{\mathcal{O}} = \frac{1}{N} \sum_\omega^N \mathcal{O}(\omega) \tag{3.4}$$

Recalling the previous Markov process 3.1, one can construct transition probabilities for a discrete set of evolving micro-states of some system $\mathbf{X}$ with some transition probabilities $\mathbb{P}(\mathbf{X}_{t+1} = j \mid \mathbf{X}_t = i)$.

Imposing the detailed balance condition

$$\mathbb{P}(\mathbf{X}_t = i)\mathbb{P}(\mathbf{X}_{t+1} = j \mid \mathbf{X}_t = i) = \mathbb{P}(\mathbf{X}_t = j)\mathbb{P}(\mathbf{X}_{t+1} = i \mid \mathbf{X}_t = j) \tag{3.5}$$

where using equation 2.7 the probability of being in state i is $\mathbb{P}(\mathbf{X}_t = i) = e^{-\beta \mathscr{H}_i}/Z_\beta$ gives an expression for the probability of transitioning from one state to the next. The difficulty lies in knowing the probability of being in a certain micro-state. This is not an issue when performing a Markov chain simulation, where each next micro-state is explored with a certain probability not depending on the previous one.

This can be seen when re-arranging equation 3.5 where the known probability of being in state i and j reduces the micro-state transition probability to the change in energy of the system

$$\frac{\mathbb{P}(\mathbf{X}_{t+1} = j \mid \mathbf{X}_t = i)}{\mathbb{P}(\mathbf{X}_{t+1} = i \mid \mathbf{X}_t = j)} = e^{-\beta \Delta \mathscr{H}} \tag{3.6}$$

where $\Delta \mathscr{H} = \mathscr{H}_i - \mathscr{H}_j$ is the change in energy between the two micro-states.

3.3 Metropolis-Hastings Algorithm

A common choice of transition probability in simulations of statistical physics systems such as the Ising model that satisfies the detailed balance condition is the Metropolis-Hastings form [18],

$$\mathbb{P}(\mathbf{X}_{t+1} = j \mid \mathbf{X}_t = i) = \begin{cases} e^{-\beta \Delta \mathscr{H}}, & \Delta \mathscr{H} > 0 \\ 1, & \Delta \mathscr{H} < 0 \end{cases} \tag{3.7}$$

The pseudo-code for importance sampling using Metropolis-Hastings algorithm to simulate quenching a two-dimensional Ising lattice is

Algorithm 1: Metropolis algorithm for Ising model Pseudo-Code

```
1. Initialize the lattice with uniform random spin distribution
2. Choose a random lattice site and calculate ΔH if site was flipped
3. Generate a random number 0 ≤ r ≤ 1
4. if r < e^{-βΔH} flip the spin
5. Go to step 2
```

A more complete version of this algorithm is available in Appendix B.1.

The question of convergence rate still remains. Frigessi et al. show that for temperatures in the range $T \leq 5$ Metropolis algorithm is ideal [19], the proof and discussions of critical temperature are outside of the scope of this project as the temperature range of interest falls within this upper-bound.

For large lattices and in an attempt to avoid the critical slow down behavior of the Metropolis algorithm, several algorithms have been developed using cluster-update rules instead of single-spin updates, the most common being the Wolff-cluster algorithm [20]. This project deals with smaller lattice sizes and does not require Wolff-cluster algorithm, however it is key to simulating larger systems.

Note that the random uniform initial state of our lattice is equivalent to an infinite temperature state, the process of convergence from this state is often referred to as a quenching process as the system reaches stability at a lower energy state.

A metropolis (or MCMC) step (or iteration) will be defined as going through step 2, 3 and 4 once, whereas some literature refers to steps as every site having been checked. This definition then means that for a system $\mathbf{X} \in \mathbb{R}^{n \times n}$, and assuming perfect uniform randomness, on average cn^2 steps are required for every spin site to be visited c times. This also serves as a reminder of the exponential nature of the number of steps required to converge Ising systems.

4 Introduction to Machine Learning

Machine learning is a branch of computational statistics focusing on algorithms that can in an automated fashion learn patterns in data. Machine learning can be informally broken down into two groups of methods, supervised learning and unsupervised learning. These groupings are not strict constructs, and algorithms can be a part of both, such as semi-supervised learning.

Supervised learning algorithms attempt to find patterns in data that has a known ground truth solution. For example an algorithm can learn pixel patterns to classify images that have been labelled by humans. Common supervised learning tasks include regression and classification.

Unsupervised learning algorithms look for patterns in data that has no label or known ground truth. A common class of algorithms would be clustering, where patterns are looked for to take observations and group them into discrete clusters often based on distance or density metrics.

This project only requires supervised learning, with synthetically generated labelled data, using modern algorithms often used for image classification.

4.1 Data Preparation & Processing

Computers depend on robustness for computation, with ambiguity resulting in at best an error and at worst false results. Statistical methods also require this robustness, for example t-tests require a normal distribution for correctness.

Some key properties that sample data must have for most supervised methods to generalize accurately to the population:

- Randomly shuffled data with no perceivable pattern in ordering.
- Equivalent distribution among training, validation and test sets.
- Samples must have equivalent distribution to population.
- Large enough sample space for generalization, including large enough training, validation and test set.

4.1.1 Train, Validation & Test Split

A common problem of automated learning algorithms is over-fitting, this is best illustrated by a quick example.

Imagine having a discrete number of observations for temperature and the time each temperature was recorded, for which an accurate model of the relationship between temperature and time is to be constructed. A regression learning algorithm can be used, whose only goal is to find a polynomial to minimize its evaluation metric, such as residual distances. Given infinite time the algorithm will inevitably construct a high dimensional polynomial that best fits the discrete observations. The issue then becomes when this sample data is not necessarily representative of the true population, often due to issues that arise with discrete sampling of a continuous process, large populations and statistical noise. The truth may be a much lower order polynomial, which the model ignored in the search for any other polynomial that would maximize its evaluation metric at the cost of generalizing to the population.

A common way to protect against this behavior, is to split the data into two sets, a training set and test set. The algorithm is then exposed to the training set, and subsequently evaluated on the test set which it has not been allowed to see. This then attempts to insure that the polynomial the algorithm has found, is not only representative of the training data but also of the overall population.

In practice data is split into three sets, training, validation and test. The training and test sets are the same as outlined above, however the validation set is used to refine the algorithm until it is ready to be evaluated on the test set. This is done by continuously training on training set, then testing against validation and fine-tuning repeatedly [21, page 222].

4.1.2 Data Distribution

A fundamental aspect of a sample is that its distribution follows that of the populations as closely as realistically possible. With synthetic data generation this becomes a question of computation and control. Using modern random number generator algorithms and a large enough sample, distribution is rarely an issue.

More commonly, even in literature, an issue that arises is improperly insuring that the data is ordered in a random manner, assuming the data does not have a spatial or temporal structure, thus accidentally introducing relationships that are not present in the population.

Data distribution in samples can often become problematic when some events or features occur in only a small percentage of cases, making it difficult to model without using prohibitively large amounts of data. The Ising model exhibits complex and rapidly evolving behaviors near the critical temperature region, this type of behavior can be difficult to accurately represent in a synthetic dataset.

5 Neural Networks

Our treatment of neural networks will not be thorough, several key aspects will be skipped but can be found in the Appendix, this includes concepts and discussions of activation functions, loss functions, learning rates, optimizers and more.

In the previous section, an example of building a relationship model between two variables was discussed. In reality, problems are often much more complex, involving many thousands of variables related in non-linear and unique ways. Attempting to understand this relationship accurately using classical methods is often an insurmountable task.

Regression or classification problem solutions can be thought of as functions that takes some input space and map it to a, usually lower dimensional, output space. A common example would be classification of images, such as recognizing a cat in an image. An image is constructed of individual pixels, usually several million. A mapping would then be to take these pixels and find a function that can understand when this specific set of pixels contains a cat. As can be imagined this function is almost impossible to find, due to the complexity of the problem, instead the aim is to produce a function approximator built up by layers of simpler functions.

5.1 Fundamentals

however for the purposes of this project a number of precise yet seemingly random choices were made, in what to introduce, based on architectural decisions that will become clear in later sections. Goodfellow et al. offers a thorough introduction to the subject [22].

5.1.1 Artificial Neuron (Perceptron)

The fundamental building block of neural networks is the previously mentioned perceptron [3]. It is often compared to the biological neuron, although it is only loosely related in representation and behavior.

The perceptron is a simple computational unit that takes in some vector of inputs x and aims to map it onto a scalar valued output y. This is done by creating a simple function that takes the sum of the product of inputs and weights. A term referred to as the bias b is then added. An algorithm is then used to attempt to find the ideal values for the weight and bias terms.

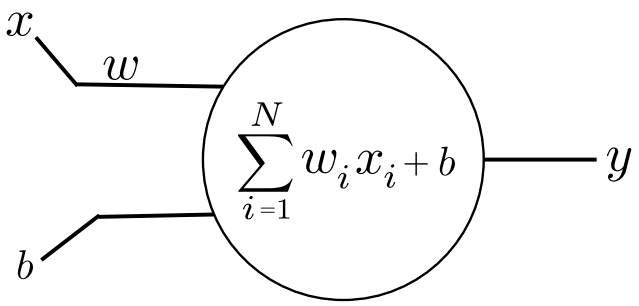


Figure 2: Illustration of a artificial neuron (also called Perceptron) with inputs $x \in \mathbb{R}^N$, output y, weights w and bias b. Activation is not illustrated.

The weights and bias are unknown parameters that we wish to approximate. Conceptually, with absolute knowledge of the weight and bias values perfect mappings could be produced. Realistically, the complexity of finding such a function requires the problem to be broken down further.

5.1.2 Activation Functions

The definition from Figure 2 is not complete, it lacks a normalizing function. In the original perceptron a threshold was used to produce a binary output of 0 or 1, this can be done through a logistic function where for example any value below 0.5 is 0, and above 0.5 is 1. The perceptron was later renamed to neuron, with the difference being the production of a continuous output instead of a binary state.

Modern neurons use a variety of functions called activation functions, these are often non-linear in nature, it can be shown that using these neurons with a nonlinear function can theoretically produce a universal function approximator [23]. The simplest being a sigmoid such as the logistic function with no threshold.

$$S(x) = \frac{e^x}{e^x + 1} \tag{5.1}$$

which produces an output $0 \leq S(x) \leq 1$.

The non-linearity in sigmoid functions allows for the constructions of complex interactions and capturing non-linear behavior in the inputs, however the gradient results in slow convergence. Due to the shape of sigmoid functions, and the fact that after some training input values are either large or small, the gradient becomes increasingly small causing a slowdown in the updates of the weights and biases, this is referred to as gradient saturation.

An alternative that has shown nearly an order of magnitude speed up in convergence are rectified linear units (ReLU) [24] which have become increasingly popular in modern architectures [25].

ReLU works on the simple principle of zeroing any negative inputs, and returning a gradient of 1 for all positive inputs, which resolves the issue of gradient saturation.

This can be defined as follows for some function $f(x)$

$$f(x) = max(0, x) \tag{5.2}$$

where x is the input.

Another issue arises, during training neurons with ReLU activation functions can learn large negative biases which effectively zero out the output of the neuron for any input, once a ReLU activated neuron arrives at this state it is unlikely to recover. This has given birth to a variety of ReLU functions to overcome this problem by introducing small gradients, such as Leaky ReLUs.

The zeroing out behavior of ReLUs introduces a sparsity which has been argued to be relevant in building more performant architectures [26]. Nonetheless, to tackle this zeroing behavior one can introduce a small gradient synthetically, this is a variant of ReLU named Leaky ReLU.

Leaky ReLU can then be defined as

$$f(x) = \begin{cases} \alpha x, & \text{if } x \leq 0 \\ x, & \text{if } x > 0 \end{cases} \tag{5.3}$$

where $\alpha \in \mathbb{R}_{\geq 0}$ is some small scaling parameter, this introduces non-linearity into ReLU.

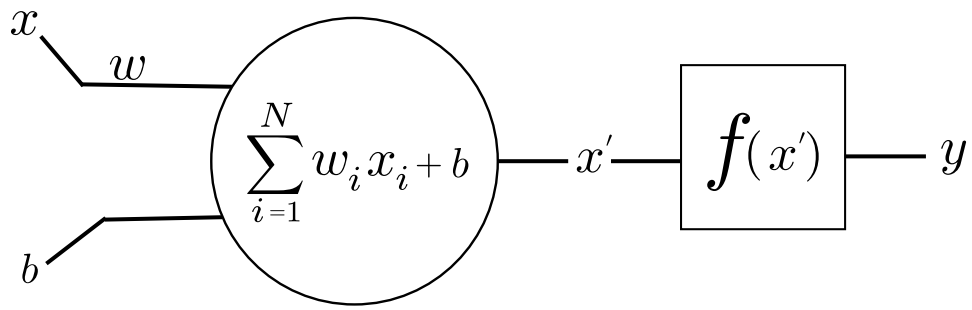


Figure 3: Illustration of a artificial neuron (also called Perceptron) with inputs $x \in \mathbb{R}^N$, unactivated output $x^{'} \in \mathbb{R}$, activation function $f(x^{'})$ and output y, with weights w and bias b.

Recalling the previously mentioned unactivated neuron in Figure 2, the addition of an activation function is illustrated in Figure 3.

5.1.3 Fully Connected Layer

Neurons can then be arranged in bipartite graph type structures, with each perceptron being connected to the previous layers activations. This allows the neuron function approximator to target smaller subsets of the input space, allowing for simpler functions to be found. These simpler functions are commonly discussed with respect to object recognition in images, with these simpler functions representing features such as edges, with the next layers features representing larger features such as ears or eyes.

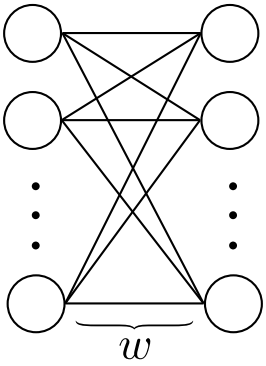


Figure 4: Illustration of a fully connected hidden layer with some input.

The number of neurons used within these layers is problem dependent, however is generally on the order of 10^3.

Defining two subsequent layers as a fully-connected bipartite graph $G = (\mathscr{U}, \mathscr{V}, \mathscr{E})$, where $|\mathscr{U}| + |\mathscr{V}| = n$, then $|\mathscr{U}| = k$ and $|\mathscr{V}| = n - k$.

The total number of edges can then be found as

$$\sum deg(\mathscr{U}) + \sum deg(\mathscr{V}) = 2k(n-k) \tag{5.4}$$

where if $|\mathscr{U}| = |\mathscr{V}|$, then $n = 2k$, which from equation 5.4 makes the total number of degrees equal to $n^2/2$.

The number of weights in fully connected layers is then twice that due to counting the number of edges for each layer, meaning n^2 parameters per layer.

This growth restricts architectures from employing large number of neurons in fully connected layers. The majority of parameters come from fully connected layers, with layers having thousands of neurons having millions of parameters.

The core parameter of a dropout layer is

```
- Number of neurons
```

where the number of neurons is often on the order of 10^3.

5.2 Classes of Problems

Many tasks involve investigating the relationship between variables, with the desired output dictating the method in which these tasks are performed. The most common tasks in statistics, and machine learning, are classification and regression.

5.2.1 Classification

Classification based problems dominate the deep learning literature, due in part to their prevalence in image based tasks. Classification concerns itself with building a model that can place predictions into a discrete number of distinct groups called classes. A popular example is the task of detecting a cat, millions of pixels can be taken as an input, with the output simply being true or false, based on whether the model believes there is a cat present.

Let us define a linear classifier for a set $\mathbf{x}$ of inputs $\mathbf{x} \in \mathbb{R}^s$, classifying into a set of finite classes $\mathbf{y} \in \mathbb{R}^c$, where s is the dimensionality of our inputs, and c is the number of classes where $c \in \mathbb{Z}^+$. Our goal is then to construct a function that maps $f : \mathbb{R}^s \mapsto \mathbb{R}^c$. Usually $c < s$ as problems try to fit a large number of variables to a smaller set of classes.

We can begin with a simple linear mapping with a bias b of dimension $c \times 1$ and a matrix of weights $\mathbf{w}$ of dimension $s \times C$, and the dimensionality of our inputs x having been flattened into a vector of dimension $s \times 1$. This function can be defined as

$$f(\mathbf{x}, \mathbf{w}, \mathbf{b}) = \mathbf{w} \cdot \mathbf{x} + b. \tag{5.5}$$

The problem then breaks down to finding the appropriate value of the weights $\mathbf{w}$ and the bias b that best fit the model.

Classification can be used to determine whether an Ising system is in a high or low temperature state. However, the problem this project focuses on is that of a continuous model.

5.2.2 Regression

Unlike classification, regression looks to find a continuous value to represent the relationship between variables, this value generally has a physical meaning, such as in this projects case the temperature associated with an Ising system.

One can similarly define a regression problem for the previous set $\mathbf{x} \in \mathbb{R}^s$ of Ising systems, however the aim is now to construct a function to map these systems to a single continuous variable $f : \mathbb{R}^s \mapsto \mathbb{R}$.

Regression problems in deep learning are uncommon, with the majority of modern literature focusing on classification.

Regression is the problem of interest in this project, where instead of looking to classify Ising systems as being above or below the critical temperature, the goal becomes to retrieve a continuous value for the temperature of the system.

Although classification dominates the deep learning literature, there is a need for the development and testing of regression based approaches in machine learning.

5.3 Training

From equation 5.5 and previous discussion, the problem of building a model is one of discovering the ideal values of W and b.

Finding the weights and biases requires a method of quantifying their performance, and then iteratively altering them until the wanted performance is reached. This method is commonly referred to as a loss or cost function, with the algorithms for altering these parameters being called optimizers.

5.3.1 Loss functions

Quantifying the performance of machine learning algorithms can be difficult. When dealing with neurons and their associated weights and biases, one must develop a method of reaching the goal of finding the ideal values required for fitting a model.

Remembering the previous linear classifier, one can see that to quantify the performance of this classifier, metrics such as accuracy could be used. The neuron would then look to maximize this metric through changing the values of the weights and biases until an appropriate accuracy is reached.

The problem of predicting the temperature of Ising models is however not one of classification, but of regression. Many metrics for quantifying the quality of regression exist from classical statistics, such as root mean squared error (RMSE), mean squared error (MSE), mean absolute error (MAE) and R^2.

The choice in evaluation metric can influence the model performance. MSE was chosen for this model, however investigation into using different metrics could be of interest. MSE is commonly used in both classical regression methods, as well as in deep learning regression models. Due to the lack of statistical mechanics deep learning regression in the literature, it is reasonable to presume MSE will perform well as has been shown in similar problems [27].

MSE can be computed as

$$\mathcal{L}(y, \hat{y}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{5.6}$$

where n is the number of predictions, $\hat{y}_i$ is the prediction i, and y_i is the ground truth value of sample i. One wishes to minimize the MSE, so henceforth when discussing optimization it will be assumed that the loss function is MSE or another function that we wish to minimize.

Recalling the introduction of the neuron and the ReLU activation, the prediction $\hat{y}_i$ for one neuron can be reformulated and our loss function is then described by

$$\mathcal{L}(y, x, w, b) = \frac{1}{n}\sum_{i=1}^{n}(y_i - max(0, \sum_{i}^{N} w_i x_i + b))^2 \tag{5.7}$$

where w are the weights, b the bias term and x the inputs. This is then generalized to larger number of neurons.

5.3.2 Optimizers

Our discussion of loss functions hinted at the manipulation of weights and biases in order to minimize an evaluation metric. The algorithms that perform these manipulations are key in the training of a model, and are aptly named optimizers.

Optimization is theoretically simple, yet in reality infinitely complex, with the perpetual goal of increasing correctness while reducing the time taken to reach this correctness. Covering numerical optimization methods is not within the scope of this project, Nocedal offers a thorough treatment of the subject [28].

Recalling our problem of finding the combination of weights and biases that minimize the MSE loss function, one can represent this as a K dimensional space where $w, b \in \mathbb{R}^K$. It will also be helpful to package w and b into a single parameter θ.

In this K dimensional topological landscape one wishes to find the global minimum, one thorough method that insures correctness would be of checking each point on this landscape. This is however unrealistically slow, and thus in reality more efficient alternatives must be used. This landscape can be navigated through the use of gradients at certain locations, with the goal of finding the global minimum.

A fundamental method in optimization of some differentiable function $F(\theta)$ is the gradient descent algorithm. We can define our position as θ_0 with subsequent positions being the sequence $\theta_1, \theta_2, \cdots$ such that,

$$\theta_{n+1} = \theta_n - \gamma \nabla_\theta F(\theta_n) \tag{5.8}$$

where γ is the step size, often called the learning rate, and $\nabla_x F(\theta_n)$ is the gradient at location θ_n. The value of γ is usually on the order of $\sim 10^{-3}$ depending on the optimizer used.

As can be seen in equation 5.8, gradient descent only converges to a global minimum assuming F is convex due to having no way to differentiate between local and global minima and not having the means to escape a minimum.

The choice of γ at each step is critical to the speed of convergence, with values too large potentially resulting in missing the global minimum, and values too small requiring a long time to converge. Several methods have been developed for this purpose, such as line search [28, pages 7–13]. The step size γ is also commonly referred to as the learning rate of an algorithm.

Updating γ after each iteration over the entire training set is slow, an alternative to speed up convergence is to update γ for each training example, this is called stochastic gradient descent (SGD) first introduced by Monro et al. [29].

For a set of training examples x with labels y we can define SGD to be

$$\theta_{n+1} = \theta_n - \gamma \nabla_\theta F(\theta_n, x^{(i)}, y^{(i)}) \tag{5.9}$$

where x_i is the ith training example, and y_i is the ith label.

Albeit SGD having significantly faster parameter update times, this can often be too fast and cause an overshoot and failure to finding the optimum local minimum. One approach to handling this issue is implementing adaptive step size γ, which can be lowered and increased based on current gradient.

A common improvement to stochastic gradient descent is mini-batch gradient descent where instead of updating parameters on each training example, parameters are updated after a subset of m examples.

$$\theta_{n+1} = \theta_n - \gamma \nabla_\theta F(\theta_n, x^{(i:i+m)}, y^{(i:i+m)}) \tag{5.10}$$

usually $30 \leq m \leq 250$ depending on memory constraints, architecture and dimensionality of parameters w.

Although these alternatives and manipulations have improved some issues, it is clear that many of the issues remain and can cause difficulty. Choosing the correct γ can be a difficult task and have large impacts on convergence and computation time. SGD algorithms also have been shown to get trapped at saddle-node points failing to find optimal minima [30].

Another problem that causes slowdown behavior is when the parameters are located in a narrow valley, where the parameters will oscillate back and forth due to the gradient being almost perpendicular to the desired trajectory.

