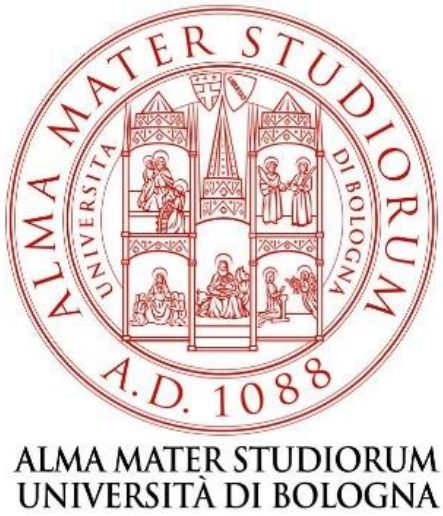
Deep Learning Semester Project Report

Project: Conditional Generative Adversarial Network



Submitted by:

Muhammad Salman Razzaq s0000953223

Date: 30rd May 2020

Submitted to: Professor Dr. Andrea Asperti

Department of Computer Science and Engineering - DISI

Alma Mater Studiorum – Universita Di Bologna

Contents

Abstract

The field of deep learning is evolving very rapidly and for every Artificial Intelligence engineer, it is imperative to learn the depth of this field. Generative adversarial networks are one of the most researched branches of deep learning. This project deals with the implementation of conditional generative adversarial networks on CelebA dataset. The main inspiration of the models and project comes from “Generative Adversarial Networks with Python: Deep Learning Generative Models for Image Synthesis and Image Translation” by Jason Brownlee. Along with development of CDCGAN models on various datasets, this project also experiments on the various applications of GAN like vector arithmetic and vector interpolation on the latent space.

Theoretical Background

Introduction to Generative Adversarial Networks

Generative Adversarial Networks, or GANs for short, are an approach to generative modeling using deep learning methods, such as convolutional neural networks. Generative modeling in an unsupervised learning task in machine learning that involves automatically discovering and learning the regularities or patterns in input data in such a way that the model can be used to generate or output new examples that plausibly could have been drawn from the original dataset. GANs are a clever way of training a generative model by framing the problem as a supervised learning problem with two sub-models: the generator model that we train to generate new examples, and the discriminator model that tries to classify examples as either real (from the domain) or fake (generated). The two models are trained together in an adversarial zero-sum game until the discriminator model is fooled about half the time, meaning the generator model is generating plausible examples.

Generative Adversarial Networks, or GANs, are a deep-learning-based generative model. More generally, GANs are a model architecture for training a generative model, and it is most common to use deep learning models in this architecture. The GAN architecture was first described in the 2014 paper by Ian Goodfellow, et al. titled Generative Adversarial Networks. The initial models worked but where unstable and difficult to train. A standardized approach called Deep Convolutional Generative Adversarial Networks, or DCGAN, that led to more stable models was later formalized by Alec Radford, et al. in the 2015 paper titled Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks.

The GAN model architecture involves two sub-models: a generator model for generating new examples and a discriminator model for classifying whether generated examples are real (from the domain) or fake (generated by the generator model).

- Generator Model that is used to generate new plausible examples from the problem domain.
- Discriminator Model that is used to classify examples as real (from the domain) or fake (generated).

Generative adversarial networks are based on a game theoretic scenario in which the generator network must compete against an adversary. The generator network directly produces samples. Its adversary, the discriminator network, attempts to distinguish between samples drawn from the training data and samples drawn from the generator.

Conditional GANs

An important extension to the GAN is in their use for conditionally generating an output. The generative model can be trained to generate new examples from the input domain, where the input, the random vector from the latent space, is provided with (conditioned by) some additional input. The additional input could be a class value, such as male or female in the generation of photographs of people, or a digit, in the case of generating images of handwritten digits.

The discriminator is also conditioned, meaning that it is provided both with an input image that is either real or fake and the additional input. In the case of a classification label type conditional input, the discriminator would then expect that the input would be of that class, in turn teaching the generator to generate examples of that class in order to fool the discriminator. In this way, a conditional GAN can be used to generate examples from a domain of a given type.

Taken one step further, the GAN models can be conditioned on an example from the domain, such as an image. This allows for applications of GANs such as text-to-image translation, or image-to-image translation. This allows for some of the more impressive applications of GANs, such as style transfer, photo colorization, transforming photos from summer to winter or day to night, and so on. In the case of conditional GANs for image-to-image translation, such as transforming day to night, the discriminator is provided examples of real and generated nighttime photos as well as (conditioned on) real daytime photos as input. The generator is provided with a random vector from the latent space as well as (conditioned on) real daytime photos as input.

Project Implementation:

The practical implementation of deep learning generative adversarial networks in done through conditional generation of CelebA dataset with user specified attributes using Deep Convolutional Generative Adversarial Networks. The project is divided into following sections.

