IMPORTANCE SAMPLING AND MACHINE LEARNING APPROACH FOR CLASSICAL ISING MODEL

A thesis Submitted
in Partial Fulfilment of the Requirements
for the Degree of

MASTER OF SCIENCE

by

ARITRA MUKHOPADHYAY

to the

School of Physical Sciences

National Institute of Science Education and Research

Bhubaneswar

Date 15th May, 2025

DECLARATION

I hereby declare that I am the sole author of this thesis in partial fulfillment of the requirements for a postgraduate degree from National Institute of Science Education and Research (NISER). I authorize NISER to lend this thesis to other institutions or individuals for the purpose of scholarly research.

Aritra Mukhopadhyay
Signature of the Student

Date: 16/5/25

The thesis work reported in the thesis entitled *Importance Sampling and Machine Learning Approach for Classical Ising Model* was carried out under my supervision, in the *School of Physical Sciences* at NISER, Bhubaneswar, India.

Signature of the thesis supervisor

School: Physical Sciences

Date: 16/5/25

Acknowledgement

I would like to express my deep gratitude to my supervisor, *Prof. Anamitra Mukherjee*, for his guidance and support throughout this project. I also appreciate the insightful discussions with my friends *Deependra Singh*, *Jabed Umar* and *Sajag Kumar*, which helped me gain valuable perspectives on related topics.

Abstract

This thesis explores the integration of machine learning techniques into Monte Carlo simulations of spin systems, with a focus on accelerating the generation of equilibrium configurations in the classical Ising model. The traditional Metropolis algorithm, though widely used, can be computationally intensive for large systems or when extended to more complex interactions. To address this, we formulate the sampling process as a conditional generation task, where generative neural networks are trained to produce spin configurations consistent with a given set of physical parameters, such as temperature and coupling strengths.

Two classes of generative models—Generative Adversarial Networks (GANs) and Diffusion Models—are implemented and evaluated based on their ability to reproduce physical observables and statistical properties of the target distribution. The models are benchmarked against classical Monte Carlo techniques to assess improvements in computational efficiency and physical fidelity.

Beyond the classical regime, this framework is preliminarily extended to semiclassical and quantum systems, where traditional approaches require costly $O(n^3)$ matrix diagonalization steps. Early results suggest that generative models may offer a viable alternative for bypassing such computational bottlenecks, though ensuring physical consistency remains a central challenge.

Overall, this work demonstrates the potential of modern generative modeling to enhance the scalability and speed of statistical physics simulations, offering a new paradigm for studying equilibrium properties in complex many-body systems.

Contents

Chapter 1

The Classical Ising Model and Monte Carlo Simulations

This chapter provides an overview of the Classical Ising Model, a fundamental statistical mechanics model used to study phase transitions in magnetic materials. We begin by introducing the mathematical formulation of the Ising model, including its Hamiltonian and key quantities such as energy and magnetization. We then discuss the challenges associated with solving the Ising model exactly, highlighting the poor time complexity of traditional methods. To overcome these limitations, we turn to Monte Carlo simulations, which provide a powerful tool for approximating the behavior of complex systems. In this chapter, we introduce the basic principles of Monte Carlo simulations, including importance sampling and Markov chains, and describe the Metropolis algorithm used in our research.

1.1 The Classical Ising Model and its Extensions

The Classical Ising Model is a foundational framework in statistical physics for modeling the behavior of magnetic materials in thermal equilibrium. It consists of a lattice of discrete sites, each occupied by a spin variable S_i which can take values ± 1. These spins interact with their neighbors through exchange interactions, and optionally, with an external magnetic field.

Despite its apparent simplicity, the Ising model captures key physical phenomena such as phase transitions and collective behavior, making it a valuable tool in fields ranging from condensed matter physics to computational neuroscience.

1.1.1 Basic Formulation

The energy of a spin configuration in the classical Ising model is given by the Hamiltonian:

$$\mathcal{H} = -J \sum_{\langle i,j \rangle} S_i S_j - h \sum_i S_i, \tag{1.1}$$

where: - S_i is the spin at lattice site i, - J is the exchange coupling constant between nearest neighbors, - h is the external magnetic field, - $\langle i, j \rangle$ denotes a sum over nearest-neighbor pairs.

We considered a **square lattice** geometry, where each site interacts with its four immediate neighbors: up, down, left, and right. This geometry, with periodic boundary conditions, allows for simulating bulk-like behavior in finite systems. An illustration of the lattice and its

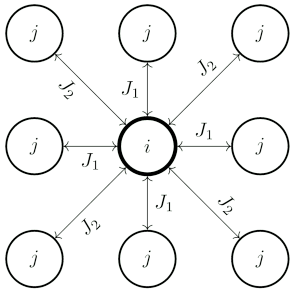


Fig. 1.1: Square Lattice Neigbors

neighborhood structure is provided in Figure 1.1.

1.1.2 Extensions and Current Work

In the current work, we extend the Ising model in two key ways:

1. **Non-zero external magnetic field ($h \neq 0$)**: We retain the full form of the Hamiltonian in Eq. (1.1) to study the influence of an applied field on magnetization and phase transitions.
2. **Higher-order interactions**: We also include next-nearest neighbor (NNN) interactions, leading to a generalized Hamiltonian:

$$\mathcal{H} = -J_1 \sum_{\langle i,j \rangle} S_i S_j - J_2 \sum_{\langle\langle i,j \rangle\rangle} S_i S_j - \cdots - h \sum_i S_i, \tag{1.2}$$

   where $\langle\langle i, j \rangle\rangle$ denotes next-nearest neighbor pairs, and J_2 is the corresponding coupling strength.

These additions allow us to investigate richer behavior such as magnetic frustration, especially on square lattices where next-nearest neighbors are the diagonal corners of a spin's local neighborhood.

In subsequent chapters, we simulate this model using the Metropolis algorithm and analyze physical observables such as energy, magnetization, and the structure factor.

1.1.3 Calculating some important properties

Energy: The Hamiltonian function represents the energy of the system, and calculating it is necessary to understanding the behavior of the Ising model. To compute the total energy, we need to visit each lattice point i and sum over the products $S_i S_j$ for all its nearest neighbors j.

Magnetization: The magnetization of the system is a measure of the net magnetic moment per unit volume. In the context of the Ising model, it can be defined as the average spin value over all lattice sites. Mathematically, the magnetization M can be expressed as:

$$M = \frac{1}{N} \sum_{i=1}^{N} S_i \tag{1.3}$$

where N is the total number of lattice sites and S_i is the spin value at site i.

Structure Factor: To characterise the spatial correlations in spin configurations, we compute the static structure factor. It is a fundamental tool in condensed matter physics for identifying ordering tendencies and characteristic length scales in a system.

For a spin configuration $S(\mathbf{r})$ defined over a lattice, the structure factor $S(\mathbf{q})$ is given by the following equation:

$$S(\mathbf{q}) = \frac{1}{N} \left| \sum_{\mathbf{r}} S(\mathbf{r})\, e^{-i\mathbf{q}\cdot\mathbf{r}} \right|, \tag{1.4}$$

where $\mathbf{r}$ denotes the position vector on the lattice, $\mathbf{q}$ is the corresponding momentum-space vector, and N is the total number of lattice sites.

This quantity captures the Fourier components of spin correlations and reveals underlying patterns or periodic order present in the system. Peaks in $S(\mathbf{q})$ typically indicate dominant wavevectors associated with magnetic or structural order.

1.2 The Double Exchange Model

In the Double Exchange (DE) model [5], each lattice site hosts two types of degrees of freedom: a classical localised spin representing the t_{2g} electrons, and a mobile quantum particle (electron) corresponding to the e_g orbital. The localised spins are fixed in magnitude and orientation at each site, while the quantum particles can hop between neighbouring sites.

The full Hamiltonian of the system is given by:

$$H = \sum_{\langle i,j\rangle,s} -t_{ij} c^{\dagger}_{i,s} c_{j,s} - J_H \sum_i \mathbf{S}_i \cdot \mathbf{s}_i, \tag{1.5}$$

where t_{ij} is the hopping amplitude, $c^{\dagger}_{i,s}$ and $c_{j,s}$ are the fermionic creation and annihilation operators, and J_H is the strength of the Hund's coupling between the classical spin $\mathbf{S}_i$ and the spin of the quantum particle $\mathbf{s}_i$ at site i.

In the strong coupling limit $J_H \to \infty$, the electron's spin aligns with the local classical spin, and the system reduces to an effective spinless hopping model with a modified, spin-dependent hopping term:

$$\tilde{t}_{ij} = t_{ij} \sqrt{\left(1 + \frac{\mathbf{S}_i \cdot \mathbf{S}_j}{S^2}\right)}, \tag{1.6}$$

$$H_t = -\sum_{i,j} \tilde{t}_{ij} c^{\dagger}_i c_j. \tag{1.7}$$

This effective Hamiltonian captures the influence of the spin background on the mobility of charge carriers and is central to understanding double exchange-driven transport phenomena.

1.3 Challenges to Solve the Ising Model

In this section, we will outline the steps required to generate one of the multiple least energy states at a given temperature for the Ising model on a lattice with N points in total, regardless of the lattice dimensionality (1D, 2D, or 3D). We will see how that becomes impractical as the system size grows; hence a motivation to use monte carlo techniques.

To generate one of these low-energy states, we would ideally need to explore all possible configurations and calculate their energies. The Hamiltonian is a function of all the spins, $\mathcal{H}(S_1, S_2, \ldots, S_n)$. In principle, we should go through each of the 2^N possible states, compute the energy for each one, and then select one of the configurations with minimum energy.

Finding the energy given a state takes $O(N)$ time, and there are 2^N states, so the total time complexity will be $O(2^N \cdot N)$. This is exponential which is bad, making this algorithm impractical as the system size grows. So we use monte carlo algorithms which help us reduce the search space by multiple orders.

1.4 About Monte Carlo and Markov Chains

Monte Carlo (MC) methods use repeated random sampling to solve complex problems that are difficult or impossible to solve deterministically. These algorithms rely on randomness to obtain numerical results, particularly in optimization, numerical integration, and sampling from probability distributions. While MC simulations are often faster than deterministic approaches, they may fail with a certain probability - a limitation that can be mitigated by multiple runs.

Importance sampling enhances MC efficiency by introducing an auxiliary proposal distribution that focuses sampling on the most relevant regions of the state space. This technique is particularly valuable when dealing with distributions that have small support or exhibit rare events.

Markov Chains and their State Transition Dynamics: Markov Chains, which form the theoretical foundation for more advanced MC methods, are systems where future states depend only on the current state, not on past states. The evolution of state probabilities is governed by the **master equation**:

$$\frac{\partial P_x(t)}{\partial t} = \sum_y [P_y(t)\pi_{yx} - P_x(t)\pi_{xy}] \tag{1.8}$$

where $P(x,t)$ represents the probability of being in state x at time t, and π_{xy} denotes transition probabilities. These concepts form the foundation for Markov Chain Monte Carlo (MCMC) methods, which construct Markov Chains with equilibrium distributions matching desired target distributions. For detailed discussion of these concepts, see Appendix A.

