21
Generative Adversarial Networks for Synthetic Time-Series Data

Following the coverage of autoencoders in the previous chapter, this chapter introduces a second unsupervised deep learning technique: **generative adversarial networks** (**GANs**). As with autoencoders, GANs complement the methods for dimensionality reduction and clustering introduced in *Chapter 13*, *Data-Driven Risk Factors and Asset Allocation with Unsupervised Learning*.

GANs were invented by Goodfellow et al. in 2014. Yann LeCun has called GANs the "most exciting idea in AI in the last ten years." A **GAN** trains two neural networks, called the **generator** and **discriminator**, in a competitive setting. The generator aims to produce samples that the discriminator is unable to distinguish from a given class of training data. The result is a generative model capable of producing synthetic samples representative of a certain target distribution but artificially and, thus, inexpensively created.

GANs have produced an avalanche of research and successful applications in many domains. While originally applied to images, Esteban, Hyland, and Rätsch (2017) applied GANs to the medical domain to generate **synthetic time-series data**. Experiments with financial data ensued (Koshiyama, Firoozye, and Treleaven 2019; Wiese et al. 2019; Zhou et al. 2018; Fu et al. 2019) to explore whether GANs can generate data that simulates alternative asset price trajectories to train supervised or reinforcement algorithms, or to backtest trading strategies. We will replicate the Time-Series GAN presented at the 2019 NeurIPS by Yoon, Jarrett, and van der Schaar (2019) to illustrate the approach and demonstrate the results.

More specifically, in this chapter you will learn about the following:

- How GANs work, why they are useful, and how they can be applied to trading
- Designing and training GANs using TensorFlow 2
- Generating synthetic financial data to expand the inputs available for training ML models and backtesting

You can find the code samples for this chapter and links to additional resources in the corresponding directory of the GitHub repository. The notebooks include color versions of the images.

Creating synthetic data with GANs

This book mostly focuses on supervised learning algorithms that receive input data and predict an outcome, which we can compare to the ground truth to evaluate their performance. Such algorithms are also called **discriminative models** because they learn to differentiate between different output values.

GANs are an instance of **generative models** like the variational autoencoder we encountered in the previous chapter. As described there, a generative model takes a training set with samples drawn from some distribution $p_{\mathbf{data}}$ and learns to represent an estimate $p_{\mathbf{model}}$ of that data-generating distribution.

As mentioned in the introduction, GANs are considered one of the most exciting recent machine learning innovations because they appear capable of generating high-quality samples that faithfully mimic a range of input data. This is very attractive given the absence or high cost of labeled data required for supervised learning.

GANs have triggered a wave of research that initially focused on the generation of surprisingly realistic images. More recently, GAN instances have emerged that produce synthetic time series with significant potential for trading since the limited availability of historical market data is a key driver of the risk of backtest overfitting.

In this section, we explain in more detail how generative models and adversarial training work and review various GAN architectures. In the next section, we will demonstrate how to design and train a GAN using TensorFlow 2. In the last section, we will describe how to adapt a GAN so that it creates synthetic time-series data.

Comparing generative and discriminative models

Discriminative models learn how to differentiate among outcomes y, given input data X. In other words, they learn the probability of the outcome given the data: $p(y \mid X)$. Generative models, on the other hand, learn the joint distribution of inputs and outcome $p(y, X)$. While generative models can be used as discriminative models using Bayes' rule to compute which class is most likely (see *Chapter 10, Bayesian ML – Dynamic Sharpe Ratios and Pairs Trading*), it often seems preferable to solve the

prediction problem directly rather than by solving the more general generative challenge first (Ng and Jordan 2002).

GANs have a generative objective: they produce complex outputs, such as realistic images, given simple inputs that can even be random numbers. They achieve this by modeling a probability distribution over the possible outputs. This probability distribution can have many dimensions, for example, one for each pixel in an image, each character or token in a document, or each value in a time series. As a result, the model can generate outputs that are very likely representative of the class of outputs.

Richard Feynman's quote "**What I cannot create, I do not understand**" emphasizes that modeling generative distributions is an important step towards more general AI and resembles human learning, which succeeds using much fewer samples.

Generative models have several **use cases** beyond their ability to generate additional samples from a given distribution. For example, they can be incorporated into model-based **reinforcement learning** (**RL**) algorithms (see the next chapter). Generative models can also be applied to time-series data to simulate alternative past or possible future trajectories that can be used for planning in RL or supervised learning more generally, including for the design of trading algorithms. Other use cases include semi-supervised learning where GANs facilitate feature matching to assign missing labels with much fewer training samples than current approaches.

Adversarial training – a zero-sum game of trickery

The key innovation of GANs is a new way of learning the data-generating probability distribution. The algorithm sets up a competitive, or adversarial game between two neural networks called the **generator** and the **discriminator**.

The generator's goal is to convert random noise input into fake instances of a specific class of objects, such as images of faces or stock price time series. The discriminator, in turn, aims to differentiate the generator's deceptive output from a set of training data containing true samples of the target objects. The overall GAN objective is for both networks to get better at their respective tasks so that the generator produces outputs that a machine can no longer distinguish from the originals (at which point we don't need the discriminator, which is no longer necessary, and can discard it).

Figure 21.1 illustrates adversarial training using a generic GAN architecture designed to generate images. We assume the generator uses a deep

CNN architecture (such as the VGG16 example from *Chapter 18*, *CNNs for Financial Time Series and Satellite Images*) that is reversed just like the decoder part of the convolutional autoencoder we discussed in the previous chapter. The generator receives an input image with random pixel values and produces a *fake* output image that is passed on to the discriminator network, which uses a mirrored CNN architecture. The discriminator network also receives *real* samples that represent the target distribution and predicts the probability that the input is *real*, as opposed to *fake*. Learning takes place by backpropagating the gradients of the discriminator and generator losses to the respective network's parameters:

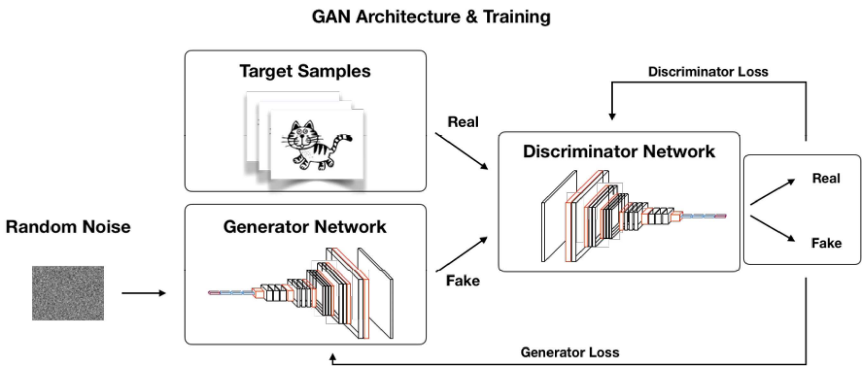

Figure 21.1: GAN architecture

The recent GAN Lab is a great interactive tool inspired by TensorFlow Playground, which allows the user to design GANs and visualize various aspects of the learning process and performance over time (see resource links on GitHub).

