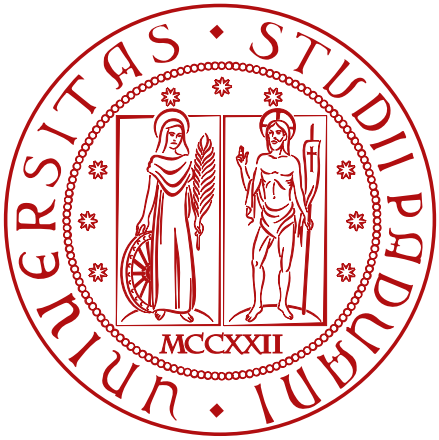
Digital Forensics Project

Generative Adversarial Network on Fashion MNIST and DeepFashion Datasets

Francesco Marchiori

ID = 2020389



2 July 2021

Contents

1 Introduction

A **Generative Adversarial Network** is a class of neural networks, first introduced by Ian Goodfellow in a homonym paper [1], that exploits concepts from Game Theory to train two models called **Generator** and **Discriminator** in order to generate images resembling the ones given in the training dataset. Actually, only the generator model is in charge of the creation of the new data instances, while the discriminator is another neural network that learns how to distinguish real images from fake ones. More in particular, the generator represents a differentiable function $G(z;\theta_G)$ where z are latent spaces with distribution $p_z(z)$ and θ_G are the parameters of the generator: it takes in input some random noise that represent the latent space and gives in output an image according to a distribution p_G. The discriminator instead represents another differentiable function $D(x;\theta_D)$ where x are the images and θ_G are the parameters of the discriminator: it takes in input images from the training dataset coming from a distribution p_{DS} and the generator output ($x \sim p_G$) and learns how to distinguish their authenticity, giving in output the probability that a given input x comes from the legitimate training dataset.

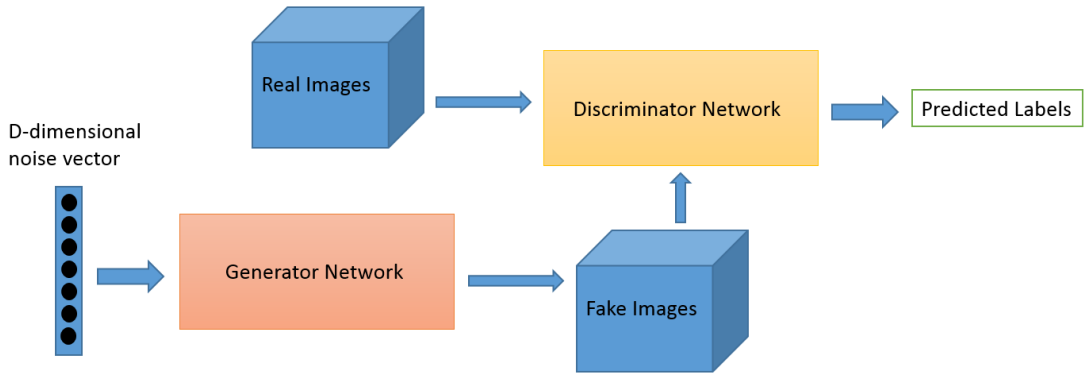


Figure 1: Structure of a Generative Adversarial Network. Image from [2].

1.1 Loss Function

As stated in Sec. (1), the Generative Adversarial Network model (which will be called GAN for short) uses concepts from Game Theory in order to define loss functions and train the overall model to perform its specific task. The overall objective of the model is to generate images that are very similar to the ones that are present in the dataset, therefore the generator must create data instances that are able to fool the discriminator into thinking that they come from the probability distribution that it learned from the training process: in mathematical terms this means that the aim of the generator is to obtain that $D(G(z)) = 1$, where $D(x)$ is the discriminator output, $G(x)$ is the generator output and $D(G(x))$ is the discriminator estimated probability of the generator's output being real. On the contrary, while creating convincing images is the aim of the generator, the discriminator is trained in order to distinguish examples coming from different distributions from the one of the dataset, thus trying to obtain $D(G(z)) = 0$.

This contrast between the task given to the two models of the overall network is called a **Zero-Sum Von-Neumann Game**, which represent a situation in which the advantage obtained by one party corresponds to the disadvantage inflicted to the other party. By defining the loss of the overall model comprising of the generator and the discrimiator as:

$$L(D,G) = \underset{x\sim p_r}{\mathbb{E}}\left[\log D(x)\right] + \underset{z}{\mathbb{E}}\left[\log(1 - D(G(z)))\right] \tag{1}$$

learning then is implemented by the optimization of a *minmax* game between the two models,

where the generator wants to minimize this loss but the discriminator wants to maximize it, thus obtaining:

$$\min_G \max_D \mathop{\mathbb{E}}_{x\sim p_r} \left[\log D(x)\right] + \mathop{\mathbb{E}}_{z} \left[\log\left(1 - D(G(z))\right)\right] \tag{2}$$

It's possible to notice that the loss function has a global minimum when $p_G \sim p_{DS}$, and therefore the model can converge to a situation in which the generator is actually able to generate convincing images.

1.2 Training and Backpropagation

After defining a loss function for the model, it's now important to define a way to implement learning: indeed, just defining the function to minimize/maximize is not enough, because it's crucial to decide a way to update the model parameters in order to "keep" the memory of what it learned in previous iterations. In Machine Learning and more in particular in Deep Learning, this is implemented with a technique called **Backpropagation**; the execution of the model is divided in two parts: the *forward propagation* phase in which the input is propagated through the network and produces the network output and the relative cost function, and the *back propagation* phase in which the information from the cost flows backward to compute the gradient for each parameter (which for deeper models represents the derivative chain rule of calculus).

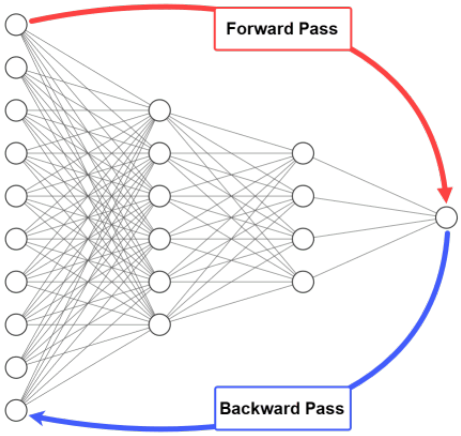


Figure 2: Visualization of the two phases of learning. Image from [3].

It's important to clarify that Backpropagation is just a way to compute the gradients of the parameters of a model (namely the weights), while there are other algorithms that performs learning using gradient: examples are **Gradient Descent** (GD) and its counterpart **Stochastic Gradient Descent** (SGD), where in the former the gradient of the loss on the whole training set is computed and then the weights are updated and in the latter the gradient of the loss a single example of the training set is computed and then the weights are updated (tradeoff between the two is called *Mini-Batch Stochastic Gradient Descent*). However, in the last years another optimizer has been developed and has become very popular among data scientists and it's the **Adam** optimizer [4], which makes use of the first and second moment in order to adapt the learning rate for each network parameter as learning unfolds. Since this optimizer has been empirically proven to be particularly suited for Deep Learning task, it will be the one that will be used in the GAN model presented in this report.

1.3 Convolutional Neural Networks

After having defined the loss function and the optimization and learning algorithm of the network, it's now important to have a look at the structure of it. The decision on the type of layers to use is strictly dictated by the task that the model is given: since both the generator and the discriminator have to

work with fixed-size images, the best solution are the **Convolutional Layers**. These are specialized layers for processing grid-like structures (in the case of images, 2D grid structures) which are inspired by neuroscience (namely the concept of visual cortex) and based on the mathematical operation of convolution, an operation between two functions that consists in integrating the product between one of them and the other translated of a certain value.

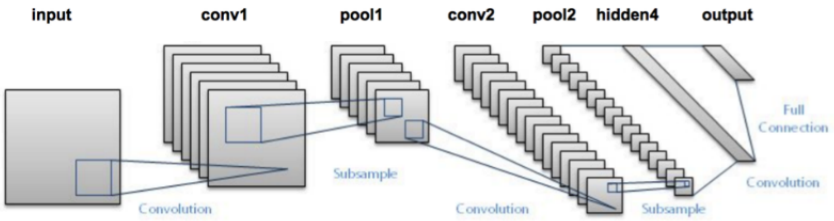


Figure 3: Typical structure of a Convolutional Neural Network. Image from [5].

These layers can be used to implement feature extraction in a network or for image recognition. In the generator and discriminator models, Convolutional Layers will be used to perform upsampling and downsampling of the images.

