3 Generative Adversarial Networks

Machine Learning is a field that aims to learn from data and make predictions. Machine Learning can be defined as a subfield of Artificial Intelligence with the goal of developing algorithms capable of learning from data automatically. The basic procedure for a Machine Learning method is to use input data and statistical analysis to produce an output. Meanwhile, the parameters of the model are updated to obtain better outputs evaluated in terms of a cost function.

The last decades have seen an increase in the ability to produce and analyzing big datasets thanks to an unprecedented rise in computational capabilities. This development sets Machine Learning as an exploitable tool in categorical problems and generative models. In particular, Physics plays a unique role because it contributes to Machine Learning, indeed many techniques came from ideas in statistics, neuroscience, and biophysics, and it benefits as well considering that physicists are also at the forefront of using "big data" in various branches like particle physics and statistical mechanics.

In this thesis, an innovative training process has been utilized called Generative Adversarial Networks (GAN). It consists of training two separate neural networks which are learning techniques that emerged as one of the most powerful based on nature's concept of the nervous system.

In this chapter, the standard setup for a Machine Learning problem is described. Then are presented the components of a neural network and its training process. Finally, these building blocks are combined to define a GAN and the particular architecture used in this thesis.

3.1 Introduction to Machine Learning

Machine Learning can be divided into three categories:

- Supervised learning;
- Unsupervised learning;
- Reinforcement learning.

notice that there is no distinct separation between these categories and many applications combine them in different ways.

In supervised learning, each input data is labeled with the correct value. The goal is to construct a mapping function that is able to give the correct output for new unseen data. This category is commonly used for categorical classification, where the labels are the possible classes for the problem (e.g. the nine classes for a written single-digit integer), or regression, where the labels are the values of the function that we wish to approximate.

On the other hand, unsupervised learning uses unlabeled data and tries to model the underlying structure or distribution of the input data. The most used applications are clustering, finding the inherent groupings of the data, and generative modeling, generate a sample with the same distribution of the input data. GANs lies in this category, and one of the most outstanding application is the generation of high-quality images of non-existent people from a sample of close ups.

Finally, reinforcement learning consists of an algorithm or agent, that learns by interacting with its environment. Given a task, the agent will receive rewards by performing correctly and penalties for incorrect moves. The result of the model is an agent that learns without intervention from a human by maximizing its rewards and minimizing its penalties.

The workflow for a Machine Learning problem always combines three steps:

- Choosing how to represent the model. For example, neural networks, decision trees, or graphical models;

- Evaluate the model using the appropriate cost function. Subsequently, the parameters of the model are updated accordingly to the minimization of this function. Some examples include: accuracy, Mean Squared Error (MSE), Mean Absolute Error (MAE), likelihood, entropy, and Kullback-Leibler divergence;

- The search process to update the model or optimization. The most used are derivative-based algorithms like Stochastic Gradient Descend (SGD) and its more effective implementations like Adam, Adadelta, or Adagrad.

Nevertheless, correctly training and optimizing a model is a difficult task. To obtain better results and to ease the convergence of the model is often necessary to transform, or preprocess, the data before using them as input, the choice of the correct preprocess depends on the representation used. Furthermore, all the parameters related to the model architecture and the training procedure, also called hyperparameters, should be optimized because they can heavily influence the model output.

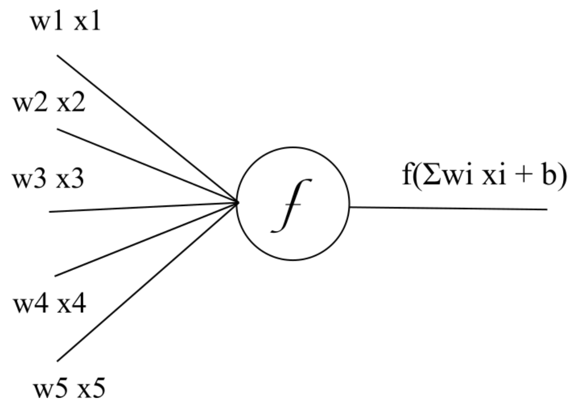


Figura 3.1: Artificial neuron with five inputs $x_1, \dots, x_5$. Each inputs has its weight $w_1, \dots, w_5$, after adding the bias b, the output value is obtained with the mapping function f as $O = f(\sum_i w_i x_i + b)$.

3.2 Neural networks

Nature's concept that neural networks are trying to imitate is the nervous system. The building block of the nervous system is a single neuron, which receives signals from other neurons and sends signals, through the axion, to another neuron.