The rapid evolution of the GAN architecture zoo

Since the publication of the paper by Goodfellow et al. in 2014, GANs have attracted an enormous amount of interest and triggered a corresponding flurry of research.

The bulk of this work has refined the original architecture to adapt it to different domains and tasks, as well as expanding it to include additional information and create conditional GANs. Additional research has focused on improving methods for the challenging training process, which requires achieving a stable game-theoretic equilibrium between two networks, each of which can be tricky to train on its own.

The GAN landscape has become more diverse than we can cover here; see Creswell et al. (2018) and Pan et al. (2019) for recent surveys, and Odena (2019) for a list of open questions.

Deep convolutional GANs for representation learning

Deep convolutional GANs (DCGANs) were motivated by the successful application of CNNs to supervised learning for grid-like data (Radford, Metz, and Chintala 2016). The architecture pioneered the use of GANs for unsupervised learning by developing a feature extractor based on adversarial training. It is also easier to train and generates higher-quality images. It is now considered a baseline implementation, with numerous open source examples available (see references on GitHub).

A DCGAN network takes uniformly distributed random numbers as input and outputs a color image with a resolution of 64×64 pixels. As the input changes incrementally, so do the generated images. The network consists of standard CNN components, including deconvolutional layers that reverse convolutional layers as in the convolutional autoencoder example in the previous chapter, or fully connected layers.

The authors experimented exhaustively and made several recommendations, such as the use of batch normalization and ReLU activations in both networks. We will explore a TensorFlow implementation later in this chapter.

Conditional GANs for image-to-image translation

Conditional GANs (cGANs) introduce additional label information into the training process, resulting in better quality and some control over the output.

cGANs alter the baseline architecture displayed previously in *Figure 21.1* by adding a third input to the discriminator that contains class labels. These labels, for example, could convey gender or hair color information when generating images.

Extensions include the **generative adversarial what-where network (GAWWN**; Reed et al. 2016), which uses bounding box information not only to generate synthetic images but also to place objects at a given location.

GAN applications to images and time-series data

Alongside a large variety of extensions and modifications of the original architecture, numerous applications to images, as well as sequential data like speech and music, have emerged. Image applications are particularly diverse, ranging from image blending and super-resolution to video generation and human pose identification. Furthermore, GANs have been used to improve supervised learning performance.

We will look at a few salient examples and then take a closer look at applications to time-series data that may become particularly relevant to al-

gorithmic trading and investment. See Alqahtani, Kavakli-Thorne, and Kumar (2019) for a recent survey and GitHub references for additional resources.

CycleGAN – unpaired image-to-image translation

Supervised image-to-image translation aims to learn a mapping between aligned input and output images. CycleGAN solves this task when paired images are not available and transforms images from one domain to match another.

Popular examples include the synthetic "painting" of horses as zebras and vice versa. It also includes the transfer of styles, by generating a realistic sample of an impressionistic print from an arbitrary landscape photo (Zhu et al. 2018).

StackGAN – text-to-photo image synthesis

One of the earlier applications of GANs to domain-transfer is the generation of images based on text. **Stacked GAN**, often shortened to **StackGAN**, uses a sentence as input and generates multiple images that match the description.

The architecture operates in two stages, where the first stage yields a low-resolution sketch of shape and colors, and the second stage enhances the result to a high-resolution image with photorealistic details (Zhang et al. 2017).

SRGAN – photorealistic single image super-resolution

Super-resolution aims at producing higher-resolution photorealistic images from low-resolution input. GANs applied to this task have deep CNN architectures that use batch normalization, ReLU, and skip connection as encountered in ResNet (see *Chapter 18*, *CNNs for Financial Time Series and Satellite Images*) to produce impressive results that are already finding commercial applications (Ledig et al. 2017).

Synthetic time series with recurrent conditional GANs

Recurrent GANs (RGANs) and **recurrent conditional GANs (RCGANs)** are two model architectures that aim to synthesize realistic real-valued multivariate time series (Esteban, Hyland, and Rätsch 2017). The authors target applications in the medical domain, but the approach could be highly valuable to overcome the limitations of historical market data.

RGANs rely on **recurrent neural networks (RNNs)** for the generator and the discriminator. RCGANs add auxiliary information in the spirit of

cGANs (see the previous *Conditional GANs for image-to-image translation* section).

The authors succeed in generating visually and quantitatively compelling realistic samples. Furthermore, they evaluate the quality of the synthetic data, including synthetic labels, by using it to train a model with only minor degradation of the predictive performance on a real test set. The authors also demonstrate the successful application of RCGANs to an early warning system using a medical dataset of 17,000 patients from an intensive care unit. Hence, the authors illustrate that RCGANs are capable of generating time-series data useful for supervised training. We will apply this approach to financial market data this chapter in the *TimeGAN – adversarial training for synthetic financial data* section.

How to build a GAN using TensorFlow 2

To illustrate the implementation of a GAN using Python, we will use the DCGAN example discussed earlier in this section to synthesize images from the Fashion-MNIST dataset that we first encountered in *Chapter 13, Data-Driven Risk Factors and Asset Allocation with Unsupervised Learning*.

See the notebook `deep_convolutional_generative_adversarial_network` for implementation details and references.

