CoSynthEx

Release 1.0

Camila Riccio-Rengifo, Jorge Finke, Camilo Rocha

Oct 04, 2023

CONTENTS:

Here we introduce a software tool designed to generate synthetic expression data conditioned on phenotypic traits and sample conditions, such as control or stress. This software leverages the capabilities of a Conditional Generative Adversarial Network (cGAN) to create realistic and customizable synthetic expression data for a wide range of applications in genomics and bioinformatics.

CHAPTER

ONE

MAIN

1.1 CoSynthEx module

class CoSynthEx.**Discriminator**(*input_size*, *n_covariates*, *hidden_size*)

Bases: torch.nn.modules.module.Module

Class for the discriminator of a cGAN.

The discriminator has 4 layers: one input layer, two hidden layers, and one output layer.

The input layer has input_size + n_covariates nodes.

The hidden layers have hidden_size nodes.

The output layer has 1 node.

The activation function for the hidden layers is ReLU.

The activation function for the output layer is sigmoid.

The hidden layers have dropout with probability 0.3.

forward(*x*, *condition*, *phenotype*)

Defines the computation performed at every call.

Should be overridden by all subclasses.

> **Note:** Although the recipe for forward pass needs to be defined within this function, one should call the Module instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

training: bool

class CoSynthEx.**Generator**(*input_size*, *n_covariates*, *hidden_size*, *output_size*)

Bases: torch.nn.modules.module.Module

Class for the generator of a cGAN.

The generator has 4 layers: one input layer, two hidden layers, and one output layer.

The input layer has input_size + n_covariates nodes.

The hidden layers have hidden_size nodes.

The output layer has output_size nodes.

The activation function for the hidden layers is ReLU.

The activation function for the output layer is sigmoid.

forward(*x*, *condition*, *phenotype*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

training: bool

CoSynthEx.**plot_all_expression_fake_vs_real**(*scaled_data_real*, *scaled_data_fake*, *output_path_figures='./'*)
Function to plot the expression of all genes in the real and fake data.

Parameters

- **scaled_data_real** (*pandas.DataFrame*) – scaled real data (i.e., count data after log2 transformation and minmax normalization).
- **scaled_data_fake** (*pandas.DataFrame*) – scaled fake data (i.e., expression data as output by the generator).
- **output_path_figures** (*str*) – path to save the figure.

CoSynthEx.**plot_gene_expression_fake_vs_real**(*control_samples*, *stress_samples*, *scaled_data_real*, *scaled_data_fake*, *gene=None*, *output_path_figures='./'*)
Function to plot the expression of a gene in the real and fake data.

Parameters

- **control_samples** (*list*) – list of the control samples.
- **stress_samples** (*list*) – list of the stress samples.
- **scaled_data_real** (*pandas.DataFrame*) – scaled real data (i.e., count data after log2 transformation and minmax normalization).
- **scaled_data_fake** (*pandas.DataFrame*) – scaled fake data (i.e., expression data as output by the generator).
- **gene** (*str*) – gene to plot.
- **output_path_figures** (*str*) – path to save the figure.

Returns the gene that was plotted.

Return type str

CoSynthEx.**plot_loss**(*D_losses*, *G_losses*, *output_path_figures='./'*)
Function to plot the loss curves for the generator and discriminator.

Parameters

- **D_losses** (*list*) – list of the losses for the discriminator.
- **G_losses** (*list*) – list of the losses for the generator.
- **output_path_figures** (*str*) – path to save the figure.

CoSynthEx.**train_model**(*G*, *D*, *criterion*, *G_optimizer*, *D_optimizer*, *num_epochs*, *batch_size*, *input_size*, *X_train*, *condition_train*, *phenotype_train*)
Function to train a cGAN. For each epoch, the function loops through the batches of the training data. For each batch, the function trains the discriminator and generator.

Parameters

- **G** (*Generator.*) – instance of the generator.
- **D** (*Discriminator.*) – instance of the discriminator.
- **G_optimizer** (*torch.optim.Adam.*) – optimizer for the generator.
- **D_optimizer** (*torch.optim.Adam.*) – optimizer for the discriminator.
- **num_epochs** (*int.*) – number of epochs.
- **batch_size** (*int.*) – batch size.
- **input_size** (*int.*) – input size for the generator and discriminator.
- **X_train** (*pandas.DataFrame.*) – training/real data.
- **condition_train** (*pandas.Series.*) – binary covariate for the training/real data.
- **phenotype_train** (*pandas.DataFrame.*) – continuous covariate for the training/real data.

Returns lists of the losses for the generator and discriminator, the trained generator, and the trained discriminator.

Return type list, list, *Generator*, Discriminator.

CHAPTER

TWO

COSYNTHEX USER MANUAL

Below we present examples of how to use the main functions of the software.