1: Getting Large-Scale Celeb Faces Dataset (CelebA) and Setting up the Environment
2: Preprocessing of the CelebA Faces Dataset
3: Develop a Conditional DCGAN model for CelebA
4: Generation of Images using the trained model
5: Fréchet Inception Distance (FID) score
6: Vector Arithmetic in Latent Space
7: Explore the Latent Space for Vector Interpolation of Faces
8: Proposed Metric for conditionality assessment

1. Getting Large-Scale Celeb Faces Dataset (CelebA) and Setting up the Environment

The first step is to select a dataset of faces and setting up the environment. In this project, we will use the Large-scale CelebFaces Attributes Dataset, referred to as CelebA. This dataset was developed and published by Ziwei Liu, et al. for their 2015 paper titled From Facial Parts Responses to Face Detection: A Deep Learning Approach. The dataset provides about 200,000 photographs of celebrity faces along with annotations for what appears in given photos, such as glasses, face shape, hats, hair type, etc. As part of the dataset, the authors provide a version of each photo centered on the face and cropped to the portrait with varying sizes around 150 pixels wide and 200 pixels tall. We will use this as the basis for developing our GAN model. The dataset can be easily
downloaded from the Kaggle webpage (https://www.kaggle.com/jessicali9530/celeba-dataset)

In consideration to the environment, we will be using Google Drive for storing the code and output results and Google Colab for the execution of the code. The main libraries that we need are:

- Python
- Tensorflow
- Keras
- MTCNN
- InceptionV3
- Pillow
- OS
- Numpy
- Matplotlib
- Pandas
- Collections

- Pyplot

2: Preprocessing of the CelebA Faces Dataset

We will be using MTCNN library from extracting the faces out of images and cropping it. We will make 2 datasets of 50000 images each, one of which only contains the face and the other contains face and background. The flow of program for preprocessing is as follows:

- The First Step is to import all the required libraries which will be used for data loading, preprocessing and saving.

- Then, we will read the attributes file of CelebA dataset which is saved in CSV format. We will then select only the Attractive column and covert it to numpy as we want to convert 50000 images classified as attractive. This is because source material is very important, and it greatly affects the performance of the model. It is easy to miss a critical factor - what does the transformed data going into the cGAN look like. When the data going into a stream is a derivative of another process, as in this case, it is critical to examine the quality of the input data before declaring the results to be useful or invalid. The code to examine the data going into the cGAN is trivial and is included in the final stream. It's worth remembering that the GAN process sees the images at the convoluted pixel level - it sees every spot and wrinkle, every imperfection.

- The next step is to develop code to load the images. We can use the Pillow library to load a given image file, convert it to RGB format (if needed) and return an array of pixel data.

- Next, we can enumerate the directory of images, load each as an array of pixels in turn, and return an array with the images. There are 200K images in the dataset, which is probably more than we need so we can also limit the number of images to load with an argument. The load faces () function implements this. The other worth mentioning that this function does is that it extracts the filename of the images and then uses this filename to filter rows of the attribute file so that it uses the attributes of specific images.

- When working with a GAN, it is easier to model a dataset if the images are small and square in shape. Further, as we are only interested in the face in each photo, and not the background, we can perform face detection and extract only the face before resizing the result to a fixed size. There are many ways to perform face detection. In this case, we will use a pre-trained Multi-Task Cascaded Convolutional Neural Network, or MTCNN. This is a state-of-the-art deep learning model for face detection, described in the 2016 paper titled Joint Face Detection and Alignment Using Multitask Cascaded Convolutional Networks. We will use the implementation provided by Ivan de Paz Centeno in the ipazc/mtcnn project. The MTCNN model is very easy to use.

- First, an instance of the MTCNN model is created, then the detect faces function can be called passing in the pixel data for one image. The result is a list of detected faces, with a bounding box defined in pixel offset values. We can update our example to extract the face from each loaded photo and resize the extracted face pixels to a fixed size. In this case, we will use the square shape

of 64 x 64 pixels. The extract face function implements this, taking the MTCNN model and pixel values for a single photograph as arguments and returning a 64 x 64 × 3 array of pixel values with just the face, or None if no face was detected (which can happen rarely). We will be making two data set for our modelling.

1: The set of images containing only the faces
2: The set of images containing face along with background.

- Finally, we specify the directory of images and saves the required preprocessed images and their ids in compressed npz form. The output of the 2 datasets is shown below.

Figure 1: Sample of Dataset 1 containing only the faces

Figure 2: Sample of Dataset 2 containing faces and background

3. Develop a Conditional DCGAN model for CelebA:

This is the main part of our project. We have used Keras API from tensorflow for training of the model. We will develop 3 different type of models for our conditional generation. Moreover, many small tweaks have been made in the project to avoid modal collapse. The flow of the program in this part is as follow:

- Initially we will import all the required libraries which are necessary for training of our model.

- We will mount our google drive to our Google Colab platform so that we can access files and the save the results in the drive.