The introduction of the momentum [31] technique allows for the reduction of these oscillations by adding a parameter ρ to the learning scheme and incorporating the previous step information

$$\theta_{n+1} = \theta_n - \gamma \nabla_\theta F(\theta_n) + \rho(\theta_n - \theta_{n-1}) \tag{5.11}$$

where ρ is usually chosen to be close to 1, often $\rho = 0.9$. It can be helpful to re-write this as

$$\Delta\theta_n = \rho \Delta\theta_{n-1} - \gamma \nabla_\theta F(\theta_n) \tag{5.12}$$

where $\Delta\theta_n = \theta_{n+1} - \theta_n$ and ρ is the momentum. This momentum parameter helps average out the oscillations which reduces the slowdown behavior.

Gradient descent updates the parameters w and b globally at each step, which is what momentum improves upon, allowing the parameters to be updated in desired dimensions along which the gradient points in a specific direction, and slowing or stopping updates along points where the gradient direction oscillates.

This can be further improved using a method developed by Nesterov [32] allowing for updates further acceleration in convergence by approximating the next move by pre-computing the gradient at $\theta_n - \rho\Delta\theta_{n-1}$ [33] and then correcting the path.

$$\Delta\theta_n = \rho\Delta\theta_{n-1} - \gamma\nabla_\theta F(\theta_n + \rho\Delta\theta_{n-1}) \tag{5.13}$$

This method of accelerating algorithms can be applied to a variety of optimization methods, and many Nesterov acceleration variations have been written.

Another recent development that combines a chosen global learning rate γ with dynamic learning rates along each dimension is Adagrad [34]

$$\Delta\theta_n = -\frac{\gamma}{\sqrt{\epsilon I + diag(\sum_{\tau=1}^{n}\mathcal{G}_\tau\mathcal{G}_\tau^\top)}}\mathcal{G}_n \tag{5.14}$$

where $\mathcal{G}_n = \frac{1}{n}\sum_{i=1}^{n}\nabla_\theta F(\theta_n, x^{(i)}, y^{(i)})$, ϵ is a small quantity to avoid division by zero and I is the identity matrix.

One of the problems with this method is the squaring of the gradients which for large parameter values decreases the learning rate quickly. For convex functions this is not an issue as it will result in fast convergence, however for non-convex topologies the algorithm can quickly become stuck in a non-optimal minimum.

Two independently developed algorithms were proposed to approach this issue, Adadelta [35] and RMSProp [36]. Adadelta is a second order method and thus requires the use of approximating the diagonal of the Hessian to control the learning rate. Adadelta and RMSProp look to use a moving average of the square gradients to avoid the issue of lowering of the learning rate that Adagrad experiences.

RMSProp and Adadelta both begin with this formulation

$$\Delta\theta_n = -\frac{\gamma}{\sqrt{\mathbb{E}[\mathcal{G}\mathcal{G}^\top]_n + \epsilon}}\mathcal{G}_n \tag{5.15}$$

where $\mathbb{E}[\mathcal{G}\mathcal{G}^\top]_n = \lambda\mathbb{E}[\mathcal{G}\mathcal{G}^\top]_{n-1} + (1-\lambda)\mathcal{G}_n \odot \mathcal{G}_{n-1}$ where λ is a decay constant.

Adadelta instead looks at the difference of the average moving window of the squared previous updates

$$\Delta\theta_n = -\frac{\sqrt{\mathbb{E}[\Delta\theta^2]_{n-1} + \epsilon}}{\sqrt{\mathbb{E}[\Delta\theta^2]_n + \epsilon}}\mathcal{G}_n \tag{5.16}$$

Finally, the Adam algorithm [37] offers the benefits of Adagrad, RMSProp and momentum, using the approximation of the first and second order gradients.

$$\Delta\theta_n = -\gamma\frac{m_n\sqrt{1-\beta_n^2}}{(1-\beta_1^n)(\sqrt{v_n} + \epsilon\sqrt{1-\beta_2^n})} \tag{5.17}$$

where $v_n = \beta_2 v_{n-1} + (1-\beta_2)\mathcal{G}_n^2$ where $\mathcal{G}_n^2 = \mathcal{G}_n \odot \mathcal{G}_n$ and $m_n = \beta_1 m_{n-1} + (1-\beta_1)\mathcal{G}_n$ and β_1, β_2 are decay coefficients.

This is a simplified pseudo-algorithm of how Adam should be run according to the original paper,

Algorithm 2: Adam Optimizer Pseudo-Code

```
Initialization
    Set γ
    Set β1, β2 ∈ [0, 1)
    Set v0, m0 = 0
    Compute θ0
While θn not converged:
    Compute Gn
```

```
    Compute v_n, m_n
    Compute θ_n
return θ_n
```

with recommended values from the original paper being $\gamma = 10^{-3}$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$.

Recalling our loss function in equation 5.7, one can write the gradient using the partial derivatives with respect to the weights and biases.

$$\mathcal{G}_n = \frac{1}{n}\sum_{i=1}^{n} \nabla_{w,b}\mathcal{L}(y^{(i)}, x^{(i)}, w_n, b_n) \tag{5.18}$$

The gradient is then generalized across a network of neurons.

Optimization algorithms such as Adam have grown in popularity due to them performing well across a wide domain of problems. However the need for more specialized optimizers that are able to out-perform algorithms such as Adam in specific tasks exists.

In this project Adam is used in all of the final models, in order to offer concrete comparison to the literature. Some investigation into other algorithms was performed in a casual manner, and thus are not included in this project.

Nocedal offers a thorough overview of numerical optimization algorithms [38], as well as covering second order methods in significant depth.

5.3.3 Epochs, Batches & Iterations

When training, it is often prohibitive to use the entire dataset. Many datasets are several orders of magnitude larger than the amount of memory that is available for computation. A simple solution is splitting the data into smaller subsets called batches, these batches are then fed into the algorithm one at a time. An iteration is then defined as the algorithm completing training over a single batch, and an epoch as having trained over the entire dataset.

The effect that batch sizes have on training is not well understood. Smith et al. show that with the Adam optimizer batch sizes did not have significant effects on final convergence, however larger batches resulted in requiring less epochs to converge [39]. However in general large batch sizes result in a lack of generalization [40]. Common batch sizes in the literature are between 32 and 512.

Repeatedly training over the dataset allows for better convergence, however over-fitting can quickly become problematic. Splitting the set into a training and validation set forces a reduction in over-fitting and improves generalization. The number of epochs greatly depends on the depth of the architecture and size of dataset.

5.4 Regularization

The non-linearity of neural networks allows them to form complex variable relationships, however a previously discussed issue arises, over-fitting. Due to samples usually not being fully representative of the population, statistical noise and small deviations in distribution cause over-fitting, resulting in a slower model to train or a decrease in predictive performance.

Several methods of regularization have shown promise in reducing this behavior, such as ℓ_1 and ℓ_2 regularization, dropout layers and batch normalization.

5.4.1 Dropout Layer

The concept behind dropout [41] is a simple one, during the training phase a certain ratio of stochastically selected neurons and their weights are temporarily removed from the network.

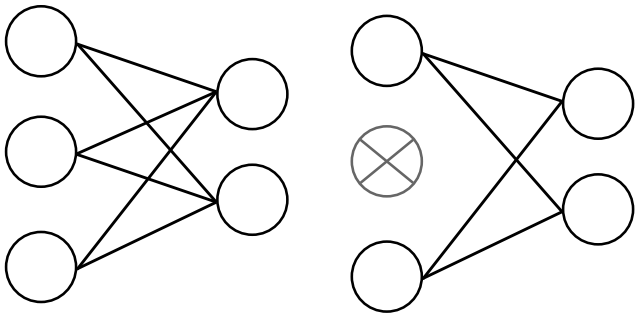


Figure 5: Illustration of a fully connected hidden layer on the left, with a dropout on the right.

When a neuron and its weights are removed, such as illustrated in Figure 3, the neighbouring neurons are forced to reduce their dependence on the missing one. Done repeatedly, this can reduce the inter-dependence within networks. This can be thought of as a way of training a multitude of variations of a neural network architecture, and subsequently averaging the result.

Dropout layers only have one core parameter,

- `Dropout rate`

where the dropout rate is the probability with which neurons are dropped. The choice of probability is problem dependent, with normal values ranging from 10%, and 50%.

5.4.2 Batch-Normalization

The weights and bias parameters are updated during training, this can shift the distribution of parameters. What batch normalization aims to do is normalize this behavior by fixing the mean and variance of the parameter distribution during training [42], and as the name implies this is done on a per-batch basis.

Algorithm 3: Batch-Normalization Algorithm Pseudo-Code

Initialize a mini-batch $\mathcal{B}$ for some inputs x where $\mathcal{B} = \{x_{1\cdots n}\}$

Compute $\mu_{\mathcal{B}} = \frac{1}{n}\sum_{i=1}^{n} x_i$

Compute $\sigma^2_{\mathcal{B}} = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu_{\mathcal{B}})^2$

Set $\hat{x}_i = \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}}$

Set $y_i = \gamma\hat{x}_i + \beta = \mathrm{BN}_{\gamma,\beta}(x_i)$

return y_i

where $\mu_{\mathcal{B}}$ is the mini-batch mean, $\sigma^2_{\mathcal{B}}$ is the mini-batch variance, and γ, β are parameters to be learned. $\mathrm{BN}_{\gamma,\beta}$ is the batch normalization transform.

The parameters used in batch normalization are

- `momentum`
- ϵ
- `centering`
- `scaling`

where the momentum is a parameter to control the speed of the moving average $\hat{x}_{i+1} = \hat{x}_i(1 - \text{momentum}) + x_i \times \text{momentum}$, ϵ is a some small value to disallow division by zero and centering and scaling are Booleans for controlling the normalization.

Having parameters with similar distribution throughout training, theoretically allows for the use of larger learning rates. In classical optimization algorithms, as the Adam optimizer outlined in equation 5.17, large global γ can cause the explosion or vanishing of gradients. Batch normalization attempts to address this behavior by reducing the ability of small changes in parameters from snowballing and causing gradient saturation or zeroing out the activation functions.

Ioffe and Szegedy argue that the use of batch normalization methods can replace dropout layers as a regularization method in network architectures [42, page 5]. It is also known that the use of batch normalization and dropout in conjunction, can damage the predictive performance of networks [43]. The literature however does not investigate this behavior in regression based problems.

Recent literature counters the claim that the normalized mean and deviation are the cause for the positive effects of batch normalization, and instead claims that this is due to smoothing of the parameter landscape [44].

There is to date, no concrete evidence as to how batch normalization improves performance, and little to no literature of its effects in regression based tasks.

6 Convolutional Networks

Convolutional networks, henceforth referred to as CNNs, are a class of deep neural networks that have revolutionized approaches to image recognition. CNNs were popularized with architectures such as AlexNet [24] in the popular ImageNet competition in 2012 in which AlexNet beat second place by a significant margin. It was one of the first deep convolutional architectures with several repeated convolution layers.

They have very successfully been used for large-scale image classification at, under ideal conditions, a higher accuracy than human candidates [45].

CNNs assume their input data is an image with a width, height and depth, where in an average image the width and height would be the pixel width and height such as 1920x1080, and the depth would be the channels, such as red, green and blue, making the input data 1920x1080x3. As one can immediately see, this is a large amount of data, at almost 6 million unique values per image. Classical neural networks with fully connected layers would struggle with this amount of data, whereas CNNs are specifically designed to handle this by having three dimensional convolution layers which have a width, height and depth.

CNNs are composed of several layers that perform different tasks and can be repeated to achieve superior results. We will design and implement a deep CNN architecture later.

CNN layers include convolution, pooling and fully connected layers.

6.1 Convolution Layer

The convolutional layer is arguably the most important development in neural networks for image modelling. Layers of neurons lack the capability of understanding spatial structure along more than one dimension, convolutional layers allow for a higher dimensional representation.

Rather than taking a vector of values as an input, which arguably removes a large part of the spatial structure in an image or in our case an Ising system, we wish to use a neuron that preserves this structure.

Instead of applying a vector of weights to the inputs, one can use a kernel of the same dimensionality as the input space. For the two-dimensional Ising model this is then a two dimensional kernel.

Convolutional layers use multiple kernels, each having their own set of weights, and sequentially move over the input space, computing the sum of the product of the inputs and weights, adding a bias and placing this value into a new two dimensional matrix.

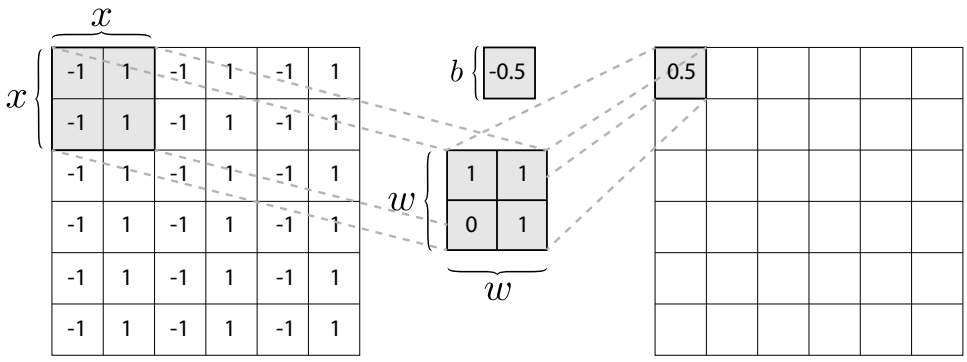


Figure 6: Illustration of a convolution layer in action on a 2D input array on the left, with a kernel size (2,2) and stride of 1. Weights w are visualized in the 2x2 kernel with a bias b of 0.5 added.

In Figure 6 a single 2×2 kernel of weights is applied to an input space. The operation is then $(-1 \times 1) + (1 \times 1) + (-1 \times 0) + (1 \times 1) + 0.5 = 1.5$.

The pseudo-code for a pooling operation over an $n \times n$ input with a 2×2 kernel and stride of 1 is

Algorithm 4: Convolution Operation Pseudo-Code

```
Initialize output matrix y ∈ ℝ^{n×n}

for n iterations
    for n iterations
        y_{i,j} = Σ_{i,j}^{i+1,j+1} w_i x_i + b
        i = i + 1
    j = j + 1

return output
```

A convolutional layer is then constructed of several filters, often several hundred, which can be thought of as neurons that learn individual features. Each filter then goes through the steps in Algorithm 4 and has its own weights and bias parameters it learns to optimize. This is often thought of as learning a specific feature, such as edge detection.

The parameters of a convolutional layer are then

- Number of filters
- Kernel size
- Stride
- Padding

where the number of filters is on the order of 10^3, the kernel size is usually 3×3 but can be larger, stride is 1 to not down-sample the inputs and padding is set to zeros to not get dimensionality error.

6.2 Pooling Layer

Pooling layers, specifically max pool, are a commonly used method for down-sampling systems within convolutional network architectures. They are applied after convolution operations, taking in the output of the convolution and reducing its dimensionality.

Pooling works in a similar sequential manner, however instead of having a stride of 1, a stride of 2 is used to down-sample the system by half. Larger strides can be used but are uncommon in the literature.

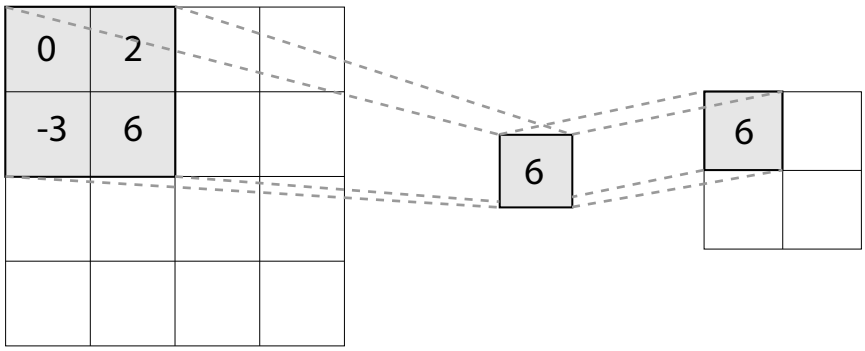


Figure 7: Illustration of a pooling layer (using max pool) in action on a 2D input array on the left, with a pool size (2, 2).

The simplest and most common pooling operation is max pool, where the largest value within a filter is chosen and the rest destroyed. This form of down-sampling results in the destruction of 75% of the data when using a stride of 2. The impact of this is problem dependent and a highly debated topic, and some modern architectures have stopped using pooling layers.

Algorithm 5: Max Pool Pseudo-Code

```
Initialize output matrix y ∈ ℝ^{n/2×n/2}

for n iterations
    for n iterations
        y_{i,j} = max_{i,j}^{i+1,j+1} x_{i,j}
        i = i + 2
    j = j + 2

return output
```

The parameters of a pooling layer are then

- `Pool Size`
- `Stride`
- `Padding`

where the pool size is usually 2×2, stride is 2 to down-sample the inputs by half along each dimension and padding is set to zeros to not get dimensionality error.

Pooling layers remain a powerful tool in reducing the dimensionality of systems down while allowing for an increased number of convolutional filters.

Due to the halving of dimensionality, many architectures and problem datasets use powers of 2 in the input data, allowing for several pooling operations to be performed.

Some modern architectures have replaced the pooling operation by instead having the final convolution layer in each set of convolutions have a stride of 2, thus doing the down-sampling that way. Dosovitskiy et al. introduce the concept of a network comprised of purely convolutional layers [46].

The effect of various down-sampling techniques within networks are still up to debate, with little literature being available that investigates the effect in regression problems.

6.3 Convolutional Architecture

To construct a convolutional neural network one then combines the aforementioned layers.

Convolutional layers are used in conjunction with pooling layers in repeated sets to reduce the input space before using fully connected layers.

The standard modern architectures usually use two or three convolutional layers, before applying a pooling layer. This is then repeated several times until the desired dimensionality is reduced.

As the model becomes deeper, the number of filters of each convolutional layer usually increases. An input may start with dimensionality $128 \times 128 \times 1$, and after several pooling and convolutions go to $4 \times 4 \times 512$. The 512 refers to each learned filter, where the 4×4 are the parameters each filter has learned.

Fully connected layers then take a flattened vector $(4 \times 4 \times 512) \rightarrow (8192 \times 1 \times 1)$ and produce some output. So a fully connected layer with 4096 neurons would take in $8192/4096 = 2$ parameter for each neuron and produce some output.

7 Synthetic Data

Deep learning algorithms benefits from vast amounts of data, with a recommended amount of 10^4 per discrete class in classification problems at minimum [47], however for continuous outputs this becomes harder to estimate. The choice of algorithm, the efficiency of the implementation and the system dimension will impact the amount of Ising systems that can realistically be generated.

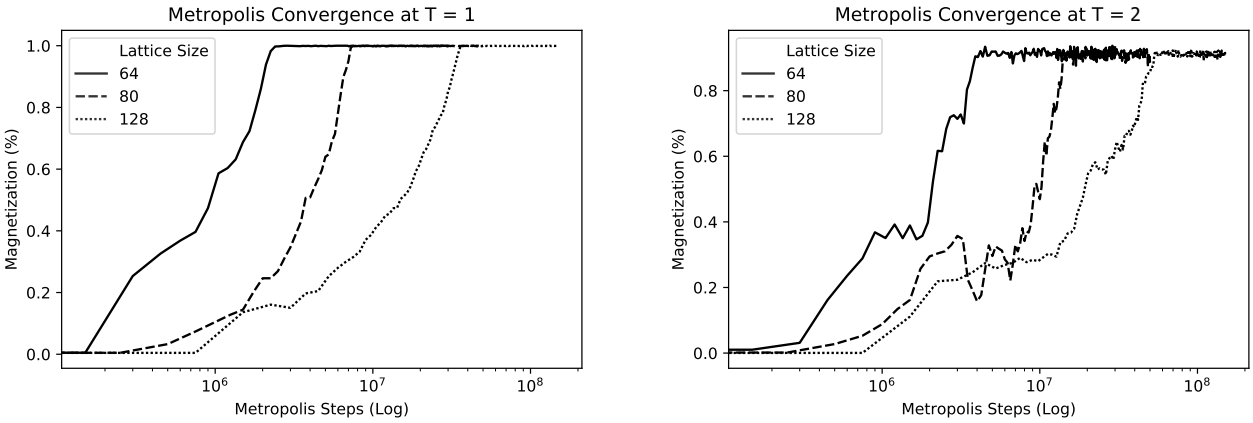
7.1 Metropolis-Hastings Python Implementation

Metropolis-Hastings MCMC methods popularity has been in part driven due to its simplicity [48], requiring only a few lines of code implement.

Implementing Metropolis-Hastings algorithm in Python is a trivial task. Python has many libraries for optimization, some popular ones such as NumPy come with efficient matrix methods [49], while others such as Cython and Numba look at compiling specific parts of Python code.

Numba was used for the implementation of the Metropolis algorithm, as it offers over two orders of magnitude speedup over NumPy using the JIT compiler [50]. The implementation can be seen in Appendix B.2, as well as further discussion on optimization.

The algorithm runs at around 2 to 3×10^7 steps per second. The difference in performance can be attributed to different processor architectures and core frequencies. This allows for fast convergence of systems with small dimensions as seen in Figure 8. However for larger systems we require above 10^7 and 10^8 Metropolis steps respectively at $T \leq 2$, as seen in Figure 9.



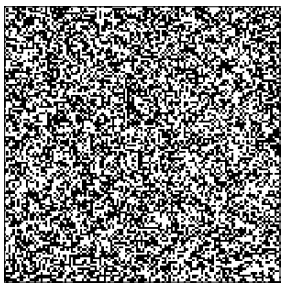
(a) Convergence rate at $T = 1$.

(b) Convergence rate at $T = 2$.

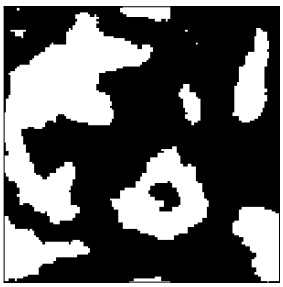
Figure 8: Discrete time steps required for magnetization convergence of Ising systems of size $n \times n$ using Metropolis-Hastings algorithm at $T = 1, 2$. Systems are of size 64, 80 and 128. 200 steps collected uniformly distributed for each system.

The choice of system size depends on the number of systems required and the availability of compute time. Other considerations such as storage size are negligible, with emphasis placed on data generation and training.

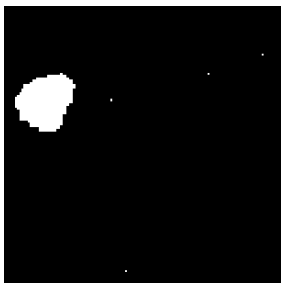
Due to the amount of systems required, and the computational cost of converging large systems, a dimension of 128x128 was chosen for the systems in this project. This is 16 times larger than similar tasks attempted in the literature [51].



(a) System at step 0. Initialization phase.



(b) System at step 10^6, complex patterns forming.



(c) System at step 12×10^6. System converging to $\sigma = +1$.

(d) Step 1.5×10^8, system fully converged with some noise.

Figure 9: 128x128 Ising system discrete time evolution with Metropolis-Hastings algorithm at $T = 1$. MCMC Steps refer to each individual metropolis check.

A 128x128 system comfortably converges after 5×10^8 metropolis steps, which remembering our previous formula cn^2 would mean $c \times (128^2) = 5 \times 10^8$, which gives a value of $c \approx 30,000$.

7.2 Scalable Metropolis Architecture

Generating $\sim 10^6$ Ising systems, even with an efficient Metropolis implementation, would require unreasonable amounts of time. Modern distributed computing architectures offer several tools for expediting this process, such as multi-core CPUs and GPU architectures.

GPUs offer an order of magnitude improvement in performance of certain Monte Carlo algorithms [52], however the complexity and difficulty of implementing the algorithms on CUDA coupled with the lack of GPU availability made this task unfeasible.

Modern workstation processors possess up to 32 cores (64 threads), which can offer vast performance improvements over single threaded compute. It is worth noting that not all workloads can be easily multi-threaded. However due to requiring a large amount of data, a single threaded workload run in parallel over these cores can theoretically offer 100% scale-ability.

A multi-threaded scale-able Metropolis algorithm was written for converging Ising systems, performing at ≈ 1 step per 100 clock cycles. Modern workstation processors run at $\approx 10^9$ cycles per second across each core, meaning that running on a 48 core (96 threads) workstation allows for over 10^9 Metropolis steps per second.

An example of the difference in time in a theoretically perfectly scale-able architecture can be seen in Appendix B.3.

The multi-threaded Metropolis implementation runs asynchronously, with each core converging an Ising system, saving it to disk when converged, and repeating. This implementation reached over 70% scale-ability over a 96 core system, more discussion on the reason for this value is available in Appendix B.4.

Table 1: Metropolis Performance on Multi-Core Systems

	Number of cores		
	1	8	96
Steps/second	21,582,733	143,884,892	1,453,488,372
Steps/core/second	21,582,733	17,985,612	15,140,504
Efficiency	-	83.3%	70.1%

Metropolis performance on 3 systems with 1, 8 and 96 threads each. Systems did not have same clock speed, and were all AWS EC2 instances.

This allows for 1.5×10^9 steps per second on a 96 core system (1.5×10^7 steps per core second).

7.3 Data-structure and Data Handling

The choice of data-structure for storing and handling the 200,000 Ising systems impacts further decisions for data handling, integrity, and processing.

Each system is two dimensional, with $128^2 = 16384$ values of ± 1. Each system also has an attributed continuous temperature label. Due to wide availability and support by the NumPy library, int8 was chosen for spin values, and float32 for temperatures. Further storage optimization could be performed by using boolean data type for spin values, this was deemed unnecessary. Further discussion is available with some calculations in Appendix A.4.1.

Temperatures are stored with 3 decimal places, more decimal places could be warranted in future work.

The NumPy library provides an efficient custom storage format (NPY), written in pure Python. This allows for structured preservation of our Ising systems, however does not allow for mixed data-types, meaning that a minimum of two files are required, one for systems and another for temperature labels. It is integral that order is preserved, several backups were performed.

Insuring that data labels and the Ising systems belonging to them are properly stored is crucial, as an error in order would require the re-creation of the dataset. Initially methods were attempted to store labels inside of the Ising arrays through simple insertion, however due to the difference in data type handling a mixed data type array proved difficult. Instead we rely on a comprehensive suite of custom tests that attempt to identify any discernible issues.

Due to the array values essentially being boolean, compression algorithms can reduce file sizes by $\sim 90\%$, enabling ease of storage and transfer.

7.4 Generated Data

From the 200,000 generated systems, important properties can be studied in order to insure correctness when compared to both analytical results and other MCMC literature.

The final data is comprised of 200,000 Ising systems with uniformly distributed temperatures $T = [1, 4]$. Each system was run for 5×10^8 steps resulting in a total of 10^{14} steps across all systems. This took 1850 core hours, a total of 5000 core hours were used for generating various datasets before reaching a decision on the final system configurations.

Of primary interest is the magnetization, calculated as in equation 2.9, this can be visualized against temperature. In order to calculate and visualize the susceptibility one would require to collect spin and energy values

within the metropolis algorithm, due to the requirement for efficiency this was not done for the core data, a small sample using smaller lattice sizes was used instead.

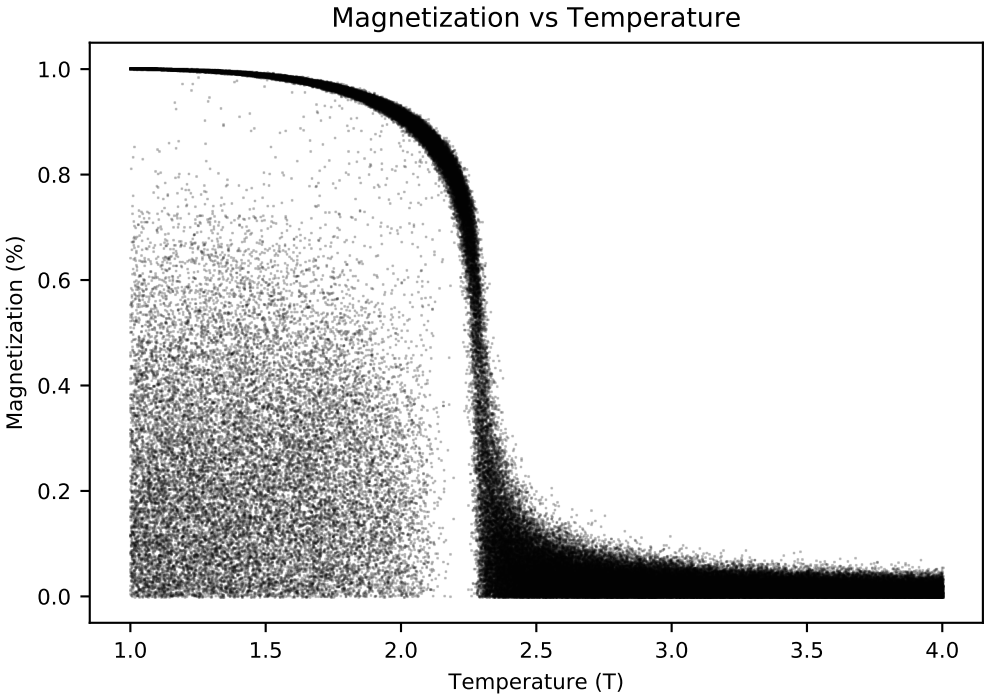


Figure 10: The average magnetization of 200,000 Ising systems of size 128x128 against continuous temperatures $T = [1, 4]$

From Figure 10 one can see several features of interest, for one there is a clear phase transition from an ordered ferromagnetic state to a disordered nonmagnetic state, with a critical temperature at $\approx 2.3\ T$, which is close to the known value. Furthermore one can see a significant amount of systems that do not follow the primary trendline, these are systems that did not reach convergence. The reason for this is due to the existence of striped states, these were not removed from the primary model, however further investigation into the effect of these striped states on the CNN prediction behavior is of interest.

7.4.1 Striped States

During the quenching process using the Metropolis algorithm a convergence issue was observed at low temperatures $T \leq 2.1$ where some systems did not reach the expected micro-state.

This freezing of states results in striped systems, where the system converges to a state where some cluster of spins is aligned into a stripe that can be horizontal, diagonal or vertical.

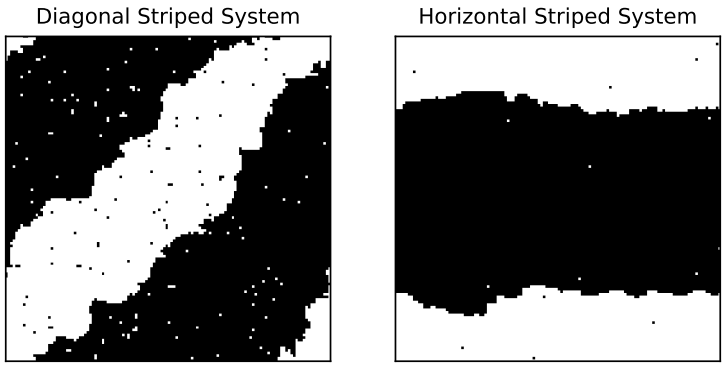


Figure 11: Image of two striped Ising systems of dimension 128x128, left system at $T = 1.596$ and $M = 0.2098$ with right system at $T = 1.122$ and $M = 0.1826$.

Krapivsky, Redner et al. discussed the probability of striped states in the quenching process and found a probability of about 1/3 of a system not reaching the desired state [53]. Notably they also noticed this probability being reduced with increased system sizes, which is something that was also observed in this project.

The striped states were retained during the training of the network, removing them was discussed however due to their existence being considered the norm this was not done. The existence of these striped states can be clearly observed in Figure 10, where a relatively large amount of micro-states do not follow the expected trend.

In Figure 11, two randomly chosen striped micro-states are illustrated, of note is that both of these, as well as the majority of striped micro-states, are not converged. This is due to striped micro-states requiring a very large number of steps to converge, as well as some reaching an infinitely locked state, where they cannot converge due to the nature of single spin-flip MCMC simulations.

7.4.2 Observables

Due to the computational requirement for efficiency, observable calculations outlined in equations 2.12 and 2.13 were not performed on the entire dataset. Instead a smaller dataset was created where these observables were computed for smaller systems for illustration purposes and to insure the correctness of the Metropolis implementation.

In Figure 13, the energy per temperature can be seen following the expected trend, with definitive change in behavior across the Curie temperature $T_c \approx 2.269$. Of note is the distinctive behavioral change across three unique states that can be seen on the lower left tail, where the energy per spin settles at three unique values.

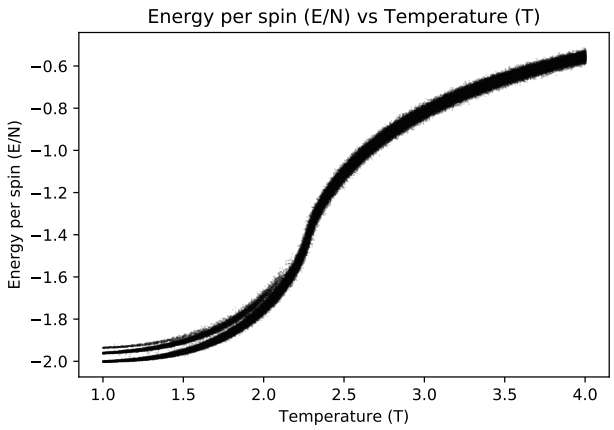


Figure 12: Energy per spin E/n^2 against temperature $T = [1, 4]$ for 200,000 Ising systems of size 128x128.

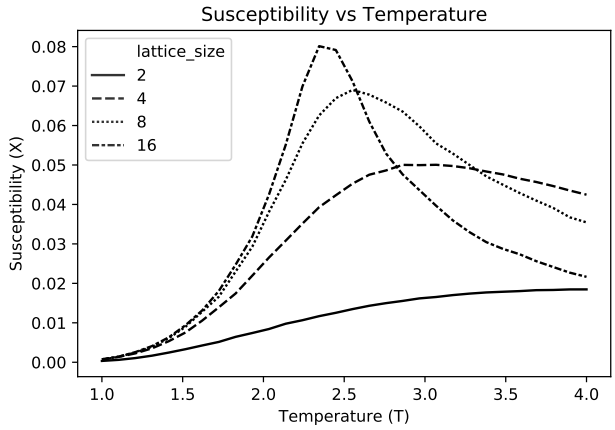


Figure 13: Magnetic susceptibility per spin χ/n^2 vs temperature $T = [1, 4]$ for varying Ising system sizes.

In Figure 12, one can see that at larger lattice sizes the peak magnetic susceptibility grows, and that the peak comes closer to the Curie temperature T_c. As the system size grows $n \to \infty$ the susceptibility will be centered around the Curie temperature $T_c \approx 2.269$ and the susceptibility per spin will reach singularity. This is the second order phase transition that the Ising model with the Hamiltonian described in equation 2.5 exhibits.

At $T < 2$ there are three clear distinct energies per spin that the systems converge to. The reason for this behavior could be due to the previously mentioned striped states, as well as systems that did not reaching convergence. There are three types of systems in the final data, converged ones, horizontal and vertical stripes, and diagonal stripes.

8 Convolutional Network Regression Architecture

Since the implementation of AlexNet in 2012 [24], there has been a steady trend of deeper CNN architectures as compute becomes more widely available. The fundamentals have largely remained unchanged, with repeated convolution layers followed by pooling layers being used prior to fully connected layers.

As discussed previously, architectures generally end with a fully connected layer output based on the number of discrete classes. Regression on the other hand generally uses a single unactivated perceptron.

Modern topologies vary based on task, however popular trends have emerged such as the VGGnet16 [54] architecture, which is comprised of 16 layers, of which 13 are convolutions, and 3 are fully connected. Other more complex topologies have emerged ,such as ResNet50 [55] which reduces number of operations when compared to VGGnet16, but maintains similar performance.

Due to being primarily interested in the performance of regression, in CNNs when applied to problems such as the Ising model, a standard VGGnet inspired topology was chosen for the core architecture.

Deep learning approaches to the Ising model and other statistical systems are not new, but have been focused on basic classification techniques [56] between states, and on non-standard architectures such as stochastic recurrent neural networks [57].

Multiple architectures are implemented and tested in this project, with all following the same overall topology with the differences lying in the use of normalization, down-sampling and learning rates.

8.1 Building the Architecture

The architectures were implemented in TensorFlow [58], using the Keras API [59] in Python. The Keras API allows for a simple approach to building a sequential network topology such as VGGnet. Each network followed the same initial setup and processing as seen in Appendix D.2.1.

Eight separate models were trained, with choices made to investigate some contradictory claims from the modern literature.

8.1.1 Core Layers

Sets of convolution layers with increasing number of filters were chosen, starting from 64 filters and doubling after each pooling layer. All convolution layers have a kernel of dimension 3×3 and a stride of 1 with zero padding. This allows for an arbitrary amount of adjacent convolution layers with no loss in dimensionality.

All pooling layers had a pool size of 2×2 with a stride of 2×2, and all dropout layers had a probability of 25%.

Table 2: Layers in Each Model

Model 1, 2, 7 & 8	**Model 3**	**Model 4**	**Model 5 & 6**
3 × Conv-64	3 × Conv-64	3 × Conv-64	3 × Conv-64
Max Pool		Max Pool	Max Pool
		Dropout	Batch-norm
3 × Conv-128	3 × Conv-128	3 × Conv-128	3 × Conv-128
Max Pool		Max Pool	Max Pool
		Dropout	Batch-norm
3 × Conv-256	3 × Conv-256	3 × Conv-256	3 × Conv-256
Max Pool		Max Pool	Max Pool
		Dropout	Batch-norm
3 × Conv-512	3 × Conv-512	3 × Conv-512	3 × Conv-512
Max Pool		Max Pool	Max Pool
		Dropout	Batch-norm
4 × Conv-512	4 × Conv-512	4 × Conv-512	4 × Conv-512
Max Pool		Max Pool	Max Pool
		Dropout	Batch-norm
4 × Conv-512	4 × Conv-512	4 × Conv-512	4 × Conv-512
Max Pool		Max Pool	Max Pool
		Dropout	
FC-4096	FC-4096	FC-4096	FC-4096
Dropout	Dropout	Dropout	Dropout
FC-4096	FC-4096	FC-4096	FC-4096
Dropout	Dropout	Dropout	Dropout

Core layers in each model, with no hyper-parameters or activation functions included. Conv-xx refers to a 2 dimensional convolution, where xx is the number of filters. FC-yy refers to fully connected layers, where yy is the number of neurons. Model 4 has convolutions with stride 2 at the end of every set.

Investigation into the use of convolutional layers with a stride of 2 with no pooling layer is of interest, as some modern architectures [55] no longer use pooling for down-sampling but instead use purely convolution. However, for the primary model, max pooling is used after each set of convolutions, reducing the dimensionality by a factor of four, or two in each spatial dimension. The dimension of the Ising systems is reduced from 128 to 2 by the use of 6 pooling layers.

The final layers are comprised of 2 fully connected layers of 4096 neurons, again based on the original VGGnet paper, with 2 dropout layers each having probability 25%. The final output is a single unactivated neuron, allowing for a continuous output.

Table 2 gives an overview of these layers compared, however does not include full details. For example models 5 & 6 vary in order of batch normalization, and model 3 has convolutions of stride 2 as the final layer in each set.

The simplified architecture is illustrated in Figure 14. Of note is that model 3 does not make use of the pooling layers.

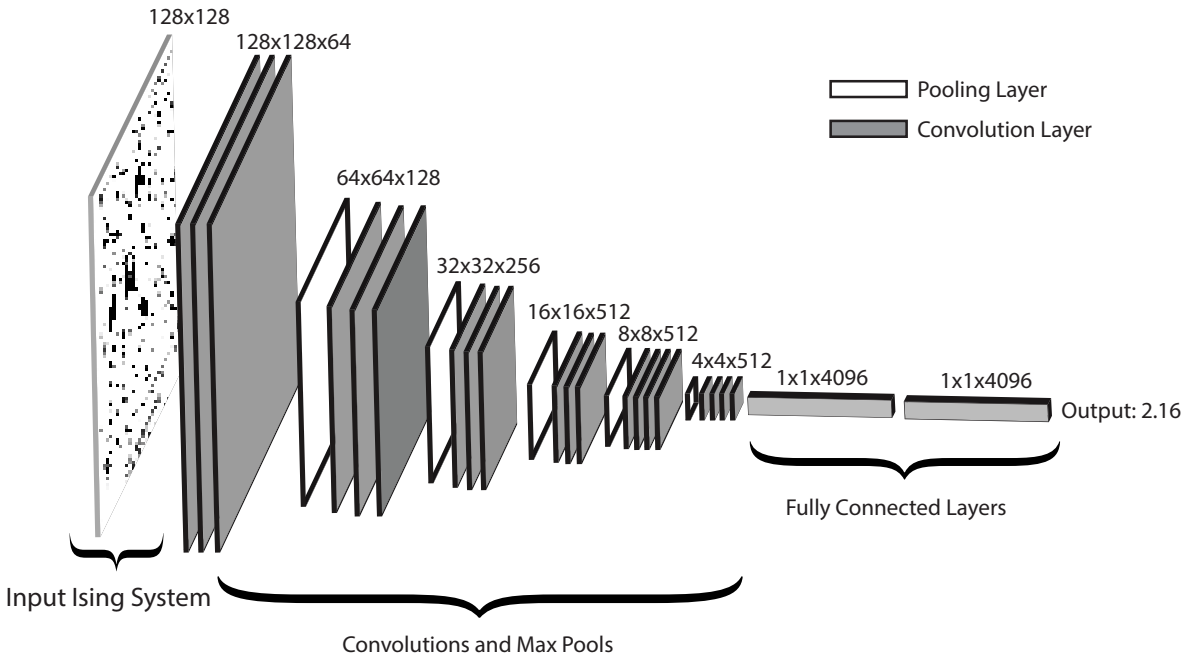


Figure 14: Final convolutional network architecture illustrated, dropout layers are not included. Dimensions of each layer written above, with input Ising system visualized and network output. Final pooling layer prior to fully connected is not visualized.

8.1.2 Different Architectures Investigated

Of interest is the investigation of several questions posed in the recent literature; the effects of batch size on convergence rate and generalization, the differences in various regularization methods, the mode of down-sampling and the benefit of more complex activation functions.

In depth comparisons of the impact of these variations in architecture are difficult, primarily due to the amount of computational power required to train the models as well as the element of randomness involved.

Two dozen models were implemented and tested, with eight interesting architectures being chosen to answer the questions posed above. All models follow the same core architecture, with the only deviations outlined here,

Model 1: Batch size 32 to investigate the effect of using small batch sizes.

Model 2: Batch size 128 for the core standard model to compare against.

Model 3: Convolution layers where the final layer has a stride of 2, for down-sampling, meaning no pooling is required.

Model 4: Dropout after every pooling layer.

Model 5: Batch-normalization before activation after every convolution preceding a pooling layer.

Model 6: Batch-normalization after activation after every convolution preceding a pooling layer.

Model 7: Leaky ReLU in order to investigate more complex activation functions when compared to ReLU.

Model 8: Batch size 384, which is the maximum that fits in memory, to investigate the effect of large batch sizes.

where the model numbers follow the order of implementation.

Four sets of comparisons are of interest, to investigate four contradictions and claims in the literature.

Firstly, the effect of batch size on convergence rate and performance using the Adam optimizer is of interest. Thus models 1, 2 and 8 are compared.

Secondly, models 2 and 3 are used to investigate the mode of down-sampling, with the question of having the final convolution in each set having a stride of 2, instead of a pooling layer.

Thirdly, models 4, 5 and 6 are compared to answer the question of the impact of different normalization methods, with the core investigation lying in dropout layers versus batch-normalization.

Lastly, models 2 and 7 are used to test the performance of Leaky ReLU activation against ReLU.

8.2 Training

A 90/7.5/2.5 train, validation, test split was chosen, meaning out of the 200,000 Ising systems, 180,000 were for training, 15,000 for validation and 5,000 for testing. A more aggressive split could have been used but was deemed not necessary and non-standard.

A Kolmogorov-Smirnov two-sample test [60, pages 318–344] was performed to insure that the training, test and validation set came from the same continuous distribution, this can be found in Appendix A.4.4.

The core model was trained on a mini-batch size of 128 for 40 epochs on a Tesla V100 GPU. The batch size was chosen due to memory constraints, as well as 128 falling within the normal range for this type of problem. This results in roughly $\approx 1,406$ iterations per epoch, and $\approx 40,000$ iterations in total.

The core architecture has 770 million floating point operations per image passed through, in order to update over 50 million parameters, primarily from the fully connected layers in output. For a calculation of memory useage and computational constraints see Appendix C.2.

Due to the complexity of the architecture limited computational resources are available, with only the most modern and expensive GPUs having enough memory to store the number of weights of the model for a reasonable batch size. The Tesla V100 GPU available during this project offers 16GB of memory [61] which allows for batch sizes of ≈ 128 for batch normalization models, and up to ≈ 384 for the core architecture.

8.2.1 Hyper-parameters

Hyper-parameters were initially set to the default recommendations in the original literature, and later tuned to reach a higher performance. The primary difference from the default recommendations is that of the global learning rate set to $\gamma = 0.0002$, due to tests with $\gamma = 0.001$ showing a failure to converge.

Adaptive global step-sizes were used, with a simple reduction algorithm being applied when the loss function did not improve for more than 3 epochs, and an early return if it did not improve for more than 10 epochs.

Algorithm 6: Global Learning Rate Reduction

```
Initialize hyper-parameters
    Set γ = 0.0002
    Set β1 = 0.9
    Set β2 = 0.999
    Set ϵ = 10^−7
While L_{n+1} − 5 × 10^−5 ≤ min({L_i}_{i=n−10}^{n})
    if γ > 10^−7 and L_{n+1} − 10^−4 ≤ min({L_i}_{i=n−2}^{n})
        if γ/4 > 10^−4
            γ ← γ/4
        else
            γ ← γ − 10^−4
```

This learning rate reduction is not reported in the primary document but a full training outline of each model is available in Appendix C.3.1 where the learning rate decreases can be observed.

8.2.2 Loss Rate

The rate of convergence is often measured in number of epochs, this can be disingenuous due to epochs between various models requiring differing amounts of time to complete. For the models used in this project, each epoch takes between 5 and 7 minutes on a NVIDIA Tesla V100 GPU [61].

The loss rate refers to the difference in value of the loss function outlined in equation 5.7 over several epochs. As the model is trained, assuming it is converging to a reasonable local minimum, the loss $\mathcal{L}_n(y, x, \theta)$ will continuously decrease. The speed at which this happens is of interest, with faster convergences requiring less computational power, but not necessarily being indicative to the models ability to generalize. Batch normalization had the largest effect on epoch time, however due to the limited amount of runs a definitive comparison could not be performed.

On average models required 20 epochs to converge, with the only significant outlier being model 4, which used dropout regularization.

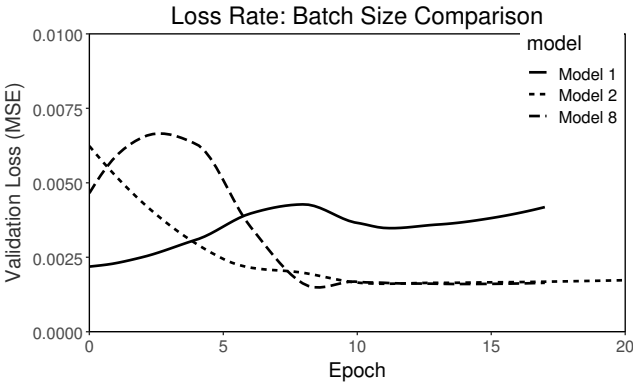


Figure 15: Model 1 vs 2 vs 8. Comparing the effect of batch size 32, 128 and 384 on model performance.

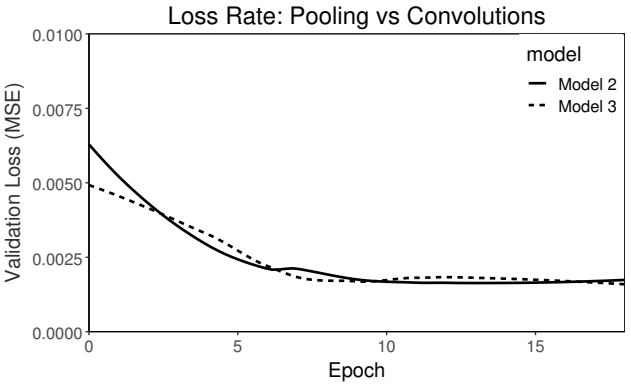


Figure 16: Validation loss of Model 2 vs 3. investigating the difference in performance of max pooling vs convolutions of stride 2.

From Figure 15 a significant result is that small batch sizes performed significantly worse, with a minimum batch of 128 being required before proper convergence was achieved. Batch size did not improve the model convergence rate above 128.

The use of down-sampling also did not result in any significant change in convergence time, with the only notable difference being the fully convolutional model exhibiting a more stable loss rate as seen in Figure 16.

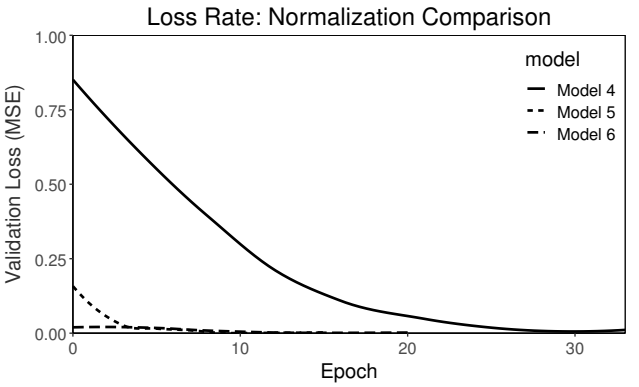


Figure 17: Model 4 vs 5 vs 6. Comparison of dropout against batch-normalization used before and after activation.

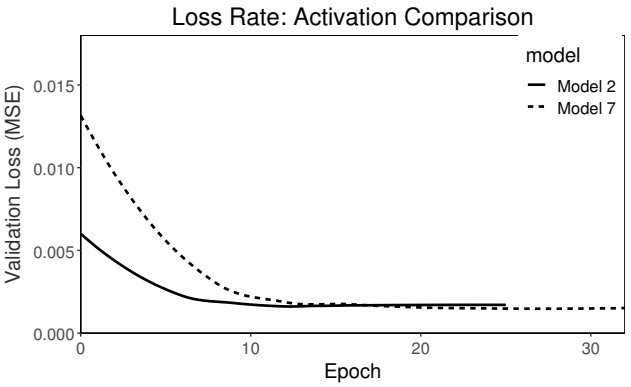


Figure 18: Model 2 vs 7. Comparison of ReLU activation to Leaky ReLU.

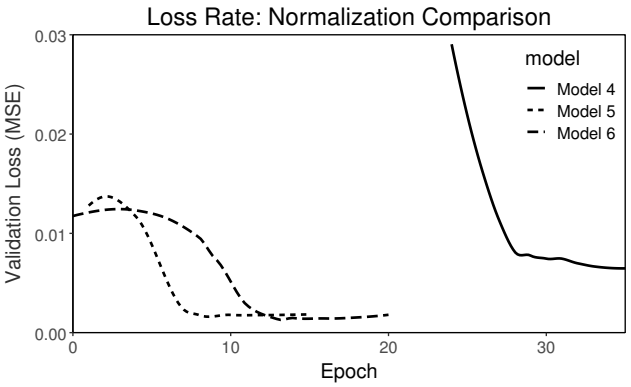


Figure 19: Model 4 vs 5 vs 6 zoomed in. Comparison of dropout against batch-normalization used before and after activation.

The comparison of normalization loss rate is difficult, with the dropout model (model 4) having been trained on a multi-GPU setup due to memory constraints.

This could be culpable for the major difference in convergence rate, with model 4 requiring twice the number of epochs.

Also due to the major differences in starting value Figure 17, it is hard to observe the learning rates of models 5 and 6.

Figure 19 offers a cropped view of the loss rate, showing an interesting loss rate resembling a Gaussian distribution for models 5 and 6.

Longer convergence rates are expected when using regularization methods, with the models attempting to produce a generalized model. Of interest is that model 5 with the batch-normalization applied before activation converged significantly faster.

Finally Figure 18 shows a clear difference in activation function behavior, with Leaky ReLU resulting in a slower but smoother convergence overall.

9 Results

Evaluation of modern architectures is difficult to perform, often requiring the training of other architectures for comparison. This can prove difficult as hyper-parameters and differences in normalization effects on the performance can be unique to the input data. The aim of this project is to offer a more generalized understanding of the performance of the deep convolutional neural network architecture to modelling continuous behavior in statistical systems.

9.1 Model Comparison

Full information on the epochs and their associated training loss, validation loss and the learning rate is available in Appendix C.3.1.

Table 3: Overview of the Trained Architectures

Metrics	Model 1	Model 2	Model 3	Model 4	Model 5	Model 6	Model 7	Model 8
Total Epochs	18	26	19	42	16	21	35	18
Converged Epoch	18	15	19	42	16	21	24	14
Training Time (Minutes)	95	130	110	210	80	100	190	90
Training Loss (MSE) $\times 10^3$	2.4	3.1	2.9	3.0	3.7	4.1	2.3	5.1
Validation Loss (MSE) $\times 10^3$	2.1	1.6	1.6	5.3	1.6	1.6	1.5	1.6
Test Error (%)	2.07	1.31	1.28	2.90	1.30	1.27	1.22	1.27
Test Error (MSE) $\times 10^3$	2.65	1.63	1.56	5.28	1.67	1.57	1.5	1.68
Test Error (MAE) $\times 10^2$	4.26	3.00	2.90	5.59	3.02	2.95	2.83	2.98
Test Error (MBE) $\times 10^2$	-2.28	-0.80	-0.07	-1.54	-0.26	0.34	-0.076	-0.04
Test Error (%) $2 \leq T \leq 3$	1.50	0.81	0.73	1.56	0.77	0.81	0.72	0.78

Overview of the eight models training time and performance on the training, validation and test set. Full outlines of the testing methods and formulae are available in Appendix C.4.1.

The formulae and definitions for the evaluation metrics mentioned in Table 3 can be found in Appendix C.4.1.

From Table 3 it is evident that almost all models achieved an accuracy close to 98% over the entire test set. Models 1 and 4 performed significantly worse than the rest, having test errors approximately twice the magnitude of the rest.

The total number of epochs was dependent on the adaptive step size from Algorithm 6, which has early stopping capabilities when no significant decrease in validation loss is detected over several epochs. However some models reached a stable validation loss earlier than the early stopping algorithm detected, and did stop due to minute changes in validation loss, specifically models 2 and 7.

From Table 3 the error rate for temperatures close to the critical region were also reported, where the models performed significantly more accurately, some reaching above 99.2% prediction performance, meaning a temperature error difference of ≈ 0.015 T. This error rate is close to the maximum performance that can realistically be expected from these types of models, recalling that only 3 decimal places were stored in the temperatures, meaning the primary avenue of improvement is expanding the temperature range at which the models perform at above 99% performance.

9.2 Predictive Performance

Using the epoch of each model with the lowest validation loss, predictions were performed on the test set. Recall that the test set has not been used in any way prior to this, making it equivalent to new data, hence testing the performance of the models ability to generalize to the population.

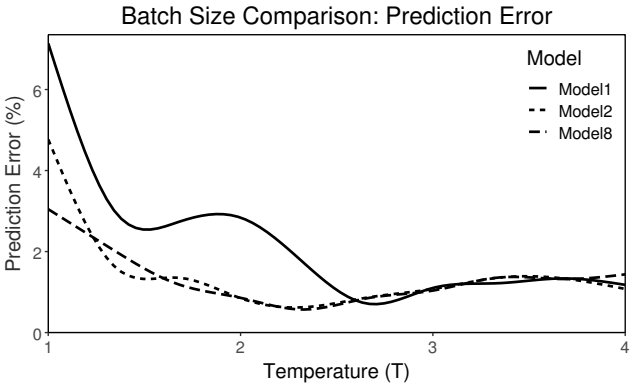


Figure 20: Model 1 vs 2 vs 8. Comparing the effect of batch size 32, 128 and 384 on model performance. Converged epoch refers to the epoch where the model achieved close to peak performance. Training time is an estimate based on epochs and epoch per minute of the model.

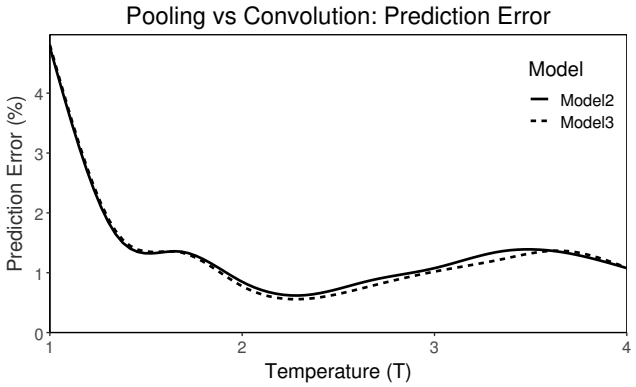


Figure 21: Validation loss of model 2 vs 3, investigating the difference in performance of max pooling vs convolutions of stride 2.

Although smaller batch sizes are in general recommended for generalization [40], from Figure 20 there is a clear trend of lower errors at $T < 2.5$ as the batch size increases from 32 to 128. The error difference between a batch size of 128 and 384 is rather small, although model 8 has lower error at low temperature states. This is supported by Smith et al., where increased batch size was an improvement over decaying the learning rate, when using the Adam optimizer [39]. Of note is that the behavior of the error rate at different batch sizes, insinuates that low temperature states benefit from the increase more. This could be due to the striped states, or due to the fact more distinctive micro-states exist at low temperatures and the increased batch-size allows for the development of a more complex model.

The method of down-sampling seems to have no obvious impact on the predictive performance of the model, however model 3 required less time to train and had a smoother convergence as seen in Figure 16. This incidates that smaller models using purely a convolution approach to down-sampling could be beneficial in the future, and further evaluation using normalization methods would be of interest.

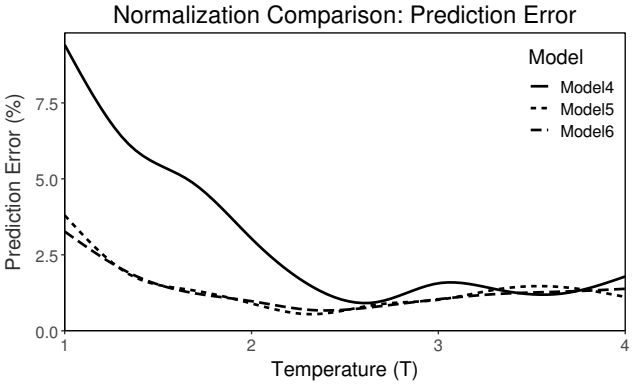


Figure 22: Model 4 vs 5 vs 6. Comparison of dropout against batch-normalization used before and after activation.

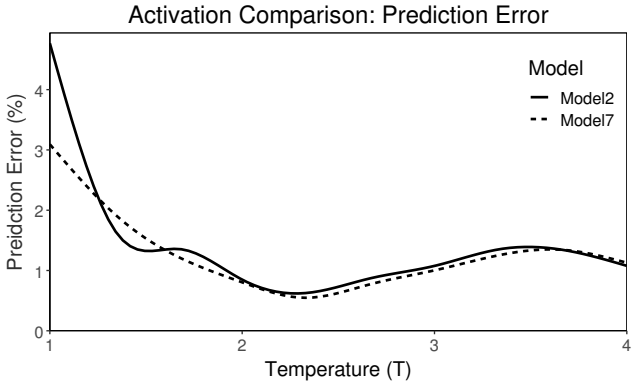


Figure 23: Model 2 vs 7. Comparison of ReLU activation to Leaky ReLU.

The behavior of normalization and regularization methods in deep architectures is difficult to analytically predict. This coupled with the fact that model 4 was trained on a multi-GPU server makes conclusions drawn from the comparison of models 4, 5 and 6 unreliable. Of note is that both models 5 and 6 performed identically, with the only difference being evident during training with model 6 reaching convergence at a faster rate.

Batch normalization being used before or after activation of neurons is a debated topic, and the results do indicate that batch normalization has a greater effect before activation, however in this projects case that effect was negative during training time. Both models did not offer a significant improvement over the other

architectures in predictive ability, although this could be due to small batch sizes [62]. However model 5 converged at a faster rate than most other models.

Finally the discussion of using more complex optimizers for training also results in very similar performance, with the Leaky ReLU model performing smoother and more consistently, but not with much greater accuracy. Leaky ReLU begun convergence at a notably slower rate, as can be seen in Figure 18, however it reached similar validation loss as the ReLU model a few epochs later.

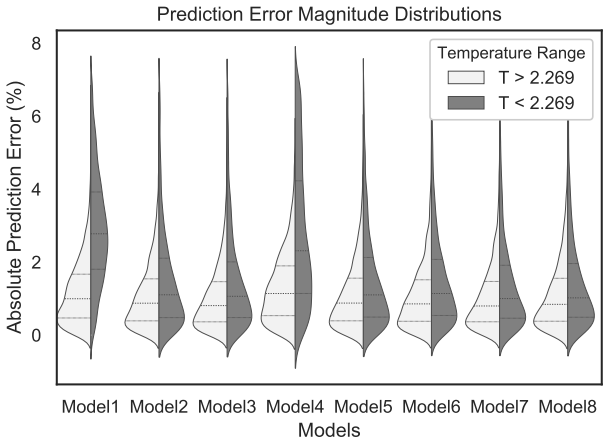


Figure 24: Distribution of prediction errors on the test set at temperatures below and above T_c visualized with violin plots. All errors above 7% were removed, a full plot with full data is available in Appendix C.4.2.

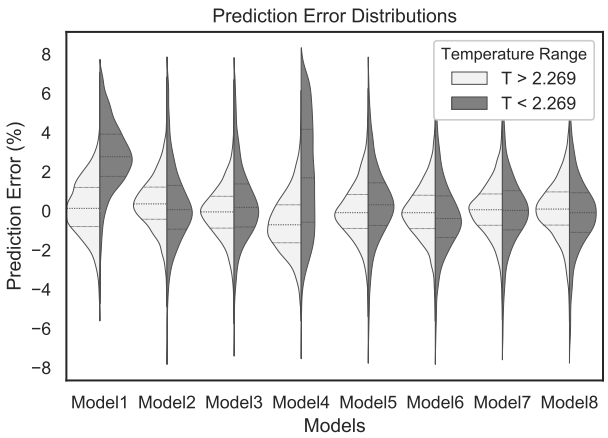


Figure 25: Distribution of prediction errors on the test set at temperatures below and above T_c visualized with violin plots. All errors above 7% were removed, a full plot with full data is available in Appendix C.4.2.

Investigating the distributions of the errors further is of interest, with Figure 24 showing clear differences in distribution across the critical temperature region. The error distribution above T_c has a distribution resembles that of a beta distribution with $\alpha = 2$, $\beta = 5$ with a heavy tail. Below the critical region the distributions experience such heavy tails that they approach an almost uniform appearance.

Once more the impact of striped states could be a culprit in this behavior, however the above critical region distribution can be explained by the exponential drop in magnetization experienced by finite systems, thus creating an impossible task of attempting to distinguish behind high temperature disordered states.

The model bias estimation (MBE) from Table 3 is also of interest, indicating whether errors are primarily positive or negative, meaning that its an overview of whether the model is under-estimating or over-estimating temperatures during prediction.

As can be seen in Table 3, nearly all of the errors were under-estimated, with model 1 having a significant difference, with most of its errors coming from under-estimations.

Further investigation reveals that some of the models have significant differences in error estimation across temperatures, this can also be clearly observed in Figure 25. Temperatures above the critical region are centered around zero, with the temperatures below exhibiting significant variation in behavior.

Model 1 and 4 are the most interesting, with the former showing two Gaussian type distributions with similar standard deviations but shifted means, whereas model 4 has a Gaussian distribution at above critical temperature, and an almost uniform one below. This indicates that model 4 modeled low temperature behavior equally, with no better or lower performance across the range of temperatures. This is an intriguing result as low temperature states exhibit the most varied and complex behaviors.

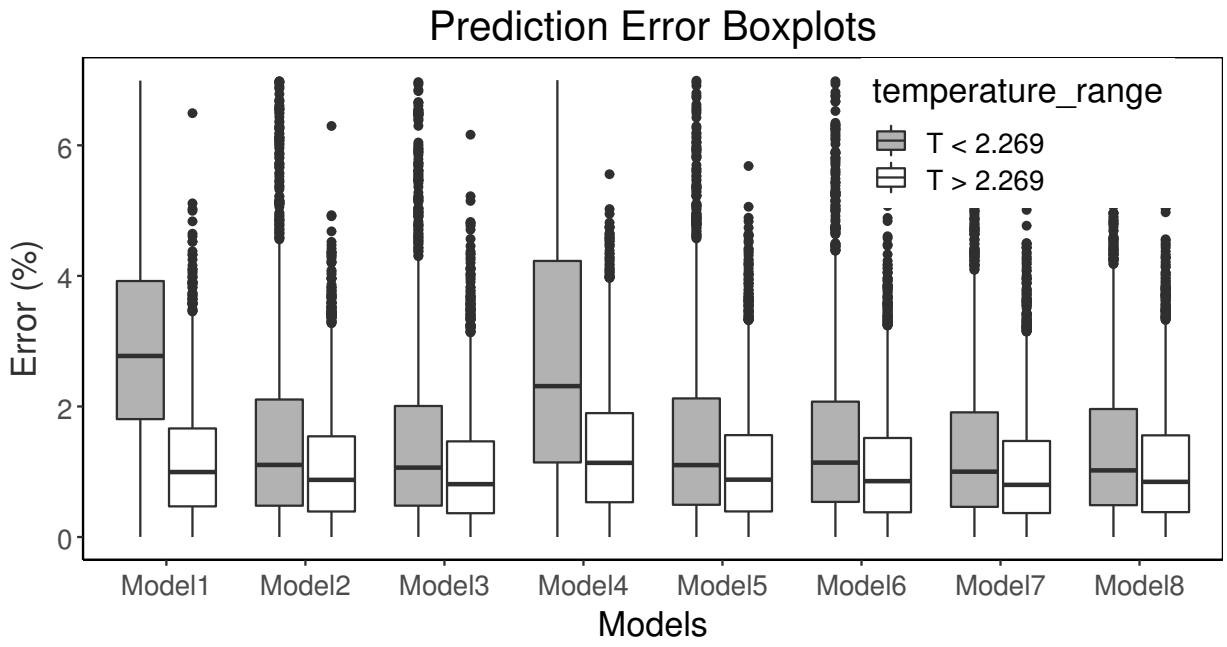


Figure 26: Box plots of prediction errors of models below and above the critical region. Errors above 7% were removed.

The box plots in Figure 26 gives further insight into the difference in error distribution below and above the critical region. Models 1 and 4 exhibiting the most major differences in error distribution. With model 1 having an interquartile ranges that are fully separate, with the bottom 25% of errors at $T < T_c$ being larger than the top 25% at $T > T_c$.

The other models do not feature such a divide, however all models have a larger mean and larger range of errors at $T < T_c$ than at $T > T_c$, re-asserting the conclusion that the models struggled with modelling low temperature states.

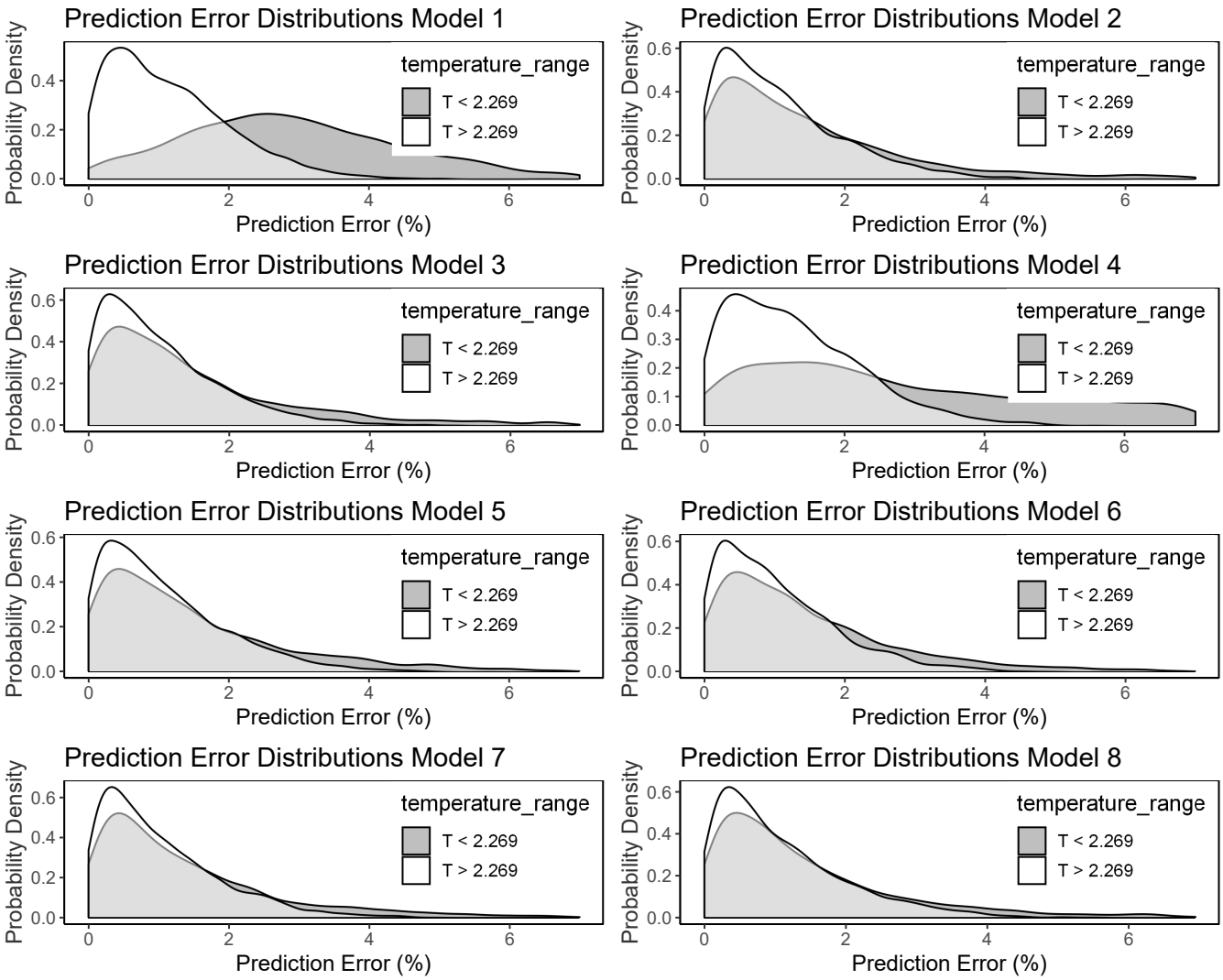


Figure 27: Density plot of error predictions of models, below and above the critical temperature. Errors above 7% were removed.

Investigating the distributions using a smoothed kernel probability distribution estimate as shown in Figure 27 gives more information on the behavior seen in Figure 24. Model 1 and 4 again are the only models to show such a major shift in mean, with model 4 being the only model to exhibit an exceptionally large tail that almost resembles a uniform distribution.

All models other than 1 and 4 show peaks in their probability density at around 0.5%, with heavier tails at $T < T_c$. Several randomly picked systems are illustrated in Figure 30 in Appendix C.4.2.

9.2.1 Low Temperature Behavior

Low temperature states exhibit great variation, with complex behaviors and major differences in small temperature differences when compared to high temperature models. The under-predicted and lower performance of almost all models at low temperatures warrants further investigation.

Removal or control of striped states should be performed in future work, with the current impact to the models prediction performance being unknown. Striped states could theoretically be responsible for the dramatic difference in performance for models below and above the critical temperature.

Figure 32 visualizes the gradients from two output values of the parameters of the first pooling layer, further visualization and comparisons between models could help uncover what different models pay attention to.

10 Conclusion

Deep learning approaches in statistical physics have not yet reached wide adoption, in part due to their recent conception. The lack of regression based approaches using convolutional network architectures is also problematic, since many statistical physics problems involve the extraction and modelling of continuous behaviours.

This project shows that standard deep learning architectures on statistical systems can extract continuous state information with high accuracy.

The performance across models implies that further research into the effect of regularization methods on deep architectures for regression is required.

Some clear results emerge from this project regarding the behavior of deep learning convolutional architecture topologies and hyper-parameters for regression based task.

Firstly, small batch sizes struggle to model fast moving behavior, this is evident with model 1 having a clear separation in error rate as seen in Figure 31, where it was unable to model the fast moving behavior due to the small batch size.

Secondly, the use of convolutional layers for down-sampling does not seem to negatively impact training or model performance.

Thirdly, investigations into regularization methods is needed to further test that the literature regarding classification tasks is relevant and applicable to regression based tasks.

Lastly, the use of Leaky ReLU and other more complex activation functions has the potential to increase performance at the expense of slightly longer training times.

Overall the results of this project suggest that assumptions based on results from literature regarding classification problems are not guaranteed to behave similarly in regression based tasks.

10.1 Future Work

Time and computational constraints required the simplification of certain work, as well as restricting the potential for investigating different models and problem sets.

10.1.1 Statistical Physics

The Ising model was chosen due to its simplicity as a toy model allowing for investigation of the performance of various architectures. The implementation of more complex models such as magnetic models with long-range interactions and models with more complex external interactions are of interest, as well as the extraction of more complex state information.

One of the primary avenues of interest, is the investigation of using generative adversarial networks (GANs) [63] for generating systems that cannot be solved analytically or efficiently. GANs have been used successfully for generating accurate models of several systems, such as generating solutions to the steady state heat conduction and incompressible fluid flow [64]. The use of these for generating statistical physics models accurately would allow for the generation of large datasets that currently require super-computing clusters.

GAN architectures have been developed for the Ising model [51], however the work was limited to systems from $2 \leq T \leq 2.7$ and was only performed on a small lattice of dimension 32×32. Creating a GAN for larger lattice size as used in this project, with a larger temperature range could provide more interesting data.

Another use of GAN architectures is in super-resolving systems, where a GAN produces an output system with larger dimensionality than the input. Melko et al. has produced an implementation for the 1 and 2 dimensional Ising model which showed promise [65] and was based on prior super resolution techniques developed for image processing [66]. This could also be implemented on systems where large dimensional data is required but computationally difficult to implement.

10.1.2 Numerical Statistical Physics Simulations

The metropolis Python implementation, while relatively efficient, is not fully optimized. A CUDA implementation is of interest, with most papers in the literature having been implemented more than a decade ago showing many orders of magnitude speedup [67]. A modern implementation that can utilize the modern architectures such as the Tesla V100 could be attempted.

10.1.3 Deep Learning Architectures

Deep learning architectures are evolving at a rapid pace, this project implements the relevant core modern architectures and methods discussed in the literature, however implementing more exotic and complex methods could be of interest.

Due to a lack of compute, more network visualizations were not performed, however reproducing methods from Zeiler et al. [68] could uncover what convolutional architectures use to model statistical systems is of interest. Specifically to help understand the features that are of interest for models with better performance, and how regularization methods affect the modelling of these features.

Data augmentation methods are another avenue that is of interest, where various techniques can be compared to having similar amounts of raw data. More discussion on data augmentation approaches can be found in Appendix C.5.

Attempting to implement smaller models such as MobileNet [69] could give insight into the ability of reducing computational requirements for the deployment of statistical physics architectures.

Direct comparisons to classification performance was not performed due to time and computational constraints, this however would assist in solidifying the conclusions drawn in this project, instead of relying on the literature.

Bibliography

[1] Sacha Friedli and Yvan Velenik. *Statistical Mechanics of Lattice Systems: A Concrete Mathematical Introduction*. Cambridge University Press, 2017.

[2] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, Dec 1943.

[3] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, pages 65–386, 1958.

[4] Marvin Minsky and Seymour Papert. *Perceptrons: An Introduction to Computational Geometry*. MIT Press, Cambridge, MA, USA, 1969.

[5] Bradley J. Erickson, Panagiotis Korfiatis, Zeynettin Akkus, and Timothy L. Kline. Machine Learning for Medical Imaging. *RadioGraphics*, 37(2):505–515, 2017.

[6] J B Heaton. Deep Learning in Finance. (February):1–20, 2016.

[7] Author Manuscript. Machine Learning for Detecting Gene-Gene Interactions:. 5(2):77–88, 2011.

[8] Zachary Chase Lipton. The mythos of model interpretability. *CoRR*, abs/1606.03490, 2016.

[9] Junghwan Cho, Kyewook Lee, Ellie Shin, Garry Choy, and Synho Do. Medical image deep learning with hospital PACS dataset. *CoRR*, abs/1511.06348, 2015.

[10] Daniël M. Pelt and James A. Sethian. A mixed-scale dense convolutional neural network for image analysis. *Proceedings of the National Academy of Sciences*, 115(2):254–259, 2018.

[11] K. D. Humbird, J. L. Peterson, and Ryan G. McClarren. Transfer learning to model inertial confinement fusion experiments. *CoRR*, abs/1812.06055, 2018.

[12] Ernst Ising. Beitrag zur Theorie des Ferromagnetismus. *Zeitschrift für Physik*, 31(1):253–258, 1925.

[13] Lars Onsager. Crystal statistics. I. A two-dimensional model with an order-disorder transition. *Physical Review*, 65(3-4):117–149, 1944.

[14] B.M. McCoy and T.T. Wu. *The two-dimensional Ising model*. Harvard University Press, 1973.

[15] Kurt Binder. *Monte Carlo Methods in Statistical Physics*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1986.

[16] Sean Meyn and Richard L. Tweedie. *Markov Chains and Stochastic Stability*. Cambridge University Press, New York, NY, USA, 2nd edition, 2009.

[17] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21(6):1087–1092, 1953.

[18] W.K. Hastings. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika*, 57(1):97–109, 1970.

[19] Chii-Ruey Hwang Arnoldo Frigessi , Patrizia Di Stefano and Shuenn-Jyi She. Convergence Rates of the Gibbs Sampler, the Metropolis Algorithm and Other Single-Site Updating Dynamics. *Journal of the Royal Statistical Society*, 55(1):205–219, 1993.

[20] Ulli Wolff. Comparison between cluster Monte Carlo algorithms in the Ising model. *Physics Letters B*, 228(3):379–382, 1989.

[21] Trevor Hastie, Robert Tibshirani, and Jerome H. Friedman. *The elements of statistical learning: data mining, inference, and prediction, 2nd Edition*. Springer series in statistics. Springer, 2009.

[22] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. The MIT Press, 2016.

[23] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, abs/1609.04747:303–314, 1989.

[24] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. pages 1097–1105, 2012.

[25] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, may 2015.

[26] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In Geoffrey Gordon, David Dunson, and Miroslav Dudík, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pages 315–323, Fort Lauderdale, FL, USA, 11–13 Apr 2011. PMLR.

[27] Yevhen Kuznietsov, Jorg Stuckler, and Bastian Leibe. Semi-supervised deep learning for monocular depth map prediction. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[28] Jorge Nocedal and Stephen J. Wright. *Numerical optimization*. Springer series in operations research and financial engineering. Springer, 2. ed. edition, 2006.

[29] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The Annals of Mathematical Statistics*, 22(3):400–407, 1951.

[30] Yann N. Dauphin, Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, pages 2933–2941, Cambridge, MA, USA, 2014. MIT Press.

[31] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural Networks*, 12(1):145 – 151, 1999.

[32] Yurii Nesterov. A method of solving a convex programming problem with convergence rate O(1/sqr(k)). *Soviet Mathematics Doklady*, 27:372–376, 1983.

[33] Sebastian Ruder. An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747, 2016.

[34] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[35] Matthew D. Zeiler. ADADELTA: an adaptive learning rate method. *CoRR*, abs/1212.5701, 2012.

[36] T. Tieleman and G. Hinton. Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning, 2012.

[37] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[38] Léon Bottou, Frank E. Curtis, and Jorge Nocedal. Optimization methods for large-scale machine learning. *SIAM Review*, 60(2):223–311, 1 2018.

[39] Samuel L. Smith, Pieter-Jan Kindermans, and Quoc V. Le. Don't decay the learning rate, increase the batch size. In *International Conference on Learning Representations*, 2018.

[40] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima. *ICLR*, 2017.

[41] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

[42] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, pages 448–456. JMLR.org, 2015.

[43] Xiang Li, Shuo Chen, Xiaolin Hu, and Jian Yang. Understanding the disharmony between dropout and batch normalization by variance shift. *CoRR*, abs/1801.05134, 2018.

[44] Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. How does batch normalization help optimization? In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 2483–2493. Curran Associates, Inc., 2018.

[45] Samuel F. Dodge and Lina J. Karam. A study and comparison of human and deep learning recognition performance under visual distortions. *CoRR*, abs/1705.02498, 2017.

[46] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin A. Riedmiller. Striving for simplicity: The all convolutional net. In *ICLR (Workshop)*, 2015.

[47] D. C. Cireşan, U. Meier, and J. Schmidhuber. Transfer learning for latin and chinese characters with deep neural networks. pages 1–6, June 2012.

[48] M. E. J. Newman and G. T. Barkema. Monte Carlo methods in statistical physics. page 475, 1999.

[49] Travis E. Oliphant. *Guide to NumPy*. CreateSpace Independent Publishing Platform, USA, 2nd edition, 2015.

[50] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, pages 7:1–7:6, New York, NY, USA, 2015. ACM.

[51] Zhaocheng Liu, Sean P. Rodrigues, and Wenshan Cai. Simulating the Ising Model with a Deep Convolutional Generative Adversarial Network. *arXiv e-prints*, page arXiv:1710.04987, Oct 2017.

[52] Tobias Preis, Peter Virnau, Wolfgang Paul, and Johannes J. Schneider. Gpu accelerated monte carlo simulation of the 2d and 3d ising model. *Journal of Computational Physics*, 228(12):4468 – 4477, 2009.

[53] V. Spirin, P.L. Krapivsky, and S. Redner. Freezing in ising ferromagnets. *Phys. Rev. E*, 65:016119, Dec 2001.

[54] S. Liu and W. Deng. Very deep convolutional neural network based image classification using small training sample size. In *2015 3rd IAPR Asian Conference on Pattern Recognition (ACPR)*, pages 730–734, Nov 2015.

[55] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, June 2016.

[56] Juan Carrasquilla and Roger G. Melko. Machine learning phases of matter. *Nature Physics*, 13(5):431–434, 2017.

[57] Alan Morningstar and Roger G. Melko. Deep Learning the Ising Model Near Criticality. 2017.

[58] Martín Abadi et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[59] François Chollet et al. Keras. https://keras.io, 2015.

[60] John W. Pratt and Jean D. Gibbons. *Kolmogorov-Smirnov Two-Sample Tests*, pages 318–344. Springer New York, New York, NY, 1981.

[61] NVIDIA. Nvidia tesla v100 gpu architecture. https://www.nvidia.com/en-us/data-center/tesla-v100/. Accessed 27/03/2019.

[62] Sergey Ioffe. Batch renormalization: Towards reducing minibatch dependence in batch-normalized models. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 1945–1953. Curran Associates, Inc., 2017.

[63] Ian Goodfellow et al. Generative adversarial nets. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 2672–2680. Curran Associates, Inc., 2014.

[64] Amir Barati Farimani, Joseph Gomes, and Vijay S. Pande. Deep learning the physics of transport phenomena. *CoRR*, abs/1709.02432, 2017.

[65] Stavros Efthymiou, Matthew J. S. Beach, and Roger G. Melko. Super-resolving the ising model with convolutional neural networks. *Phys. Rev. B*, 99:075113, Feb 2019.

[66] Chao Dong, Chen Change Loy, Kaiming He, and Xiaoou Tang. Learning a deep convolutional network for image super-resolution. In David Fleet, Tomas Pajdla, Bernt Schiele, and Tinne Tuytelaars, editors, *Computer Vision – ECCV 2014*, pages 184–199, Cham, 2014. Springer International Publishing.

[67] Martin Weigel. Performance potential for simulating spin models on GPU. *Journal of Computational Physics*, 231(8):3064–3082, 2012.

[68] Matthew D. Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *Computer Vision – ECCV 2014*, pages 818–833, Cham, 2014. Springer International Publishing.

[69] Andrew G. Howard and Menglong Zhu et al. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017.

[70] Liangchen Luo, Yuanhao Xiong, and Yan Liu. Adaptive gradient methods with dynamic bound of learning rate. In *International Conference on Learning Representations*, 2019.

[71] Stanford University. Cs231n convolutional networks. http://cs231n.github.io/convolutional-networks. Accessed 22/03/2019.

[72] Luis Perez and Jason Wang. The effectiveness of data augmentation in image classification using deep learning. *CoRR*, abs/1712.04621, 2017.

[73] J. Shijie, W. Ping, J. Peiyi, and H. Siping. Research on data augmentation for image classification based on convolution neural networks. In *2017 Chinese Automation Congress (CAC)*, pages 4165–4170, Oct 2017.

Glossary

convergence When discussing deep learning models convergence refers to the loss function reaching some steady-state value against the validation set. Ideally at a good local minimum.. 17

core hour Number of cores used for computation multiplied by number of hours computation took to complete. 30

down-sampling Down-sampling refers to the reduction of dimensionality along some dimension. When referring to convolutional and pooling layers this usually means reducing the number of weights and biases within each layer.. 26

generalization In computational statistics, when referring to a trained model, the term generalization means the ability for the model to perform similarly to new data as it does against the training data.. 15

regularization This term refers to methods to reduce over-fitting in models. Usually referring to dropout layers or batch normalization.. 23

Acronyms

AI Artificial Intelligence. 8

API Application Programming Interface: Often provides a higher level way to communicate and utilize lower level constructs. 49

AWS Amazon Web Services: Cloud compute products offered by Amazon.. 49

CNN Convolutional Neural Network: a type of deep learning neural network architecture primarily used for image classification. 25

CPU Central Processing Unit: a generalized computing logic unit.. 29

EC2 Elastic Compute Cloud: An AWS product offering virtual or dedicated servers at an hourly cost.. 49

GPU Graphics Processing Unit: a common hardware component used for specialized processing such as rendering. 29

JIT Just In Time: Refers to just in time compilation of bytecode to machine code. 28

MCMC Markov Chain Monte Carlo: a popular class of algorithms used for sampling from probability distributions. 12

SGD Stochastic Gradient Descent: A common numerical optimization algorithm. 21

STEM Refers to the fields of Science, Technology, Engineering & Mathematics. 8

Appendices

A Software, Hardware, Package information

Discussion of the software and hardware used in this project were deferred to this section in the appendix to not distract from the theory.

Information on programming languages, libraries, and the hardware the code was run on is available here.

A.1 Python Frameworks & Hardware

The project primarily used Python, with a few visualizations being performed in R with the ggplot2 library.

Python was chosen due to being easily understood by readers not familiar with programming, as well as being widely supported and efficient for the tasks required by this project.

A.1.1 MCMC

NumPy is Python's core matrix data-structure and manipulation library, however due to the sequential nature of Metropolis-Hastings algorithm it is still several orders of magnitude slower when compared to equivalent C++ code. The Numba library with JIT compiling appeases this lack of efficiency, by providing speedup to C levels, for most NumPy functionality. The implementation is discussed later with code available. Wolff-cluster and other algorithms were not implemented due to small system sizes being chosen, and CUDA acceleration was deemed unnecessary due to complexity of implementation and time-scale of project, as well as wanting to retain readability.

A.1.2 Machine Learning

Python has a large machine learning ecosystem, with a myriad of libraries including TensorFlow, Caffe, Scikit-learn, PyTorch and Theano. TensorFlow was chosen for this project, due in part to the high level Keras API and the ease of creating sequential neural network topologies. At the time of writing TensorFlow and Keras are experiencing a major update, refer to subsection A.2 for information on the versions used in this project.

Models were designed based on literature using TensorFlow and then run on CUDA using cuDNN library from NVIDIA. Only the final model code is currently included in the appendix.

A.1.3 Cloud Compute

The largest computational requirements were the generation of the 200,000 synthetic Ising systems, as well as the training of the convolutional network model.

The generation of Ising systems was done through the multi-threaded Metropolis-Hastings algorithm implementation, this was a CPU bound program.

Amazon Web Services (AWS) offer a wide variety of cloud services including their EC2 virtual servers.

The availability of cloud compute allowed this project to easily scale to large work-loads and allowed for faster generation of data and training.

The cost of the servers used is often >$50,000 USD and therefore not readily available. The ability to rent these servers for shorter periods of time allowed this project to be completed.

As well as the CPU useage for data generation, GPU was required for training of the convolutional network architecture. The Tesla V100 from NVIDIA is the top performing chip at the time of writing, and allowed for fast iterations and fine-tuning of parameters of the network.

A.2 Software Versions

TensorFlow [58] has recently received a major update, integrating the Keras [59] API. This project was written and implemented just prior to these releases, this may impact the reproducibility and use of the code.

Python Version: Python 3.7.1 [MSC v.1915 64 bit (AMD64)] :: Anaconda, Inc. on win32

Libararies Used: NumPy, Numba, matplotlib, Keras, TensorFlow, tqdm

Library Versions

NumPy: 1.15.4

Tensorflow: 1.12.0

Keras: 2.2.4

Numba: 0.41.0

A.3 Hardware

Hardware and software specifications may impact the reproducibility, specifically the clock-speed and performance of the CPU in regards to the benchmarks of the Metropolis algorithm, as well as the CUDA version and VRAM amount in regards to the convolutional network.

