Bachelor's Degree in Mathematics

Bachelor's Thesis

Generative adversarial networks applied to Science

Miguel Tajada Ferrer

supervised by

Dr. Luis Martín Moreno

Faculty of Sciences. University of Zaragoza
Department of Condensed Matter
Academic Year 2022-2023

Resumen

Estamos ante una de las mayores revoluciones vividas en las últimas décadas. La irrupción de la inteligencia artificial abre un nuevo paradigma que transformará la sociedad en la que vivimos. Tras los lanzamientos realizados por la empresa OpenAI durante los últimos años, los modelos generativos se han abierto paso como unos de los más relevantes en el ámbito de redes neuronales. Este proyecto nace con la intención de entender los fundamentos de las redes neuronales y cómo funcionan las Redes Generativas Antagónicas (GANs), una de las arquitecturas de redes neuronales más usadas en el ámbito generativo.

El presente trabajo se centra en realizar un análisis profundo y detallado sobre estas redes, desglosando meticulosamente en cada sección las principales ideas tanto teóricas como prácticas, facilitando una comprensión completa de la materia y poniendo de manifiesto el potencial de esta arquitectura en escenarios del mundo real. En una primera instancia, se asientan las bases teóricas. En la sección introductoria a las redes neuronales, se desgranan tres elementos cruciales: la arquitectura subyacente que da forma a estas estructuras computacionales, el papel vital del descenso del gradiente en la optimización, y el uso estratégico de las funciones de activación para modelar complejidades intrínsecas. Seguidamente, se particulariza al caso de las GAN, donde se exponen de forma detallada las dinámicas de entrenamiento. Adicionalmente, se desarrollan mejoras como las DCGAN y WGAN que serán empleadas en la sección práctica para obtener resultados muy prometedores. Concretamente, se discuten tres problemas de tres ramas diferentes del conocimiento.

En la primera aplicación detallada en el capítulo 4, se explora en profundidad la utilización de las GANs en el ámbito de la generación de secuencias de ADN, centrando la atención especialmente en la segmentación de secuencias del COVID. Esta sección inicia con la construcción y visualización del conjunto de datos que será utilizado, permitiendo una comprensión clara y detallada de la estructura de los datos con los que se está trabajando. Posteriormente, se sumerge en las arquitecturas específicas de los generadores y discriminadores, explicando de qué manera estos elementos clave operan y colaboran para producir resultados significativos. A continuación, el texto desentraña el procedimiento de entrenamiento de la red, dando una visión detallada del proceso que ayuda a configurar y afinar la red para lograr los resultados deseados. Finalmente, se realiza un análisis de los resultados obtenidos, destacando tanto los éxitos como las áreas potenciales de mejora.

La segunda sección de la sección práctica detalla cómo a partir de una GAN es posible generar imágenes. La principal novedad es la incorporación de redes convolucionales, una estrategia que ha demostrado ser notablemente eficaz en el tratamiento y análisis de datos bidimensionales. Tras explicar cómo generar un *dataset* con imágenes con cierta estructura pero asumibles para un modelo sencillo, se describe la novedosa arquitectura de las redes, tanto del generador como del discriminador. En esta ocasión destaca la posibilidad de visualizar cómo el generador va mejorando las imágenes mostradas a lo largo del proceso de entrenamiento, notándose de forma evidente el aprendizaje.

Finalmente, la última aplicación detalla la exploración de una simulación del modelo 2D Ising utilizando Wasserstein GANs (WGANs), una clase avanzada de GANs que supera desafíos encontrados en las GANs tradicionales, principalmente en lo que respecta a la convergencia y estabilidad durante el entrenamiento. El modelo de Ising trata de describir fenómenos de transiciones de fase, especialmente en el contexto de sistemas ferromagnéticos, donde las interacciones entre partículas adyacentes (o espines) se modelan en una red bidimensional. La construcción detallada y la visualización del conjunto de datos marcan el inicio de esta sección, donde se hace uso de los conocimientos adquiridos en física estadística. Seguidamente, tras describir la arquitectura del generador y discriminador, se subrayan los

matices intrínsecos a las WGAN, así como las modificaciones que han de hacerse para mejorar su convergencia. Finalizando la sección, se lleva a cabo una evaluación técnica de los resultados, demostrando con métricas cuantitativas la eficacia de las WGANs en la simulación del modelo 2D Ising; concretamente, con el estudio de la magnetización en función de la temperatura. Por último, se destaca la inclusión de estrategias para futuras mejoras, con especial énfasis en la aplicación de una búsqueda *grid search*, una técnica que busca encontrar los hiperparámetros óptimos para el modelo tratado.

Como conclusión de este proyecto, se destaca la exploración a fondo las Redes Generativas Antagónicas (GANs), desgranando tanto sus aspectos fundamentales como las recientes mejoras técnicas que amplían su eficacia y campo de aplicación. Iniciando con una clara introducción a las redes neuronales, hemos asentado una base firme para comprender las evoluciones recientes en el ámbito de las GANs, incorporando avances sustanciales como las GANs convolucionales (DCGANs) y las WGANs. Cada subsección de la investigación no solo detalla las metodologías aplicadas y los desafíos encontrados, sino que también proporciona una evaluación detallada de los resultados logrados, uniendo con eficacia teoría y práctica en el análisis de la aplicación de estas redes en situaciones prácticas. Los resultados son prometedores, especialmente en la generación de imágenes detalladas y la replicación de comportamientos macroscópicos, tal como se observa en la sección del modelo de Ising, donde demuestran el potencial significativo de las GANs. El reto principal ha sido el hallar la combinación ideal de hiperparámetros, lo cual sugiere futuras líneas de trabajo, donde se exploren técnicas más avanzadas para afinar aún más el rendimiento, superando las limitaciones actuales debido a los altos requerimientos computacionales.

Contents

1 Introduction.

In recent years, the technological landcape has witnessed a remarkable breakthrough in the field of artificial intelligence, with neural networks and Generative Adversarial Networks (GANs) emerging as central components in this transformation. Originating from the intent to replicate the neural structures observed in biological brains, these mathematical models have enabled machines to learn, reason, and predict. Their layers of interconnected nodes, known as neurons, adjust weights based on input data, allowing them to recognize patterns, classify information, and even make decisions.

This past year, with the recent launch of the *state-of-the-art* models developed by OpenAI, the significance of generative models has skyrocketed. Generative models, including GANs, are particularly relevant in the current technological landscape because they have proven to be an outstanding tool for producing high-quality, realistic content across various industries, including the arts and scientific research. This document seeks to demystify the complexities of these systems, detailing their structure, functionalities, and emerging applications in contemporary fields.

This project offers a detailed study of neural networks and GANs, providing readers with a deep insight into their fundamental concepts and functionalities. Initially, it introduces the basics of neural network architecture and training, eventually leading to the intricate dynamics of GANs. This document goes beyond theoretical concepts, exploring the practical applications and analyses of GANs in several modern domains such as DNA sequence generation, image creation, and the simulation of a well-known problem in statistical mechanics. Each section is meticulously crafted to furnish both theoretical and practical insights, facilitating a comprehensive understanding of the subject and demonstrating the potential of these networks in real-world scenarios.

In this project, we emphasize the replicability of all presented methodologies. The accompanying annexes contain the comprehensive Python code utilized throughout this research, predominantly developed using the TensorFlow library for its facilitative features in designing and training neural network architectures. It is noteworthy to mention that while the majority of the code is developed in-house, the segment pertaining to the Ising model is adapted from a pre-existing network initially designed for analyzing a separate cosmic ray problem [1], though significant modifications have been made to tailor it to the specific requirements of the Ising model analysis. This initiative demonstrates our commitment to advancing the field through collaboration and resource sharing.

2 Introduction to Neural Networks.

Neural networks, often referred to as artificial neural networks (ANNs) to distinguish them from biological neural networks, are a subset of machine learning models inspired by the way neurons in the human brain process information. In this section we will cover its basic structure and some of the most popular algorithms used to conduct training or *learning*.

2.1 Neural Network Architecture.

When first diving into machine learning, particularly deep learning, the term *Neural Networks* might seem intimidating. Essentially, they are mathematical graphs composed of nodes, called neurons, interconnected by links. To better grasp this concept, it's helpful to examine biological neurons. A neuron is the fundamental unit of the nervous system, responsible for receiving, processing, and transmitting information. Simplified, a neuron takes in information through its dendrites. If the signal is potent enough, the neuron then transmits this message down its axon, eventually passing it on to other neurons via the axon terminals and across synapses.

In ANNs the neurons are the nodes of the graph and the basic computational unit. As in biological neural networks, each neuron receives one or more inputs, processes them, and produces an output. The first type of artificial neuron that we will look at are *perceptrons*. A perceptron [2] takes several binary inputs (x_1, x_2, x_3, ...) and produces a single binary output:

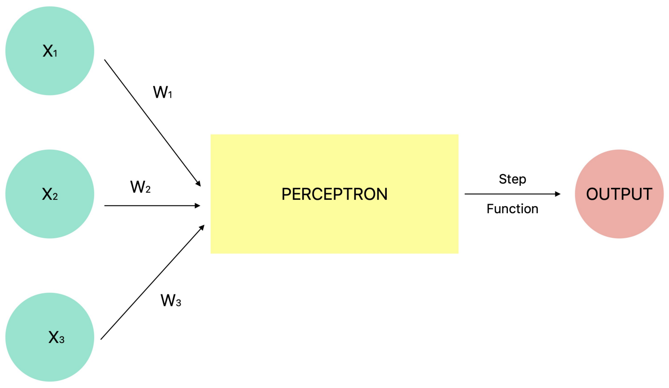


Figure 1: *Visualization of a perceptron.*

In order to compute the output three elements have to be taken into account: the inputs, the weights of each input and the activation function of the neuron. The weights (ω_1, ω_2, ...) are real numbers representing the *importance* of the respective inputs to the output. On the other hand, the activation function for perceptrons will be the *step function*. For clarity, the notation will be $w \cdot x = \sum_j w_j x_j$, where b is the neuron's bias or threshold, x_j represents a given input and w_j denotes its corresponding weight. Therefore,

$$\text{output} = \begin{cases} 0 & \text{if } wx + b \leq 0 \\ 1 & \text{if } wx + b > 0 \end{cases} \tag{2.1}$$

A way that the perceptron can be thought of is a device that makes decisions by weighing up evidence. As an example, suppose that you're deciding whether to order a pizza for dinner. You might base your decision on three factors, let x_1, x_2, and x_3 represent:

- Hunger level
- Amount of money
- Weekend (1 if it's the weekend, 0 if not)

You assign weights: $w_1 = 4$ for hunger, $w_2 = 2$ for money, and $w_3 = 1$ for the weekend. In this case, a higher hunger level matters most to you, followed by the availability of money and then whether it's a weekend. The threshold is set at $b = 6$. If the weighted sum of the factors $4 * x1 + 2 * x2 + 1 * x3$ is greater than or equal to the threshold of 6, you decide to order pizza (output 1), indicating that you're motivated to eat. Otherwise, you decide not to order pizza (output 0), suggesting that you might opt for something else to eat. This perceptron model reflects your decision-making process based on hunger, affordability, and whether it's a weekend, demonstrating how weights and thresholds can shape the outcome.

As detailed in section 2.2, training involves adjusting the weights and biases to ensure that the outputs align with the given data. Ideally, a minor change in weight should result in only a slight change in the network's output. However, with perceptrons, a small change can flip the output from 0 to 1. This issue can be addressed by introducing different activation functions. One of the most commonly used neurons is the *sigmoid neuron*, whose inputs and outputs can assume real values between 0 and 1. Its activation function is given by

$$f(z) = \frac{1}{1 + e^{-z}} \tag{2.2}$$

where z represents the weighted sum of the inputs, defined as $z = \sum_j w_j x_j + b$, b is the neuron's bias or threshold, x_j represents a given input and w_j denotes its corresponding weight.

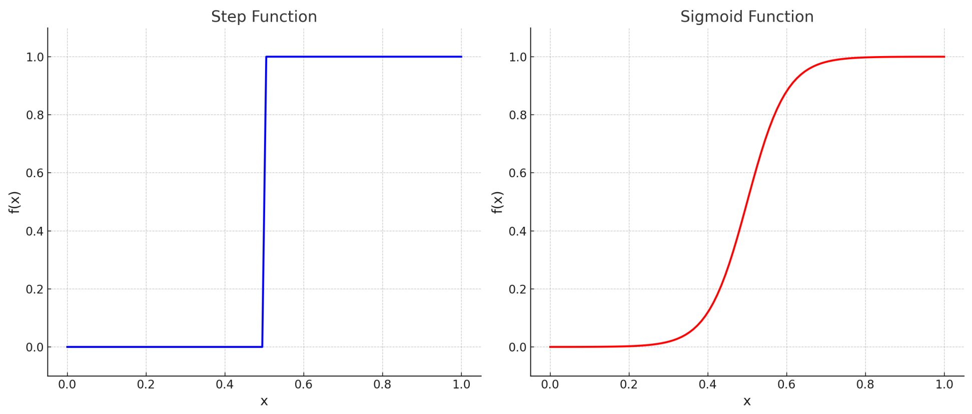


Figure 2: *Comparison of the step and sigmoid function plotted over the range [0,1].*

As illustrated in figure 2, the sigmoid function resembles a smoothed version of the step function. This smoothness allows the sigmoid to be differentiable, a key property necessary for gradient-based optimization methods used in training neural networks. While various activation functions are suited for different needs, Table 1 displays some of the most prevalent ones. It is worth noting that while the *linear*, *sigmoid*, and *softmax* activation functions are often employed in the output layer, the *ReLU* (Rectified Linear Unit) and its variants are typically found in the hidden layers. This choice will be explained in more detail later in section 2.2. Note that *ReLU* introduces non-linearity into the model, enabling neural networks to approximate complex functions.

Activation Function	**Formula**	**Best Used For**
Linear	$f(z) = z$	Regression (o.l.)
Sigmoid	$f(z) = \frac{1}{1+e^{-z}}$	Binary Classification (o.l.)
Softmax	$f(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}$	Multi-Class Classification (o.l.)
tanh	$f(z) = \frac{e^{2z}-1}{e^{2z}+1}$	Hidden Layers
ReLU	$f(z) = \max(0, z)$	Hidden Layers
Leaky ReLU	$f(z) = x \quad \text{for } z > 0$ $f(z) = \alpha z \quad \text{for } z \leq 0$	Hidden Layers

Table 1: *Popular activation functions and their uses. Note that o.l. stands for outer layer.*

As it was previously mentioned, a neural network can be seen as a directed weighted graph $G = (V, E)$, where V is the set of nodes (neurons) and E is a set of ordered pairs of vertices, representing the connections between neurons. Each node $v \in V$ has an associated activation function f. The output y_v of a node is given by:

$$y_v = f\left(\sum_{u \to v} w(u, v) \times y_u\right)$$

The sum runs over all nodes u that have an edge to node v, where:

- $u \to v$ denotes an edge from node u to node v.
- $w(u, v)$ is the weight of the edge from u to v.
- y_u is the output of the node u.

The graph representation provides a visual means to understand the architecture and flow of data in the network. The directionality of the edges indicates the direction in which data flows or is processed. Typically nodes are organized into layers, so:

$$V = V_{\text{input}} \cup V_{\text{hidden1}} \cup V_{\text{hidden2}} \cup \cdots \cup V_{\text{output}}$$

The input layer receives the data and forwards it to the subsequent layer. Typically, each neuron in this layer represents one feature or attribute that is being modeled. However, in the context of GANs (Generative Adversarial Networks), a noise vector of arbitrary length is commonly used instead. The hidden layers process the data from the input layer by extracting patterns and relevant features, and then pass this processed data to the next layers. When a model has multiple hidden layers, it is considered a deep learning model, enabling the capture of more intricate representations. Lastly, the output layer delivers the final results, with the number of neurons corresponding to the number of desired outputs.

2.2 Training Neural Networks.

When we say a neural network "learns", we mean it is adjusting these weights and biases so that its output becomes closer to the desired or expected results. This adjustment is done based on a set of provided data. As will be detailed in section 3.1, training GANs differs slightly from traditional supervised learning tasks, where you typically have labeled *datasets* that are split into training, validation, and test sets. The following section covers basic traditional learning.

In the first place, a cost function is defined to measure how well the output of the network, $y(x)$, is approximated for all the training inputs, x. Although the cost function, $C(\omega, b)$, can be defined in different ways depending on the usage of the ANN, the *mean square error* function (equation 2.3) will be the easiest to explain the training algorithm.

$$C(\omega, b) = \frac{1}{2n} \sum_x ||y(x) - a||^2 \tag{2.3}$$

Here ω denotes the collection of all the weights, b all the biases, n is the total number of training inputs and a is the vector of outputs from the network when x is input. Note that $C(\omega, b)$ is non-negative and that $C(\omega, b) \approx 0$ if the outputs match the training data. The goal of the training algorithm is to minimize the cost function. Central to this optimization are two intertwined concepts: backpropagation and gradient descent. Backpropagation determines the contribution of each parameter to the total error, and gradient descent outlines the method for updating these parameters.

2.2.1 Gradient descent.

Our objective is to determine the weights and biases that minimize the cost function. To achieve this, let's explore the mathematical intuition behind the formulas employed in gradient descent for parameter updates. Given a function f around a point x, the first-order Taylor series expansion is:

$$f(x + \Delta x) \approx f(x) + \Delta x \cdot \nabla f(x) \tag{2.4}$$

This tells us that the change in f when perturbed by a small amount Δx is approximately the dot product of the gradient and that perturbation. In this case, the cost function is $C : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$ that maps the parameters $\omega \in \mathbb{R}^n$ and $b \in \mathbb{R}^m$ to a real value. In order to find the minimum, $\Delta\omega$ and Δb have to be selected in a way that dot product $\Delta\omega \cdot \nabla C(\omega)$ and $\Delta b \cdot \nabla C(b)$ will be most negative, which is in the opposite direction to the gradient. Hence, the equations 2.5 and 2.6 are derived, where α is a factor called *learning rate* that controls how much to move in the descent direction.

$$\omega_k' = \omega_k + \Delta w = \omega_k - \alpha \frac{\partial C}{\partial \omega_k} \tag{2.5}$$

$$b_l' = b_l + \Delta b = b_l - \alpha \frac{\partial C}{\partial b_l} \tag{2.6}$$

Note that with the gradient descent, the global minimum is not always found due to the presence of local minima, saddle points, and flat regions in the loss surface. Starting from different initial points, the algorithm might converge to different local optima. Also, the learning rate can be a crucial factor in the convergence.

Notice that $C = \frac{1}{n}\sum_x C_x$ where $C_x = \frac{||y(x)-a||^2}{2}$ is an average over cost for individual training examples, assuming the cost function 2.3. Then, $\nabla C = \frac{1}{n}\sum_x \nabla C_x$. One of the main issues with backpropagation is that it is very slow. Therefore, the *stochastic gradient descent* is introduced to speed up learning. Instead of using individual training examples, mini-batches X_1, X_2, ..., X_m of sample size m are used.

$$\nabla C = \frac{1}{n}\sum_x \nabla C_x \approx \frac{1}{m}\sum_{j=1}^{m} \nabla C_{X_j} \tag{2.7}$$

Over multiple iterations, across different mini-batches, the noise tends to average out. For a given X_j, the expected value of the gradient is the true gradient, $E[\nabla C_{X_j}] = \nabla C$.

Given that the samples X_j are i.i.d., the expected value of the gradient estimate using the mini-batch is:

$$E[\nabla C'] = E\left[\frac{1}{m}\sum_{j=1}^{m} \nabla C_{X_j}\right] = \frac{1}{m}\sum_{j=1}^{m} E[\nabla C_{X_j}] = \frac{1}{m}\sum_{j=1}^{m} \nabla C = \nabla C \tag{2.8}$$

Now, let's consider the variance. If the gradients are noisy, they'll have a non-zero variance. Let's denote the variance of the gradient for a single data point as $\mathrm{Var}(\nabla C_{X_j}) = \sigma^2$. Given the properties of variance for i.i.d. variables:

$$\mathrm{Var}(\nabla C') = \mathrm{Var}\left(\frac{1}{m}\sum_{j=1}^{m} \nabla C_{X_j}\right) = \frac{1}{m^2}\sum_{j=1}^{m} \mathrm{Var}(\nabla C_{X_j}) = \frac{1}{m^2}\sum_{j=1}^{m} \sigma^2 = \frac{\sigma^2}{m} \tag{2.9}$$

The stochastic gradient equations for updating the parameters are:

$$\omega_k' = \omega_k + \Delta w = \omega_k - \frac{\alpha}{m}\sum_j \frac{\partial C_{X_j}}{\partial \omega_k} \tag{2.10}$$

$$b_l' = b_l + \Delta b = b_l - \frac{\alpha}{m}\sum_j \frac{\partial C_{X_j}}{\partial b_l} \tag{2.11}$$

2.2.2 Backpropagation.

Backpropagation [3] is an algorithm that lets us compute efficiently the partial derivatives with respect to the weights and biases, $\frac{\partial C_{X_j}}{\partial \omega_k}$ and $\frac{\partial C_{X_j}}{\partial b_l}$, present in equations 2.10 and 2.11. The way to proceed will be to compute the error δ_j^l of the neuron j in the l layer (defined in equation 2.12) and then relate those errors to the previously mentioned partial derivatives.

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} \text{ where } z_l^j = \sum_k \omega_{jk}^l a_k^{l-1} + b_j^l \text{ and } a^l = \sigma(z^l) \tag{2.12}$$

Note that ω_{jk}^l denotes the weight for the connection from the k-th neuron in the (l-1)-th layer to the j-th neuron in the l-th layer. The algorithm is based on equations 2.13, 2.14, 2.15 and 2.16. Notation-wise, the equations have been written in their vectorized form where $\odot$ denotes the Hadamard product.

An equation for the error δ^L of the output layer:

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \tag{2.13}$$

where δ^L is the error of the output layer, $\nabla_a C$ is the gradient of the cost function with respect to the activations and $\sigma'(z^L)$ is the derivative of the activation function with respect to the weighted inputs to the neurons.

An equation to relate the error δ^l of one layer ($l = 2,\ 3,\ ...,\ L$) to the error δ of the previous layer:

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \tag{2.14}$$

An equation for the rate of change of the cost with respect to any bias in the network:

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \tag{2.15}$$

An equation for the rate of change of the cost with respect to any weight in the network:

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l \tag{2.16}$$

Based on these equations, the backpropagation algorithm follows this steps:

a. **Input** x**:** Set the corresponding activation a^1 for the input layer.

b. **Feedforward:** For each layer $l = 2, 3, \ldots, L$, compute the weighted input

$$z^l = w^l a^{l-1} + b^l$$

and the activation

$$a^l = \sigma(z^l)$$

where w^l is the weight matrix for the l^{th} layer, b^l is the bias vector for the l^{th} layer, and σ is the activation function.

c. **Output error δ^L:** Compute the vector

$$\delta^L = \nabla_a C \odot \sigma'(z^L)$$

where $\nabla_a C$ is the gradient of the cost function with respect to the output activations and $\sigma'(z^L)$ represents the derivative of the activation function.

d. **Backpropagate the error:** For each $l = L-1, L-2, \dots, 2$, compute

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$$

This equation computes the error in layer l based on the error in layer $l+1$.

e. **Output:** The gradient of the cost function is given by

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$$

for the weights and

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j$$

for the biases.

2.2.3 Explanation of the usage of the activation functions.

At the end of section 2.1, different activation functions were introduced. It was also mentioned that some functions were often used in the output layer, whereas others worked better in the hidden layers. Now that the training algorithm is stated, the reason behind the various choices can be understood better.

Activation functions like the sigmoid squash their output into a small range between 0 and 1. For deep neural networks, using these functions can result in gradients that are too small for the network to learn effectively. The *vanishing gradient* problem is that σ' terms reduce the gradient and that slows down the learning. Therefore, the sigmoid function works better for binary classification, when a clearly differentiated output (0 or 1) is needed.

On the other hand, *ReLU* provides a solution to the slowing down problem, as it does not saturate. However, one of its main issues is that some neurons can sometimes output a constant zero value for all input examples. This can happen especially during the training phase, where, due to specific weight initialization or during the gradient descent process, certain neurons essentially "die" and remain inactive no matter the input. This means they do not make any modifications and do not contribute to the learning of the network. The activation function *LeakyReLU* attempts to solve this problem by introducing a small positive parameter α. This ensures that neurons remain slightly activated and, more importantly, maintain a non-zero gradient. As a result, neurons are less likely to die during training.

3 Generative Adversarial Networks (GANs).

Generative Adversarial Networks (GANs), introduced in the paper [4] "Generative Adversarial Nets," represent a class of generative models. At its core, a GAN consists of two intertwined deep neural networks. The first network, the Generator, aims to produce data, while the second, the Discriminator, evaluates it. This framework corresponds to a *minimax* two-player game, where a unique solution exists, with the generator G recovering the training data distribution and the discriminator D always equal to $\frac{1}{2}$. The great advancement of this architecture is that a generative model can be built just by passing random noise through a multilayer perceptron, and the discriminator is also a *MLP*. Therefore, both models can be trained using backpropagation algorithms, which have been proven very successful. There is no need for inference or Markov chains.

The generator and discriminator are defined as follows [5]. The Generator $G : \mathcal{Z} \to \mathcal{X}$ accepts noise data $z \in \mathcal{Z}$ (usually sampled from a Gaussian noise model) and produces samples $x \in \mathcal{X}$. $\mathcal{X}$ is the *data space* and $\mathcal{Z}$ is a *noise space*; $\mathcal{Z} = \mathcal{R}^{d_Z}$ where d_Z is a hyperparameter. On the other hand, the discriminator $D : \mathcal{X} \to [0, 1]$ accepts samples and predicts the probability that x came from the empirical data distribution rather than from the generative model.

3.1 Training Procedure of GANs: The Min-Max Game.

The adversarial process [4] is what gives GANs their name and unique characteristics. The objective function for GANs is given by:

$$\min_G \max_D V(D, G) = \mathbb{E}_{x\sim p_{\text{data}}(x)}[\log D(x)] + \mathbb{E}_{z\sim p_z(z)}[\log\bigl(1 - D(G(z))\bigr)] \tag{3.1}$$

where $\mathbb{E}$ denotes expectation, x are samples from the real data distribution p_{data}, z are samples from a noise space following the distribution p_z, $D(x)$ is the Discriminator's estimate of the probability that the real data instance x is genuine, and $G(z)$ is the data generated by the Generator.

To train the GAN the cost functions need to be defined, combined they represent the *minimax* game. The Discriminator's goal is to correctly classify real samples as real and generated samples as fake. Equation 3.2 is its cost function, which is minimized. The first term encourages the Discriminator to assign high probabilities to real samples, and the second term encourages it to assign low probabilities to generated samples.

$$J_D = -\mathbb{E}_{x\sim p_{\text{data}}(x)}[\log D(x)] - \mathbb{E}_{z\sim p_z(z)}[\log\bigl(1 - D(G(z))\bigr)] \tag{3.2}$$

On the other hand, the Generator's goal is to produce samples that deceive the Discriminator into thinking they are real. The generator aims to minimize the cost function 3.3, trying to make the Discriminator assign high probabilities to its generated samples.

$$J_G = \mathbb{E}_{z\sim p_z(z)}[\log\bigl(1 - D(G(z))\bigr)] \tag{3.3}$$

It is crucial to note that training alternates between two phases, the generator and discriminator cannot be trained simultaneously. In the first phase, the Generator is frozen, and the Discriminator is trained to distinguish between real and generated samples. Then, the Discriminator is frozen, and the Generator is updated to produce samples that deceive the Discriminator. This iterative process continues until a Nash equilibrium is reached. In the paper is proven that the optimum is reached when the probability distribution of the generator p_g converges to p_{data}.

3.2 Key architectures and variants.

DCGAN: Deep Convolutional GANs

One of the main advances in the evolution of GANs was introduced by Radford, Metz, and Chintala in 2015 [6] when they formulated a new architecture called *Deep Convolutional Generative Adversarial Networks* (DCGANs). Instead of fully connected (dense) layers, convolutional layers are used in both the generator and discriminator.

Convolutional Neural Networks (CNNs) are a category of deep learning models primarily designed to process data with a grid-like topology, such as an image, which can be viewed as a 2D grid of pixels. In a convolutional layer we define a filter, which is a smaller-sized matrix in terms of width and height, and contains a set of learnable parameters. These parameters are adjusted during the training process. The convolution operation, defined in equation 3.4, works as follows. At each position, the filter performs an element-wise multiplication with the portion of the input it is currently on, and the results are summed up. Once this operation is computed over the entire input, it results in the output of the layer called *feature map.*

$$(I * F)(x, y) = \sum_{i=1}^{w} \sum_{j=1}^{h} I(i, j) \cdot F(x - i + c, y - j + d) \tag{3.4}$$

Here w and h denote the width and height of the filter F, respectively. The constants c and d are defined as $c = \frac{w}{2}$ and $d = \frac{h}{2}$, ensuring the filter is centered at the position (x, y) during the convolution.

Often used after convolutional layers, pooling layers reduce the spatial dimensions of the feature maps, retaining the most important information. Also, dense layers can be used to produce the final output of the generator. The main advantage of DCGANs is that they can learn spatial patterns of bi-dimensional data structures, which signify the most common usage of these types of networks: image generation. In sections 4.2 and 4.3 DCGANs are tested, showing remarkable results.