- We will set our flags and turn off the warnings for a precise output. Tensorflow and Keras are both very good at giving warnings when syntax being used is out of date, dimensions do not match, or features (such as trainable=True) are not used as required. The problem is you sometimes must run through many warnings before seeing the impact of the issue. In debugging circumstances, being able to shut off warnings can be helpful.

- ## Discriminator Model:

  In this section, we develop a GAN for the faces dataset that we have prepared. The first step is to define the models. The best way to design models in Keras to have multiple inputs is by using the Functional API, as opposed to the Sequential API. We will use the functional API to implement the discriminator, generator, and the composite model.

  Starting with the discriminator model, a new second input is defined that takes an integer for the class label of the image. This has the effect of making the input image conditional on the provided class label. The class label is then passed through an Embedding layer. This means that each of the classes for the CelebA dataset will map to a different 16-element vector representation that will be learned by the discriminator model.

  The output of the embedding is then passed to a fully connected layer with a linear activation. Importantly, the fully connected layer has enough activations that can be reshaped into one channel of a 64x64 image. The activations are reshaped into single 64x64 activation map and concatenated with the input image. This has the effect of looking like a two-channel input image to the next convolutional layer.

  We will use a functional modelling of Keras while using the embedding layers for labels. It is implemented as a modest 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 down sample, and the Adamax version of stochastic gradient descent with a learning rate of 0.0007. While Adam optimizers are generally used, Adamax is recommended when there are embeddings. The discriminator model takes as input one 64 × 64 color image and a class label as embedded vector and outputs a binary prediction as to whether the image is real (class = 1) or fake (class = 0).

  The loss function that we use for our project is binary cross-entropy.

- ## Generator Model:

  Next, the generator model must be updated to take the class label. This has the effect of making the point in the latent space conditional on the provided class label.

  As in the discriminator, the class label is passed through an embedding layer to map it to a unique 16-element vector and is then passed through a fully connected layer with a linear activation before being resized. In this case, the activations of the fully connected layer are resized into a single 4x4 feature map. This is to match the 4x4 feature map activations of the unconditional generator model. The new 4x4 feature map is added as one more channel to the existing 128, resulting in 129 feature maps that are then up sampled as in the prior model.

  The generator model takes as input a point in the latent space and embedded labels, and outputs a single 64x64 color image. This is achieved by using a fully connected layer to interpret the point in the latent space and provide enough activations that can be reshaped into many different (in this case 128) of a low-resolution version of the output image (e.g. 4x4). This is then up sampled 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.

- GAN Model:

Finally, the composite GAN model requires updating. A GAN model can be defined that combines both the generator model and the discriminator model 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.

The new GAN model will take a point in latent space as input and a class label and generate a prediction of whether input was real or fake, as before.

Using the functional API to design the model, it is important that we explicitly connect the image generated output from the generator as well as the class label input, both as input to the discriminator model. This allows the same class label input to flow down into the generator and down into the discriminator.

This larger GAN model takes as input a point in the latent space, uses the generator model to generate an image, which is fed as input to the discriminator model, then output or classified as real or fake.

