02_TimeGAN_TF2

September 29, 2021

Time-series Generative Adversarial Network (TimeGAN)

1 Imports & Settings

Adapted from the excellent paper by Jinsung Yoon, Daniel Jarrett, and Mihaela van der Schaar:
Time-series Generative Adversarial Networks,
Neural Information Processing Systems (NeurIPS), 2019.

- Last updated Date: April 24th 2020
- Original code author: Jinsung Yoon (jsyoon0823@gmail.com)

[66]:
```
import warnings
warnings.filterwarnings('ignore')
```

[67]:
```
import pandas as pd
import numpy as np
from sklearn.preprocessing import MinMaxScaler
import tensorflow as tf
from pathlib import Path
from tqdm import tqdm

from tensorflow.keras.models import Sequential, Model
from tensorflow.keras.layers import GRU, Dense, RNN, GRUCell, Input
from tensorflow.keras.losses import BinaryCrossentropy, MeanSquaredError
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import TensorBoard
from tensorflow.keras.utils import plot_model

import matplotlib.pyplot as plt
import seaborn as sns
```

[68]:
```
gpu_devices = tf.config.experimental.list_physical_devices('GPU')
if gpu_devices:
    print('Using GPU')
    tf.config.experimental.set_memory_growth(gpu_devices[0], True)
else:
    print('Using CPU')
```

Using CPU

[69]:
```
sns.set_style('white')
```

2 Experiment Path

[5]:
```
results_path = Path('time_gan')
if not results_path.exists():
    results_path.mkdir()
```

[6]:
```
experiment = 0
```

[7]:
```
log_dir = results_path / f'experiment_{experiment:02}'
if not log_dir.exists():
    log_dir.mkdir(parents=True)
```

[8]:
```
hdf_store = results_path / 'TimeSeriesGAN.h5'
```

3 Prepare Data

3.1 Parameters

[9]:
```
seq_len = 24
n_seq = 6
batch_size = 128
```

[10]:
```
tickers = ['BA', 'CAT', 'DIS', 'GE', 'IBM', 'KO']
```

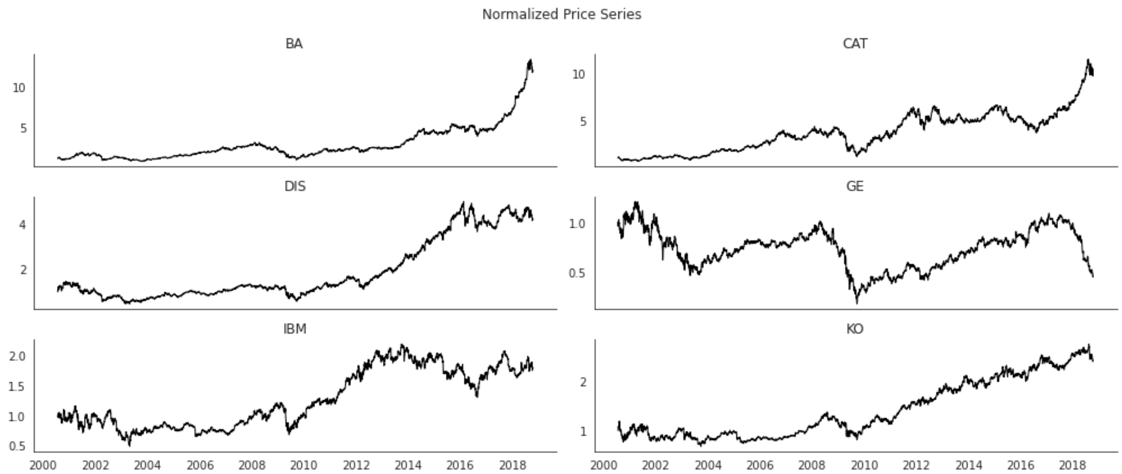
[11]:
```
def select_data():
    df = (pd.read_hdf('../data/assets.h5', 'quandl/wiki/prices')
          .adj_close
          .unstack('ticker')
          .loc['2000':, tickers]
          .dropna())
    df.to_hdf(hdf_store, 'data/real')
```

[12]:
```
select_data()
```

3.2 Plot Series

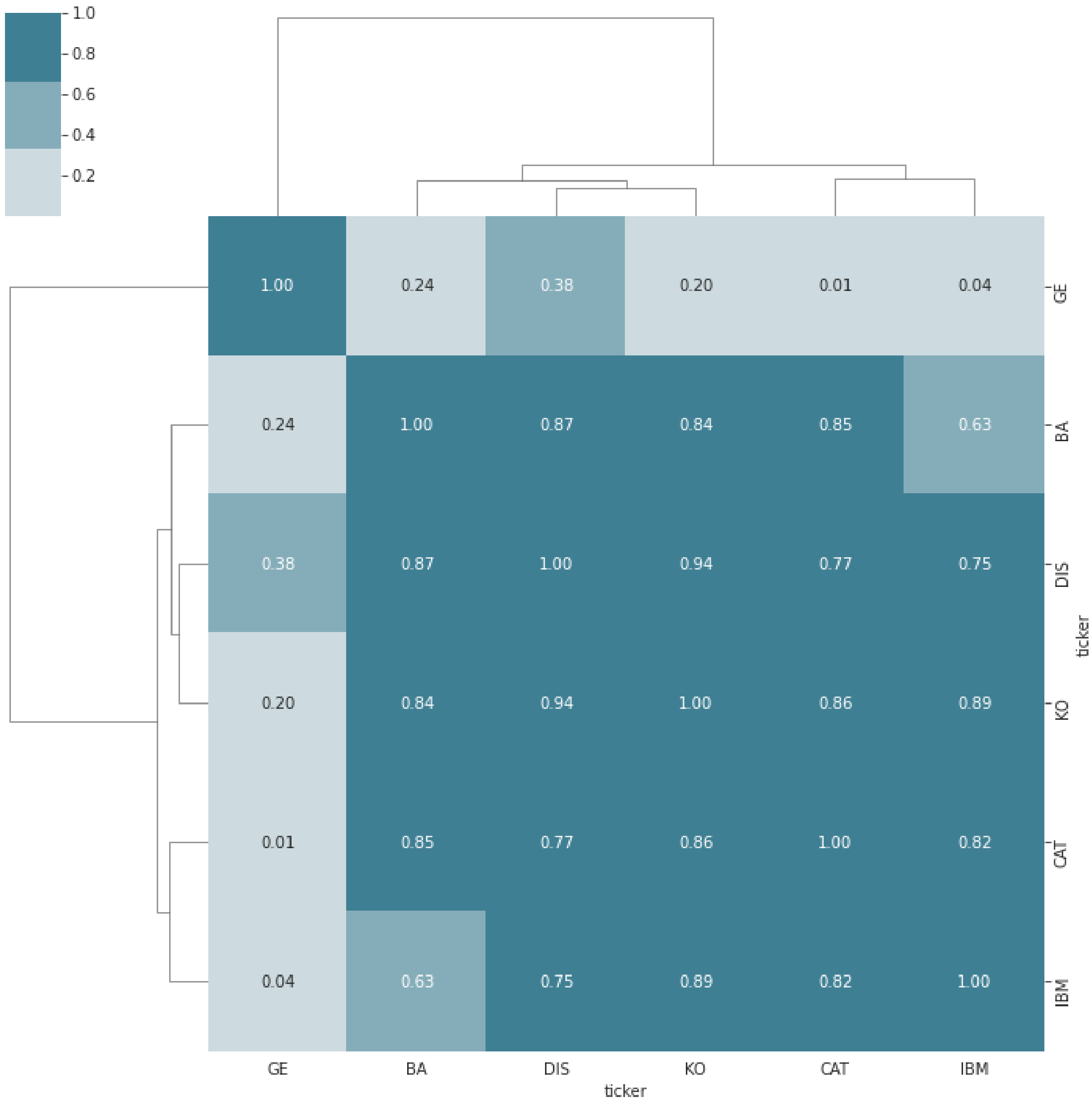
[13]:
```
df = pd.read_hdf(hdf_store, 'data/real')
axes = df.div(df.iloc[0]).plot(subplots=True,
                               figsize=(14, 6),
                               layout=(3, 2),
                               title=tickers,
                               legend=False,
```

```
                                          rot=0,
                                          lw=1,
                                          color='k')
for ax in axes.flatten():
    ax.set_xlabel('')

plt.suptitle('Normalized Price Series')
plt.gcf().tight_layout()
sns.despine();
```



3.3 Correlation

[14]:
```
sns.clustermap(df.corr(),
               annot=True,
               fmt='.2f',
               cmap=sns.diverging_palette(h_neg=20,
                                          h_pos=220), center=0);
```



3.4 Normalize Data

[15]:
```
scaler = MinMaxScaler()
scaled_data = scaler.fit_transform(df).astype(np.float32)
```

3.5 Create rolling window sequences

[16]:
```
data = []
for i in range(len(df) - seq_len):
    data.append(scaled_data[i:i + seq_len])

n_windows = len(data)
```

3.6 Create tf.data.Dataset

[17]:
```
real_series = (tf.data.Dataset
               .from_tensor_slices(data)
               .shuffle(buffer_size=n_windows)
               .batch(batch_size))
real_series_iter = iter(real_series.repeat())
```

3.7 Set up random series generator

[18]:
```
def make_random_data():
    while True:
        yield np.random.uniform(low=0, high=1, size=(seq_len, n_seq))
```

We use the Python generator to feed a `tf.data.Dataset` that continues to call the random number generator as long as necessary and produces the desired batch size.

[19]:
```
random_series = iter(tf.data.Dataset
                     .from_generator(make_random_data, output_types=tf.float32)
                     .batch(batch_size)
                     .repeat())
```

4 TimeGAN Components

The design of the TimeGAN components follows the author's sample code.

4.1 Network Parameters

[20]:
```
hidden_dim = 24
num_layers = 3
```

4.2 Set up logger

[21]:
```
writer = tf.summary.create_file_writer(log_dir.as_posix())
```

4.3 Input place holders

[22]:
```
X = Input(shape=[seq_len, n_seq], name='RealData')
Z = Input(shape=[seq_len, n_seq], name='RandomData')
```

4.4 RNN block generator

We keep it very simple and use a very similar architecture for all four components. For a real-world application, they should be tailored to the data.

[23]:
```
def make_rnn(n_layers, hidden_units, output_units, name):
    return Sequential([GRU(units=hidden_units,
                           return_sequences=True,
                           name=f'GRU_{i + 1}') for i in range(n_layers)] +
                      [Dense(units=output_units,
                             activation='sigmoid',
                             name='OUT')], name=name)
```

4.5 Embedder & Recovery

[24]:
```
embedder = make_rnn(n_layers=3,
                    hidden_units=hidden_dim,
                    output_units=hidden_dim,
                    name='Embedder')
recovery = make_rnn(n_layers=3,
                    hidden_units=hidden_dim,
                    output_units=n_seq,
                    name='Recovery')
```

4.6 Generator & Discriminator

[25]:
```
generator = make_rnn(n_layers=3,
                     hidden_units=hidden_dim,
                     output_units=hidden_dim,
                     name='Generator')
discriminator = make_rnn(n_layers=3,
                         hidden_units=hidden_dim,
                         output_units=1,
                         name='Discriminator')
supervisor = make_rnn(n_layers=2,
                      hidden_units=hidden_dim,
                      output_units=hidden_dim,
                      name='Supervisor')
```

5 TimeGAN Training

5.1 Settings

[26]:
```
train_steps = 10000
gamma = 1
```

5.2 Generic Loss Functions

[27]:
```
mse = MeanSquaredError()
bce = BinaryCrossentropy()
```

6 Phase 1: Autoencoder Training

6.1 Architecture

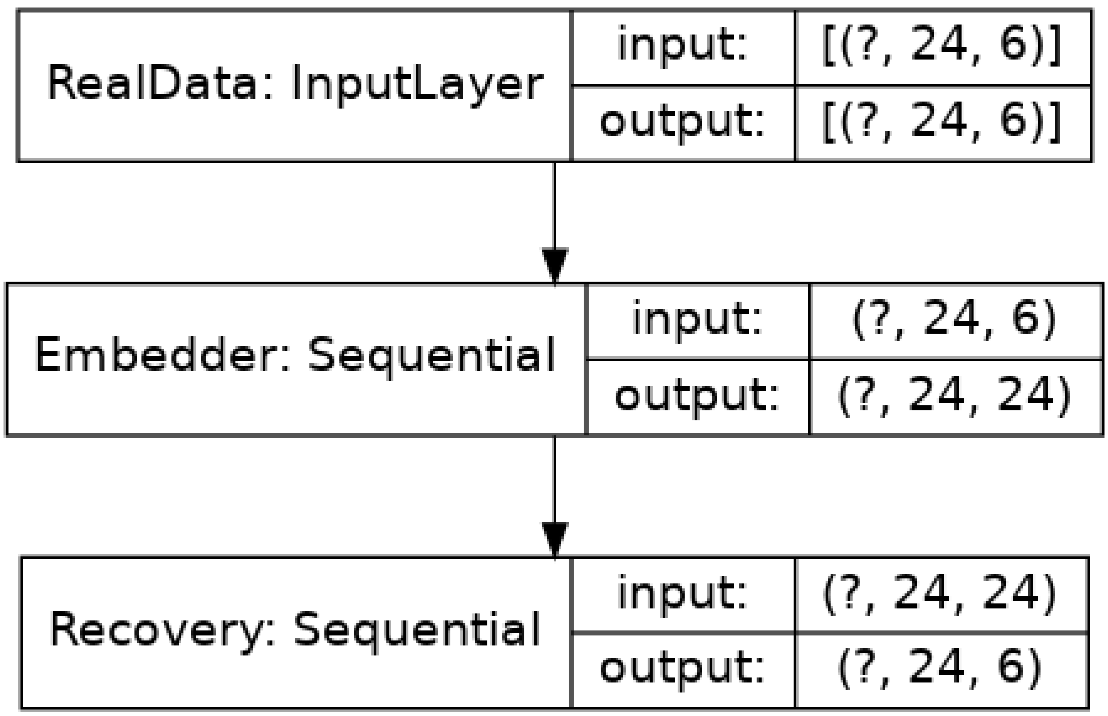
[28]:
```
H = embedder(X)
X_tilde = recovery(H)

autoencoder = Model(inputs=X,
                    outputs=X_tilde,
                    name='Autoencoder')
```

[29]:
```
autoencoder.summary()
```

```
Model: "Autoencoder"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
RealData (InputLayer)        [(None, 24, 6)]           0
_________________________________________________________________
Embedder (Sequential)        (None, 24, 24)            10104
_________________________________________________________________
Recovery (Sequential)        (None, 24, 6)             10950
=================================================================
Total params: 21,054
Trainable params: 21,054
Non-trainable params: 0
_________________________________________________________________
```

[30]:
```
plot_model(autoencoder,
           to_file=(results_path / 'autoencoder.png').as_posix(),
           show_shapes=True)
```

[30]:



6.2 Autoencoder Optimizer

[31]:
```
autoencoder_optimizer = Adam()
```

6.3 Autoencoder Training Step

[32]:
```
@tf.function
def train_autoencoder_init(x):
    with tf.GradientTape() as tape:
        x_tilde = autoencoder(x)
        embedding_loss_t0 = mse(x, x_tilde)
        e_loss_0 = 10 * tf.sqrt(embedding_loss_t0)

    var_list = embedder.trainable_variables + recovery.trainable_variables
    gradients = tape.gradient(e_loss_0, var_list)
    autoencoder_optimizer.apply_gradients(zip(gradients, var_list))
    return tf.sqrt(embedding_loss_t0)
```

6.4 Autoencoder Training Loop

[33]:
```
for step in tqdm(range(train_steps)):
    X_ = next(real_series_iter)
    step_e_loss_t0 = train_autoencoder_init(X_)
    with writer.as_default():
        tf.summary.scalar('Loss Autoencoder Init', step_e_loss_t0, step=step)
```

```
100%|      | 10000/10000 [05:20<00:00, 31.23it/s]
```

6.5 Persist model

[34]:
```
# autoencoder.save(log_dir / 'autoencoder')
```

7 Phase 2: Supervised training

7.1 Define Optimizer

[35]:
```
supervisor_optimizer = Adam()
```

7.2 Train Step

[36]:
```
@tf.function
def train_supervisor(x):
    with tf.GradientTape() as tape:
        h = embedder(x)
        h_hat_supervised = supervisor(h)
        g_loss_s = mse(h[:, 1:, :], h_hat_supervised[:, :-1, :])

    var_list = supervisor.trainable_variables
    gradients = tape.gradient(g_loss_s, var_list)
    supervisor_optimizer.apply_gradients(zip(gradients, var_list))
    return g_loss_s
```

7.3 Training Loop

[37]:
```
for step in tqdm(range(train_steps)):
    X_ = next(real_series_iter)
    step_g_loss_s = train_supervisor(X_)
    with writer.as_default():
        tf.summary.scalar('Loss Generator Supervised Init', step_g_loss_s,␣
 ↪step=step)
```

```
100%|      | 10000/10000 [03:04<00:00, 54.31it/s]
```

7.4 Persist Model

[38]:
```
# supervisor.save(log_dir / 'supervisor')
```

8 Joint Training

8.1 Generator

8.1.1 Adversarial Architecture - Supervised

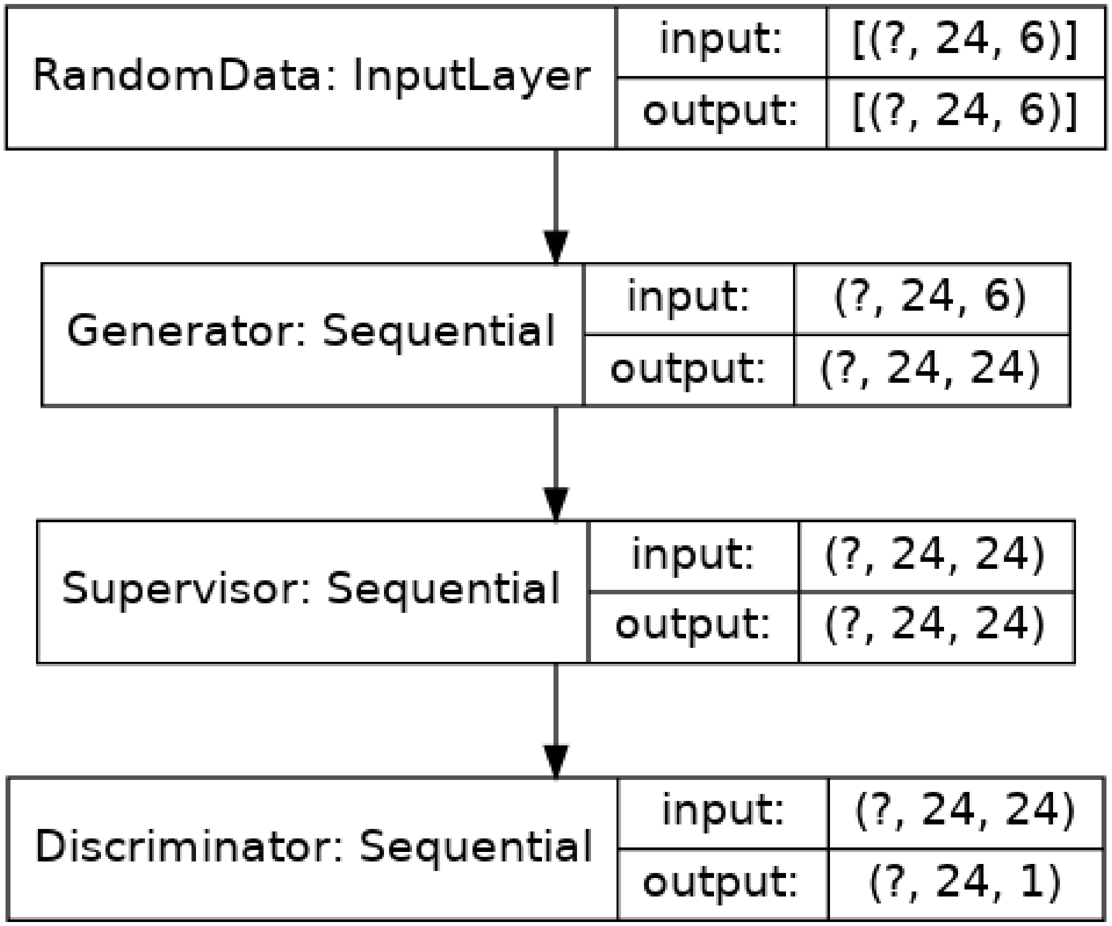
[39]:
```
E_hat = generator(Z)
H_hat = supervisor(E_hat)
Y_fake = discriminator(H_hat)

adversarial_supervised = Model(inputs=Z,
                               outputs=Y_fake,
                               name='AdversarialNetSupervised')
```

[40]:
```
adversarial_supervised.summary()
```

```
Model: "AdversarialNetSupervised"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
RandomData (InputLayer)      [(None, 24, 6)]           0
_________________________________________________________________
Generator (Sequential)       (None, 24, 24)            10104
_________________________________________________________________
Supervisor (Sequential)      (None, 24, 24)            7800
_________________________________________________________________
Discriminator (Sequential)   (None, 24, 1)             10825
=================================================================
Total params: 28,729
Trainable params: 28,729
Non-trainable params: 0
_________________________________________________________________
```

[41]:
```
plot_model(adversarial_supervised, show_shapes=True)
```

[41]:



8.1.2 Adversarial Architecture in Latent Space

[42]:
```
Y_fake_e = discriminator(E_hat)

adversarial_emb = Model(inputs=Z,
                        outputs=Y_fake_e,
                        name='AdversarialNet')
```

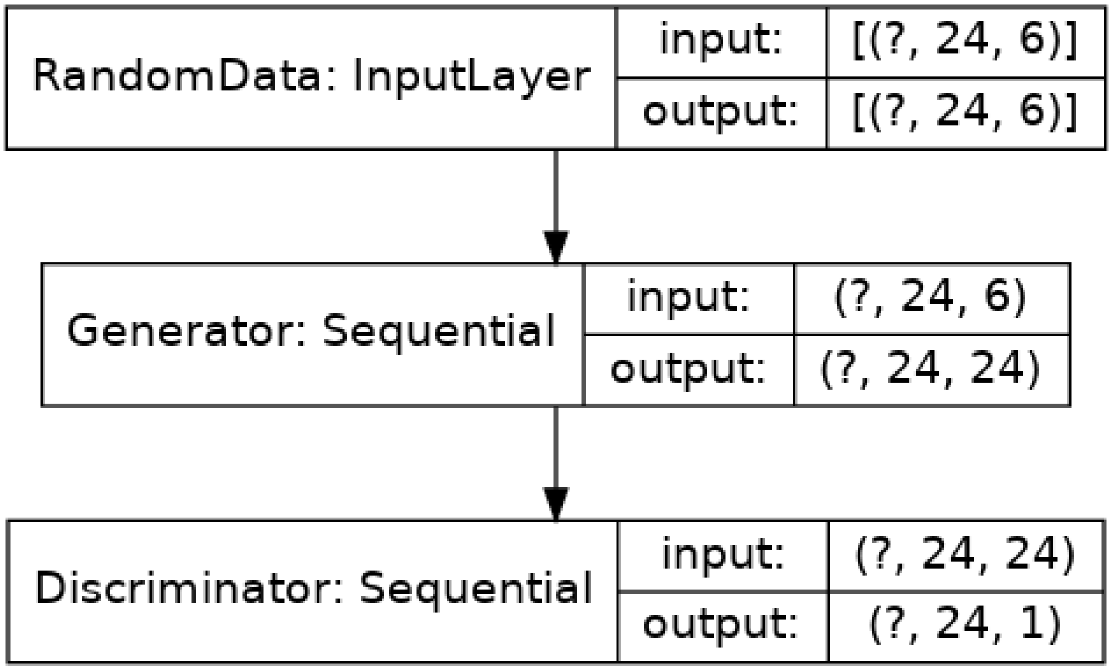
[43]:
```
adversarial_emb.summary()
```

```
Model: "AdversarialNet"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
RandomData (InputLayer)      [(None, 24, 6)]           0
_________________________________________________________________
Generator (Sequential)       (None, 24, 24)            10104
_________________________________________________________________
```

```
Discriminator (Sequential)   (None, 24, 1)             10825
=================================================================
Total params: 20,929
Trainable params: 20,929
Non-trainable params: 0
_________________________________________________________________
```

[44]:
```
plot_model(adversarial_emb, show_shapes=True)
```

[44]:



8.1.3 Mean & Variance Loss

[45]:
```
X_hat = recovery(H_hat)
synthetic_data = Model(inputs=Z,
                       outputs=X_hat,
                       name='SyntheticData')
```

