# Gretel Synthetics

Gretel.ai

Feb 25, 2025

# CONTENTS:

# CHAPTER ONE

# DOCUMENTATION

- Get started with gretel-synthetics
- Configuration
- Train your model
- Generate synthetic records

CHAPTER

TWO

# TRY IT OUT NOW

If you want to quickly discover gretel-synthetics, simply click the button below and follow the tutorials!

Check out additional examples here.

CHAPTER

# THREE

# GETTING STARTED

This section will guide you through installation of `gretel-synthetics` and dependencies that are not directly installed by the Python package manager.

## 3.1 Dependency Requirements

By default, we do not install certain core requirements, the following dependencies should be installed *external to the installation* of `gretel-synthetics`, depending on which model(s) you plan to use.

- Torch: Used by Timeseries DGAN and ACTGAN (for ACTGAN, Torch is installed by SDV), we recommend version 2.0
- SDV (Synthetic Data Vault): Used by ACTGAN, we recommend version 0.17.x

These dependencies can be installed by doing the following:

```
pip install sdv<0.18 # for ACTGAN
pip install torch==2.0 # for Timeseries DGAN
```

To install the actual `gretel-synthetics` package, first clone the repo and then…

```
pip install -U .
```

*or*

```
pip install gretel-synthetics
```

*then…*

```
pip install jupyter
jupyter notebook
```

When the UI launches in your browser, navigate to `examples/synthetic_records.ipynb` and get generating!

If you want to install `gretel-synthetics` locally and use a GPU (recommended):

1. Create a virtual environment (e.g. using `conda`)

```
conda create --name tf python=3.9
```

1. Activate the virtual environment

```
conda activate tf
```

1. Run the setup script `./setup-utils/setup-gretel-synthetics-tensorflow24-with-gpu.sh`

The last step will install all the necessary software packages for GPU usage, `tensorflow=2.8` and `gretel-synthetics`. Note that this script works only for Ubuntu 18.04. You might need to modify it for other OS versions.

# CHAPTER FOUR

# TIMESERIES DGAN OVERVIEW

The timeseries DGAN module contains a PyTorch implementation of a DoppelGANger model that is optimized for timeseries data. Similar to tensorflow, you will need to manually install pytorch:

```
pip install torch==1.13.1
```

This notebook shows basic usage on a small data set of smart home sensor readings.

# CHAPTER FIVE

# ACTGAN OVERVIEW

ACTGAN (Anyway CTGAN) is an extension of the popular CTGAN implementation that provides some additional functionality to improve memory usage, autodetection and transformation of columns, and more.

To use this model, you will need to manually install SDV:

```
pip install sdv<0.18
```

Keep in mind that this will also install several dependencies like PyTorch that SDV relies on, which may conflict with PyTorch versions installed for use with other models like Timeseries DGAN.

The ACTGAN interface is a superset of the CTGAN interface. To see the additional features, please take a look at the ACTGAN demo notebook in the `examples` directory of this repo.

CHAPTER

SIX

# MODULES

## 6.1 Config

This module provides a set of dataclasses that can be used to hold all necessary confguration parameters for training a model and generating data.

For example usage please see our Jupyter Notebooks.

`class gretel_synthetics.config.`**`BaseConfig`**`(`*input_data_path: str | None = None*, *validation_split: bool = True*, *checkpoint_dir: str | None = None*, *training_data_path: str | None = None*, *field_delimiter: str | None = None*, *field_delimiter_token: str = '<d>'*, *model_type: str | None = None*, *max_lines: int = 0*, *overwrite: bool = False*, *epoch_callback: Callable | None = None*, *max_training_time_seconds: int | None = None*, *vocab_size: int = 20000*, *character_coverage: float = 1.0*, *pretrain_sentence_count: int = 1000000*, *max_line_len: int = 2048*`)`

This class should not be used directly, engine specific classes should derived from this class.

**`as_dict`**`()`

Serialize the config attrs to a dict

**`checkpoint_dir: str = None`**

Directory where model data will be stored, user provided.

**`epoch_callback: Callable | None = None`**

Callback to be invoked at the end of each epoch. It will be invoked with an EpochState instance as its only parameter. NOTE that the callback is deleted when save_model_params is called, we do not attempt to serialize it to JSON.

**`field_delimiter: str | None = None`**

If the input data is structured, you may specify a field delimiter which can be used to split the generated text into a list of strings. For more detail please see the `GenText` class in the `generate.py` module.

**`field_delimiter_token: str = '<d>'`**

Depending on the tokenizer used, a special token can be used to represent characters. For tokenizers, like SentencePiece that support this, we will replace the field delimiter char with this token to provide better learning and generation. If the tokenizer used does not support custom tokens, this value will be ignored

`abstract` **`get_generator_class`**`()` → None

This must be implemented by all specific configs. It should return the class that should be used as the Generator for creating records.

`abstract get_training_callable()` → Callable

This must be implemented by all specific configs. It should return a callable that should be used as the entrypoint for training a model.

`gpu_check()`

Optionally do a GPU check and warn if a GPU is not available, if not overridden, do nothing

`input_data_path: str = None`

Path to raw training data, user provided.

`max_lines: int = 0`

The maximum number of lines to utilize from the raw input data.

`max_training_time_seconds: int | None = None`

If set, training will cease after the number of seconds specified elapses. This timeout will be evaluated after each epoch.

`model_type: str = None`

A string version of the model config class. This is used to keep track of what underlying engine was used when writing the config to a file. This will be automatically updated during construction.

`overwrite: bool = False`

Set to `True` to automatically overwrite previously saved model checkpoints. If `False`, the trainer will generate an error if checkpoints exist in the model directory. Default is `False`.

`training_data_path: str = None`

Where annotated and tokenized training data will be stored. This attr will be modified during construction.

`validation_split: bool = True`

Use a fraction of the training data as validation data. Use of a validation set is recommended as it helps prevent over-fitting and memorization. When enabled, 20% of data will be used for validation.

