Generative AI: How to Fix a Failing GAN Synthesizer

Vincent Granville and Shakti Chaturvedi
vincentg@MLtechniques.com
Version 1.0, August 2023
www.MLtechniques.com

1 Context

Shakti Chaturvedi was working on a Telecom project (public data) while participating in the GenAI training program offered by my AI/ML research lab, to evaluate the quality of the data generated by some home-made generative adversarial networks (GAN). The goal was to predict churn. In particular, we wanted to synthesize observations for the minority group – the users who left and switched to a different provider – in order to augment and rebalance the dataset, to obtain an enhanced training set and get more robust insights and predictions. It quickly became clear that the synthetization was very poor.

We first narrowed down on the three numerical features – tenure (in months), monthly charges, and total charges – ignoring the numerous categorical features. The first 2D GAN based on tenure and monthly charges only, worked as expected. However, when adding total charges (the charges accumulated by a user over the time period studied), the results were bad. This was a surprise since enriching the dataset, for instance by adding labels, usually enhance GAN performance. We then tried many variations of the hyperparameters attached to the three deep neural networks used in GAN: the discriminator, the generator, and the full model. In particular, we tested:

- The learning rates controlling the speed at which the three gradient descent algorithms progress to minimize the loss functions. Too fast leads you to miss the minimum; too slow gets you stuck in local minima.
- The type of gradient descent, Adam being quite popular. You can use a different one for each of the three GAN components: generator, discriminator, full model.
- The loss functions to be minimized. Ideally, we wanted the loss function attached to the generator to be the evaluation metric measuring the quality of the synthesized data, that is, the distance between the real and the synthesized data.
- The activation functions that turn the output of the final layer into synthetic data. In particular, we tried softmax for integer-valued feature, as recommended. In the end, we used a different initialization for each feature, using Keras branches, see here.
- The seeds that initialize the various random number generators. GAN is very sensitive to the seed, and starting with a good seed means starting with a configuration close to an optimum, thus increasing the chance of convergence to a good solution. Our implementation is the only one that uses seeds, allowing for full replicapibily.
- When to stop, that is, how many epochs to use. In the GAN algorithm, we created a synthetization after each epoch, evaluating its quality with a metric called `g_dist` in the Python code, instead of focusing on the values of the loss functions `d_hist` and `g_hist` respectively for the discriminator and generator. In a typical example, the best synthetization was obtained after 12,749 epochs, while the final one after 20,000 epochs was considerably inferior.

There are many other options to play with, for instance the batch size, or the kind of neural network architecture. In particular, Wasserstein GAN (WGAN) seems promising [6]. It consists of using a particular loss function. Despite all our efforts, we were able to obtain modest improvements only. So we worked on a different approach, knowing that adding the feature "total charges" was the source of all the problems.

The issue is that "tenure" and "total charges" are highly correlated, thus hiding the extra information buried in the latter, and confusing GAN. Indeed, tenure (the number of months you stayed with the company), is an excellent predictor of total charges. By decorreleting the two, replacing total charges by total charges residues – the residues in a linear regression – we were able to significantly boost the performance, especially when combined with all the tests previously mentioned. Section 2 describes how we did it, and how we went from GAN to NoGAN.

One important aspect is how to evaluate the quality of synthetic data. Many metrics widely used in many applications have critical flaws, failing to capture non-linear inderdepencies among features. Indeed, even the well-know SDV library, failing just as dramatically as we did with our home-made GAN, rates its own generated data as excellent. We also discuss how to detect these false negatives.

2 GAN enhancements techniques

Before describing the various methods outlined in section 1 and their impact on the results, I discuss the strategy that produced the first substantial improvement. It consists of transforming the real data, then applying GAN to the transformed data to produce synthetic observations, and finally applying the inverse transform to the synthetic data. In essence, it is similar to using transformers in large language models.

