Deep Learning Project

Andrea Espis
andrea.espis2@studio.unibo.it

Petru Potrimba
petru.potrimba@studio.unibo.it

October 6, 2020

Contents

1 Abstract

Generative Adversarial Networks (GANs) [5] are generative neural network models which aim to produce images that look like real data. We discuss here both a GAN and a conditional GAN we implemented and trained on all the CelebA dataset. We have evaluated the models using the FID score. We also implemented a model which should have been used to evaluate the quality of the generated images, but it has turned out to be not reliable.

2 Techniques

In this section we discuss some important techniques which we applied in our models.

2.1 Pixel Normalization

Pixel Normalization (or better "Pixel-wise feature vector normalization") is a normalization technique [11]. The aim of the technique is to control the magnitude of the activations of the generator. The idea behind Pixel Normalization is a variation of the technique called "Local Response Normalization" described in the paper [2]. The pixel normalization does simply this: normalize the feature vector in each pixel to unit vector after each convolutional layer in the generator. In other words: it normalizes the activation of each node such that the sum of the activations of the nodes coding for different features in the same layer and for the same spatial position (x, y) in the image is equal to 1. The authors of the paper suggest to use this normalization layer only in the generator model.

Formula:

$$out_{x,y} = \frac{in_{x,y}}{\sqrt{\frac{1}{N}\sum_{j=0}^{N-1} in_{x,y}^{i} + \epsilon}}$$

Where "in" is the activation value, "out" is the normalized activation value, N is the number of feature maps for the considered convolutional layer and $\epsilon = 10^{-8}$, used to ensure that the denominator is not zero.

2.2 Mini-Batch Standard Deviation

This technique is used to increase the diversity of the generated images [11]. It can be splitted in these steps:

- First compute the standard deviation for each feature in each spatial location over the minibatch.
- Then average these estimates over all features and spatial locations to arrive at a single value.

- Replicate the value and concatenate it to all the spatial locations and over the minibatch, yielding one additional (constant) feature map.

This layer could be inserted anywhere in the discriminator, but the authors of the paper suggest to insert it towards the end.

2.3 Spectral Normalization

Spectral Normalization [10] is a weight normalization that stabilizes the training of the discriminator of GANs. It controls the Lipschitz constant of the discriminator to limitate the exploding gradient problem and the mode collapse problem.

Specifically, it normalizes the weight for each layer with the spectral norm $\sigma(W)$ such that the Lipschitz constant for each layer as well as the one of the whole network is equal to one. The formulas to achieve this are the following:

$$W_{SN}(W) := \frac{W}{\sigma(W)}$$

$$\sigma(W_{SN}(W)) = 1$$

$$||f||_{Lip} = 1$$

where $W_{SN}(W)$ are the network weights to which spectral normalization is applied, W are the weights for each layer of the network, $\sigma(W)$ is the maximum singular value of matrix W and $||f||_{Lip}$ is the Lipschitz constant for each layer.

The Lipschitz constant for the whole deep network f is calulated as follows:

$$||f||_{Lip} = \prod_{l=1}^{L+1} \sigma(W^l)$$

3 GAN

Generative Adversarial Networks (GANs) [5] are a deep learning architecture for training powerful generator models. A generator model is capable of generating new artificial samples that plausibly could have come from an existing distribution of samples. GANs are comprised of both *generator* and *discriminator* models [4] [7]. The generator is responsible for generating new samples from the domain (real faces in our case), and the discriminator is responsible for classifying whether samples are real or fake. Importantly, the performance of the discriminator model is used to update both the model weights of the discriminator itself and the generator model. This means that the generator never actually sees examples from the domain and is adapted based on how well the discriminator performs.

Before implementing the GAN, we need to process our images following the requirements. The requirements specify to resize the images to 64×64 and to crop them at the following coordinates [45:173,25:153] to have the faces centered.

3.1 Discriminator

The discriminator model takes as input a 64×64 color image and outputs a binary prediction indicating whether the image is real (class = 1) or fake (class = 0).