Wasserstein GAN (WGAN)

Wasserstein GANs (WGANs) were introduced in a paper [7] titled ”Wasserstein GAN” by Martin Arjovsky, Soumith Chintala, and Léon Bottou. The authors proposed an alternative cost function modification to address some of the challenges faced by traditional GANs, such as mode collapse, training instability, and the vanishing gradient problem. The loss functions of both the critic (3.5) and generator (3.6) in WGANs are directly derived from the Wasserstein distance: $W(p_r, p_g) = \inf_{\gamma \in \Pi(p_r, p_g)} \mathbb{E}_{(x,y)\sim\gamma}[\|x - y\|]$, where γ is a joint distribution over data pairs and $\Pi(p_r, p_g)$ is the set of all joint distributions whose marginals are p_r and p_g.

$$L_{\text{critic}} = \mathbb{E}[D(G(z))] - \mathbb{E}[D(x)] \tag{3.5}$$

$$L_{\text{generator}} = -\mathbb{E}[D(G(z))] \tag{3.6}$$

One of the main differences is the shift from a probability-based discriminator to a score-based *critic*. While the discriminator in traditional GANs classifies samples as real or fake based on probabilities, the critic in WGANs evaluates the quality of samples using continuous scores. This is possible because the architecture of the critic does not end with a sigmoid activation function; therefore, the output is not constrained to [0, 1]. The critic's goal is to assign higher scores to real samples and lower scores to fake samples, providing a continuous score, rather than a binary classification.

Section 4.3 covers the implementation of WGANs to simulate a 2D Ising Model. However, before deploying this architecture, an improvement needs to be made. The main drawback of the initial Wasserstein loss is that it is not 1-Lipschitz continuous, which can cause vanishing or exploding gradients during training. For that reason, a *gradient penalty* is added to the critic's loss to enforce the 1-Lipschitz constraint. This penalty is computed by taking interpolated samples between real and generated samples, passing them through the critic, and then penalizing deviations of the gradient norm from 1. Specifically, the gradient penalty is defined as:

$$\mathcal{P} = \lambda \left(\left\| \nabla_{\hat{x}} D(\hat{x}) \right\|_2 - 1 \right)^2 \tag{3.7}$$

where $\hat{x}$ is the interpolated sample, ∇ denotes the gradient, and λ is a hyperparameter controlling the penalty's strength. By interpolating examples, we ensure that the critic behaves well not just where data exists but in the intermediate space between real and generated samples. Mathematically, $\hat{x} = \epsilon x_{real} + (1 - \epsilon) x_{generated}$. The total loss function of WGANs after implementing gradient penalty is defined as:

$$L_{\text{total}} = \mathbb{E}[D(G(z))] - \mathbb{E}[D(x)] + \lambda \left(\left\| \nabla_{\hat{x}} D(\hat{x}) \right\|_2 - 1 \right)^2 \tag{3.8}$$

where D is the critic, x the real examples, $G(z)$ the generated ones where z is the latent noise and λ is a hyperparameter that controls the strength of the gradient penalty.

4 Applications and Analysis.

This section covers the practical implementations of GANs across diverse domains. Initially, we explore the potential of GANs in generating DNA sequences, placing a keen emphasis on segmenting COVID-related sequences. We subsequently transition into the realm of image generation, highlighting the utilization of GANs in producing Gaussian images. Lastly, the section culminates with an examination of the 2D Ising Model simulated using Wasserstein GANs, presenting the methodology, challenges faced, and optimization strategies employed. Through each subsection, we explore insights into dataset construction, architectural decisions, detailed training procedures, and a comprehensive analysis of the resultant outputs.

4.1 GANs for DNA sequence generation: A focus on COVID sequence segmentation.

In the realm of mathematical biology, the precise generation and analysis of DNA sequences play a pivotal role in the profound understanding of genomic data. In this section, we delve into the application of GANs for generating DNA sequences, specifically focusing on the SARS-CoV-2 virus, commonly known as COVID. The primary objective is to segment the complete COVID sequence, which comprises 29,903 nucleotides, into smaller subsequences of 60 nucleotides each, and employ GANs to generate analogous sequence segments. This granular perspective may reveal localized patterns or variations that might be overlooked when considering the entire sequence.

4.1.1 Dataset construction and visualization.

The complete SARS-CoV-2 genome sequence was retrieved from the GenBank repository [8]. This sequence serves as the primary foundation for our dataset. For the purpose of granular analysis and subsequent employment of GANs, the entire sequence was segmented into smaller subsequences. Each subsequence contains precisely 60 nucleotides, resulting in an array of subsequences. The segmentation was performed by sliding a window of size 60 nucleotides across the genome sequence, moving one nucleotide at a time. This overlap ensures the capture of all possible 60-nucleotide sequences within the genome, providing a rich dataset for the GANs to train on. Some examples of the dataset are:

Sequence 1: `ATTAAAGGTTTATACCTTCCCAGGTAACAAACCAACCAACTTTCGATCTCTTGTAGATCT`

Sequence 2: `GTTCTCTAAACGAACTTTAAAATCTGTGTGGCTGTCACTCGGCTGCATGCTTAGTGCACT`

Sequence 3: `CACGCAGTATAATTAATAACTAATTACTGTCGTTGACAGGACACGAGTAACTCGTCTATC`

Each nucleotide in the sequence can be represented as a unique vector in a 4-dimensional space, corresponding to the four primary nucleotide bases: Adenine (A), Thymine (T), Cytosine (C), and Guanine (G). For instance, the nucleotide A might be represented as [1, 0, 0, 0], T as [0, 1, 0, 0], C as [0, 0, 1, 0], and G as [0, 0, 0, 1].

4.1.2 Generator and discriminator architecture.

As this is the first application that we are going to introduce, we will start considering the simplest case. The generator, as detailed in Table 2, is a one-dimensional feed-forward neural network comprising three dense (fully connected) layers interleaved with two *LeakyReLU* activation functions, which were covered in section 2.1. The input noise vector has a size of 101 and follows a normal distribution. This vector is first processed by 256 nodes, then expanded to 512 nodes, and finally compressed to a size of 240. This resultant vector represents a sequence of 60 nucleotides, with each nucleotide being represented by a vector of size 4.

Table 2: *Summary of the generator Model*

Layer (type)	Output Shape	Param #	Size
dense (Dense)	(None, 256)	25,856	25.1 KB
leaky_re_lu (LeakyReLU)	(None, 256)	0	0.00 Byte
dense_1 (Dense)	(None, 512)	131,584	512.6 KB
leaky_re_lu_1 (LeakyReLU)	(None, 512)	0	0.00 Byte
dense_2 (Dense)	(None, 240)	123,120	479.5 KB
Total params		280,560	1.07 MB

The discriminator's structure is presented in Table 3. Observe its pyramid-like configuration, with each successive layer reducing in size. Such an architectural choice is prevalent in discriminators, designed to iteratively refine and consolidate the information. Ultimately, the one-neuron output layer provides the likelihood that the input sequence is genuine, rather than generated.

Table 3: *Summary of the discriminator Model*

Layer (type)	Output Shape	Param #	Size
dense_3 (Dense)	(None, 512)	123,392	480.2 KB
leaky_re_lu_2 (LeakyReLU)	(None, 512)	0	0.00 Byte
dense_4 (Dense)	(None, 256)	131,328	511.8 KB
leaky_re_lu_3 (LeakyReLU)	(None, 256)	0	0.00 Byte
dense_5 (Dense)	(None, 1)	257	1.0 KB
Total params		254,977	993.00 KB

4.1.3 Training procedure.

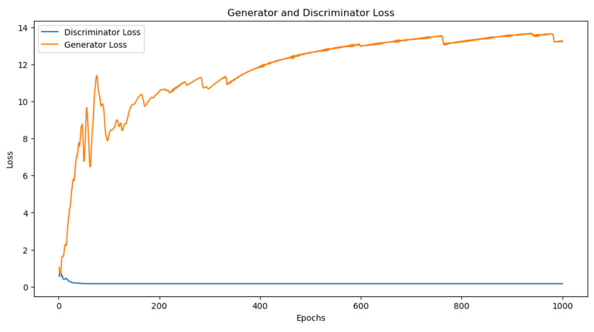
To begin, both the generator and discriminator are initialized. The optimization algorithm employed is *Adam*, which stands for "Adaptive Moment Estimation". This serves as an enhancement to the stochastic gradient descent. The *Keras* default learning rate for the Adam optimizer is set at 0.001, and this rate is applied to both neural networks. Based on our subsequent findings, we recommend setting the number of epochs between 5000 and 10000. Lastly, the binary cross-entropy is chosen as the loss function for both networks.

It is crucial to train separately the generator and discriminator to maintain the adversarial balanced. At the beginning of each epoch, the generator receives random noise sourced from a normal distribution. This noise is then transformed by the generator into synthetic DNA sequences. On the other hand, the discriminator samples genuine DNA sequences from the training dataset, which are subsequently merged with the synthetic sequences produced by the generator. The primary task of the discriminator is to discern between these real and fabricated sequences. To enhance the stability of the GAN's training, a technique called *label smoothing* is applied. Instead of tagging genuine sequences with a straightforward label of '1', a slightly diminished value, such as '0.9', is adopted. Once the data and its corresponding labels are established, the discriminator undergoes training on this composite batch.

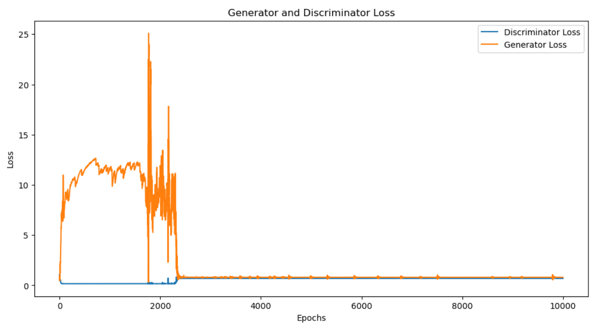
In the joint GAN training, new random input (noise) is created. To focus on improving the generator, the discriminator's training is temporarily turned off. This noise goes through the entire GAN, with the generator creating sequences that the discriminator then checks. The main goal is to make the generator produce sequences that the discriminator thinks are real. So, during training, we label these sequences as 'real' to challenge the generator to make better fake sequences. By doing this back-and-forth, the generator gets better at creating believable DNA sequences.

