cDCGAN for Ising Lattice Generation

Shokry Ahmedeo

ID number: 1750037

E-mail: shokry.1750037@studenti.uniroma1.it

Abstract

GANs or Generative Adversarial Networks, are generative models capable of generating new random plausible examples once learned from a training set the probability distribution of that kind of data. In general a simple GAN can't generate specific output, it generates data by randomly sampling from a latent space. Ideally it could be possible generate certain sample by knowing the complex mapping that exist between the latent space and the output space. In practice it is almost impossible to do, that's why the idea of a conditional generation. We can impose constrains during the training phase obtaining a specific desired output. This kind of architecture are called cGAN (conditional GAN). In this work it will be presented a cGAN architecture for the generation of Ising square lattice using as a condition the temperature at which the lattice must be generated. I refer to cDCGAN instead of just cGAN just because the specific architecture I used is a deep convolutional one (conditional Deep Convolutional Generative Adversarial Network).

Contents

1 From GAN to cGAN

1.1 GAN

GAN architectures are adversarial networks whose objective is to estimate generative models.
The main idea is to simultaneously train two networks: a generative one G that learn from the train set the distribution of the data, and a discriminative one D that should estimate if a given sample came from the training set or is a generated one. This framework is a particular minmax two-player game, infact during the training phase we want that G maximize the probability of D making a mistake, but simultaneously we want also that D minimize it. Here the solution we want to achive at convergence is that G effectively recover the data distribution and that D is equal to $1/2$ for every sample.
In vanilla GAN architecture we have seen that the generator learns the distribution p_g over data $\mathbf{x}$, to do so we sample noise variables from a prior distribution in the latent space $p_{\mathbf{z}}(\mathbf{z})$, then we define a representation that map those variables to data space as $G(\mathbf{z};\theta_g)$, where G is a differentiable function represented by the network with parameters θ_g. The second function $D(\mathbf{x};\theta_d)$ is also represented by a neural network but it outputs a single scalar. As said before it represent the probability that $\mathbf{x}$ came from the data rather than p_g.
So during the training we have D that tries to maximize the probability to correctly label both the training samples and the generated one, simultaneously G tries to minimize $\log(1 - D(G(\mathbf{z})))$.
We have the following min-max problem [1]:

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

In the related paper are mentioned some of the advantages and disadvantages of this architecture, I'd like to point out at least one of both. One of the main disadvantages is the extreme importance to carefully synchronize D with G during the training in order to avoid the "Helvetica scenario", in which too many values of the latent point $\mathbf{z}$ are mapped into the same sample $\mathbf{x}$ by the generator G, thus inducing a bad generative capacity.
instead one of the main advantages is that adversarial network are capable of generating very sharp distribution (images in our case), instead of some other architecture such as AE that generally give blurry results.

1.2 cGAN

One of the features of the vanilla GAN is the sampling procedure of the generator network. In fact the generated data $\mathbf{x}$ are given by randomly sampling latent points $\mathbf{z}$ and then mapping them through G. In this scenario we don't have any control on the generated data, especially when we are dealing with data that belongs to a particular class.
The main idea of cGAN (conditional GAN), is to conditionate the model by adding new information to direct the data generation progress. The conditioning can be made both in the generator and the discriminator, in mathematical terms we can just add the label $\mathbf{y}$ in the definition of G and D. $\mathbf{y}$ can be any kind of information such as class label or data.
The new min-max problem can be formulated as follow:

$$\min_G \max_D V(G,D) = \mathbb{E}_{\mathbf{x}\sim p_{\text{data}}(\mathbf{x})}[\log D(\mathbf{x}|\mathbf{y})] + \mathbb{E}_{\mathbf{z}\sim p_{\mathbf{z}}(\mathbf{z})}[\log(1 - D(G(\mathbf{z}|\mathbf{y})))] \tag{2}$$

We can think to conditionate the generator by combining the $\mathbf{y}$ with the input noise $p_{\mathbf{z}}(\mathbf{z})$ into a hidden representation. instead for the discriminator both $\mathbf{x}$ and $\mathbf{y}$ are given as inputs [2]. In the following we have a simple representation of the conditioning method:

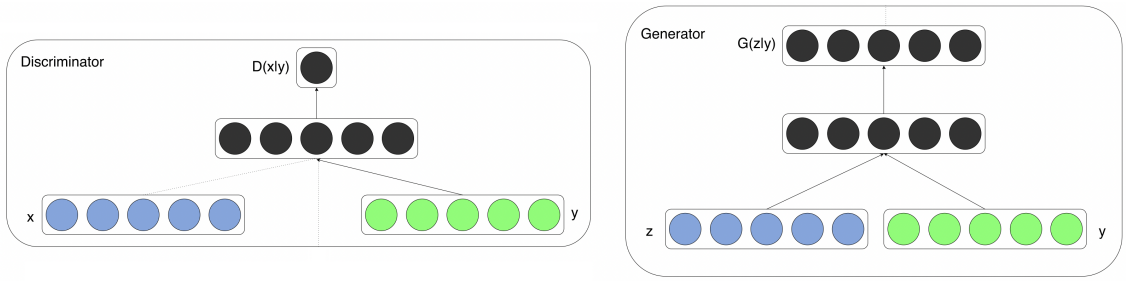

Figure 1: Generator and discriminator of a conditional generative adversarial network.

2 Ising Model

The Ising model is an important statistical model which describes the magnetism in matter, in particular the evolution of spin orientation given a specific temperature. The model is composed by discrete variables (-1,1) that are the spin orientation inside the lattice; the particular orientation of them determines the macroscopic behaviour of matter. In particular we can distinguish three kind of phase, the ordered, the critical and the disordered. In the simplest model, the spins in the lattice interact with their first neighbors in order to obtain the minimum energy configuration, in contrast there is the termal disturbance that hampers alignment. In particular, this disturbance gives the possibility of the realization of the three phase described before.
The general hamiltonian of the system is the following:

$$H(\sigma) = -\sum_{\langle i,j \rangle} J_{ij}\sigma_i\sigma_j - \mu\sum_j h_j\sigma_j$$

Where the σ_is are the spin discrete variable of the lattice, and if we imagine a 2D lattice NxN, the number of spin variable will be N^2, J_{ij} is the coupling constant between two neighbors spin and h_i is a generic external magnetic field. In the simplest case (the one I used) the couplings are constant and equal to 1, also the external magnetic field is zero. The equation simplifies like this:

$$H(\sigma) = -J\sum_{\langle i,j \rangle} \sigma_i\sigma_j$$

So the energy is given just by the sum over i and j of the product of two neighbors spin. It is trivial to states that the configuration with minimun energy is the one with all the spin aligned, and this is what we would obtain if there is no heat disturbance of the environment. In particular it can be demonstrate that if the temperature T is higher than a critical temperature $T_c = \frac{2J}{k\ln(1+\sqrt{(2)})} \approx 2.26$, then the heat disturbance dominates the spin alignment, on the contrary if $T < T_c$ the spin behaviour will be dominated by the interaction term. The third region is the critical one and it is verified when $T \approx T_c$. In the following we have the three region with the corresponding temperatures.