2 Experimental Setup

In order to build the Generative Adversarial Networks in practice, a programming language called *Python* has been used and, more in particular, some libraries for Deep Learning and neural network generation called *TensorFlow* and *Keras*. Their versions are, respectively, the 2.4.0 version and 2.4.3 which are not the latest one on the market, but are the ones that allowed to run the programs locally on a NVIDIA RTX 2070 and a RYZEN 7 3700x: the program has been run mainly on the GPU in order to exploit the advantage given by it on the Convolutional Layers and the use of Google Colab has been neglected because of its limited memory (while the local machine has 16GB of memory). All of the environments necessary in order to run the programs have been provided through Anaconda, a Python distribution platform. The program has been written in a Jupyter Notebook, an open document format that contains both code and text in order to clarify some steps and visualize the output of the model in real time.

3 Generative Adversarial Network on Fashion MNIST Dataset

In this section, a GAN that takes in input examples from the Fashion MNIST Dataset [1] will be created. The **Fashion MNIST** is a dataset of Zalando's article images consisting of a training set of 60,000 examples and a test set of 10,000 examples where each example is a 28x28 grayscale image, associated with a label from 10 classes (the classes are T-shirt/Top, Trouser, Pullover, Dress, Coat, Sandal, Shirt, Sneaker, Bag and Ankle boot). This dataset can be imported in various ways inside the Jupyter Notebook, for example by importing images one by one or by importing directly the `.csv` file present in the website, but Keras include a function call that allows for the import of the dataset in a more convenient way with the command `fashion_mnist.load_data()`, which will be used in order to import just the training data (used to train the GAN) and the label of the training data (used to name the plots of the model). In Fig. (4) it's possible to see the first 50 examples of the dataset.

The images of the dataset are then divided into batches of size 256 and they are reshaped and normalized in order to have pixel values in the range $[-1,1]$. This is done because originally the pixel values were in the range $[0,255]$ which is not ideal for neural networks, while with the new range it's possible to use an appropriate activation function for the layers in order to have a coherent output.

[1] https://www.kaggle.com/zalando-research/fashionmnist

Figure 4: First 50 elements of the Fashion MNIST Dataset.

3.1 Generator

The generator, as anticipated in Sec. (1.3), makes use of Convolutional Layers in order to generate an image from a noise vector. The image in output must be coherent with the images in the training set and therefore they will be 28x28 images with pixel value in the range $\in [-1,1]$, while it takes in input a noise vector belonging to a latent space. The size of this vector has been chosen to 100. The first layer of the network is a `Dense` layer which represent the standard fully connected layer in a neural network. It's then followed by a cascade of `Conv2DTranspose` layers which implement the upsampling (from 7x7 to 28x28) with a `Leaky ReLU` activation for each of them (which, with respect to the standard ReLU activation function, the Leaky ReLU is differentiable also for values lower than 0). However in the output the activation function used is the **tanh** because it takes in input an $x \in R$ and outputs a value $y \in [-1,1]$, therefore being consistent with the values of the images in the dataset.

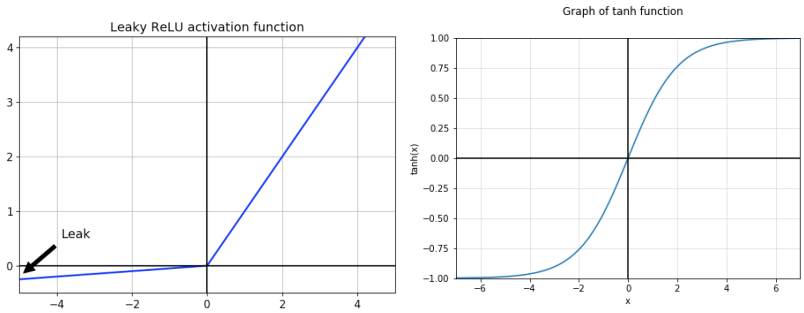


Figure 5: Plots of Leaky ReLU (left) and Tanh (right) activation functions.

The detailed structure of the generator can be found in Fig. (7). After defining the generator, it's possible to see what is the output of its randomly initialized state when given in input a random noise vector of size 100.

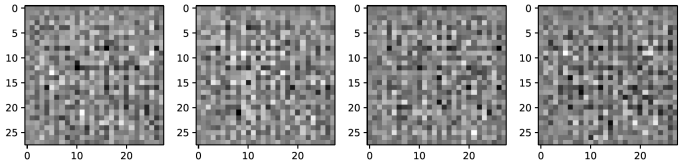


Figure 6: Generator output of 4 random noise vectors input.

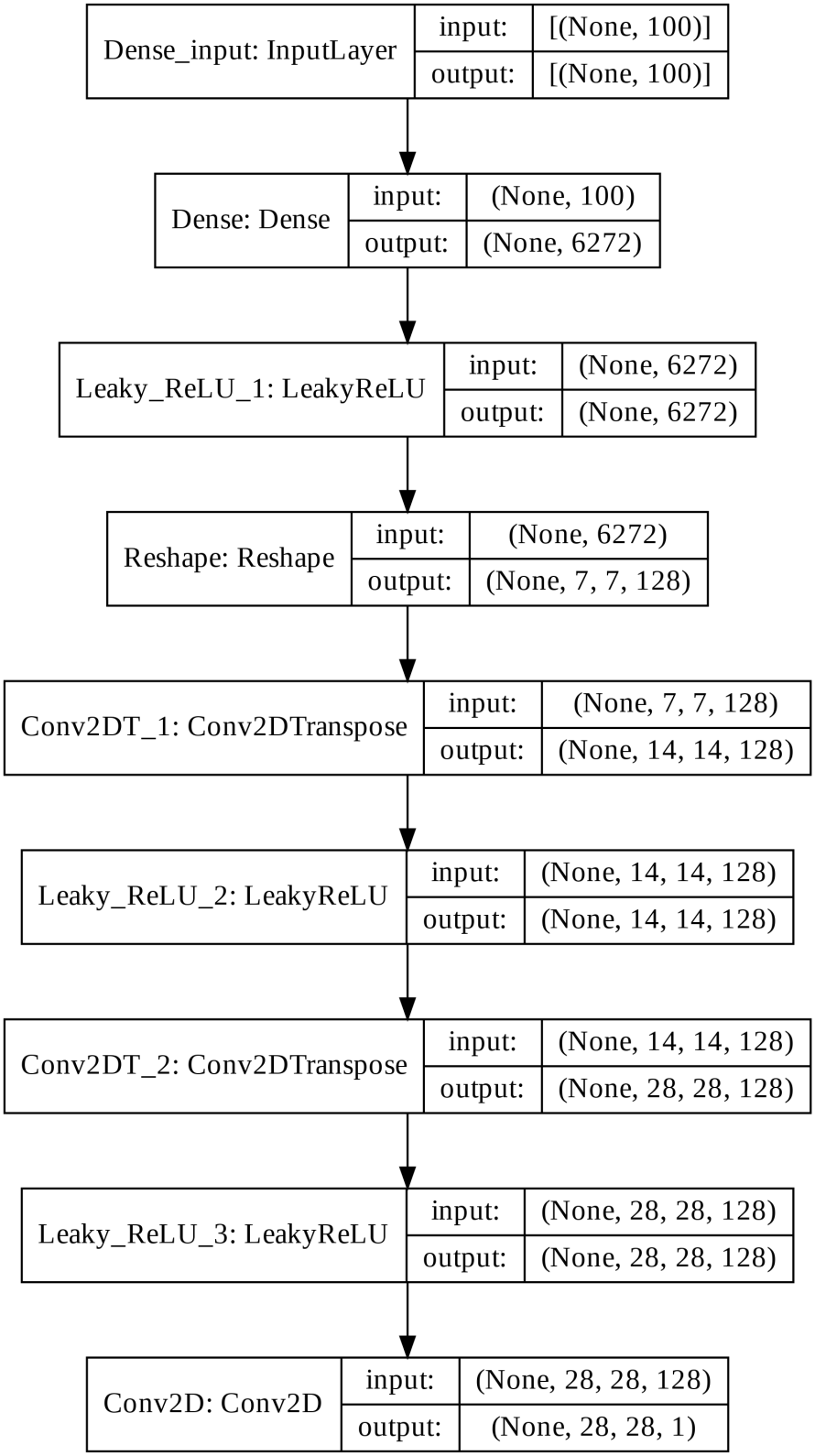


Figure 7: Structure of the Generator.

3.2 Discriminator

The discriminator also uses Convolutional Layers to perform its task, but this time it uses them in order to "deconvolve" the image given in input by progressively downsampling it in order to generate a numerical output: it takes in input an image of size 28x28 (with pixel value $\in [-1,1]$ as the images in the dataset) and returns in output a value between 0 and 1. This output represent the prediction of the discriminator of the image taken in input, where 0 means that the image is fake and 1 means that the image is real. The `Leaky ReLU` activation function is still used except for the last layer which is a `Flatten` layer followed by `Dropout` (a technique used in machine learning in which some weight values are dropped in order to prevent overfitting) and a `Dense` layer which uses a **sigmoid** activation function because it takes in input an $x \in R$ and outputs a value $y \in [0,1]$, therefore being consistent with the output values expected by the discriminator.