2.1 Linear transform to decorrelate features

The transformation must be invertible. For the telecom dataset, due to the high correlation 0.95 between tenure and total charges, linear decorrelation of these two features is the obvious choice. This partial principal component analysis (PCA) worked well. When more features are present, a full PCA is a good choice. It is indeed a linear transformation with invertible matrix. How to inverse PCA is described here. In our case, we simply replaced "total charges" by

$$\text{total charge residues} = \text{total charges} - \beta \times \text{tenure}, \tag{1}$$

where $\beta = 85.20$ is chosen so that residues average zero on the real data. It reduced the correlation from 0.95 down to 0.24. This gives the impression that GAN may underperform in the presence of strong multicollinearity, and that feature orthogonalization to eliminate this problem also improves GAN, in the same way that it improves linear regression. A stronger decorrelation consists in using

$$\beta = \frac{\text{Covar}[\,\text{tenure}, \text{total charges}\,]}{\text{Var}[\,\text{tenure}\,]}$$

in (1), computed on the real data. It leads to zero correlation between tenure and the residues, but with reduced interpretability and $\beta = 92.31$. Scaling the residues (multiplication by a factor) to get its variance similar to that of the other features, may further improve GAN. The quoted values for β and the correlations are based on the real data in this spreadsheet.

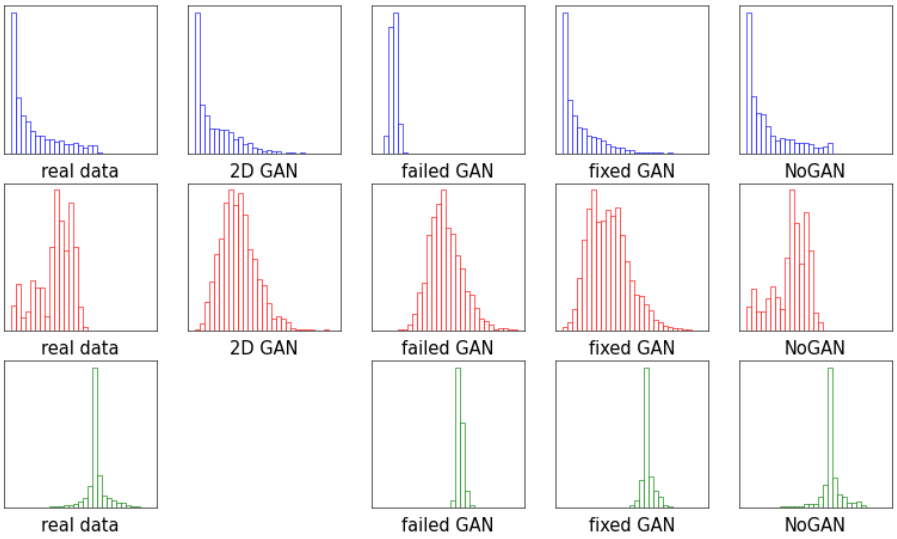

Figure 1: Real data (leftmost) vs synthetic: each row shows a specific feature

2.2 WGAN with PCA transform and standardization

Wasserstein GANs are said to avoid some of the drawbacks of traditional GANs, see here: mode collapse, vanishing gradients, failure to converge and so on. They are based on a different loss function, the Wasserstein loss [Wiki]. See simple example here.

A successful implementation is discussed in [6]. It also involves an artificial binary feature called "labels", to discriminate between real and synthetic observations. However, many of the evaluation metrics fail to detect poor performance. This is illustrated on the circle dataset [2], where marginal distributions and lack of correlations is correctly replicated, but more complex multivariate patterns such as circular distribution, involving two or more features jointly, are entirely missed. Better evaluation metrics for deep comparison of real and synthetic data, are discussed in [3]. See also [1].

Some of the issues related to GAN or WGAN applied to tabular data are: synthetizations strongly depend on the seed, quality significantly varies from epoch to epoch; it may perform poorly on small datasets, requires data standardization, works well on some datasets but not at all on other datasets, and relies heavily on the choice of hyperparameters. This is the reason why I developed NoGAN alternatives [3]. Our version of WGAN is implemented here. Thus far, we are still experimenting with it. However you might want to look at the code, as it illustrates several improvements such as the use of labels and various data transforms including PCA.