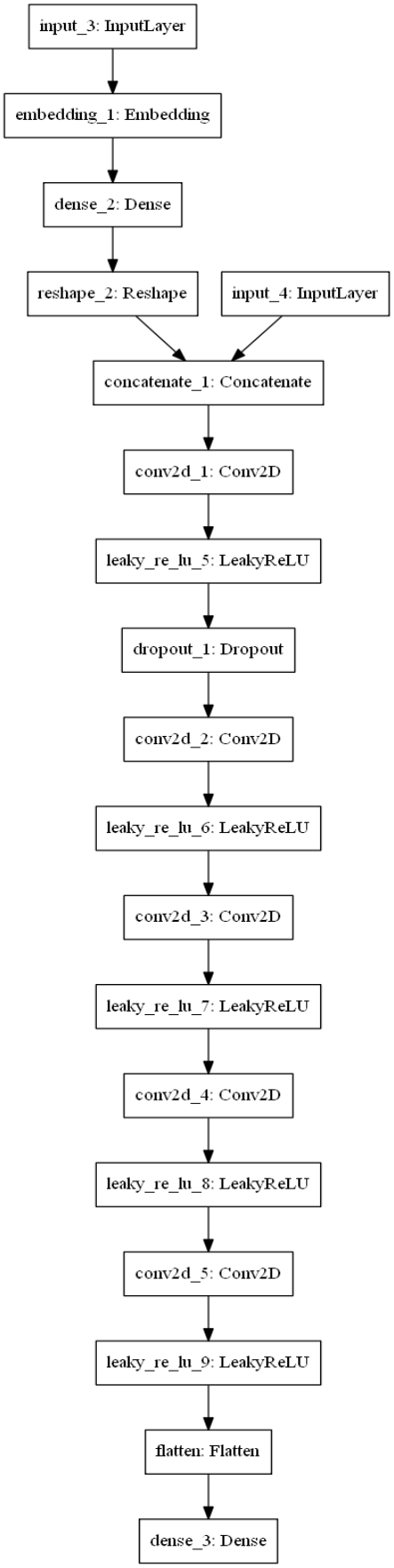


Figure 3: Discriminator Model

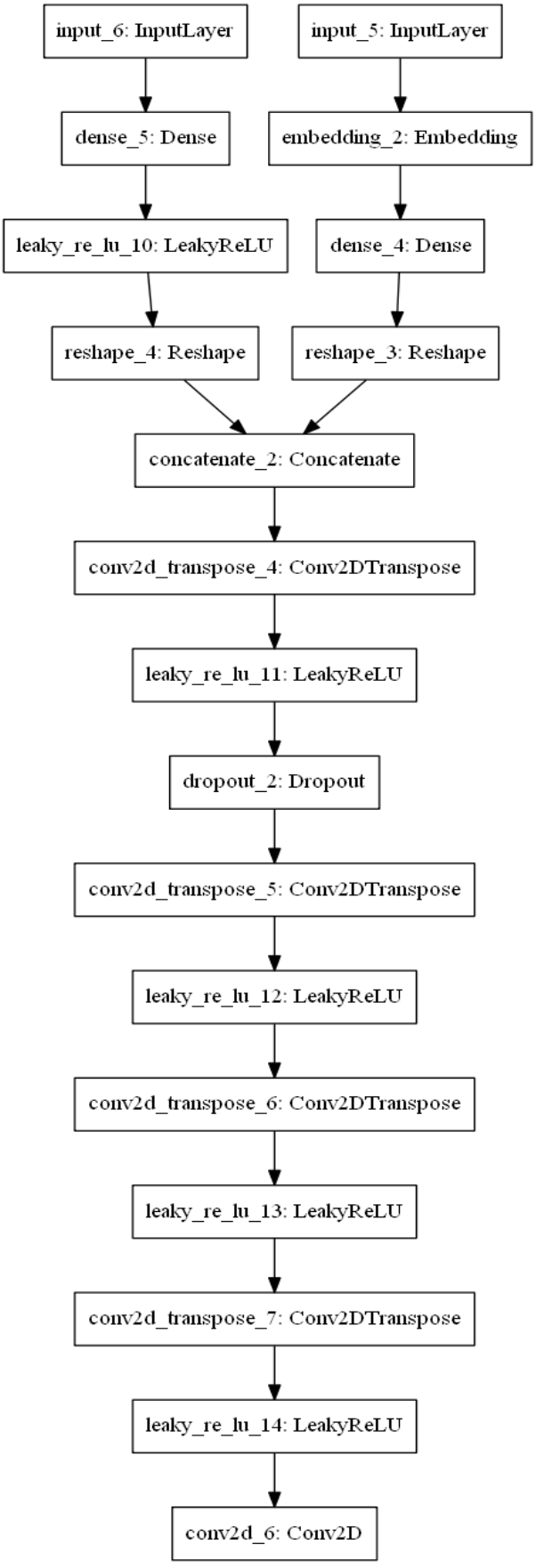


Figure 4: Generator Model

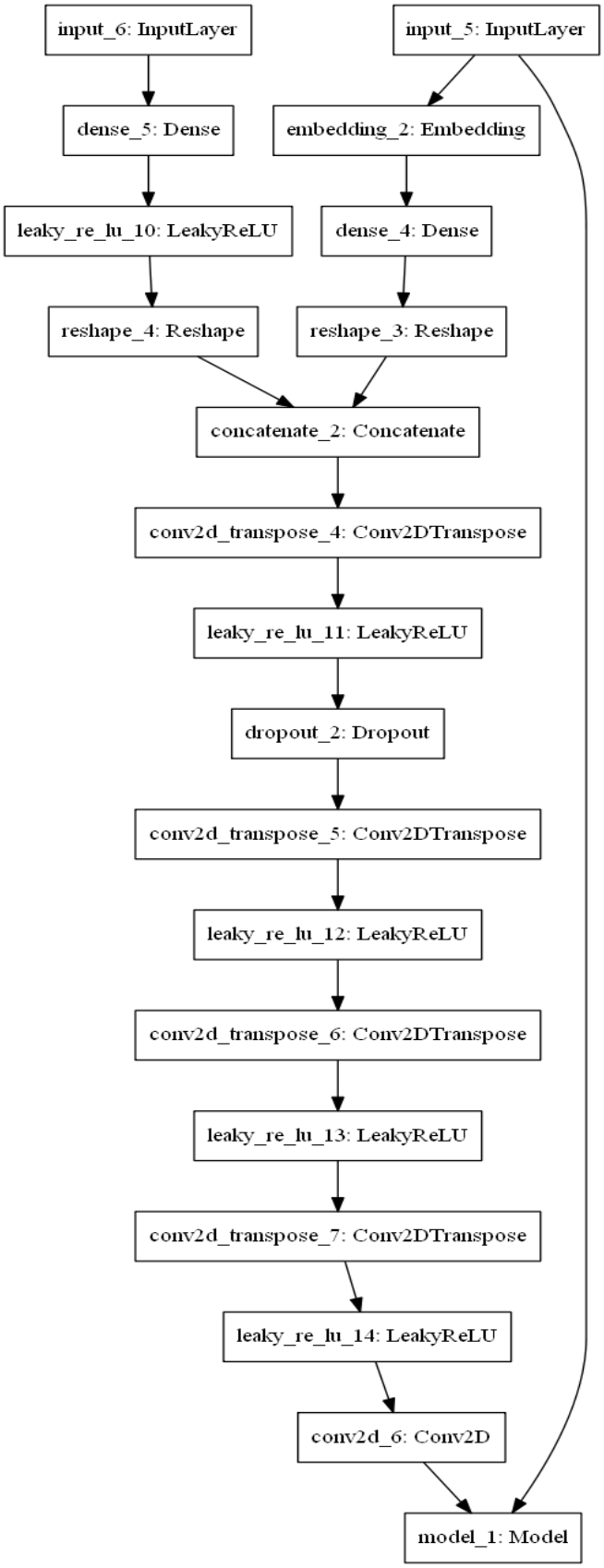


Figure 5: GAN Model

- Assigning Categories to Images:

We have developed three kinds of categories.

- **Dataset 1 (Combined Categories):**

There are circumstances where we want to ensure that a generated image has characteristics, such as a face being attractive, selecting a gender, and having facial features such as high cheek bones and large lips. Looking into the near future, it will be possible to create realistic GAN generated images of models wearing fashionable clothing, with specific expressions, and poses for catalogues. In this project, we could enter in the features: attractive, female, high cheek bones, and large lips in order to get many faces for fashion models.

There were three parts to this process: selecting a subset of faces (only those identified as being "attractive"): Details of the process are discussed in data preprocessing. identifying the characteristics or attributes to be used and their probabilities in the population of images:

0 = featured as attractive and female and not high cheek bone and not large lips
1 = featured as attractive and male
2 = featured as attractive and female and high cheek bone
3 = featured as attractive and female and not high cheek bone and large lips

There are four kinds of embedding and the identity of the embedding (0 thru 3) is included in the generated face. In many ways, those faces identified as being 0 are "female without high cheek bones and without large lips". Those faces identified as 1 (male), are clearly male. Those faces identified as 2 are female with high cheek bones. Feature 3 identifies those faces which supposedly have large lips. The labels (0 thru 3) are added when creating the image.

Dataset 2 (Independent Categories):

In this type of modelling, we have assigned a unique integer to the features and then used these features to train the model independently.

Dataset 3 (Multi-Class Categories):