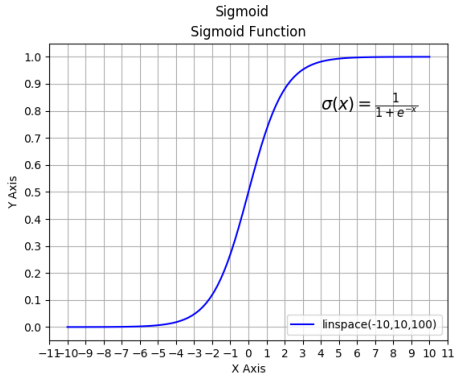


Figure 8: Plot of Sigmoid activation function.

The sigmoid activation function is widely used in Deep Learning applications because of its structure. In the case of the discriminator, as stated in Sec. (1.1), its aim is to maximize a loss function that from its point of view means obtaining $D(G(z)) = 0$. The output range of the discriminator (that is $[0,1]$) is perfectly in line with the output range of the sigmoid activation function, and that's why it has been chosen as the output of the last layer (which comprise only one neuron). The detailed structure of the discriminator can be found in Fig. (9).

3.3 Optimization and Loss

In order to train the model, it's now important to define its loss and its optimizers. The loss function chosen for the models is the **Cross Entropy** because, given two probability distributions $P(x)$ and $Q(x)$, it can be written as:

$$H(P,Q) = H(P) + D_{KL}(P||Q) \tag{3}$$

where $D_{KL}(P||Q)$ is the Kullback-Leibler divergence between the two distributions which is a measure of difference between the two distributions: in this way, minimizing the Cross Entropy is equivalent to minimizing the Kullback-Leibler divergence between the two distributions. After defining this function, the generator loss is given by the Cross Entropy between the generated images and a vector of ones (since the aim is to have them classified as legitimate by the discriminator, thus obtaining $D(G(z)) = 1$), while the discriminator loss is given by the sum of the Cross Entropy between the generated images and a vector of zeros and the Cross Entropy between the training images and a vector of ones (since in this other case the discriminator wants to obtain $D(G(z)) = 0$ and $D(x) = 1$, where x are the real training images).
As stated in Sec. (1.2), the Adam optimizer is used with a learning rate value of 0.0001 and the model is trained for a total of 150 epochs.

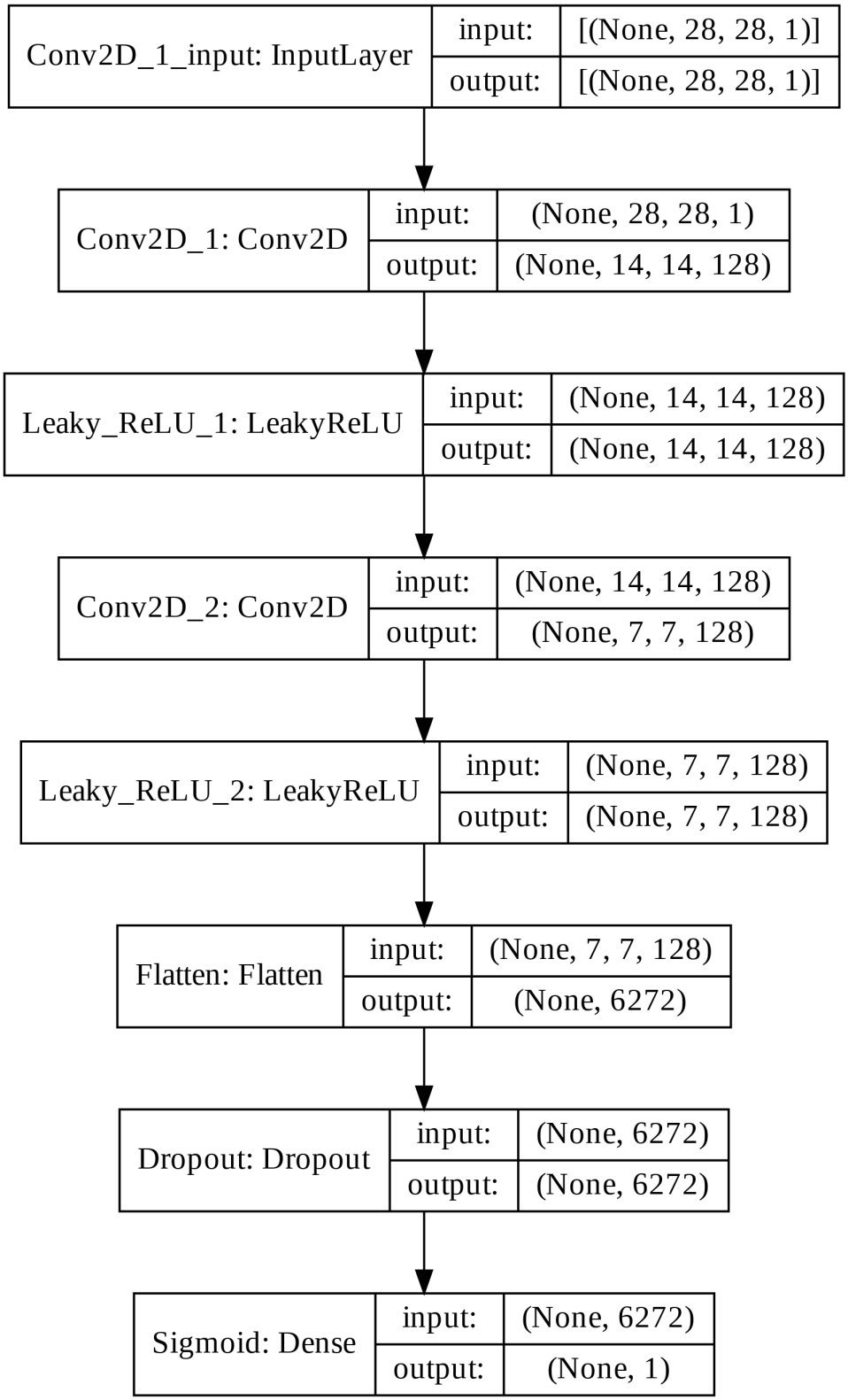
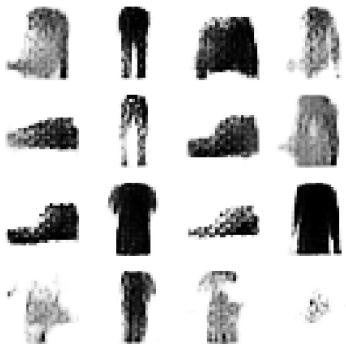


Figure 9: Structure of the Discriminator.

3.4 Results

In Fig. (10) it's possible to see 16 examples of the generator's output for different epoch values. The program has been run in one shot without checkpoints and thus it's possible to see the evolution of the generator's output as learning unfolds.

(a) Generator's output at Epoch 1.



(b) Generator's output at Epoch 50.

(c) Generator's output at Epoch 100.

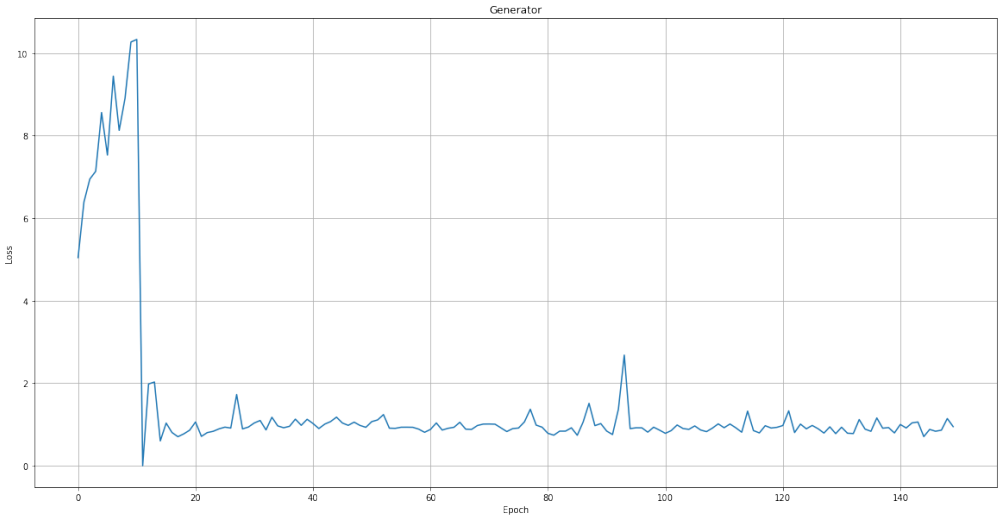
(d) Generator's output at Epoch 150.

Figure 10: Generator's output during training.