1.5 The Metropolis Algorithm

As seen in Section 1.3, the Classical Ising Model has an exponentially large configuration space, with 2^N possible states. In order to find a target state that satisfies a specific energy criterion, we would need to exhaustively search this entire space, which becomes computationally prohibitive for even moderately sized systems. However, as we discussed

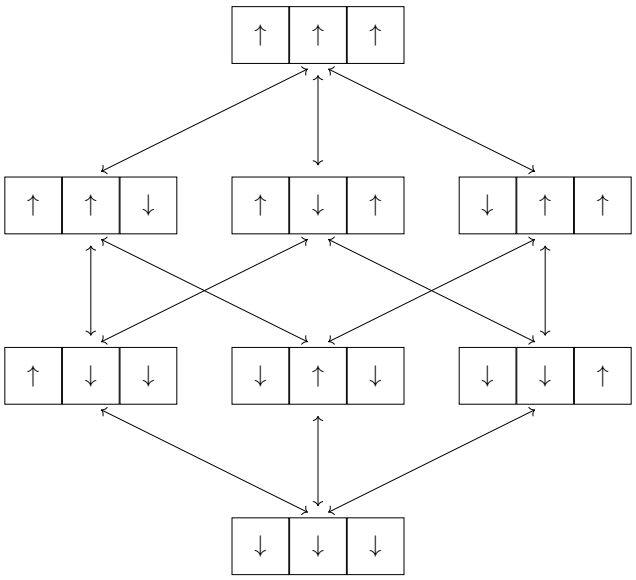


Fig. 1.2: Drawing the whole graph for the markov chain for a lattice of size $N = 3$. Here we can see that the nodes are connected by edges if they differ by a single spin flip. Hence there are $2^N (= 8)$ nodes and each node has $N (= 3)$ neighbours.

earlier, Monte Carlo methods offer a way to reduce the search space and efficiently sample from the configuration space. The goal of the Metropolis algorithm, which we will introduce in this section, is to provide a systematic approach to reducing the search space from 2^N possible states to a smaller set of relevant configurations that satisfy our energy criterion.

Formulate the Problem in terms of the Markov Chain: Let us imagine a graph data structure consisting of Nodes and Edges. Here each node represents a possible configuration of the Ising Model, and the nodes which can be reached by flipping a single spin are connected by an edge. In Figure 1.2 the whole graph for $N = 3$ has been shown for example. The plan is to start from some random node in this graph, and traverse the graph using some clever monte carlo technique that we soon get closer and closer to the required state. The path taken in the graph will be the Markov Chain.

Finding the Solution: We would start from some random node in the graph. This node has N edges. We would need to pick an edge uniformly at random (equal preference to all lattice sites) and decide whether to move to that neighbour or stay back. But how to decide whether to transition or not.

Here we will use the master equation – Equation 1.8. Here $P_x(t)$ is the probability of being at state x at time t. In equilibrium, the probability of being at state x should be independent of time. Hence $\partial P_x(t)/\partial t = 0$. Thus the relation becomes:

$$P_x(t)\pi_{xy} = P_y(t)\pi_{yx} \tag{1.9}$$

This expression 1.9 is known as the **'detailed balance'** condition. So, this states that *the probability of being at this state and moving to another state should be equal to the probability of being at the other state and moving to this one.* This is a necessary condition for the system to be in equilibrium.

Now we are at state x and we want to move to state y. The probability of moving to y (denoted by π_{xy}) is supposed to be found. By simplifying Equation 1.9 we get:

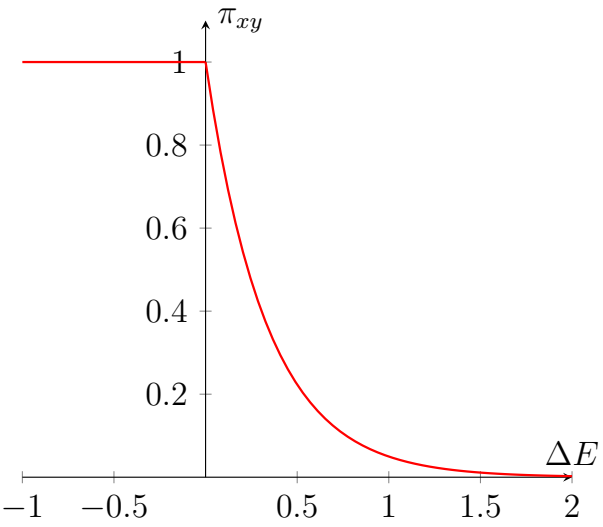


Fig. 1.3: Graph plotting Equation 1.11, how the transition probability changes with ΔE. Here we have chosen $\beta = 3$.

$$\pi_{xy} = \pi_{yx} \frac{P_y(t)}{P_x(t)} \tag{1.10}$$

Now, the probability of some state occuring is obtained from the Boltzmann distribution:

$$P_x(t) = \frac{e^{-\beta E_x}}{Z} \tag{1.11}$$

where $\beta = {}^1/_{k_B T}$, E_x is the energy of the state x and Z is the partition function. This probability is not easy to find, because of the value of Z, but that won't be necessary, because it will be cancelled out in the ratio.

Any transition probability which satisfies the detailed balance condition (Equation 1.9) is acceptable. The famous metropolis algorithm uses this transition probability:

$$\pi_{xy} = \begin{cases} e^{-\beta \Delta E} & \text{if } \Delta E > 0, \\ 1 & \text{if } \Delta E \leq 0. \end{cases} \tag{1.12}$$

This probability is plotted in graph 1.3. Thus we get the metropolis algorithm, which is summarized in Algorithm 1. Here we are setting π_{yx} to 1 for maximum efficiency.

Time Complexity Analysis

Although the exact time for this algorithm to converge to the sought for state cannot be calculated, we sure can estimate an upper bound to that. We can try to solve it in 2 ways; both of which give the same upper bound:

- As we have N sites on the lattice, the starting lattice and the end lattice can vary in atmost N places. So, the shortest distance in the graph (Figure 1.2) between the start and the end should be atmost N. Now we are using a random walk between these two states. We know that a random walk takes $O(n \log n)$ steps to reach a point which is n distance away.

Algorithm 1 Metropolis Algorithm

Compute initial energy: $E = \text{hamiltonian(state)}$
Compute initial magnetization: $M = \sum \text{state}$ (see Eq. 1.3)
Initialize empty arrays: `energies`, `magnetizations`
for $i = 1$ to n **do**
  Pick a site S_i uniformly at random
  Compute energy change ΔE if S_i is flipped
  Compute magnetization change: $\Delta M = -2 \times S_i$
  Sample $r \sim \text{Uniform}(0, 1)$
  if $\Delta E < 0$ **or** $r < e^{-\beta \Delta E}$ **then**
    Flip spin S_i in `state`
    $E \leftarrow E + \Delta E$
    $M \leftarrow M + \Delta M$
  end if
  Append E to `energies`; append M to `magnetizations`
end for
return `state`, `energies`, `magnetizations`

- There are N sites in the lattice, we are randomly picking sites and trying to flip them. What is the expected number of steps to make sure every site has got atleast one (or some constant number of) opportunity to flip. This problem is equivalent to the famous **Coupon Collector's Problem** and hence the well knows answer is $O(N \log N)$ steps.

Thus from both the above ways we found that the monte carlo method takes $O(N \log N)$ time to complete the task (N denotes number of lattice sites) as opposed to the exponential time taken by the general algorithm.

1.6 Some Results

Methodology: We start with a 2D (100×100) lattice. As the starting point we have a lattice where each point has a 25% chance of being down and 75% chance of being up. This configuration has a magnetization of +0.5. This is specifically chosen to easily demonstrate both ferromagnetic (magnetization +1 or -1) and anti-ferromagnetic (magnetization 0) conditions. We have kept $|J| = 1$. J positive denotes ferromagnetic conditions and a negative value of J denotes anti-ferromagnetic conditions. We have assumed Periodic Boundary Condition while calculating the energy using the Ising hamiltonian.

Explanation for the provided results in Figure 1.5:

- **Row 1:** We have set $|J| = -1$ and the temperature low. Thus we can see how the spins have started to align in the antiferromagnetic order. The magnetization converges to 0. And the energy converges to -40000. (This is because each site contributes a minimum energy of -4. And there are 100x100 sites in total.)
- **Row 2:** We have set $|J| = 1$ and the temperature low. Thus we can see how the spins have started to align in the ferromagnetic order. The magnetization converges to 1. And the energy converges to -40000.

- **Row 3:** We have set $|J| = -1$ and the temperature high. Thus we can see how the spins are randomly oriented and the magnetization converges to 0. And the energy randomly fluctuates around -9000.
- **Row 4:** We have set $|J| = 1$ and the temperature high. Thus we can see how the spins are randomly oriented and the magnetization converges to 0. And the energy randomly fluctuates around -9000. In both the high temperature cases we see that the energy keeps fluctuating around a -9000 irrespective of the hamiltonian.

In figure 1.4 we can see how the material acts as a ferromagnet at low temperatures and as the temperature increases, it suddenly looses it's magnetic properties and the spins align at random directions to make the net magnetization 0.

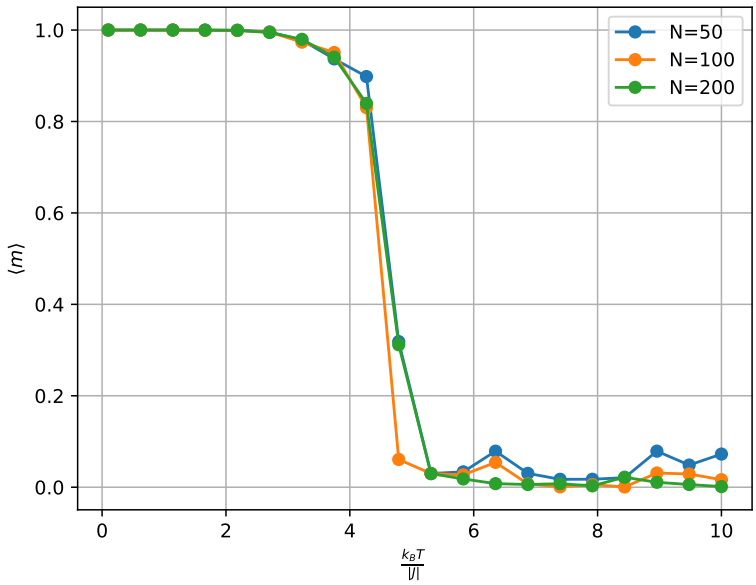


Fig. 1.4: Phase diagram of the 2D Ising model with ferromagnetic interactions ($J = +1$). The absolute value of the magnetization $\langle m \rangle$ is plotted against the reduced temperature $k_BT/|J|$ for different system sizes N. The data points represent simulation results for $N = 50$, 100, and 200 spins, each averaged over 10^7 Monte Carlo steps.