`gretel_synthetics.config.CONFIG_MAP = {'TensorFlowConfig': <class 'gretel_synthetics.config.TensorFlowConfig'>}`

A mapping of configuration subclass string names to their actual classes. This can be used to re-instantiate a config from a serialized state.

`gretel_synthetics.config.LocalConfig`

alias of *`TensorFlowConfig`*

**class** gretel_synthetics.config.**TensorFlowConfig**(*input_data_path: str | None = None*, *validation_split: bool = True*, *checkpoint_dir: str | None = None*, *training_data_path: str | None = None*, *field_delimiter: str | None = None*, *field_delimiter_token: str = '<d>'*, *model_type: str | None = None*, *max_lines: int = 0*, *overwrite: bool = False*, *epoch_callback: Callable | None = None*, *max_training_time_seconds: int | None = None*, *vocab_size: int = 20000*, *character_coverage: float = 1.0*, *pretrain_sentence_count: int = 1000000*, *max_line_len: int = 2048*, *epochs: int = 100*, *early_stopping: bool = True*, *early_stopping_patience: int = 5*, *best_model_metric: str | None = None*, *early_stopping_min_delta: float = 0.001*, *batch_size: int = 64*, *buffer_size: int = 10000*, *seq_length: int = 100*, *embedding_dim: int = 256*, *rnn_units: int = 256*, *learning_rate: float = 0.01*, *dropout_rate: float = 0.2*, *rnn_initializer: str = 'glorot_uniform'*, *dp: bool = False*, *dp_noise_multiplier: float = 0.1*, *dp_l2_norm_clip: float = 3.0*, *dp_microbatches: int = 1*, *gen_temp: float = 1.0*, *gen_chars: int = 0*, *gen_lines: int = 1000*, *predict_batch_size: int = 64*, *reset_states: bool = True*, *save_all_checkpoints: bool = False*, *save_best_model: bool = True*)

TensorFlow config that contains all of the main parameters for training a model and generating data.

**Parameters**

- **epochs** (*optional*) – Number of epochs to train the model. An epoch is an iteration over the entire training set provided. For production use cases, 15-50 epochs are recommended. The default is `100` and is intentionally set extra high. By default, `early_stopping` is also enabled and will stop training epochs once the model is no longer improving.
- **early_stopping** (*optional*) – deduce when the model is no longer improving and terminating training.
- **early_stopping_patience** (*optional*) – in the model. After this number of epochs, training will terminate.
- **best_model_metric** (*optional*) – The metric to use to track when a model is no longer improving. Alternative options are "val_acc" or "acc". A error will be raised if a valid value is not specified.
- **early_stopping_min_delta** (*optional*) – as an improvement, i.e. an absolute change of less than min_delta will count as no improvement.
- **batch_size** (*optional*) – Number of samples per gradient update. Using larger batch sizes can help make more efficient use of CPU/GPU parallelization, at the cost of memory. If unspecified, batch_size will default to `64`.
- **buffer_size** (*optional*) – Buffer size which is used to shuffle elements during training. Default size is `10000`.
- **seq_length** (*optional*) – The maximum length sentence we want for a single training input in characters. Note that this setting is different than max_line_length, as seq_length simply affects the length of the training examples passed to the neural network to predict the next token. Default size is `100`.

- **embedding_dim** (*optional*) – Vector size for the lookup table used in the neural network Embedding layer that maps the numbers of each character. Default size is `256`.
- **rnn_units** (*optional*) – Positive integer, dimensionality of the output space for LSTM layers. Default size is `256`.
- **dropout_rate** (*optional*) – Float between 0 and 1. Fraction of the units to drop for the linear transformation of the inputs. Using a dropout can help to prevent overfitting by ignoring randomly selected neurons during training. 0.2 (20%) is often used as a good compromise between retaining model accuracy and preventing overfitting. Default is 0.2.
- **rnn_initializer** (*optional*) – Initializer for the kernal weights matrix, used for the linear transformation of the inputs. Default is `glorot_transform`.
- **dp** (*optional*) – If `True`, train model with differential privacy enabled. This setting provides assurances that the models will encode general patterns in data rather than facts about specific training examples. These additional guarantees can usefully strengthen the protections offered for sensitive data and content, at a small loss in model accuracy and synthetic data quality. The differential privacy epsilon and delta values will be printed when training completes. Default is `False`.
- **learning_rate** (*optional*) – The higher the learning rate, the more that each update during training matters. Note: When training with differential privacy enabled, if the updates are noisy (such as when the additive noise is large compared to the clipping threshold), a low learning rate may help with training. Default is `0.01`.
- **dp_noise_multiplier** (*optional*) – The amount of noise sampled and added to gradients during training. Generally, more noise results in better privacy, at the expense of model accuracy. Default is `0.1`.
- **dp_l2_norm_clip** (*optional*) – The maximum Euclidean (L2) norm of each gradient is applied to update model parameters. This hyperparameter bounds the optimizer's sensitivity to individual training points. Default is `3.0`.
- **dp_microbatches** (*optional*) – Each batch of data is split into smaller units called micro-batches. Computational overhead can be reduced by increasing the size of micro-batches to include more than one training example. The number of micro-batches should divide evenly into the overall `batch_size`. Default is `1`.
- **gen_temp** (*optional*) – Controls the randomness of predictions by scaling the logits before applying softmax. Low temperatures result in more predictable text. Higher temperatures result in more surprising text. Experiment to find the best setting. Default is `1.0`.
- **gen_chars** (*optional*) – Maximum number of characters to generate per line. Default is `0` (no limit).
- **gen_lines** (*optional*) – Maximum number of text lines to generate. This function is used by `generate_text` and the optional `line_validator` to make sure that all lines created by the model pass validation. Default is `1000`.
- **predict_batch_size** (*optional*) – How many words to generate in parallel. Higher values may result in increased throughput. The default of `64` should provide reasonable performance for most users.
- **reset_states** (*optional*) – Reset RNN model states between each record created guarantees more consistent record creation over time, at the expense of model accuracy. Default is `True`.
- **save_all_checkpoints** (*optional*) – which can be useful for optimal model selection. Set to `False` to save only the latest checkpoint. Default is `True`.