Just like a neuron, an artificial neuron receives several inputs and produces a single output. In order to compute the output, to each input x_i is associated a weight w_i which express its importance. The output value O is not simply the sum of the weighted inputs, generally a bias weight b is added and mapped by a chosen function f called "activation function" $O = f(\sum_i w_i x_i + b)$. An example of an artificial neuron with five inputs is depicted in Fig. 3.1.

All the possible activation functions can be distinct in two groups: non-saturating functions with non-finite codomain like ReLU, Leaky ReLU, and linear, and saturating functions with possible saturation problems, caused by limited codomains, like Sigmoid, Tanh, and binary. The definitions of these functions are presented in Appendix B.1.

The combination of artificial neurons in different layers defines neural networks. The first and the last layers of the NN are respectively called input and output layers, while the layers in the middle are called hidden layers. Typically the input and output dimensions are fixed by the particular problem, for example, if the networks try to determine the digit in a handwritten image of $a \times b$ pixels, the input layer will have $a \times b$ neurons, one for each pixel, while the input layer 10 neurons, one for each digit.

The hidden layers that connect the network can be defined to obtain different architectures, here two of them are presented: a Feedforward Neural Network and

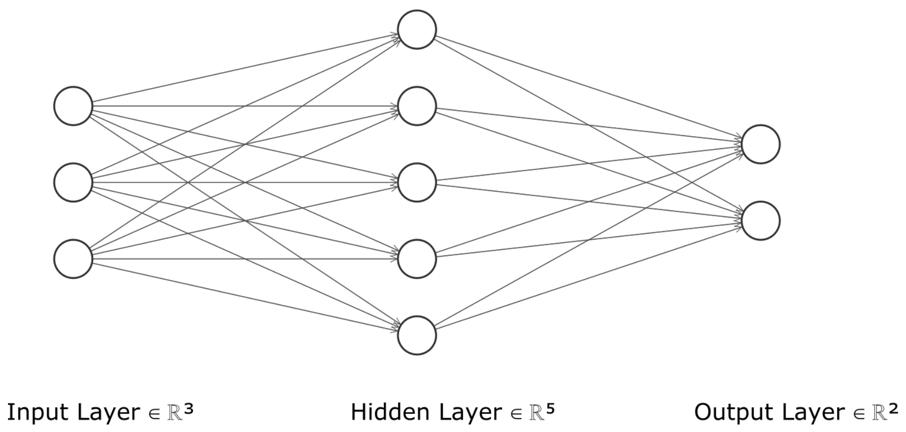


Figura 3.2: Example of the connections in a Feedforward Neural Network with one hidden layer.

a Convolutional Neural Network.

Feedforward Neural network
A FFNN is the simplest neural network architecture in which the data flows in one direction, from the input neurons to the output ones. Each node is connected to all nodes of the next layer creating a densely connected network. The importance of this kind of network concerns the universal approximation theorems which states that a FFNN with a single hidden layer can approximate any continuous function with arbitrary accuracy but with an exponentially large width. On the other hand, the representational power of the network increases also adding more hidden layers in a more computationally efficient way, this leads to the Deep Feedforward neural networks. An example of a FFNN is shown in Fig. 3.2.

Convolutional Neural Network
Architectures like a FFNN typically fail to exploits structures like locality and translational invariance which may be possessed by many datasets in Physics. To solve this problem, a particular class of neural networks has been developed initially for feature recognition in images called Convolutional Neural Networks (CNN). A 2D convolutional layer is characterized by the dimension of the spatial space (w, h) and by the number of channels c. The convolution consists of running several filters of fixed dimension over all locations in the spatial plane. Moreover, the stride s sets how many neurons of the convolutional layer the filter translates when performing the convolution. Finally, it is possible to pad the input with p zeros in order to get the desired output dimension. Fig. 3.3 shows an example convolution on one spatial plane.