It is implemented as a convolutional neural network using best practices for GAN design such as using the *LeakyReLU* activation function with a slope of 0.2, using a 2×2 stride to downsample, and the Adam version of stochastic gradient descent with a learning rate of 0.0002 and a momentum of 0.5 [1]. Furthermore, we applied the *Spectral Normalization* for each convolutional layer as well as the *Mini-Batch Standard Deviation*.

The function reported below implements the discriminator model:

```
def discriminator(filters=128, kernel_size=4, strides=(2, 2), in_shape=(64,64,3)):
    model = Sequential()

    # normal
    model.add(SNConv2D(filters=filters, kernel_size=kernel_size,
        strides=strides, padding='same', kernel_initializer="orthogonal",
        spectral_normalization=True,
        input_shape=in_shape))
    model.add(LeakyReLU(alpha=0.2))

    # downsample to 32x32
    model.add(SNConv2D(filters=filters, kernel_size=kernel_size,
        strides=strides, padding='same', kernel_initializer="orthogonal",
        spectral_normalization=True))
    model.add(LeakyReLU(alpha=0.2))

    # downsample to 16x16
    model.add(SNConv2D(filters=filters, kernel_size=kernel_size,
        strides=strides, padding='same', kernel_initializer="orthogonal",
        spectral_normalization=True))
    model.add(LeakyReLU(alpha=0.2))

    # downsample to 8x8
    model.add(SNConv2D(filters=filters, kernel_size=kernel_size,
        strides=strides, padding='same', kernel_initializer="orthogonal",
        spectral_normalization=True))
    model.add(LeakyReLU(alpha=0.2))

    # downsample to 4x4
    model.add(SNConv2D(filters=filters, kernel_size=kernel_size,
        strides=strides,padding='same', kernel_initializer="orthogonal",
        spectral_normalization=True))
    model.add(LeakyReLU(alpha=0.2))
```

```
    # apply mini-batch standard deviation
    model.add(MinibatchStdev())
    model.add(GlobalAveragePooling2D())

    # classifier
    model.add(Flatten())
    model.add(Dropout(0.4))
    model.add(Dense(1, activation='sigmoid'))

    # compile model
    opt = Adam(lr=0.0002, beta_1=0.5)
    model.compile(loss='binary_crossentropy', optimizer=opt,
        metrics=['accuracy'])
    return model
```

3.2 Generator

The generator model takes as input a point in the latent space and outputs a 64×64 color image. This is achieved by using a fully connected layer to interpret the point in the latent space and provide sufficient activations that can be reshaped into many different of a low-resolution version of the output image. This is then upsampled four times, doubling the size and quadrupling the area of the activations each time using transpose convolutional layers.

The model uses best practices such as the *LeakyReLU* activation, a kernel size that is a factor of the stride size, and a hyperbolic tangent (*tanh*) activation function in the output layer. Moreover, we applied the *Spectral Normalization* for each convolutional layer as well as the *Pixel Normalization*.

The function reported below implements the generator model:

```
def generator(latent_dim, filters=128, kernel_size=4, strides=(2, 2)):
    model = Sequential()

    n_nodes = filters * 4 * 4
    model.add(Dense(n_nodes, input_dim=latent_dim))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Reshape((4, 4, filters)))

    # upsample to 4x4
    model.add(SNConv2DTranspose(filters=filters, kernel_size=kernel_size,
        strides=strides, padding='same', kernel_initializer="orthogonal",
        spectral_normalization=True))
    model.add(PixelNormalization())
    model.add(LeakyReLU(alpha=0.2))

    # upsample to 8x8
    model.add(SNConv2DTranspose(filters=filters, kernel_size=kernel_size,
```

```
        strides=strides, padding='same', kernel_initializer="orthogonal",
        spectral_normalization=True))
    model.add(PixelNormalization())
    model.add(LeakyReLU(alpha=0.2))

    # upsample to 16x16
    model.add(SNConv2DTranspose(filters=filters, kernel_size=kernel_size,
        strides=strides, padding='same', kernel_initializer="orthogonal",
        spectral_normalization=True))
    model.add(PixelNormalization())
    model.add(LeakyReLU(alpha=0.2))

    # upsample to 32x32
    model.add(SNConv2DTranspose(filters=filters, kernel_size=kernel_size,
        strides=strides, padding='same', kernel_initializer="orthogonal",
        spectral_normalization=True))
    model.add(PixelNormalization())
    model.add(LeakyReLU(alpha=0.2))

    # output layer 64x64x3
    model.add(SNConv2D(filters=3, kernel_size=kernel_size, activation='tanh',
        padding='same', kernel_initializer="orthogonal",
        spectral_normalization=True))
    return model
```

3.3 Combined model

A GAN model can be defined as the combination of both the generator and the discriminator into one larger model. This larger model will be used to train the model weights in the generator, using the output and error calculated by the discriminator model. The discriminator model is trained separately, and as such, the model weights are marked as not trainable in this larger GAN model to ensure that only the weights of the generator model are updated. This change to the trainability of the discriminator weights only has an effect when training the combined GAN model, not when training the discriminator standalone. This larger GAN model takes as input a point in the latent space, uses the generator model to generate an image. The image is fed as input to the discriminator model which outputs a value to classify the input image as real or fake. The combined model is compiled using the Adam optimizer [7] with a learning rate of 0.0002 and the decay rates of these moving average parameter of 0.5 [12].

The function reported below implements the GAN model:

```
def GAN(g_model, d_model):
    # make weights in the discriminator not trainable
    d_model.trainable = False
```

```
# connect them
model = Sequential()
# add generator
model.add(g_model)
# add the discriminator
model.add(d_model)

# compile model
opt = Adam(lr=0.0002, beta_1=0.5)
model.compile(loss='binary_crossentropy', optimizer=opt)
return model
```

3.4 Images generated

We trained the model for 100 epochs using a training set of 180k images, latent dimension of 128 and batch size of 64. A few of generated images are shown below:

Figure 1: First random batch of images.

Figure 2: Second random batch of images.

The images show a strong variability: different genders, skin color, hair color, presence of beard, hair styles, ages, facial expressions, face orientation.

4 ACGAN

A conditional GAN (CGAN) is an extension of the GAN architecture that adds the possibility of conditioning the generated images by specifying some attributes. In the CGAN architecture, the generator is provided both with a point in the latent space and an attributes vector as input and its aim is to generate an image containing the specified attributes. The discriminator instead is provided with an image and an attributes vector as input and its aim is to predict whether the image is real of fake.

A further extension of the GAN architecture which seems to have a more stable training and better results is the Auxiliary Classifier GAN (ACGAN) [3]. The main difference between CGAN and ACGAN is in the discriminator model. The CGAN's discriminator is provided with the image and an attributes vector as input and the output is just the classification of the input images as real or fake. The ACGAN's discriminator instead is only provided with the image as input and it must predict whether the given image is real or fake and must also predict the attributes vector of the image.

Below we show the differences of the architectures between CGAN and ACGAN:

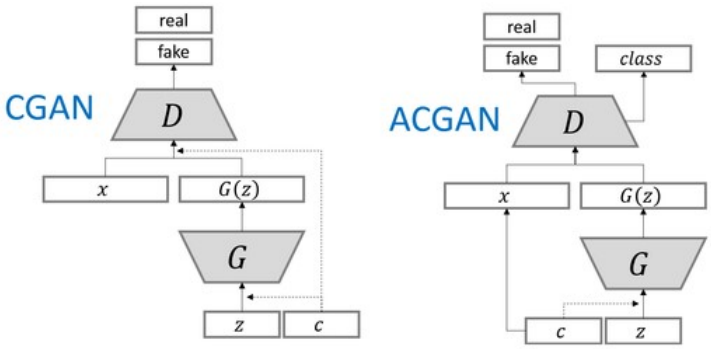


Figure 3: Differences about the architectures of a CGAN and an ACGAN.

4.1 Discriminator

We followed the tips to implement a stable ACGAN [3][1], and additionally we applied a technique to increase the variation of the data captured by the discriminator, which consists in adding the MinibatchStdev layer towards the end of the model [11]. The function reported below implements the discriminator model:

```
def discriminator(n_attr, filters = 128, kernel_size = 4, in_shape=(64,64,3)):

    in_img = Input(shape=(in_shape))

    # 64 x 64 x FILTERS
    disc = SNConv2D(filters=filters, kernel_size=kernel_size,
        strides=(2,2), padding='same', kernel_initializer="orthogonal",
        spectral_normalization=True)(in_img)
    disc = LeakyReLU(alpha=0.2)(disc)

    # 32 x 32 x FILTERS
    disc = SNConv2D(filters=filters, kernel_size=kernel_size,
    strides=(2,2), padding='same', kernel_initializer="orthogonal",
    spectral_normalization=True)(disc)
    disc = LeakyReLU(alpha=0.2)(disc)

    # 16 x 16 x FILTERS
    disc = SNConv2D(filters=filters, kernel_size=kernel_size,
    strides=(2,2), padding='same', kernel_initializer="orthogonal",
    spectral_normalization=True)(disc)
    disc = LeakyReLU(alpha=0.2)(disc)

    # 8 x 8 x FILTERS
    disc = SNConv2D(filters=filters, kernel_size=kernel_size,
    strides=(2,2), padding='same', kernel_initializer="orthogonal",
    spectral_normalization=True)(disc)
    disc = LeakyReLU(alpha=0.2)(disc)

    # apply MinibatchStdev
    disc = MinibatchStdev()(disc)

    # current: 4 x 4 x FILTERS
    disc = GlobalAveragePooling2D()(disc)

    # output about fake/real image:
    out1 = Dense(1, activation='sigmoid')(disc)
    # output about attributes:
    out2 = Dense(n_attr, activation='softmax')(disc)

    # define model
    model = Model(in_img, [out1, out2])
    # compile model
    opt = Adam(lr=0.0002, beta_1=0.5)
    model.compile(loss=["binary_crossentropy", "binary_crossentropy"],
        optimizer=opt, metrics=['accuracy'])
    return model
```

4.2 Generator

The generator takes as input both a point in the latent space and the attributes vector. There are two main approaches: one is to concatenate the two inputs before feeding them to the generator, and another approach is to feed the generator with the two inputs separately and use an embedding layer to concatenate them. For our model the second approach was really bad since the generator was not able to do conditioning with more than 3 attributes. Instead the first approach allowed us to implement a generator able to generate images conditioned by most of the 40 attributes. The generator architecture has been chosen by following the tips for having a stable training [1]. We also followed the fashion of the architecture described in the paper introducing the ACGAN [3], namely the use of a small number of filters going deeper into the model, and the use of a small kernel (4x4 in our model). Additionally we used the PixelNormalization [11]. The function reported below implements the generator model:

```
def generator(latent_dim, n_attr, filters=128, kernel_size=4, strides=(2, 2)):

    in_gen = Input(shape=(latent_dim + n_attr,))
    gen = Dense(4 * 4 * filters)(in_gen)
    gen = LeakyReLU(alpha=0.2)(gen)
    gen = Reshape((4, 4, filters))(gen)

    # 4x4 -> 8x8
    gen = SNConv2DTranspose(filters=filters, kernel_size=kernel_size,
        strides=strides, padding='same', kernel_initializer="orthogonal",
        spectral_normalization=True)(gen)
    gen = PixelNormalization()(gen)
    gen = LeakyReLU(alpha=0.2)(gen)

    # 8x8 -> 16x16
    gen = SNConv2DTranspose(filters=filters, kernel_size=kernel_size,
        strides=strides, padding='same', kernel_initializer="orthogonal",
        spectral_normalization=True)(gen)
    gen = PixelNormalization()(gen)
    gen = LeakyReLU(alpha=0.2)(gen)

    # 16x16 -> 32x32
    gen = SNConv2DTranspose(filters=filters, kernel_size=kernel_size,
        strides=strides, padding='same', kernel_initializer="orthogonal",
        spectral_normalization=False)(gen)
    gen = PixelNormalization()(gen)
    gen = LeakyReLU(alpha=0.2)(gen)

    # 32x32 -> 64x64
    gen = SNConv2DTranspose(filters=filters, kernel_size=kernel_size,
        strides=strides, padding='same', kernel_initializer="orthogonal",
```

```
        spectral_normalization=True)(gen)
    gen = PixelNormalization()(gen)
    gen = LeakyReLU(alpha=0.2)(gen)

    # 64 x 64 x FILTERS -> 64 x 64 x 3
    image = SNConv2D(filters=3, kernel_size=kernel_size, activation='tanh',
        padding='same', kernel_initializer="orthogonal",
        spectral_normalization=True)(gen)

    # define model
    model = Model(in_gen, image)
    return model
```

4.3 Combined model

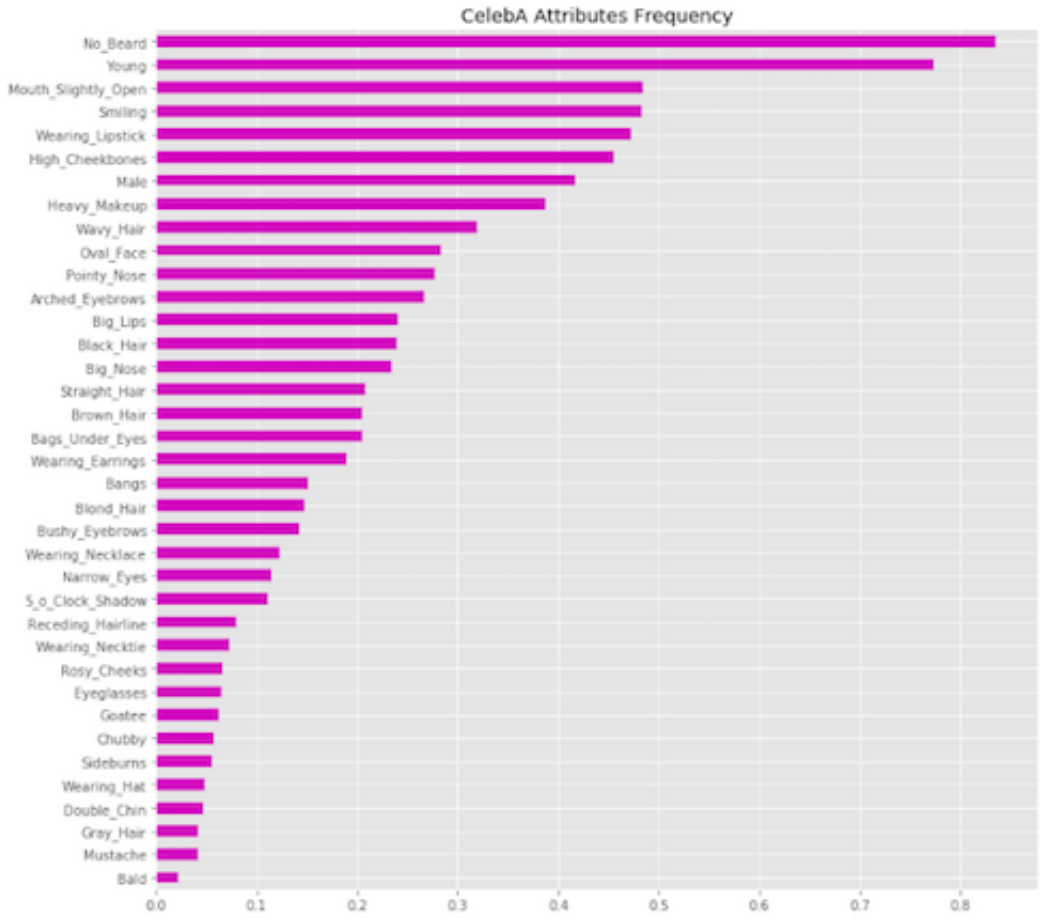
The ACGAN model is obtained combining the generator and the discriminator into one larger model and trained analogously to the GAN model described in the subsection 3.3. The function reported below implements the ACGAN model:

```
def ACGAN(g_model, d_model):
    img = g_model.output
    d_model.trainable = False
    valid, target_label = d_model(img)

    noise_plus_label = g_model.input
    model = Model(noise_plus_label, [valid, target_label])
    opt = Adam(lr=0.0002, beta_1=0.5)
    model.compile(loss=["binary_crossentropy", "binary_crossentropy"],
        optimizer=opt)
    return model
```

4.4 Images generated

The generator we implemented is able to generate conditional images on many of the 40 attributes, but not the most difficult ones. Indeed some of them are more difficult to be used since they have a low frequency in the CelebA dataset. The frequency of the attributes is shown in the following histogram[6]:



Among those which have a low occurrence within the dataset, some are not effective to conditionate the generation, such as "mustache" and "goatee", some others generate bad quality images, such as "gray_hair","eyeglasses":

Figure 4: male, gray_hair, straight_hair.

Figure 5: male, black_hair, straight_hair, eyeglasses, no_beard.

Figure 6: male, black_hair, straight_hair, attractive, young, high_cheekbones, smiling.

Below we show some generated images for different combinations of attributes:

Figure 7: big_lips, blond_hair, heavy_makeup, rosy_cheeks, smiling, wearing_lipstick, young, no_beard, straight_hair, chubby, oval_face, double_chin.

Figure 8: attractive, big_lips, blond_hair, heavy_makeup, rosy_cheeks, smiling, wearing_lipstick, young, no_beard, straight_hair.

Figure 9: attractive,big_lips, blond_hair, heavy_makeup, rosy_cheeks, smiling, wearing_lipstick, young, no_beard, straight_hair, bangs.

Figure 10: attractive, big_lips, blond_hair, heavy_makeup, rosy_cheeks, smiling, wearing_lipstick, young, no_beard, straight_hair, mouth_slightly_open.

5 Metrics

5.1 FID score

The Fréchet Inception Distance [8], or FID for short, is a metric for evaluating the quality of generated images and specifically developed to evaluate the performance of generative adversarial networks.

The FID score uses the Inception v3 model. Specifically, the coding layer of the model (the last pooling layer prior to the output classification of images) is used to capture computer-vision-specific features of an input image. These activations are calculated for a collection of real and generated images. The activations are summarized as a multivariate Gaussian by calculating the mean and covariance of the images. These statistics are then calculated for the activations across the collection of real and generated images. The distance between these two distributions is then calculated using the Fréchet distance, also called the Wasserstein-2 distance.

A lower FID indicates better-quality images; conversely, a higher score indicates a lower-quality image.

5.1.1 How to Calculate the FID

A 2,048 feature vector is then predicted for a collection of real images from the problem domain to provide a reference for how real images are represented. Feature vectors can then be calculated for synthetic images. The result will be two collections of 2,048 feature vectors for real and generated images.

The FID score is then calculated using the following equation taken from the paper:

$$d^2 = ||mu_1 - mu_2||^2 + Tr(C_1 + C_2 - 2 \times \sqrt{C_1 \times C_2})$$

The score is referred to as d^2, showing that it is a distance and has squared units. The mu_1 and mu_2 refer to the feature-wise mean of the real and generated images, e.g. 2.048 element vectors where each element is the mean feature observed across the images. The C_1 and C_2 are the covariance matrix for the real and generated feature vectors, often referred to as sigma. The $||mu_1 - mu_2||^2$ refers to the sum squared difference between the two mean vectors. Tr refers to the trace linear algebra operation.

5.1.2 Results

To calculate the FID score for our problem, we took apart a dataset composed of 20k images (not used during the training) and then we randomly picked 10k images from this dataset to compute the FID.

- The FID score we obtained on the GAN model is: 11.21.
- To compute the FID for the conditional GAN model, we generated images using as attribute vectors the groundtruth labels randomly picked from

the CelebA attributes. The FID score we obtained on the ACGAN model is: 19.42.

5.2 Our metric

We implemented a model which aim is to evaluate the quality of the generated images. However it has turned out to not be a reliable quality index, so we report it here as a negative result.

The model we implemented is a CNN which takes as input an image and should output a value close to 0 if the quality of the image is high, and close to 1 if the image has a bad quality. This is the architecture we used:

```
def define_CNN(in_shape=(64,64,3)):

    # input layer:
    in_image = Input(shape=in_shape)

    # kernel initializer:
    init = RandomNormal(stddev=0.02)

    # downsample to 32x32
    fe = SNConv2D(128, kernel_size=4, strides=(4,4), padding='same',
         kernel_initializer=init, spectral_normalization=True)(in_image)
    fe = LeakyReLU(alpha=0.2)(fe)
    fe = Dropout(0.5)(fe)

    # downsample to 16x16
    fe = SNConv2D(128, kernel_size=4, strides=(4,4), padding='same',
        kernel_initializer=init, spectral_normalization=True)(fe)
    fe = LeakyReLU(alpha=0.2)(fe)
    fe = Dropout(0.5)(fe)

    # downsample to 8x8
    fe = SNConv2D(128, kernel_size=4, strides=(4,4), padding='same',
        kernel_initializer=init, spectral_normalization=True)(fe)
    fe = LeakyReLU(alpha=0.2)(fe)
    fe = Dropout(0.5)(fe)

    # downsample to 4x4
    fe = SNConv2D(128, kernel_size=4, strides=(2,2), padding='same',
        kernel_initializer=init, spectral_normalization=True)(fe)
    fe = LeakyReLU(alpha=0.2)(fe)
    fe = Dropout(0.5)(fe)

    # downsample 2x2
    fe = SNConv2D(128, kernel_size=4, strides=(2,2), padding='same',
```

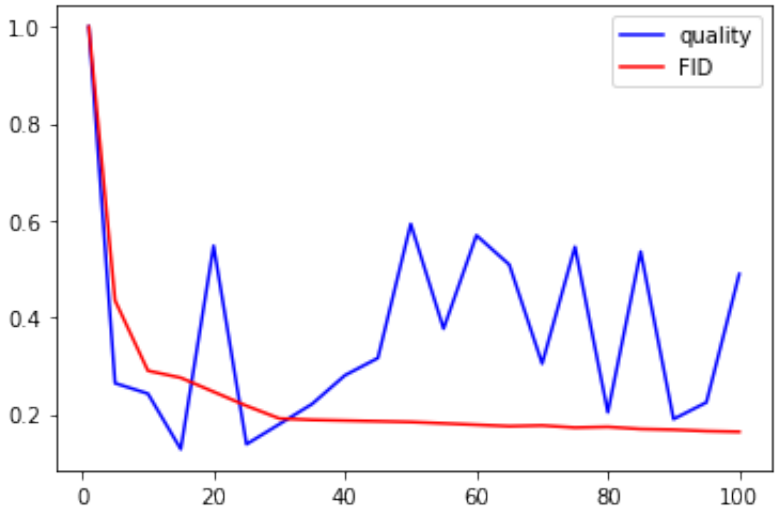
```
    kernel_initializer=init, spectral_normalization=True)(fe)
fe = LeakyReLU(alpha=0.2)(fe)
fe = Dropout(0.5)(fe)

# flatten feature maps
fe = Flatten()(fe)

# output about fake/real image:
out = Dense(1, activation='sigmoid')(fe)
model = Model(inputs=in_image, outputs=out)
return model
```

The model has been trained using two sets of images: the CelebA dataset images to which we associated the label 0, and the CelebA dataset images after a transformation called "elastic deformation" [9], to which we associated the label 1. The model has been trained using a training dataset of 20k images, reaching a test accuracy of 97% at the 100th epoch. The elastic deformation has been applied with the parameters alpha = 5 and sigma = 0.9. Here we show an example of the application of the elastic deformation with these parameters:

The plot below shows, in blue, the values of the quality index given by the implemented model and the FID score, in red, computed on the images generated by the unconditional GAN model at the epochs 1, 5, 10, ..., 100.



As we can see the quality value given by our model is not reliable. A solution could be to try the same idea with different kind of deformations.

6 Conclusions

In this project we implemented a Generative Adversarial Network (GAN) to generate synthetic faces and an Auxiliary Classifier GAN (ACGAN) to condition the generation of the images with specific attributes. Both models have been trained using the CelebA dataset.

GANs are difficult to train due to their intrinsic instability. Thus we applied the Spectral Normalization to reduce this problem. This technique lead to a more stable training and, at the end, a better generation of the images. To improve the quality of the generated images we used the Pixel Normalization technique applied to the generator, whereas to increase the diversity of the generated images we applied the Mini-Batch Standard Deviation technique.

The main challenge is to construct a model that is able to generate images conditioning with all the 40 CelebA attributes. We first attempted to build an ACGAN using the Embedding layer. However, this turned out to be not a good solution, probably because the Embedding layer suffer of the curse of dimensionality. In particular, by using it, our model was not able to condition the generation with more than 3 attributes. Then we decided to replace the Embedding layer with a Dense one and we concatenated the noise to the labels before feeding them to the ACGAN. This turned out to be a good solution as, at the end, the model was able to successfully generate good quality images conditioned on many of the 40 attributes.

The code we developed can be found at the following link: `https://github.com/pptr3/fakefaces-gan`

References

[1] Soumith Chintala Alec Radford, Luke Metz. *Unsupervised representation learning with deep convolutional generative adversarial networks*. 2015.

[2] Geoffrey E. Hinton Alex Krizhevsky, Ilya Sutskever. *ImageNet Classification with Deep Convolutional Neural Networks*. 2018.

[3] Jonathon Shlens Augustus Odena, Christopher Olah. *Conditional Image Synthesis with Auxiliary Classifier GANs*. 2017.

[4] Aaron Courville Ian Goodfellow, Yoshua Bengio. *Deep Learning*. MIT Press, 2016.

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

[6] SilvioBarra Fabio Narducci Michele Nappi Luca Anzalone, Paola Barra. *Transfer Learning for Facial Attributes Prediction and Clustering*. 2019.

[7] Eyal Klang Michal Amitai Jacob Goldberger Maayan Frid-Adar, Idit Diamant and Hayit Greenspan. *GAN-based Synthetic Medical Image Augmentation for increased CNN Performance in Liver Lesion Classification*. IEEE, 2018.

[8] Thomas Unterthiner Bernhard Nessler Sepp Hochreiter Martin Heusel, Hubert Ramsauer. *GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium*. 2018.

[9] John C. Platt Patrice Y. Simard, Dave Steinkraus. *Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis*. 2003.

[10] Masanori Koyama Yuichi Yoshida Takeru Miyato, Toshiki Kataoka. *Spectral Normalization For Generative Adversarial Networks*. 2018.

[11] Samuli Laine Jaakko Lehtinen Tero Karras, Timo Aila. *Progressive growing GANs for improved quality, stability, and variation*. 2018.

[12] Jiakai Wei. *Forget the Learning Rate, Decay Loss*. 2019.