Laptop Specifications: (Dell XPS 15 9560)

CPU: i7-7700HQ 3.4GHz

vCPUs: 8

RAM: 32GB DDR4 2400MHz

GPU: GTX 1050 (Mobile)

VRAM: 4GB

Storage: 1TB 970 Pro NVME SSD

OS: Windows 10 Version 1809

CUDA: Version 9.2

Compute Server Specifications: (AWS EC2 m5d.24xlarge)

CPU: 2x Intel Xeon Platinum 8175 2.5 GHz

vCPUs: 96

RAM: 384GB DDR4

Storage: 4 x 900GB NVMe SSD

AMI: Ubuntu 18.04 AMI with .NET Core 2.1

GPU Server Specifications: (AWS EC2 p3.2xlarge)

CPU: Intel Xeon E5-2686 v4 2.3 GHz

vCPUs: 8

RAM: 61GB DDR4

GPU: NVIDIA Tesla V100

VRAM: 16GB

Storage: 2 x 60GB NVMe SSD

AMI: Deep Learning AMI (Ubuntu) Version 21.0

GPU Server Specifications: (AWS EC2 p3.8xlarge)

CPU: 4x Intel Xeon E5-2686 v4 2.3 GHz

vCPUs: 32

RAM: 244GB DDR4

GPU: 4x NVIDIA Tesla V100

VRAM: 4x 16GB = 64GB (NVLINK)

Storage: 2 x 60GB NVMe SSD

AMI: Deep Learning AMI (Ubuntu) Version 21.2

A.4 Data Storage & Processing

The planning of how to design and store the data-structures associated with the generated Ising data as well as the trained convolutional network architecture are outlined here.

A.4.1 Data Storage Ising Models

For the main project, over 200,000 matrices of dimension 128x128 were generated, with only two different integer values in each cell. We also have 200,000 floating point labels, this is however a small amount in comparison to our spin values, so it is negligible.

This is then $200,000 \times 128 \times 128 = 3,276,800,000$ values to store. The smallest storage medium would be boolean, which would allow us to theoretically store our values in 3×10^9 bits, or ~ 400 megabytes. For ease of computation int8 was however chosen, which requires 8 bits or 1 byte per value, meaning around 3 gigabytes total.

The internal representation uses the custom NPY format developed by NumPy, and allows for high compression up to a ~ 90 % rate, primarily due to the previously mentioned binary values of the systems, as well as the regularity of low temperature systems. High temperature systems cannot be compressed more than the simple boolean representation. This means that 200,000 systems, in the worst case scenario of high temperature, can be stored in ≈ 400 megabytes when compressed.

The files are one 3 dimensional array, with the first 2 dimensions being an Ising system with the subsequent 3rd dimension being used to store the other systems. For compressibility and simplicity these are in reality stored as a 2 dimensional array with each system being flattened. Meaning from dimensionality $200,000 \times 128 \times 128$ to $200,000 \times 16384$.

A.4.2 Data Storage TensorFlow Architectures

The trained algorithms were stored in Hierarchical Data Format (HDF5) which allows for compressed sizes below 1 gigabyte for the deep architecture used in this project. This method of storage allows weights, biases and other information to be stored and loaded into Python easily, allowing for predictions and evaluation to be performed.

Each stored model ended up being roughly 600 megabytes each, including storing the parameters, and the rest of the topological information.

A.4.3 Data Processing

Due to memory constraints, attention was paid to many computations for data handling being in-place. This however impacts the readability of some methods. In-place operations were used during the train, validation and test split phase, as well as to insure data integrity.

The Tesla V100 was able to run most models at ≈ 500 iterations per second, with a 4 GPU setup running at ≈ 1400 iterations per second. Further tests of optimizing this pipeline would be of interest using some of TensorFlow data pipe-lining tools.

A.4.4 Data Distribution

Temperatures were generated uniformly, and although one can trust the implementation of the NumPy Mersenne Twister pseudo-random number generator, statistical tests to check that the distributions have been well sampled amongst the training and test sets were performed.

The plot of the distribution and histogram indicate that the two samples are identically distributed however a valid test is still required.

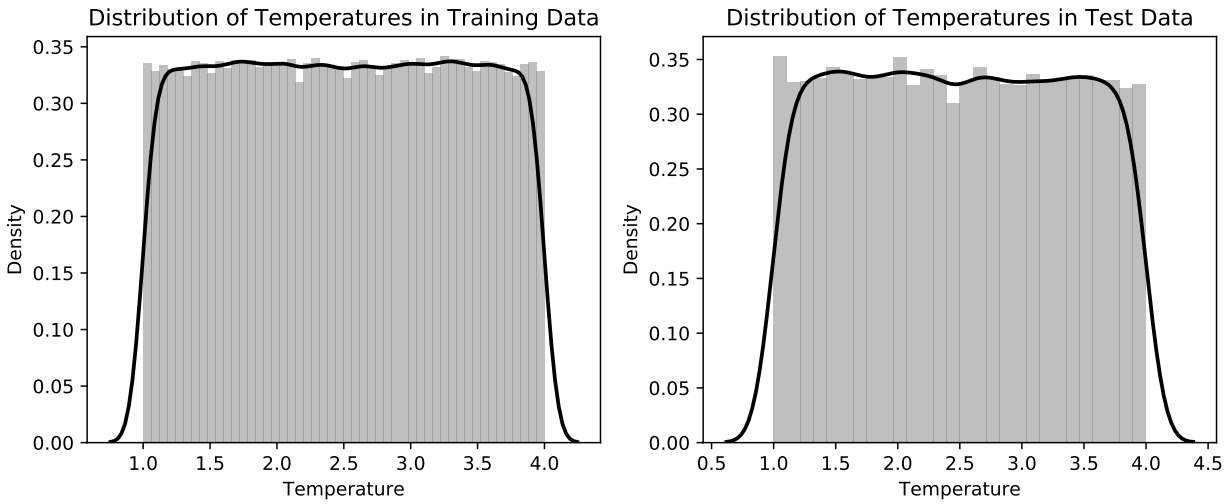


Figure 28: Histogram and density plot of the distribution of the test and training sets of the 200,000 Ising systems.

A simple statistical two-sample test to check for significance that two samples come from different continuous distributions was performed.

The Kolmogorov-Smirnov [60, pages 318–344] two-sample test computes the empirical cumulative distribution function (CDF) of each set and compares them.

The CDF can be computed for some set of data $x_1, x_2, \cdots, x_n$ as

$$\hat{F}_n(x) = \frac{1}{n}\sum_{i=1}^{n} \mathbf{x}_{x_i \leq x} \tag{A.1}$$

where $\mathbf{x}_{x_i \leq x}$ is the indicator.

The data can then be scaled to $0 \leq x \leq 1$ and sorted in ascending order. Then a test statistic D_n is used as

$$D_n = max_i\big(max\big(\big|x_i - \frac{i}{n}\big|, \big|x_i - \frac{i-1}{n}\big|\big)\big) \tag{A.2}$$

The D_n is then used to compare the two samples.

For the 200,000 Ising systems used in this system, the 180,000 training samples are compared to the 20,000 validation and test samples.

The $D_n = 0.00594$ which is a large p-value of $p = 0.547 \gg 0.01$, thus the null hypothesis can be rejected and there is significant evidence that both samples come from the same continuous distribution.

B Metropolis-Hastings Algorithm

Several Metropolis implementations for the Ising model exist, however several are slow and inefficient, being used primarily for exercises rather than large data generation.

The generation of large datasets was integral to the success of this project, and thus an efficient simple implementation was required.

Further work implementing this in lower-level languages such as C, or on GPUs via CUDA is of interest.

B.1 Metropolis-Hastings pseudo-algorithm

Algorithm 7: Metropolis algorithm for Ising model Pseudo-Code

Initialization
    Set $n \in \mathbb{Z}^+$
    Set $nsteps \in \mathbb{Z}^+$
    Set $T \in \mathbb{R}$ where $1 \leq T \leq 4$
    Set $J \in \mathbb{R}_{\geq 0}$
    Set $H \in \mathbb{R}$
    Compute $\beta = 1/T$
    Generate a 2D array $\mathbf{X}$ of size $n \times n$ of uniformly distributed values chosen from $\{\pm 1\}$
For $nsteps$
    Generate 2 random integers $i, j \in \mathbb{Z}^+$ of size $0 \leq i, j \leq n-1$
    Compute $Sn = \mathbf{X}_{i-1,j} + \mathbf{X}_{i+1,j} + \mathbf{X}_{i,j-1} + \mathbf{X}_{i,j+1}$ using periodic boundary conditions
    Compute $\Delta\mathscr{H} = 2\mathbf{X}_{i,j}(H + J \times Sn)$
    Generate a uniformly distributed random floating point number $r \in \mathbb{R}$ where $0 \leq r \leq 1$
    Set $\mathbf{X}_{i,j} = \begin{cases} -\mathbf{X}_{i,j}, & \text{if } r < e^{-\beta\Delta\mathscr{H}} \\ \mathbf{X}_{i,j}, & \text{otherwise} \end{cases}$
return $\mathbf{X}$

B.2 Metropolis-Hastings Core Implementation

```python
import numpy as np
from numba import jit

def mcmc_ising(n=128,
               nsteps=500_000_000,
               T=1,
               J=1,
               H=0):
    """
    This function performs Metropolis MCMC algorithm on a 2D Ising lattice of
    size nxn using numba to JIT compile. It can do around 25,000,000+ iters/second.
    It will return the 2D numpy array as well as the steps and the temperature used.

    :param n: The length/width of the Ising system (numpy array square)
    :param nsteps: The number of steps to run algorithm for
    :param T: The temperature 1/kT for the metropolis switch probability
    :param J: The constant applied to the Hamiltonian
    :param H: External magnetic Field
    :return: 3 returns, a 2D numpy array, the number of steps and the temperature.
    """

    # initialize our lattice
    ising_lattice = np.random.choice([1, -1], size=(n, n))

    # From here on down we are in numba jit
    @jit(nopython=True)
    def ising_calculation(lattice):
        # Pre-calculate the temperature
        pre_temp = 1 / T

        for step in range(nsteps):

            # Randomly choose i j to index into array
            i = np.random.randint(n)
```

```
            j = np.random.randint(n)

            # Boundary conditions and neighbours
            Sn = lattice[(i - 1) % n, j] + lattice[(i + 1) % n, j] + \
                 lattice[i, (j - 1) % n] + lattice[i, (j + 1) % n]

            # Calculating local change in energy
            dE = 2 * lattice[i, j] * (H + J * Sn)

            # Metropolis check
            if np.random.random() < np.exp(-dE * pre_temp):
                lattice[i, j] = -lattice[i, j]

        return lattice

    return ising_calculation(ising_lattice), nsteps, T
```

Albeit being relatively performant, there are still several pieces that could be optimized.

For one the NumPy *randint* random integer generator is unoptimized for single integer generation. Generating larger numbers of integers prior to looping might be beneficial, however the raw storage cost of storing 500 million int8 integers would be 500 megabytes, with our 96 core system requiring at the very least 48 gigabytes of RAM simply for storing these random numbers.

It would also be more efficient to have a greater understanding of convergence rates, allowing us to put an early stop. Currently the algorithm runs for the required number of steps before breaking, an early break when a system is believed to be converged or an adaptive number of steps based on temperature would decrease the total compute time significantly.

A CUDA implementation was briefly attempted, however this proved difficult and due to lack of access to GPUs was deemed unnecessary.

B.3 Multi-core Time Calculation

If we assume 100% scale-ability across g cores, with N being the number of systems we want, τ being the number of steps our algorithm can perform per second, c being the number of steps required for convergence and t being the time required to generate the systems, where $g, N, \tau, c \in \mathbb{Z}^+$ and $t \in \mathbb{R}^+$.

$$t = \frac{Nc}{g\tau} \tag{B.1}$$

This can be simply implemented in Python:

```
from datetime import timedelta

def time_req(cores, n_sys):
        step_rate = 2E7
        step_req = 2E8
        time = (n_sys * step_req) / (cores * step_rate)


        print(f"""Time required to produce {n_sys} Ising systems
        on {cores} cores is {timedelta(seconds=time)}""")


core_systems = [1, 8, 24, 96]

for core in core_systems:
        time_req(core, 100000)
```

```
Time required to produce 100000 Ising systems on 1 cores is 11 days, 13:46:40
Time required to produce 100000 Ising systems on 8 cores is 1 day, 10:43:20
Time required to produce 100000 Ising systems on 24 cores is 11:34:27
Time required to produce 100000 Ising systems on 96 cores is 2:53:37
```

In reality the amount of time is greater due to lower clock speeds on systems with large core counts. The 200,000 systems required $\approx$ 4000 core hours to generate.

B.4 Benchmarks

In order to compute the scale-ability of our multicore code, three separate AWS EC2 instances were rented, a t2.small, t2.2xlarge and a m5d.24xlarge. Each had the same fresh OS setup of ubuntu 18.04 and the same Python packages.

Identical scripts were run on each server, however, the number of systems to compute was reduced on smaller servers for time-saving purposes. We computed 150 systems on t2.small, 1000 on t2.2xlarge and 5000 on m5d.24xlarge.

Each system was 28x28 with 10^8 Metropolis steps, meaning that t2.small computed 1.5×10^{10} steps, t2.2xlarge computed 10^{11} steps and m5d.24xlarge computed 5×10^{11} steps.

The time command was used to get back full run information from console.

Table 4: Full Metropolis performance information

EC2 Instance	# of Cores	# of Systems	Time Information
t2.small	1	150	real 11m34.805s user 11m34.465s sys 0m0.157s
t2.2xlarge	8	1000	real 10m32.474s user 84m14.617s sys 0m1.802s
m5d.24xlarge	96	5000	real 5m44.233s user 540m42.733s sys 1m27.530s

Full performance information from Metropolis run using time command on Ubuntu for each EC2 instance.

We then can compute the steps per second by using formula B.1.

A full performance analysis was not performed, however theoretically the drop in performance may be due to the code not being fully asynchronous, or simply lower clock speeds on the larger systems resulting in fewer steps per second.

C Convolutional Neural Network Topology & Discussion

C.1 Discussion

All discussion out of scope of the primary project regarding the convolutional neural network is included in this section.

C.1.1 Data Processing

TensorFlow and Keras by default set channels to be last, thus the dimensionality of the input data is set to $(N, n, n, 1)$ where N is the number of systems, n is the width and height, and 1 due to the Ising systems not

having any associated depth.

C.1.2 Loss Functions

MSE was used in this project, however this resulted in negative bias in almost all models as seen in Table 3. The use of MAE might deter this behavior.

RMSE could also be used to attempt to reduce the power of outliers influencing the optimizer, however the effect of this was not studied.

The choice of loss function was driven due to its simplicity and prevalence in the literature.

C.1.3 Optimizers

Adam was the only optimizer used in this project, however new optimizers are being developed such as Adabound [70] which was just accepted into ICLR 2019 and may prove to be an improvement over Adam.

In general Adam is an optimizer that has risen in popularity due to its predictable behavior across all problem domains, however the investigation of more specialized optimizers that perform well on regression architectures in statistical physics may be worthwhile.

C.1.4 Batch Normalization pre versus post Activation

A debated topic is the use of batch normalization before or after activation. It has been inferred that Szegedy performs batch normalization after activation in the original paper [43].

This project did indicate as can be seen in Table 3 that batch normalization used prior to activation performs better. The differences however were minor, whether this is due to the nature of regression tasks, or anomalous behavior is unknown.

Batch normalization is not yet fully understood, and its effects on regularizing regression based architectures has not been studied. Its use and comparison to dropout warrants further work.

C.2 Network Topology

Computing the memory and computational requirements of a network can be difficult due to different methods of propagation, data types etc... The Stanford convolutional network course has a simplified calculation for number of parameters [71].

Here CONV2-X refers to a two-dimensional filtered convolutional layer, with X referring to the number of filters. POOL2 refers to two-dimensional pooling and FC to fully connected layers.

The dimension is then shown in brackets, with a memory calculation performed where K is for 10^3. The number of weights are then estimated, and the total memory useage assumes FP32 data-type.

```
INPUT: [128x128x1]        memory:  128*128*1=16K   weights: 0
CONV2-64: [128x128x64]  memory:  128*128*64=1M   weights: (3*3)*64 = 576
CONV2-64: [128x128x64]  memory:  128*128*64=1M   weights: (3*64)*64 = 12,288
CONV2-64: [128x128x64]  memory:  128*128*64=1M   weights: (3*64)*64 = 12,288
POOL2: [64x64x128]  memory:  64*64*64=262k   weights: 0
CONV2-128: [64x64x128]  memory:  64*64*128=524K   weights: (3*64)*128 = 24,576
CONV2-128: [64x64x128]   memory:  64*64*128=524K   weights: (3*128)*128 = 49,152
CONV2-128: [64x64x128]    memory:  64*64*128=524K   weights: (3*128)*128 = 49,152
POOL2: [32x32x256]  memory:  32*32*128=131K   weights: 0
CONV2-256: [32x32x256]  memory:  32*32*256=262K   weights: (3*128)*256 = 98,304
CONV2-256: [32x32x256]  memory:  32*32*256=262K   weights: (3*256)*256 = 196,608
CONV2-256: [32x32x256]  memory:  32*32*256=262K   weights: (3*256)*256 = 196,608
POOL2: [16x16x512]  memory:  16*16*256=65K   weights: 0
```

```
CONV2-512: [16x16x512]  memory:  16*16*512=130K   weights: (3*256)*512 = 393,216
CONV2-512: [16x16x512]  memory:  16*16*512=130K   weights: (3*512)*512 = 786,432
CONV2-512: [16x16x512]  memory:  16*16*512=130K   weights: (3*512)*512 = 786,432
POOL2: [8x8x512]  memory:  8*8*512=32K   weights: 0
CONV2-512: [8x8x512]  memory:  8*8*512=32K   weights: (3*512)*512 = 786,432
CONV2-512: [8x8x512]  memory:  8*8*512=32K   weights: (3*512)*512 = 786,432
CONV2-512: [8x8x512]  memory:  8*8*512=32K   weights: (3*512)*512 = 786,432
CONV2-512: [8x8x512]  memory:  8*8*512=32K   weights: (3*512)*512 = 786,432
POOL2: [4x4x512]  memory:  4*4*512=8K   weights: 0
CONV2-512: [4x4x512]  memory:  4*4*512=8K   weights: (3*512)*512 = 786,432
CONV2-512: [4x4x512]  memory:  4*4*512=8K   weights: (3*512)*512 = 786,432
CONV2-512: [4x4x512]  memory:  4*4*512=8K   weights: (3*512)*512 = 786,432
CONV2-512: [4x4x512]  memory:  4*4*512=8K   weights: (3*512)*512 = 786,432
POOL2: [2x2x512]  memory:  2*2*512=2K  weights: 0
FC: [1x1x4096]  memory:  4096  weights: 2*2*512*4096 = 8,388,608
FC: [1x1x4096]  memory:  4096  weights: 4096*4096 = 16,777,216
FC: [1x1x1]  memory:  1 weights: 4096*1 = 4,096

TOTAL memory: 14.6M * 4 bytes ~= 58MB / image forwards, and around 100MB/image with backwards prop
TOTAL params: 53M parameters

128 Batch Size = 13GB memory consumption
```

C.3 Full Model Data

This section contains large amounts of data including tables and figures relating to the 8 models trained and evaluated.

C.3.1 Training

Table 5: Model 1 Training Information

Model 1			
Epochs	**Loss**	γ	**Validation Loss**
0	0.059	0.0002	0.0026
1	0.0065	0.0002	0.0022
2	0.0053	0.0002	0.0025
3	0.0047	0.0002	0.0021
4	0.0044	0.0002	0.0026
5	0.0043	0.0002	0.0028
6	0.0041	0.0002	0.0043
7	0.0032	5e-05	0.0059
8	0.0031	5e-05	0.0037
9	0.003	5e-05	0.0031
10	0.003	5e-05	0.0039
11	0.003	5e-05	0.0043
12	0.0029	5e-05	0.0027
13	0.0029	5e-05	0.0042
14	0.0028	5e-05	0.0035
15	0.0028	5e-05	0.0042
16	0.0025	1.25e-05	0.0037
17	0.0024	1.25e-05	0.0042

Core model with a small batch size of 32.

Table 6: Model 2 Training Information

Model 2			
Epochs	**Loss**	γ	**Validation Loss**
0	0.1631	0.0002	0.0046
1	0.0079	0.0002	0.0098
2	0.0067	0.0002	0.0019
3	0.0062	0.0002	0.0021
4	0.0054	0.0002	0.0025
5	0.005	0.0002	0.004
6	0.0037	5e-05	0.0016
7	0.0037	5e-05	0.0022
8	0.0037	5e-05	0.0017
9	0.0036	5e-05	0.0018
10	0.0032	1.25e-05	0.0017
11	0.0032	1.25e-05	0.0016
12	0.0032	1.25e-05	0.0017
13	0.0032	1.25e-05	0.0016
14	0.0031	3.13e-06	0.0017
15	0.0031	3.13e-06	0.0017
16	0.0031	3.13e-06	0.0017
17	0.0031	3.13e-06	0.0017
18	0.0031	3.13e-06	0.0017
19	0.0031	3.13e-06	0.0017
20	0.0031	3.13e-06	0.0017
21	0.0031	3.13e-06	0.0017
22	0.0031	3.13e-06	0.0017
23	0.0031	3.13e-06	0.0017
24	0.0031	3.13e-06	0.0017
25	0.0031	3.13e-06	0.0017

Core model with a batch size of 128.

Table 7: Model 3 Training Information

Model 3			
Epochs	**Loss**	γ	**Validation Loss**
0	0.1835	0.0002	0.0039
1	0.0078	0.0002	0.0044
2	0.0068	0.0002	0.0072
3	0.0063	0.0002	0.002
4	0.0056	0.0002	0.0038
5	0.0051	0.0002	0.0029
6	0.0048	0.0002	0.0019
7	0.0035	5e-05	0.0017
8	0.0035	5e-05	0.0017
9	0.0035	5e-05	0.0017
10	0.0034	5e-05	0.0017
11	0.0031	1.25e-05	0.0016
12	0.0031	1.25e-05	0.0026
13	0.0031	1.25e-05	0.0016
14	0.0031	1.25e-05	0.0016
15	0.003	3.125e-06	0.0016
16	0.003	3.125e-06	0.0017
17	0.003	3.125e-06	0.0016
18	0.0029	7.8125e-07	0.0017

Model comprised of purely convolutional layers for down-sampling.

Table 8: Model 5 Training Information

Model 5			
Epochs	**Loss**	γ	**Validation Loss**
0	0.0497	0.0002	0.2372
1	0.019	0.0002	0.0074
2	0.0129	0.0002	0.0204
3	0.0113	0.0002	0.0154
4	0.0075	5e-05	0.0129
5	0.0066	5e-05	0.0533
6	0.0051	1.25e-05	0.0022
7	0.005	1.25e-05	0.0031
8	0.0047	1.25e-05	0.0021
9	0.0044	3.125e-06	0.0019
10	0.0043	3.125e-06	0.0016
11	0.0043	3.125e-06	0.0019
12	0.0042	3.125e-06	0.0018
13	0.0041	7.8125e-07	0.0019
14	0.0041	7.8125e-07	0.0018
15	0.0041	1.953125e-07	0.0018

Batch normalization before activation model, stabilized at around 10 epochs.