In this type of modelling, each feature has been assigned a unique integer and then modelling is done using combined 3 categories embedding at a time.

Figure 6: Samples of Categorically assigned dataset 1

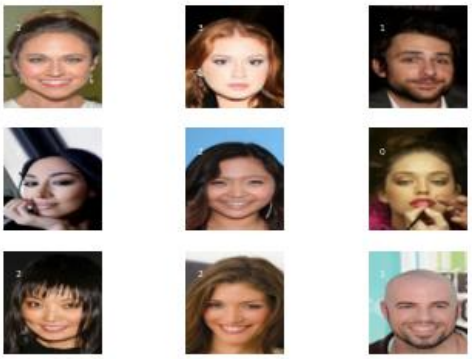


Figure 7: Samples of Categorically assigned dataset 2

- Loading the dataset:
  Now that we have defined the GAN model, we need to train it. But, before we can train the model, we require input data. The first step is to load and scale the pre-processed faces dataset. The saved NumPy array can be loaded, as we did in the previous section, then the pixel values must be scaled to the range [-1,1] to match the output of the generator model. The

- Generating Real Samples:
  We will require one batch (or a half batch) of real images from the dataset each update to the GAN model. A simple way to achieve this is to select a random sample of images from the dataset each time. This, taking the prepared dataset as an argument, selecting and returning a random sample of face images and their corresponding class label for the discriminator, specifically class = 1, indicating that they are real images.

- *Generating Latent Points:* Next, we need inputs for the generator model. These are random points from the latent space, specifically Gaussian distributed random variables. The generate latent points function implements this, taking the size of the latent space as an argument and the number of points required and returning them as a batch of input samples for the generator model.

- Generating Fake Samples:
  Next, we need to use the points in the latent space as input to the generator in order to generate new images. The generate fake samples function implements this, taking the generator model and size of the latent space as arguments, then generating points in the latent space and using them as input to the generator model. The function returns the generated images and their corresponding class label for the discriminator model, specifically class = 0 to indicate they are fake or generated.