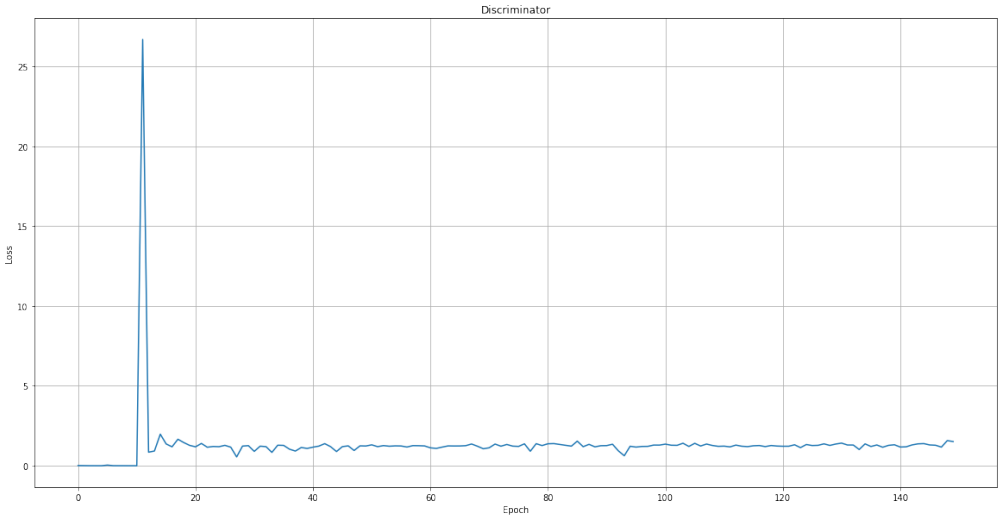
It's possible to notice how around the 50^{th} epoch most of the images are already recognizable, since it's possible to clearly distinguish shoes from shirts and trousers, but it's only after another 50/100 epochs that it's also possible to recognize the subcategories of the dataset, such as ankle boots and sneakers, or tops and shirts. This can be caused by the peculiar behavior of the Convolutional Layers discussed in Sec. (1.3) since, taking inspiration from the visual cortex of the brain, they first learn to extract coarse features (such as the difference between trousers and shirts), and then they gradually learn to extract more accurate features from the previous layers (like the length of the sleeve that differentiate a t-shirt from a shirt).

The losses of both the generator and discriminator are shown in Fig. (11). It's possible to see that, after some initial bounces, the two losses converge to some values, which can mean that the GAN found an optimum where it can't improve anymore. However, it's also possible to notice that the model is a little bit unstable (especially in the first part, since for the first 10 epochs the output of the generator was very similar to what shown in Fig. (10a), where the discriminator easily identify all instances as fake), but after some epochs the model is able to escape the local minimum in which

it was trapped. It's also possible to notice that the losses at epoch 50 are very similar to the losses at the final epoch, while as discussed before the visual difference between the generator's output at the same epoch is quite pronounced: this could mean that it could be possible to train the model longer in order to have it generate more detailed images and that the created model suits the task quite well; however the images in Fig. (10d) are satisfactory enough for a "low quality" dataset with images with size 28x28.



(a) Generator's loss during training.



(b) Discriminator's loss during training.

Figure 11: GAN's loss during training.

4 Generative Adversarial Network on DeepFashion Dataset

After assessing the appropriateness of the models in Sec. (3.1) and Sec. (3.2) on the Fashion MNIST dataset, it's now possible to train the same model (with some slight modifications) to another similar dataset called **DeepFashion Dataset**[2] [6]. The DeepFashion is a dataset that comprises over 800,000 RGB fashion images labeled in 50 categories and every image also has some descriptive attributes.

[2] http://mmlab.ie.cuhk.edu.hk/projects/DeepFashion.html

However, due to the limited capability of the local machine, only a subset of this dataset will be taken, comprising 9425 total images and 15 different classes (which are Blazer, Blouse, Cardigan, Dress, Jacket, Jeans, Jumpsuit, Romper, Shorts, Skirt, Sweater, Sweatpants, Tank, Tee and Top). The total number of images had to be lower than the one of the Fashion MNIST dataset because not only every image has two additional channels (because they're all RGB), but they also have higher quality and different sizes than the one of the previous dataset.

The images of the DeepFashion dataset are then divided into batches of size 128 (instead of 256 because the 8GB of GPU memory on the RTX 2070 were not enough) and they are reshaped to a size of 64x64 pixels. Every channel is then normalized in order to have pixel values in the range $[-1, 1]$, as done in the previous model.

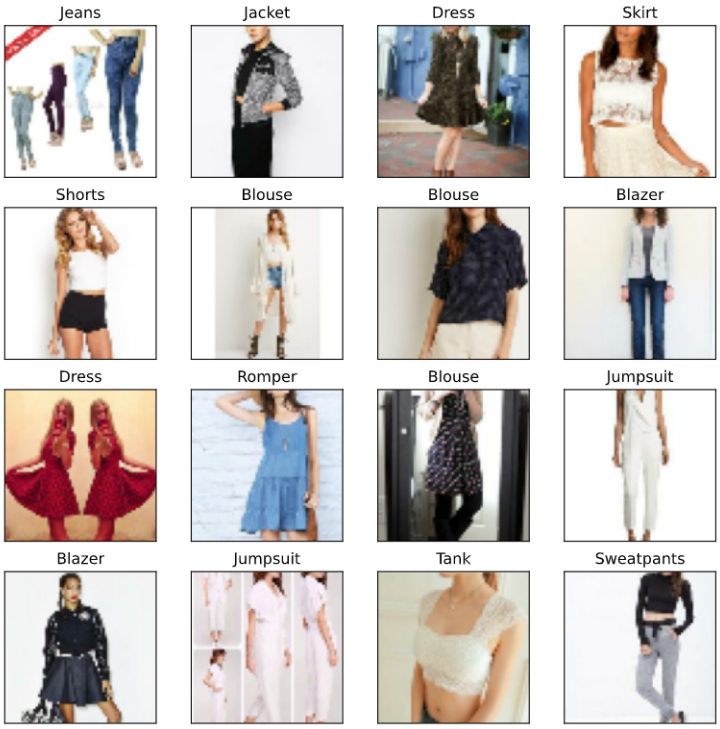
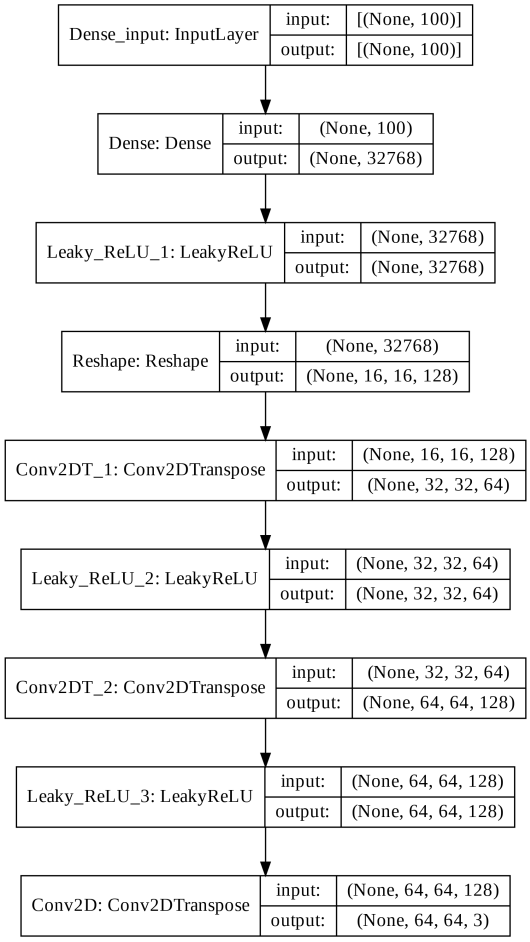


Figure 12: Random examples taken from the DeepFashion Dataset resized to a 64x64 resolution. It's possible to notice how, since the images of the dataset are taken from the internet, some images can contain text, people or multiple instances of an item.

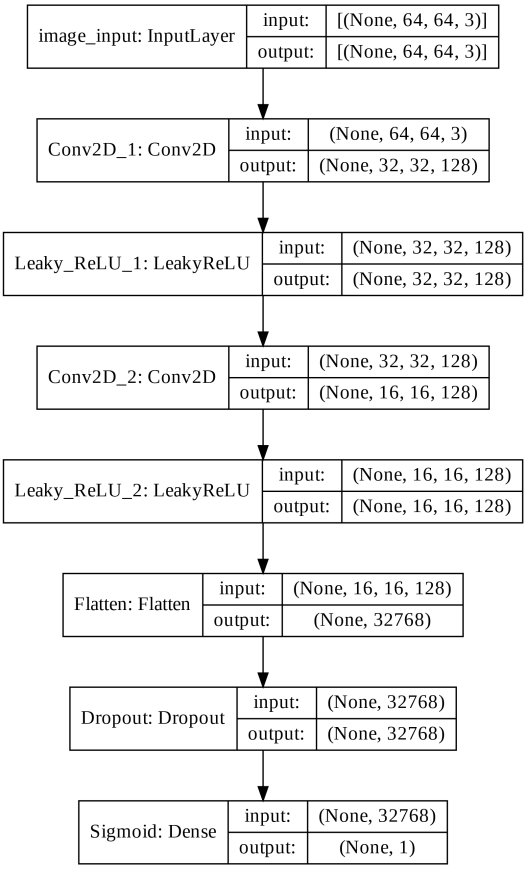
4.1 Generator and Discriminator

As stated before, this first attempt of building a GAN working on a more complex and diverse dataset has the aim to compare the results with the ones obtained with the simpler Fashion MNIST dataset with the same overall architecture. For this reason, the generator and discriminator model have been kept the same except for some minor modifications needed in order to be able to run the program without errors. In the generator model for example, the upscaling is done with two Convolutional Layers from a 16x16 image to a 64x64 image and in the output layer 3 channels have been set in order to have an RGB image, but the noise dimension has been kept to 100 and the size of the kernel of the layers has been kept equal. The same happens in the discriminator, where the input has been modified in order to accept a 64x64 image with 3 channels and the downsampling has been set in order to decrease the size until a 16x16 image through 2 deconvolutional layers, other parameters as activation functions and dropout rate has been kept equal too. The overall model structure can be

seen in Fig. (13).



(a) Structure of the Generator.



(b) Structure of the Discriminator.

Figure 13: Structure of the model.

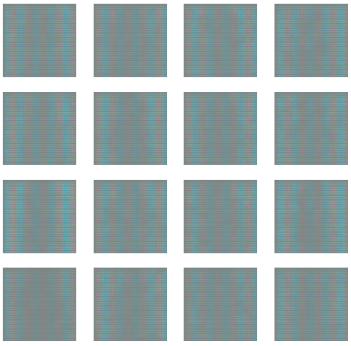
4.2 Optimization and Loss

The same discussion did in Sec. (3.3) holds also in this case. The loss function used is the Cross Entropy and the optimizer used is Adam with a learning rate of 0.0001. However, since the examples to generate are far more complex than the ones in the Fashion MNIST dataset, the number of epochs chosen to train the model is 1500.

4.3 Results

In Fig. (14) it's possible to see 16 examples of the generator's output for different epoch values. The program this time hasn't been run in one shot and checkpoints were used, thus there might be some abrupt jumps in the examples shown as learning unfolds.

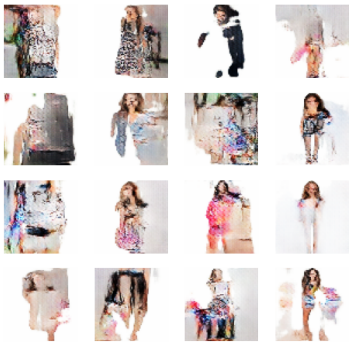
It's immediately noticeable how the situation this time is very different to the one seen in Sec. (4.3). If on the Fashion MNIST Dataset after 50 epochs it was possible to distinguish the objects in the images, the same model working on the DeepFashion Dataset isn't even close to those results even after 10 times the epochs. It's possible to glimpse a human silhouette in most of the examples generated (with the exception of a few ones where there are some random patches of color) and in later epochs is also possible to separate the head from the body, but it's really difficult to pick up any characteristics



(a) Generator's output at Epoch 1.

(b) Generator's output at Epoch 500.

(c) Generator's output at Epoch 1000.



(d) Generator's output at Epoch 1500.

Figure 14: Generator's output during training.

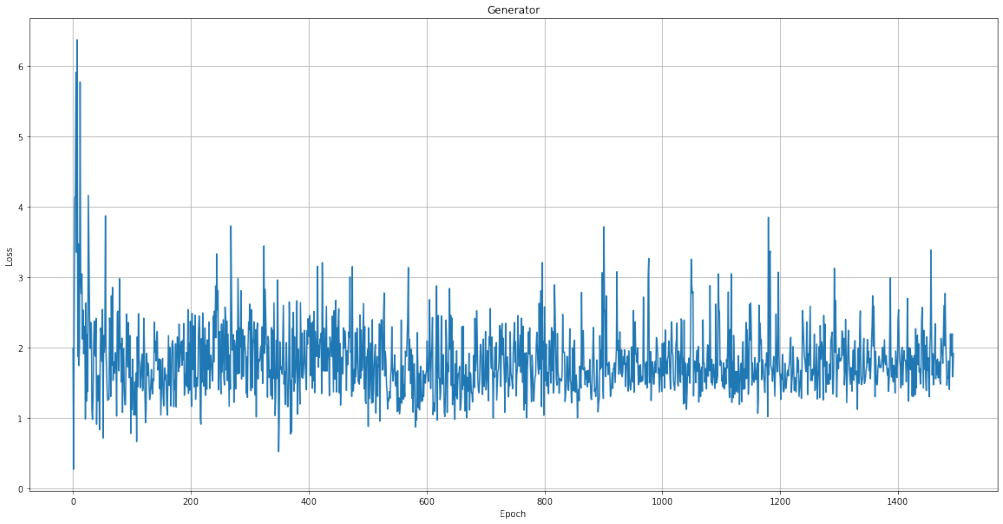
of the clothes (just in some of them it's possible to distinguish the color and the type of clothing).

This behavior of the model can be caused by various factors:

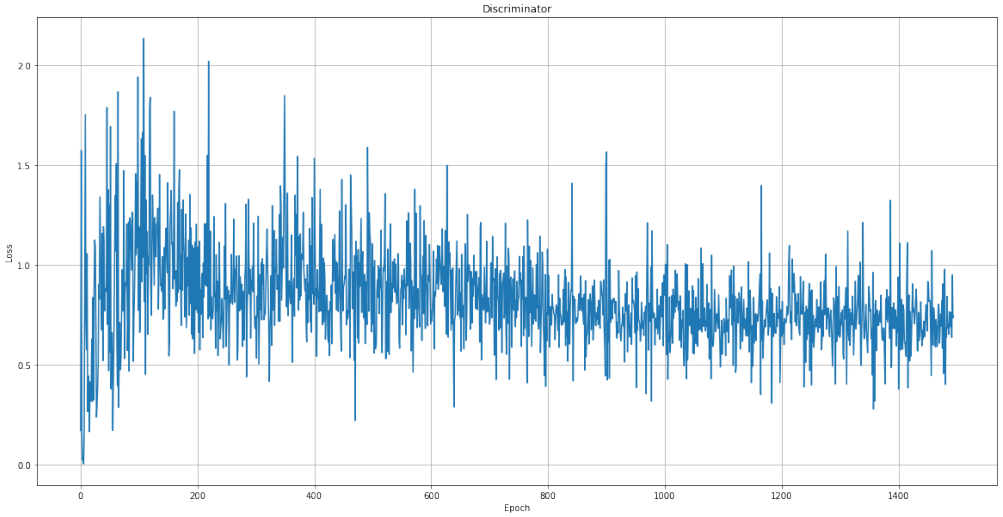
- **Insufficient number of epochs:** since the dataset is far more complex than the previous one, the model could take longer to grasp also the coarsest features of it. After the initial part, it's indeed possible to see that the variation of the images between the epoch 500 and 1500 is not as pronounced as the difference between epoch 50 and 150 in the Fashion MNIST GAN model. Longer running times (in the order of the tens of thousands of epochs) might be needed in order to have satisfactory results.
- **Complexity of the dataset:** as seen in Fig. (12), the DeepFashion dataset is really complex and has a lot of different elements in it. Even after reducing the number of classes and the number of total images processed, it's still possible to have images that differs a lot from one another even inside the same class. The presence of writings, duplicates and mirrored images can cause the GAN to misunderstand the features in the image and focus on some parts of it that are not interesting for the sake of the task given to it.
- **Complexity of the model:** since a lot of parameters changed in the dataset but not so much changed in the model architecture, that might not be deep and large enough to fit the task. Adding layers for the upscaling and downscaling and adding units to the layers might be helpful,

since the images dealt with have more than double the scale and uses more channels for having them RGB.

The losses of both the generator and discriminator are shown in Fig. (15). It's possible to notice than, as opposed to the loss graphs of the model trained on the Fashion MNIST dataset, the loss vary a lot during training and struggle in finding a convergence point. This could mean that even by raising the number of training epochs the model might not be able to generate satisfactory images, and therefore it's the model itself that should be modified in order to fit the more complex task given to it.



(a) Generator's loss during training.



(b) Discriminator's loss during training.

Figure 15: GAN's loss during training.

5 Conditional Generative Adversarial Network on Fashion MNIST Dataset

The aim of this and the following sections will be, as before, the comparison of a similar model trained on the to dataset presented in the previous sections. This time, the model will be a **Conditional Gen-**

erative Adversarial Network (or cGAN for short) [7], a type of GAN that generate images conditioned on a label, allowing for the generation of a particular class of images given a label. Therefore, the new loss function will become:

$$\min_G \max_D \mathop{\mathbb{E}}_{x\sim p_r} \left[\log D(x|y)\right] + \mathop{\mathbb{E}}_{z} \left[\log\left(1 - D(G(z|y))\right)\right] \tag{4}$$

because both the generator and the discriminator are conditioned on y, which represent the label. Thus, also the models will be modified in order to have them take an additional input. The cGAN can have some advantages with respect to a "standard" GAN, since in this way it's possible to control the output of the generator and convergence will be faster since by giving in input the label some patterns in the data are created. This last advantage is particularly interesting for the sake of this report, since the convergence of the loss was the main problem of the model trained on the DeepFashion dataset.

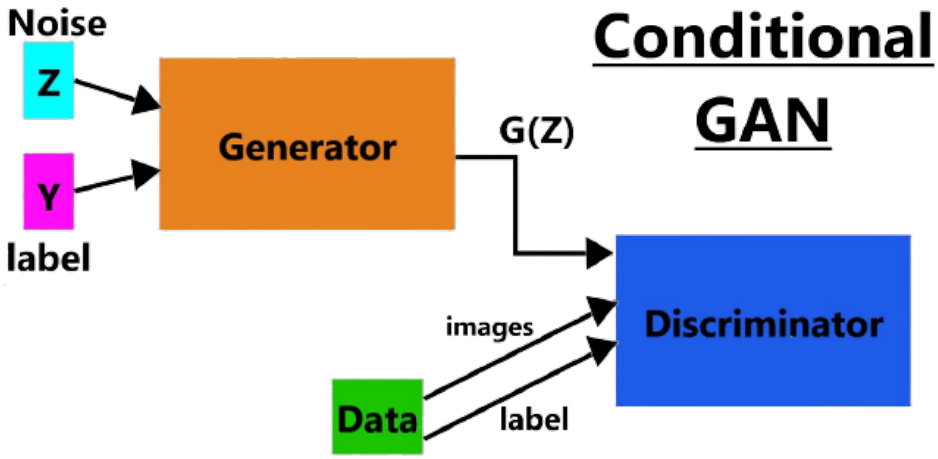


Figure 16: Structure of a Conditional Generative Adversarial Network.

5.1 Generator

As stated before, both the generator and the discriminator now have to take two inputs: the generator will have in input the noise vector and a label. Therefore, a new type of layer is needed, called `Concatenate`, which merges the two inputs. The noise input goes through a `Dense` layers, gets normalized, passes through the `Leaky ReLU` activation function and then gets reshaped into a 7x7 image. Meanwhile, the label input goes through an `Embedding` layer which turns the label (which is an integer between 0 and 9) into a fixed size vector (in this case of size 100), it then gets reshaped into a 7x7 image too. The two 7x7 image (coming respectively from the noise and the label) get now concatenated together into a 7x7 image which then gets upsampled to 28x28 with two Convolutional Layers as shown in Sec. (3.1). The detailed structure of the generator can be found in Fig. (17).

5.2 Discriminator

The discriminator will have in input the images from the dataset and generator's output and a label. The label's input goes through the same passages of the label input of the generator (`Embedding` layer of size 100 and `Dense`) but gets reshaped to a 28x28 image in order to be coherent with the other image input, which doesn't get any particular treatment. Indeed, these two images are then concatenated together with the `Concatenate` layer in a 28x28 image which then gets downsampled to 7x7 with two Convolutional Layers as shown in Sec. (3.2). After that it gets flattened and dropout is applied. The detailed structure of the generator can be found in Fig. (18).

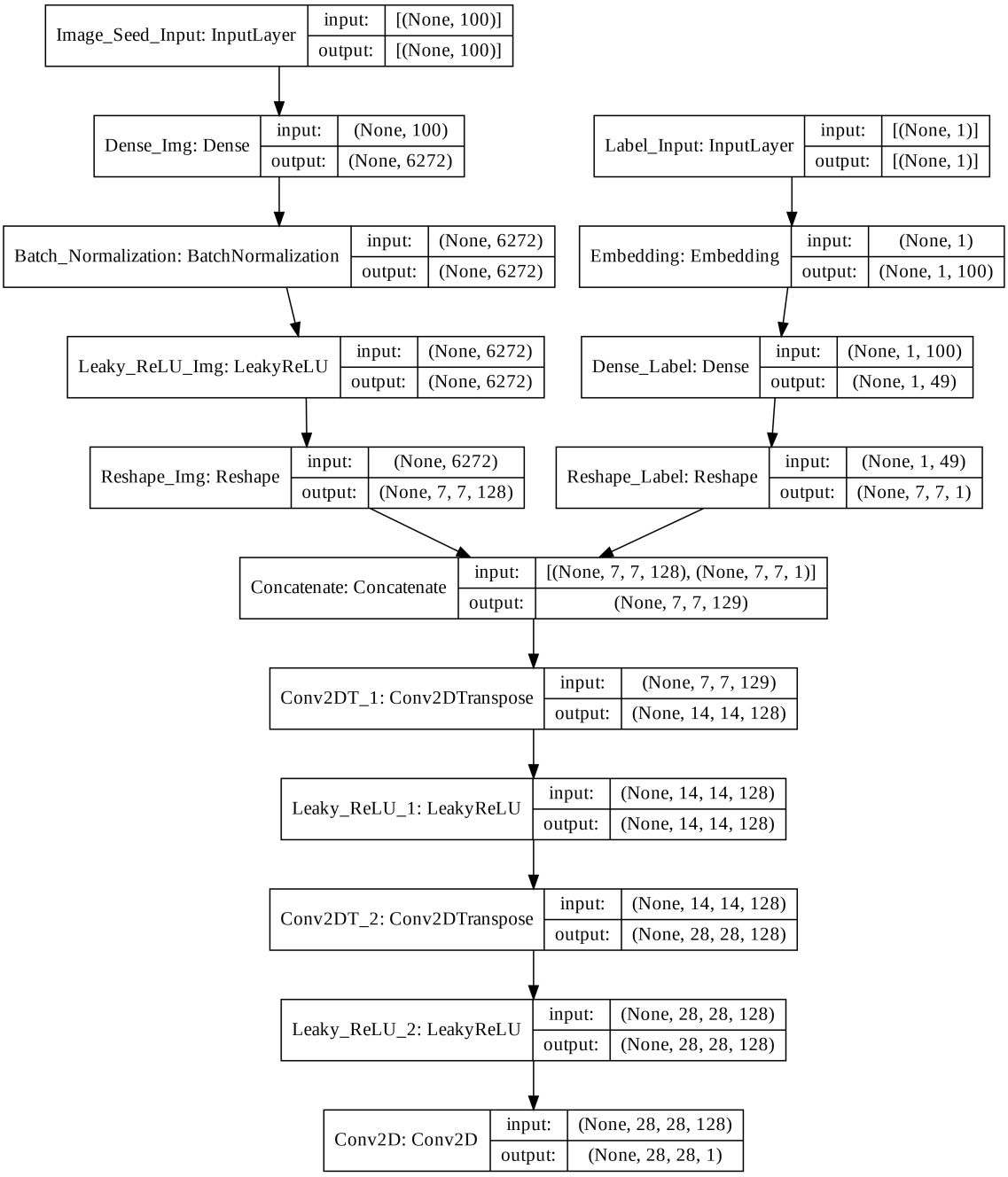


Figure 17: Structure of the Generator.