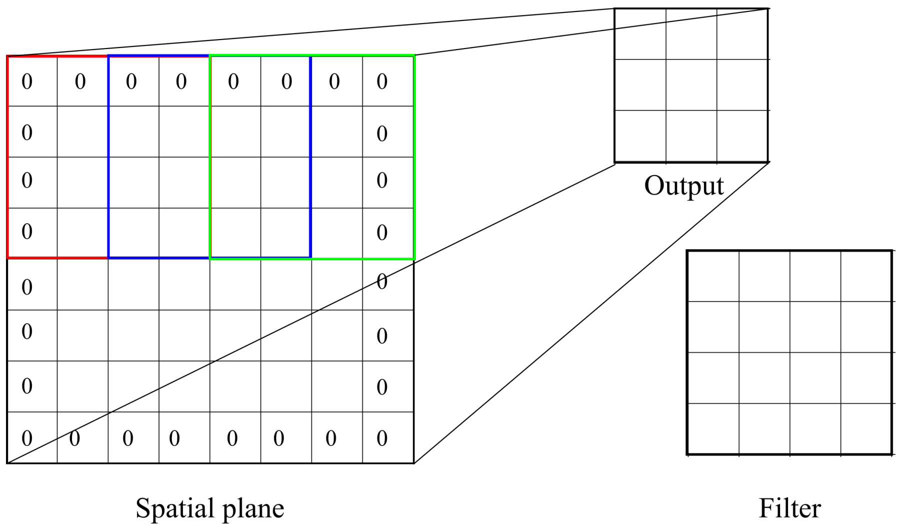


Figura 3.3: Example of convolution. The size of the filter is $(4, 4)$, the convolution is performed with strides 2 and 1 zero padding. The grids of different colors represent the filter moving in the spatial plane mapping the initial space to a single output. The final output is a grid of dimensions $(3, 3)$.

Given these informations, the number of output neurons in the layer is given by:

$$dim = \frac{w - f + 2p}{s} + 1 \tag{3.1}$$

while the number of filters is fixed by the user. An example of a CNN with a layer of 8 filters of dimension $(32, 32)$ coarse grained in 16 filters of dimension $(16, 16)$ is shown in Fig. 3.4. The filter is of size $(4, 4)$, with stride 2 and 1 zero padding.

The neural networks used in this thesis are Deep Convolutional Neural Networks and their detailed structure is described in Sec. 3.5.

3.3 Training a neural network

Once the architecture has been defined the neural network must be trained to optimize the weights for the specific problem. The training of a neural network requires steps that are common to the vast possible architectures:

- Construct a cost/loss function to minimize;
- calculate the gradients of the cost function with respect to all parameters via the “backpropagation algorithm”;
- use a gradient descent-based procedure to optimize all the weights and biases.

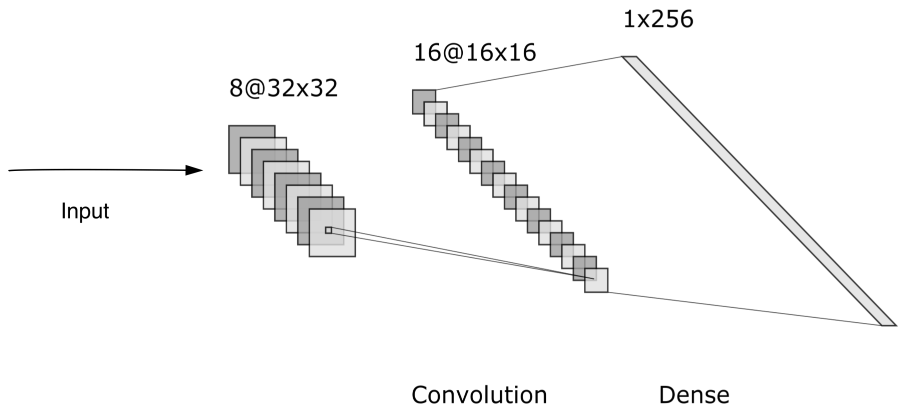


Figura 3.4: Example of a Convolutional Neural Network with two convolutional layers with 8 and 16 filters, see text for details, and a final densely connected layer with 256 nodes.

The loss function is mainly fixed by the output layer. Given a batch size N of expected values $\bar{y}_i$, a neural network with parameters $\bar{w}$ and predicted values $f(\bar{y}_i;\bar{w})$, for continuous data the commonly used functions are the Mean Squared Error (MSE) and the Mean Absolute Error (MAE):

$$L_{MSE}(\bar{w}) = \frac{1}{N}\sum_{i=1}^{N} |\bar{y}_i - f(\bar{y}_i;\bar{w})|^2 \tag{3.2}$$

$$L_{MAE}(\bar{w}) = \frac{1}{N}\sum_{i=1}^{N} |\bar{y}_i - f(\bar{y}_i;\bar{w})| \tag{3.3}$$