2.1 Import libraries

[1]:
```
import pandas as pd
import numpy as np
#from sklearn import preprocessing
import torch
import torch.nn as nn
import matplotlib.pyplot as plt
import seaborn as sns
import CoSynthEx as cse
```

2.2 Specify file paths

[2]:
```
input_path = '../input_data/'
output_path = '../output_data/'
```

2.3 Load Real Data

For this example we will use expression and phenotypic data of rice (Oryza sativa). The RNA-seq data was accessed from the GEO database (accession number GSE98455). It corresponds to 57 845 gene expression profiles of shoot tissues measured for control and salt conditions in 92 accessions of the Rice Diversity Panel 1, with 2 biological replicates. The phenotypic data is a subset of the supplementary file 1 included in the work of Campbell, M. T. (2017). A total of 3 phenotypic traits were used: shoot K content, and root and shoot biomass. These traits were measured for the same 92 genotypes, under control and stress conditions.

2.3.1 Raw count expression data

We will use only a random subset of genes to speed up the computation.

[3]:
```
# Expression data
Expr_control = pd.read_csv(input_path + 'Exp_control_GSE98455.csv', index_col=0)
Expr_stress = pd.read_csv(input_path + 'Exp_stress_GSE98455.csv', index_col=0)
data_concat = pd.concat([Expr_control, Expr_stress], axis=1)

# random sample 5000 genes for proof of concept
data_concat = data_concat.sample(n=5000, axis=0) #,random_state=1)

count_data_real = data_concat.T #rows are samples, columns are genes
```

2.3.2 Pre-processing

Gene expression data often follows a skewed or log-normal distribution, where most values are concentrated around a few low values with a long tail of high values. Applying a **log2 transformation** helps to make the data more symmetric and reduces the impact of extreme values. It can also make the data more amenable to linear modeling assumptions, which many machine learning algorithms, including neural networks, assume.

Min-Max normalization scales the data to a fixed range (typically between 0 and 1). This is often preferred when you want to preserve the relative differences in gene expression levels while ensuring that the data is within a consistent range. It helps neural networks converge more efficiently during training.

[4]:
```
# log2 transformation
log_data_real = np.log2(count_data_real + 1)

# Min-max normalization
# minmax_scaler = preprocessing.MinMaxScaler(feature_range=(0, 1))
# scaled_data_real = minmax_scaler.fit_transform(log_data_real.values)

# manual Min-max normalization
eps=1e-8
log_min = log_data_real.values.min()
log_max = log_data_real.values.max()
scaled_data_real = log_data_real.apply(lambda x: (x-log_min+eps)/(log_max-log_min+eps))

# scaled_data_real as dataframe
scaled_data_real = pd.DataFrame(scaled_data_real, index=log_data_real.index, columns=log_
↪data_real.columns)

# dimensions of expression data
sample_length = len(log_data_real.columns)
num_samples = len(log_data_real.index)
```

2.3.3 Covariates data

(sample condition and phenotype values)

[5]:
```
sample_info_real = pd.read_csv(input_path + 'sample_info_GSE98455.csv', index_col=0)
sample_info_real['name'] = sample_info_real.index
sample_info_real['genotype'] = sample_info_real['genotype'].astype(str)
sample_info_real['genotype_name'] = sample_info_real[['genotype', 'name']].apply(lambda␣
↪x: '_'.join(x), axis=1)

control_samples = sample_info_real[sample_info_real.condition == 'control'].index
stress_samples = sample_info_real[sample_info_real.condition == 'stress'].index

# categorical variable (control or stress)
conditions = ['control','stress']
condition_data = pd.Series(pd.factorize(sample_info_real['condition'])[0])

# numerical variables (phenotype: K_shoot, BM_root, BM_shoot)
traits = ['K_shoot', 'BM_root', 'BM_shoot']
phenotype_data_real_df = sample_info_real[['condition'] + traits]
phenotype_data_real = sample_info_real[traits].values
```

2.4 Define parameters

[6]:
```
# parameters for cGAN
input_size = sample_length
hidden_size = 80
output_size = sample_length
n_covariates = 4 # condition (control/stress) + 3 phenotypic traits
num_epochs = 200
batch_size = 46
learning_rate = 0.0001
```

2.5 Train cGAN

[7]:
```
# create model
G = cse.Generator(input_size, n_covariates, hidden_size, output_size)
D = cse.Discriminator(input_size, n_covariates, hidden_size)

# loss function
criterion = nn.BCELoss()

# optimizer
G_optimizer = torch.optim.Adam(G.parameters(), lr=learning_rate)
D_optimizer = torch.optim.Adam(D.parameters(), lr=learning_rate)

# train model
D_losses, G_losses, G, D = cse.train_model(G, D, criterion, G_optimizer, D_optimizer,
                                           num_epochs, batch_size, input_size,
```