- **`save_best_model`** (optional). Defaults to `True`. Track the best version of the model (checkpoint) – If `save_all_checkpoints` is disabled, then the saved model will be overwritten by newer ones only if they are better.

**`get_generator_class()`**

This must be implemented by all specific configs. It should return the class that should be used as the Generator for creating records.

**`get_training_callable()`**

This must be implemented by all specific configs. It should return a callable that should be used as the entrypoint for training a model.

**`gpu_check()`**

Optionally do a GPU check and warn if a GPU is not available, if not overridden, do nothing

`gretel_synthetics.config.`**`config_from_model_dir`**(*model_dir: str*) → *BaseConfig*

Factory that will take a known directory of a model and return a class instance for that config. We automatically try and detect the correct BaseConfig sub-class to use based on the saved model params.

If there is no `model_type` param in the saved config, we assume that the model was saved using an earlier version of the package and will instantiate a TensorFlowConfig

## 6.2 Tokenizers

## 6.3 Train

## 6.4 Generate

## 6.5 Batch

## 6.6 Utils

The utils module provides a number of different methods that are useful for training and working with synthetic data.

Some of these methods carry heavy dependencies such as scikit-learn. To prevent adding unnecessary requirements to the main gretel-synthetics package, util dependencies are shipped under an extra called, `utils`. To install the `utils` extra, you may run

```
pip install -U gretel-synthetics[utils]
```

### 6.6.1 Stats

Generates correlation reports between data sets.

`gretel_synthetics.utils.stats.`**`calculate_correlation`**(*df: DataFrame*, *nominal_columns: List[str] | None = None*, *job_count: int = 4*, *opt: bool = False*) → DataFrame

Given a dataframe, calculate a matrix of the correlations between the various rows. We use the calculate_pearsons_r, calculate_correlation_ratio and calculate_theils_u to fill in the matrix values.

**Parameters**

- **df** – The input dataframe.
- **nominal_columns** – Columns to treat as categorical.
- **job_count** – For parallelization of computations.
- **opt** – “optimized.” If opt is True, then go the faster (just not quite as accurate) route of global replace missing with 0.

**Returns**
A dataframe of correlation values.

`gretel_synthetics.utils.stats.`**`calculate_correlation_ratio`**`(`*x*, *y*, *opt*`)`

Calculates the Correlation Ratio for categorical-continuous association. Used in constructing correlation matrix. See http://shakedzy.xyz/dython/modules/nominal/#correlation_ratio.

**Parameters**

- **x** – first input array, categorical.
- **y** – second input array, numeric.
- **opt** – “optimized.” If False, drop missing values if y (the numeric column) is null/nan.

**Returns**
float in the range of [0,1].

`gretel_synthetics.utils.stats.`**`calculate_pearsons_r`**`(`*x*, *y*, *opt*`)` → Tuple[float, float]

Calculate the Pearson correlation coefficient for this pair of rows of our correlation matrix. See https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.pearsonr.html.

**Parameters**

- **x** – first input array.
- **y** – second input array.
- **opt** – “optimized.” If False, drop missing values when either the x or y value is null/nan. If True, we’ve already replaced nan’s with 0’s for entire datafile.

**Returns**
As per scipy, tuple of Pearson’s correlation coefficient and Two-tailed p-value.

`gretel_synthetics.utils.stats.`**`calculate_theils_u`**`(`*x*, *y*`)`

Calculates Theil’s U statistic (Uncertainty coefficient) for categorical-categorical association. Used in constructing correlation matrix. See http://shakedzy.xyz/dython/modules/nominal/#theils_u.

**Parameters**

- **x** – first input array, categorical.
- **y** – second input array, categorical.

**Returns**
float in the range of [0,1].

`gretel_synthetics.utils.stats.`**`compute_distribution_distance`**`(`*d1: dict*, *d2: dict*`)` → float

Calculates the Jensen Shannon distance between two distributions.

**Parameters**

- **d1** – Distribution dict. Values must be a probability vector (all values are floats in [0,1], sum of all values is 1.0).
- **d2** – Another distribution dict.

**Returns**
The distance between the two vectors, range in [0, 1].

**Return type**
float

`gretel_synthetics.utils.stats.`**`compute_pca`**(*df: DataFrame*, *n_components: int = 2*) → DataFrame

Do PCA on a dataframe. See https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html.

**Parameters**

- **`df`** – The dataframe to analyze for principal components.
- **`n_components`** – Number of components to keep.

**Returns**
Dataframe of principal components.

`gretel_synthetics.utils.stats.`**`count_memorized_lines`**(*df1: DataFrame*, *df2: DataFrame*) → int

Checks for overlap between training and synthesized data.

**Parameters**

- **`df1`** – DataFrame of training data.
- **`df2`** – DataFrame of synthetic data.

**Returns**
int, the number of overlapping elements.

`gretel_synthetics.utils.stats.`**`count_missing`**(*df: DataFrame*) → int

Count the number of missing values in a dataframe.

**Parameters**
**`df`** – The dataframe to analyze.

**Returns**
int, the number of missing values.

`gretel_synthetics.utils.stats.`**`get_categorical_field_distribution`**(*field: Series*) → dict

Calculates the normalized distribution of a categorical field.

**Parameters**
**`field`** – A sanitized column extracted from one of the df's.

**Returns**
keys are the unique values in the field, values are percentages (floats in [0, 100]).

**Return type**
dict

`gretel_synthetics.utils.stats.`**`get_numeric_distribution_bins`**(*training: Series*, *synthetic: Series*)

To calculate the distribution distance between two numeric series a la categorical fields we need to bin the data. We want the same bins between both series, based on scrubbed data.

**Parameters**

- **`training`** – The numeric series from the training dataframe.
- **`synthetic`** – The numeric series from the synthetic dataframe.

**Returns**
bin_edges, numpy array of dtype float

`gretel_synthetics.utils.stats.`**`get_numeric_field_distribution`**(*field: Series*, *bins*) → dict

Calculates the normalized distribution of a numeric field cut into bins.

**Parameters**

- **`field`** – A sanitized column extracted from one of the df's.
- **`bins`** – Usually an np.ndarray from get_bins, but can be anything that can be safely passed to pandas.cut.

**Returns**
keys are the unique values in the field, values are floats in [0, 1].

**Return type**
dict

`gretel_synthetics.utils.stats.`**`normalize_dataset`**(*df: DataFrame*) → DataFrame

Prep a dataframe for PCA. Divide the dataframe into numeric and categorical, fill missing values and encode categorical columns by the frequency of each value and standardize all values.

**Parameters**
**`df`** – The dataframe to be subjected to PCA.

**Returns**
The dataframe, normalized.

### 6.6.2 Header Clusters

`gretel_synthetics.utils.header_clusters.`**`cluster`**(*df: DataFrame*, *header_prefix: List[str] | None = None*, *maxsize: int = 20*, *average_record_length_threshold: float = 0*, *method: str = 'single'*, *numeric_cat: List[str] | None = None*, *plot: bool = False*, *isolate_complex_field: bool = True*) → List[List[str]]

Given an input dataframe, extract clusters of similar headers based on a set of heuristics. :param df: The dataframe to cluster headers from. :param header_prefix: List of columns to remove before cluster generation. :param maxsize: The max number of fields in a cluster. :param average_record_length_threshold: Threshold for how long a cluster's records can be.

The default, 0, turns off the average record length (arl) logic. To use arl, use a positive value. Based on our research we recommend setting this value to 250.0.

**Parameters**

- **`method`** – Linkage method used to compute header cluster distances. For more information please refer to the scipy docs, https://docs.scipy.org/doc/scipy/reference/generated/scipy.cluster.hierarchy.linkage.html#scipy-cluster-hierarchy-linkage. # noqa
- **`numeric_cat`** – A list of fields to define as categorical. The header clustering code will automatically define pandas "object" and "category" columns as categorical. The `numeric_cat` parameter may be used to define additional categorical fields that may not automatically get identified as such.
- **`plot`** – Plot header list as a dendogram.
- **`isolate_complex_field`** – Enables isolation of complex fields when clustering.

**Returns**
A list of lists of column names, each column name list being an identified cluster.

## 6.7 Timeseries DGAN

The Timeseries DGAN module contains a PyTorch implementation of the DoppelGANger model, see https://arxiv.org/abs/1909.13403 for a detailed description of the model.

```
import numpy as np
from gretel_synthetics.timeseries_dgan.dgan import DGAN
from gretel_synthetics.timeseries_dgan.config import DGANConfig

attributes = np.random.rand(10000, 3)
features = np.random.rand(10000, 20, 2)

config = DGANConfig(
    max_sequence_len=20,
    sample_len=5,
    batch_size=1000,
    epochs=10
)
model = DGAN(config)

model.train(attributes, features)

synthetic_attributes, synthetic_features = model.generate(1000)
```

**class** `gretel_synthetics.timeseries_dgan.config.`**`DGANConfig`**(*max_sequence_len: int*, *sample_len: int*, *attribute_noise_dim: int = 10*, *feature_noise_dim: int = 10*, *attribute_num_layers: int = 3*, *attribute_num_units: int = 100*, *feature_num_layers: int = 1*, *feature_num_units: int = 100*, *use_attribute_discriminator: bool = True*, *normalization: Normalization = Normalization.ZERO_ONE*, *apply_feature_scaling: bool = True*, *apply_example_scaling: bool = True*, *binary_encoder_cutoff: int = 150*, *forget_bias: bool = False*, *gradient_penalty_coef: float = 10.0*, *attribute_gradient_penalty_coef: float = 10.0*, *attribute_loss_coef: float = 1.0*, *generator_learning_rate: float = 0.001*, *generator_beta1: float = 0.5*, *discriminator_learning_rate: float = 0.001*, *discriminator_beta1: float = 0.5*, *attribute_discriminator_learning_rate: float = 0.001*, *attribute_discriminator_beta1: float = 0.5*, *batch_size: int = 1024*, *epochs: int = 400*, *discriminator_rounds: int = 1*, *generator_rounds: int = 1*, *cuda: bool = True*, *mixed_precision_training: bool = False*)

Config object with parameters for training a DGAN model.

**Parameters**

- **max_sequence_len** – length of time series sequences, variable length sequences are not supported, so all training and generated data will have the same length sequences
- **sample_len** – time series steps to generate from each LSTM cell in DGAN, must be a divisor of max_sequence_len
- **attribute_noise_dim** – length of the GAN noise vectors for attribute generation
- **feature_noise_dim** – length of GAN noise vectors for feature generation
- **attribute_num_layers** – # of layers in the GAN discriminator network
- **attribute_num_units** – # of units per layer in the GAN discriminator network
- **feature_num_layers** – # of LSTM layers in the GAN generator network
- **feature_num_units** – # of units per layer in the GAN generator network
- **use_attribute_discriminator** – use separaste discriminator only on attributes, helps DGAN match attribute distributions, Default: True
- **normalization** – default normalization for continuous variables, used when metadata output is not specified during DGAN initialization
- **apply_feature_scaling** – scale each continuous variable to [0,1] or [-1,1] (based on normalization param) before training and rescale to original range during generation, if False then training data must be within range and DGAN will only generate values in [0,1] or [-1,1], Default: True
- **apply_example_scaling** – compute midpoint and halfrange (equivalent to min/max) for each time series variable and include these as additional attributes that are generated, this provides better support for time series with highly variable ranges, e.g., in network data, a dial-up connection has bandwidth usage in [1kb, 10kb], while a fiber connection is in [100mb, 1gb], Default: True
- **binary_encoder_cutoff** – use binary encoder (instead of one hot encoder) for any column with more than this many unique values. This helps reduce memory consumption for datasets with a lot of unique values.
- **forget_bias** – initialize forget gate bias paramters to 1 in LSTM layers, when True initialization matches tf1 LSTMCell behavior, otherwise default pytorch initialization is used, Default: False
- **gradient_penalty_coef** – coefficient for gradient penalty in Wasserstein loss, Default: 10.0
- **attribute_gradient_penalty_coef** – coefficient for gradient penalty in Wasserstein loss for the attribute discriminator, Default: 10.0
- **attribute_loss_coef** – coefficient for attribute discriminator loss in comparison the standard discriminator on attributes and features, higher values should encourage DGAN to learn attribute distributions, Default: 1.0
- **generator_learning_rate** – learning rate for Adam optimizer
- **generator_beta1** – Adam param for exponential decay of 1st moment
- **discriminator_learning_rate** – learning rate for Adam optimizer
- **discriminator_beta1** – Adam param for exponential decay of 1st moment

- **attribute_discriminator_learning_rate** – learning rate for Adam optimizer
- **attribute_discriminator_beta1** – Adam param for exponential decay of 1st moment
- **batch_size** – # of examples used in batches, for both training and generation
- **epochs** – # of epochs to train model discriminator_rounds: training steps
- **discriminator** (*for the*) – batch
- **generator_rounds** – training steps for the generator in each batch
- **cuda** – use GPU if available
- **mixed_precision_training** – enabling automatic mixed precision while training in order to reduce memory costs, bandwith, and time by identifying the steps that require full precision and using 32-bit floating point for only those steps while using 16-bit floating point everywhere else.

**to_dict**()

Return dictionary representation of DGANConfig.

**Returns**

Dictionary of member variables, usable to initialize a new config object, e.g., *DGANConfig(**config.to_dict())*

class gretel_synthetics.timeseries_dgan.config.**DfStyle**(*value*)

Supported styles for parsing pandas DataFrames.

See *train_dataframe* method in dgan.py for details.

class gretel_synthetics.timeseries_dgan.config.**Normalization**(*value*)

Normalization types for continuous variables.

Determines if a sigmoid (ZERO_ONE) or tanh (MINUSONE_ONE) activation is used for the output layers in the generation network.

class gretel_synthetics.timeseries_dgan.config.**OutputType**(*value*)

Supported variables types.

Determines internal representation of variables and output layers in generation network.

PyTorch implementation of DoppelGANger, from https://arxiv.org/abs/1909.13403

Based on tensorflow 1 code in https://github.com/fjxmlzn/DoppelGANger

DoppelGANger is a generative adversarial network (GAN) model for time series. It supports multi-variate time series (referred to as features) and fixed variables for each time series (attributes). The combination of attribute values and sequence of feature values is 1 example. Once trained, the model can generate novel examples that exhibit the same temporal correlations as seen in the training data. See https://arxiv.org/abs/1909.13403 for additional details on the model.

As a reference for terminology, consider open-high-low-close (OHLC) data from stock markets. Each stock is an example, with fixed attributes such as exchange, sector, country. The features or time series consists of open, high, low, and closing prices for each time interval (daily). After being trained on historical data, the model can generate more hypothetical stocks and price behavior on the training time range.

Sample usage:

```python
import numpy as np
from gretel_synthetics.timeseries_dgan.dgan import DGAN
from gretel_synthetics.timeseries_dgan.config import DGANConfig
```

(continues on next page)

(continued from previous page)

```
attributes = np.random.rand(10000, 3)
features = np.random.rand(10000, 20, 2)

config = DGANConfig(
    max_sequence_len=20,
    sample_len=5,
    batch_size=1000,
    epochs=10
)

model = DGAN(config)

model.train_numpy(attributes=attributes, features=features)

synthetic_attributes, synthetic_features = model.generate_numpy(1000)
```

**class** `gretel_synthetics.timeseries_dgan.dgan.`**`DGAN`**(*config: DGANConfig, attribute_outputs: List[Output] | None = None, feature_outputs: List[Output] | None = None*)

DoppelGANger model.

Interface for training model and generating data based on configuration in an DGANConfig instance.

DoppelGANger uses a specific internal representation for data which is hidden from the user in the public interface. Standard usage of DGAN instances should pass continuous variables as floats in the original space (not normalized), and discrete variables may be strings, integers, or floats. This is the format expected by both train_numpy() and train_dataframe() and the generate_numpy() and generate_dataframe() functions will return data in this same format. In standard usage, the detailed transformation info in attribute_outputs and feature_outputs are not needed, those will be created automatically when a train* function is called with data.

If more control is needed and you want to use the normalized values and one-hot encoding directly, use the _train() and _generate() functions. transformations.py contains internal helper functions for working with the Output metadata instances and converting data to and from the internal representation. To dive even deeper into the model structure, see the torch_modules.py which contains the torch implementations of the networks used in DGAN. As internal details, transformations.py and torch_modules.py are not part of the public interface and may change at any time without notice.

**`__init__`**(*config: DGANConfig, attribute_outputs: List[Output] | None = None, feature_outputs: List[Output] | None = None*)

Create a DoppelGANger model.

**Parameters**

- **`config`** – DGANConfig containing model parameters
- **`attribute_outputs`** – custom metadata for attributes, not needed for standard usage
- **`feature_outputs`** – custom metadata for features, not needed for standard usage

**`generate_dataframe`**(*n: int | None = None, attribute_noise: Tensor | None = None, feature_noise: Tensor | None = None*) → DataFrame

Generate synthetic data from DGAN model.

Once trained, a DGAN model can generate arbitrary amounts of synthetic data by sampling from the noise distributions. Specify either the number of records to generate, or the specific noise vectors to use.

**Parameters**

- `n` – number of examples to generate
- `attribute_noise` – noise vectors to create synthetic data
- `feature_noise` – noise vectors to create synthetic data

**Returns**
pandas DataFrame in same format used in 'train_dataframe' call

`generate_numpy`(*n: int | None = None*, *attribute_noise: Tensor | None = None*, *feature_noise: Tensor | None = None*) → Tuple[ndarray | None, list[numpy.ndarray]]

Generate synthetic data from DGAN model.

Once trained, a DGAN model can generate arbitrary amounts of synthetic data by sampling from the noise distributions. Specify either the number of records to generate, or the specific noise vectors to use.

**Parameters**

- `n` – number of examples to generate
- `attribute_noise` – noise vectors to create synthetic data
- `feature_noise` – noise vectors to create synthetic data

**Returns**
Tuple of attributes and features as numpy arrays.

`classmethod load`(*file_name: str*, ***kwargs*) → *DGAN*

Load DGAN model instance from a file.

**Parameters**

- `file_name` – location to load from
- `kwargs` – additional parameters passed to torch.load, for example, use map_location=torch.device("cpu") to load a model saved for GPU on a machine without cuda

**Returns**
DGAN model instance

`save`(*file_name: str*, ***kwargs*)

Save DGAN model to a file.

**Parameters**

- `file_name` – location to save serialized model
- `kwargs` – additional parameters passed to torch.save

`train_dataframe`(*df: DataFrame*, *attribute_columns: List[str] | None = None*, *feature_columns: List[str] | None = None*, *example_id_column: str | None = None*, *time_column: str | None = None*, *discrete_columns: List[str] | None = None*, *df_style: DfStyle = DfStyle.WIDE*, *progress_callback: Callable[[ProgressInfo], None] | None = None*) → None

Train DGAN model on data in pandas DataFrame.

Training data can be in either "wide" or "long" format. "Wide" format uses one row for each example with 0 or more attribute columns and 1 column per time point in the time series. "Wide" format is restricted to 1 feature variable. "Long" format uses one row per time point, supports multiple feature variables, and uses additional example id to split into examples and time column to sort.

**Parameters**

- **`df`** – DataFrame of training data
- **`attribute_columns`** – list of column names containing attributes, if None, no attribute columns are used. Must be disjoint from the feature columns.
- **`feature_columns`** – list of column names containing features, if None all non-attribute columns are used. Must be disjoint from attribute columns.
- **`example_id_column`** – column name used to split "long" format data frame into multiple examples, if None, data is treated as a single example. This value must be unique from the other column list parameters.
- **`time_column`** – column name used to sort "long" format data frame, if None, data frame order of rows/time points is used. This value must be unique from the other column list parameters.
- **`discrete_columns`** – column names (either attributes or features) to treat as discrete (use one-hot or binary encoding), any string or object columns are automatically treated as discrete
- **`df_style`** – str enum of "wide" or "long" indicating format of the DataFrame

**`train_numpy`**(*features: ndarray | list[numpy.ndarray]*, *feature_types: List[OutputType] | None = None*, *attributes: ndarray | None = None*, *attribute_types: List[OutputType] | None = None*, *progress_callback: Callable[[ProgressInfo], None] | None = None*) → None

Train DGAN model on data in numpy arrays.

Training data is passed in 2 numpy arrays, one for attributes (2d) and one for features (3d), features may be a ragged array with variable length sequences, and then it is a list of numpy arrays. This data should be in the original space and is not transformed. If the data is already transformed into the internal DGAN representation (continuous variable scaled to [0,1] or [-1,1] and discrete variables one-hot or binary encoded), use the internal _train() function instead of train_numpy().

In standard usage, attribute_types and feature_types may be provided on the first call to train() to setup the model structure. If not specified, the default is to assume continuous variables for floats and integers, and discrete for strings. If outputs metadata was specified when the instance was initialized or train() was previously called, then attribute_types and feature_types are not needed.

**Parameters**

- **`features`** – 3-d numpy array of time series features for the training, size is (# of training examples) X max_sequence_len X (# of features) OR list of 2-d numpy arrays with one sequence per numpy array, each numpy array should then have size seq_len X (# of features) where seq_len <= max_sequence_len
- **`feature_types`** (*`Optional`*) – Specification of Discrete or Continuous type for each variable of the features. If None, assume continuous variables for floats and integers, and discrete for strings. Ignored if the model was already built, either by passing *output params at initialization or because train_* was called previously.
- **`attributes`** (*`Optional`*) – 2-d numpy array of attributes for the training examples, size is (# of training examples) X (# of attributes)
- **`attribute_types`** (*`Optional`*) – Specification of Discrete or Continuous type for each variable of the attributes. If None, assume continuous variables for floats and integers, and discrete for strings. Ignored if the model was already built, either by passing *output params at initialization or because train_* was called previously.

`gretel_synthetics.timeseries_dgan.dgan.`**`find_max_consecutive_nans`**(*array: ndarray*) → int

Returns the maximum number of consecutive NaNs in an array.

**Parameters**
`array` – 1-d numpy array of time series per example.

**Returns**
The maximum number of consecutive NaNs in a times series array.

**Return type**
max_cons_nan

`gretel_synthetics.timeseries_dgan.dgan.`**`nan_linear_interpolation`**(*features: list[numpy.ndarray]*, *continuous_features_ind: list[int]*)

Replaces all NaNs via linear interpolation.

Changes numpy arrays in features in place.

**Parameters**

- **`features`** – list of 2-d numpy arrays, each element is a sequence of shape (sequence_len, #features)
- **`continuous_features_ind`** – features to apply nan interpolation to, indexes the 2nd dimension of the sequence arrays of features

`gretel_synthetics.timeseries_dgan.dgan.`**`validation_check`**(*features: list[numpy.ndarray]*, *continuous_features_ind: list[int]*, *invalid_examples_ratio_cutoff: float = 0.5*, *nans_ratio_cutoff: float = 0.1*, *consecutive_nans_max: int = 5*, *consecutive_nans_ratio_cutoff: float = 0.05*) → ndarray

Checks if continuous features of examples are valid.

Returns a 1-d numpy array of booleans with shape (#examples) indicating valid examples. Examples with continuous features fall into 3 categories: good, valid (fixable) and invalid (non-fixable). - “Good” examples have no NaNs. - “Valid” examples have a low percentage of nans and a below a threshold number of consecutive NaNs. - “Invalid” are the rest, and are marked “False” in the returned array. Later on, these are omitted from training. If there are too many, later, we error out.

**Parameters**

- **`features`** – list of 2-d numpy arrays, each element is a sequence of possibly varying length
- **`continuous_features_ind`** – list of indices of continuous features to analyze, indexes the 2nd dimension of the sequence arrays in features
- **`invalid_examples_ratio_cutoff`** – Error out if the invalid examples ratio in the dataset is higher than this value.
- **`nans_ratio_cutoff`** – If the percentage of nans for any continuous feature in an example is greater than this value, the example is invalid.
- **`consecutive_nans_max`** – If the maximum number of consecutive nans in a continuous feature is greater than this number, then that example is invalid.
- **`consecutive_nans_ratio_cutoff`** – If the maximum number of consecutive nans in a continuous feature is greater than this ratio times the length of the example (number samples), then the example is invalid.

**Returns**
1-d numpy array of booleans indicating valid examples with shape (#examples).

**Return type**
valid_examples

## 6.8 ACTGAN

The ACTGAN sub-package contains an alternate implementation of the SDV CTGAN model. It provides some improvement and automation around automatic detection of datetime fields and optional usage of a binary encoder for discrete columns for better memory usage.

Please see the "ACTGAN_Demo" Notebook in the "examples" directory in the repository root.

Wrapper around ACTGAN model.

`class gretel_synthetics.actgan.actgan_wrapper.ACTGAN`(*field_names: List[str] | None = None, field_types: Dict[str, dict] | None = None, field_transformers: Dict[str, BaseTransformer | str] | None = None, auto_transform_datetimes: bool = False, anonymize_fields: Dict[str, str] | None = None, primary_key: str | None = None, constraints: List[Constraint] | List[dict] | None = None, table_metadata: Metadata | dict | None = None, embedding_dim: int = 128, generator_dim: Sequence[int] = (256, 256), discriminator_dim: Sequence[int] = (256, 256), generator_lr: float = 0.0002, generator_decay: float = 1e-06, discriminator_lr: float = 0.0002, discriminator_decay: float = 1e-06, batch_size: int = 500, discriminator_steps: int = 1, binary_encoder_cutoff: int = 500, binary_encoder_nan_handler: str | None = None, cbn_sample_size: int | None = 250000, log_frequency: bool = True, verbose: bool = False, epochs: int = 300, epoch_callback: Callable[[*EpochInfo*], None] | None = None, pac: int = 10, cuda: bool = True, learn_rounding_scheme: bool = True, enforce_min_max_values: bool = True, conditional_vector_type:* ConditionalVectorType *= ConditionalVectorType.SINGLE_DISCRETE, conditional_select_mean_columns: float | None = None, conditional_select_column_prob: float | None = None, reconstruction_loss_coef: float = 1.0, force_conditioning: bool = False*)

**Parameters**

- **`field_names`** – List of names of the fields that need to be modeled and included in the generated output data. Any additional fields found in the data will be ignored and will not be included in the generated output. If `None`, all the fields found in the data are used.
- **`field_types`** – Dictinary specifying the data types and subtypes of the fields that will be modeled. Field types and subtypes combinations must be compatible with the SDV Metadata Schema.

- **`field_transformers`** – Dictinary specifying which transformers to use for each field. Available transformers are:
  - `FloatFormatter`: Uses a `FloatFormatter` for numerical data.
  - `FrequencyEncoder`: Uses a `FrequencyEncoder` without gaussian noise.
  - `FrequencyEncoder_noised`: Uses a `FrequencyEncoder` adding gaussian noise.
  - `OneHotEncoder`: Uses a `OneHotEncoder`.
  - `LabelEncoder`: Uses a `LabelEncoder` without gaussian nose.
  - `LabelEncoder_noised`: Uses a `LabelEncoder` adding gaussian noise.
  - `BinaryEncoder`: Uses a `BinaryEncoder`.
  - `UnixTimestampEncoder`: Uses a `UnixTimestampEncoder`.

  NOTE: Specifically for ACTGAN, some attributes such as `auto_transform_datetimes` will automatically attempt to detect field types and will automatically set the `field_transformers` dictionary at construction time. However, autodetection of `field_types` and `field_transformers` will not be over-written by any concrete values that were provided to this constructor.
- **`auto_transform_datetimes`** – If set, prior to fitting, each column will be checked for being a potential "datetime" type. For each column that is discovered as a "datetime" the *field_types* and *field_transformers* SDV metadata dicts will be automatically updated such that datetimes are transformed to Unix timestamps. NOTE: if fields are already specified in *field_types* or *field_transformers* these fields will be skipped by the auto detector.
- **`anonymize_fields`** – Dict specifying which fields to anonymize and what faker category they belong to.
- **`primary_key`** – Name of the field which is the primary key of the table.
- **`constraints`** – List of Constraint objects or dicts.
- **`table_metadata`** – Table metadata instance or dict representation. If given alongside any other metadata-related arguments, an exception will be raised. If not given at all, it will be built using the other arguments or learned from the data.
- **`embedding_dim`** – Size of the random sample passed to the Generator. Defaults to 128.
- **`generator_dim`** – Size of the output samples for each one of the Residuals. A Residual Layer will be created for each one of the values provided. Defaults to (256, 256).
- **`discriminator_dim`** – Size of the output samples for each one of the Discriminator Layers. A Linear Layer will be created for each one of the values provided. Defaults to (256, 256).
- **`generator_lr`** – Learning rate for the generator. Defaults to 2e-4.
- **`generator_decay`** – Generator weight decay for the Adam Optimizer. Defaults to 1e-6.
- **`discriminator_lr`** – Learning rate for the discriminator. Defaults to 2e-4.
- **`discriminator_decay`** – Discriminator weight decay for the Adam Optimizer. Defaults to 1e-6.
- **`batch_size`** – Number of data samples to process in each step.
- **`discriminator_steps`** – Number of discriminator updates to do for each generator update. From the WGAN paper: https://arxiv.org/abs/1701.07875. WGAN paper default is 5. Default used is 1 to match original CTGAN implementation.

- **binary_encoder_cutoff** – For any given column, the number of unique values that should exist before switching over to binary encoding instead of OHE. This will help reduce memory consumption for datasets with a lot of unique values.
- **binary_encoder_nan_handler** – Binary encoding currently may produce errant NaN values during reverse transformation. By default these NaN's will be left in place, however if this value is set to "mode" then those NaN' will be replaced by a random value that is a known mode for a given column.
- **cbn_sample_size** – Number of rows to sample from each column for identifying clusters for the cluster-based normalizer. This only applies to float columns. If set to `0`, no sampling is done and all values are considered, which may be very slow. Defaults to 250_000.
- **log_frequency** – Whether to use log frequency of categorical levels in conditional sampling. Defaults to `True`.
- **verbose** – Whether to have print statements for progress results. Defaults to `False`.
- **epochs** – Number of training epochs. Defaults to 300.
- **epoch_callback** – An optional function to call after each epoch, the argument will be a `EpochInfo` instance
- **pac** – Number of samples to group together when applying the discriminator. Defaults to 10.
- **cuda** – If `True`, use CUDA. If a `str`, use the indicated device. If `False`, do not use cuda at all. Defaults to `True`.
- **learn_rounding_scheme** – Define rounding scheme for `FloatFormatter`. If `True`, the data returned by `reverse_transform` will be rounded to that place. Defaults to `True`.
- **enforce_min_max_values** – Specify whether or not to clip the data returned by `reverse_transform` of the numerical transformer, `FloatFormatter`, to the min and max values seen during `fit`. Defaults to `True`.
- **conditional_vector_type** – Type of conditional vector to include in input to the generator. Influences how effective and flexible the native conditional generation is. Options include SINGLE_DISCRETE (original CTGAN setup) and ANYWAY. Default is SINGLE_DISCRETE.
- **conditional_select_mean_columns** – Target number of columns to select for conditioning on average during training. Only used for ANYWAY conditioning. Use if typical number of columns to seed on is known. If set, conditional_select_column_prob must be None. Equivalent to setting conditional_select_column_prob to conditional_select_mean_columns / # of columns. Defaults to None.
- **conditional_select_column_prob** – Probability to select any given column to be conditioned on during training. Only used for ANYWAY conditioning. If set, conditional_select_mean_columns must be None. Defaults to None.
- **reconstruction_loss_coef** – Multiplier on reconstruction loss, higher values focus the generator optimization more on accurate conditional vector generation. Defaults to 1.0.
- **force_conditioning** – Directly set the requested conditional generation columns in generated data. Will bypass rejection sampling and be faster, but may reduce quality of the generated data and correlations between conditioned columns and other variables may be weaker. Defaults to False.

**fit**(**args*, ***kwargs*)

Fit the ACTGAN model to the provided data. Prior to fitting, specific auto-detection of data types will be done if the provided `data` is a DataFrame.

`sample`(**args*, ***kwargs*)

Sample rows from this table.

**Parameters**

- **num_rows** (*int*) – Number of rows to sample. This parameter is required.
- **randomize_samples** (*bool*) – Whether or not to use a fixed seed when sampling. Defaults to True.
- **max_tries_per_batch** (*int*) – Number of times to retry sampling until the batch size is met. Defaults to 100.
- **batch_size** (*int or None*) – The batch size to sample. Defaults to *num_rows*, if None.
- **output_file_path** (*str or None*) – The file to periodically write sampled rows to. If None, does not write rows anywhere.
- **conditions** – Deprecated argument. Use the *sample_conditions* method with *sdv.sampling.Condition* objects instead.

**Returns**

Sampled data.

**Return type**

pandas.DataFrame

`sample_remaining_columns`(**args*, ***kwargs*)

Sample rows from this table.

**Parameters**

- **known_columns** (*pandas.DataFrame*) – A pandas.DataFrame with the columns that are already known. The output is a DataFrame such that each row in the output is sampled conditionally on the corresponding row in the input.
- **max_tries_per_batch** (*int*) – Number of times to retry sampling until the batch size is met. Defaults to 100.
- **batch_size** (*int*) – The batch size to use per sampling call.
- **randomize_samples** (*bool*) – Whether or not to use a fixed seed when sampling. Defaults to True.
- **output_file_path** (*str or None*) – The file to periodically write sampled rows to. Defaults to a temporary file, if None.

**Returns**

Sampled data.

**Return type**

pandas.DataFrame

**Raises**

- **ConstraintsNotMetError** – If the conditions are not valid for the given constraints.
- **ValueError** – If any of the following happens: * any of the conditions' columns are not valid. * no rows could be generated.

Complex datastructures for ACTGAN

**class** `gretel_synthetics.actgan.structures.ColumnIdInfo`(*discrete_column_id: 'int'*, *column_id: 'int'*, *value_id: 'np.ndarray'*) `class gretel_synthetics.actgan.structures.`**`ColumnTransformInfo`**(*column_name: 'str'*, *column_type: 'ColumnType'*, *transform: 'BaseTransformer'*, *encodings: 'List[ColumnEncoding]'*)

`class gretel_synthetics.actgan.structures.`**`ColumnType`**(*value*)

An enumeration.

`class gretel_synthetics.actgan.structures.`**`ConditionalVectorType`**(*value*)

An enumeration.

`class gretel_synthetics.actgan.structures.`**`EpochInfo`**(*epoch: int*, *loss_g: float*, *loss_d: float*, *loss_r: float*)

When creating a model such as ACTGAN if the `epoch_callback` attribute is set to a callable, then after each epoch the provided callable will be called with an instance of this class as the only argument.

CHAPTER

# SEVEN

## INDICES AND TABLES

- genindex
- modindex
- search

# PYTHON MODULE INDEX

# INDEX