Building the generator network

Both generator and discriminator use a deep CNN architecture along the lines illustrated in *Figure 20.1*, but with fewer layers. The generator uses a fully connected input layer, followed by three convolutional layers, as defined in the following `build_generator()` function, which returns a Keras model instance:

```
def build_generator():
    return Sequential([Dense(7 * 7 * 256,
                             use_bias=False,
                             input_shape=(100,),
                             name='IN'),
                       BatchNormalization(name='BN1'),
                       LeakyReLU(name='RELU1'),
                       Reshape((7, 7, 256), name='SHAPE1'),
                       Conv2DTranspose(128, (5, 5),
                                       strides=(1, 1),
                                       padding='same',
                                       use_bias=False,
```

```
                    name='CONV1'),
        BatchNormalization(name='BN2'),
        LeakyReLU(name='RELU2'),
        Conv2DTranspose(64, (5, 5),
                    strides=(2, 2),
                    padding='same',
                    use_bias=False,
                    name='CONV2'),
        BatchNormalization(name='BN3'),
        LeakyReLU(name='RELU3'),
        Conv2DTranspose(1, (5, 5),
                    strides=(2, 2),
                    padding='same',
                    use_bias=False,
                    activation='tanh',
                    name='CONV3')],
    name='Generator')
```

The generator accepts 100 one-dimensional random values as input, and it produces images that are 28 pixels wide and high and, thus, contain 784 data points.

A call to the `.summary()` method of the model returned by this function shows that this network has over 2.3 million parameters (see the notebook for details, including a visualization of the generator output prior to training).

Creating the discriminator network

The discriminator network uses two convolutional layers that translate the input received from the generator into a single output value. The model has around 212,000 parameters:

```
def build_discriminator():
    return Sequential([Conv2D(64, (5, 5),
                              strides=(2, 2),
                              padding='same',
                              input_shape=[28, 28, 1],
                              name='CONV1'),
                       LeakyReLU(name='RELU1'),
                       Dropout(0.3, name='DO1'),
                       Conv2D(128, (5, 5),
                              strides=(2, 2),
                              padding='same',
                              name='CONV2'),
                       LeakyReLU(name='RELU2'),
                       Dropout(0.3, name='DO2'),
                       Flatten(name='FLAT'),
```

```
        Dense(1, name='OUT')],
    name='Discriminator')
```

Figure 21.2 depicts how the random input flows from the generator to the discriminator, as well as the input and output shapes of the various network components:

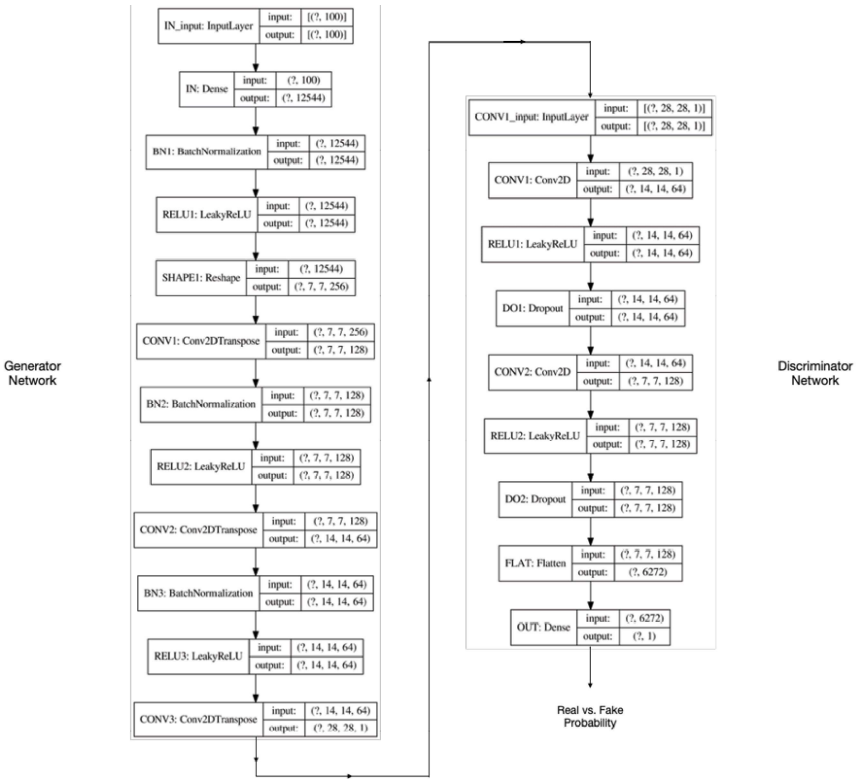

Figure 21.2: DCGAN TensorFlow 2 model architecture

Setting up the adversarial training process

Now that we have built the generator and the discriminator models, we will design and execute the adversarial training process. To this end, we will define the following:

- The loss functions for both models that reflect their competitive interaction
- A single training step that runs the backpropagation algorithm
- The training loop that repeats the training step until the model performance meets our expectations

Defining the generator and discriminator loss functions

The generator loss reflects the discriminator's decision regarding the fake input. It will be low if the discriminator mistakes an image produced by

the generator for a real image, and high otherwise; we will define the interaction between both models when we create the training step.

The generator loss is measured by the binary cross-entropy loss function as follows:

```
cross_entropy = BinaryCrossentropy(from_logits=True)
def generator_loss(fake_output):
    return cross_entropy(tf.ones_like(fake_output), fake_output)
```

The discriminator receives both real and fake images as input. It computes a loss for each and attempts to minimize the sum with the goal of accurately recognizing both types of inputs:

```
def discriminator_loss(true_output, fake_output):
    true_loss = cross_entropy(tf.ones_like(true_output), true_output)
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
    return true_loss + fake_loss
```

To train both models, we assign each an Adam optimizer with a learning rate lower than the default:

```
gen_optimizer = Adam(1e-4)
dis_optimizer = Adam(1e-4)
```

The core – designing the training step

Each training step implements one round of stochastic gradient descent using the Adam optimizer. It consists of five steps:

1. Providing the minibatch inputs to each model
2. Getting the models' outputs for the current weights
3. Computing the loss given the models' objective and output
4. Obtaining the gradients for the loss with respect to each model's weights
5. Applying the gradients according to the optimizer's algorithm

The function `train_step()` carries out these five steps. We use the `@tf.function` decorator to speed up execution by compiling it to a TensorFlow operation rather than relying on eager execution (see the TensorFlow documentation for details):

```
@tf.function
def train_step(images):
    # generate the random input for the generator
```

```
    noise = tf.random.normal([BATCH_SIZE, noise_dim])
    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        # get the generator output
        generated_img = generator(noise, training=True)
        # collect discriminator decisions regarding real and fake input
        true_output = discriminator(images, training=True)
        fake_output = discriminator(generated_img, training=True)
        # compute the loss for each model
        gen_loss = generator_loss(fake_output)
        disc_loss = discriminator_loss(true_output, fake_output)
    # compute the gradients for each loss with respect to the model variables
    grad_generator = gen_tape.gradient(gen_loss,
                                       generator.trainable_variables)
    grad_discriminator = disc_tape.gradient(disc_loss,
                                            discriminator.trainable_variables)
    # apply the gradient to complete the backpropagation step
    gen_optimizer.apply_gradients(zip(grad_generator,
                                      generator.trainable_variables))
    dis_optimizer.apply_gradients(zip(grad_discriminator,
                                      discriminator.trainable_variables))
```

Putting it together – the training loop

The training loop is very straightforward to implement once we have the training step properly defined. It consists of a simple `for` loop, and during each iteration, we pass a new batch of real images to the training step. We also will sample some synthetic images and occasionally save the model weights.