Summary: In this chapter, we introduced the Classical Ising Model and discussed how we can use Monte Carlo techniques generate a ground state configuration. I have writen the whole code on my own. For faster operations, the main metropolis function has been writen in *Rust* which is being called from *Python* for convenience. In the *next we plan to use the generated states as data to train some machine learning models.*

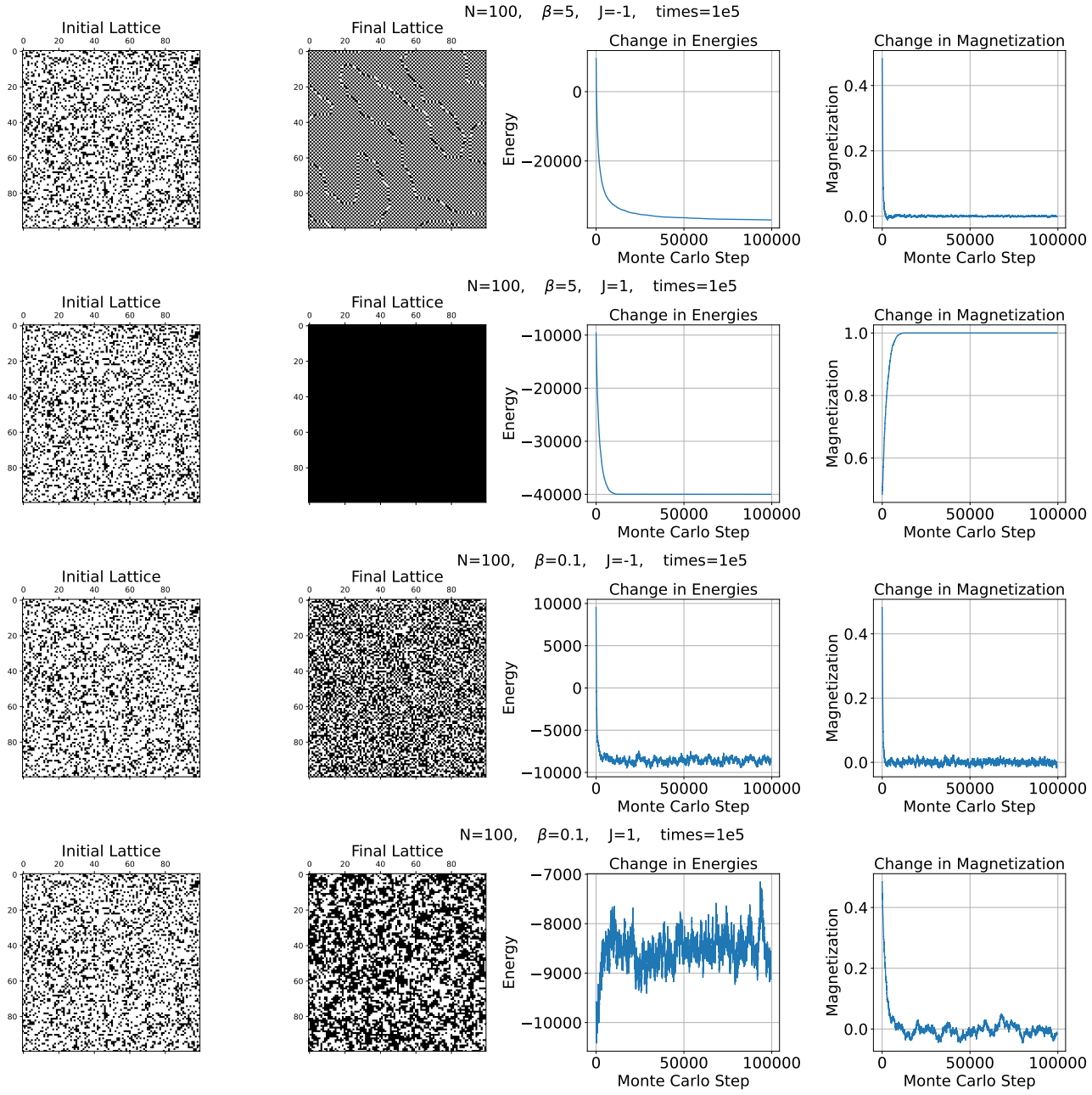


Fig. 1.5: Results for different temperatures and coupling constants. The rows show results for $\beta = 5$ (example low temperature) and $\beta = 0.1$ (example high temperature), with $J = +1$ (ferromagnetic) and $J = -1$ (antiferromagnetic) respectively. Note that the beta values have been chosen for demonstration purposes only.

Chapter 2

Machine Learning and Neural Networks

Our plan was to learn Machine Learning (ML) and apply it to Physics. For learning ML, I tried solving a toy problem: fitting the MNIST (Modified National Institute of Standards and Technology) dataset using an MLP (Multi-Layer Perceptron). The MNIST dataset consists of 28×28 grayscale labelled images of handwritten digits from 0 to 9. Although this is a toy project, it covers most of the basic challenges of training neural networks. Later, we will think about how we can apply this in Physics next. In this chapter, we would build up the theory of that and show some results.

2.1 The Linear Model

To understand neural networks, we need to first know about a simple linear model. A linear model is a mathematical representation of a relationship between inputs and outputs, where the output is assumed to be a linear combination of the inputs. In the context of handwritten digit recognition, we can formulate the problem as follows: given a grayscale image represented as a matrix of numbers, where each pixel is an 8-bit integer (0 to 255), we want to predict which handwritten digit is present in the image. Let's assume that the verdict is a function of the 784 (28x28) variables representing the pixels in the image.

To start with, let's make a strong assumption that this function is linear. Mathematically, this can be represented as:

$$\begin{aligned} y &= f(x_0, x_1, x_2, \ldots, x_{n-1}) \\ &= w_0 x_0 + w_1 x_1 + w_2 x_2 + \ldots + w_{n-1} x_{n-1} + b \end{aligned}$$

We can now shift to a more compact matrix notation, which will help us in easier abstraction and managing larger calculations with simpler notations. Incidentally, this also facilitates parallel computing, as speeding up calculations using parallel processing is well-studied in matrix notation. The above equation can be written as:

$$\hat{Y} = XW + B \tag{2.1}$$

where $\hat{Y} \in \mathbb{R}^{m \times 1}$ is the predicted output vector, $X \in \mathbb{R}^{m \times f_1}$ is the input matrix (i.e., the pixel

values), with m being the number of images and $f_1 = 784$ being the number of input features (pixel values), $W \in \mathbb{R}^{f_1 \times 1}$ is the weight matrix, and $B \in \mathbb{R}^1$ is the bias vector.

Now, let us go a bit deeper and consider predicting multiple outputs from the same 784 input values. In general we would have to make a model for each separate prediction, but with this notation we can easily do it just by changing the shapes of the matrices. For m images with $f_2 = 10$ outputs each, the predicted output vector $\hat{Y}$ has dimensions $(m \times f_2)$.

The input matrix X remains $(m \times f_1)$. To accommodate multiple outputs, the weight matrix W expands to $(f_1 \times f_2)$ and the bias vector B becomes $(1 \times f_2)$. Thus the expression remains the same:

$$\hat{Y}_{(m \times f_2)} = X_{(m \times f_1)} W_{(f_1 \times f_2)} + B_{(1 \times f_2)} \tag{2.2}$$

2.2 The Loss Function

To find the optimal values of W and B, we need to determine what constitutes a "good fit" for our model. A good fit is one where the predicted output $\hat{Y}$ is close to the actual output Y. However, we need a way to quantify how good a given fit is.

This is where the *loss function* (also known as the *cost function*) comes in. The loss function measures the difference between our model's predictions and the actual values. By minimizing this loss, we can find the optimal values of W and B that result in the best possible fit to the data.

Our goal is to minimize the loss by adjusting W and B. To do this, we need a specific mathematical function that calculates the loss. One commonly used loss function for regression tasks is the Mean Squared Error (MSE) Loss:

$$L = \frac{1}{m} \sum (\hat{Y} - Y)^2 \tag{2.3}$$

where L is the loss, m is the number of samples, $\hat{Y}$ is the predicted output, and Y is the actual output (ground truth).

This function measures the average squared difference between predicted and actual values, giving us a quantitative measure of how well our model fits the data. In Equation 2.3, each of Y and $\hat{Y}$ are matrices of shape (m, f_2), where m is the number of samples and f_2 is the number of features in the output. The subtraction operation is performed element-wise, and the sum is taken over all elements in the matrix.

2.3 Gradient Descent: A Way to Minimize the Loss

To minimize the loss function, we will explore the Gradient Descent algorithm. Other methods for minimizing the loss function include the Chi Square method, Matrix Inversion equation solve methods, Gauss-Newton method etc. However, these methods can be computationally expensive but assures exact solutions. Gradient Descent is a popular choice because it provides a good approximation of the optimal solution in lesser time.

Here's how we can use Gradient Descent to minimize the loss function:

- Initialize the model with random weights

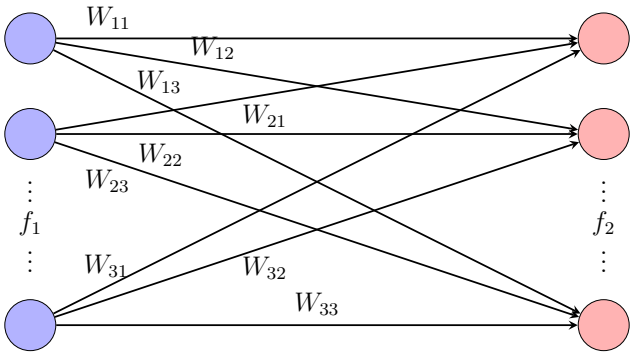


Fig. 2.1: One Linear Layer

- Pass the input through the model and calculate the predictions as in Equation 2.1.
- Find the loss value using Equation 2.3:
- Substituting $Y = WX + B$ into the loss function and differentiating with respect to W and B gives:

$$\frac{\partial L}{\partial W} = \frac{2}{m} X^T (WX + B - Y), \quad \frac{\partial L}{\partial B} = \frac{2}{m}(WX + B - Y) \tag{2.4}$$

- Update the weights and biases using the following rules:

$$W \leftarrow W - \alpha \frac{\partial L}{\partial W}, \quad B \leftarrow B - \alpha \frac{\partial L}{\partial B} \tag{2.5}$$

We run this process multiple times, updating the weights and biases at each step. As we iterate, the loss function decreases, and we converge to a minimum value. This process is repeated until convergence or a stopping criterion is reached.

2.4 The Multi Layer Perceptron

In Figure 2.1, we can see a single linear layer (without the bias) as in Equation 2.2. This simple linear layer is the building block of more complex neural networks. By combining multiple layers and adding non-linear activation functions, we can create powerful models that can learn and represent complex relationships between inputs and outputs.

A classic example of such a deep neural network is the Multi-Layer Perceptron (MLP) architecture. In an MLP, we stack multiple linear layers on top of each other, with non-linear activation functions in between. This allows the model to learn hierarchical representations of the input data and make more accurate predictions. The general structure of an MLP can be represented as:

$$\begin{aligned} H_1 &= \sigma(XW_1 + B_1) \\ H_2 &= \sigma(H_1 W_2 + B_2) \\ &\vdots \\ Y &= H_{n-1} W_n + B_n \end{aligned}$$

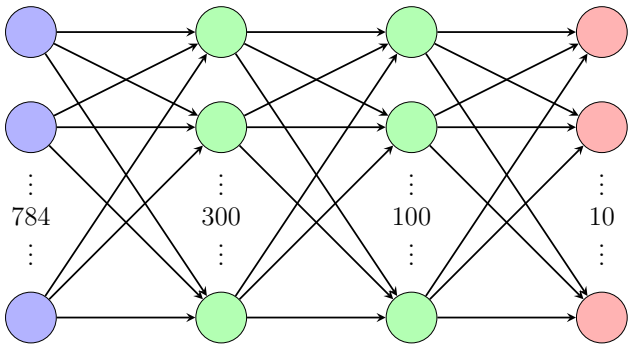


Fig. 2.2: Schematics of Our Model: Here we have 3 linear layers. The first layer takes the 784 dimensional input to 300, the second layer takes the 300 dimentions to 100, and the third layer takes the 100 dimensional input to 10 (output). There are ReLU activation functions between the layers. And the output layer has a softmax activation function.

where x is the input, W_i and b_i are the weights and biases of each layer, σ is a non-linear activation function, and y is the output.