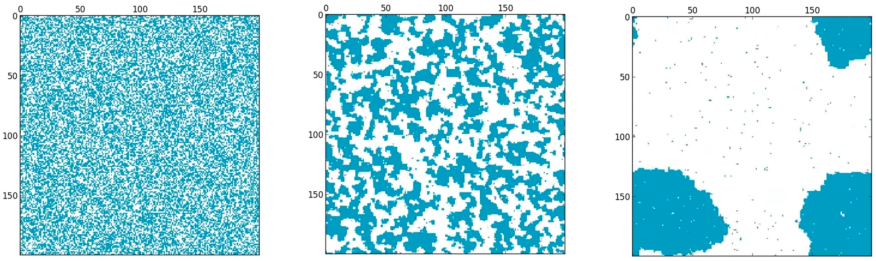

Figure 2: On the left an Ising lattice with temperature $T < T_c$, on the center an Ising lattice with temperature $T \approx T_c$ and on the right an Ising lattice with temperature $T > T_c$

Lars Onsager gave the 2D Ising model analytical solution in 1944, this is import since it help us to verify the correct behaviour given by the generated lattice with respect not only the simulated lattice (via Montecarlo) but also with respect its mathematical solution. In this project I'm going to study in particular four important measures, the magnetization, the magnetic susceptibility, the energy and the heat capacity. Those four quantity in general will be dependent on the lattice size and the coupling costant J. The general analytical solution is of course valid for the thermodynamic limit, so for $N \to \infty$ and $V \to \infty$, where N is the number of atoms and V is the volume of the lattice. Since we cannot have an infinite lattice, my solution will be an approximation of the analytical one.

2.1 Magnetization

Briefly, the magnetization is defined like this:

$$\langle M \rangle = \frac{1}{N}\sum_i^N \sigma_i \tag{3}$$

In practice is the sum over all the spin values of the atoms in the lattice. We can expect 0 magnetization when $T > T_c$, approaching 1/-1 when $T < T_c$. If we take the absolute value we will obtain a curve that goes from 1 to 0 for increasing temperature.

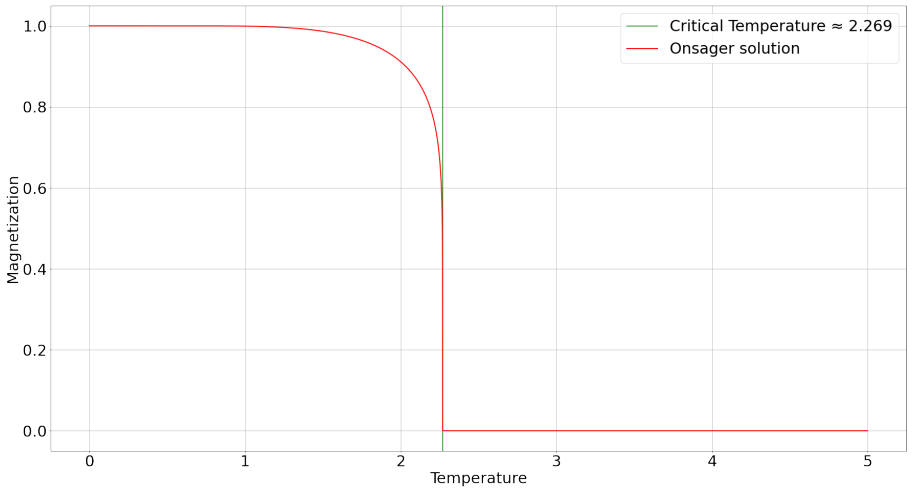

Figure 3: The Onsager solution in the thermodynamic limit.

One particular fact is the presence of a discontinuity point for $T = T_c$.
In the following, the analytical solution for the magnetization:

$$M = \left[1 - \frac{1}{\sinh(2\beta J)^4}\right]^{1/8} \tag{4}$$

Where β is $1/T$

2.2 Magnetic susceptibility

Also here briefly, we can define the magnetic susceptibility as a measure of how much a material will become magnetized, in this case it is clear that it is related with the magnetization value. Operationally, we can define it as the *variance* of M in this way:

$$\chi = \frac{\partial \langle M \rangle}{\partial H} = \frac{1}{k_B T}\left(\langle M^2 \rangle - \langle M \rangle^2\right) \tag{5}$$

Also here we expect a critical behaviour when $T \approx T_c$, in particular a divergent one. Intuitively we can think of it as how much the single atom is "sensible" (and in some sense correlated) to the other atoms presents in the whole lattice. In fact when the temperature is near the critical point we observe a phase transition in which all the atom spins are correlated and a change in one of them affect drastically the behaviour of the others.

2.3 Energy

Also for the energy we can have a value of it just by applying the definition:

$$E = \langle H \rangle = \frac{1}{N} \sum_{\langle i,j \rangle} \sigma_i \sigma_j \tag{6}$$

Where J is direcly set to 1 since here we are in this situation.
Here the expected value of the energy would be $-2J$ when all the spins are aligned ($T \sim 0$). We expect a continuous function of the temperature and when $T \approx T_c$ we expect a point of inflection. For $T > T_c$ we also expect $E \sim 0$. So the energy tells us that a phase change has ocurred and the material has become disordered and paramagnetic when we pass from $T < T_c$ to $T > T_c$.

2.4 Heat capacity

The last quantity I'm going to study is the heat capacity of the lattice defined as follow:

$$C = \frac{\partial \langle H \rangle}{\partial T} = \frac{1}{K_B T^2}\left(\langle H^2 \rangle - \langle H \rangle^2\right)$$

Also this quantity in the thermodynamic limit has a divergent behavior for $T \approx T_c$, but since I'm in the case of finite size 2D lattice, I'm expecting instead a peak maximum.

3 The Dataset

For the training of the network I used a dataset composed by 10000 squared Ising lattice of dimension 96x96 with its specific temperature label. Metropolis algorithm was used to generate the spin square lattices, in particular it uniformly sample a temperature value between 0 and 5 and let evolve a square lattice with random oriented spins. Once arrived at convergece the program print on a file the resulting square lattice and on another file the specific temperature. This loop continues until the generation of 10000 lattice. The Ising lattices as well as the temperature labels can be downloaded here. We can have a look at the generated lattices; here we have the three region of interest:

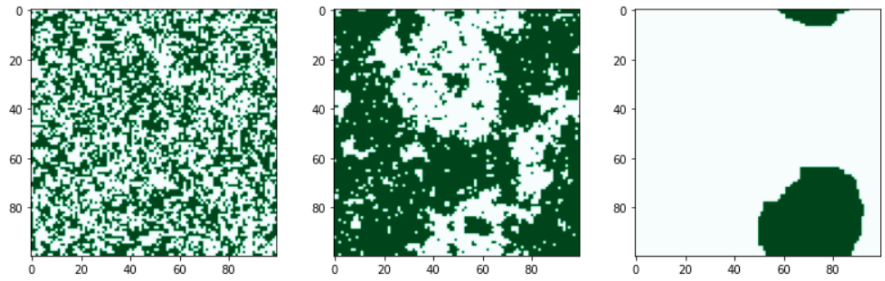

Figure 4: The three regions generated by the Metropolis algorithm.