Note that we track progress using the `tqdm` package, which shows the percentage complete during training:

```
def train(dataset, epochs, save_every=10):
    for epoch in tqdm(range(epochs)):
        for img_batch in dataset:
            train_step(img_batch)
        # produce images for the GIF as we go
        display.clear_output(wait=True)
        generate_and_save_images(generator, epoch + 1, seed)
        # Save the model every 10 EPOCHS
        if (epoch + 1) % save_every == 0:
            checkpoint.save(file_prefix=checkpoint_prefix)
        # Generator after final epoch
    display.clear_output(wait=True)
    generate_and_save_images(generator, epochs, seed)
train(train_set, EPOCHS)
```

Evaluating the results

After 100 epochs that only take a few minutes, the synthetic images created from random noise clearly begin to resemble the originals, as you can see in *Figure 21.3* (see the notebook for the best visual quality):

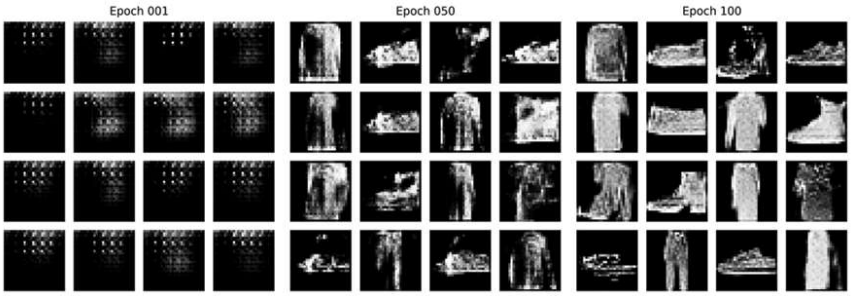

Figure 21.3: A sample of synthetic Fashion-MNIST images

The notebook also creates a dynamic GIF image that visualizes how the quality of the synthetic images improves during training.

Now that we understand how to build and train a GAN using TensorFlow 2, we will move on to a more complex example that produces synthetic time series from stock price data.

TimeGAN for synthetic financial data

Generating synthetic time-series data poses specific challenges above and beyond those encountered when designing GANs for images. In addition to the distribution over variables at any given point, such as pixel values or the prices of numerous stocks, a generative model for time-series data should also learn the temporal dynamics that shape how one sequence of observations follows another. (Refer also to the discussion in *Chapter 9*, *Time-Series Models for Volatility Forecasts and Statistical Arbitrage*).

Very recent and promising research by Yoon, Jarrett, and van der Schaar, presented at NeurIPS in December 2019, introduces a novel **time-series generative adversarial network** (**TimeGAN**) framework that aims to account for temporal correlations by combining supervised and unsupervised training. The model learns a time-series embedding space while optimizing both supervised and adversarial objectives, which encourage it to adhere to the dynamics observed while sampling from historical data during training. The authors test the model on various time series, including historical stock prices, and find that the quality of the synthetic data significantly outperforms that of available alternatives.

In this section, we will outline how this sophisticated model works, highlight key implementation steps that build on the previous DCGAN exam-

ple, and show how to evaluate the quality of the resulting time series. Please see the paper for additional information.

Learning to generate data across features and time

A successful generative model for time-series data needs to capture both the cross-sectional distribution of features at each point in time and the longitudinal relationships among these features over time. Expressed in the image context we just discussed, the model needs to learn not only what a realistic image looks like, but also how one image evolves from the previous as in a video.

Combining adversarial and supervised training

As mentioned in the first section, prior attempts at generating time-series data, like RGANs and RCGANs, relied on RNNs (see *Chapter 19*, *RNNs for Multivariate Time Series and Sentiment Analysis*) in the roles of generator and discriminator. TimeGAN explicitly incorporates the autoregressive nature of time series by combining the **unsupervised adversarial loss** on both real and synthetic sequences familiar from the DCGAN example with a **stepwise supervised loss** with respect to the original data. The goal is to reward the model for learning the **distribution over transitions** from one point in time to the next that are present in the historical data.

Furthermore, TimeGAN includes an embedding network that maps the time-series features to a lower-dimensional latent space to reduce the complexity of the adversarial space. The motivation is to capture the drivers of temporal dynamics that often have lower dimensionality. (Refer also to the discussions of manifold learning in *Chapter 13*, *Data-Driven Risk Factors and Asset Allocation with Unsupervised Learning* and nonlinear dimensionality reduction in *Chapter 20*, *Autoencoders for Conditional Risk Factors and Asset Pricing*).

A key element of the TimeGAN architecture is that both the generator and the embedding (or autoencoder) network are responsible for minimizing the supervised loss that measures how well the model learns the dynamic relationship. As a result, the model learns a latent space conditioned on facilitating the generator's task to faithfully reproduce the temporal relationships observed in the historical data. In addition to time-series data, the model can also process static data that does not change or changes less frequently over time.

The four components of the TimeGAN architecture

The TimeGAN architecture combines an adversarial network with an autoencoder and thus has four network components, as depicted in *Figure 21.4*:

1. **Autoencoder**: embedding and recovery networks
2. **Adversarial network**: sequence generator and sequence discriminator components

The authors emphasize the **joint training** of the autoencoder and the adversarial networks by means of **three different loss functions**. The **reconstruction loss** optimizes the autoencoder, the **unsupervised loss** trains the adversarial net, and the **supervised loss** enforces the temporal dynamics. As a result of this key insight, the TimeGAN simultaneously learns to encode features, generate representations, and iterate across time. More specifically, the embedding network creates the latent space, the adversarial network operates within this space, and supervised loss synchronizes the latent dynamics of both real and synthetic data.

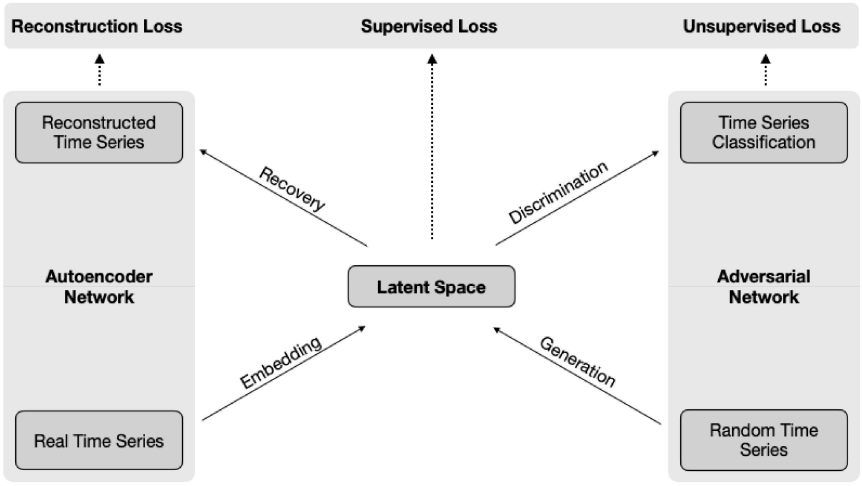

Figure 21.4: The components of the TimeGAN architecture

The **embedding and recovery** components of the autoencoder map the feature space into the latent space and vice versa. This facilitates the learning of the temporal dynamics by the adversarial network, which learns in a lower-dimensional space. The authors implement the embedding and recovery network using a stacked RNN and a feedforward network. However, these choices can be flexibly adapted to the task at hand as long as they are autoregressive and respect the temporal order of the data.

The **generator and the discriminator** elements of the adversarial network differ from the DCGAN not only because they operate on sequential data but also because the synthetic features are generated in the latent space that the model learns simultaneously. The authors chose an RNN as

the generator and a bidirectional RNN with a feedforward output layer for the discriminator.

Joint training of an autoencoder and adversarial network

The three loss functions displayed in *Figure 21.4* drive the joint optimization of the network elements just described while training on real and randomly generated time series. In more detail, they aim to accomplish the following:

- The **reconstruction loss** is familiar from our discussion of autoencoders in *Chapter 20*, *Autoencoders for Conditional Risk Factors and Asset Pricing*; it compares how well the reconstruction of the encoded data resembles the original.
- The **unsupervised loss** reflects the competitive interaction between the generator and the discriminator described in the DCGAN example; while the generator aims to minimize the probability that the discriminator classifies its output as fake, the discriminator aims to optimize the correct classification or real and fake inputs.
- The **supervised loss** captures how well the generator approximates the actual next time step in latent space when receiving encoded real data for the prior sequence.

Training takes place in **three phases**:

1. Training the autoencoder on real time series to optimize reconstruction
2. Optimizing the supervised loss using real time series to capture the temporal dynamics of the historical data
3. Jointly training the four components while minimizing all three loss functions

TimeGAN includes several **hyperparameters** used to weigh the components of composite loss functions; however, the authors find the network to be less sensitive to these settings than one might expect given the notorious difficulties of GAN training. In fact, they **do not discover significant challenges during training** and suggest that the embedding task serves to regularize adversarial learning because it reduces its dimensionality while the supervised loss constrains the stepwise dynamics of the generator.

We now turn to the TimeGAN implementation using TensorFlow 2; see the paper for an in-depth explanation of the math and methodology of the approach.

Implementing TimeGAN using TensorFlow 2

In this section, we will implement the TimeGAN architecture just described. The authors provide sample code using TensorFlow 1 that we will port to TensorFlow 2. Building and training TimeGAN requires several steps:

1. Selecting and preparing real and random time series inputs
2. Creating the key TimeGAN model components
3. Defining the various loss functions and training steps used during the three training phases
4. Running the training loops and logging the results
5. Generating synthetic time series and evaluating the results

We'll walk through the key items for each of these steps; please refer to the notebook `TimeGAN_TF2` for the code examples in this section (unless otherwise noted), as well as additional implementation details.

Preparing the real and random input series

The authors demonstrate the applicability of TimeGAN to financial data using 15 years of daily Google stock prices downloaded from Yahoo Finance with six features, namely open, high, low, close and adjusted close price, and volume. We'll instead use close to 20 years of adjusted close prices for six different tickers because it introduces somewhat higher variability. We will follow the original paper in targeting synthetic series with 24 time steps.

Among the stocks with the longest history in the Quandl Wiki dataset are those displayed in normalized format, that is, starting at 1.0, in *Figure 21.5*. We retrieve the adjusted close from 2000-2017 and obtain over 4,000 observations. The correlation coefficient among the series ranges from 0.01 for GE and CAT to 0.94 for DIS and KO.

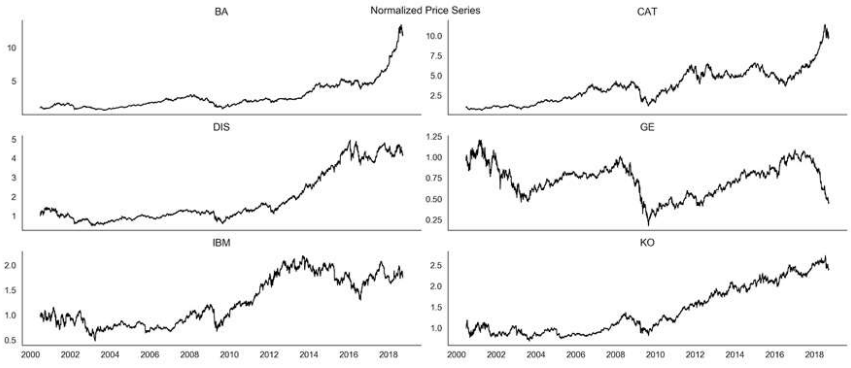

Figure 21.5: The TimeGAN input—six real stock prices series

We scale each series to the range [0, 1] using scikit-learn's `MinMaxScaler` class, which we will later use to rescale the synthetic data:

```
df = pd.read_hdf(hdf_store, 'data/real')
scaler = MinMaxScaler()
scaled_data = scaler.fit_transform(df).astype(np.float32)
```

In the next step, we create rolling windows containing overlapping sequences of 24 consecutive data points for the six series:

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

We then create a `tf.data.Dataset` instance from the list of `NumPy` arrays, ensure the data gets shuffled while training, and set a batch size of 128:

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

We also need a random time-series generator that produces simulated data with 24 observations on the six series for as long as the training continues.

To this end, we will create a generator that draws the requisite data uniform at random and feeds the result into a second `tf.data.Datase` t instance. We set this dataset to produce batches of the desired size and to repeat the process for as long as necessary:

```
def make_random_data():
    while True:
        yield np.random.uniform(low=0, high=1, size=(seq_len, n_seq))
random_series = iter(tf.data.Dataset
                     .from_generator(make_random_data,
                                     output_types=tf.float32)
                     .batch(batch_size)
                     .repeat())
```

We'll now proceed to define and instantiate the TimeGAN model components.

Creating the TimeGAN model components

We'll now create the two autoencoder components and the two adversarial network elements, as well as the supervisor that encourages the generator to learn the temporal dynamic of the historical price series.

We will follow the authors' sample code in creating RNNs with three hidden layers, each with 24 GRU units, except for the supervisor, which uses only two hidden layers. The following `make_rnn` function automates the network creation:

```
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)
```

The `autoencoder` consists of the `embedder` and the recovery networks that we instantiate here:

```
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')
```

We then create the generator, the discriminator, and the supervisor like so:

```
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')
```

We also define two generic loss functions, namely `MeanSquaredError` and `BinaryCrossEntropy`, which we will use later to create the various

specific loss functions during the three phases:

```
mse = MeanSquaredError()
bce = BinaryCrossentropy()
```

Now it's time to start the training process.

Training phase 1 – autoencoder with real data

The autoencoder integrates the embedder and the recovery functions, as we saw in the previous chapter:

```
H = embedder(X)
X_tilde = recovery(H)
autoencoder = Model(inputs=X,
                    outputs=X_tilde,
                    name='Autoencoder')
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
=================================================================
Trainable params: 21,054
```

It has 21,054 parameters. We will now instantiate the optimizer for this training phase and define the training step. It follows the pattern introduced with the DCGAN example, using `tf.GradientTape` to record the operations that generate the reconstruction loss. This allows us to rely on the automatic differentiation engine to obtain the gradients with respect to the trainable embedder and recovery network weights that drive `backpropagation`:

```
autoencoder_optimizer = Adam()
@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)
```

The reconstruction loss simply compares the autoencoder outputs with its inputs. We train for 10,000 steps in a little over one minute using this training loop that records the step loss for monitoring with TensorBoard:

```
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)
```

Training phase 2 – supervised learning with real data

We already created the supervisor model so we just need to instantiate the optimizer and define the train step as follows:

```
supervisor_optimizer = Adam()
@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
```

In this case, the loss compares the output of the supervisor with the next timestep for the embedded sequence so that it learns the temporal dynamics of the historical price sequences; the training loop works similarly to the autoencoder example in the previous chapter.

Training phase 3 – joint training with real and random data

The joint training involves all four network components, as well as the supervisor. It uses multiple loss functions and combinations of the base components to achieve the simultaneous learning of latent space embeddings, transition dynamics, and synthetic data generation.

We will highlight a few salient examples; please see the notebook for the full implementation that includes some repetitive steps that we will omit here.

To ensure that the generator faithfully reproduces the time series, TimeGAN includes a moment loss that penalizes when the mean and variance of the synthetic data deviate from the real version:

```
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
```

The end-to-end model that produces synthetic data involves the generator, supervisor, and recovery components. It is defined as follows and has close to 30,000 trainable parameters:

```
E_hat = generator(Z)
H_hat = supervisor(E_hat)
X_hat = recovery(H_hat)
synthetic_data = Model(inputs=Z,
                       outputs=X_hat,
                       name='SyntheticData')
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
=================================================================
Trainable params: 28,854
```

The joint training involves three optimizers for the autoencoder, the generator, and the discriminator:

```
generator_optimizer = Adam()
discriminator_optimizer = Adam()
embedding_optimizer = Adam()
```

The train step for the generator illustrates the use of four loss functions and corresponding combinations of network components to achieve the desired learning outlined at the beginning of this section:

```
@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)
```

Finally, the joint training loop pulls the various training steps together and builds on the learning from phase 1 and 2 to train the TimeGAN components on both real and random data. We run the loop for 10,000 iterations in under 40 minutes:

```
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} | '
              f'g_loss_u: {step_g_loss_u:6.4f} | '
              f'g_loss_s: {step_g_loss_s:6.4f} | '
              f'g_loss_v: {step_g_loss_v:6.4f} | '
              f'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)
```

Now we can finally generate synthetic time series!

Generating synthetic time series

To evaluate the `TimeGAN` results, we will generate synthetic time series by drawing random inputs and feeding them to the `synthetic_data` network just described in the preceding section. More specifically, we'll create roughly as many artificial series with 24 observations on the six tickers as there are overlapping windows in the real dataset:

```
generated_data = []
for i in range(int(n_windows / batch_size)):
    Z_ = next(random_series)
    d = synthetic_data(Z_)
    generated_data.append(d)
len(generated_data)
35
```

The result is 35 batches containing 128 samples, each with the dimensions 24×6, that we stack like so:

```
generated_data = np.array(np.vstack(generated_data))
generated_data.shape
(4480, 24, 6)
```

We can use the trained `MinMaxScaler` to revert the synthetic output to the scale of the input series:

```
generated_data = (scaler.inverse_transform(generated_data
                                           .reshape(-1, n_seq))
                  .reshape(-1, seq_len, n_seq))
```

Figure 21.6 displays samples of the six synthetic series and the corresponding real series. The synthetic data generally reflects a variation of behavior not unlike its real counterparts and, after rescaling, roughly (due to the random input) matches its range:

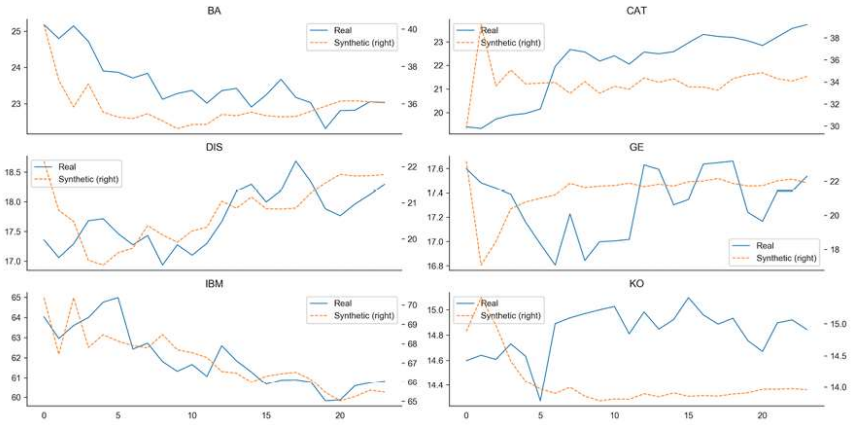

Figure 21.6: TimeGAN output—six synthetic prices series and their real counterparts

Now it's time to take a closer look at how to more thoroughly evaluate the quality of the synthetic data.

Evaluating the quality of synthetic time-series data

The TimeGAN authors assess the quality of the generated data with respect to three practical criteria:

- **Diversity**: The distribution of the synthetic samples should roughly match that of the real data.
- **Fidelity**: The sample series should be indistinguishable from the real data.
- **Usefulness**: The synthetic data should be as useful as its real counterparts for solving a predictive task.

They apply three methods to evaluate whether the synthetic data actually exhibits these characteristics:

- **Visualization**: For a qualitative diversity assessment of diversity, we use dimensionality reduction—**principal component analysis (PCA)** and **t-SNE** (see *Chapter 13*, *Data-Driven Risk Factors and Asset Allocation with Unsupervised Learning*)—to visually inspect how closely the distribution of the synthetic samples resembles that of the original data.
- **Discriminative score**: For a quantitative assessment of fidelity, the test error of a time-series classifier, such as a two-layer LSTM (see *Chapter 18*, *CNNs for Financial Time Series and Satellite Images*), lets us evaluate whether real and synthetic time series can be differentiated or are, in fact, indistinguishable.

- **Predictive score**: For a quantitative measure of usefulness, we can compare the test errors of a sequence prediction model trained on, alternatively, real or synthetic data to predict the next time step for the real data.

We'll apply and discuss the results of each method in the following sections. See the notebook `evaluating_synthetic_data` for the code samples and additional details.

Assessing diversity – visualization using PCA and t-SNE

To visualize the real and synthetic series with 24 time steps and six features, we will reduce their dimensionality so that we can plot them in two dimensions. To this end, we will sample 250 normalized sequences with six features each and reshape them to obtain data with the dimensionality 1,500×24 (showing only the steps for real data; see the notebook for the synthetic data):

```
# same steps to create real sequences for training
real_data = get_real_data()
# reload synthetic data
synthetic_data = np.load('generated_data.npy')
synthetic_data.shape
(4480, 24, 6)
# ensure same number of sequences
real_data = real_data[:synthetic_data.shape[0]]
sample_size = 250
idx = np.random.permutation(len(real_data))[:sample_size]
real_sample = np.asarray(real_data)[idx]
real_sample_2d = real_sample.reshape(-1, seq_len)
real_sample_2d.shape
(1500, 24)
```

PCA is a linear method that identifies a new basis with mutually orthogonal vectors that, successively, capture the directions of maximum variance in the data. We will compute the first two components using the real data and then project both real and synthetic samples onto the new coordinate system:

```
pca = PCA(n_components=2)
pca.fit(real_sample_2d)
pca_real = (pd.DataFrame(pca.transform(real_sample_2d))
            .assign(Data='Real'))
pca_synthetic = (pd.DataFrame(pca.transform(synthetic_sample_2d))
                 .assign(Data='Synthetic'))
```

t-SNE is a nonlinear manifold learning method for the visualization of high-dimensional data. It converts similarities between data points to joint probabilities and aims to minimize the Kullback-Leibler divergence between the joint probabilities of the low-dimensional embedding and the high-dimensional data (see *Chapter 13*, *Data-Driven Risk Factors and Asset Allocation with Unsupervised Learning*). We compute t-SNE for the combined real and synthetic data as follows:

```
tsne_data = np.concatenate((real_sample_2d,
                            synthetic_sample_2d), axis=0)
tsne = TSNE(n_components=2, perplexity=40)
tsne_result = tsne.fit_transform(tsne_data)
```

Figure 21.7 displays the PCA and t-SNE results for a qualitative assessment of the similarity of the real and synthetic data distributions. Both methods reveal strikingly similar patterns and significant overlap, suggesting that the synthetic data captures important aspects of the real data characteristics.

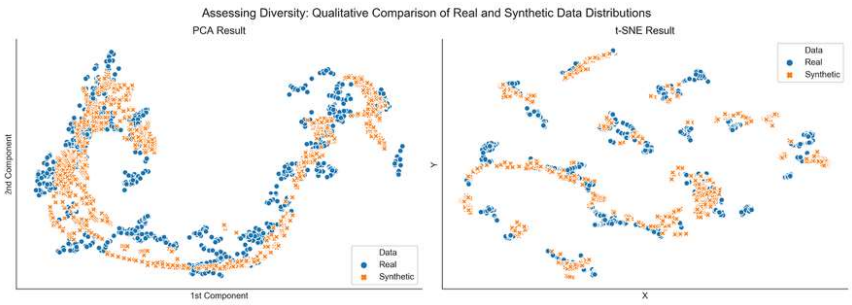

Figure 21.7: 250 samples of real and synthetic data in two dimensions

Assessing fidelity – time-series classification performance

The visualization only provides a qualitative impression. For a quantitative assessment of the fidelity of the synthetic data, we will train a time-series classifier to distinguish between real and fake data and evaluate its performance on a held-out test set.

More specifically, we will select the first 80 percent of the rolling sequences for training and the last 20 percent as a test set, as follows:

```
synthetic_data.shape
(4480, 24, 6)
n_series = synthetic_data.shape[0]
idx = np.arange(n_series)
n_train = int(.8*n_series)
train_idx, test_idx = idx[:n_train], idx[n_train:]
```

```
train_data = np.vstack((real_data[train_idx],
                        synthetic_data[train_idx]))
test_data = np.vstack((real_data[test_idx],
                       synthetic_data[test_idx]))
n_train, n_test = len(train_idx), len(test_idx)
train_labels = np.concatenate((np.ones(n_train),
                               np.zeros(n_train)))
test_labels = np.concatenate((np.ones(n_test),
                              np.zeros(n_test)))
```

Then we will create a simple RNN with six units that receives mini batches of real and synthetic series with the shape 24×6 and uses a sigmoid activation. We will optimize it using binary cross-entropy loss and the Adam optimizer, while tracking the AUC and accuracy metrics:

```
ts_classifier = Sequential([GRU(6, input_shape=(24, 6), name='GRU'),
                            Dense(1, activation='sigmoid', name='OUT')])
ts_classifier.compile(loss='binary_crossentropy',
                      optimizer='adam',
                      metrics=[AUC(name='AUC'), 'accuracy'])
Model: "Time Series Classifier"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
GRU (GRU)                    (None, 6)                 252
_________________________________________________________________
OUT (Dense)                  (None, 1)                 7
=================================================================
Total params: 259
Trainable params: 259
```

The model has 259 trainable parameters. We will train it for 250 epochs on batches of 128 randomly selected samples and track the validation performance:

```
result = ts_classifier.fit(x=train_data,
                           y=train_labels,
                           validation_data=(test_data, test_labels),
                           epochs=250, batch_size=128)
```

Once the training completes, evaluation of the test set yields a classification error of almost 56 percent on the balanced test set and a very low AUC of 0.15:

```
ts_classifier.evaluate(x=test_data, y=test_labels)
56/56 [==============================] - 0s 2ms/step - loss: 3.7510 - AUC: 0.1596 - accur
```

Figure 21.8 plots the accuracy and AUC performance metrics for both train and test data over the 250 training epochs:

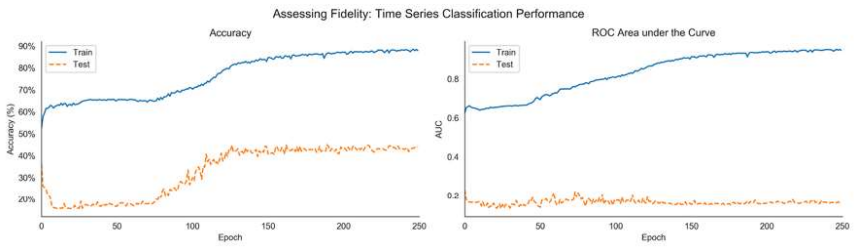

Figure 21.8: Train and test performance of the time-series classifier over 250 epochs

The plot shows that that model is not able to learn the difference between the real and synthetic data in a way that generalizes to the test set. This result suggests that the quality of the synthetic data meets the fidelity standard.

Assessing usefulness – train on synthetic, test on real

Finally, we want to know how useful synthetic data is when it comes to solving a prediction problem. To this end, we will train a time-series prediction model alternatively on the synthetic and the real data to predict the next time step and compare the performance on a test set created from the real data.

More specifically, we will select the first 23 time steps of each sequence as input, and the final time step as output. At the same time, we will split the real data into train and test sets using the same temporal split as in the previous classification example:

```
real_data.shape, synthetic_data.shape
((4480, 24, 6), (4480, 24, 6))
real_train_data = real_data[train_idx, :23, :]
real_train_label = real_data[train_idx, -1, :]
real_test_data = real_data[test_idx, :23, :]
real_test_label = real_data[test_idx, -1, :]
real_train_data.shape, real_train_label.shape
((3584, 23, 6), (3584, 6))
```

We will select the complete synthetic data for training since abundance is one of the reasons we generated it in the first place:

```
synthetic_train = synthetic_data[:, :23, :]
synthetic_label = synthetic_data[:, -1, :]
```

```
synthetic_train.shape, synthetic_label.shape
((4480, 23, 6), (4480, 6))
```

We will create a one-layer RNN with 12 GRU units that predicts the last time steps for the six stock price series and, thus, has six linear output units. The model uses the Adam optimizer to minimize the mean absolute error (MAE):

```
def get_model():
    model = Sequential([GRU(12, input_shape=(seq_len-1, n_seq)),
                        Dense(6)])
    model.compile(optimizer=Adam(),
                  loss=MeanAbsoluteError(name='MAE'))
    return model
```

We will train the model twice using the synthetic and real data for training, respectively, and the real test set to evaluate the out-of-sample performance. Training on synthetic data works as follows; training on real data works analogously (see the notebook):

```
ts_regression = get_model()
synthetic_result = ts_regression.fit(x=synthetic_train,
                                     y=synthetic_label,
                                     validation_data=(
                                         real_test_data,
                                         real_test_label),
                                     epochs=100,
                                     batch_size=128)
```

Figure 21.9 plots the MAE on the train and test sets (on a log scale so we can spot the differences) for both models. It turns out that the MAE is slightly lower after training on the synthetic dataset:

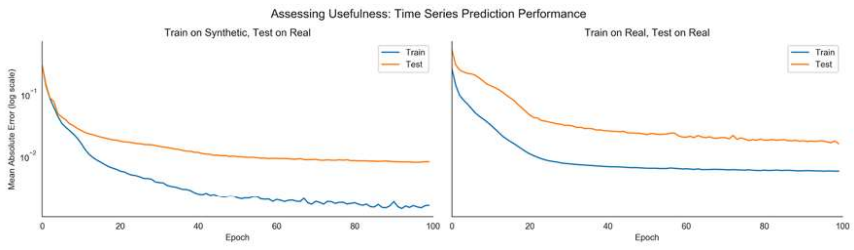

Figure 21.9: Train and test performance of the time-series prediction model over 100 epochs

The result shows that synthetic training data may indeed be useful. On the specific predictive task of predicting the next daily stock price for six

tickers, a simple model trained on synthetic TimeGAN data delivers equal or better performance than training on real data.

Lessons learned and next steps

The perennial problem of overfitting that we encountered throughout this book implies that the ability to generate useful synthetic data would be quite valuable. The TimeGAN example justifies cautious optimism in this regard. At the same time, there are some **caveats**: we generated price data for a small number of assets at a daily frequency. In reality, we are probably interested in returns for a much larger number of assets, possibly at a higher frequency. The **cross-sectional and temporal dynamics** will certainly become more complex and may require adjustments to the TimeGAN architecture and training process.

These limitations of the experiment, however promising, imply natural next steps: we need to expand the scope to higher-dimensional time series containing information other than prices and also need to test their usefulness in the context of more complex models, including for feature engineering. These are very early days for synthetic training data, but this example should equip you to pursue your own research agenda towards more realistic solutions.

Summary

In this chapter, we introduced GANs that learn a probability distribution over the input data and are thus capable of generating synthetic samples that are representative of the target data.

While there are many practical applications for this very recent innovation, they could be particularly valuable for algorithmic trading if the success in generating time-series training data in the medical domain can be transferred to financial market data. We learned how to set up adversarial training using TensorFlow. We also explored TimeGAN, a recent example of such a model, tailored to generating synthetic time-series data.

In the next chapter, we focus on reinforcement learning where we will build agents that interactively learn from their (market) environment.