Transforming the data via PCA prior to synthesizing, followed by the inverse PCA transform applied to the generated data, has the following benefit. It creates non-correlated features, thus GAN or WGAN does not need to generate the correct correlation structure, as long as it is able to preserve the absence of correlations in the transformed features. The inverse PCA does the job of putting the correct correlations back into the generated data. However, it significantly decreases the value brought in by neural networks. Indeed, it is very easy to sample from the empirical distribution function separately for each feature, without neural networks. Combined with restoring the proper correlation structure, this is identical to using copulas for synthetization. See chapter 10 in my book [5].

There are few examples on how to implement the full inverse PCA transform, even though it is a simple linear transformation. See here for an illustration. Since the goal is to work with uncorrelated features, there are simpler alternatives to PCA and inverse PCA. In section 7.2.1 in [5], I describe a simple method to decorrelate features: it requires the computation of a square root of the covariance matrix. In the end, we haven't succeeded yet with WGAN, which is a back-end modification of GAN. However, in section 3, I explain how we fixed many of the issues using front-end modifications. Yet so far, NoGAN outperforms everything else by a long shot, both in computing time and quality of the synthetizations [3].

3 Front-end enhancements to GAN technology

Eventually, the most significant improvements over standard GAN, were obtained by the following front-end mechanisms:

- Using the simple linear transform described in section 2.1, rather than a full PCA.
- Testing with different seeds, choosing the seed that results in the best synthetization.
- Creating a full synthetization and evaluating the quality of the generated data at each epoch. The final synthetization corresponds to the epoch yielding the best result.

The last item in the above list amounts to using the evaluation metric as the loss function, outside the neural network / gradient descent framework. In the Python code, the evaluation is performed by the `gan_distance` function. The corresponding value (one per epoch) is denoted as `g_dist`. Finally, to generate a full synthetization and evaluation at each epoch, we use `n_eval=1`. The evaluation metric measures the distance between the generated and real data, with 0 being best, and 1 being worst. It is based on a combination of the correlation matrix distance (real vs synthetic) and the Kolmogorov-Smirnov distances for each feature separately, comparing the empirical distributions, synthetic vs real. It is further described in [2]. Better evaluation metrics are described in [3], in particular using the distance between the multivariate empirical distributions(ECDFs). This metric is similar to the total variation distance [Wiki], and works well both with categorical and numerical features.

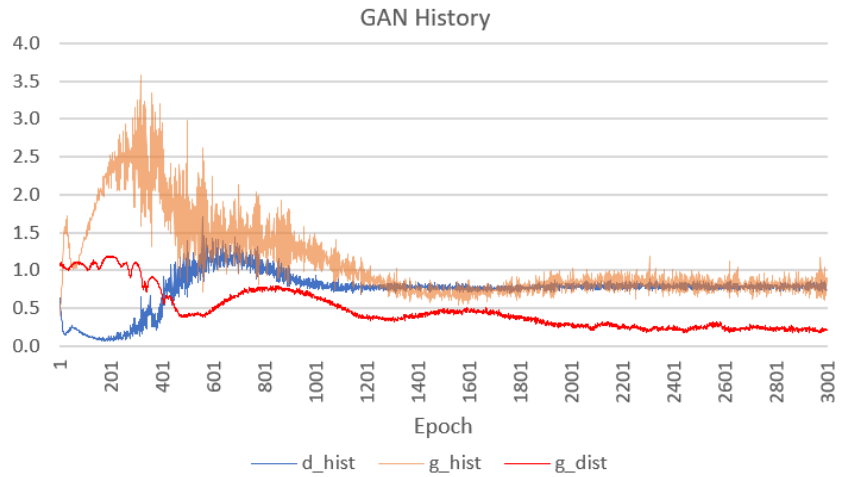

Figure 2: Typical behavior of working GAN, here using two features only

4 Evaluating synthetizations

In this section, I compare the results obtained initially using only two numerical features from the telecom dataset, how adding a new feature made GAN to fail, and the improvements obtained by fixing it. Comparison is based on summary statistics, the `g_dist` distance described in section 3, and plots of the loss functions over the successive epochs. These plots feature the back-end losses `d_hist` and `g_hist` associated respectively to the discriminator and generator models internal to GAN, as well as the front-end `g_dist` evaluation metric.

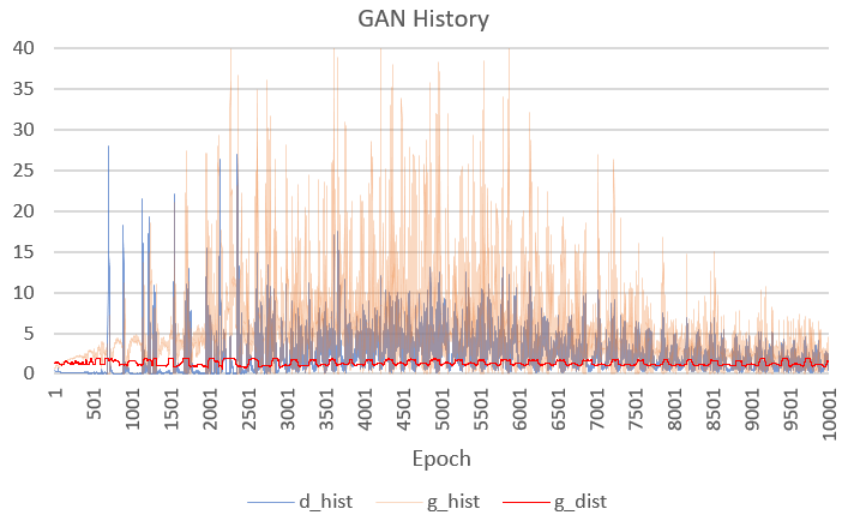

Figure 3: Failed GAN with very high and volatile loss functions

4.1 Loss function history log

Figure 2 shows the history log when using only two features in the dataset: tenure (the number of months the customer has been with the company), and average monthly charges per customer. Everything is working as expected, with losses becoming smaller and smaller over successive epochs, and stabilizing in less than 3000 epochs. By contrast, Figure 3 shows the history log after adding a third feature: total charges per customer, accumulated over the entire tenure period. This new feature is highly correlated to tenure, causing GAN not to converge. In particular, losses are highly volatile and much higher, showing no sign of stabilization even after 10,000 epochs. Figure 4 is identical to Figure 3 except that the Y-axis (loss values) has been truncated to make it comparable to Figure 2. It truly shows how bad the situation is.

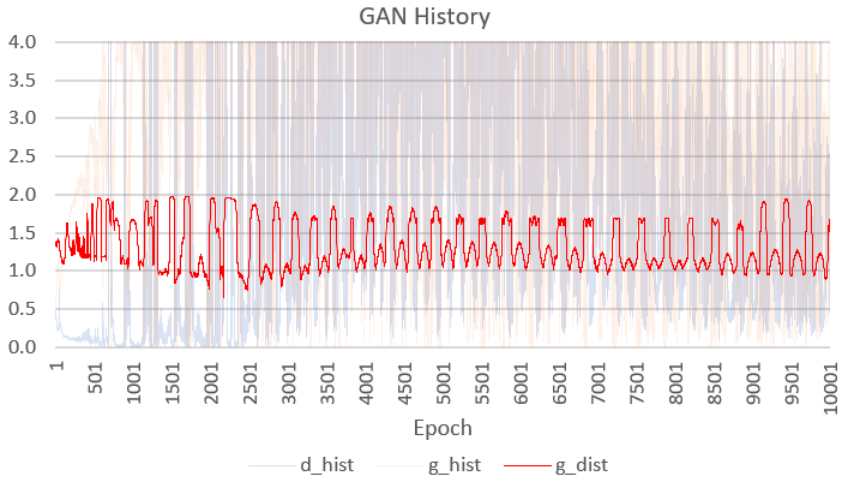

Figure 4: Failed GAN, same as Figure 3 but using scale of Figure 2

Finally, Figure 5 show the loss functions after fixing the issue, using the techniques discussed in section 3. While losses are higher than in Figure 2 (partly because the loss function depends on the number of features), the behavior is back to normal, with convergence at least to a local minimum. Other metrics discussed later in this section confirms the significant improvement.

4.2 Histograms: real versus synthetic data

In Figure 1, we show how the empirical distributions attached to the real data (leftmost plots) compare to that observed in the synthetizations. The top, middle and bottom plots correspond respectively to the following features: tenure, monthly charges, and total charge residues obtained via the transform discussed in section 2.1. From left to right, the histograms correspond to the real data, the 2D GAN based on two features (thus the bottom plot is absent), the failed GAN after the introduction of total monthly charges, the fixed GAN obtained as discussed in section 3, and finally (rightmost plots), the NoGAN synthesizer discussed in [3] and significantly outperforming all other methods both in term of speed and quality.

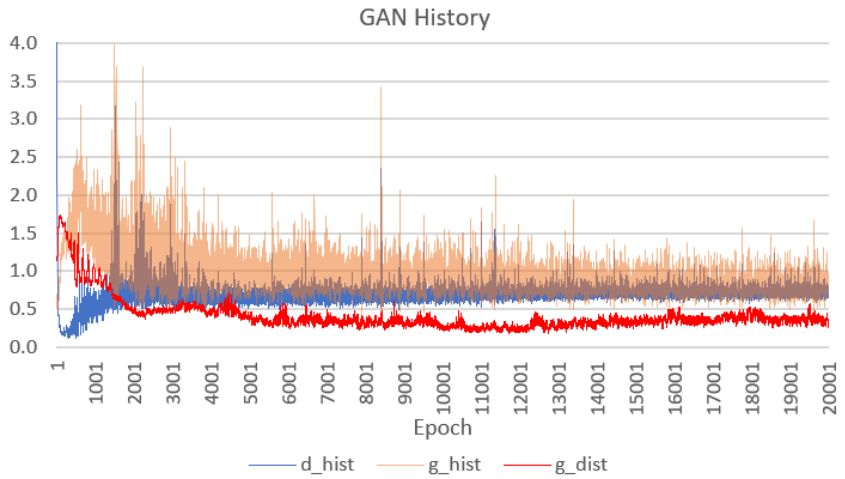

Figure 5: Fixed GAN, with behavior similar to that in Figure 2

4.3 Summary statistics: real versus synthetic data

The summary statistics presented here are an extract from a spreadsheet with detailed results, available here on GitHub. I first compare the impact of choosing the best epoch and best seed, on the quality of the generated data. Then I show how statistics such as correlations, means, and standard deviations are replicated in the generated data, depending on the synthesizer.

	2D GAN		Failed GAN		Fixed, seed 104		Fixed, seed 108	
	g_dist	epoch	g_dist	epoch	g_dist	epoch	g_dist	epoch
Best epoch	0.1739	2971	0.6458	2172	0.4960	1670	0.1793	12,749
Last epoch	0.2152	3000	1.6765	10,000	0.5778	3000	0.3360	20,000

Table 1: Quality g_dist of synthetization, depending on model and epoch number

Table 1 shows how sensitive to the seed GANs are (see parameter seed in the code). It also illustrates that stopping at the last epoch in a run of (say) 10,000 epochs is not only time consuming, but also quite inefficient, due to the volatility of g_dist between successive epochs. The best synthetizations depend strongly on the seed and the stopping rule: at which epoch you decide to stop the training. Clearly the fixed GAN does a lot better than the failed one, when the distance between the real and synthetic data is measured with g_dist.

In the remaining of this section, "Fixed GAN" is the fixed version based on the best tested seed (seed 108). Also, features 1, 2, and 3 are respectively tenure, monthly charges, and total monthly charges (reconstructed if applicable, rather than the residues). This applies to Tables 2 and 3.

Correlations	1/2	1/3	2/3
Real data	0.40	0.95	0.55
Failed GAN	0.50	0.81	0.91
Fixed GAN	0.36	0.99	0.43
NoGAN	0.38	0.95	0.53

Table 2: Feature pairwise correlations

Synthetization	Feature	Mean	Stdev	$P_{.25}$	$P_{.75}$	Min	Max
Real data	1	17.98	19.53	2.00	29.00	1.00	72.00
Failed GAN	1	11.99	2.80	9.96	13.82	4.68	22.29
Fixed GAN	1	14.10	15.98	1.84	21.63	−0.22	106.66
NoGAN	1	17.57	19.10	2.00	28.00	1.00	71.00
Real data	2	74.44	24.67	56.15	94.20	18.85	118.35
Failed GAN	2	107.21	23.39	90.73	121.40	46.02	207.97
Fixed GAN	2	77.84	28.43	55.56	95.44	13.22	194.45
NoGAN	2	74.61	24.75	59.56	94.42	18.97	117.79
Real data	3	1531.80	1890.82	134.50	2331.30	18.85	8684.80
Failed GAN	3	851.73	166.74	733.97	965.02	426.87	1485.58
Fixed GAN	3	1213.48	1464.58	118.47	1893.85	−58.08	9392.70
NoGAN	3	1487.23	1841.66	139.39	2211.96	16.91	8378.25

Table 3: Summary statistics, comparing real data with various synthetizations

Table 3 shows the substantial improvements obtained with the fixed GAN. Yet NoGAN brings the quality to a whole new level, in just 5 seconds rather than 15 minutes to train GAN. In addition, it generates integer numbers for ordinal or binary features, avoiding truncation issues. This is visible if you look at feature 1 (tenure, measured in months). Note that GAN can be accelerated without quality loss by randomly eliminating a large number of observations from the training set, see [4]. NoGAN is discussed in details in [3].

5 Python code for home-made GAN

This version corresponds to the fixed GAN with three features, the linear transform discussed in section 2.1 (thus with total charges replaced by residues), and `seed=108`. A full synthetization is produced and evaluated with `g_dist` at each epoch, thanks to `n_eval=1`. It allows you to identify which epoch produces the best synthetization. The Python code is also on GitHub, here.

```
import numpy as np
import pandas as pd
import os
import matplotlib.pyplot as plt
import random as python_random
from tensorflow import random
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import Adam # type of gradient descent optimizer
from numpy.random import randn
from matplotlib import pyplot
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn import metrics
import scipy
from scipy.stats import ks_2samp
from statsmodels.distributions.empirical_distribution import ECDF


#--- read data and only keep features and observations we want

url = "https://raw.githubusercontent.com/VincentGranville/Main/main/Telecom.csv"
data = pd.read_csv(url)
# data.dropna(how="any",inplace = True)

# keep minority group only (Churn = 'Yes')
data.drop(data[(data['Churn'] == 'No')].index, inplace=True)

# use numerical features only
```

```
features = ['tenure', 'MonthlyCharges', 'TotalCharges']
X = data[features]

# transforming TotalCharges to TotalChargeResidues, add to dataframe
arr1 = data['tenure'].to_numpy()
arr2 = data['TotalCharges'].to_numpy()
arr2 = [eval(i) for i in arr2]          # turn strings to floats
residues = arr2 - arr1 * np.sum(arr2) / np.sum(arr1) # also try arr2/arr1
data['TotalChargeResidues'] = residues

# use numerical features only
# features = ['tenure', 'MonthlyCharges', 'TotalCharges']
features = ['tenure', 'MonthlyCharges', 'TotalChargeResidues']
X = data[features]

# without this, Tensorflow fails
X.to_csv('telecom_temp.csv')
data = pd.read_csv('telecom_temp.csv')

print(data.head())
print (data.shape)
print (data.columns)

nobs = len(X)
n_features = len(features)


#--- some initializations

seed = 108 #104 # to make results replicable (much better than 102, 103)
np.random.seed(seed) # for numpy
random.set_seed(seed) # for tensorflow/keras
python_random.seed(seed) # for python

g_adam = Adam(learning_rate=0.05) # gradient descent for generator
d_adam = Adam(learning_rate=0.001) # gradient descent for discriminator
adam = Adam(learning_rate=0.001) # gradient descent for full GAN
latent_dim = 20 ##
batch_size = 128
n_inputs = n_features
n_outputs = n_features
mode = 'Enhanced' # options: 'Standard' or 'Enhanced'


#--- define models and components (latent data)

def generate_latent_points(latent_dim, n_samples):
   x_input = randn(latent_dim * n_samples)
   x_input = x_input.reshape(n_samples, latent_dim)
   return x_input

def generate_fake_samples(generator, latent_dim, n_samples):
   x_input = generate_latent_points(latent_dim, n_samples) # random N(0,1) data
   X = generator.predict(x_input,verbose=0)
   y = np.zeros((n_samples, 1)) # class label = 0 for fake data
   return X, y

def generate_real_samples(n):
   data_real = pd.DataFrame(data=data, columns=features)
   X = data_real.sample(n) # sample from real data
   y = np.ones((n, 1)) # class label = 1 for real data
   return X, y

def define_generator(latent_dim, n_outputs):
   model = Sequential()
   model.add(Dense(15, activation='relu', kernel_initializer='he_uniform',
```

```
        input_dim=latent_dim))
    model.add(Dense(30, activation='relu'))
    model.add(Dense(n_outputs, activation='linear'))
    model.compile(loss='mean_absolute_error', optimizer=g_adam,
        metrics=['mean_absolute_error']) #
    return model

def define_discriminator(n_inputs):
    model = Sequential()
    model.add(Dense(25, activation='relu', kernel_initializer='he_uniform',
        input_dim=n_inputs))
    model.add(Dense(50, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(loss='binary_crossentropy', optimizer=d_adam, metrics=['accuracy'])
    return model

def define_gan(generator, discriminator):
    discriminator.trainable = False # weights must be set to not trainable
    model = Sequential()
    model.add(generator)
    model.add(discriminator)
    model.compile(loss='binary_crossentropy', optimizer=adam)
    return model


#--- model evaluation, also generate the synthetic data

def gan_distance(model, latent_dim, nobs_synth):

    # generate nobs_synth synthetic rows as X, and return it as data_fake
    # also return correlation distance between data_fake and real data


    latent_points = generate_latent_points(latent_dim, nobs_synth)
    X = model.predict(latent_points, verbose=0)
    data_fake = pd.DataFrame(data=X, columns=features)
    data_real = pd.DataFrame(data=data, columns=features)

    # convert Outcome field to binary 0/1