Since the cGAN algorithm need discrete class label, it is important to discretize the temperature label. In order to obtain good results I decided to discretize it in 100 classes, here we can have a look at the non discretized temperature distribution and the discretized one:

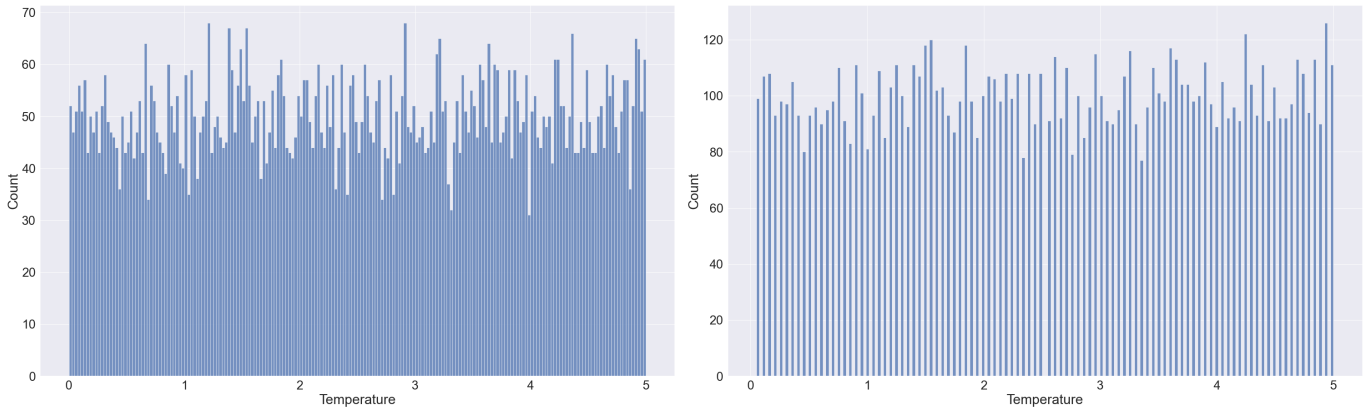

Figure 5: On the left the histogram of the continuous temperature, on the right the discretized one. The two distribution are exactly the same but the discretized version is much more sparse.

4 Algorithm

As we have seen in the section "From GAN to cGAN", the general GAN architecture is composed essentially by two deep neural network: the generator and the discriminator. We have seen also that the vanilla GAN has no control in the type of images that generates. Once trained, the only way to do it is to try to understand the complex map G that associate a specific value of $\mathbf{z}$ to a specific image $\mathbf{x}$. The simple solution to solve this problem is to conditionate the generator and the discriminator, and one way to do it is to embed the label $\mathbf{y}$ in a higher representation and concatenate it with the specific sample $\mathbf{x}$ during the training of both the generator and the discriminator.

The following code is written using the TensorFlow library using the functional API in Keras. The choice of using the functional API is simply because it allows to create complex network in which multiple input or output can be used also in parallel in a way such that the merging of parallel branch can be done. Essentially this flexibility is necessary in particular when we want to conditionate the generator and the discriminator, impossible when the sequential API is used.

4.1 Discriminator: D

In the following I present the particular approach I used in order to conditionate and train the discriminator.

```
# Define the discriminator model
# The model take 96x96x1 images and the number of classes as input
def define_discriminator(in_shape=(96,96,1), n_classes=100):

    # Define the input layer
    in_label = Input(shape=(1,))
    # Embedding for categorical input (dimension = 500)
    embed = Embedding(n_classes, 500)(in_label)
    # Scale up to image dimensions with linear activation
    n_nodes = in_shape[0] * in_shape[1]
    embed = Dense(n_nodes)(embed)
    # Reshape to additional channel
    embed = Reshape((in_shape[0], in_shape[1], 1))(embed)
    # Image input
    in_image = Input(shape=in_shape)
    # Concatenate label as a channel
    merge = Concatenate()([in_image, embed])
    .
    .
    .
```

As we can see there is an embedding layer in which a positive integer (n_classes) is mapped into a dense vector of fixed size (500 in this case). This vector is then fed in a dense layer with 96*96 nodes and then reshaped into a matrix 96x96. The resulting matrix is then merged with the actual image and fed into the ConvNet.

```
    .
    .
    .
    # Downsample
    conv = Conv2D(64, (3,3), strides = (2,2), padding='same')(merge)
    conv = ELU(alpha=1.0)(conv)
    conv = Conv2D(64, (3,3), strides = (2,2), padding='same')(conv)
    conv = ELU(alpha=1.0)(conv)
    conv = Conv2D(128, (3,3), strides = (2,2), padding='same')(conv)
    conv = ELU(alpha=1.0)(conv)
    # Flatten feature maps
    flat = Flatten()(conv)
    # Dropout
    flat = Dropout(0.45)(flat)
    # Output
    out_layer = Dense(1, activation='sigmoid')(flat)
    # Define model
    model = Model([in_image, in_label], out_layer)
    # Compile model
    optim = Adam(learning_rate = 0.0002, beta_1=0.9, beta_2=0.999)
    model.compile(loss='binary_crossentropy', optimizer=optim, metrics=['accuracy'])

    return model
```

The next part is a simple ConvNet, all the three layers are performed using a kernel map (3x3), stride (2,2) and padding = "same"[1]. Between each layer non-linear activation function is used, in particular here I used ELU function with α param set equal to 1.0 [2]. The main reason to use a ELU instead of a regular ReLU lies in the fact that the latter suffer from the problem called "dying ReLU" in which the neurons with negative value are set to 0 once the activation is applied. Generally speaking this is not a big problem since it promotes sparsity of the network which is a good feature in terms of generalization, however this is not the case since the architecture I'm using (GAN) as said before is very unstable, and since the ReLU activation tends to set to zero a good portion of the network, in the end I would obtain a very unstable model.

[1] As written in the Tensorflow documentation: *"same" results in padding with zeros evenly to the left/right or up/down of the input.* Of course the padding size is given by the stride value.

[2] During the construction of the algorithm also the LeakyReLU function were used, the ELU activation was the best in terms of performance.

Once the input image is shriked by a factor of 8 (from 96x96 to 12x12), the resulting image is flattened and passed through a dense net with just one output value. Dropout were used in order to have better generalization power forcing weight sharing. The discriminator is trained with Adam optimizer with learning rate = 0.0002, using as a loss the binary crossentropy and as metrics the accuracy.
In the following ReLU, Leaky ReLU and ELU functions are shown:

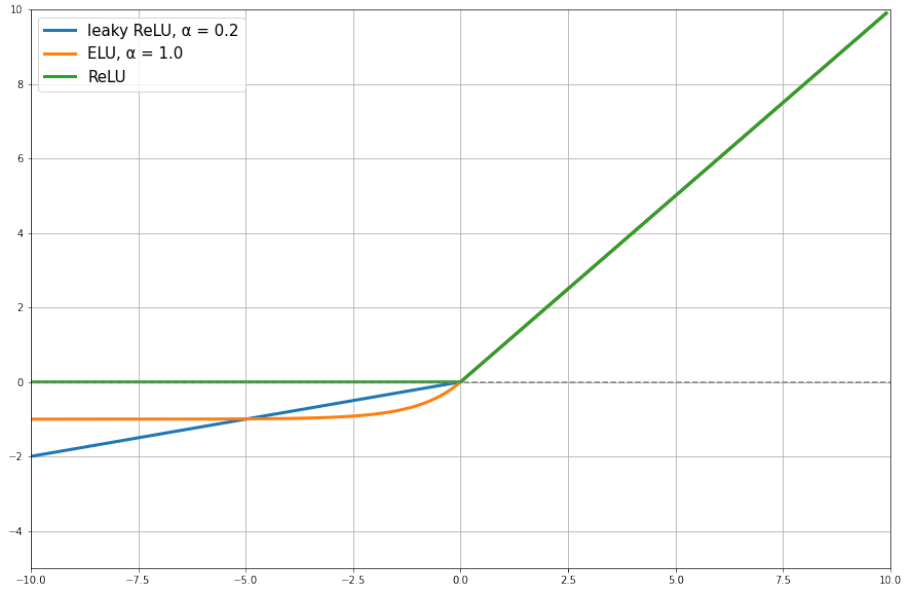

Figure 6: The LeakyReLU activation function.

4.2 Generator: G

Here is the code for the generator.

```
# Define the generator model
def define_generator(latent_dim, n_classes=100):

    # Define the input layer
    in_label = Input(shape=(1,))
    # Embedding for categorical input (dimension = 500)
    embed = Embedding(n_classes, 500)(in_label)
    # Linear dense net
    n_nodes = 12 * 12
    embed = Dense(n_nodes)(embed)
    # Reshape to additional channel
    embed = Reshape((12, 12, 1))(embed)
    # Image generator input
    in_lat = Input(shape=(latent_dim,))
    # Create foundation image 12*12
    n_nodes = 128 * 12 * 12
    gen = Dense(n_nodes)(in_lat)
    gen = ELU(alpha=1.0)(gen)
    gen = Reshape((12, 12, 128))(gen)
    # Merge the generated image and label input
    merge = Concatenate()([gen, embed])
    .
    .
    .
```

Here the procedure is a little bit different but the idea is the same. There is always an embedding layer for the label representation that is then fed into a dense net and reshaped using the dimension of the foundation image. In parallel a second branch is created where the foundation of the image is created (12*12 vector), this is then fed into a dense layer with 128*12*12 nodes , the resulting vector is then passed through the activation function (ELU) and then reshaped in a 12x12x128 tensor in order to merge it with the label representation created before.

```
    .
    .
    .
    # Upsample
    gen = Conv2DTranspose(128, (3,3), strides=(2,2), padding='same')(merge)
    gen = ELU(alpha=1.0)(gen)

    gen = Conv2DTranspose(128, (3,3), strides=(2,2), padding='same')(gen)
    gen = ELU(alpha=1.0)(gen)

    gen = Conv2DTranspose(64, (3,3), strides=(2,2), padding='same')(gen)
    gen = ELU(alpha=1.0)(gen)
    # Output
    out_layer = Conv2D(1, (7,7), activation='tanh', padding='same')(gen)
    # Define model
    model = Model([in_lat, in_label], out_layer)

    return model
```

Here we have the actual CNN architecture for the generator. In total there are three layers with 2DTranspose ConvNet [3], also here we have a kernel size of 3x3, stride=(2,2) and padding="same".
As activation function I just used a ELU as for the discriminator. In general I tried to follow this paper [3], an architecture guidelines for stable DCGANs. Ideally one should use Batch-Normalization both in the discriminator and generator, but during training I found some problem in the loss that went rapidly to zero, therefore collapsing the equilibrium model.

4.3 GAN

Here is now the composition of the two architectures:

```
# Define the combined generator and discriminator model, for updating the generator
def define_gan(gen_model, disc_model):

    # Make weights in the discriminator not trainable
    disc_model.trainable = False
    # Get noise and label inputs from generator model
    gen_noise, gen_label = gen_model.input
    # Get image output from the generator model
    gen_output = gen_model.output
    # Connect image output and label input from generator as inputs to discriminator
    gan_output = disc_model([gen_output, gen_label])
    # Define gan model as taking noise and label and outputting a classification
    model = Model([gen_noise, gen_label], gan_output)
    # Compile model
    opt = Adam(learning_rate = 0.0002, beta_1=0.9, beta_2=0.999)
    model.compile(loss='binary_crossentropy', optimizer=opt)

    return model
```

The GAN model essentially take as input a point in latent space and a class label and generate an image using the generator network, then it tries to predict if the input is real or fake using the discriminator model. As we can see, by creating an architecture like this, we are able to pass the same class label input both to the generator and the discriminator, this is important if we want to create an actual conditioning in the generation of a fake image. So the main role of the *define_gan()* function is to train the generator alone to create conditionated fake images.

[3] Here the Conv2DTranspose, also called *deconvolution*, is used to upsample the input feature map.
A simple explanation on how it work can be found here: Transposed Convolution Demystified.

4.4 Load real samples and fake samples

Here I show the functions used to generate real samples and fake samples that must be send to the discriminator.

```
# Load real images
def load_real_samples():

    # Save as a tensor called trainX the 2D Ising lattices
    trainX = data
    # Save as an array called trainy the label of each Ising lattices
    y = np.ravel(cat)
    # Expand to 3D the trainX tensor adding a channel
    X = expand_dims(trainX, axis=-1)

    return [X, y]

# Select real samples
def generate_real_samples(dataset, n_samples):

    # Split into images and labels
    images, labels = dataset
    # Choose a random index
    idx = randint(0, images.shape[0], n_samples)
    # Select images and labels using the random index
    X,labels = images[idx],labels[idx]
    # Generate class labels
    y = ones((n_samples, 1))

    return [X, labels], y
```

In the code above we can see the two functions used to select the real images and labels that are going to the discriminator.
In the first one I just rename the datas and expand the dimension of the images. In the second one I select random samples in a batch and define the real class label for the discriminator which in this case is 1 since they are real images.

```
# Generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples, n_classes=100):

    # Generate points in the latent space
    x_input = randn(latent_dim * n_samples)
    # Reshape into a batch of inputs for the network
    z_input = x_input.reshape(n_samples, latent_dim)
    # Generate labels
    labels = randint(0, n_classes, n_samples)

    return [z_input, labels]

# Use the generator to generate n fake examples, with class labels
def generate_fake_samples(generator, latent_dim, n_samples):

    # Generate points in latent space
    z_input, labels_input = generate_latent_points(latent_dim, n_samples)
    # Predict outputs
    images = generator.predict([z_input, labels_input])
    # Create class labels
    y = zeros((n_samples, 1))

    return [images, labels_input], y
```

In the first function I generate the random latent points vector and the corresponding random labels. In the second one I use them to generate a fake image using the generator and I create the class label for the discriminator which in this case is 0 since the samples are fake.