Table 9: Model 4 Training Information

Model 4			
Epochs	**Loss**	γ	**Validation Loss**
0	4.1585	0.0002	0.7843
1	0.0361	0.0002	0.7376
2	0.025	0.0002	0.7567
3	0.0202	0.0002	0.81
4	0.0182	0.0002	0.5868
5	0.0165	0.0002	0.5968
6	0.0146	0.0002	0.4838
7	0.0142	0.0002	0.4226
8	0.0123	0.0002	0.4004
9	0.0116	0.0002	0.381
10	0.0113	0.0002	0.3236
11	0.0104	0.0002	0.2313
12	0.01	0.0002	0.1485
13	0.0097	0.0002	0.1812
14	0.0081	0.0002	0.1575
15	0.0082	0.0002	0.1398
16	0.0077	0.0002	0.1316
17	0.0074	0.0002	0.0875
18	0.0064	0.0002	0.0733
19	0.0061	0.0002	0.0697
20	0.0064	0.0002	0.0592
21	0.0057	0.0002	0.0496
22	0.0053	0.0002	0.0347
23	0.0053	0.0002	0.0319
24	0.005	0.0002	0.0265
25	0.005	0.0002	0.0276
26	0.0049	0.0002	0.0131
27	0.0045	0.0002	0.0111
28	0.0045	0.0002	0.0076
29	0.0046	0.0002	0.0086
30	0.0044	0.0002	0.0074
31	0.0045	0.0002	0.007
32	0.0042	0.0002	0.0076
33	0.0043	0.0002	0.0069
34	0.0042	0.0002	0.0053
35	0.0043	0.0002	0.0071
36	0.0039	0.0002	0.0086
37	0.004	0.0002	0.0067
38	0.0033	5e-05	0.0093
39	0.0032	5e-05	0.0105
40	0.0032	5e-05	0.0094
41	0.0031	1.25e-05	0.0103

Model 4 is the dropout layer model. Model 4 is the only model trained on a multi-GPU setup, this was done due to lack of compute being available.

Table 10: Model 6 Training Information

Model 6			
Epochs	**Loss**	γ	**Validation Loss**
0	7.3679	0.001	0.0168
1	0.0236	0.001	0.0067
2	0.0183	0.001	0.0428
3	0.0138	0.001	0.0333
4	0.0086	0.00025	0.0103
5	0.0083	0.00025	0.0026
6	0.0079	0.00025	0.0266
7	0.0073	0.00025	0.0041
8	0.0057	6.25e-05	0.0168
9	0.0056	6.25e-05	0.0059
10	0.005	1.5625e-05	0.0019
11	0.005	1.5625e-05	0.002
12	0.0049	1.5625e-05	0.0021
13	0.0047	3.90625e-06	0.0016
14	0.0046	3.90625e-06	0.0021
15	0.0046	3.90625e-06	0.0017
16	0.0045	9.765625e-07	0.0016
17	0.0045	9.765625e-07	0.0016
18	0.0045	2.4414064e-07	0.0016
19	0.0045	2.4414064e-07	0.0017
20	0.0045	1e-07	0.0017

Core model with batch-normalization after activation.

Table 11: Model 8 Training Information

Model 8			
Epochs	**Loss**	γ	**Validation Loss**
0	0.511	0.0002	0.0041
1	0.0131	0.0002	0.0038
2	0.0112	0.0002	0.0083
3	0.0093	0.0002	0.0097
4	0.0084	0.0002	0.0099
5	0.0061	5e-05	0.0017
6	0.0058	5e-05	0.0035
7	0.0059	5e-05	0.0018
8	0.0057	5e-05	0.002
9	0.0054	1.25e-05	0.0017
10	0.0053	1.25e-05	0.0016
11	0.0054	1.25e-05	0.0017
12	0.0052	3.125e-06	0.0017
13	0.0052	3.125e-06	0.0017
14	0.0052	3.125e-06	0.0016
15	0.0051	3.125e-06	0.0016
16	0.0052	3.125e-06	0.0016
17	0.0051	3.125e-06	0.0016

Core model with a batch size of 384. This was the maximum that fit into memory.

Table 12: Model 7 Training Information

Model 7			
Epochs	**Loss**	γ	**Validation Loss**
0	4.0421	0.0002	0.0176
1	0.0093	0.0002	0.0029
2	0.0085	0.0002	0.0171
3	0.0074	0.0002	0.0071
4	0.007	0.0002	0.0022
5	0.0062	0.0002	0.0078
6	0.006	0.0002	0.0028
7	0.0054	0.0002	0.0024
8	0.0032	5e-05	0.0021
9	0.0035	5e-05	0.0022
10	0.0035	5e-05	0.0034
11	0.0036	5e-05	0.0023
12	0.0028	1.25e-05	0.0017
13	0.0028	1.25e-05	0.0019
14	0.0028	1.25e-05	0.002
15	0.0028	1.25e-05	0.0019
16	0.0025	3.125e-06	0.0016
17	0.0025	3.125e-06	0.0017
18	0.0025	3.125e-06	0.0016
19	0.0025	3.125e-06	0.0017
20	0.0024	7.8125e-07	0.0015
21	0.0024	7.8125e-07	0.0015
22	0.0024	7.8125e-07	0.0016
23	0.0024	1.953125e-07	0.0015
24	0.0023	1.953125e-07	0.0015
25	0.0024	1.953125e-07	0.0015
26	0.0023	1e-07	0.0015
27	0.0024	1e-07	0.0015
28	0.0023	1e-07	0.0015
29	0.0023	1e-07	0.0015
30	0.0023	1e-07	0.0015
31	0.0023	1e-07	0.0015
32	0.0023	1e-07	0.0015
33	0.0023	1e-07	0.0015
34	0.0023	1e-07	0.0015

Core model with Leaky ReLU activation.

C.4 Evaluation

This section contains more information on the evaluation metrics used to analyze the predictions from the trained models.

C.4.1 Evaluation Metrics

Evaluation metrics used in the analysis of regression models varies based on task, however here a varied set of analysis is performed, starting with percentage error computation,

$$\mathcal{L}_{\%error}(y, \hat{y}) = \frac{1}{n}\sum_{i}^{n}\frac{|y_i - \hat{y}_i|^2}{y_i} \tag{C.1}$$

where n is the number of samples, y_i is the ith value and $\hat{y}_i$ is the ith predicted value.

Percentage error doesn't punish similar errors at larger temperatures which is useful.

Then the loss function used during training as well as a common metric for evaluating the performance of predictive models is the mean squared error (MSE).

$$\mathcal{L}_{MSE}(y, \hat{y}) = \frac{1}{n}\sum_{i}^{n}|y - \hat{y}|^2 \tag{C.2}$$

The square root of the mean squared error (RMSE) is also useful at giving a better overview by punishing outliers.

$$\begin{aligned}\mathcal{L}_{RMSE}(y, \hat{y}) &= \sqrt{\mathcal{L}_{MSE}(y, \hat{y})} \\ &= \frac{1}{n}\sum_{i}^{n}\sqrt{|y - \hat{y}|^2}\end{aligned} \tag{C.3}$$

For investigating the actual error values one can use the mean absolute error (MAE) giving a concrete value to the mean error in the relevant temperature units.

$$\mathcal{L}_{MAE}(y, \hat{y}) = \frac{1}{n}\sum_{i}^{n}|y - \hat{y}| \tag{C.4}$$

One downside of MAE is the fact that it doesn't explore the difference in models that may under-predict or over-predict. This is essentially equivalent to exploring the bias, and a metric called mean bias error is the MAE without taking the absolute.

$$\mathcal{L}_{MBE}(y, \hat{y}) = \frac{1}{n}\sum_{i}^{n}y - \hat{y} \tag{C.5}$$

C.4.2 Prediction Error Distribution

Some of the information and plots regarding error distribution are moved to this section.

In Figure 29 it is clear that model 4 at low temperatures made several very large errors. The influence this could have had using the MSE loss function may have impacted the training. Interestingly model 4 is not the worst performing model.

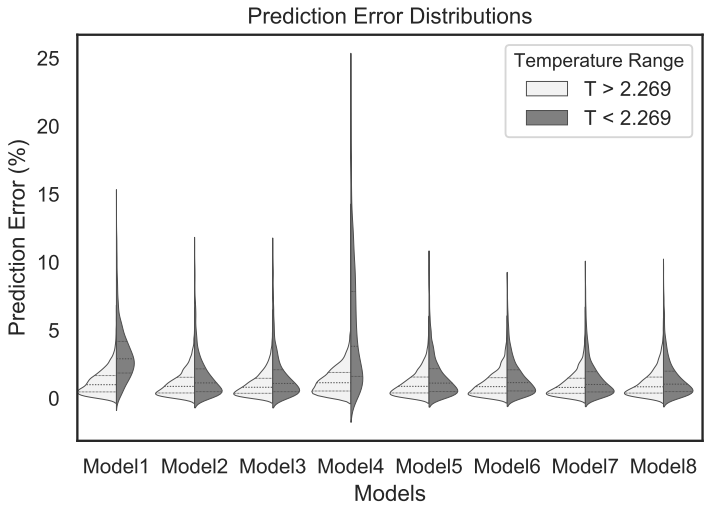
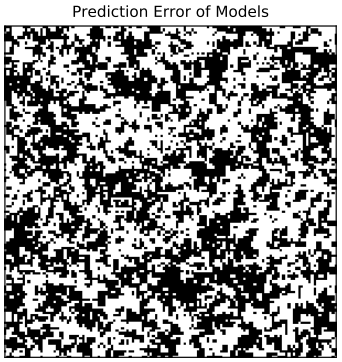


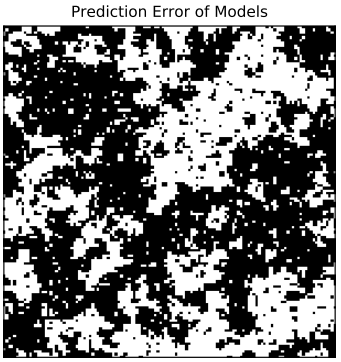
Figure 29: Distribution of Prediction Error Distribution Visualization Full Test Data



True Value: 3.075

Model	Value	Error (%)
1	3.078	-0.083%
2	3.089	-0.439%
3	3.064	0.371%
4	3.051	0.767%
5	3.088	-0.411%
6	3.074	0.042%
7	3.073	0.071%
8	3.076	-0.032%

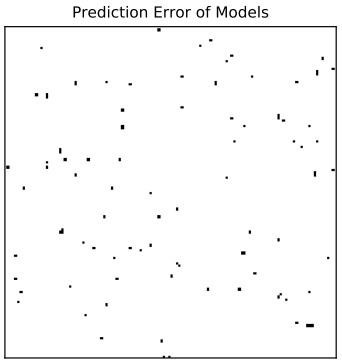
(a) Prediction on system with $T = 3.075$, model 6 and 7 perform best.



True Value: 2.372

Model	Value	Error (%)
1	2.410	-1.582%
2	2.381	-0.377%
3	2.376	-0.156%
4	2.326	1.919%
5	2.361	0.469%
6	2.362	0.426%
7	2.362	0.416%
8	2.366	0.244%

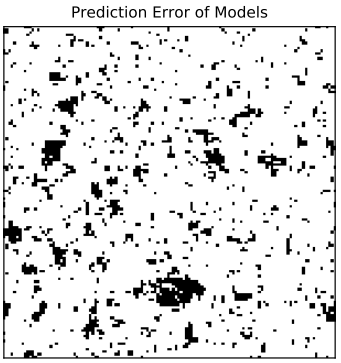
(b) Prediction on system with $T = 2.372$, model 3 and 8 perform best.



True Value: 1.517

Model	Value	Error (%)
1	1.500	1.126%
2	1.464	3.471%
3	1.480	2.463%
4	1.632	-7.575%
5	1.486	2.073%
6	1.470	3.095%
7	1.472	2.99%
8	1.474	2.858%

(c) Prediction on system with $T = 1.517$, model 3 and 5 perform best. Model 4 under-predicts by a large amount.



True Value: 2.229

Model	Value	Error (%)
1	2.261	-1.454%
2	2.199	1.329%
3	2.207	0.967%
4	2.260	-1.393%
5	2.214	0.666%
6	2.202	1.231%
7	2.207	0.974%
8	2.210	0.839%

(d) Prediction on system with $T = 2.229$, model 3 and 5 perform best. Models 1 and 4 under-predict.

Figure 30: Visualization of 4 randomly selected system predictions.

An interesting visualization is the cubemap plot in Figure 31 which gives a closer insight into the error distribution over temperatures.

All models follow the same valley error curve where the central point around the critical temperature has the least errors. However several models such as model 4 have a very large error at low temperature, which was

also observed in Figure 29.

One interesting behavior is that of model 4, which has an almost split central cluster of errors just below the critical temperature.

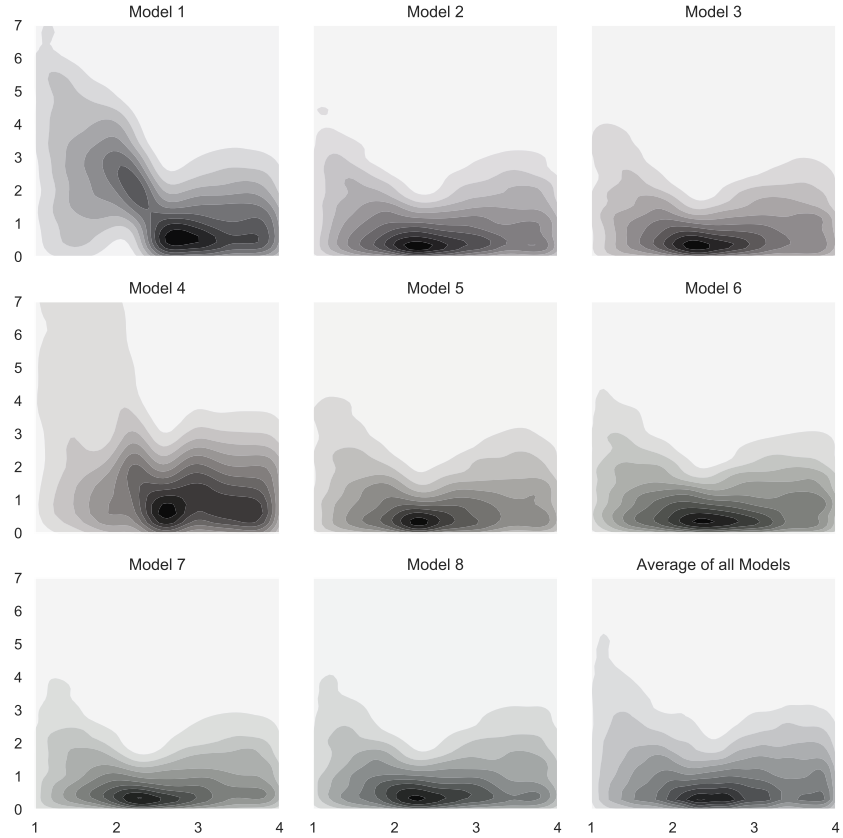


Figure 31: Cubemap plots of the distribution of the prediction error. The x axis refers to the temperature, and the y axis to the percentage error. All errors above 7% were removed.

C.5 Data Augmentation

Although for systems such as the Ising model there are well studied and relatively efficient numerical tools for generating data, the ability to train models with lower amounts of data is ideal.

One approach for reducing the amount of raw data required is generating more data from the raw data already acquired.

The Ising model exhibits structure that is invariant, apart from striped systems which are for now ignored, to several transformations such as rotation, reflection and shifting.

Using these transformations one can double and create new data, however care must be taken to not change observables used in the modelling of the system.

The effectiveness of data augmentation in convolutional networks has primarily been studied in classification problems [72].

Performance has been shown to significantly increase using data augmentation in convolutional network architectures [73], however most papers use very sparse datasets for complex problems.

C.6 Activation Maps

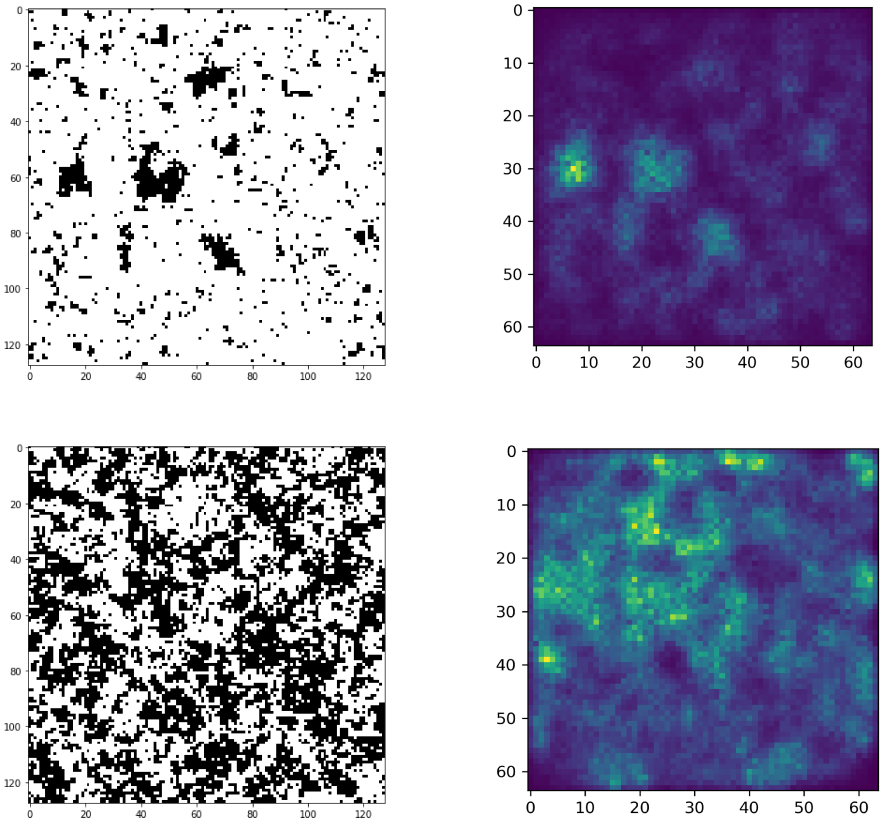


Figure 32: Visualization showing activation of two predictions on trained model. Colours indicate the value of activation with brighter being higher.

D Code

This section contains snippets of code that are of interest.

D.1 Metropolis-Hastings Multi-Core Implementation

The file `multicore_functions.py` contains functions that are used by the file `main.py` to produce the Ising systems.

The functions are `mcmc_ising_basic` which is the Metropolis Hastings algorithm. On line 38 the temperature is generated in the operating system due to an issue using the multi-processing module in Python.

This can be run by using the `main.py` script and selecting the number of systems to generate.

A csv file containing some data will be generated alongside it.

multicore_functions.py

```
1  import numpy as np
2  from numba import jit
3  import os
4  import pathlib
5  import csv
6
7
```

```
def mcmc_ising_basic(n=128,
                     nsteps=500000000,
                     T=1,
                     J=1,
                     H=0):
    """
    Metropolis MCMC algorithm on the square 2D Ising lattice, extremely fast

    Inputs:
        n = size of the ising array, always square
        nsteps = amount of MCMC steps
        T = Temperature of the system (kT)
        J = The constant outside of hamiltonian
        H = Outside field

    Outputs:
        ising_calculation(ising_lattice) = The 2D ising array
        nsteps = amount of MCMC steps
        T = Temperature of the system (kT)

    Requirements:
        NumPy (import numpy as np)
        Numba jit (from numba import jit)
    """

    # initialize our lattice
    ising_lattice = np.random.choice([1, -1], size=(n, n))
    ising_lattice = np.int8(ising_lattice)

    if T == "random_val":
        np.random.seed(int.from_bytes(os.urandom(4), byteorder='little'))
        T = round(np.random.uniform(1, 4), 3)

    # Precalculate the temperature
    pre_temp = 1/T

    # From here on down we are in numba jit
    @jit(nopython=True)
    def ising_calculation(lattice):

        for step in range(nsteps):

            # Randomly choose i j to index into array
            i = np.random.randint(n)
            j = np.random.randint(n)

            # Boundary conditions and neighbours
            Sn = lattice[(i - 1) % n, j] + lattice[(i + 1) % n, j] + \
                 lattice[i, (j - 1) % n] + lattice[i, (j + 1) % n]

            # Calculating local change in energy
            dE = 2 * lattice[i, j] * (H + J * Sn)

            # Metropolis check
            if np.random.random() < np.exp(-dE*pre_temp):
                lattice[i, j] = -lattice[i, j]

        return lattice

```

```
    return ising_calculation(ising_lattice), T


def create_ising(T, ID):
    p = pathlib.Path("numpy_ising_systems/")

    # Check if the folder exists
    if p.exists() is False:
        p.mkdir(parents=True, exist_ok=True)

    sys_loc = p / str(ID)

    lattice, T = mcmc_ising_basic(T=T)
    lattice = np.append([ID, T, np.sum(lattice)], lattice)

    np.save(sys_loc, lattice)

    return ID, T, np.sum(lattice[3:])


def id_check():
    p = pathlib.Path("numpy_ising_systems/")
    excel_path = pathlib.Path("ising_data.csv")

    write_header = False
    last_id = 0

    # Check if excel file exists, if not write header
    if excel_path.exists() is False:
        write_header = True
    else:
        # Get the last file ID back to continue writing
        file_ids = []

        for i in p.glob('*.*'):
            file_ids.append(int(i.stem))

        try:
            last_id = max(file_ids)
        except ValueError:
            last_id = 0

        del file_ids

    return last_id, write_header, excel_path
```

main.py

```
from multicore_functions import mcmc_ising_basic, create_ising, id_check
from data_processing import create_large_np_array
import numpy as np
from numba import jit
import os
import pathlib
from multiprocessing import Pool
import time
from functools import partial
```

```
import csv


def create_multicore_data(T, n_systems=1):
    """
    Creates n_systems number of new ising systems.
    Uses multicore_functions script to do this.
    """

    last_id, write_header, excel_path = id_check()

    with open(excel_path, 'a', newline='') as csvfile:
        mywriter = csv.writer(csvfile)

        # Write headers
        if write_header is True:
            mywriter.writerow(["ID", "T", "Lattice Sum"])

        with Pool() as pool:
            iterable = range(last_id + 1, last_id + 1 + n_systems)
            func = partial(create_ising, T)

            for i in pool.imap_unordered(func, iterable):
                mywriter.writerow(i)
                csvfile.flush()
                print(i)

if __name__ == '__main__':
    create_multicore_data(T="random_val", n_systems=200000)
    create_large_np_array(save=True)
```

D.2 Convolutional Neural Network

This section contains all the code pertaining to the machine learning aspect of the project, including the training, the prediction and evaluation.