Instead, for categorical data, when the output layer is a classifier with two (binary) or more labels (softmax), the most used loss function is the Cross-Entropy (CE). For binary labels, where $y_i \in \{0,1\}$, the CE is defined by:

$$L_{CE}(\bar{w}) = -\sum_{i=1}^{N} y_i \log f(y_i;\bar{w}) + (1-y_i)\log\left[1 - f(y_i;\bar{w})\right] \tag{3.4}$$

Fixed the loss function, a specialized algorithm called backpropagation is used to determine the gradients of the loss function. To see this algorithm in actions, consider a FFNN with L layers indexed by l. Define the weight, which connect the k-th neuron in layer l-1 to the j-th neuron in layer l, w^l_{jk} and the bias b^l_j. The next steps is to relate the output of the j-th neuron of the l-th layer a^l_j to the outputs of the previous layer:

$$a^l_j = f(\sum_k w^l_{jk} a^{l-1}_k + b^l_j = f(z^l_j) \tag{3.5}$$

where f is the activation function and z_j^l is defined by the linear weighted sum. Moreover define the variation of the j-th neuron in the l-th layer Δ_j^l as the variation in the cost function L:

$$\Delta_j^l = \frac{\partial L}{\partial z_j^l} \tag{3.6}$$

where the derivative of L and f must be known.

It is possible to find four equations that relate the activations a_j^l, the weighted inputs z_j^l and the variations Δ_j^l of all the neurons. The first is the Eq. 3.6, the second relates the cost function and the biases:

$$\Delta_j^l = \frac{\partial L}{\partial z_j^l} = \frac{\partial L}{\partial b_j^l}\frac{\partial b_j^l}{\partial z_j^l} = \frac{\partial L}{\partial b_j^l} \tag{3.7}$$

The third and the fourth equation are defined via chain rule of Δ_j^l and $\frac{\partial L}{\partial w_{jk}^l}$:

$$\Delta_j^l = \sum_k \frac{\partial L}{\partial z_k^{l+1}}\frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k \Delta_k^{l+1}\frac{\partial z_k^{l+1}}{\partial z_j^l} = (\sum_k \Delta_k^{l+1} w_{jk}^{l+1}) f'(z_j^l) \tag{3.8}$$

$$\frac{\partial L}{\partial w_{jk}^l} = \frac{\partial L}{\partial z_j^l}\frac{\partial z_j^l}{\partial w_{jk}^l} = \Delta_j^l a_k^{l-1} \tag{3.9}$$

The equations Eqs. 3.6, 3.7, 3.8, 3.9 are combined to get a backpropagation algorithm to calculate the gradients of the loss function:

– Calculate the activations a_j^1 of all the neurons in the input layer;

– compute z^l and a^l for each subsequent layer;

– calculate the error of the top layer using Eq. 3.6

– use Eq. 3.8 to propagate the variation backwards and calculate Δ_j^l for all layers;

– use Eqs. 3.7, 3.9 to calculate $\frac{\partial L}{\partial w_{jk}^l}$ and $\frac{\partial L}{\partial b_j^l}$.

The power of this algorithm allows to calculate all the derivatives of the loss function with only a "forward" and a "backward" pass of the neural network.

Finally, the parameters are updated by a gradient descent-based algorithm. The simplest method is the Stochastic Gradient Descent (SGD), the basic idea is to take, at each iteration, steps proportional to the negative gradient of the loss function because this direction defines the fastest way to decrease L. Given the parameters $\bar{w}$ and the loss function $L(\bar{w})$, the update is done according to the equation:

$$\bar{w}_{n+1} = \bar{w}_n - \eta\bar{\nabla}L(\bar{w}_n) \tag{3.10}$$

where n is the training iteration and η is a hyperparameter fixed by the user called "learning rate". The choice of η is fundamental because a small learning rate causes slow convergence, while a high learning rate may cause jumps around the minimum of the loss function. The stochastic nature of the algorithm is tied to the usage of a subsample, defined by the batch size, of the input data to evaluate the gradients.

3.4 Generative Adversarial Networks

A Generative Adversarial Network (GAN) [**goodfellow**] is composed of two neural networks called Generator and Discriminator. The two networks compete against each other in a zero-sum game where an improvement for one agent implies the decay of the other. The task of the generator is to reproduce the distribution of the input data $p_{data}(\bar{x})$, to do so an input random noise $p_z(\bar{z})$ is mapped by the generator to the data space $G(\bar{z};\bar{w}_g)$ where $\bar{w}_g$ are the parameters of the network, resulting in the distribution $p_{gen}(\bar{x})$.