4.5 Training algorithm

Finally the function that calls the training itself.

```
# Define the number of epoch
n_epoch = 1000
n_batches = 128

# Train the generator and discriminator
def train(gen_model, disc_model, gan_model, dataset, latent_dim, n_epochs=n_epoch, n_batch=n_batches):

    batch_per_epoch = int(dataset[0].shape[0] / n_batch)
    half_batch = int(n_batch / 2)
    # Create an empty array for the losses
    losses = []

    # Manually enumerate epochs
    for i in range(n_epochs):
        # Enumerate batches over the training set
        for j in range(batch_per_epoch):
            # Get randomly selected 'real' samples
            [X_real, labels_real], y_real = generate_real_samples(dataset, half_batch)
            # Update discriminator model weights
            disc_loss_real, _ = disc_model.train_on_batch([X_real, labels_real], y_real)
            # Generate 'fake' examples
            [X_fake, labels], y_fake = generate_fake_samples(gen_model, latent_dim, half_batch)
            # Update discriminator model weights
            disc_loss_fake, _ = disc_model.train_on_batch([X_fake, labels], y_fake)
            # Prepare points in latent space as input for the generator
            [z_input, labels_input] = generate_latent_points(latent_dim, n_batch)
            # Create inverted labels for the fake samples
            y_gan = np.ones((n_batch, 1))
            # Update the generator via the discriminator's error
            gan_loss = gan_model.train_on_batch([z_input, labels_input], y_gan)
            # Appen the loss of the three models every batch per epoch
            losses.append([disc_loss_real, disc_loss_fake, gan_loss])
            # Summarize loss on this batch
            print('Epoch: %d / %d, Batch: %d/%d, Loss_Real = %.3f, Loss_Fake = %.3f Loss_GAN = %.3f' %
                (i+1, n_epoch, j+1, batch_per_epoch, disc_loss_real, disc_loss_fake, gan_loss))

    # Save the generator, discriminator and gan models
    gen_model.save('cgan_generator.h5')
    disc_model.save('cgan_discriminator.h5')
    gan_model.save('cgan_gan.h5')

    return losses
```

Once defined the training algorithm, in which we have a summary of the losses for each batch per epoch, we can call the training itself defining some variables.

```
# Define the size of the latent space
latent_dim = 500
# Create the discriminator
disc_model = define_discriminator()
# Create the generator
gen_model = define_generator(latent_dim)
# Create the gan
gan_model = define_gan(gen_model, disc_model)
# Load image data
dataset = load_real_samples()
# Train model
loss = train(gen_model, disc_model, gan_model, dataset, latent_dim)
```

5 Results

In this section we can have a look at the results obtained after trained and fine-tuned the model. As seen in the chapter "Ising Model", once generated some lattices, we can study some statistical property with respect to the temperature. We'll see the magnetization curve, the magnetic susceptibility, the energy and the heat capacity of the generated lattices vs the real one (the one used during the training).
But first of all let's take a look at the generated data from $T = 0$ to $T = 5$:

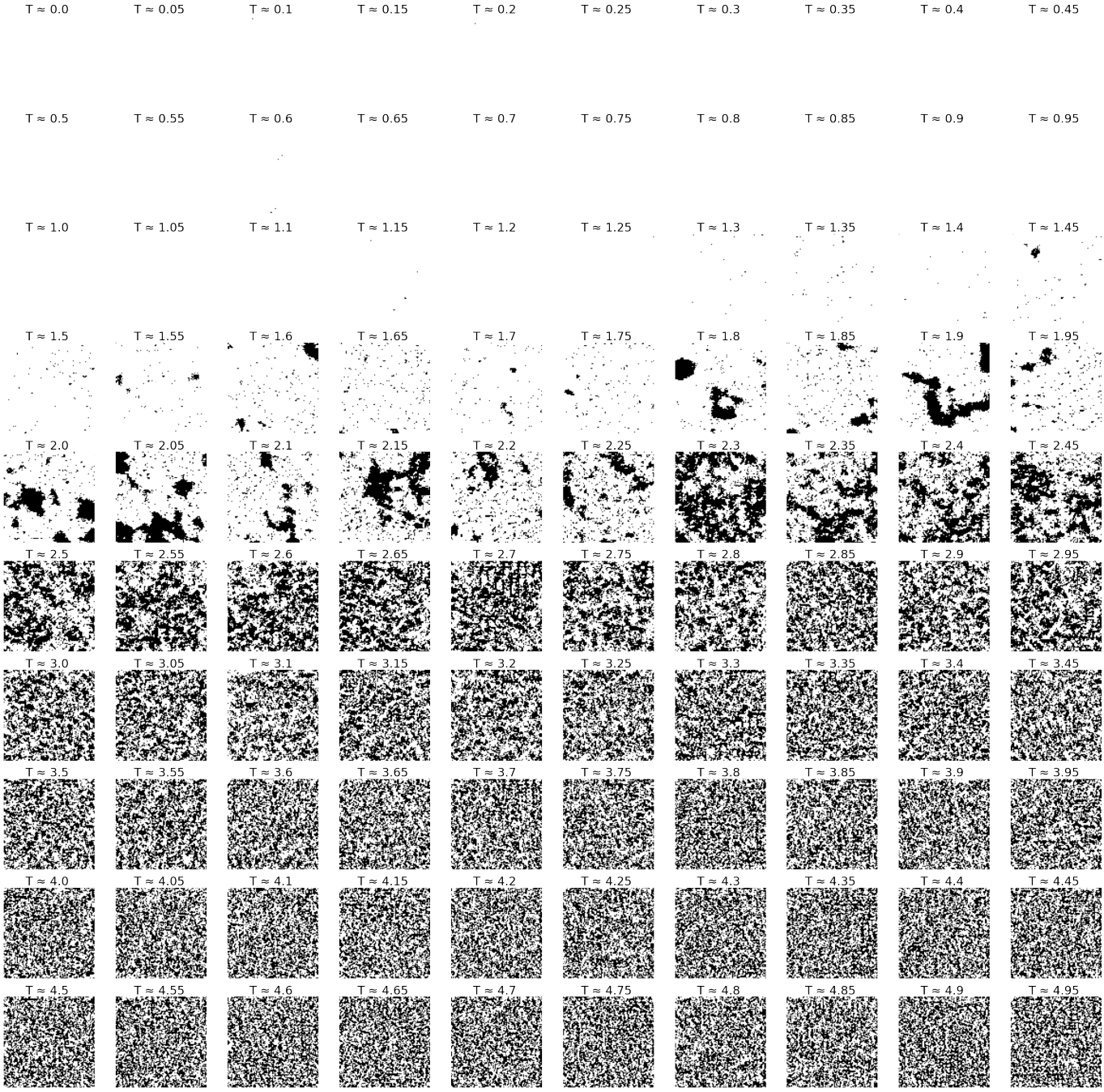

Figure 7: Generated lattices from $T = 0$ to $T = 5$.

5.1 Loss curves

A few words to discuss a little about the losses we can see as output of the training phase.