4.1.4 Results Analysis.

The training process yields the following results. Ideally, the generator and discriminator should have a similar loss to learn at the same pace. However, as we can see clearly in figure 3a, at the beginning the discriminator learns very quickly to distinguish the generated images. This is very common, as at first the generator outputs random noise and the discriminator has real DNA sequences to compare them with. If we extend the number of epochs, the generator receives information from the discriminator and, after many modifications, it learns how to trick it. Therefore, after more than 2000 epochs the loss converges. Although it might seem very small, usually for this cost function a loss between 0.5 and 2 is what is expected when the learning is stabilized.



(a) 1000 *epochs.*



(b) 10000 *epochs.*

Figure 3: *Evolution of generator and discriminator loss across epochs.*

Generated sequence: `TAATTTCCAAATATGGAAGGGTCCATTGTTTGGTTGGTTGAAAGCTAGAGAACATCTAGA`

Finally, once our GAN is trained, the generator can be loaded to generate DNA sequences. Note that, as is explained in the article, validation of GAN-generated data can be challenging. This is especially true in domains like genomics, where the implications of generated sequences might not be immediately clear. However, there are two facts that suggest that the GAN works well. First, the generator and discriminator losses stabilized. If the generator's loss continuously increases or the discriminator's loss drops too quickly to zero (like in figure 3a), it might indicate that the GAN is not learning effectively. Secondly, the generator produces sequences that *look* similar to the provided data.

4.2 Gaussian Image Generation using GANs.

GANs have shown substantial efficacy in diverse applications. While in the previous section we covered 1D representations in DNA sequences, now our focus shifts to images, which are inherently represented as 2D data structures. The transition to 2D data representation presents unique challenges, necessitating a shift in neural architecture.

This section introduces the usage of convolutional layers in the GAN framework. These layers, with their capacity to process spatial data, can handle the complexities of image generation and evaluation. We will delve into the creation of a Gaussian blob dataset, explore the architecture of the convolutional architectures of the generator and discriminator, and discuss the subtleties of the training procedure. A notable aspect of this training is the challenge of *mode collapse*. We address this pitfall, ensuring a comprehensive representation of the data distribution. Through the subsequent sections, we aim to provide a rigorous exploration of Gaussian image generation using GANs, detailing methodologies, challenges, and solutions.

4.2.1 Dataset construction and visualization.

The data will be provided with a dataset of images constructed using a method that superimposes multiple 2D Gaussian blobs onto a single 32 x 32 image matrix. A Gaussian blob is defined in equation 4.1, where A is the blob's amplitude, (x_0, y_0) denotes its center, and σ_x and σ_y are the standard deviations in the x and y directions, dictating its spread.

$$f(x,y) = A \cdot e^{-\left(\frac{(x-x_0)^2}{2\sigma_x^2} + \frac{(y-y_0)^2}{2\sigma_y^2}\right)} \tag{4.1}$$

For each image, a random number of Gaussian blobs, ranging between 3 to 9, is determined. The amplitude is set to 1, the centers are randomly selected within the range [0, 28) for both the x and y axes, and σ_x, σ_y are randomly chosen from the interval [2, 5). Then, a 2D grid of x and y coordinates is formed spanning the entire image, and the Gaussian function is applied to compute the intensity values for each grid point. After that, the intensity values of each Gaussian blob are added to the image matrix, resulting in the final image being a cumulative superposition of all Gaussian blobs. This process ensures each image in the dataset exhibits a unique combination of Gaussian structures, providing a varied set of synthetic Gaussian field images suiTable for GAN training. In figure 4 some examples of the dataset are displayed.

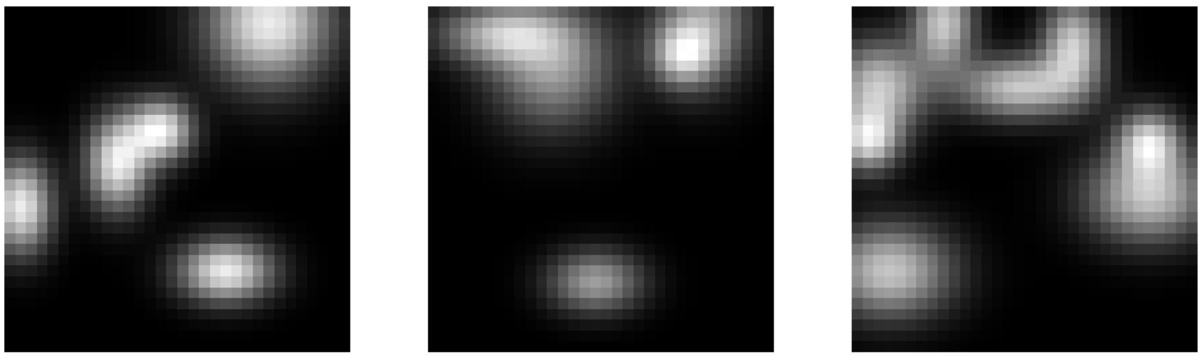


Figure 4: *Three example images from the dataset.*

4.2.2 Generator and discriminator architecture.

The shift from 1D to 2D data representations requires more advanced neural architecture [9]. In Table 4 the summary of the generator model is shown. The aim of the generator is to produce 32 x 32 images. It initiates with a dense layer which is responsible for learning a transformation from some low-dimensional input (a noise vector in this case) to a higher-dimensional representation, which will be reshaped in the subsequent layer to a 8 x 8 x 128 tensor. The transpose convolutional layers, *Conv2DTranspose*, in the architecture successively up-sample this tensor, first to 16x16 and then to 32x32, refining image details at each step. The LeakyReLU activation functions interspersed between these layers ensure non-linearity and maintain gradient flow during training.

Table 4: *Summary of the Generator Model*

Layer (type)	Output Shape	Param #	Size
dense_1 (Dense)	(None, 8192)	1,056,768	4.04 MB
reshape (Reshape)	(None, 8, 8, 128)	0	0.00 Byte
conv2d_transpose (Conv2DTranspose)	(None, 16, 16, 128)	262,272	1.00 MB
leaky_re_lu_3 (LeakyReLU)	(None, 16, 16, 128)	0	0.00 Byte
conv2d_transpose_1 (Conv2DTranspose)	(None, 32, 32, 256)	524,544	2.01 MB
leaky_re_lu_4 (LeakyReLU)	(None, 32, 32, 256)	0	0.00 Byte
conv2d_3 (Conv2D)	(None, 32, 32, 1)	6,401	24.7 KB
Total params		1,849,985	7.06 MB
Trainable params		1,849,985	7.06 MB
Non-trainable params		0	0.00 Byte

Examining the Table, one might wonder about the significance of the fourth element in the output shape of the convolutional layers. For instance, in the first layer, this value is 128 as indicated by the shape (None, 8, 8, 128). This value represents the number of *channels* at that layer. Each channel corresponds to a distinct feature map, which is the outcome of a specific convolutional filter applied to the input. These feature maps are matrices that emphasize or detect specific patterns or features in the data.

As for the discriminator (Table 5), its purpose is to classify 32 x 32 images as either real or generated. The idea behind the architecture is to reduce the spatial dimensions with a succession of convolutional layers, where initial layers capture basic patterns and deeper layers capture more complex features. Following these convolutional layers, the model flattens the tensor and introduces a dropout layer, which aids in preventing overfitting by randomly setting a fraction of inputs to zero during training. The final dense layer with a single node outputs the classification result, representing the likelihood of the input image being real.

Table 5: *Summary of the Discriminator Model*

Layer (type)	Output Shape	Param #	Size
conv2d (Conv2D)	(None, 16, 16, 32)	544	2.11 KB
leaky_re_lu (LeakyReLU)	(None, 16, 16, 32)	0	0.00 Byte
conv2d_1 (Conv2D)	(None, 8, 8, 64)	32,832	127.5 KB
leaky_re_lu_1 (LeakyReLU)	(None, 8, 8, 64)	0	0.00 Byte
conv2d_2 (Conv2D)	(None, 4, 4, 128)	131,200	509.8 KB
leaky_re_lu_2 (LeakyReLU)	(None, 4, 4, 128)	0	0.00 Byte
flatten (Flatten)	(None, 2048)	0	0.00 Byte
dropout (Dropout)	(None, 2048)	0	0.00 Byte
dense (Dense)	(None, 1)	2,049	7.95 KB
Total params		166,625	650.88 KB
Trainable params		166,625	650.88 KB
Non-trainable params		0	0.00 Byte

Note that while the discriminator uses convolutional layers, the generator employs transposed convolutional layers. The convolution operation in the discriminator aids in reducing the dimensions of a 32x32 image down to a single output, capturing the hierarchical features of the image. In contrast, the generator's task is to expand: it starts with a noise vector, which gets transformed into an initial spatial representation (like an 8x8 image), and then uses transposed convolutional layers to upscale this to a higher resolution, such as a 32x32 image. Finally, observe that the generator model is significantly heavier than the discriminator one (7.06 MB vs 650.88 KB). This is due to the fact that the generator requires more parameters to transform low-dimensional noise into detailed images. Conversely, the discriminator, focused on binary classification (real vs. fake), achieves its goal with a simpler structure and fewer parameters.

4.2.3 Training procedure.

The training of this GAN is analogous to the procedure described in section 4.1.3, we will only comment on the significant differences. The loss function used is the *binary cross-entropy* and the number of epochs is set to 100. Original images are labeled with 1's and generated images with 0's. In this case, a slightly more sophisticated *label smoothing* system is implemented, adding a small random noise to the original labels. Precisely, the noise is sampled from a uniform distribution and scaled by a factor of 0.05. Given that images are generated after each epoch, we incorporated a callback system to track the GAN's progression over time. Figure 5 illustrates the evolution of the generator's output. Initially, the generated structure doesn't capture the desired attributes, but within a few epochs, the generator begins to produce results closely resembling the dataset's samples.