D.2.1 Core Code without Model

This section contains the code code used in all the architectures when training the networks. On line 57 the model would be inserted as a list.

cnn_model.py

```
import numpy as np
import numba
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Leaky ReLU, BatchNormalization
from keras import backend as K
from keras.layers.core import Dense, Dropout, Activation, Flatten
from keras import optimizers
from keras.models import model_from_json
from keras.models import load_model
from keras.callbacks import EarlyStopping, ModelCheckpoint, ReduceLROnPlateau, CSVLogger
from keras_tqdm import TQDMCallback
from keras.utils import multi_gpu_model
import json
```

```
14
15  print("Starting Model")
16
17  print("Importing Training Data and Labels")
18  training_data = np.load("data/training_data.npy")
19  training_labels = np.load("data/training_labels.npy")
20
21  print("Importing Test Data and Labels")
22  validation_data = np.load("data/validation_data.npy")
23  validation_labels = np.load("data/validation_labels.npy")
24
25
26  def cnn_regressor(training_data,
27                    training_labels,
28                    validation_data,
29                    validation_labels,
30                    model_name="model1",
31                    save_model=True,
32                    save_history=True,
33                    batch_size=128,
34                    epochs=12,
35                    predictions=True):
36
37
38      # input image dimensions
39      img_rows, img_cols = 128, 128
40
41      input_shape = (img_rows, img_cols, 1)
42
43      if K.image_data_format() == 'channels_first':
44          training_data = training_data.reshape(training_data.shape[0], 1, img_rows, img_cols)
45          test_data = test_data.reshape(test_data.shape[0], 1, img_rows, img_cols)
46          input_shape = (1, img_rows, img_cols)
47      else:
48          training_data = training_data.reshape(training_data.shape[0], img_rows, img_cols, 1)
49          test_data = test_data.reshape(test_data.shape[0], img_rows, img_cols, 1)
50          input_shape = (img_rows, img_cols, 1)
51
52      print("Running CNN")
53      print('x_train shape:', training_data.shape)
54      print(training_data.shape[0], 'training samples')
55      print(test_data.shape[0], 'test samples')
56
57      model = Sequential([MODEL_HERE])
58
59      model.compile(optimizer=optimizers.Adam(lr=0.0002),
60                    loss='mean_squared_error')
61
62      # Call Backs for early stopping and optimizing
63      csv_logger = CSVLogger('TRAININGNAME.csv')
64      early_stop = EarlyStopping(monitor='val_loss',
65                                 min_delta=0.0005,
66                                 patience=10,
67                                 verbose=1,
68                                 mode='min')
69      mcp_save = ModelCheckpoint('.MODELNAME.hdf5',
70                                 save_best_only=True,
71                                 monitor='val_loss',
72                                 mode='min')
```

```
    reduce_lr_loss = ReduceLROnPlateau(monitor='val_loss',
                                       factor=0.25,
                                       patience=3,
                                       verbose=1,
                                       min_delta=1e-4,
                                       mode='min',
                                       min_lr=1e-6)

    history = model.fit(training_data, training_labels,
                        batch_size=batch_size,
                        epochs=epochs,
                        verbose=2,
                        validation_data=(test_data, test_labels),
                        callbacks=[TQDMCallback(leave_inner=True,leave_outer=True),
                                   early_stop,
                                   mcp_save,
                                   reduce_lr_loss,
                                   early_stop,
                                   csv_logger])

    score = model.evaluate(test_data, test_labels, verbose=0)
    print(score)

    if save_model:
        print("Saving model")
        model.save(model_name + ".h5")

    if save_history:
        print("Saving history")
        with open(model_name + "_history.json", 'w') as f:
            pickle.dump(history.history, f)

    if predictions:
        print("Running predictions")
        predictions = model.predict(test_data)
        return predictions


print("Training Network")
predictions = cnn_regressor(training_data = training_data,
                            training_labels = training_labels,
                            test_data = validation_data,
                            test_labels = validation_labels,
                            epochs = 50,
                            batch_size = 128)
```

D.2.2 Primary CNN topology (Model 1, 2 8)

The code for the primary CNN topology follows a similar scheme to VGGNet, with the only normalization performed by the dropout layers on line 31 and 33.

model1.py

```
model = Sequential([
                    Conv2D(64, (3, 3), input_shape=input_shape, padding='same',
                           activation='relu'),
                    Conv2D(64, (3, 3), activation='relu', padding='same'),
```

```
                    Conv2D(64, (3, 3), activation='relu', padding='same'),
                    MaxPooling2D(pool_size=(2, 2), strides=(2, 2)),
                    Conv2D(128, (3, 3), activation='relu', padding='same'),
                    Conv2D(128, (3, 3), activation='relu', padding='same',),
                    Conv2D(128, (3, 3), activation='relu', padding='same',),
                    MaxPooling2D(pool_size=(2, 2), strides=(2, 2)),
                    Conv2D(256, (3, 3), activation='relu', padding='same',),
                    Conv2D(256, (3, 3), activation='relu', padding='same',),
                    Conv2D(256, (3, 3), activation='relu', padding='same',),
                    MaxPooling2D(pool_size=(2, 2), strides=(2, 2)),
                    Conv2D(512, (3, 3), activation='relu', padding='same',),
                    Conv2D(512, (3, 3), activation='relu', padding='same',),
                    Conv2D(512, (3, 3), activation='relu', padding='same',),
                    MaxPooling2D(pool_size=(2, 2), strides=(2, 2)),
                    Conv2D(512, (3, 3), activation='relu', padding='same',),
                    Conv2D(512, (3, 3), activation='relu', padding='same',),
                    Conv2D(512, (3, 3), activation='relu', padding='same',),
                    Conv2D(512, (3, 3), activation='relu', padding='same',),
                    MaxPooling2D(pool_size=(2, 2), strides=(2, 2)),
                    Conv2D(512, (3, 3), activation='relu', padding='same',),
                    Conv2D(512, (3, 3), activation='relu', padding='same',),
                    Conv2D(512, (3, 3), activation='relu', padding='same',),
                    Conv2D(512, (3, 3), activation='relu', padding='same',),
                    MaxPooling2D(pool_size=(2, 2), strides=(2, 2)),
                    Flatten(),
                    Dense(4096, activation='relu'),
                    Dropout(0.25),
                    Dense(4096, activation='relu'),
                    Dropout(0.25),
                    Dense(1)
                    ])
```

D.2.3 CNN with no Pooling (Model 3)

Model 3 follows the same overall topology as the core network with the removal of the pooling layer and instead the increased stride. No additional convolutional layers were added, instead every layer previous to a pooling one had its stride increased.

model3.py

```
model = Sequential([
                    Conv2D(64, (3, 3), input_shape=input_shape, padding='same',
                           activation='relu'),
                    Conv2D(64, (3, 3), activation='relu', padding='same'),
                    Conv2D(64, (3, 3), strides = 2, activation='relu', padding='same'),
                    Conv2D(128, (3, 3), activation='relu', padding='same'),
                    Conv2D(128, (3, 3), activation='relu', padding='same',),
                    Conv2D(128, (3, 3), strides = 2, activation='relu', padding='same',),
                    Conv2D(256, (3, 3), activation='relu', padding='same',),
                    Conv2D(256, (3, 3), activation='relu', padding='same',),
                    Conv2D(256, (3, 3), strides = 2, activation='relu', padding='same',),
                    Conv2D(512, (3, 3), activation='relu', padding='same',),
                    Conv2D(512, (3, 3), activation='relu', padding='same',),
                    Conv2D(512, (3, 3), strides = 2, activation='relu', padding='same',),
                    Conv2D(512, (3, 3), activation='relu', padding='same',),
                    Conv2D(512, (3, 3), activation='relu', padding='same',),
                    Conv2D(512, (3, 3), activation='relu', padding='same',),
```

```
                    Conv2D(512, (3, 3), strides = 2, activation='relu', padding='same',),
                    Conv2D(512, (3, 3), activation='relu', padding='same',),
                    Conv2D(512, (3, 3), activation='relu', padding='same',),
                    Conv2D(512, (3, 3), activation='relu', padding='same',),
                    Conv2D(512, (3, 3), strides = 2, activation='relu', padding='same',),
                    Flatten(),
                    Dense(4096, activation='relu'),
                    Dropout(0.25),
                    Dense(4096, activation='relu'),
                    Dropout(0.25),
                    Dense(1)
                    ])
```

D.2.4 CNN with Dropout (Model 4)

This is the same code as the previous core model, with the addition of dropout layers after each pooling operation.

model4.py

```
model = Sequential([
                    Conv2D(64, (3, 3), input_shape=input_shape, padding='same',
                           activation='relu'),
                    Conv2D(64, (3, 3), activation='relu', padding='same'),
                    Conv2D(64, (3, 3), activation='relu', padding='same'),
                    MaxPooling2D(pool_size=(2, 2), strides=(2, 2)),
                    Dropout(0.25),
                    Conv2D(128, (3, 3), activation='relu', padding='same'),
                    Conv2D(128, (3, 3), activation='relu', padding='same',),
                    Conv2D(128, (3, 3), activation='relu', padding='same',),
                    MaxPooling2D(pool_size=(2, 2), strides=(2, 2)),
                    Dropout(0.25),
                    Conv2D(256, (3, 3), activation='relu', padding='same',),
                    Conv2D(256, (3, 3), activation='relu', padding='same',),
                    Conv2D(256, (3, 3), activation='relu', padding='same',),
                    MaxPooling2D(pool_size=(2, 2), strides=(2, 2)),
                    Dropout(0.25),
                    Conv2D(512, (3, 3), activation='relu', padding='same',),
                    Conv2D(512, (3, 3), activation='relu', padding='same',),
                    Conv2D(512, (3, 3), activation='relu', padding='same',),
                    MaxPooling2D(pool_size=(2, 2), strides=(2, 2)),
                    Dropout(0.25),
                    Conv2D(512, (3, 3), activation='relu', padding='same',),
                    Conv2D(512, (3, 3), activation='relu', padding='same',),
                    Conv2D(512, (3, 3), activation='relu', padding='same',),
                    Conv2D(512, (3, 3), activation='relu', padding='same',),
                    MaxPooling2D(pool_size=(2, 2), strides=(2, 2)),
                    Dropout(0.25),
                    Conv2D(512, (3, 3), activation='relu', padding='same',),
                    Conv2D(512, (3, 3), activation='relu', padding='same',),
                    Conv2D(512, (3, 3), activation='relu', padding='same',),
                    Conv2D(512, (3, 3), activation='relu', padding='same',),
                    MaxPooling2D(pool_size=(2, 2), strides=(2, 2)),
                    Dropout(0.25),
                    Flatten(),
                    Dense(4096, activation='relu'),
                    Dropout(0.25),
```

```
                    Dense(4096, activation='relu'),
                    Dropout(0.25),
                    Dense(1)
                    ])
```

D.2.5 CNN with Batch-normalization before activation (Model 5)

model5.py

```
model = Sequential([
                    Conv2D(64, (3, 3), input_shape=input_shape, padding='same'),
                    Activation('relu'),
                    Conv2D(64, (3, 3), padding='same'),
                    Activation('relu'),
                    Conv2D(64, (3, 3), padding='same'),
                    Activation('relu'),
                    MaxPooling2D(pool_size=(2, 2), strides=(2, 2)),
                    Conv2D(128, (3, 3), padding='same'),
                    BatchNormalization(),
                    Activation('relu'),
                    Conv2D(128, (3, 3), padding='same',),
                    Activation('relu'),
                    Conv2D(128, (3, 3), padding='same',),
                    Activation('relu'),
                    MaxPooling2D(pool_size=(2, 2), strides=(2, 2)),
                    Conv2D(256, (3, 3), padding='same',),
                    BatchNormalization(),
                    Activation('relu'),
                    Conv2D(256, (3, 3), padding='same',),
                    Activation('relu'),
                    Conv2D(256, (3, 3), padding='same',),
                    Activation('relu'),
                    MaxPooling2D(pool_size=(2, 2), strides=(2, 2)),
                    Conv2D(512, (3, 3), padding='same',),
                    BatchNormalization(),
                    Activation('relu'),
                    Conv2D(512, (3, 3), padding='same',),
                    Activation('relu'),
                    Conv2D(512, (3, 3), padding='same',),
                    Activation('relu'),
                    MaxPooling2D(pool_size=(2, 2), strides=(2, 2)),
                    Conv2D(512, (3, 3), padding='same',),
                    BatchNormalization(),
                    Activation('relu'),
                    Conv2D(512, (3, 3), padding='same',),
                    Activation('relu'),
                    Conv2D(512, (3, 3), padding='same',),
                    Activation('relu'),
                    Conv2D(512, (3, 3), padding='same',),
                    Activation('relu'),
                    MaxPooling2D(pool_size=(2, 2), strides=(2, 2)),
                    Conv2D(512, (3, 3), padding='same',),
                    BatchNormalization(),
                    Activation('relu'),
                    Conv2D(512, (3, 3), padding='same',),
                    Activation('relu'),
```

```
                        Conv2D(512, (3, 3), padding='same',),
                        Activation('relu'),
                        Conv2D(512, (3, 3), padding='same',),
                        Activation('relu'),
                        MaxPooling2D(pool_size=(2, 2), strides=(2, 2)),
                        Flatten(),
                        Dense(4096, activation='relu'),
                        Dropout(0.25),
                        Dense(4096, activation='relu'),
                        Dropout(0.25),
                        Dense(1)
                        ])
```

D.2.6 CNN with Batch-normalization after activation (Model 6)

model6.py

```
    model = Sequential([
                        Conv2D(64, (3, 3), input_shape=input_shape, padding='same'),
                        Activation('relu'),
                        Conv2D(64, (3, 3), padding='same'),
                        Activation('relu'),
                        Conv2D(64, (3, 3), padding='same'),
                        Activation('relu'),
                        MaxPooling2D(pool_size=(2, 2), strides=(2, 2)),
                        Conv2D(128, (3, 3), padding='same'),
                        Activation('relu'),
                        BatchNormalization(),
                        Conv2D(128, (3, 3), padding='same',),
                        Activation('relu'),
                        Conv2D(128, (3, 3), padding='same',),
                        Activation('relu'),
                        MaxPooling2D(pool_size=(2, 2), strides=(2, 2)),
                        Conv2D(256, (3, 3), padding='same',),
                        Activation('relu'),
                        BatchNormalization(),
                        Conv2D(256, (3, 3), padding='same',),
                        Activation('relu'),
                        Conv2D(256, (3, 3), padding='same',),
                        Activation('relu'),
                        MaxPooling2D(pool_size=(2, 2), strides=(2, 2)),
                        Conv2D(512, (3, 3), padding='same',),
                        Activation('relu'),
                        BatchNormalization(),
                        Conv2D(512, (3, 3), padding='same',),
                        Activation('relu'),
                        Conv2D(512, (3, 3), padding='same',),
                        Activation('relu'),
                        MaxPooling2D(pool_size=(2, 2), strides=(2, 2)),
                        Conv2D(512, (3, 3), padding='same',),
                        Activation('relu'),
                        BatchNormalization(),
                        Conv2D(512, (3, 3), padding='same',),
                        Activation('relu'),
                        Conv2D(512, (3, 3), padding='same',),
                        Activation('relu'),
```

```
                        Conv2D(512, (3, 3), padding='same',),
                        Activation('relu'),
                        MaxPooling2D(pool_size=(2, 2), strides=(2, 2)),
                        Conv2D(512, (3, 3), padding='same',),
                        Activation('relu'),
                        BatchNormalization(),
                        Conv2D(512, (3, 3), padding='same',),
                        Activation('relu'),
                        Conv2D(512, (3, 3), padding='same',),
                        Activation('relu'),
                        Conv2D(512, (3, 3), padding='same',),
                        Activation('relu'),
                        MaxPooling2D(pool_size=(2, 2), strides=(2, 2)),
                        Flatten(),
                        Dense(4096, activation='relu'),
                        Dropout(0.25),
                        Dense(4096, activation='relu'),
                        Dropout(0.25),
                        Dense(1)
                        ])
```

D.2.7 CNN with Leaky ReLU (Model 7)

model7.py

```
model = Sequential([
                    Conv2D(64, (3, 3), input_shape=input_shape, padding='same'),
                    Leaky ReLU(),
                    Conv2D(64, (3, 3), padding='same'),
                    Leaky ReLU(),
                    Conv2D(64, (3, 3), padding='same'),
                    Leaky ReLU(),
                    MaxPooling2D(pool_size=(2, 2), strides=(2, 2)),
                    Conv2D(128, (3, 3), padding='same'),
                    Leaky ReLU(),
                    Conv2D(128, (3, 3), padding='same',),
                    Leaky ReLU(),
                    Conv2D(128, (3, 3), padding='same',),
                    Leaky ReLU(),
                    MaxPooling2D(pool_size=(2, 2), strides=(2, 2)),
                    Conv2D(256, (3, 3), padding='same',),
                    Leaky ReLU(),
                    Conv2D(256, (3, 3), padding='same',),
                    Leaky ReLU(),
                    Conv2D(256, (3, 3), padding='same',),
                    Leaky ReLU(),
                    MaxPooling2D(pool_size=(2, 2), strides=(2, 2)),
                    Conv2D(512, (3, 3), padding='same',),
                    Leaky ReLU(),
                    Conv2D(512, (3, 3), padding='same',),
                    Leaky ReLU(),
                    Conv2D(512, (3, 3), padding='same',),
                    Leaky ReLU(),
                    MaxPooling2D(pool_size=(2, 2), strides=(2, 2)),
                    Conv2D(512, (3, 3), padding='same',),
                    Leaky ReLU(),
```

```
                    Conv2D(512, (3, 3), padding='same',),
                    Leaky ReLU(),
                    Conv2D(512, (3, 3), padding='same',),
                    Leaky ReLU(),
                    Conv2D(512, (3, 3), padding='same',),
                    Leaky ReLU(),
                    MaxPooling2D(pool_size=(2, 2), strides=(2, 2)),
                    Conv2D(512, (3, 3), padding='same',),
                    Leaky ReLU(),
                    Conv2D(512, (3, 3), padding='same',),
                    Leaky ReLU(),
                    Conv2D(512, (3, 3), padding='same',),
                    Leaky ReLU(),
                    Conv2D(512, (3, 3), padding='same',),
                    Leaky ReLU(),
                    MaxPooling2D(pool_size=(2, 2), strides=(2, 2)),
                    Flatten(),
                    Dense(4096),
                    Leaky ReLU(),
                    Dropout(0.25),
                    Dense(4096),
                    Leaky ReLU(),
                    Dropout(0.25),
                    Dense(1)
                    ])
```

D.3 Data Processing

The need for a function to take in a dataset and split it into a training and validation set without the increase of memory was great. This enables the use of computers with smaller memory, however sacrifices some readability.

This function requires temperatures to be packaged inside

data_processing.py

```python
def create_ml_sets(data, split=0.75, verbose=False):
    """
    Creates training and test sets and labels.

    :param large_array: The properly formatted numpy array of ising arrays
    :param split: The % of data inside the training data
    :param verbose: Information about your data
    :return: returns training set, labels, and test set and labels
    """

    print(f"""Splitting data into {round(split * 100)}% training, and
    {round((1 - split) * 100)}% test.""")
    len_training = round(len(data) * split)

    data = np.split(data, [len_training], axis = 0)

    print("Creating training data and labels")
    data[0] = np.split(data[0], [1,2,3], axis = 1)
    del data[0][0]
    del data[0][1]
    data[0][0] = np.squeeze(data[0][0])
    data[0][1] = np.squeeze(data[0][1])
```

```
23
24      print("Creating test data and labels")
25      data[1] = np.split(data[1], [1,2,3], axis = 1)
26      del data[1][0]
27      del data[1][1]
28      data[1][0] = np.squeeze(data[1][0])
29      data[1][1] = np.squeeze(data[1][1])
30
31      print("Data has been returned")
32
33      if verbose is True:
34          print(f"""\nInformation about our Training/Test Data
35
36          Size of Training Data: {data[0][0].shape[0]}
37          Size of Test Data: {data[1][0].shape[0]}
38          Dimensions of data:
39          {int(np.sqrt(data[0][1].shape[1]))}x{int(np.sqrt(data[0][1].shape[1]))}""")
40
41      return data
```

D.4 Predictions

Prediction Code

```
1   import numpy as np
2   import numba
3   from keras.models import Sequential
4   from keras.layers import Conv2D, MaxPooling2D, BatchNormalization, Leaky ReLU
5   from keras import backend as K
6   from keras.layers.core import Dense, Dropout, Activation, Flatten
7   from keras import optimizers
8   from keras.models import model_from_json
9   from keras.models import load_model
10  from keras.callbacks import EarlyStopping, ModelCheckpoint, ReduceLROnPlateau, CSVLogger
11  import argparse
12
13
14  def predictions(data,
15                  model_name="model1"):
16      """
17      Perform predictions on trained models. Requires models to be .h5, preferably
18      generated using the proper model functions.
19
20      Inputs:
21          data - Systems to predict on, shape (x, n^2) for reshaping
22          model_name - file name of .h5 file without the .h5
23
24      Outputs:
25          saves file to disk using the number inside model name
26      """
27
28
29      # input image dimensions
30      img_rows, img_cols = 128, 128
31
32      input_shape = (img_rows, img_cols, 1)
```

```
33
34          print(K.image_data_format())
35
36          if K.image_data_format() == 'channels_first':
37              data = data.reshape(data.shape[0], 1, img_rows, img_cols)
38              input_shape = (1, img_rows, img_cols)
39          else:
40              data = data.reshape(data.shape[0], img_rows, img_cols, 1)
41              input_shape = (img_rows, img_cols, 1)
42
43          print("Running CNN")
44          print('x_train shape:', data.shape)
45          print(data.shape[0], 'training samples')
46          print(data.shape[0], 'test samples')
47
48          #model = load_model(model_name + ".h5")
49          model = load_model(model_name)
50
51          predictions = model.predict(data)
52
53          return predictions
54
55
56      # import test data
57      test_data = np.load("../data/test_data.npy")
58      test_labels = np.load("../data/test_labels.npy")
59
60      # Setup argument parsing
61      ap = argparse.ArgumentParser()
62      ap.add_argument("-n", "--name", required=True,
63              help="name of the model")
64      args = vars(ap.parse_args())
65
66      predictions = predictions(data = test_data, model_name=args["name"])
67
68      model_name = args["name"]
69
70      if ".hdf5" in model_name:
71              prediction_name = model_name.replace(".hdf5", "_predictions.npy")
72      elif ".h5" in model_name:
73              prediction_name = model_name.replace(".h5", "_predictions.npy")
74      else:
75              "Error model name not understood"
76
77      np.save(prediction_name, predictions)
```