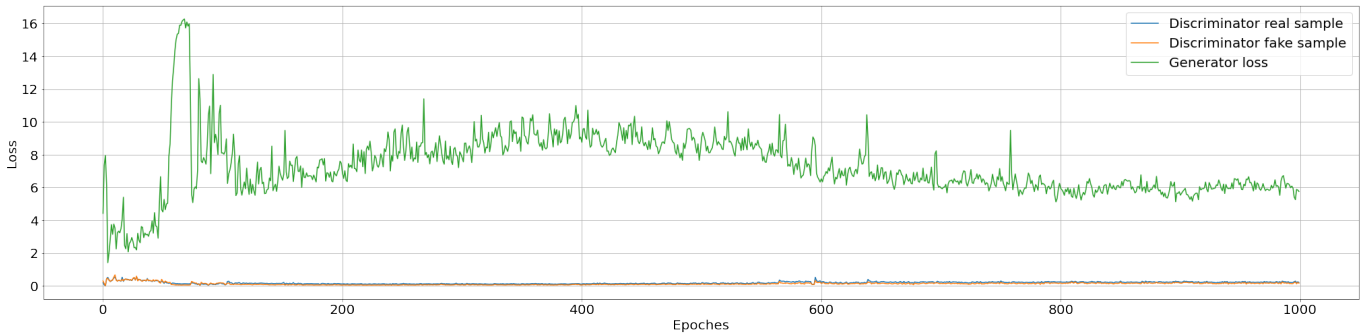

Figure 8: Losses of the generator and the discriminator w.r.t epochs.

Earlier it has been specified that the discriminator uses as metric the accuracy, this is important since we would like to see at convergence a trend in which the loss oscillates around the value of $1/2$. In the following plot we can have a look specifically at the discriminator loss:

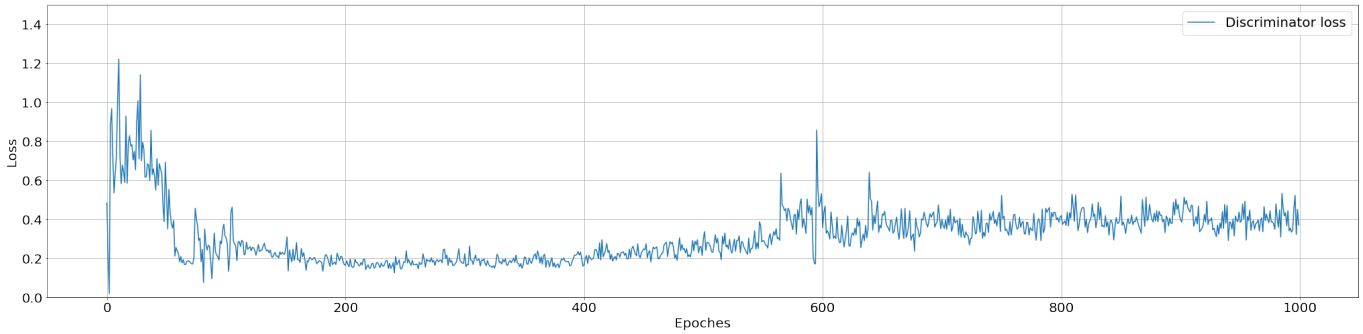

Figure 9: Discriminator loss w.r.t epochs.

As we can see, the value oscillates around 0.4, which is not exactly what we expect, even though the oscillations tend to go from 0.35 to 0.55. Certainly with more training epochs we could achive better results.
For the generator loss we have just the categorical cross entropy value for each epoch, and this can be high/low as it wants. As we know it is related to the generation process.
In general when we study the loss behavior for the generator and the discriminator, we cannot think of them as any other loss we can encounter in ML problem such as regression or classification. First of all it is important to observe an equilibrium during the training, a stationary behavior (exactly as we can see from the plot above). Secondly as just explained, we want that the discriminator starts to oscillate around 0.5, we want a randomic guess. This ensure that the generator is doing a good job in generating plausible new data. As we can see these requests are satisfied.

5.2 Magnetization

As we have seen, the magnetization is defined as the sum of all spins value of a given lattice, normalized by the number of spins. It is straightforward to implement computationally, and this is the result:

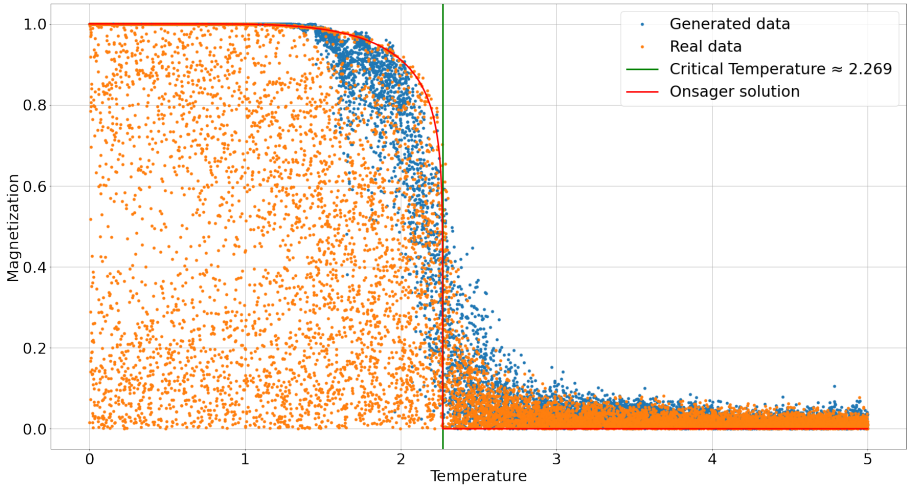

Figure 10: Here are presented the results of the magnetization for the real data, the generated data and the Onsager analytical result.

As we can see above the real data doesn't follow the Onsager curve, it could seem that something is not working here but it is not quite right. If we take a look at some data point, especially in the range of temperature from 0 to 2.3, we can see that if we calculate the sum of all spins it is absolutely possibile to obtain value far from the curve line and this is given by the intrinsic randomness of the model that define the flip of the spins.

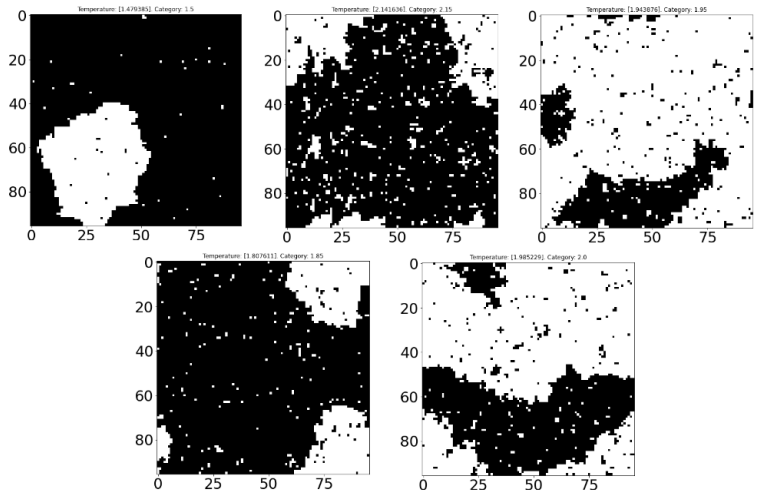

Figure 11: Five images in the range [0, 2.3] that show particular lattices in which the total magnetization is not at all along the Onsager solution.

We can say that lattices with this particular behaviour are particularly interesting because they show some ordered pattern like islands or stains. Also analyzing the plot above we can see that the generated images also can represent this kind of randomness but not quite well as the real one do, the generated data tends to stay near the analytical curve and this is interesting. The interesting fact is that in some sense the generator learned well the theoretical distribution the lattices should follow in the thermodynamic limit. Let's see some generated data in the range [0, 2.3]:

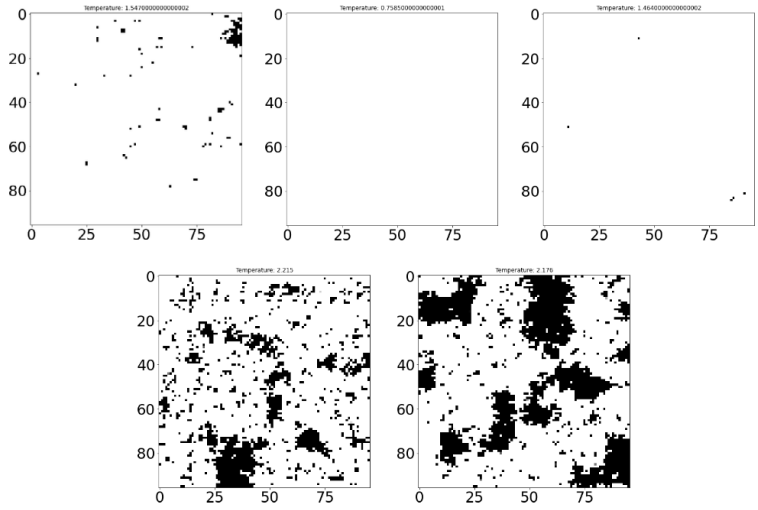

Figure 12: Five images in the range [0, 2.3] of the generated lattices.

As we can see the generator can produce realistic lattices but they are not good enough to capture some particular pattern observable in the dataset.
The last comment I would like to point out is that the Onsager solution shows a discontinuity point for T = 2.269, behavior not visible just by looking at real or generated data. This is because this particular discontinuous behavior is observable just in the thermodynamic limit, and of course this is not the case.

5.3 Magnetic susceptibility

Once obtained the magnetization we can easily pass to the magnetic susceptibility which is defined as the variance of the magnetization. Also here we can take a look at the behaviour of real and generated data:

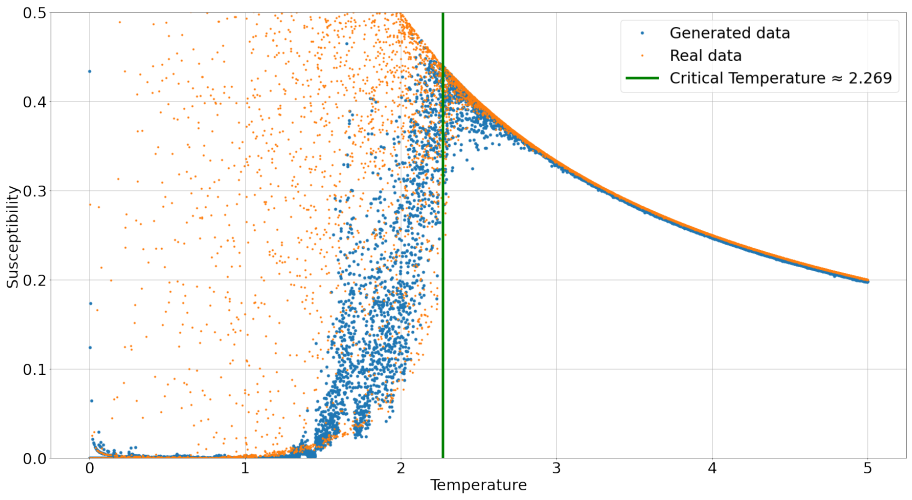

Figure 13: Magnetic susceptibility of real and generated data.

Also here real data shows a much more randomic behavior in the range of temperature between 0 and 2.3, while the generated one shows a much more theoretical coherent behavior. Here the theoretical view tells us that a divergent behavior is expected when $T \approx T_c$ as already said in the subsection related to the magnetic susceptibility. Of course this divergent behavior is "real" in the thermodynamic limit i.e. for $N \to \infty$ and $V \to \infty$, in our case we can at least expect a peak in proximity of $T \approx T_c$.

5.4 Energy

Another very important measure is the energy of every square lattice. As discussed above in the dedicated section, we expect a continuous transition from a value $E = -2$ to $E \to 0$ for $T \to +\infty$, passing by an inflection point for $T = 2.269$.
In the plot below we can see the behaviour for real and generated data:

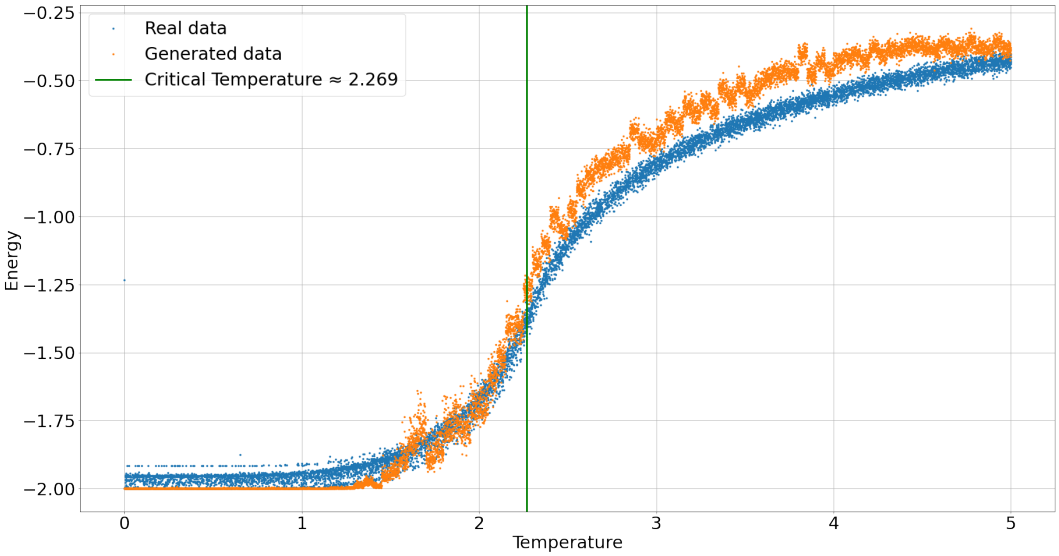

Figure 14: Energy plot for generated and real data.

As we can see from the plot above, the real data follows a perfect continuous path from the energy value -2 to something like -0.5, exactly like we expected.
The generated data follows not exactly the same curve but seems to respect the theoretical behavior. In particular we can see that for temperature between 0 and 1.4 the behavior is "too much perfect", in the sense that the model couldn't capture well the noisy pattern. The transition from -2 to -0.5 is well represented as well as the presence of the inflection point for $T = T_c$. For $T > T_c$ we can see higher energy value and a general trend in which the curve appear discontinuous almost everywere. This is easly explainable if we remember the discretization technique we used. Let's take a zoom of a particular region of the plot above:

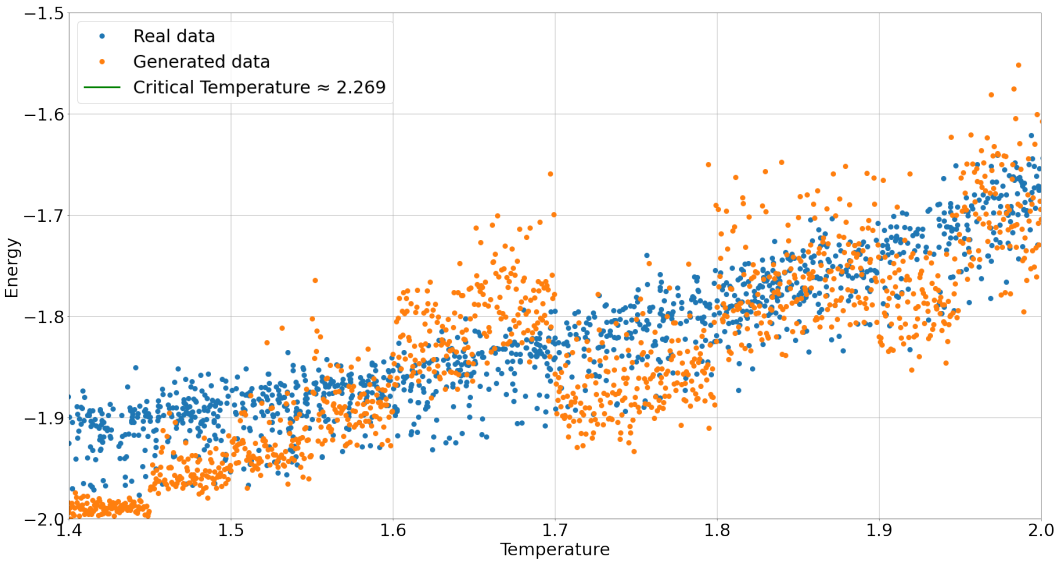

Figure 15: Zoom in the region of temperature between 1.4 and 2.

In order to conditionate the GAN, I discretized the temperature label in 100 classes. The discretization was carried out with a step size equal to 0.05, thus having a new label vector with element like: (1.25, 4.50, 2.34, ...). From the image above we can see that the discontinuity appear exactly with a step size of 0.05 in correspondence of temperature value like 1.45, 1.50, 1.6, 1.7 and so on. This argument justify this strange (in a sense odd) behaviur and give us a hint on how to overcome this problem. Since the main problem is the discretization technique, we can think of discretize the label more finely or direcly find a way to inject a continuous conditioning into the generator and the discriminator.

5.5 Heat capacity

For the last measurement we can have a look at the heat capacity as shown below:

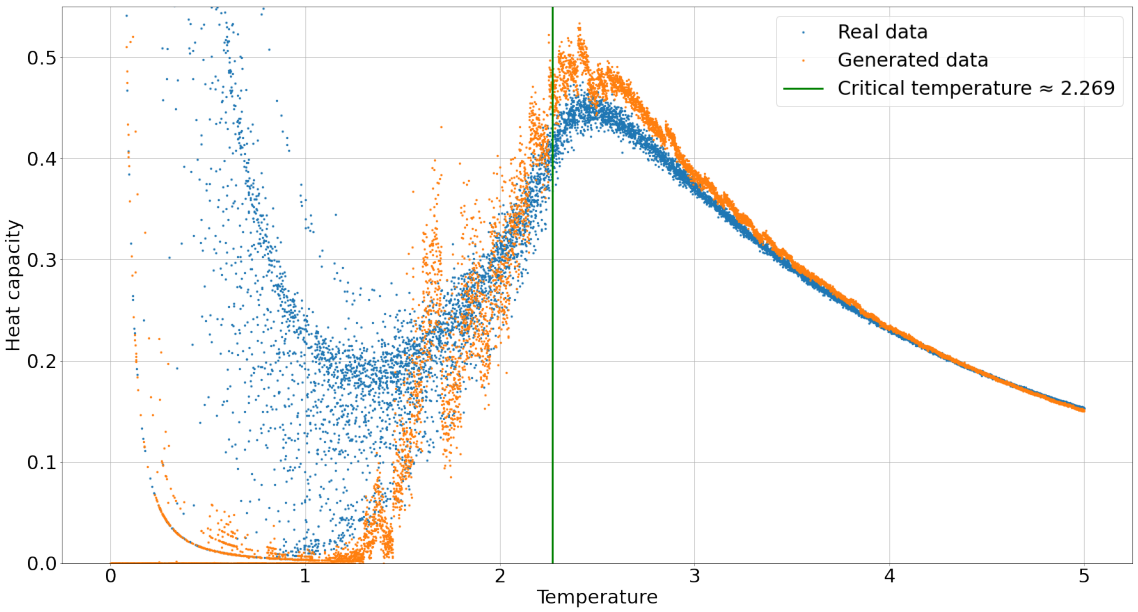

Figure 16: Heat capacity for real and generated data.

As we can see from the plot above, a nice peak is present near the critical temperature, but here it is even more visible the disalignment between the analytical critical temperature and the one observed by the data. This phenomenon is not surprising at all, in fact if we look at the other plots with care, we can certainly see this discrepancy every time. As always this is given by the fact that we are approximating an analytical result.
As last thing, we can see also in this plot the discretization effect I showed in the subparagraph above, and also here it is even more visible.

6 Conclusions

In this project I had the opportunity to work with a generative architecture I find particularly interesting: Generative Adversarial Network. The aim of the project was to better understand the concept behind this architecture and use it to test its general expressive capability. The problem I faced was the generation of 2D Ising lattices given specific temperature labels and test if the model was expressive enough to capture the interesting physical details. As we have seen, the model itself is indeed good enough to generate plausible conditioned lattices; also the physical quantity shown above are good enough for such a simple architecture. The interesting fact is that in some sense the generator can effectively capture not only the data distribution itself but also the theoretical background, which I think can be considered as a good starting point for more complex works.
One of the possible future work could be to test different conditioning, also concatenating more than one label (such as the quantities measured in the paper); injecting regularizer or maybe change the base idea and try to generate lattices starting from specific given initial random lattices using also here the temperature as constrain, thus doing the same work the Metropolis algorithm does.

References

[1] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, Yoshua Bengio *Generative Adversarial Nets*, (2014).

[2] Mehdi Mirza, Simon Osindero *Conditional Generative Adversarial Nets*, (2014).

[3] Alec Radford, Luke Metz, Soumith Chintala, *Unsupervised Representation Learning With Deep Convolutional Generative Adversarial Networks*, (2016).