Instead, the discriminator outputs a single scalar $D(\bar{x};\bar{w}_d)$ that represents the probability that $\bar{x}$ belong to the input data or the generated sample. The training simultaneously optimize D, maximizing the accuracy of the assignment of the labels, and G, minimizing the quantity $\log(1-D(G(\bar{z})))$ which is the probability to correctly recognize generated samples by the discriminator.

This algorithm defines a two-player minmax game with loss function:

$$\underset{G}{min}\,\underset{D}{max}\,L(D,G) = E_{\bar{x}\sim p_{data}(\bar{x})}\left[\log D(\bar{x})\right] + E_{\bar{z}\sim p_z(\bar{z})}\left[\log(1-D(G(\bar{z})))\right] \tag{3.11}$$

This defined game has a global minimum for $p_{gen} = p_{data}$. To show that, let's prove that for G fixed the best D is the quantity:

$$D^*(\bar{x}) = \frac{p_{data}(\bar{x})}{p_{data}(\bar{x}) + p_{gen}(\bar{x})} \tag{3.12}$$

by definition of the minmax game in Eq. 3.11 the procedure for D is to maximize $L(D)$ which can be written as:

$$L(D,G) = L(G) = \int_{\bar{x}} p_{data}(\bar{x})\log(D(\bar{x})) + p_{gen}(\bar{x})\log(1-D(\bar{x}))dx \tag{3.13}$$

the maximum in $[0,1]$ of the integrating function corresponds to $\frac{p_{data}(\bar{x})}{p_{data}(\bar{x})+p_{gen}(\bar{x})}$.

For $p_{gen} = p_{data}$, $D^*(\bar{x}) = 1/2$ and $L(D^*,G) = L(G) = -\log 4$. To prove that this is the global minimum of the loss function add and subtract to L the quantity:

$$E_{x\sim p_{data}}\left[-\log 2\right] + E_{x\sim p_{gen}}\left[-\log 2\right] = -\log 4 \tag{3.14}$$

With this manipulation the loss function can be written as:

$$
\begin{aligned}
L(G) &= -\log 4 + E_{x\sim p_{data}}\left[\log(2D^*(\bar{x}))\right] + E_{x\sim p_{gen}}\left[\log(2-2D^*(\bar{x}))\right] \\
&= -\log 4 + E_{x\sim p_{data}}\left[\log(2\frac{p_{data}(\bar{x})}{p_{data}(\bar{x})+p_{gen}(\bar{x})})\right] + E_{x\sim p_{gen}}\left[\log(2\frac{p_{gen}(\bar{x})}{p_{data}(\bar{x})+p_{gen}(\bar{x})})\right] \\
&= -\log 4 + KL(p_{data}||\frac{p_{data}+p_{gen}}{2}) + KL(p_{gen}||\frac{p_{data}+p_{gen}}{2}) \\
&= -\log 4 + 2\cdot JS(p_{data}||p_{gen})
\end{aligned}
\tag{3.15}
$$

where the Kullback-Leibler (KL) and the Jensen-Shannon (JS) divergences have been defined such that:

$$KL(p||q) = \int_x p(x)\log\left(\frac{p(x)}{q(x)}\right) \tag{3.16}$$

$$JS(p||q) = \frac{1}{2}\left(KL(p||\frac{p+q}{2}) + KL(p||\frac{p+q}{2})\right) \tag{3.17}$$

where $p(x)$ and $q(x)$ are probability distribution functions. The KL and the JS divergences are measures of how much two distributions are different. In particular, notice that the JS divergence is always positive and zero when the two distributions are identical. Thus, the global minimum of $L(D^*, G)$ is $-\log(4)$ when the true and generated distributions match $p_{gen} = p_{data}$.

The workflow for training a GAN can be summarized in these steps:

– Some random noise is generated from a chosen distribution (e.g. uniform or Gaussian);

– the generator produces a sample from the random noise;

– the generated and the true samples are feed into the discriminator who outputs 0 for the fake sample and 1 for the true one;

– finally, the parameters of the neural networks are updated via backpropagation.

A chart of the workflow of a GAN is shown in Fig. 3.5. However, even if the defined algorithm used to train a GAN is simple, GANs have failure modes that make the achievement of good training a hard process. The most common failures are:

- Mode collapse, when the generator learns to produce a small set of outputs. In response, the discriminator learns only to reject these particular outputs;

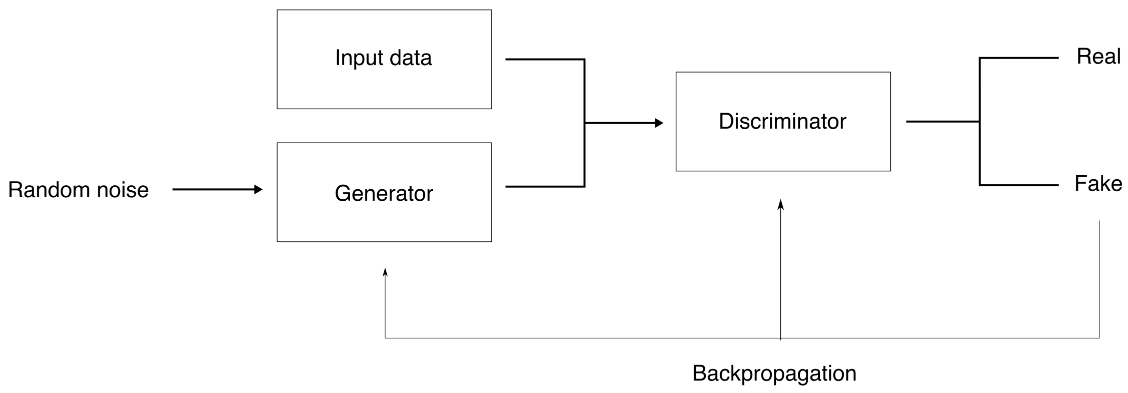


Figura 3.5: Workflow of a GAN. The generator produces samples from random noise trying to imitate the input data, while the discriminator discerns the true samples from the fake ones. The parameters of the networks are updated via backpropagation from the discriminator to the generator.

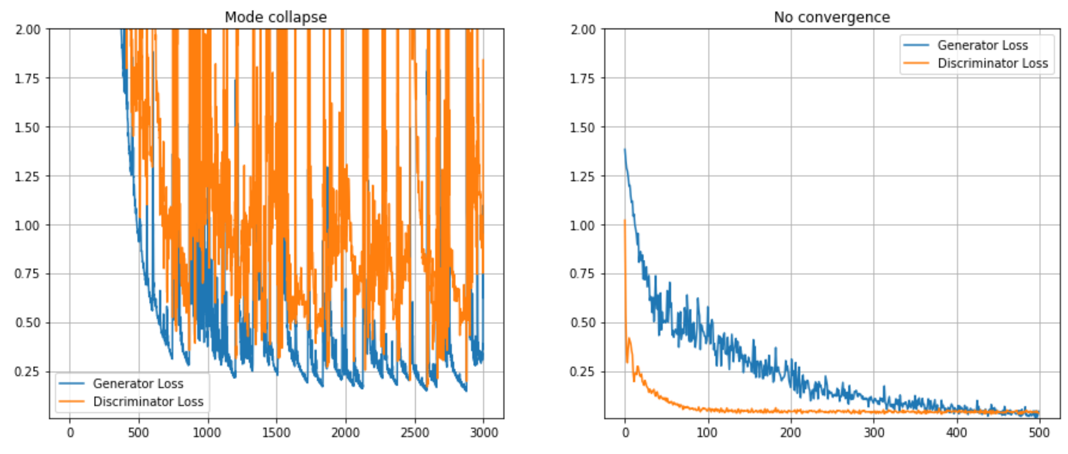
- convergence failure is referred to as the inability of the GAN to converge at an equilibrium point.

None of these problems have been fully solved and only attempts can be done to resolve these issues depending on the particular training. A way to identify these failures is to watch the quantities that each network optimizes. In a mode collapse, both losses present fluctuations whose minimums correspond to the specialization of a single set output. Instead, with convergence failure one loss function, typically the discriminator, goes to zero while the other one rise meaning that one neural network is too good respect the other, or both go to zero. In Fig. 3.6 are shown the loss function of the failure modes encountered in this thesis.

3.5 GAN architecture

This section will describe the architecture utilized in this thesis which emerged as the one that allowed to resolve the training issues of the previous section. The chosen architecture is a Deep Convolutional GAN (DCGAN) where the two neural networks are deep convolutional neural networks.

Generator
The generator has 125835 trainable parameters and is formed by the following layers:



(a) (b)

Figura 3.6: Failure modes encountered. (a) Mode collapse: the loss functions keep fluctuating without stabilize. (b) Convergence failure: the GAN fail to converge.

- Two **Dense** layers, the first used to add the required neurons to reshape the input in a 2D 10×10 spatial plane and the last used to obtain the desired number of outputs from the generator.
- Four **Transposed convolution** layers using several filters decreasing from 128 to 16, they can be seen as convolutional layers where the filters are applied to the output to get the spatial plane;
- Each layer is followed by a **Batch Normalization** layer, they normalize the inputs to a zero-mean and unit variance distribution. The task of these layers is to stabilize the training and prevent possible failures;
- **Leaky ReLU** layers with $\alpha = 0.2$ used to apply this activation function to the outputs of batch normalization;
- A **Flatten** layer is used to transform the 2D spatial plane back to an array of neurons.

The generator training is done by initially sampling an array of size 100 from a uniform distribution. Then, the noise is used as input in the network formed by the union of generator and discriminator where only the parameters of the generator are trainable. Finally, the binary cross-entropy between the outputs and an array of ones, corresponding to the probability of predicting the generated sample as true, is calculated and used for the weights update. The output layers is activated, depending on the used preprocessing, by a linear function or a hyperbolic tangent.

Discriminator

The discriminator has 101513 trainable variables and is formed by the following layers:

- Two **Dense** layers, the single scalar output, activated by a sigmoid function, and the initial layer to reshape in a spatial plane of the same dimension of the generator;
- Four **Convolutional** layers with a decreasing number of filters from 128 to 16;
- **LeakyReLU** layers to apply the activation function;
- One **Dropout** layers with rate= 0.2 used to prevent overfitting. It sets to 0 the input neurons with probability given by the rate.

The discriminator training is done by calculating the binary cross-entropy of a generated sample with an array of zeros and the cross-entropy of a true sample with an array of ones.

This GAN has been implemented and trained using the software library `Keras` [**keras**] with `TensorFlow 2.3`[**tensorflow**] backend. A summary of both networks alongside a visual representation is shown in Figs. 3.7, 3.8, 3.9.

```
Model: "Generator"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 200)               20200
_________________________________________________________________
batch_normalization (BatchNo (None, 200)               800
_________________________________________________________________
leaky_re_lu (LeakyReLU)      (None, 200)               0
_________________________________________________________________
reshape (Reshape)            (None, 10, 10, 2)         0
_________________________________________________________________
conv2d_transpose (Conv2DTran (None, 10, 10, 128)       2304
_________________________________________________________________
batch_normalization_1 (Batch (None, 10, 10, 128)       512
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 10, 10, 128)       0
_________________________________________________________________
conv2d_transpose_1 (Conv2DTr (None, 10, 10, 64)        73728
_________________________________________________________________
batch_normalization_2 (Batch (None, 10, 10, 64)        256
_________________________________________________________________
leaky_re_lu_2 (LeakyReLU)    (None, 10, 10, 64)        0
_________________________________________________________________
conv2d_transpose_2 (Conv2DTr (None, 10, 10, 32)        18432
_________________________________________________________________
batch_normalization_3 (Batch (None, 10, 10, 32)        128
_________________________________________________________________
leaky_re_lu_3 (LeakyReLU)    (None, 10, 10, 32)        0
_________________________________________________________________
conv2d_transpose_3 (Conv2DTr (None, 10, 10, 16)        4608
_________________________________________________________________
batch_normalization_4 (Batch (None, 10, 10, 16)        64
_________________________________________________________________
leaky_re_lu_4 (LeakyReLU)    (None, 10, 10, 16)        0
_________________________________________________________________
flatten (Flatten)            (None, 1600)              0
_________________________________________________________________
dense_1 (Dense)              (None, 3)                 4803
=================================================================
Total params: 125,835
Trainable params: 124,955
Non-trainable params: 880
_________________________________________________________________
```

Figura 3.7: Generator structure.

```
Model: "Discriminator"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_2 (Dense)              (None, 200)               600
_________________________________________________________________
reshape_1 (Reshape)          (None, 10, 10, 2)         0
_________________________________________________________________
conv2d (Conv2D)              (None, 10, 10, 128)       2432
_________________________________________________________________
leaky_re_lu_5 (LeakyReLU)    (None, 10, 10, 128)       0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 10, 10, 64)        73792
_________________________________________________________________
leaky_re_lu_6 (LeakyReLU)    (None, 10, 10, 64)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 10, 10, 32)        18464
_________________________________________________________________
leaky_re_lu_7 (LeakyReLU)    (None, 10, 10, 32)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 10, 10, 16)        4624
_________________________________________________________________
flatten_1 (Flatten)          (None, 1600)              0
_________________________________________________________________
leaky_re_lu_8 (LeakyReLU)    (None, 1600)              0
_________________________________________________________________
dropout (Dropout)            (None, 1600)              0
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 1601
=================================================================
Total params: 101,513
Trainable params: 101,513
Non-trainable params: 0
_________________________________________________________________
```

Figura 3.8: Discriminator structure.

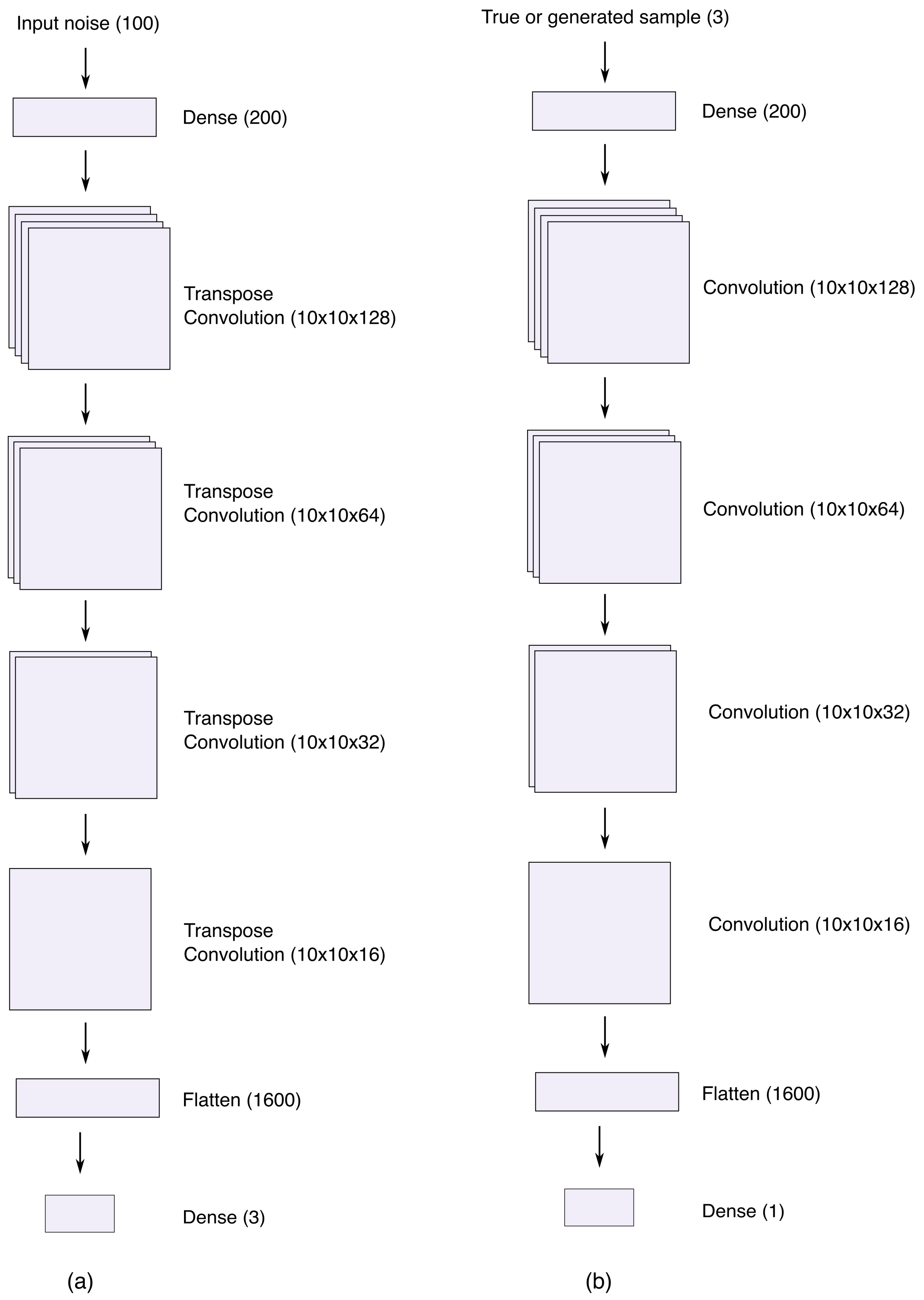


Figura 3.9: Visual representation of the GAN with layers dimension for the (a) Generator and (b) Discriminator.