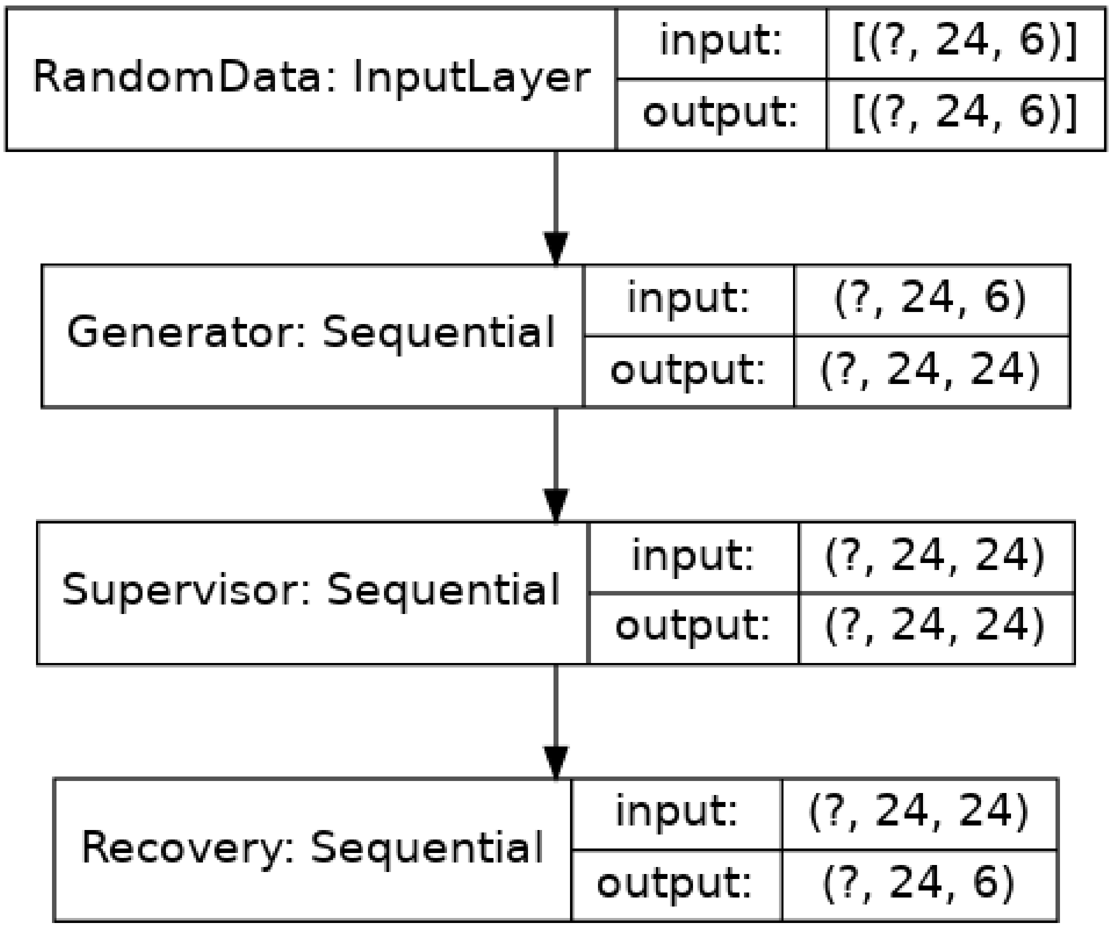
[46]:
```
synthetic_data.summary()
```

```
Model: "SyntheticData"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
RandomData (InputLayer)      [(None, 24, 6)]           0
_________________________________________________________________
Generator (Sequential)       (None, 24, 24)            10104
```

```
_________________________________________________________________
Supervisor (Sequential)      (None, 24, 24)            7800
_________________________________________________________________
Recovery (Sequential)        (None, 24, 6)             10950
=================================================================
Total params: 28,854
Trainable params: 28,854
Non-trainable params: 0
_________________________________________________________________
```

[47]:
```
plot_model(synthetic_data, show_shapes=True)
```

[47]:



[48]:
```
def get_generator_moment_loss(y_true, y_pred):
    y_true_mean, y_true_var = tf.nn.moments(x=y_true, axes=[0])
    y_pred_mean, y_pred_var = tf.nn.moments(x=y_pred, axes=[0])
    g_loss_mean = tf.reduce_mean(tf.abs(y_true_mean - y_pred_mean))
    g_loss_var = tf.reduce_mean(tf.abs(tf.sqrt(y_true_var + 1e-6) - tf.
 ↪sqrt(y_pred_var + 1e-6)))
```

```
    return g_loss_mean + g_loss_var
```

8.2 Discriminator

8.2.1 Architecture: Real Data

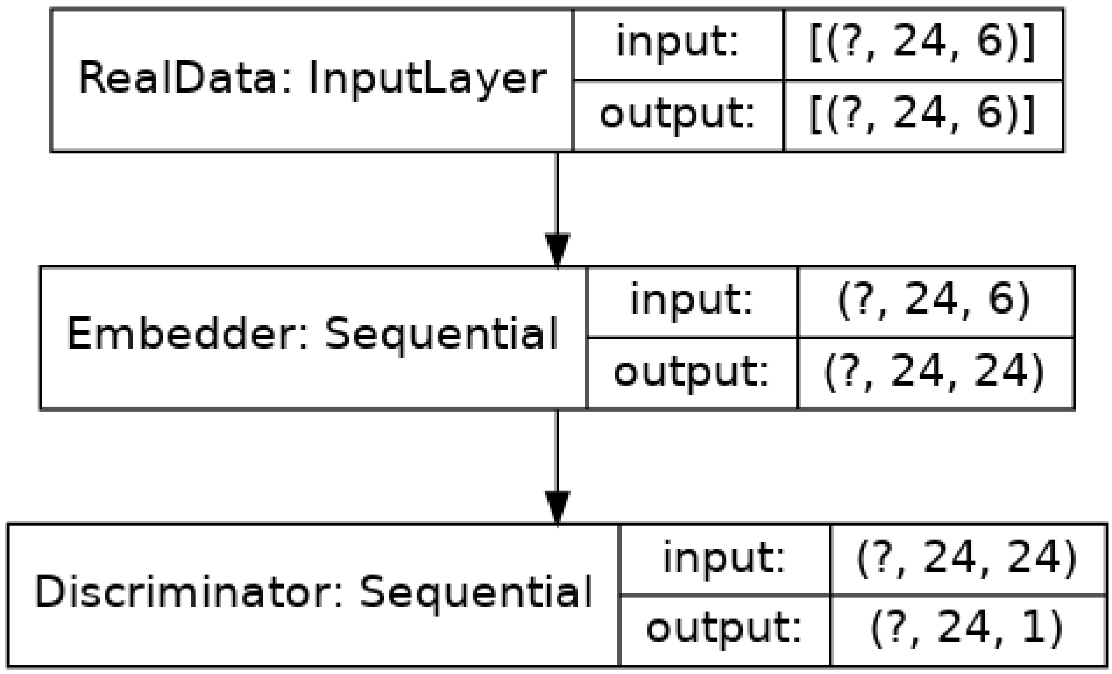
[49]:
```
Y_real = discriminator(H)
discriminator_model = Model(inputs=X,
                            outputs=Y_real,
                            name='DiscriminatorReal')
```