(continues on next page)

(continued from previous page)

```
                                        scaled_data_real, condition_data, phenotype_data_
↪real_df[traits])

# save trained model
torch.save(G.state_dict(), output_path + 'G.pth')
torch.save(D.state_dict(), output_path + 'D.pth')
```

```
100%|| 200/200 [00:27<00:00,  7.17it/s]
```

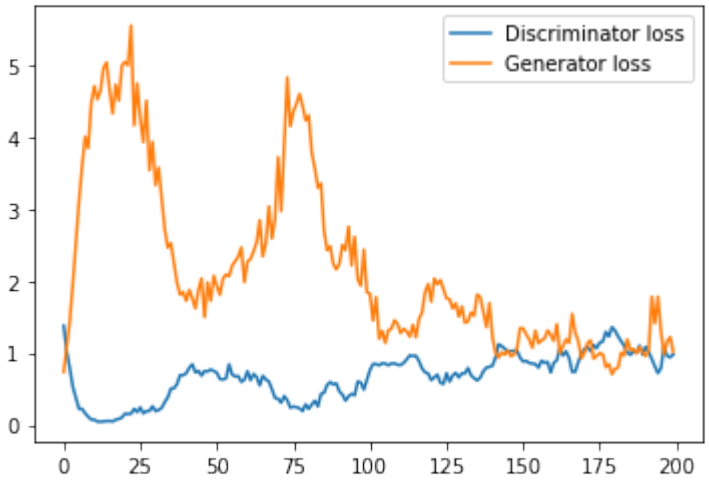
2.6 Plot loss curves

The generator's loss should initially be high and then decrease over time. It should ideally converge to a low value. This indicates that the generator is learning to generate data that is increasingly more similar to the real data distribution.

The discriminator's loss should start relatively high and gradually decrease. It should ultimately plateau or oscillate around a low value. This indicates that the discriminator is having a harder time distinguishing between real and generated data, which is a sign of successful training.

Ideally, the generator and discriminator should be in a state of competition and adaptation throughout training. The generator aims to produce data that is increasingly convincing to the discriminator, while the discriminator tries to become better at distinguishing real from fake data.

[8]:
```
cse.plot_loss(D_losses, G_losses, output_path_figures = output_path)
```



2.7 Load pre-trained cGAN

If you have already trained the cGAN and saved the discriminator and generator, you can load them without needing to retrain the model.

[9]:
```
# load trained model
G = cse.Generator(input_size, n_covariates, hidden_size, output_size)
D = cse.Discriminator(input_size, n_covariates, hidden_size)
G.load_state_dict(torch.load(output_path + 'G.pth'))
D.load_state_dict(torch.load(output_path + 'D.pth'))
```

2.8 Use trained cGAN to generate synthetic data

With the trained caGAN you can generate synthetic expression data using the actual phenotypic data for the corresponding control and stress samples. You can also reverse the initial transformations of the data to obtain the raw counts.

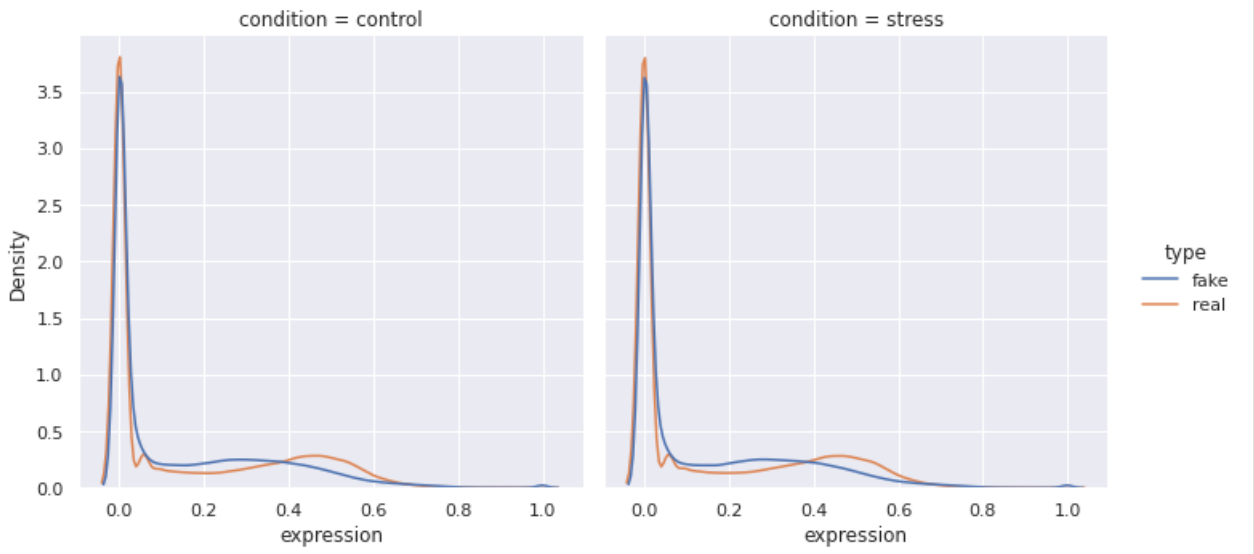
[10]:
```
 # generate fake expression data
noise = torch.randn(num_samples, input_size)
condition = torch.tensor(condition_data, dtype=torch.float32).view(-1, 1)
phenotype = torch.tensor(phenotype_data_real, dtype=torch.float32)
scaled_data_fake = G(noise, condition, phenotype)
scaled_data_fake = scaled_data_fake.detach().numpy()
scaled_data_fake = pd.DataFrame(scaled_data_fake, index=sample_info_real.index,␣
↪columns=count_data_real.columns)

# reverse minmax scaler into a dataframe
# log_data_fake = minmax_scaler.inverse_transform(scaled_data_fake)
log_data_fake = scaled_data_fake.apply(lambda x: x*(log_max-log_min) + log_min)
log_data_fake = pd.DataFrame(log_data_fake, index=scaled_data_fake.index, columns=scaled_
↪data_fake.columns)
# reverse log2 transformation into a dataframe
count_data_fake = np.power(2, log_data_fake) - 1
```