- Training the model:
  We are now ready to fit the GAN models. The model is fit for 200 training epochs, which is arbitrary, as the model begins generating plausible faces after perhaps the first few epochs. A batch size of 64 samples is used, and each training epoch involves 50000/64 or about 781 batches of real and fake samples and updates to the model. First, the discriminator model is updated for a half batch of real samples, then a half batch of fake samples, together forming one batch of weight updates. The generator is then updated via the combined GAN model. Importantly, the class label is set to 1 or real for the fake samples. This has the effect of updating the generator toward getting better at generating real samples on the next batch. The train function implements this, taking the defined models, dataset, and size of the latent dimension as arguments and parameterizing the number of epochs and batch size with default arguments.

- Avoiding the mode collapse:
  The following programming fragment also illustrates an approach which often prevents a stream from mode collapse. It depends on having captured discriminator weights, generator weights, and gan weights either during initialization or later in the process when all model losses are within bounds. The definition of model loss bounds are arbitrary but reflect expert opinion about when losses are what might be expected and when they are clearly much too high or much too low. Reasonable discriminator and generator losses are between 0.1 and 1.0, and their arbitrary

bounds are set to between 0.001 and 2.0. Reasonable gan losses are between 0.2 and 2.0 and their arbitrary bounds are set to 0.01 and 4.5.

```
if (d_loss1 < 0.001 or d_loss1 > 2.0) and ijSave > 0:
        print("RELOADING d_model weights",j+1," from ",ijSave)
        d_model.set_weights(d_trainable_weights)
if (d_loss2 < 0.001 or d_loss2 > 2.0) and ijSave > 0:
        print("RELOADING g_model weights",j+1," from ",ijSave)
        g_model.set_weights(g_trainable_weights)
if (g_loss < 0.010 or g_loss > 4.50) and ijSave > 0:
        print("RELOADING gan_models weights",j+1," from ",ijSave)
        gan_model.set_weights(gan_trainable_weights)
```

What happens then is discriminator, generator, and gan weights are collected when all three losses are "reasonable". When an individual model's loss goes out of bounds, then the last collected weights for that model are replaced, leaving the other model weights are they are, and the process moves forward. The process stops when mode collapse appears to be unavoidable even when model weights are replaced. This is identified when a set of model weights continue to be reused but repeatedly result in out of bound model losses. The programming fragment for saving the weights are:

```
if d_loss1 > 0.30 and d_loss1 < 0.95 and d_loss2 > 0.25 and d_loss2 < 0.95 and g_loss > 0.40 and g_loss < 1.50:
        d_trainable_weights = np.array(d_model.get_weights())
        g_trainable_weights = np.array(g_model.get_weights())
        gan_trainable_weights = np.array(gan_model.get_weights())
```

- Summarizing the Performance:

  Finally, after every 5 training epochs, the summarize performance function is called. There is currently no reliable way to automatically evaluate the quality of generated images. Therefore, we must generate images periodically during training and save the model at these times. This both provides a checkpoint that we can later load and use to generate images, and a way to safeguard against the training process failing, which can happen. Below defines the summarize performance and save plot functions. The summarize performance function generates samples and evaluates the performance of the discriminator on real and fake samples. The classification accuracy is reported and might provide insight into model performance.

4. Generation of Images using the trained model

In this section, we will use the trained generator model to conditionally generate new photos of CelebA faces. We can update our code example for generating new images with the model to now generate images conditional on the class label.

The workflow for generation of images is as follows:

- The first step is to load the saved model and confirm that it can generate plausible faces. The model can be loaded using the load model function in the Keras API. We can then generate several random points in the latent space and use them as input to the loaded model to generate new faces. The faces can then be plotted.

- We can explore the latent space by performing vector arithmetic with the generated faces later. First, we must generate many faces and save both the faces and their corresponding latent vectors. We can then review the plot of generated faces and select faces with features we're interested in, note their index (number), and retrieve their latent space vectors for manipulation.

Figure 8: Generated Images for Dataset 1

Figure 9: Generated Images for Dataset 2

5. Fréchet Inception Distance (FID) score

The Fréchet Inception Distance, 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 was proposed and used by Martin Heusel, et al. in their 2017 paper titled GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium. The score was proposed as an improvement over the existing Inception Score, or IS.

The inception score estimates the quality of a collection of synthetic images based on how well the top-performing image classification model Inception v3 classifies them as one of 1,000 known objects. The scores combine both the confidence of the conditional class predictions for each synthetic image (quality) and the integral of the marginal probability of the predicted classes (diversity). The inception score does not capture how synthetic images compare to real images. The goal in developing the FID score was to evaluate synthetic images based on the statistics of a collection of synthetic images compared to the statistics of a collection of real images from the target domain.
Like the inception score, 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. The use of activations from the Inception v3 model to summarize each image gives the score its name of Fréchet Inception Distance. A lower FID indicates better-quality images; conversely, a higher score indicates a lower-quality image and the relationship may be linear.

The FID score is calculated by first loading a pre-trained Inception v3 model. The output layer of the model is removed, and the output is taken as the activations from the last pooling layer, a global spatial pooling layer. This output layer has 2,048 activations; therefore, each image is predicted as 2,048 activation features. This is called the coding vector or feature vector for the image.
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 d2, showing that it is a distance and has squared units. The mu1 and mu2 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 C1 and C2 are the covariance matrix for the real and generated feature vectors, often referred to as sigma. The ||mu1 − mu2||2 refers to the sum squared difference between the two mean vectors. Tr refers to the trace linear algebra operation, e.g. the sum of the elements along the main diagonal of the square matrix. The square root of a matrix is often also written as M1/2, e.g. the matrix to the power of one half, which has the same effect. This operation can fail depending on the values in the matrix because the operation is solved using numerical methods. Commonly, some elements in the resulting matrix may be imaginary, which often can be detected and removed.

Implementation of FID:

The workflow is as follows:

- Import all the necessary libraries.

- Then, we can load the Inception v3 model in Keras directly. This will prepare a version of the inception model for classifying images as one of 1,000 known classes. We can remove the output (the top) of the model via the include top = False argument. Painfully, this also removes the global average pooling layer that we require, but we can add it back via specifying the pooling = 'avg' argument. When the output layer of the model is removed, we must specify the shape of the input images, which is 299 × 299 × 3 pixels, e.g. the input shape = (299,299,3) argument.

- This model can be used to predict the feature vector for one or more images. Our images are likely to not have the required shape. We will use the scikit-image library to resize the NumPy array of pixel values to the required size. The scale images function implements this.

- Once resized, the image pixel values will also need to be scaled to meet the expectations for inputs to the inception model. This can be achieved by calling the preprocess input function. We can update our calculate fid function defined in the theory to take the loaded inception model and two NumPy arrays of image data as arguments, instead of activations. The function will then calculate the activations before calculating the FID score as before

Results of FID:
We have used two sets of 10000 images and calculated FID Score for the three datasets.

FID Score for Dataset 1 (face only of 80 x 80 images): 26.165

FID Score for Dataset 2 (face plus background of 80 x 80 images): 36.146

FID Score for Dataset 3 (face only of 64 x 64 images): 21.361

6: Vector Arithmetic in Latent Space

Typically, new images are generated using random points in the latent space. Taken a step further, points in the latent space can be constructed (e.g. all 0s, all 0.5s, or all 1s) and used as input or a query to generate a specific image. A series of points can be created on a linear path between two points in the latent space, such as two generated images. These points can be used to generate a series of images that show a transition between the two generated images. Finally, the points in the latent space can be kept and used in simple vector arithmetic to create new points in the latent space that, in turn, can be used to generate images. This is an interesting idea, as it allows for the intuitive and targeted generation of images.

The important 2015 paper by Alec Radford, et al. titled Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks introduced a stable model configuration for training deep convolutional neural network models as part of the GAN architecture. In the paper, the authors explored the latent space for GANs fit on several different training datasets, most notably a dataset of celebrity faces. They demonstrated two interesting aspects. The first was the vector arithmetic with faces. For example, a face of a smiling woman minus the face of a neutral woman plus the face of a neutral man resulted in the face of a smiling man.

smiling woman − neutral woman + neutral man = smiling man

Specifically, the arithmetic was performed on the points in the latent space for the resulting faces. On the average of multiple faces with a given characteristic, to provide a more robust result.

Implementation of Vector Arithmetic:

The workflow is as follows:

- Running the image generator loads the model, generates faces, and saves the latent vectors and generated faces. The latent vectors are saved to a compressed NumPy array with the filename latent_points.npz. The 100 generated faces are plotted in a 10 × 10 grid and saved in a file named generated_faces.png. In this case, we have a good collection of faces to work with. Each face has an index that we can use to retrieve the latent vector. For example, the first face is 1, which corresponds to the first vector in the saved array (index 0). We will perform the operation:

  smiling woman − neutral woman + neutral man = smiling man

- Therefore, we need four faces for each of smiling woman, neutral woman, and neutral man. In this case, we will use the following indexes in the image:

  Smiling Woman: [1,61,62,91]
  Neutral Woman: [21,28,85,34]
  Neutral Man: [20,56,88,7]

- Now that we have latent vectors to work with and a target arithmetic, we can get started. First, we can specify our preferred images and load the saved NumPy array of latent points.

- We can explore the latent space by performing vector arithmetic with the generated faces. First, we must generate many faces and save both the faces and their corresponding latent vectors. We can then review the plot of generated faces and select faces with features we're interested in, note their index (number), and retrieve their latent space vectors for manipulation.

- Next, we can retrieve each vector and calculate the average for each vector type (e.g. smiling woman). We could perform vector arithmetic with single images directly, but we will get a more robust result if we work with an average of a few faces with the desired property. The average points function takes the loaded array of latent space points, retrieves each, calculates the average, and returns all the vectors.