[50]:
```
discriminator_model.summary()
```

```
Model: "DiscriminatorReal"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
RealData (InputLayer)        [(None, 24, 6)]           0
_________________________________________________________________
Embedder (Sequential)        (None, 24, 24)            10104
_________________________________________________________________
Discriminator (Sequential)   (None, 24, 1)             10825
=================================================================
Total params: 20,929
Trainable params: 20,929
Non-trainable params: 0
_________________________________________________________________
```

[51]:
```
plot_model(discriminator_model, show_shapes=True)
```

[51]:



8.3 Optimizers

[52]:
```
generator_optimizer = Adam()
discriminator_optimizer = Adam()
embedding_optimizer = Adam()
```

8.4 Generator Train Step

[53]:
```
@tf.function
def train_generator(x, z):
    with tf.GradientTape() as tape:
        y_fake = adversarial_supervised(z)
        generator_loss_unsupervised = bce(y_true=tf.ones_like(y_fake),
                                          y_pred=y_fake)

        y_fake_e = adversarial_emb(z)
        generator_loss_unsupervised_e = bce(y_true=tf.ones_like(y_fake_e),
                                            y_pred=y_fake_e)
        h = embedder(x)
        h_hat_supervised = supervisor(h)
        generator_loss_supervised = mse(h[:, 1:, :], h_hat_supervised[:, 1:, :])

        x_hat = synthetic_data(z)
        generator_moment_loss = get_generator_moment_loss(x, x_hat)
```

```
        generator_loss = (generator_loss_unsupervised +
                          generator_loss_unsupervised_e +
                          100 * tf.sqrt(generator_loss_supervised) +
                          100 * generator_moment_loss)

    var_list = generator.trainable_variables + supervisor.trainable_variables
    gradients = tape.gradient(generator_loss, var_list)
    generator_optimizer.apply_gradients(zip(gradients, var_list))
    return generator_loss_unsupervised, generator_loss_supervised,␣
↪generator_moment_loss
```

8.5 Embedding Train Step

[54]:
```
@tf.function
def train_embedder(x):
    with tf.GradientTape() as tape:
        h = embedder(x)
        h_hat_supervised = supervisor(h)
        generator_loss_supervised = mse(h[:, 1:, :], h_hat_supervised[:, 1:, :])

        x_tilde = autoencoder(x)
        embedding_loss_t0 = mse(x, x_tilde)
        e_loss = 10 * tf.sqrt(embedding_loss_t0) + 0.1 *␣
↪generator_loss_supervised

    var_list = embedder.trainable_variables + recovery.trainable_variables
    gradients = tape.gradient(e_loss, var_list)
    embedding_optimizer.apply_gradients(zip(gradients, var_list))
    return tf.sqrt(embedding_loss_t0)
```

8.6 Discriminator Train Step

[55]:
```
@tf.function
def get_discriminator_loss(x, z):
    y_real = discriminator_model(x)
    discriminator_loss_real = bce(y_true=tf.ones_like(y_real),
                                  y_pred=y_real)

    y_fake = adversarial_supervised(z)
    discriminator_loss_fake = bce(y_true=tf.zeros_like(y_fake),
                                  y_pred=y_fake)

    y_fake_e = adversarial_emb(z)
    discriminator_loss_fake_e = bce(y_true=tf.zeros_like(y_fake_e),
                                    y_pred=y_fake_e)
```

```
    return (discriminator_loss_real +
            discriminator_loss_fake +
            gamma * discriminator_loss_fake_e)
```

[56]:
```
@tf.function
def train_discriminator(x, z):
    with tf.GradientTape() as tape:
        discriminator_loss = get_discriminator_loss(x, z)

    var_list = discriminator.trainable_variables
    gradients = tape.gradient(discriminator_loss, var_list)
    discriminator_optimizer.apply_gradients(zip(gradients, var_list))
    return discriminator_loss
```

8.7 Training Loop

[57]:
```
step_g_loss_u = step_g_loss_s = step_g_loss_v = step_e_loss_t0 = step_d_loss = 0
for step in range(train_steps):
    # Train generator (twice as often as discriminator)
    for kk in range(2):
        X_ = next(real_series_iter)
        Z_ = next(random_series)

        # Train generator
        step_g_loss_u, step_g_loss_s, step_g_loss_v = train_generator(X_, Z_)
        # Train embedder
        step_e_loss_t0 = train_embedder(X_)

    X_ = next(real_series_iter)
    Z_ = next(random_series)
    step_d_loss = get_discriminator_loss(X_, Z_)
    if step_d_loss > 0.15:
        step_d_loss = train_discriminator(X_, Z_)

    if step % 1000 == 0:
        print(f'{step:6,.0f} | d_loss: {step_d_loss:6.4f} | g_loss_u:␣
↪{step_g_loss_u:6.4f} | '
              f'g_loss_s: {step_g_loss_s:6.4f} | g_loss_v: {step_g_loss_v:6.4f}␣
↪| e_loss_t0: {step_e_loss_t0:6.4f}')

    with writer.as_default():
        tf.summary.scalar('G Loss S', step_g_loss_s, step=step)
        tf.summary.scalar('G Loss U', step_g_loss_u, step=step)
        tf.summary.scalar('G Loss V', step_g_loss_v, step=step)
        tf.summary.scalar('E Loss T0', step_e_loss_t0, step=step)
        tf.summary.scalar('D Loss', step_d_loss, step=step)
```

```
     0 | d_loss: 2.1271 | g_loss_u: 0.6151 | g_loss_s: 0.0003 | g_loss_v: 0.3934
| e_loss_t0: 0.0329
 1,000 | d_loss: 0.5845 | g_loss_u: 2.8059 | g_loss_s: 0.0004 | g_loss_v: 0.0681
| e_loss_t0: 0.0094
 2,000 | d_loss: 1.3358 | g_loss_u: 1.3916 | g_loss_s: 0.0001 | g_loss_v: 0.0490
| e_loss_t0: 0.0079
 3,000 | d_loss: 1.2601 | g_loss_u: 1.0959 | g_loss_s: 0.0002 | g_loss_v: 0.0492
| e_loss_t0: 0.0072
 4,000 | d_loss: 1.0254 | g_loss_u: 1.4230 | g_loss_s: 0.0001 | g_loss_v: 0.0293
| e_loss_t0: 0.0067
 5,000 | d_loss: 0.9681 | g_loss_u: 1.5861 | g_loss_s: 0.0001 | g_loss_v: 0.0175
| e_loss_t0: 0.0058
 6,000 | d_loss: 1.4235 | g_loss_u: 1.3382 | g_loss_s: 0.0001 | g_loss_v: 0.0303
| e_loss_t0: 0.0052
 7,000 | d_loss: 1.3194 | g_loss_u: 1.4435 | g_loss_s: 0.0001 | g_loss_v: 0.0450
| e_loss_t0: 0.0049
 8,000 | d_loss: 1.3595 | g_loss_u: 1.4879 | g_loss_s: 0.0001 | g_loss_v: 0.0275
| e_loss_t0: 0.0048
 9,000 | d_loss: 1.4149 | g_loss_u: 1.3156 | g_loss_s: 0.0001 | g_loss_v: 0.0771
| e_loss_t0: 0.0043
```

8.8 Persist Synthetic Data Generator

[58]:
```
synthetic_data.save(log_dir / 'synthetic_data')
```

```
WARNING:tensorflow:From /opt/conda/envs/ml4t-dl/lib/python3.8/site-
packages/tensorflow/python/training/tracking/tracking.py:111:
Model.state_updates (from tensorflow.python.keras.engine.training) is deprecated
and will be removed in a future version.
Instructions for updating:
This property should not be used in TensorFlow 2.0, as updates are applied
automatically.
WARNING:tensorflow:From /opt/conda/envs/ml4t-dl/lib/python3.8/site-
packages/tensorflow/python/training/tracking/tracking.py:111: Layer.updates
(from tensorflow.python.keras.engine.base_layer) is deprecated and will be
removed in a future version.
Instructions for updating:
This property should not be used in TensorFlow 2.0, as updates are applied
automatically.
INFO:tensorflow:Assets written to: time_gan/experiment_00/synthetic_data/assets
```

9 Generate Synthetic Data

[59]:
```
generated_data = []
for i in range(int(n_windows / batch_size)):
    Z_ = next(random_series)
    d = synthetic_data(Z_)
```

```
    generated_data.append(d)
```

[60]:
```
len(generated_data)
```

[60]: 35

[61]:
```
generated_data = np.array(np.vstack(generated_data))
generated_data.shape
```

[61]: (4480, 24, 6)

[62]:
```
np.save(log_dir / 'generated_data.npy', generated_data)
```

9.1 Rescale

[63]:
```
generated_data = (scaler.inverse_transform(generated_data
                                            .reshape(-1, n_seq))
                  .reshape(-1, seq_len, n_seq))
generated_data.shape
```

[63]: (4480, 24, 6)

9.2 Persist Data

[64]:
```
with pd.HDFStore(hdf_store) as store:
    store.put('data/synthetic', pd.DataFrame(generated_data.reshape(-1, n_seq),
                                              columns=tickers))
```

9.3 Plot sample Series

[65]:
```
fig, axes = plt.subplots(nrows=3, ncols=2, figsize=(14, 7))
axes = axes.flatten()

index = list(range(1, 25))
synthetic = generated_data[np.random.randint(n_windows)]

idx = np.random.randint(len(df) - seq_len)
real = df.iloc[idx: idx + seq_len]

for j, ticker in enumerate(tickers):
    (pd.DataFrame({'Real': real.iloc[:, j].values,
                   'Synthetic': synthetic[:, j]})
     .plot(ax=axes[j],
           title=ticker,
           secondary_y='Synthetic', style=['-', '--'],
           lw=1))
```

```
sns.despine()
fig.tight_layout()
```