    #outcome_mean = data_fake.Outcome.mean()
    #data_fake['Outcome'] = data_fake['Outcome'] > outcome_mean
    #data_fake["Outcome"] = data_fake["Outcome"].astype(int)

    # compute correlation distance

    R_data   = np.corrcoef(data_real.T) # T for transpose
    R_data_fake = np.corrcoef(data_fake.T)
    max_R = np.max(abs(R_data-R_data_fake)) #

    # compute Kolmogorov-Smirnov (ks) distance

    max_ks = 0
    for col in features:
        # loop over each numerical feature
        dr = data_real[col]
        dt = data_fake[col]
        stats = ks_2samp(dr, dt)
        ks = stats.statistic
        if ks > max_ks:
            max_ks = ks
    return(data_fake, max_R, max_ks)


#--- main function: train the model

def train(g_model, d_model, gan_model, latent_dim, mode, n_epochs=20000,
```

```
    n_batch=batch_size, n_eval=1):

    # determine half the size of one batch, for updating the discriminator
    half_batch = int(n_batch / 2)
    d_history = []
    g_history = []
    g_dist_history = []
    if mode == 'Enhanced':
       g_dist_min = 999999999.0

    for epoch in range(0,n_epochs+1):

       # update discriminator
       x_real, y_real = generate_real_samples(half_batch) # sample from real data
       x_fake, y_fake = generate_fake_samples(g_model, latent_dim, half_batch)
       d_loss_real, d_real_acc = d_model.train_on_batch(x_real, y_real)
       d_loss_fake, d_fake_acc = d_model.train_on_batch(x_fake, y_fake)
       d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)

       # update generator via the discriminator error
       x_gan = generate_latent_points(latent_dim, n_batch) # random input for generator
       y_gan = np.ones((n_batch, 1))           # label = 1 for fake samples
       g_loss_fake = gan_model.train_on_batch(x_gan, y_gan)
       d_history.append(d_loss)
       g_history.append(g_loss_fake)

       if mode == 'Enhanced':
          (data_fake, max_R, max_ks) = gan_distance(g_model, latent_dim, nobs_synth=1869)
          g_dist = 0.5 * max_R + max_ks
          if g_dist < g_dist_min and epoch > int(0.4*n_epochs):
            g_dist_min = g_dist
            best_data_fake = data_fake
            best_epoch = epoch
            print(" --> Best epoch %6d: max_R = %8.5f | max_ks = %8.5f" %(epoch, max_R,
                max_ks))
       else:
          g_dist = -1.0
       g_dist_history.append(g_dist)

       if epoch % n_eval == 0: # evaluate the model every n_eval epochs
          print('>%d, max_R=%5.3f, max_ks=%5.3f d=%5.3f g=%5.3f g_dist=%5.3f
              g_dist_min=%5.3f'
             % (epoch, max_R, max_ks, d_loss, g_loss_fake, g_dist, g_dist_min))
          plt.subplot(1, 1, 1)
          plt.plot(d_history, label='d')
          plt.plot(g_history, label='gen')
          # plt.show() # un-comment to see the plots
          plt.close()

    OUT=open("history.txt","w")
    for k in range(len(d_history)):
       OUT.write("%6.4f\t%6.4f\t%6.4f\n" %(d_history[k],g_history[k],g_dist_history[k]))
    OUT.close()

    if mode == 'Standard':
       # best synth data is assumed to be the one produced at last epoch
       best_epoch = epoch
       (best_data_fake, max_R, max_ks) = gan_distance(g_model, latent_dim,
           nobs_synth=1869)
       g_dist_min = 0.5 * max_R + max_ks

    return(g_model, best_data_fake, g_dist_min, best_epoch)


#--- main part for building & training model
```

```
discriminator = define_discriminator(n_inputs)
discriminator.summary()
generator = define_generator(latent_dim, n_outputs)
generator.summary()
gan_model = define_gan(generator, discriminator)

model, data_fake, g_dist, best_epoch = train(generator, discriminator, gan_model,
    latent_dim, mode)

data_fake.to_csv('telecom_gan.csv')
print(data_fake.head(10))
print("Distance between real/synthetic: %5.3f" % (g_dist))
print("Based on epoch number: %5d" % (best_epoch))
```

References

[1] Fida Dankar et al. A multi-dimensional evaluation of synthetic data generators. *IEEE Access*, pages 11147–11158, 2022. [Link]. 2

[2] Vincent Granville. Generative AI: Synthetic data vendor comparison and benchmarking best practices. *Preprint*, pages 1–13, 2023. MLTechniques.com [Link]. 2, 3

[3] Vincent Granville. Generative AI technology break-through: Spectacular performance of new synthesizer. *Preprint*, pages 1–16, 2023. MLTechniques.com [Link]. 2, 3, 5, 6

[4] Vincent Granville. Massively speed-up your learning algorithm, with stochastic thinning. *Preprint*, pages 1–13, 2023. MLTechniques.com [Link]. 6

[5] Vincent Granville. *Synthetic Data and Generative AI*. Elsevier, 2024. [Link]. 3

[6] Chang Su, Linglin Wei, and Xianzhong Xie. Churn prediction in telecommunications industry based on conditional Wasserstein GAN. *IEEE International Conference on High Performance Computing, Data, and Analytics*, pages 186–191, 2022. IEEE HiPC 2022 [Link]. 1, 2