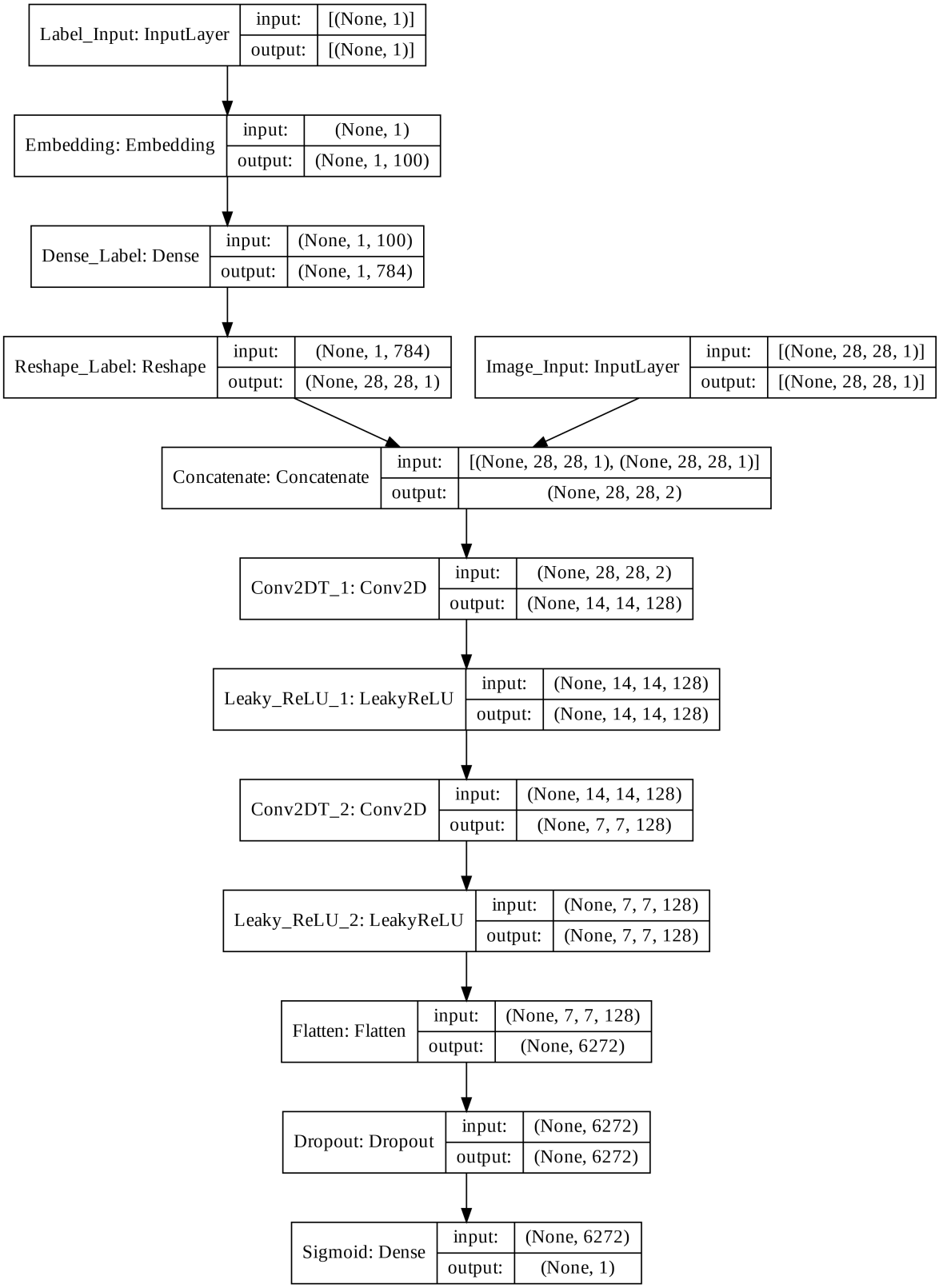


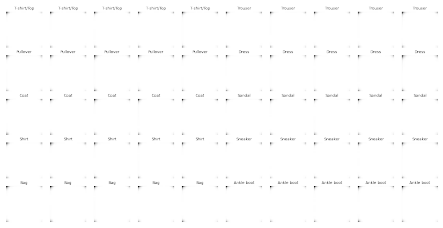
Figure 18: Structure of the Discriminator.

5.3 Optimization and Loss

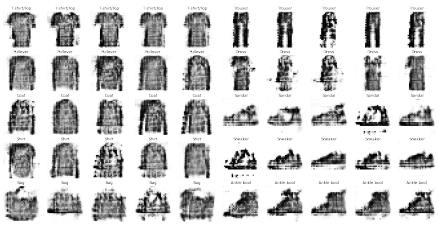
The loss is the same as in the other models, and so is the optimizer. The model gets trained for 150 epochs in order to be comparable to the standard GAN model trained on the Fashion MNIST dataset.

5.4 Results

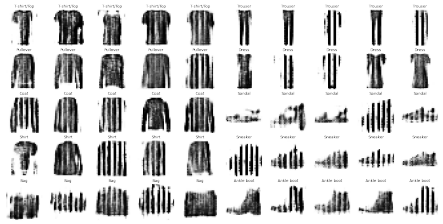
In Fig. (19) it's possible to see 50 examples of the generator's output for different epoch values. The program has been run in one shot without checkpoints and thus it's possible to see the evolution of the generator's output as learning unfolds.



(a) Generator's output at Epoch 1.



(b) Generator's output at Epoch 50.



(c) Generator's output at Epoch 100.

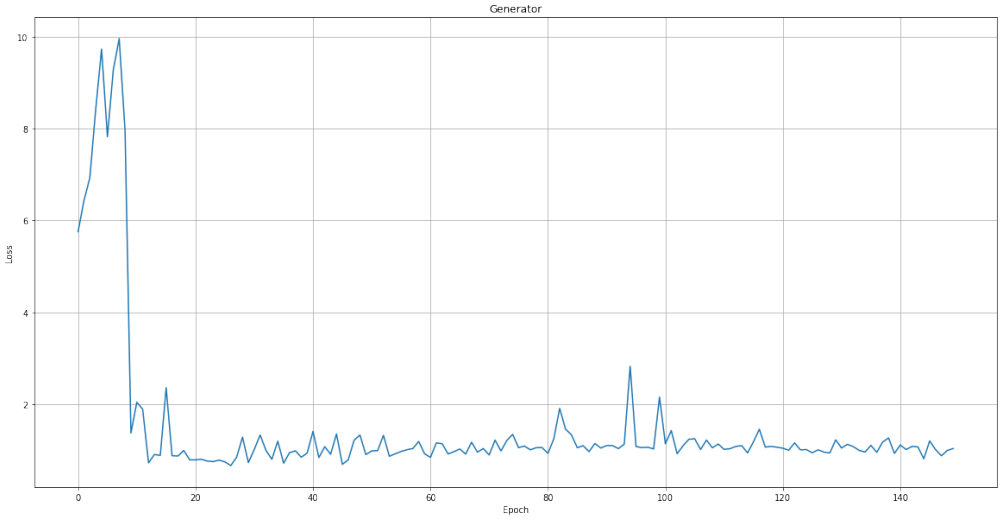
(d) Generator's output at Epoch 150.

Figure 19: Generator's output during training.

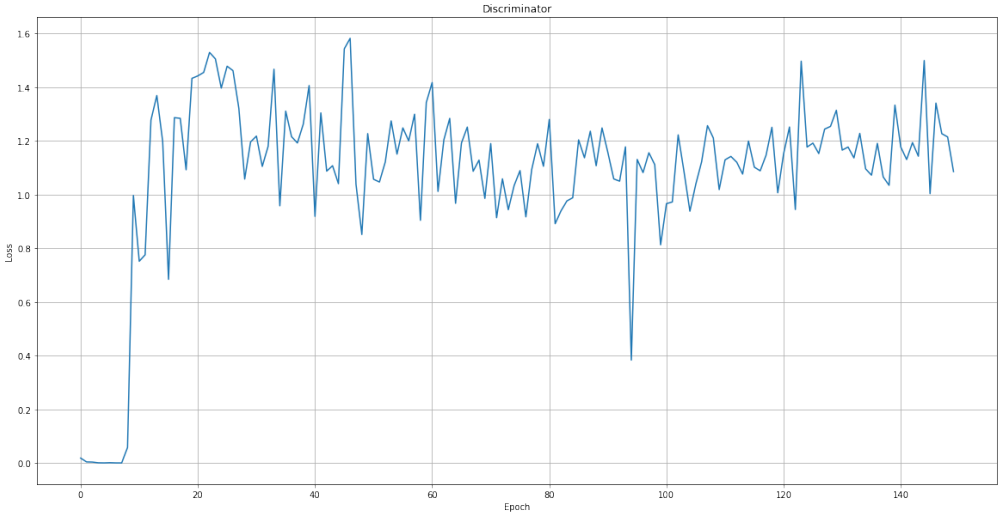
As in the standard GAN model trained with the Fashion MNIST dataset, already around the 50^{th} epoch is possible to distinguish the main categories of data instances. At epoch 150 the output is clear and well defined, pointing that the cGAN works well and very similarly to the standard GAN: this could be caused by the fact that, except for the pre-processing of the input label and images or noises, the architecture for the generator and discriminator was the same. However, at the 100^{th} epoch there were some strange stripes and this issue is also reflected on the losses plots, which can bee seen in Fig. (20). The abrupt rise in generator loss just before the 100^{th} epoch was something that was present also in the loss of the standard GAN, so the cGAN model wasn't able to compensate that. However, the model converged more steadily than the standard GAN and also the initial bounces were less pronounced (discriminator loss could seem not that steady but it's important to notice that the scale of the loss is much smaller than the scale of the loss in the standard GAN plot).