- We can now use this function to retrieve all the required points in latent space and generate images.

- Finally, we can use the average vectors to perform vector arithmetic in latent space and plot the result.

Figure 10: Inputs for vector arithmetic

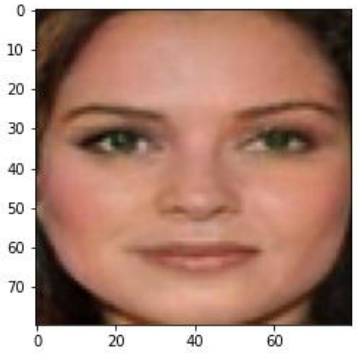


Figure 11: Output of vector arithmetic: neutral woman

7: Explore the Latent Space for Vector Interpolation of Faces

The generator model in the GAN architecture takes a point from the latent space as input and generates a new image. The latent space itself has no meaning. Typically, it is a 100-dimensional hypersphere with each variable drawn from a Gaussian distribution with a mean of zero and a standard deviation of one. Through training, the generator learns to map points onto the latent space with specific output images and this mapping will be different each time the model is trained. The latent space has structure when interpreted by the generator model, and this structure can be queried and navigated for a given model.

Implementation of Vector Interpolation of Faces

The workflow is as follow:

- We can generate points in the latent space, perform the interpolation, then generate an image for each interpolated vector. The result will be a series of images that transition between the two original images.

- we create an interpolation path between two points in the latent space and generate faces along this path. The simplest interpolation we can use is a linear or uniform interpolation between two points in the latent space. We can achieve this using the linspace NumPy function to calculate ratios of the contribution from two points, then enumerate these ratios and construct a vector for each ratio. The interpolate points function implements this and returns a series of linearly interpolated vectors between two points in latent space, including the first and last point.

- Running the example calculates the interpolation path between the two points in latent space, generates images for each, and plots the result. You can see the clear linear progression in ten steps from the first face on the left to the final face on the right.

- In these cases, we have performed a linear interpolation which assumes that the latent space is a uniformly distributed hypercube. Technically, our chosen latent space is a 100-dimension hypersphere or multimodal Gaussian distribution.

Figure12: Vector Interpolation between generated faces

8: Proposed Metric for conditionality assessment

For accessing the condition generation, I have designed a classifier which is trained on the real images and then predict the categories of the randomly generated image and check the accuracy of the generated images

I have achieved an accuracy of 74% of 4-categorical embedding modelling.

Implementation of Classifier

The workflow of classifier is as follow:

- We will define a classifier using our already made discriminator model. The difference is that now we will only use a sequential model and use a cross entropy loss function with SoftMax activation as we want our model to classify images belonging to one of the four categories.

- We will load the real samples from our data for training purpose.

- Generate real samples for each batch of epoch of training. We will convert our categories to one hot encoding so that they can be use as desired output of our classifier.

- We will train the classifier using the real images and then evaluate it on the generated images.

References:

i. Generative Adversarial Networks with Python: Deep Learning Generative Models for Image Synthesis and Image Translation by Jason Brownlee

ii. Generative Adversarial Networks, 2014
https://arxiv.org/abs/1406.2661

iii. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks, 2015.
https://arxiv.org/abs/1511.06434

iv. Jason Brownlee, How to Develop a Conditional GAN (cGAN) From Scratch:
https://machinelearningmastery.com/how-to-develop-a-conditional-generative-adversarial-network-from-scratch

v. Jason Brownlee, How to Explore the GAN Latent Space When Generating Faces,
https://machinelearningmastery.com/how-to-interpolate-and-perform-vector-arithmetic-with-faces-using-a-generative-adversarial-network

vi. Iván de Paz Centeno, MTCNN face detection implementation for TensorFlow, as a PIP package
https://github.com/ipazc/mtcnn

vii. Jeff Heaton, Jeff Heaton's Deep Learning Course
https://www.heatonresearch.com/course/

viii. CelebFaces Attributes (CelebA) Dataset, Kaggle
https://www.kaggle.com/jessicali9530/celeba-dataset

ix. linear interpolation, dcgan.torch Project, GitHub.
https://github.com/soumith/dcgan.torch/

x. GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium, 2017.
https://arxiv.org/abs/1706.08500

xi. Official Implementation in TensorFlow, GitHub.
https://github.com/bioinf-jku/TTUR

xii. Frechet Inception Distance (FID score) in PyTorch, GitHub.
https://github.com/mseitzer/pytorch-fid

xiii. Conditional Generative Adversarial Nets, 2014.
https://arxiv.org/abs/1411.1784

xiv. Conditional Generative Adversarial Nets For Convolutional Face Generation, 2015. https://www.foldl.me/uploads/2015/conditional-gans-face-generation/paper.pdf

xv. How to Train a GAN? Tips and tricks to make GANs work. https://github.com/soumith/ganhacks