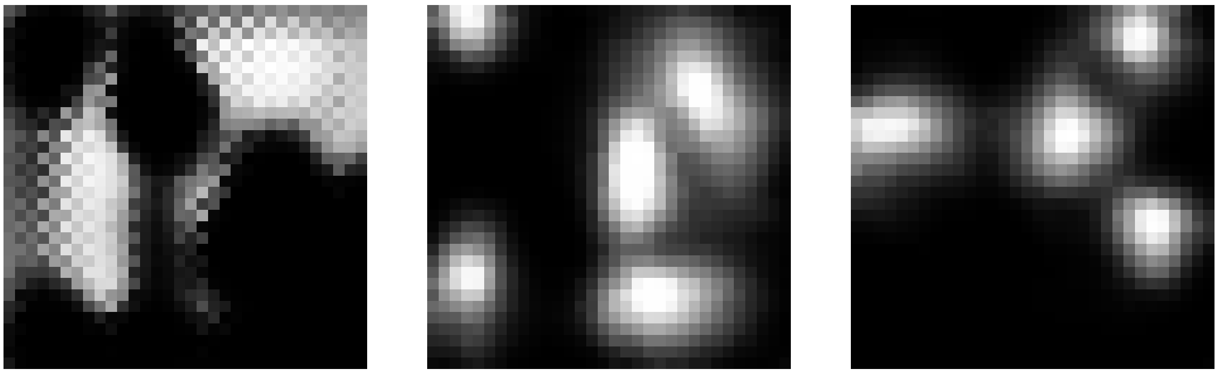

(a) 1st *epoch.* (b) 41st *epoch.* (c) 100th *epoch.*

Figure 5: *Evolution of image generation during the training process.*

After trial and error, we concluded that initializing the *Adam* algorithm with a learning rate of 0.0001 gave the best results. Initially, the approach was to penalize the discriminator with a lower learning rate because its task was less complex. However, that leads to *mode collapse.* As we previously emphasized, training GANs is notoriously difficult and one of the biggest challenges is to prevent this phenomenon. If, at any point during training, the generator finds a particular sample that the discriminator misclassifies as real with high probability, the generator might focus on producing variations of this sample to minimize its loss. It is called *collapse* because the generator produces a limited variety of samples, often focusing on a few modes (or clusters) of the data distribution, neglecting others. The primary consequence of mode collapse is that the GAN fails to represent the entire data distribution. Figure 6 shows the generator's output after 100 epochs.

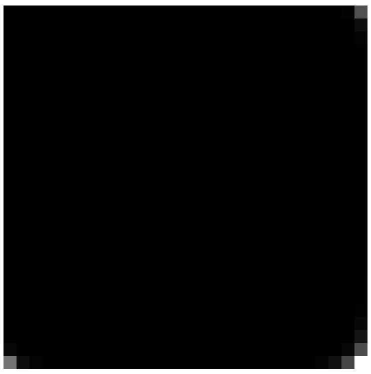


Figure 6: *Visualization of the mode collapse.*

4.2.4 Results analysis.

The evolution of the loss function of the generator and discriminator is shown in figure 7. Although it has been trained for 100 epochs, the results are very consistent. Both losses do not diverge and remain along the expected values during the whole training.

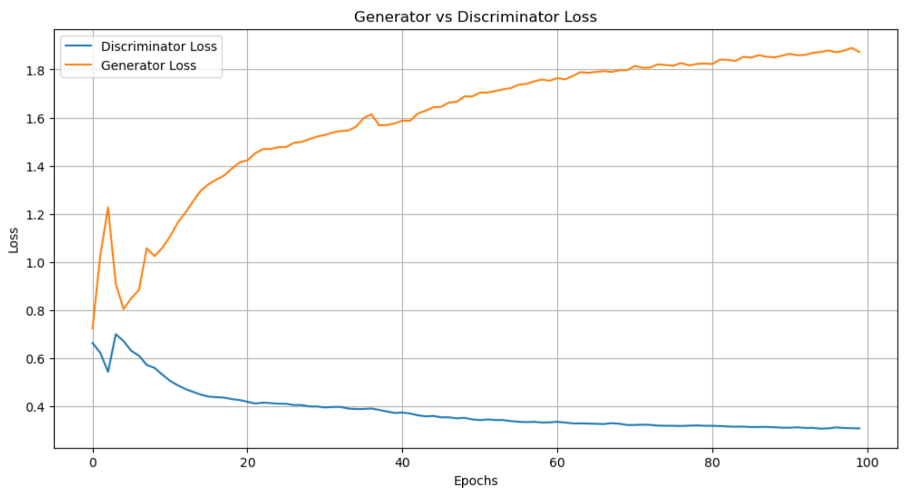


Figure 7: *Evolution of generator and discriminator loss across epochs.*

In our study, unlike the complexities encountered when determining the realism of DNA sequences, it is relatively straightforward to evaluate the authenticity of the generated images in this context. The process involves sampling a noise vector from a standard normal distribution. Once our generator has been adequately trained, it can transform this noise into images that closely resemble 2D Gaussian blobs. Figure 8 juxtaposes both the generated images and some original samples for a more direct comparison.

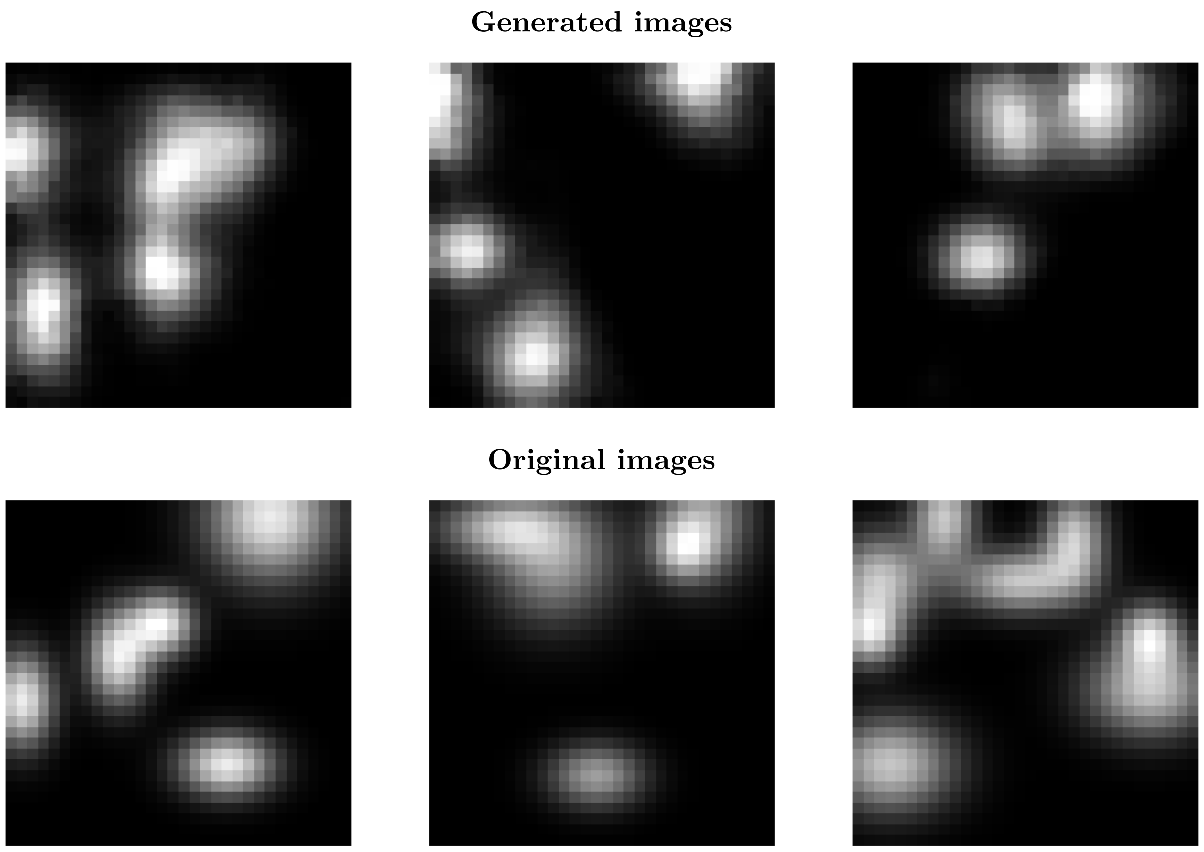


Figure 8: *Direct comparison. The top row contains generated images, while the bottom row contains original images.*

4.3 Simulating the 2D Ising Model with WGANs: Methodology, Challenges, and Optimization.

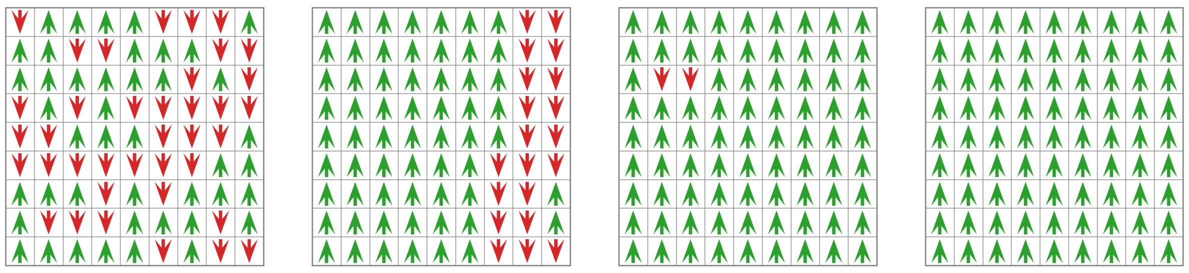
In this section, we simulate the 2D Ising Model using GANs, leveraging the Metropolis-Hastings algorithm [10] to generate a lattice dataset of spin states. Through the Monte Carlo method, we trace the system's progression over varied temperatures. The introduction of magnetization serves as a key performance metric. Emphasizing the role of the Wasserstein loss in training, we compare the GAN's outputs against traditional Monte Carlo simulations, especially in capturing phase transitions. We conclude with a focus on hyperparameter optimization via grid search to enhance the GAN's precision.

4.3.1 Dataset construction and visualization.

This research section aims to simulate a 2D Ising Model. The dataset will be constructed using the Metropolis-Hastings algorithm, which follows these steps. In the first place, the lattice needs to be initialized. Each site of the lattice i represents a spin σ_i, which can be in only one of these two states: up (represented as 1) or down (represented as -1). For this project, we have chosen to assign the values for each site of the lattice randomly (figure 9a); however, alternative initializations can be explored, such as initializing all spins in the up state or all in the down state. Next, we need to define the energy of the system, as it will determinate if each spin flips or not after each iteration. The energy of the system is determined by the interaction between neighboring spins, defined in equation 4.2. In what follows, we take J as the unit of energy, this is $J = 1$. If neighboring spins are aligned (i.e., both up or both down), the energy is lower (more negative), and if they are opposite, the energy is higher (less negative or positive).

$$E = -J \sum_{\langle i,j \rangle} \sigma_i \sigma_j \tag{4.2}$$

To simulate the temporal evolution of the Ising system, a Monte Carlo approach is employed. For each Monte Carlo step, a lattice site is chosen at random, and the energy change (ΔE) resulting from a potential spin flip at that site is computed. The Metropolis-Hastings algorithm dictates that the spin is flipped if ΔE is negative or with a probability determined by the Boltzmann factor, $e^{-\Delta E/T}$ where T is the temperature, if ΔE is positive.



(a) *Initial configuration.* (b) *1400 steps.* (c) *2000 steps.* (d) *3000 steps.*

Figure 9: *Evolution of a 9x9 lattice over a set of steps using the Monte Carlo algorithm at* $T = 1.5$.

Finally, for each temperature T defined within the range of 0 to 4, incremented by 0.1, the algorithm is executed. Note that the probability of transitioning from one state to another depends on the temperature T. Therefore, for each T we define a number of equilibration steps before collecting data. Additionally, we determine the total number of steps the algorithm will run for and the intervals at which lattice configurations will be saved. Consequently, for each T a series of configurations is preserved, capturing the system's progression. Also, for every temperature the average total energy and magnetization (defined in equation 4.3) are computed. In figure **??** we can see the progression of the lattice for a given temperature over a set of steps using the Monte Carlo algorithm. The selected temperature is $T = 1.5$, as we know that for these conditions the system is ferromagnetic and, therefore, all the spins tend to point up.

$$M = \frac{1}{N}\sum_i \sigma_i \tag{4.3}$$

where N is the total number of spins. In this example, $N = 9 \cdot 9 = 81$.

4.3.2 Generator and discriminator architecture.

Following the approach suggested in the article, we will introduce a reasonably simple generator model. In Table 6 the summary of the generator is shown. As in the case of Gaussian Image Generation, where 2D data structures were first introduced, here we will also find convolutional networks.

Table 6: *Summary of the generator Model*

Layer (type)	Output Shape	Param #	Size
noise (InputLayer)	(None, 128)	0	0.00 Byte
dense (Dense)	(None, 10368)	1,337,472	5.11 MB
reshape (Reshape)	(None, 9, 9, 128)	0	0.00 Byte
conv2d (Conv2D)	(None, 9, 9, 1)	1,153	
Total params		1,338,625	5.11 MB
Trainable params		1,338,625	5.11 MB
Non-trainable params		0	0.00 Byte

In the given generator model designed for simulating the 2D Ising system, the architecture commences with a noise input layer configured as a 128-dimensional vector, which acts like a seed for sample generation. Then a dense layer captures non-linearities and complexity, followed by a reshape operation that transforms the vector into a (9, 9, 128) tensor. This step is done to simulate the 9 by 9 grids used in the dataset. Note that as in section 4.2.2, 128 indicates the number of channels. Concluding the architecture there is a 2D convolutional layer, which diminishes the depth from 128 channels down to a singular one, yielding a 9x9 output.

The discriminator, also called *critic* when dealing with WGANs, is summarized in Table 7. Its role is to distinguish between real and generated 2D grids emanating from the Ising Model. As in previous models, dense, convolutional and LeakyReLu layers are used. The novelty of this architecture is the introduction of a *Global Max 2D Pooling layer*, which is commonly used in CNNs.

Given a 3D feature map F of size $H \times W \times C$, where H is the height, W is the width, and C is the number of channels, the operation of Global Max Pooling 2D is analogous to applying max pooling with a window of size $H \times W$. Therefore, for each channel c in the range $[1, C]$, the output is:

$$G_c = \max_{i=1}^{H} \max_{j=1}^{W} F_{i,j,c} \tag{4.4}$$

Here, G_c represents the output for channel c after applying Global Max Pooling 2D. The resulting output tensor after this operation is of size $1 \times 1 \times C$, where each value corresponds to the maximum value of the entire spatial dimension for its respective channel in the original feature map.

Table 7: *Summary of the Discriminator Model*

Layer (type)	Output Shape	Param #
images (InputLayer)	(None, 9, 9, 1)	0
conv2d_1 (Conv2D)	(None, 9, 9, 64)	640
leaky_re_lu (LeakyReLU)	(None, 9, 9, 64)	0
global_max_pooling2d (GlobalMaxPooling2D)	(None, 64)	0
dense_1 (Dense)	(None, 100)	6,500
leaky_re_lu_1 (LeakyReLU)	(None, 100)	0
dense_2 (Dense)	(None, 1)	101
Total params		7,241
Trainable params		7,241
Non-trainable params		0

4.3.3 Training procedure.

The main difference of this case of study is the introduction of the *Wasserstein loss.* As it was detailed in section 3.2, this special loss function used in WGANs is different from the standard cross-entropy (used in previous sections), providing more sTable training and helping mitigate mode collapse and vanishing gradients. Recall that a gradient penalty was introduced to ensure 1-Lipschitz continuity. During the critic's training, real data samples and fake samples (generated by the generator) are presented to the critic. The objective of the critic is to assign higher scores to real samples and lower scores to fake samples. Therefore, its job is to minimize equation 3.8, which had the form:

$$L_{\text{total}} = \mathbb{E}[D(G(z))] - \mathbb{E}[D(x)] + \lambda \left(\left\| \nabla_{\hat{x}} D(\hat{x}) \right\|_2 - 1 \right)^2$$

where D is the critic, x the real examples, $G(z)$ the generated ones where z is the latent noise and λ is a hyperparameter that controls the strength of the gradient penalty.

In the generator's training stage, the generator aims to produce samples that the critic cannot distinguish from real data samples. Thus, the generator is trained to maximize the critic's scores for its generated samples, effectively trying to decrease the Wasserstein distance between the real and generated data distributions. Training is conducted over a range of temperatures, starting from $T = 0$, and extending to $T = 4$ with increments of 0.1. Also, we divide the total dataset in batches of size 64. The term *iterations* refers to number of batches of data the algorithm has processed. In figure 10, the progression of the loss is illustrated for a $T = 0.4$ temperature, highlighting the three components from equation 3.8: the Wasserstein loss term, the gradient penalty contribution and the total loss. The GAN has been trained for 6 epochs and a learning rate of 10^{-6} for the discriminator and 10^{-4} for the generator, respectively.

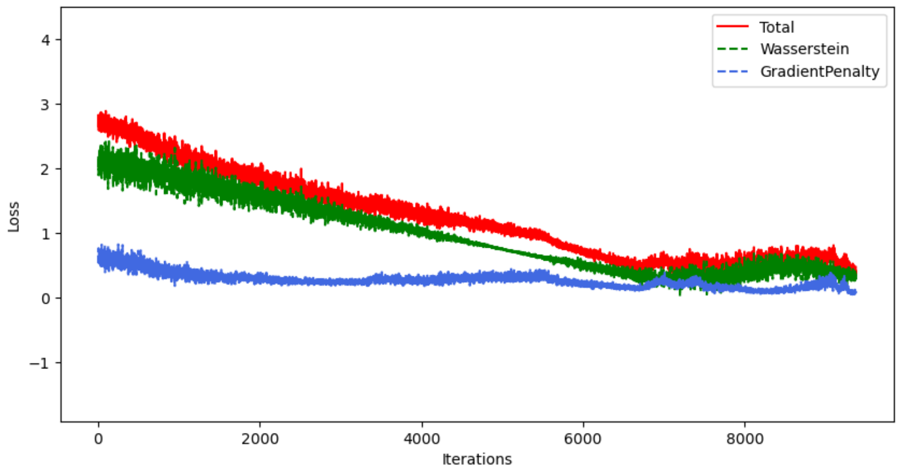
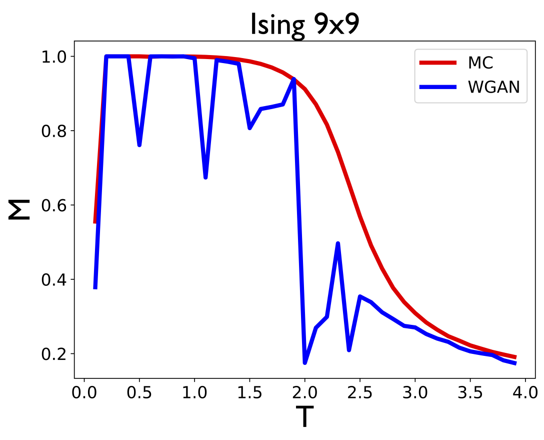


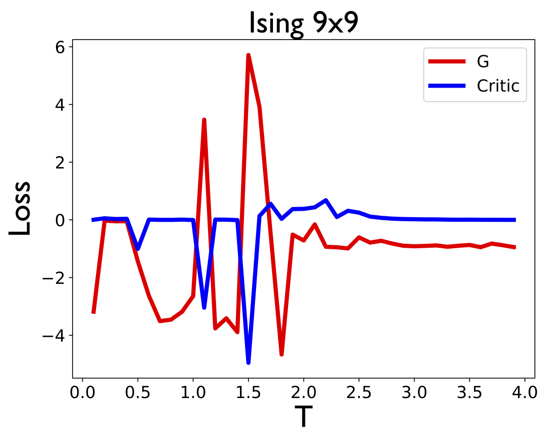
Figure 10: *Evolution of the loss (equation 3.8) for $T = 0.4$ and one epoch.*

4.3.4 Results analysis.

This experiment is notable because judging the GAN's performance by only observing the lattice configuration evolution is not straightforward. Instead of focusing on individual site states, we look at an overall measure: magnetization. In Figure 11, we compare the Ising model's progression using a Monte Carlo algorithm to the results from the GAN's generated lattices. The ANN produces outcomes that align with the system's trend, but there are some points we need to clarify.



(a) *Evolution of magnetization M as a function of temperature T.*



(b) *Evolution of the loss evolution of the generator and the critic.*

Figure 11: *Results after three epochs: Magnetization Evolution and Generator-Critic Losses*

Statistical physics [11] explains that at very low temperatures, spin systems characterized by the Ising model, tend to align in the same direction. The reason is the system's drive to minimize its Hamiltonian. This results in a magnetization of absolute value 1 (either fully "up" or fully "down"). This is correctly represented, as both lines have approximately the same tendency for $T < 0.2$.

It is interesting to note that at extremely low temperatures, not all the spins are aligned. That is due to the relaxation time, which can be especially prolonged at low temperatures due to the restrictive nature of the Metropolis-Hastings algorithm in this regime. To clarify terms, the time it takes for the system to reach its equilibrium state is known as the relaxation time. The GAN captures pretty well this behaviour.

The key result that statistical physics yields for this system is the critical temperature at which a second-order phase transition occurs. However, this critical temperature is defined in the thermodynamic limit. For finite systems, the transition is not sharp but rather smooth. The two-dimensional Ising model exhibits a phase transition at a critical temperature, approximately $T_c \approx 2.269$. Above this temperature, the system enters a disordered phase, characterized by a vanishing magnetization in the thermodynamic limit. As it is a smooth curve, the neural network cannot capture the higher order refinements and for temperatures between $T = 1.9$ and $T = 2.5$, the discrepancies between the GAN's outputs and the true Ising model behavior were markedly pronounced. It is worth noting that Figure 11b indicates that, even though the losses of the generator and discriminator remain balanced throughout the simulation, disparities tend to increase at higher temperatures.

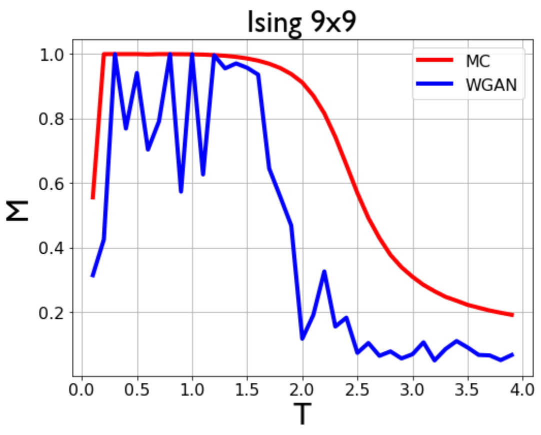
Finally, at high temperatures, the Metropolis-Hastings acceptance probability becomes more favorable. As a direct consequence, the system tends to oscillate around configurations with magnetizations closer to zero, signifying a more random orientation of spins. For most simulations, the GAN captures the behavior correctly, although the limited range between $T = 3.5$ and $T = 4$ where the curve starts to flatten can still cause some issues.

4.3.5 Improvements: Grid search.

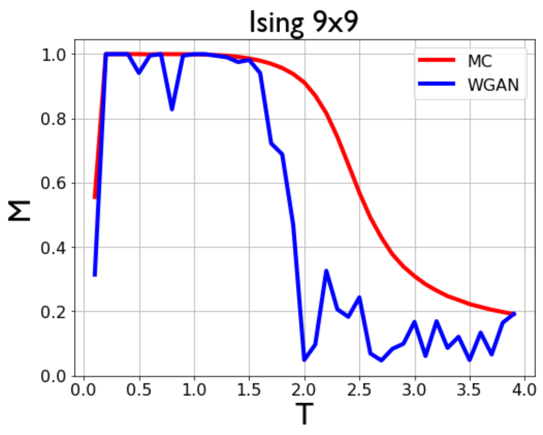
In the preceding sections, hyperparameters were selected manually, aiming for optimal results. Yet, in more sophisticated models, this process becomes challenging since a minor alteration in one parameter can influence the others. To address this complexity, we introduce a more refined technique: grid search.

Grid search is an optimization technique used to determine the best hyperparameters for a particular model. The method involves specifying potential values for each hyperparameter, and then evaluating every possible combination by training models on the data and assessing their performance. This systematic approach ensures a broad exploration of hyperparameter values, increasing the likelihood of optimal model performance. Nonetheless, the grid search approach can be computationally demanding, particularly when evaluating an extensive set of hyperparameters.

In figure 12 the two images side by side illustrate the impact of grid search. It is important to observe that the outcomes in figure 12a might appear suboptimal since we limited our consideration to just one epoch to minimize computational expense. In this case, our focus was on the variation of the learning rate of the critic and the number of epochs. Specifically, the values of the learning rate were: 10^x, where $x \in \{-6, -5.75, -5.5, \dots, -4.25\}$. In the appendix, the advanced code provides the foundation for handling grid search with the specified hyperparameters: number of epochs and learning rate of both the critic and the generator. Analyzing figure 12b, reveals that for just one epoch the results for low temperatures closely resemble the Monte Carlo simulation. In this case, we performed a search over epochs and learning rates. In Figure 12b, the results display the selection for each temperature based on the minimum of the sum of the losses of the critic and the generator.



(a) *Fixed critic learning rate and three epochs.*



(b) *Grid search: epochs and critic learning rate.*

Figure 12: *M as a function of T. A comparison highlighting the impact of employing grid search.*

5 Conclusions.

In this project, we have successfully met our objective of delving deep into the complexities of Generative Adversarial Networks, elucidating both the fundamental concepts and the advancements that have been introduced to improve their efficiency and applicability. Through the structured sections, we initially introduced the foundational elements of neural networks, laying a solid groundwork for understanding the sophisticated developments in the field of GANs. Notably, we have incorporated substantial advancements in the GAN domain, introducing improvements such as Deep Convolutional GANs (DCGANs) and WGANs.

Each subsection of the application segment not only explores the implemented methodologies and the associated challenges but also offers a thorough evaluation of the attained results, thereby meeting our objective to meld theoretical exposition with practical insights into the applications of these networks in real-world contexts. The results garnered in this project are highly promising, with section 4.2 successfully generating detailed images from the dataset and illustrating the profound capabilities of GANs in replicating the macroscopic behaviors of systems, as delineated in section 4.3. A significant challenge encountered in all three case studies was identifying the optimal combination of hyperparameters. Moving forward, we recommend the application of more sophisticated techniques such as grid search, or expanding the range of values considered for each hyperparameter to enhance the performance further. Although constrained by the substantial computational costs associated with these techniques in our current endeavor, this opens a clear path for future improvements.

References

[1] Martin Erdmann et al. “Generating and Refining Particle Detector Simulations Using the Wasserstein Distance in Adversarial Networks”. In: *Computing and Software for Big Science* 2 (2018). Accessed: [Insert Date], p. 4. DOI: 10.1007/s41781-018-0008-x. URL: https://link.springer.com/article/10.1007/s41781-018-0008-x.

[2] Michael A. Nielsen. *Neural Networks and Deep Learning*. Chapter 1. San Francisco, CA: Determination Press, 2015. URL: http://neuralnetworksanddeeplearning.com/.

[3] Michael A. Nielsen. *Neural Networks and Deep Learning*. Chapter 2. San Francisco, CA: Determination Press, 2015. URL: http://neuralnetworksanddeeplearning.com/.

[4] Ian J. Goodfellow et al. “Generative Adversarial Nets”. In: *Advances in Neural Information Processing Systems*. Vol. 27. 2014. DOI: 10.5555/2969033.2969125. URL: https://papers.nips.cc/paper/2014/hash/5ca3e9b122f61f8f06494c97b1afccf3-Abstract.html.

[5] Mehdi Mirza and Simon Osindero. “Conditional Generative Adversarial Nets”. In: *arXiv preprint arXiv:1411.1784* (2014). DOI: 10.13140/2.1.1347.4240. URL: https://arxiv.org/abs/1411.1784.

[6] Alec Radford, Luke Metz, and Soumith Chintala. “Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks”. In: *International Conference on Learning Representations (ICLR)*. 2015. DOI: 10.13140/RG.2.1.1829.1369. URL: https://arxiv.org/abs/1511.06434.

[7] Martin Arjovsky, Soumith Chintala, and Léon Bottou. “Wasserstein Generative Adversarial Networks”. In: *Proceedings of the 34th International Conference on Machine Learning*. Vol. 70. 2017, pp. 214–223. DOI: 10.5555/3305381.3305400. URL: http://proceedings.mlr.press/v70/arjovsky17a.html.

[8] GenBank. *Genome Wars Cover 2 Complete Sequence*. GenBank: [Accession number]. 2023. DOI: 10.1234/genbank.gwcover2. URL: https://www.ncbi.nlm.nih.gov/genbank/.

[9] Google. *Machine Learning Course: Training Generative Adversarial Networks (GANs)*. Online Course. 2023. URL: https://developers.google.com/machine-learning/gan/training?hl=es-419.

[10] W.K. Hastings. “Monte Carlo sampling methods using Markov chains and their applications”. In: *Biometrika* 57.1 (1970), pp. 97–109. DOI: 10.1093/biomet/57.1.97.

[11] R.K. Pathria and Paul D. Beale. *Statistical Mechanics*. 3rd. See discussion on the critical temperature of a 2D Ising model in relevant sections. Elsevier, 2011. ISBN: 978-0-12-382188-1. DOI: 10.5555/1984092. URL: https://www.elsevier.com/books/statistical-mechanics/pathria/978-0-12-382188-1.