6 Conditional Generative Adversarial Network on DeepFashion Dataset

While working on the DeepFashion dataset, the overall structure of the model has been kept equal to the one of the previous Section, but adopting the same trick used in Sec. (4.1) in order to make inputs and outputs fit together in size terms. The discussion on training and optimization is the same as before and so is the choice and descriptions of the layers, therefore it has been skipped.



(a) Generator's loss during training.



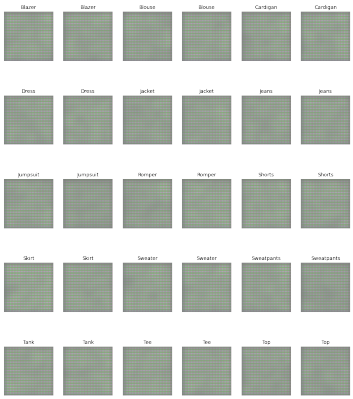
(b) Discriminator's loss during training.

Figure 20: cGAN's loss during training.

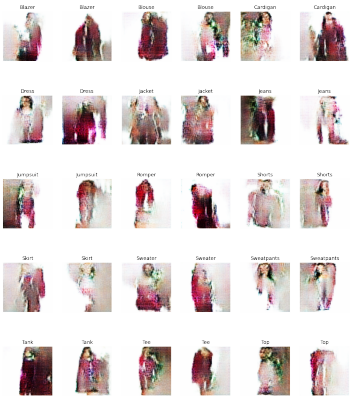
6.1 Results

In Fig. (21) it's possible to see 30 examples of the generator's output for different epoch values. The program this time hasn't been run in one shot and checkpoints were used, thus there might be some abrupt jumps in the examples shown as learning unfolds.

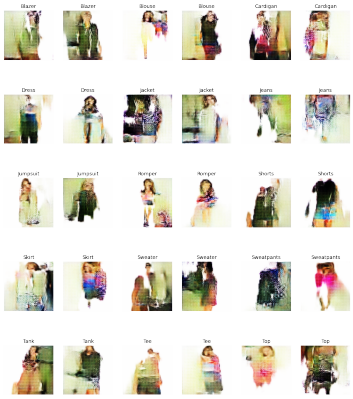
While some of the images are still not clear enough, in some of them it's possible to notice that most of the main features are represented. It's the case of the "Cardigan" labeled generated image here on the right, where even if the person is not clear enough to be distinguished, it's clearly possible to see that its main feature (the open front) is present. The same behavior is present in many other generated examples, however there are still some main problems in training, such as the slow learning over time and also the fact that the unstable conduct of the loss is still present in the cGAN: if the generator



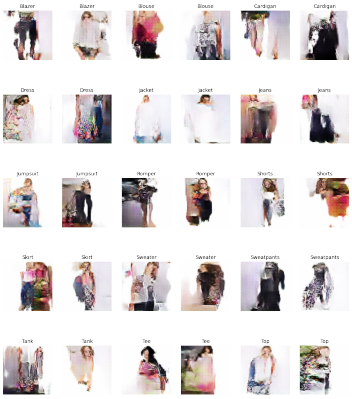
(a) Generator's output at Epoch 1.



(b) Generator's output at Epoch 500.



(c) Generator's output at Epoch 1000.



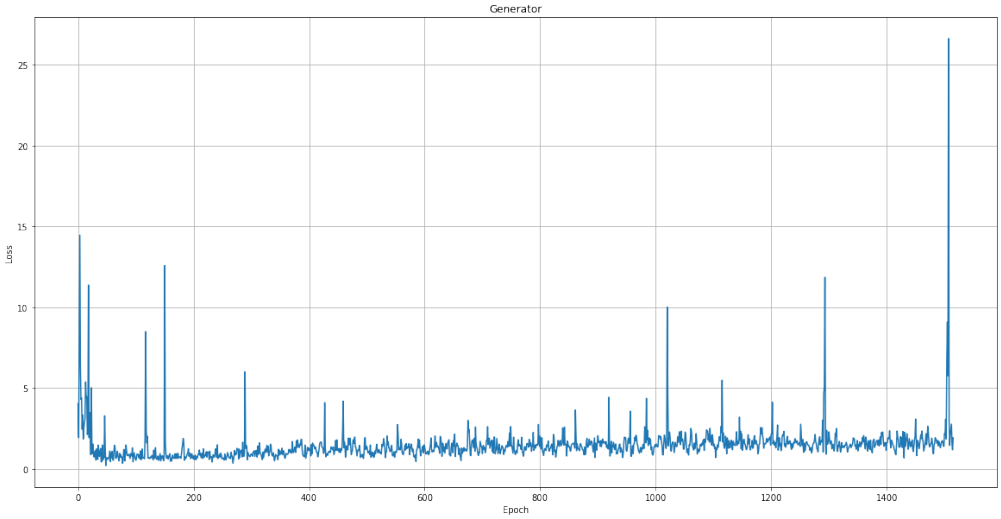
(d) Generator's output at Epoch 1500.

Figure 21: Generator's output during training.

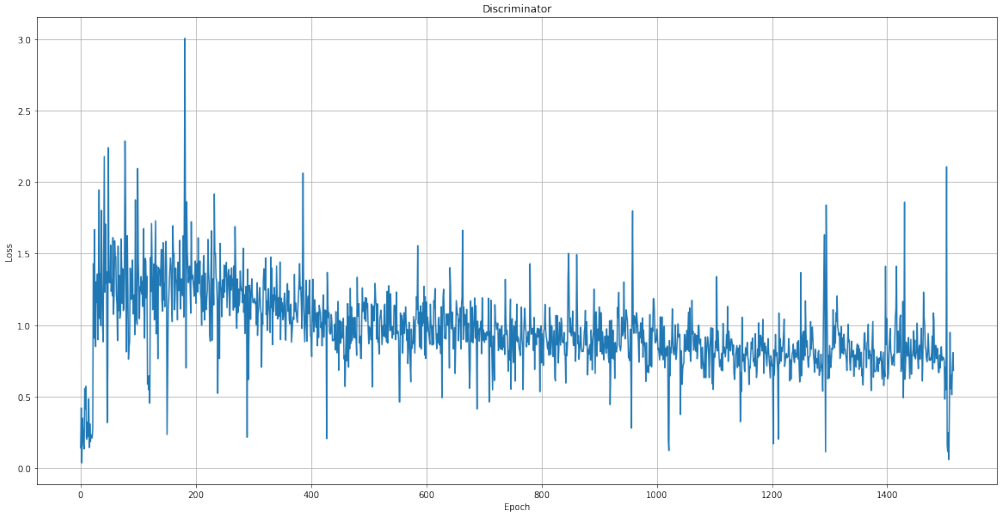
model seemed to be more stable over time, it had some huge jumps in which it could go as high as to 25, phenomenon that happened even a couple of epochs before the final one.

7 Conclusions

After trying the same Generative Adversarial Network on two very different datasets and after building a Conditional version for both of them, it is possible to conclude that the complexity of the model and the complexity of the dataset images play a crucial role in learning. Indeed, the same model that was able to produce really convincing results when trained on the Fashion MNIST dataset, where images are in a grayscale and the size was really small, struggled in generating satisfactory pictures when trained on a broader, more complex (and more realistic) dataset as the DeepFashion one, where images have 3 channels and various, larger sizes. However, if in numerical performance terms the difference was neglectable, it was possible to see a little improvement with the Conditional version



(a) Generator's loss during training.



(b) Discriminator's loss during training.

Figure 22: cGAN's loss during training.

of the networks, where, thanks to the labels given in input to the model, not only it was possible to generate particular classes of images singularly, but also it was possible to see some enhancement of the main features of some classes. Nevertheless, it was also clear how more complex model are needed for such complex operations, in which the training dataset is not kept simple for the sake of learning, but it's more realistic, comprising real images for real applications and therefore needs deeper and broader models in order to elaborate it.

References

[1] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial networks," *arXiv preprint arXiv:1406.2661*, 2014.

[2] S. Sahoo, *An intuitive introduction to generative adversarial networks*, May 2020. [Online]. Available: https://medium.com/analytics-vidhya/an-intuitive-introduction-to-generative-adversarial-networks-2d69b0be2579.

[3] A. G. M. learning researcher, A. Gad, M. l. researcher, and F. m. on, *A comprehensive guide to the backpropagation algorithm in neural networks*, Mar. 2021. [Online]. Available: https://neptune.ai/blog/backpropagation-algorithm-in-neural-networks-guide.

[4] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[5] *Cnn models*. [Online]. Available: https://datawow.io/blogs/cnn-models.

[6] Z. Liu, P. Luo, S. Qiu, X. Wang, and X. Tang, "Deepfashion: Powering robust clothes recognition and retrieval with rich annotations," in *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2016.

[7] M. Mirza and S. Osindero, "Conditional generative adversarial nets," *arXiv preprint arXiv:1411.1784*, 2014.