We used a simple MLP architecture for our MNIST digit classification task. The model architecture is as follows:

- Input layer: 784 neurones (flattened 28x28 pixels)
- Hidden layer 1: 300 neurones with ReLU (Rectified Linear Unit) activation
- Hidden layer 2: 100 neurones with ReLU activation ($ReLU(x) = max(0, x)$)
- Output layer: 10 neurones with softmax activation ($\text{softmax}(x)_i = \frac{e^{x_i}}{\sum_j e^{x_j}}$ helps in converting the output to a probability distribution with unit sum, generally used in output layers of classification tasks)

We formulate our problem in such a way that the model will predict 10 outputs. We will train the model such that each output will correspond to one of the digits and the value will represent the probability of the input image belonging to that digit. The digit with the highest probability will be the predicted digit. This will help the model be more specific. If it is a 7, we want the model to be sure that it is a 7 by increasing the value of the 7th output and also be sure that it is not any other digit by decreasing the value of the other outputs.

2.5 Results

Mrthodology: The MNIST dataset contains grayscale 28x28 pixel images of handwritten digits. The dataset is split into a training set of 60,000 images and a validation set of 10,000 images. The images are labeled with the corresponding digit from 0 to 9. We used the aforementioned MLP model to classify the images. We trained the model for 50 epochs with a batch size of 1024 with learning rate 10^{-3}. We used the Adam optimizer for weight updates and the cross-entropy loss function to calculate the loss. We use `PyTorch` Python library for this purpose and trained it on a GPU.

Results: The model achieved a training accuracy of $\approx 99.9\%$ and a test accuracy of $\approx 98\%$. The accuracy and the loss curves are shown in Figure 2.3 and Figure 2.4 respectively.

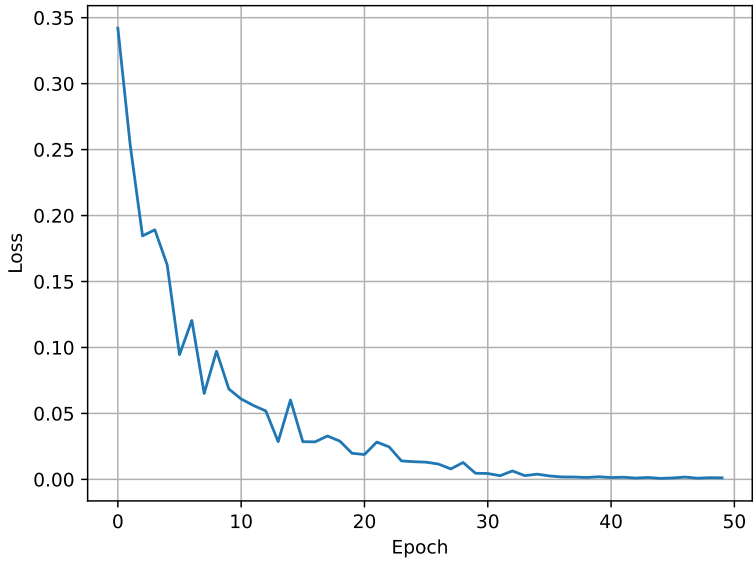


Fig. 2.3: Training loss

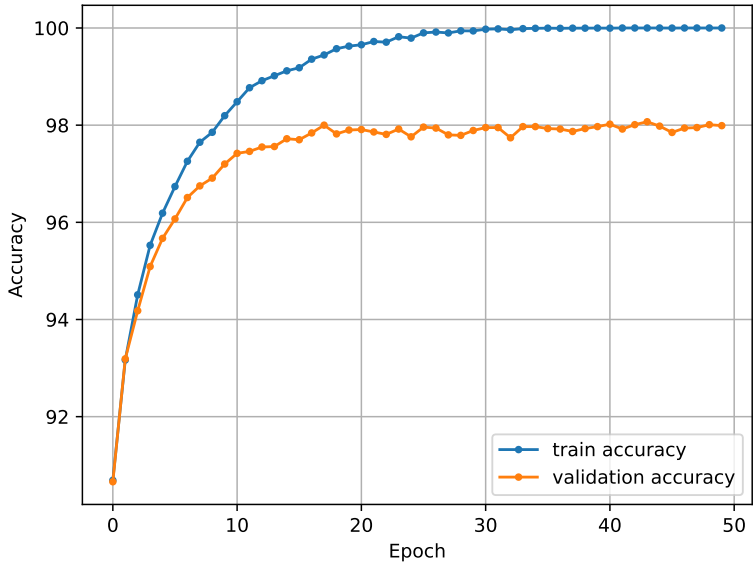


Fig. 2.4: Training and Validation accuracy

Summary: In this chapter, we introduced the basics of Machine Learning and Neural Networks. As a project, we tried to fit the MNIST dataset using an MLP. We discussed the theory of MLP, the MNIST dataset, and the training process. Next we will use this knowledge to apply ML in Physics.

Chapter 3

An Overview of Generative Algorithms

In this chapter, we explore a variety of generative algorithms that have been developed in recent years, each with their own unique theoretical motivations and practical strengths. These methods have opened the door to powerful ways of synthesizing data, generating realistic samples, and modeling complex distributions — particularly in areas where conventional methods fail.

Before diving into the specific techniques, let us understand why generative problems require a fundamentally different approach compared to the familiar classification tasks. In the previously, we worked on standard supervised learning problems — for example, given an image, the goal was to classify it into categories such as handwritten digits (MNIST) or animal classes like cat vs. dog. These tasks are well-posed: for each input, there is a clearly defined label.

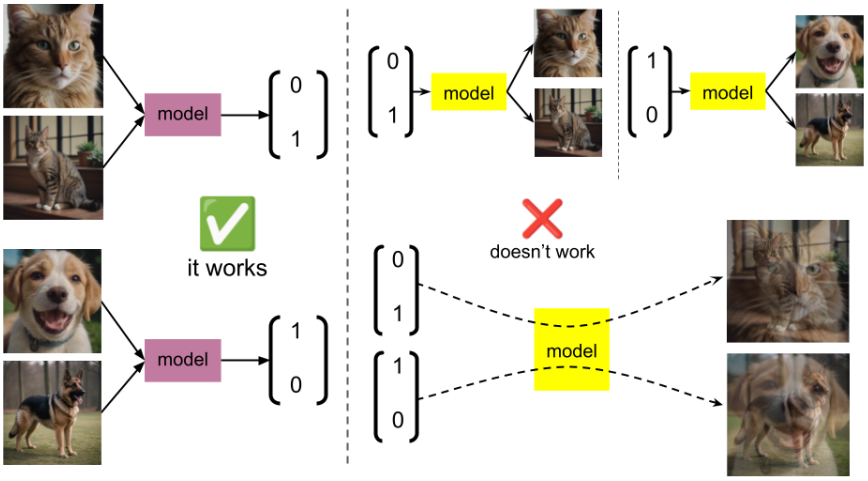


Fig. 3.1: If there are multiple possible labels (outputs) for the same input, the model will make an average of all of them. Here we can see how the model averaged the two images, the result is already quite messy, it will be illegible soon as we have more images averaged over.

Let us consider a binary classification problem between cats and dogs. Suppose we train a model that takes an image and outputs a label — a vector like $[0, 1]$ for cat and $[1, 0]$ for dog. Now, if the input image has both a cat and a dog, and the model outputs $[0.5, 0.5]$, it is still a somewhat meaningful result — the model is unsure, or interprets that both classes are present. Averaging across classes is acceptable because probabilities remain interpretable. We can see this issue described in Figure 3.1.

Now flip the problem: suppose we ask the model to take in $[0, 1]$ and output a corresponding image of a cat — and a different cat each time. Similarly, $[1, 0]$ should yield dog images. If we naively try to reverse the classification pipeline, the model tends to return blurry, average-looking outputs. Why? Because it treats the generative task like a regression problem — trying to average all possible cats to minimize error — and the result is an image that looks like none of them.

This is the core challenge in generative modeling: there is no single correct output for a given input condition. Instead of predicting one fixed answer, we must learn to sample from a distribution — to generate many plausible outputs. To do this, we need dedicated generative algorithms, built with these properties in mind. In the sections that follow, we will briefly survey some of the most influential generative algorithms developed for this purpose.

3.1 Autoregressive Models

We saw earlier that directly generating an image from a class label (like $[0, 1]$ for cat and $[1, 0]$ for dog) leads to failure — the model ends up averaging over all possibilities and generates a blurry, meaningless output. So let us **simplify the problem** further: instead of asking the model to generate the full image from scratch, let us give it the class label and the entire image except the last pixel, and **ask it to predict only that last pixel**.

It turns out — this works surprisingly well. The model can learn to predict the missing pixel with high accuracy just by seeing the rest of the image and the label.

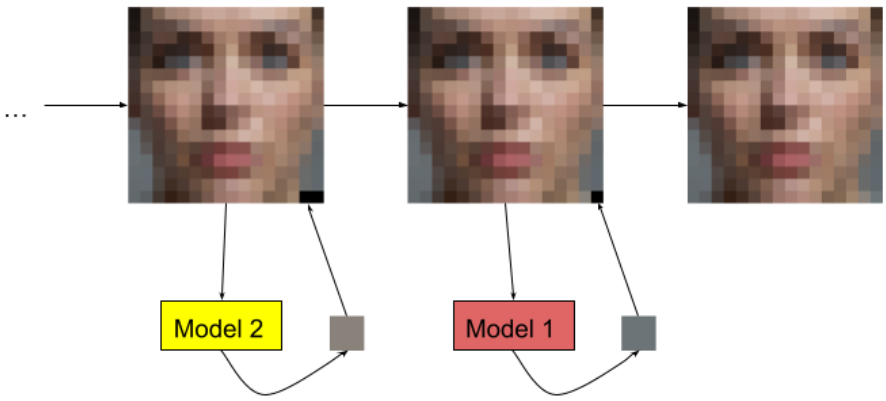


Fig. 3.2: Working of Primitive Version of Autoregression. In practice Model 1 and Model 2 are the same model and in the input we provide it with the position of the pixel to predict.

Now we ask a slightly more complex question: what if the last *two* pixels are missing? Can we train **a model that predicts the second-last pixel** based on the remaining image (excluding the last two pixels)? If yes, we can simply use our earlier model (which predicts the last pixel) to complete the image.

This approach also works. So in principle, we can learn to generate an image pixel by pixel, starting from a blank canvas, one at a time — by predicting the next pixel using a model conditioned on all previous pixels.

However, there's a **big downside**. Even for a small 16×16 resolution image, this would mean training 256 different models — one for each pixel position — and running them sequentially during inference. That's hugely inefficient.

- We train a **single** model to predict the next pixel, and provide it with a *position*

encoding (along with the class label) so it knows which pixel it is supposed to predict.

- But inference still remains sequential — we need to generate one pixel at a time, in order, because each prediction depends on all the previous ones. (This problem will be addressed in diffusion)

Mathematical Formulation

Given a data point $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, the autoregressive model factorizes the joint probability distribution into a product of conditionals:

$$P(\mathbf{x}) = P(x_1)P(x_2|x_1)P(x_3|x_1, x_2)\ldots P(x_n|x_1, x_2, \ldots, x_{n-1}) \tag{3.1}$$

This makes training a maximum likelihood problem: at each step, the model learns to predict the current pixel/token given the ones that came before.

Summary

Autoregressive models are a class of generative algorithms where each output is predicted based on the previously generated data. While they are not widely used for generating images due to the high cost of sequential pixel-by-pixel generation, they are extremely effective for one-dimensional data like text. In fact, all modern large language models — including **ChatGPT, claude, llama, deepseek** etc — are autoregressive models that generate text one token at a time, based on the prior context.

3.2 Variational Autoencoders (VAEs)

Autoregressive models generate data step-by-step, which can be slow and computationally expensive, especially for large images. So a natural question arises — can we instead find some *compressed representation* of an image (often called a **latent space**), and then learn to generate images from that?

The Core Idea

This is where **Variational Autoencoders (VAEs)** come in. The goal of a VAE is to learn a smooth, lower-dimensional latent space that captures the important structure in the data — and then learn how to decode samples from this space back into realistic images.

As shown in Figure 3.3, the architecture consists of two parts:

- An **encoder network**, which takes in an image and maps it to a latent space — specifically, it learns a mean μ and standard deviation σ representing a distribution over latent variables.
- A **decoder network**, which takes a sample from the latent distribution and reconstructs an image from it.

Instead of encoding each image to a fixed point in latent space, the VAE encodes it into a *distribution* — usually a Gaussian. During training, we sample from this distribution to feed

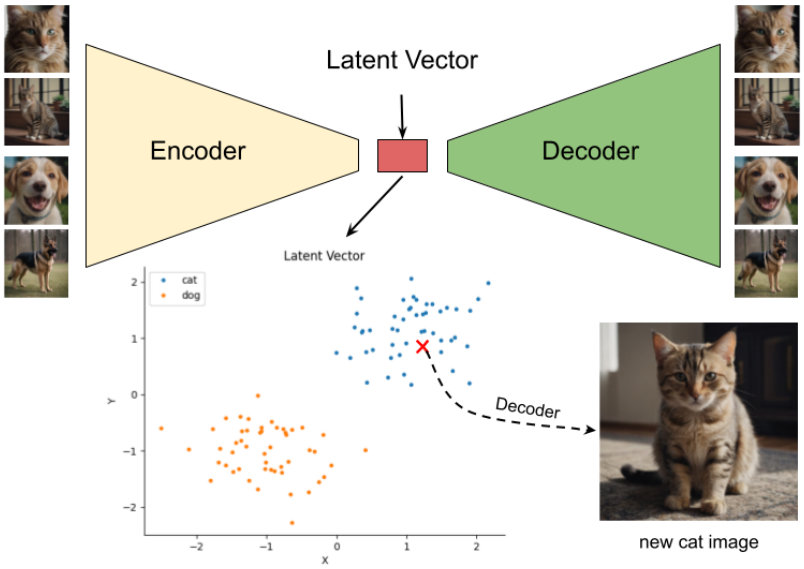


Fig. 3.3: Working of a VAE model

into the decoder. This makes the latent space continuous and allows us to generate new data points by sampling random vectors from a normal distribution.

Generation

Once trained, we can ignore the encoder entirely and just:

1. Sample a random vector z from the standard normal distribution.
2. Feed it to the decoder to generate a brand new image.

This is how VAEs can be used as generative models.

Mathematical Formulation

We assume there exists some latent variable z such that:

$$p(x) = \int p(x|z)p(z)\,dz \tag{3.2}$$

But this integral is intractable, so we approximate it using variational inference. The encoder learns an approximate posterior $q(z|x)$, and we optimize the following loss (ELBO — Evidence Lower Bound):

$$\mathcal{L}_{\text{VAE}} = \mathbb{E}_{q(z|x)}[\log p(x|z)] - \text{KL}(q(z|x)\,||\,p(z)) \tag{3.3}$$

Here:

- The first term encourages good reconstruction.
- The second term is a regularizer that ensures $q(z|x)$ stays close to the prior $p(z)$ (usually a standard normal).

Pros and Cons

Pros:

- The latent space is structured and continuous — nearby points correspond to similar outputs.
- Efficient inference — one-shot generation without sequential processing.
- VAEs are stable to train, unlike GANs.

Cons:

- Generated images tend to be blurry or less sharp — the pixel-wise reconstruction loss encourages averaging.
- The latent space might not always be rich enough to capture complex details.

VAEs were among the first successful deep generative models, and while they are not always used for final image generation due to quality issues, they are a fundamental building block for many modern hybrid generative models.

3.3 Generative Adversarial Networks (GANs)

Generative Adversarial Networks (GANs) are one of the most influential developments in the field of generative modeling, introduced by Ian Goodfellow in 2014. The core idea is a two-player game between two neural networks: a **generator** and a **discriminator** as shown in Figure 3.4.

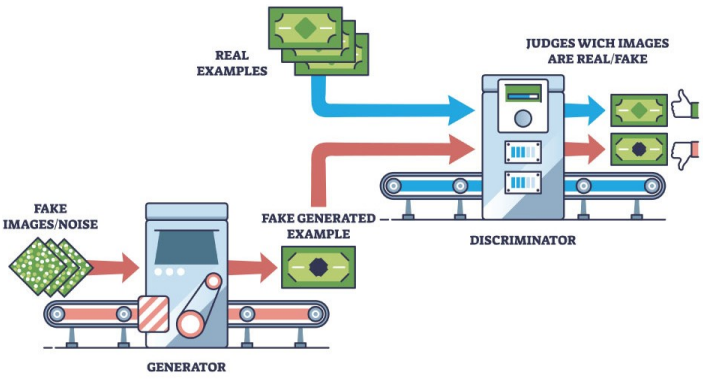


Fig. 3.4: Illustration of the adversarial training loop in a GAN. [7]

The generator network takes in a random noise vector (typically drawn from a standard Gaussian or uniform distribution) and tries to produce data samples (like images) that resemble real data. The discriminator, on the other hand, takes in both real data and fake data generated by the generator and tries to classify them correctly as either real or fake.

These two networks are trained simultaneously in a minimax game:

- The generator tries to fool the discriminator by generating data that looks real.

- The discriminator tries to distinguish between real and fake data.

This dynamic continues until the generator becomes good enough to produce data that the discriminator cannot easily distinguish from real samples. Ideally, at equilibrium, the generator produces data indistinguishable from real data, and the discriminator is maximally confused (assigning a 50-50 probability).

$$\min_G \max_D \; \mathbb{E}_{x\sim p_{\text{data}}}[\log D(x)] + \mathbb{E}_{z\sim p_z}[\log(1 - D(G(z)))] \tag{3.4}$$

Challenges and Instabilities: Although GANs can generate incredibly sharp and realistic images, their training is famously unstable. The minimax game between the generator and discriminator often leads to issues such as:

- **Mode collapse** – the generator finds a few good outputs and keeps repeating them.
- **Non-convergence** – the training oscillates without settling.
- **Vanishing gradients** – the generator receives weak learning signals when the discriminator becomes too strong.

Over the years, a variety of GAN variants have been proposed to address these issues: WGAN, LSGAN, StyleGAN, CycleGAN, etc. Each comes with architectural and loss function tweaks aimed at improving stability and diversity.

3.4 Motivating Diffusion

In the previous sections, we discussed **autoregressive models**, where we trained a single model to predict the next pixel based on the previous context. While the single-model solution solved the problem of having to train 256 different models, we still had to perform 256 sequential forward passes to generate a full 16×16 image — each depending on the result of the previous. This makes autoregressive models inherently slow and not parallelizable during generation.

(a) Idea 1: Block prediction

(b) Idea 2: Spread-out unknowns

(c) Idea 3: Random scattering

Fig. 3.5: Comparison of different inpainting strategies tested before arriving at diffusion.

Let us now consider different approaches to make this process more efficient and possibly parallel:

Idea 1: Block prediction

As shown in the Figure 3.5a, instead of predicting a single pixel, what if we masked out a whole 4×4 block (16 pixels) in the bottom-right corner and asked the model to predict it in one go? *Unfortunately, this approach fails.* Since a block of pixels contains too many plausible combinations, the model is forced to "average" over possibilities, and the result becomes blurry and unrealistic — reminiscent of the "averaging" problem discussed in our introductory motivation.

Idea 2: Spread-out unknowns
What if we still want to predict 16 pixels, but instead of placing them in one block, we spread them evenly across the image? (As in the Figure 3.5b) *This works surprisingly well.* Each unknown pixel is now surrounded by known neighbors, giving the model sufficient context to predict each one with higher confidence. The prediction is no longer blurry and remains interpretable.

Idea 3: Random scattering
Can we go even further and scatter the 16 missing pixels randomly across the image? (Like in the Figure 3.5c) *This too turns out to be quite effective*, as long as the missing pixels do not accidentally cluster together in one region. The key is maintaining contextual integrity for each unknown pixel — ensuring known pixels are sufficiently nearby.

Idea 4: Partial noise instead of complete removal
So far, we've been removing certain pixels completely while keeping the others untouched. But what if we didn't remove complete information from any pixel, and instead removed a small amount of information from *all* pixels? That is, rather than making a few pixels completely unknown, we make *every pixel slightly uncertain.* We then train the model to "denoise" the image — gradually recovering the original image from this noisy version. This is the core idea behind **diffusion models**.

Advantages of Diffusion Models:

- Stable training dynamics (unlike GANs).
- Generates highly detailed and diverse samples.
- Training objective is well-defined and interpretable (predicting noise).

Disadvantages:

- Extremely slow generation — requires hundreds or thousands of steps to generate a single image.
- Computationally expensive during inference.

Despite these limitations, diffusion models have recently shown state-of-the-art performance in high-resolution image generation and are being actively researched for acceleration and optimization [4].

Summary: In this chapter, we learned about various sorts of generative algorithms. Towards the end we motivated about diffusion models from autoregression etc. In this project we first tried our hands on VAEs and GANs, which had their own issues, and finally we made it to work with diffusion models. We have motivated them in this chapter, in the next chapter we would discuss in detail about the specific experiments we performed using diffusion.

Chapter 4

Conditional Bernoulli Diffusion Models

In the previous chapter, we briefly introduced various generative algorithms used in literature, including diffusion models. In this chapter, we delve deeper into diffusion-based approaches, with particular emphasis on the specific variant used in our experiments.

A **conditional** generative model refers to one that is guided by external inputs or context — for instance, a desired temperature or coupling constant in our case. This is analogous to image generation where, if we provide the label "car", the model generates an image of a car and not a bird. Conditional generation offers more control, though it comes at the cost of added model complexity. Nevertheless, **unconditional** models also hold significance, especially in domains where the training data is homogeneous — such as generating human faces — and no explicit conditioning is required.

Furthermore, because the data we work with consists of binary spin configurations (taking values ± 1), the standard Gaussian-based diffusion process is not well-suited. Instead, we employ a **Bernoulli diffusion model**, which better captures the discrete nature of the system and allows us to learn a physically consistent denoising process.

4.1 Denoising Diffusion Probabilistic Models (DDPM)

In this section, we closely follow the influential paper titled *Denoising Diffusion Probabilistic Models* [4], which lays the foundation for most modern diffusion-based generative models. Understanding the ideas presented in this work will provide a solid conceptual framework for diffusion models in general.

The core idea behind DDPM is to gradually corrupt a data sample $\mathbf{x}_0$ (as shown in Figure 4.1) by adding noise to it over a series of T time steps, producing a sequence of increasingly noisy images $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_T$. Then, we train a neural network to reverse this process: at each time step t, the model takes the noisy image $\mathbf{x}_t$ as input and attempts to predict the noise that was added to obtain it. Subtracting this predicted noise from $\mathbf{x}_t$ helps reconstruct a cleaner version of the original image.

The forward (noising) process is defined as follows:

$$\mathbf{x}_t = \sqrt{1-\beta_t}\,\mathbf{x}_{t-1} + \sqrt{\beta_t}\,\boldsymbol{\epsilon}, \quad \boldsymbol{\epsilon} \sim \mathcal{N}(0, \mathbf{I}) \tag{4.1}$$

where β_t is a small positive constant that controls the amount of noise added at each step.

One elegant property of this formulation is that, because the noise is Gaussian and applied incrementally, the entire forward process can be expressed in a closed form. That is, given the

original clean sample $\mathbf{x}_0$, we can directly sample a noisy version at any arbitrary time step t:

$$\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\,\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\,\boldsymbol{\epsilon}, \quad \boldsymbol{\epsilon} \sim \mathcal{N}(0, \mathbf{I}) \tag{4.2}$$

where $\bar{\alpha}_t = \prod_{s=1}^{t}(1-\beta_s)$.

This direct formulation makes training efficient, as we do not need to sample intermediate steps to simulate the noisy $\mathbf{x}_t$ from $\mathbf{x}_0$.

(a) t=0 **(b)** t=5 **(c)** t=20 **(d)** t=40 **(e)** t=100

Fig. 4.1: Evolution of the image as noise is added at each time step

A crucial component of the diffusion process is the **noise schedule** — that is, how much noise β_t is added at each time step t. Typically, less noise is added in the early stages, while stronger noise is introduced in the later steps. This ensures that the early stages preserve more of the structural content of the image, allowing the model to focus on learning finer details, while the later stages help the model deal with more chaotic inputs. Common choices include linear schedules, cosine schedules, or more sophisticated learned schedules depending on the task.

Algorithm 2 Training Algorithm for Gaussian Diffusion Model

repeat
  Sample clean data point: $\mathbf{x}_0 \sim q(\mathbf{x}_0)$
  Randomly choose diffusion step: $t \sim \text{Uniform}(\{1, \ldots, T\})$
  Sample Gaussian noise: $\boldsymbol{\epsilon} \sim \mathcal{N}(0, \mathbf{I})$
  Generate noisy input: $\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}$
  Predict noise with model: $\hat{\boldsymbol{\epsilon}} = \boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)$
  Compute loss: $\mathcal{L} = \|\boldsymbol{\epsilon} - \hat{\boldsymbol{\epsilon}}\|^2$
  Update model parameters θ using gradient descent on $\mathcal{L}$
until convergence

Algorithm 3 Sampling Algorithm from Gaussian Diffusion Model

Initialize noise: $\mathbf{x}_T \sim \mathcal{N}(0, \mathbf{I})$
for $t = T, \ldots, 1$ **do**
  If $t > 1$: sample noise $\mathbf{z} \sim \mathcal{N}(0, \mathbf{I})$, else set $\mathbf{z} = 0$
  Predict noise: $\hat{\boldsymbol{\epsilon}} = \boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)$
  Compute mean: $\boldsymbol{\mu}_t = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\hat{\boldsymbol{\epsilon}}\right)$
  Sample previous step: $\mathbf{x}_{t-1} = \boldsymbol{\mu}_t + \sigma_t \mathbf{z}$
end for
return $\mathbf{x}_0$

4.2 Bernoulli Diffusion Model

The standard DDPM assumes Gaussian noise, which suits continuous domains like natural images. However, spin configurations in our case are binary, taking values in $\{-1, +1\}$. Applying Gaussian noise to such data violates its discrete nature. To address this, we adopt the **Bernoulli Diffusion Model**, as proposed in *BerDiff: Conditional Bernoulli Diffusion Model for Medical Image Segmentation* by Chen et al. [2].

In this model, instead of adding Gaussian noise, a forward diffusion process is defined by randomly flipping bits according to a time-dependent Bernoulli distribution. Let $\mathbf{y}_0 \in \{-1, +1\}^d$ represent the original binary configuration (e.g., spin system), and $\bar{\alpha}_t \in [0, 1]$ be the cumulative noise schedule.

Algorithm 4 Training Algorithm for Bernoulli Diffusion Model

repeat
  Sample data pair $(\mathbf{x}_0, \mathbf{y}_0) \sim q(\mathbf{x}_0, \mathbf{y}_0)$
  Sample timestep $t \sim \text{Uniform}(\{1, \ldots, T\})$
  Compute noise level: $\bar{\alpha}_t = \prod_{s=1}^{t}(1 - \beta_s)$
  Sample Bernoulli noise: $\boldsymbol{\epsilon} \sim \mathcal{B}((1 - \bar{\alpha}_t)/2)$
  Corrupt binary target: $\mathbf{y}_t = \mathbf{y}_0 \oplus \boldsymbol{\epsilon}$
  Predict noise: $\hat{\boldsymbol{\epsilon}} = \boldsymbol{\epsilon}_\theta(\mathbf{y}_t, t, \mathbf{x}_0)$
  Compute loss: $\mathcal{L} = \text{BCE}(\hat{\boldsymbol{\epsilon}}, \boldsymbol{\epsilon})$
  Update θ via gradient descent on $\mathcal{L}$
until convergence

Algorithm 5 Sampling Algorithm from Bernoulli Diffusion Model

Initialize $\mathbf{y}_T \sim \mathcal{B}(0.5)$
for $t = T$ to 1 **do**
  Predict noise: $\hat{\boldsymbol{\epsilon}} = \boldsymbol{\epsilon}_\theta(\mathbf{y}_t, t, \mathbf{x}_0)$
  Compute Bernoulli mean: $\boldsymbol{\mu}_t = F_C(\mathbf{y}_t, t, \hat{\boldsymbol{\epsilon}})$
  Sample next state: $\mathbf{y}_{t-1} \sim \mathcal{B}(\boldsymbol{\mu}_t)$
end for
return $\mathbf{y}_0$

Forward Process

At each timestep t, we sample a noise mask $\boldsymbol{\epsilon} \sim \mathcal{B}((1 - \bar{\alpha}_t)/2)$ where $\mathcal{B}(p)$ denotes the Bernoulli distribution. The corrupted configuration $\mathbf{y}_t$ is obtained via element-wise XOR (denoted $\oplus$):

$$\mathbf{y}_t = \mathbf{y}_0 \oplus \boldsymbol{\epsilon} \tag{4.3}$$

This ensures that each bit in $\mathbf{y}_0$ is independently flipped with probability $(1 - \bar{\alpha}_t)/2$, preserving the binary structure.

Reverse Process (Sampling)

To denoise and sample from the model, a learned network $\boldsymbol{\epsilon}_\theta(\mathbf{y}_t, t, \mathbf{x})$ estimates the original noise. From this, we compute a mean function $\boldsymbol{\mu}(\mathbf{y}_t, t, \mathbf{x})$ using a classifier function F_C:

$$\boldsymbol{\mu}(\mathbf{y}_t, t, \mathbf{x}) = F_C(\mathbf{y}_t, t, \boldsymbol{\epsilon}_\theta(\mathbf{y}_t, t, \mathbf{x})) \tag{4.4}$$

The sample at step $t-1$ is then drawn as:

$$\mathbf{y}_{t-1} \sim \mathcal{B}(\mathbf{y}_{t-1}; \boldsymbol{\mu}(\mathbf{y}_t, t, \mathbf{x})) \tag{4.5}$$

In the DDIM-style deterministic variant, the transition is directly computed using $\bar{\alpha}_t$ and the model-predicted noise.

4.3 Model Architecture and Intuition

After numerous failed attempts with more complex architectures, we designed a custom lightweight model that not only performs well but also fits within the strict constraints of a CPU-only environment. The base version of this model has just **32 parameters**, while a slightly more powerful version has around **50 parameters**, which is negligible compared to the millions in standard deep learning models.

The core task here is closely related to modeling something akin to the Ising Hamiltonian, which governs the system dynamics in traditional Metropolis algorithms. The Hamiltonian is given by:

$$\mathcal{H} = -J_1 \sum_{\langle i,j \rangle} S_i S_j - J_2 \sum_{\langle\langle i,j \rangle\rangle} S_i S_j - \cdots - h \sum_i S_i. \tag{4.6}$$

This expression clearly resembles a second-degree polynomial of the spin variables S_i and S_j. The model is crafted based on this observation as shown in Figure 4.2.

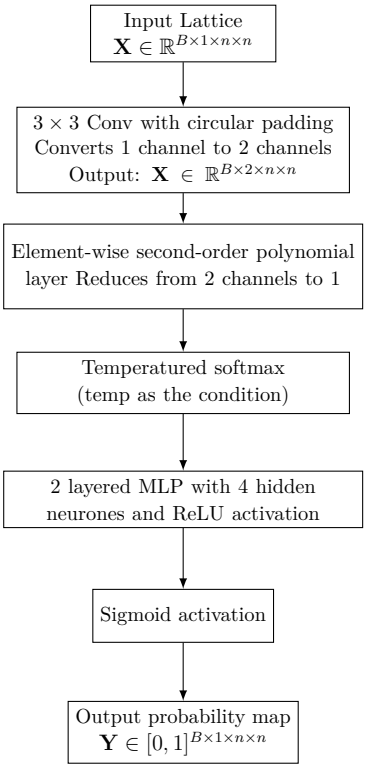


Fig. 4.2: Model Architecture

- The **first convolution layer** uses a 3×3 kernel with circular padding (mimicking PBC) and outputs two channels. One channel is meant to capture the values of S_i (spin information), and the other accumulates $J \cdot S_i$ (interaction strength-weighted spins). We used this because we were working with just 2nd nearest neighbors, but if we were working with more distant correlations, there are two ways to approach that problem:
  - use a bigger kernel, maybe 5×5 or 7×7

  - stack more 3×3 convolution layers

- Next, we apply an **element-wise second-order polynomial transformation**:

$$f(x_1, x_2) = b + w_1 x_1 + w_2 x_2 + w_3 x_1^2 + w_4 x_1 x_2 + w_5 x_2^2, \tag{4.7}$$

  where x_1, x_2 are the two per-pixel values from the two channels. This effectively acts like a small Hamiltonian approximation done locally on the lattice. we can see that the w_4 term takes care of the J_1 and J_2 terms while the w_1 or the w_2 term takes care of the h term. the rest of the ws are expected to be 0 here, but might be used for different hamiltonians.

- The output logits are then passed through a **temperatured softmax**, which resembles the Boltzmann distribution:

$$p_i = \frac{e^{z_i/T}}{\sum_j e^{z_j/T}}, \tag{4.8}$$

  where T is the temperature (conditioning input), and z_i are the logits. This turns the energy-like quantity into a meaningful probability of flipping.

- These probabilities are then passed through a small **MLP with two layers**, responsible for learning how to decide which spins to flip based on the softmax scores. The MLP is defined as:

$$\text{MLP}(x) = W_2 \cdot \max(W_1 x + b_1, 0) + b_2 \tag{4.9}$$

  where $W_1 \in \mathbb{R}^{4\times1}$, $W_2 \in \mathbb{R}^{1\times4}$, and the total hidden width is 4 neurons.

- Finally, a **sigmoid activation** is applied to output a value in $[0, 1]$ for each site:

$$\sigma(x) = \frac{1}{1 + e^{-x}}. \tag{4.10}$$

  This output is interpreted as the final decision — flip or not flip.

This minimal architecture is optimized to mimic key physical principles while maintaining extremely low parameter count and computational footprint — enabling efficient training and inference even on basic hardware.

4.4 Time Complexity Analysis

In the **classical case**, the proposed machine learning method does not offer significant improvements in asymptotic speed compared to traditional methods. The Metropolis Monte Carlo algorithm typically exhibits a time complexity of $\mathcal{O}(n \log n)$, where n is the number of spins in the lattice. Our machine learning-based diffusion model also operates with a time complexity of approximately $\mathcal{O}(n \log n)$, primarily due to the sequential nature of the denoising steps and the overall data generation pipeline.

However, it is important to note that the machine learning approach is **slower by a constant factor** when compared to Monte Carlo methods. This overhead is attributed to the model's generality — it is designed to accommodate a wide range of Hamiltonians and system parameters, resulting in some inherent redundancy and computational cost.

In contrast, for the **semi-classical or quantum cases**, the improvement is more pronounced. Although both the Monte Carlo and machine learning approaches still require $\mathcal{O}(n \log n)$ sampling steps, the computational cost per step differs significantly:

- In the Monte Carlo method, each step involves evaluating the full energy of a quantum Hamiltonian, which can be computationally intensive. This leads to a per-step complexity of $\mathcal{O}(n^3)$ due to matrix operations such as diagonalization or trace evaluation.
- In the machine learning approach, the trained model infers the configuration in **linear time**, i.e., $\mathcal{O}(n)$ per step. This is because the expensive Hamiltonian evaluations are learned and abstracted by the network during training, enabling rapid sampling during inference.

Therefore, while the asymptotic step count remains the same, the machine learning-based diffusion method offers a **significant reduction in per-step cost** in the semi-classical and quantum regimes, leading to a much faster overall generation process.

Summary: In this chapter, learned in detail the theory behind our experiment which worked (there were 15-20 other failed experiments which were omitted.

Chapter 5

Methodology and Results

This chapter presents the methodology followed in our experiments, along with the results obtained using the Conditional Bernoulli Diffusion Model. We begin by discussing the technical setup used for running the simulations and training the models.

5.1 Technical Setup

Most of the implementation was done in **Python**, utilizing standard libraries such as `NumPy`, `PyTorch`, and `Matplotlib` for numerical computations, model development, and visualization respectively. However, for generating lattice configurations using the Metropolis algorithm, we opted for a **C-based implementation**, which was then integrated into the Python workflow using the native `ctypes` library.

This design choice was driven by performance considerations: the Metropolis algorithm required running an extremely large number of iterations (in the order of 10^7 to 10^9), where each iteration was computationally trivial but involved intensive non-parallelizable looping. Python's inherent slowness in executing such loops made it unsuitable for this part of the pipeline, while C offered significant (100x to 300x) speed improvements due to its low-level efficiency.

All simulations and training were conducted on a machine equipped with dual **Intel Xeon CPUs** (2x), providing a total of **32 physical cores**, allowing for parallel execution where applicable. Even the Machine Learning tasks were performed on the same system and **NO GPU was used**.

5.2 Classical Ising Model Machine Learning Simulation

As discussed in Chapter 4, we used the custom model (described in Figure 4.2) and trained it following the Bernoulli diffusion training algorithm (Algorithm 4). Initially, we assumed that a larger number of time steps would make the learning task easier, so we began with $T = 500$–1000, inspired by the original diffusion work [4]. However, we soon observed that this led to almost no visible changes between successive time steps, making it difficult for the model to learn anything meaningful.

We used a linear schedule for the noise parameter β_t, increasing from 0.01 to 0.4 over the 10 time steps. The training objective was to minimize the **Mean Squared Error (MSE)** between the predicted and actual noise:

$$\mathcal{L}_{\text{MSE}} = \mathbb{E}_{x_0,t,\epsilon}\left[\|\epsilon - \hat{\epsilon}_\theta(x_t, t, T)\|^2\right],$$

where $\hat{\epsilon}_\theta$ denotes the model's prediction of the noise.

The model was trained using the **Adam optimizer** with mini-batches of size 64, across 2500 such batches. Each batch consisted of 64 data quadruples of the form:

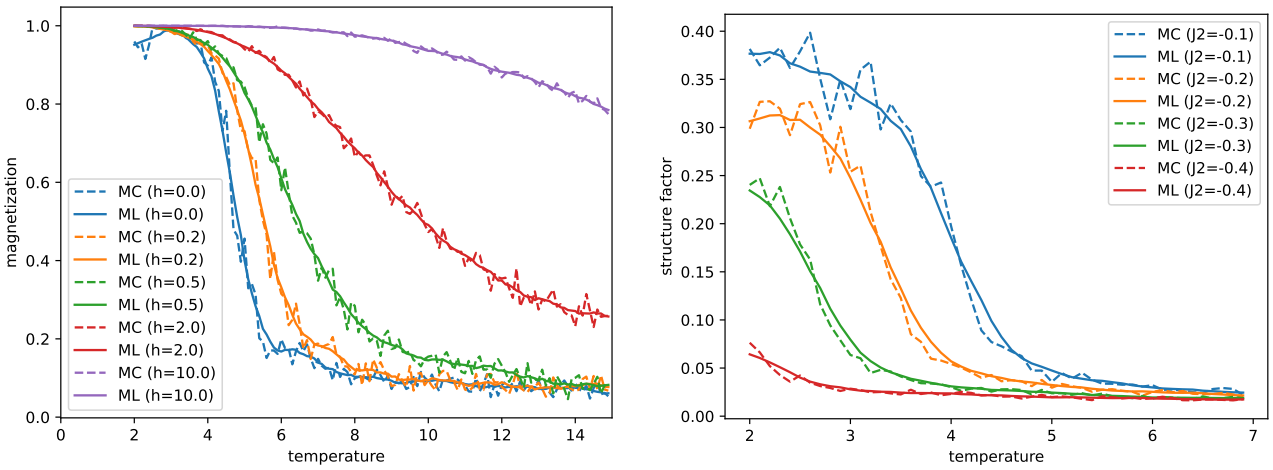
1. The final lattice configuration at some temperature T, where $2.0 < T < 7.0$ (in units of $k_B T/|J|$, transition temperature was about 5 units)
2. The corresponding temperature T,
3. The diffusion time step t,
4. The noisy version of the final lattice at time t.

To construct each batch of 64 quadruples:

- We simulated 16 unique lattice configurations, each initialized with a spin bias $p \in [0, 1]$, meaning a fraction p of the spins were up $(+1)$ and $(1 - p)$ were down (-1), where $p \sim \mathcal{U}(0, 1)$.
- Each simulation was performed at a randomly chosen temperature $T \sim \mathcal{U}(2.0, 7.0)$.
- For each of the 16 simulated lattices, we randomly selected 4 time steps $t \in \{1, \ldots, 10\}$, and used the forward process described in Equation 4.3 to generate noisy versions at those time steps.

Thus, from 16 simulations and 4 time steps per simulation, we obtained a batch of $16 \times 4 = 64$ data points.

Here in Figure 5.1 we can see how the Machine Learning results quite match with the Monte Carlo results.



(a) J1=1, J2=0, h=variable

(b) J1=-1, J2=variable, h=0

Fig. 5.1: Machine Learning (ML) banchmark against Monte Carlo (MC) results. We can see that the machine learning results quite resembles the MC results. Following Hamiltonian ??.

5.3 Approaching the Semi-Classical / Quantum Problem

By this stage, we had already completed a substantial portion of the work required for our MSc thesis. However, with around a week of time remaining, my supervisor encouraged me to take on a more challenging extension — to see whether the same techniques could be applied to systems where the energy is not easily computed via nearest-neighbour interactions.

In **quantum systems**, the spin variables become quantum operators, typically represented by Pauli matrices. As a result, the Hamiltonian — which is a function of these spin variables — becomes a matrix rather than a scalar. Consequently, computing the system's energy involves **diagonalizing** this matrix and summing a subset of its eigenvalues.

We chose to work with the **Double Exchange Model**, where the Hamiltonian is given in Equation 1.7. In this model, classical spins residing in the lower orbitals influence the transition probabilities of electrons in the upper orbitals. Our task was to generate plausible configurations of these lower-orbital spins at a given temperature, consistent with the Hamiltonian.

In traditional Monte Carlo approaches, one starts with a random configuration of classical spins. For a fixed filling fraction n, the Hamiltonian is constructed (with shape $\mathbb{R}^{L\times L}$, where L is the number of lattice sites), diagonalized (an $\mathcal{O}(L^3)$ operation), and the energy is computed as the sum of the n lowest eigenvalues.

Initially, we attempted to apply our full generative model pipeline — training a large MLP to learn this mapping. However, the model failed to learn effectively. So, we simplified the problem: **Can the neural network at least learn to compute the Hamiltonian and perform the eigenvalue decomposition for this specific form of Hamiltonian?**

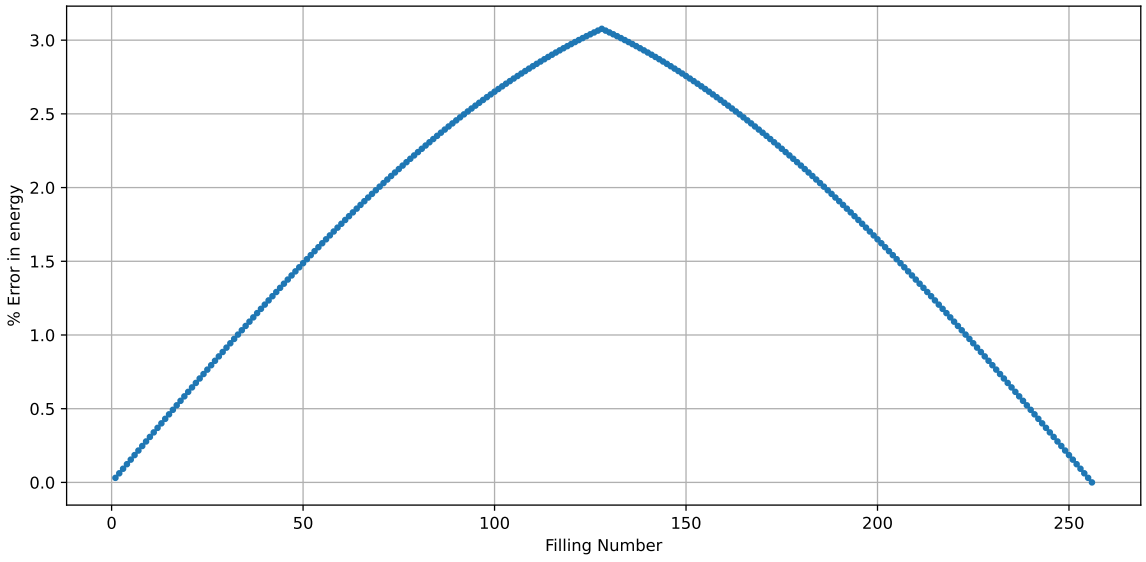


Fig. 5.2: Percentage Error in trying to learn energy given the hamiltonian for different filling fractions for the double exchange model as in Equation 1.7

There were a few reasons to remain optimistic:

- We were not asking the model to generalize to all matrices — just those arising from this specific form of the double exchange Hamiltonian.

- The Hamiltonian matrices in our case are typically **sparse**, which often makes learning easier for neural networks.
- From an information-theoretic standpoint, computing all L eigenvalues requires $\mathcal{O}(L^3)$ operations. However, since we only need the sum of the lowest n eigenvalues (and not their individual values), there is some compression of information. In the special case when $n = L$ (full filling), the energy is simply the **trace** of the Hamiltonian — the sum of its diagonal elements — which is an $\mathcal{O}(L)$ operation. This gave us hope that at least an approximation may be learnable.

We proceeded with this simplified task, and although the neural network did not produce perfect predictions, the results were quite encouraging. The **error was bounded within 3%**, and the model performed especially well at extreme filling fractions (either very low or very high). The errors were slightly larger near half-filling, but still within acceptable limits. The results are illustrated in Figure 5.2, where the prediction error is plotted across different filling fractions.

This final leg of the project — applying deep learning to quantum systems — felt like stepping into something profound, just as time began to run out. While the initial results were intriguing, the exploration remained incomplete. A few key points worth highlighting:

- **Naive Model & Loss Function:** The neural network architecture used was fairly basic, and the loss function was simplistic. This was more a result of time pressure than design choice.
- **Lack of Analysis:** There was hardly any time to analyze why the model performed better in certain regimes (like very high or low filling fractions) and worse near half-filling. A detailed error analysis might have revealed critical patterns or physical insights.
- **Unexplored Avenues:** We didn't get to experiment with better inductive biases, more expressive models, or physically-informed loss functions. All of these could have drastically improved performance.
- **Optimism for the Future:** Despite the rushed nature of this part, the model performed surprisingly well — with less than 3% error even in difficult regimes. This leaves me genuinely hopeful. With a bit more time and attention, I believe this approach could unlock powerful new ways to handle quantum systems.

Summary: In this chapter, we discussed the results and methodology of our experiments performed.

Bibliography

[1] Saientan Bag, Ayush Jha, and Florian Müller-Plathe. Machine learning assisted monte carlo simulation: Efficient overlap determination for nonspherical hard bodies. *Advanced Theory and Simulations*, 6(11):2300520, 2023. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/adts.202300520, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/adts.202300520, doi:10.1002/adts.202300520.

[2] Tao Chen, Chenhui Wang, and Hongming Shan. *BerDiff: Conditional Bernoulli Diffusion Model for Medical Image Segmentation*, page 491–501. Springer Nature Switzerland, 2023. URL: http://dx.doi.org/10.1007/978-3-031-43901-8_47, doi:10.1007/978-3-031-43901-8_47.

[3] Simone Ciarella, Jeanne Trinquier, Martin Weigt, and Francesco Zamponi. Machine-learning-assisted monte carlo fails at sampling computationally hard problems. *Machine Learning: Science and Technology*, 4(1):010501, March 2023. URL: http://dx.doi.org/10.1088/2632-2153/acbe91, doi:10.1088/2632-2153/acbe91.

[4] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models, 2020. URL: https://arxiv.org/abs/2006.11239, arXiv:2006.11239.

[5] Yurii A Izyumov and Yu N Skryabin. Double exchange model and the unique properties of the manganites. *Physics-Uspekhi*, 44(2):109, feb 2001. URL: https://dx.doi.org/10.1070/PU2001v044n02ABEH000840, doi:10.1070/PU2001v044n02ABEH000840.

[6] D.P. Landau and K. Binder. *A Guide to Monte Carlo Simulations in Statistical Physics*. Cambridge University Press, 2005. URL: https://books.google.co.in/books?id=11tun6Y5tOAC.

[7] P&G Professional. What is generative adversarial network? types. *P&G Professional Blog*, 2023. Accessed: 2025-04-22. URL: https://pg-p.ctme.caltech.edu/blog/ai-ml/what-is-generative-adversarial-network-types.

Appendix A

Monte Carlo Algorithm

A Monte Carlo simulation is a computational algorithm that uses repeated random sampling to obtain numerical results. These simulations can be used to solve problems in various fields, including optimization, numerical integration, and generating draws from a probability distribution.

One of the key characteristics of Monte Carlo simulations is that they are often multiple orders faster to run than corresponding deterministic algorithms. But, this speed comes with a catch: they might fail with a certain probability.

However, this failure related limitation can be mitigated by running the simulation multiple times. If we consider an algorithm with a success probability of 10% when run once, the cumulative success probability after running n times becomes:

$$\begin{aligned} P(\text{success}) &= 1 - (1-p)^n \\ &= 1 - (1-0.1)^{20} \\ &= 1 - (0.9)^{20} \\ &\approx 0.88 \end{aligned}$$

where p is the success probability of running once, and n is the number of times the algorithm is run. As we can see, by running the algorithm just 20 times, we can increase the success probability from 10% to almost 88%.

Another similar type of algorithm is the *Las Vegas algorithm*. The difference between Monte Carlo and Las Vegas algorithms lies in the tradeoff. In Monte Carlo we fix the time and allow failures with some probability, while in Las Vegas we fix the probability of success to 1 (always successful) and calculate the expected time taken to reach that.

A.1 Monte Carlo Simulation

Monte Carlo methods use repeated random sampling to solve complex problems that are difficult or impossible to solve deterministically. These algorithms rely on randomness to obtain numerical results and are commonly applied to three main areas: optimization, numerical integration, and generating random samples from probability distributions. Additionally, Monte Carlo methods can be used to model uncertain systems, such as assessing the risk of a nuclear power plant failure, by simulating various scenarios and outcomes. One

classic example of a Monte Carlo simulation is approximating the value of π shown in Algorithm 6.

Algorithm 6 Estimating π using Monte Carlo Method

Initialize `counter` $\leftarrow 0$ $i = 1$ to n Generate two independent random variables $x, y \sim \mathcal{U}(0, 1)$ $x^2 + y^2 < 1$ Increment `counter` by 1 $\pi_{\text{estimate}} \leftarrow \frac{4 \times \texttt{counter}}{n}$

As n approaches infinity, this ratio converges to π. This may seem surprising at first, but it can be understood by considering the probability of a randomly chosen point (x, y) falling inside the quarter circle. Since the area of the quarter circle is $\frac{\pi}{4}$ and the total area of the square is 1, the probability of a point falling inside the quarter circle is $\frac{\pi}{4}$. By repeating this process many times, we can estimate this probability and multiply it by 4 to get an estimate of π. This is demonstrated in figure A.1.

A.2 Importance Sampling

Importance sampling is a technique used in Monte Carlo simulations to efficiently sample from complex distributions. When directly sampling from the target distribution $f(x)$ is challenging, importance sampling introduces an auxiliary proposal distribution $g(x)$, which is easier to sample from and has a similar shape to $f(x)$. By carefully choosing $g(x)$ based on the properties of the system, importance sampling focuses on the most relevant regions of the state space. This increases efficiency when $f(x)$ has a small support or exhibits rare events, as traditional Monte Carlo methods may require an impractically large number of samples to accurately capture these features. Importance sampling reduces computational cost while maintaining accuracy by weighting samples with the likelihood ratio $f(x)/g(x)$.

A.3 Markov Chains

A Markov Chain is a mathematical system that undergoes transitions from one state to another according to certain probabilistic rules. The future state of the system depends only on its current state, and not on any of its past states. A Markov Chain consists of a set of possible states, transition probabilities between these states, and a transition matrix that represents these probabilities.

The memorylessness property can be mathematically expressed as

$$P(X_t = j | X_{t-1} = i, X_{t-2}, ..., X_0) = P(X_t = j | X_{t-1} = i)$$

where X_t is the state at time t. The time homogeneity property implies that this probability does not depend on t.

A.3.1 Features of the Markov Chains

- **Transition Matrix**: A Markov Chain is defined by a transition matrix π, where π_{ij} represents the probability of transitioning from state i to state j.
- **Irreducibility**: If it is possible to get from any state to any other state, either directly or indirectly, the Markov Chain is said to be irreducible.

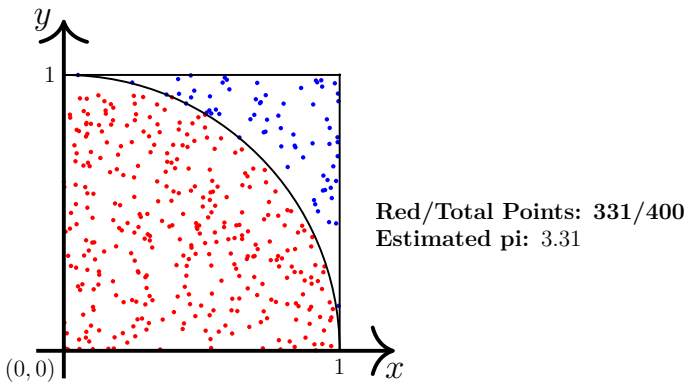


Fig. A.1: Estimating π using Monte Carlo simulation. The red points fall inside the quarter circle, while the blue points fall outside. The ratio of red points to total points is multiplied by 4 to estimate π.

- **Periodicity**: If the greatest common divisor of the lengths of all possible cycles in the chain is 1, the chain is said to be aperiodic.
- **Ergodicity**: A Markov Chain that is both irreducible and aperiodic is said to be ergodic.
- **Stationary Distribution**: A distribution π is said to be stationary if $\pi = P^T \cdot \pi$, where P is the transition matrix.
- **Master Equation**: The master equation describes how the probability of being in each state changes over time. It is given by:

$$\frac{\partial P(x,t)}{\partial t} = \sum_y [P(y,t)\pi_{yx} - P(x,t)\pi_{xy}] \tag{A.1}$$

  where $P(x,t)$ is the probability of being in state x at time t, and π_{xy} is the transition probability from state x to state y.

A.4 Markov Chain Monte Carlo (MCMC) Simulation

Markov Chain Monte Carlo (MCMC) simulation is a technique for studying the properties of a probability distribution by constructing a Markov Chain with an equilibrium distribution that matches the target distribution. By running this chain, we can generate samples from the target distribution and estimate its properties. MCMC methods are particularly useful in Bayesian inference and have been widely applied in statistics, machine learning, and physics to study complex or high-dimensional distributions.