2.9 Plot results

We can plot a compendium of all the expression data generated. It can be seen how the synthetic data are quite close to the distribution of the real data. However, at this scale it is not possible to appreciate the difference between the control and stress data.

[11]:
```
cse.plot_all_expression_fake_vs_real(scaled_data_real, scaled_data_fake, output_path)
```

```
<Figure size 432x288 with 0 Axes>
```

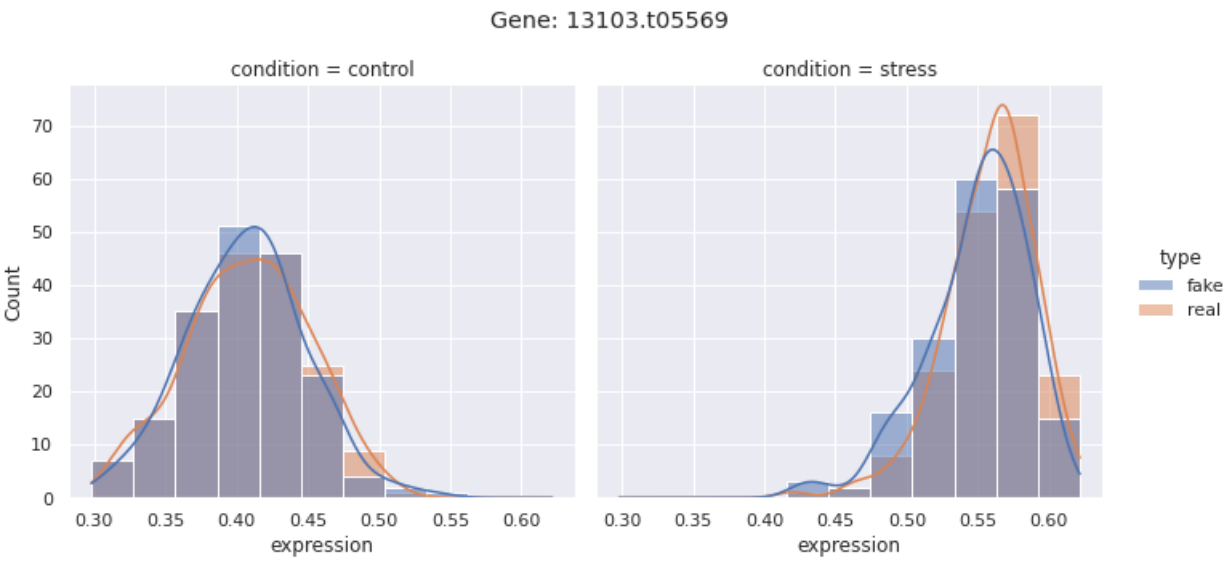


To better appreciate the difference between the control and stress data, you have to focus on one gene. As shown below,

it is possible to graph the real and synthetic expression of a random gene, where the difference between the control and stress data can be seen.

[12]:
```
gene = cse.plot_gene_expression_fake_vs_real(control_samples, stress_samples,
                                             scaled_data_real, scaled_data_fake,
                                             output_path_figures=output_path)
```

```
<Figure size 432x288 with 0 Axes>
```



CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX