

# 1

# Autoencoders and Generative Adversarial Nets

In this chapter, we present two unsupervised learning techniques that leverage deep learning: autoencoders, which have been around for decades, and **Generative Adversarial Networks (GANs)**, which were introduced by Ian Goodfellow in 2014 and which Yann LeCun has called the *most exciting idea in AI in the last ten years*. They complement the methods for dimensionality reduction and clustering introduced in Chapter 12, *Unsupervised Learning*.

Unsupervised learning addresses **machine learning (ML)** challenges such as the limited availability of labeled data and the curse of dimensionality that requires exponentially more samples for successful learning from complex, real-life data with many features. At a higher level, unsupervised learning resembles human learning and the development of common sense much more closely than supervised and reinforcement learning algorithms. More specifically, it aims to discover structure and regularities in the world from data so that it can predict missing input, that is, fill in the blanks from the observed parts, which is why it's also called **predictive learning**.

An autoencoder is a neural network trained to reproduce the input while learning a new representation of the data, encoded by the parameters of a hidden layer. Autoencoders have long been used for nonlinear dimensionality reduction and manifold learning. More recently, autoencoders have been designed as generative models that learn probability distributions over observed and latent variables. A variety of designs leverage the feedforward network, **Convolutional Neural Network (CNN)**, and **recurrent neural network (RNN)** architectures we covered in the last three chapters.

GANs are a recent innovation that train two neural nets—a generator and a 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 new (fake) samples that are representative of a certain target distribution.

GANs have produced a wave of research and can be successfully applied in many domains. An example from the medical domain that could potentially be highly relevant for trading is the generation of time-series data that simulates alternative trajectories and can be used to train supervised or reinforcement algorithms.

More specifically, in this chapter we'll cover the following topics:

- Which types of autoencoders are of practical use and how they work
- How to build and train autoencoders using Python
- How GANs work, why they're useful, and how they could be applied to trading
- How to build GANs using Python

You can find the code examples, references, and additional resources in this chapter's directory of the GitHub repository for this book at <https://github.com/PacktPublishing/Hands-On-Machine-Learning-for-Algorithmic-Trading>.

## How autoencoders work

In Chapter 17, *Deep Learning*, we saw that neural networks are successful at supervised learning by extracting a hierarchical feature representation that's useful for the given task. CNNs, for example, learn and synthesize increasingly complex patterns useful for identifying or detecting objects in an image.

An autoencoder, in contrast, is a neural network designed exclusively to learn a new representation, that is, an encoding of the input. To this end, the training forces the network to faithfully reproduce the input. Since autoencoders typically use the same data as input and output, they are also considered an instance of self-supervised learning.

In the process, the parameters of a hidden layer,  $h$ , become the code that represents the input. More specifically, the network can be viewed as consisting of an encoder function,  $h=f(x)$ , that learns the hidden layer's parameters from the input,  $x$ , and a decoder function,  $g$ , that learns to reconstruct the input from the encoding, rather than learning the identity function:

$$x = g(f(x))$$

While the identity function perfectly reproduces the input, it is more interesting to constrain the reproduction so that the hidden layer produces a new representation of a practical value.

## Nonlinear dimensionality reduction

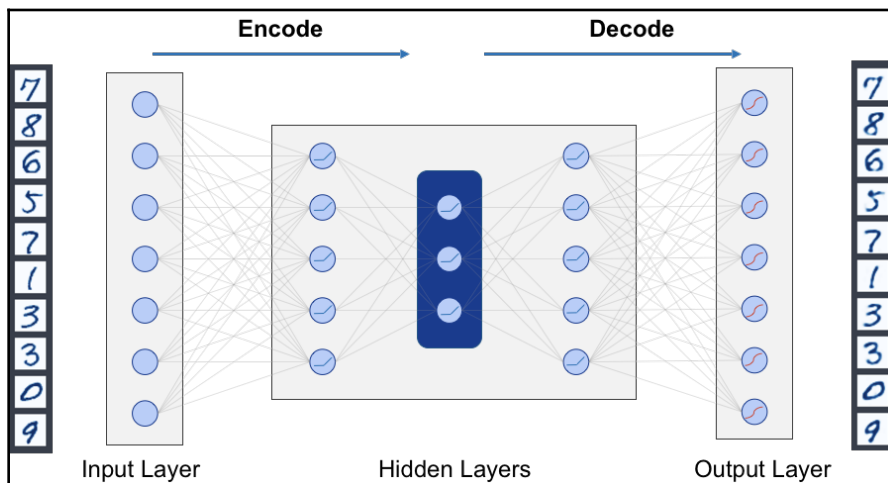
A traditional use case includes dimensionality reduction, achieved by limiting the size of the hidden layer so that it performs lossy compression. Such an autoencoder is called **undercomplete** and the purpose is to force it to learn the most salient properties of the data by minimizing a loss function,  $L$ , of the following form:

$$\mathbb{E} L(x, g(f(x)))$$

An example loss function that we will explore in the next section is simply the **Mean Squared Error (MSE)** evaluated on the pixel values of the input images and their reconstruction.

The appeal of autoencoders, when compared to linear dimensionality-reduction methods, such as **Principal Components Analysis (PCA)**—see Chapter 12, *Unsupervised Learning*, is the availability of nonlinear encoder and decoder activation functions that allow for a wider range of encodings than.

The following screenshot illustrates the encoder-decoder logic of an undercomplete feedforward autoencoder with three hidden layers. The hidden units use nonlinear activation functions such as **Rectified Linear Units (ReLU)**, sigmoid, or tanh and have fewer elements than the input that the network aims to reconstruct despite these constraints:



We will see down that autoencoders can be useful with only a single layer each for the encoder and the decoder. However, deeper autoencoders offer many advantages, just as for neural networks more generally that autoencoders are just a special case of. These advantages include the ability to learn more complex encodings, achieve better compression, and do so with less computational and training effort. See references on GitHub for details on experimental evidence: <https://github.com/PacktPublishing/Hands-On-Machine-Learning-for-Algorithmic-Trading>.

## Convolutional autoencoders

In addition to feedforward architectures, autoencoders can also use convolutional layers to learn hierarchical feature representations. As discussed in [Chapter 17, \*Deep Learning\*](#), feedforward architectures aren't well-suited to capturing local correlations typical of data with a grid-like structure.

Convolutional autoencoders, instead, leverage convolutions and parameter-sharing to learn hierarchical patterns and features irrespective of their location, translation, or changes in size.

We'll explore implementations of convolutional autoencoders for image data in the next section.

## Sparsity constraints with regularized autoencoders

The powerful capabilities of neural networks to represent complex functions require tight limitations of the capacity of the encoder and decoder to force the extraction of a useful signal rather than noise. In other words, when it is too easy for the network to recreate the input, it fails to learn only the most interesting aspects of the data.

This challenge is similar to the overfitting phenomenon that frequently occurs when using models with a high capacity for supervised learning. Just as in these settings, regularization can help by adding constraints to the autoencoder that facilitate the learning of a useful representation.

A common approach that we explore later is the use of L1 regularization, which adds a penalty to the loss function in the form of the sum of the absolute values of the weights. The L1 norm results in sparse encodings because it forces parameter values to zero that don't capture the most salient variation in the data. As a result, even overcomplete autoencoders with hidden layers of higher dimension than the input may be able to learn signal content.

## Fixing corrupted data with denoising autoencoders

The autoencoders we've discussed so far are designed to reproduce the input despite capacity constraints. An alternative approach trains autoencoders with corrupted input to output the desired, original data points.

Corrupted input are a different way of preventing the network from learning the identity function; instead, they extract the signal or salient features from the data. Denoising autoencoders have been shown to learn the data-generating process of the original data, and have become popular in generative modeling where the goal is to learn the probability distribution that gives rise to the input.

## Sequence-to-sequence autoencoders

RNNs (see Chapter 18, *Recurrent Neural Networks*) have been developed to take into account the dynamics and dependencies over potentially long ranges often found in sequential data. Similarly, sequence-to-sequence autoencoders aim to learn representations attuned to the nature of data generated in sequence.

Sequence-to-sequence autoencoders are based on RNN components, such as **Long Short-Term Memory (LSTM)** or **Gated Recurrent Units (GRUs)**. They learn a compressed representation of sequential data and have been applied to video, text, audio, and time-series data.

As mentioned in the last chapter, encoder-decoder architectures allow RNNs to process input and output sequences of variable lengths. These architectures underpin many advances in complex sequence-prediction tasks, such as speech recognition and text translation.

It roughly works as follows: the LSTM encoder processes the input sequence step by step to learn a hidden state. This state becomes a learned representation of the sequence in the form of a fixed-length vector. The LSTM decoder receives this state as input and uses it to generate the output sequence. See references on GitHub for additional details: <https://github.com/PacktPublishing/Hands-On-Machine-Learning-for-Algorithmic-Trading>.

## Variational autoencoders

**Variational Autoencoders (VAE)** are more recent developments focused on generative modeling. More specifically, VAEs are designed to learn a latent variable model for the input data. Note that we encountered latent variables in *Chapter 14, Topic Modeling*.

Hence, VAEs do not let the network learn arbitrary functions as long as it faithfully reproduces the input. Instead, they aim to learn the parameters of a probability distribution that generates the input data. In other words, VAEs are generative models because, if successful, you can generate new data points by sampling from the distribution learned by the VAE.

The operation of a VAE is more complex than the autoencoders discussed so far and the details are beyond the scope of this book. They're able to learn high-capacity input encodings without regularization—this is useful because the models aim to maximize the probability of the training data rather than to reproduce the input.

The `variational_autoencoder` notebook includes a sample VAE implementation applied to the Fashion MNIST data, adapted from a Keras tutorial. Please see references on GitHub for additional background: <https://github.com/PacktPublishing/Hands-On-Machine-Learning-for-Algorithmic-Trading>.

## Designing and training autoencoders using Python

In this section, we illustrate how to implement several of the autoencoder models introduced in the preceding section using Keras. We first load and prepare an image dataset that we use throughout this section because it makes it easier to visualize the results of the encoding process.

We then proceed to build autoencoders using deep feedforward nets, sparsity constraints, and convolutions and then apply the latter to denoise images.

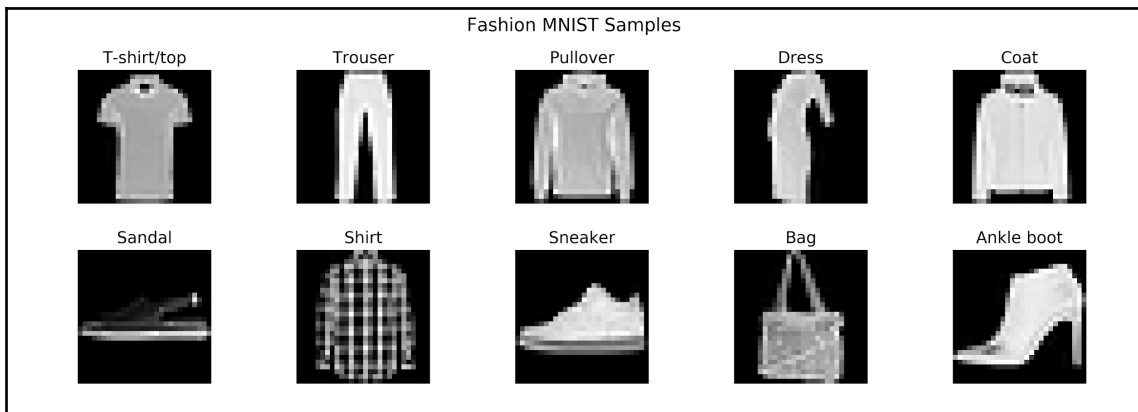
## Preparing the data

For illustration, we'll use the Fashion MNIST dataset, a modern drop-in replacement for the classic MNIST handwritten digit dataset popularized by Yann LeCun with LeNet in the 1990s. We also relied on this dataset in *Chapter 12, Unsupervised Learning*.

Keras makes it easy to access the 60,000 train and 10,000 test grayscale samples with a resolution of  $28 \times 28$  pixels:

```
from keras.datasets import fashion_mnist
(X_train, y_train), (X_test, y_test) = fashion_mnist.load_data()
X_train.shape, X_test.shape
((60000, 28, 28), (10000, 28, 28))
```

The data contains clothing items from 10 classes. The following screenshot plots a sample image for each class:



We reshape the data so that each image is represented by a flat one-dimensional pixel vector with  $28 \times 28 = 784$  elements normalized to the range of  $[0, 1]$ :

```
image_size = 28 # pixels per side
input_size = image_size ** 2 # 784

def data_prep(x, size=input_size):
    return x.reshape(-1, size).astype('float32')/255

X_train_scaled = data_prep(X_train)
X_test_scaled = data_prep(X_test)
X_train_scaled.shape, X_test_scaled.shape
((60000, 784), (10000, 784))
```

## One-layer feedforward autoencoder

We start with a vanilla feedforward autoencoder with a single hidden layer to illustrate the general design approach using the functional Keras API and establish a performance baseline.

The first step is a placeholder for the flattened image vectors with 784 elements:

```
input_ = Input(shape=(input_size,), name='Input')
```

The encoder part of the model consists of a fully-connected layer that learns the new, compressed representation of the input. We use 32 units for a compression ratio of 24.5:

```
encoding_size = 32 # compression factor: 784 / 32 = 24.5

encoding = Dense(units=encoding_size,
                 activation='relu',
                 name='Encoder')(input_)
```

The decoding part reconstructs the compressed data to its original size in a single step:

```
decoding = Dense(units=input_size,
                 activation='sigmoid',
                 name='Decoder')(encoding)
```

We instantiate the `Model` class with the chained input and output elements that implicitly define the computational graph, as follows:

```
autoencoder = Model(inputs=input_,
                   outputs=decoding,
                   name='Autoencoder')
```

The defined encoder-decoder computation uses almost 51,000 parameters:

```
autoencoder.summary()
```

Layer (type)	Output Shape	Param #
Input (InputLayer)	(None, 784)	0
Encoder (Dense)	(None, 32)	25120
Decoder (Dense)	(None, 784)	25872
Total params: 50,992		
Trainable params: 50,992		
Non-trainable params: 0		



The functional API allows us to use parts of the model's chain as separate encoder and decoder models that use the autoencoder's parameters learned during training.

## Defining the enoder

The encoder just uses the input and hidden layer with about half of the total parameters:

```
encoder = Model(inputs=input_, outputs=encoding, name='Encoder')

encoder.summary()
```

Layer (type)	Output Shape	Param #
Input (InputLayer)	(None, 784)	0
Encoder (Dense)	(None, 32)	25120

```

Total params: 25,120
Trainable params: 25,120
Non-trainable params: 0

```

Once we train the autoencoder, we can use the encoder to compress the data.

## Defining the decoder

The decoder consists of the last autoencoder layer, fed by a placeholder for the encoded data:

```
encoded_input = Input(shape=(encoding_size,), name='Decoder_Input')
decoder_layer = autoencoder.layers[-1](encoded_input)
decoder = Model(inputs=encoded_input, outputs=decoder_layer)

decoder.summary()
```

Layer (type)	Output Shape	Param #
Decoder_Input (InputLayer)	(None, 32)	0
Decoder (Dense)	(None, 784)	25872

```

Total params: 25,872
Trainable params: 25,872
Non-trainable params: 0

```

## Training the model

We compile the model to use the Adam optimizer (see Chapter 17, *Deep Learning*) to minimize the MSE between the input data and the reproduction achieved by the autoencoder. To ensure that the autoencoder learns to reproduce the input, we train the model using the same input and output data:

```
autoencoder.compile(optimizer='adam', loss='mse')
autoencoder.fit(x=X_train_scaled, y=X_train_scaled,
               epochs=100, batch_size=32,
               shuffle=True, validation_split=.1,
               callbacks=[tb_callback, early_stopping, checkpointer])
```

## Evaluating the results

Training stops after some 20 epochs with a test RMSE of 0.1122:

```
mse = autoencoder.evaluate(x=X_test_scaled, y=X_test_scaled)
mse, mse **.5
(0.012588984733819962, 0.11220064497951855)
```

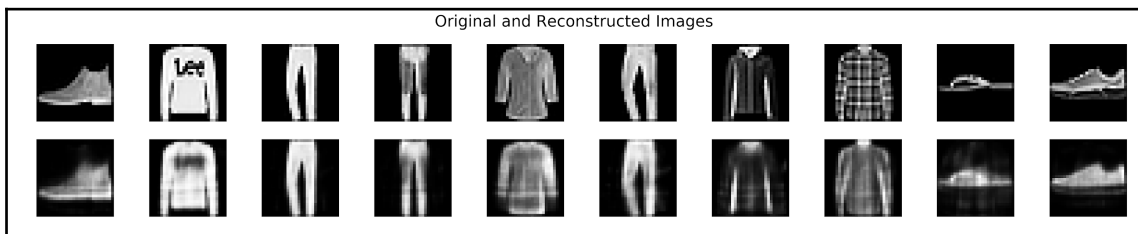
To encode data, we use the encoder we just defined, like so:

```
encoded_test_img = encoder.predict(X_test_scaled)
Encoded_test_img.shape
(10000, 32)
```

The decoder takes the compressed data and reproduces the output according to the autoencoder training results:

```
decoded_test_img = decoder.predict(encoded_test_img)
decoded_test_img.shape
(10000, 784)
```

The following screenshot shows ten original images and their reconstruction by the autoencoder and illustrates the loss after compression:



## Feedforward autoencoder with sparsity constraints

The addition of regularization is fairly straightforward. We can apply it to the dense encoder layer using Keras' `activity_regularizer`, as follows:

```
encoding_l1 = Dense(units=encoding_size,
                    activation='relu',
                    activity_regularizer=regularizers.l1(10e-5),
                    name='Encoder_L1')(input_)
```

The input and decoding layers remain unchanged. In this example, with a compression of factor 24.5, regularization negatively affects performance with a test RMSE of 0.2946.

## Deep feedforward autoencoder

To illustrate the benefit of adding depth to the autoencoder, we build a three-layer feedforward model that successively compresses the input from 784 to 128, 64, and 34 units, respectively:

```
input_ = Input(shape=(input_size,))
x = Dense(128, activation='relu', name='Encoding1')(input_)
x = Dense(64, activation='relu', name='Encoding2')(x)
encoding_deep = Dense(32, activation='relu', name='Encoding3')(x)

x = Dense(64, activation='relu', name='Decoding1')(encoding_deep)
x = Dense(128, activation='relu', name='Decoding2')(x)
decoding_deep = Dense(input_size, activation='sigmoid',
                      name='Decoding3')(x)

autoencoder_deep = Model(input_, decoding_deep)
```

The resulting model has over 222,000 parameters, more than four times the capacity of the preceding single-layer model:

```
autoencoder_deep.summary()
```

Layer (type)	Output Shape	Param #
=====		
input_1 (InputLayer)	(None, 784)	0
-----		
Encoding1 (Dense)	(None, 128)	100480
-----		
Encoding2 (Dense)	(None, 64)	8256

---

Encoding3	(Dense)	(None, 32)	2080
-----------	---------	------------	------

---

Decoding1	(Dense)	(None, 64)	2112
-----------	---------	------------	------

---

Decoding2	(Dense)	(None, 128)	8320
-----------	---------	-------------	------

---

Decoding3	(Dense)	(None, 784)	101136
-----------	---------	-------------	--------

---

```

Total params: 222,384
Trainable params: 222,384
Non-trainable params: 0

```

---

Training stops after 10 epochs and results in a 14% reduction of the test RMSE to 0.0968. Due to the low resolution, it is difficult to visually note the better reconstruction.

## Visualizing the encoding

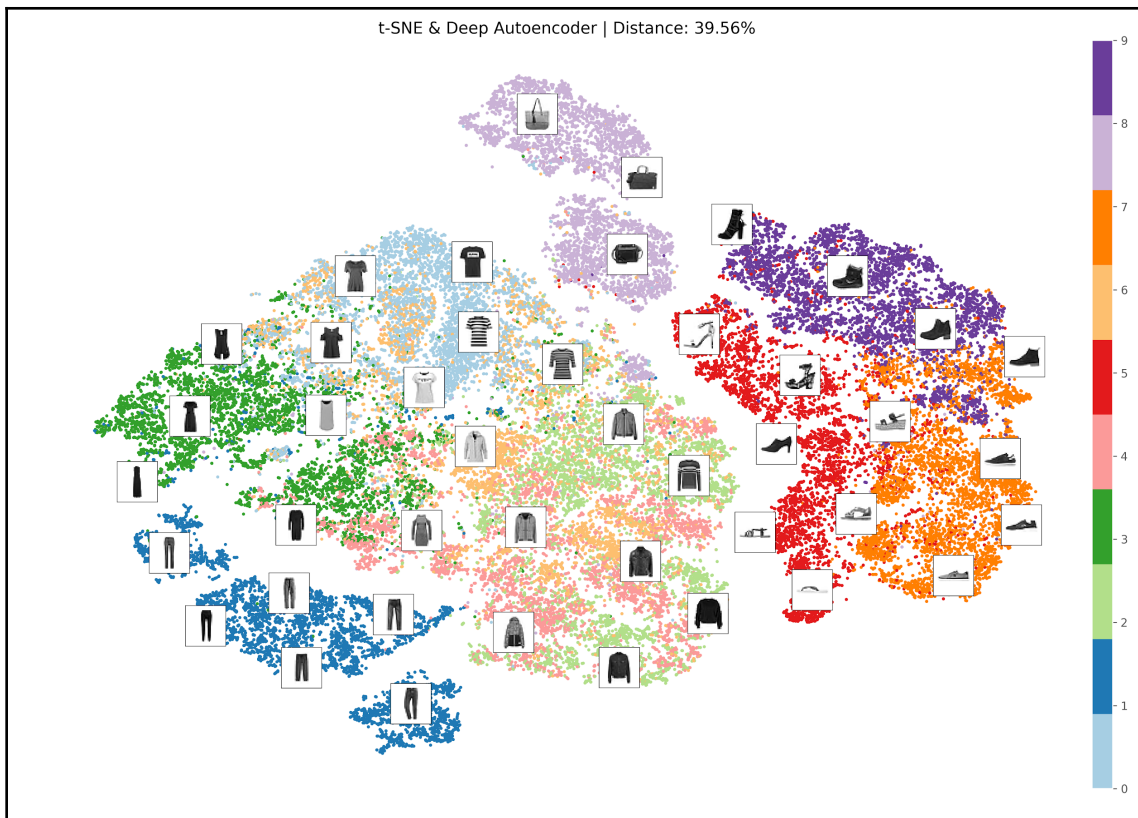
We can use the **t-distributed Stochastic Neighbor Embedding (t-SNE)** manifold learning technique, see Chapter 12, *Unsupervised Learning*, to visualize and assess the quality of the encoding learned by the autoencoder's hidden layer.

If the encoding is successful in capturing the salient features of the data, the compressed representation of the data should still reveal a structure aligned with the 10 classes that differentiate the observations.

We use the output of the deep encoder we just trained to obtain the 32-dimensional representation of the test set:

```
tsne = TSNE(perplexity=25, n_iter=5000)
train_embed = tsne.fit_transform(encoder_deep.predict(X_train_scaled))
```

The following distribution shows that the 10 classes are separated, suggesting that the encoding is useful as a lower-dimensional representation that preserves key characteristics of the data:



## Convolutional autoencoders

The insights from Chapter 19, *Convolutional Neural Networks*, suggest we incorporate convolutional layers into the autoencoder to extract information characteristic of the grid-like structure of image data.

We define a three-layer encoder that uses 2D convolutions with 32, 16, and 8 filters, respectively, ReLU activations, and 'same' padding to maintain the input size. The resulting encoding size at the third layer is  $4 \times 4 \times 8 = 128$ , higher than for the preceding examples:

```
x = Conv2D(filters=32, kernel_size=(3, 3), activation='relu',
           padding='same', name='Encoding_Conv_1')(input_)
x = MaxPooling2D(pool_size=(2, 2), padding='same',
                 name='Encoding_Max_1')(x)
x = Conv2D(filters=16, kernel_size=(3, 3), activation='relu',
```

```

        padding='same', name='Encoding_Conv_2')(x)
x = MaxPooling2D(pool_size=(2, 2), padding='same',
                 name='Encoding_Max_2')(x)
x = Conv2D(filters=8, kernel_size=(3, 3), activation='relu',
           padding='same', name='Encoding_Conv_3')(x)
encoded_conv = MaxPooling2D(pool_size=(2, 2),
                             padding='same', name='Encoding_Max_3')(x)

```

We also define a matching decoder that reverses the number of filters and uses 2D upsampling instead of max pooling to reverse the reduction of the filter sizes. The three-layer autoencoder has 12,785 parameters, a little more than 5% of the capacity of the preceding deep autoencoder.

Training stops after 75 epochs and results in a further 9% reduction of the test RMSE, due to a combination of the ability of convolutional filters to learn more efficiently from image data and the larger encoding size.

## Denoising autoencoders

The application of an autoencoder to a denoising task only affects the training stage. In this example, we add noise to the Fashion MNIST data from a standard normal distribution while maintaining the pixel values in the range of [0, 1], as follows:

```

def add_noise(x, noise_factor=.3):
    return np.clip(x + noise_factor * np.random.normal(size=x.shape), 0, 1)
X_train_noisy = add_noise(X_train_scaled)
X_test_noisy = add_noise(X_test_scaled)

```

We then proceed to train the convolutional autoencoder on noisy input with the objective to learn how to generate the uncorrupted originals:

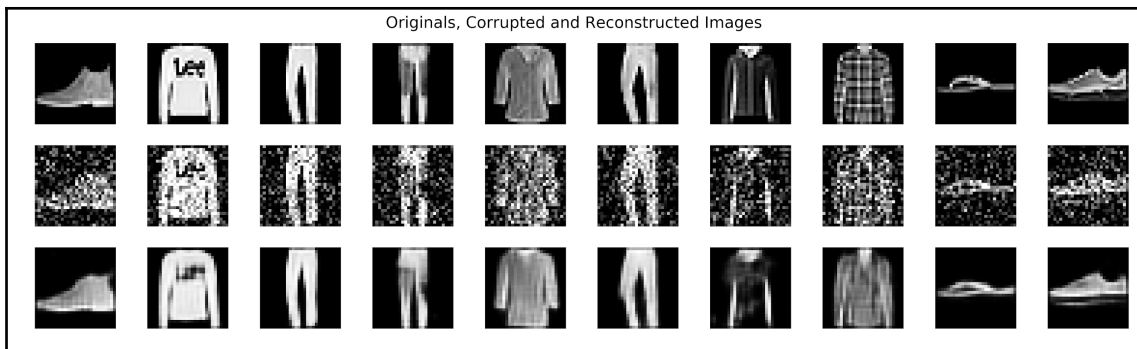
```

autoencoder_denoise.fit(x=X_train_noisy,
                        y=X_train_scaled,

                        ...)

```

After 60 epochs, the test RMSE is 0.926, unsurprisingly higher than before. The following screenshot shows, from top to bottom, the original images as well as the noisy and denoised versions. It illustrates that the autoencoder is successful in producing compressed encodings from the noisy images that are quite similar to those produced from the original images:



## How GANs work

The supervised learning algorithms that we focused on for most of this book receive input data that's typically complex and predicts a numerical or categorical label that we can compare to the ground truth to evaluate its performance. These algorithms are also called **discriminative models** because they learn to differentiate between different output classes.

## How generative and discriminative models differ

The goal of generative models is to produce complex output, such as realistic images, given simple input, which can even be random numbers. They achieve this by modeling a probability distribution over the possible output. This probability distribution can have many dimensions, for example, one for each pixel in an image or its character or token in a document. As a result, the model can generate output that are very likely representative of the class of output. In this context, we can refer Richard Feynman's quote:

*"What I cannot create, I do not understand."*

This is often used to emphasize that modeling generative distributions is an important step toward more general AI and resembles human learning, which succeeds using fewer samples.

Generative models are useful for several purposes beyond their ability to generate additional samples from a given distribution. For example, they can be incorporated into model-based reinforcement learning algorithms (see [Chapter 18, \*Recurrent Neural Networks\*](#)). Generative models can also be applied to time-series data to simulate alternative past or possible future trajectories that could be used for planning in reinforcement learning or supervised learning more generally. Other use cases include semi-supervised learning, where GANs facilitate feature-matching to assign missing labels with fewer training samples than current approaches.

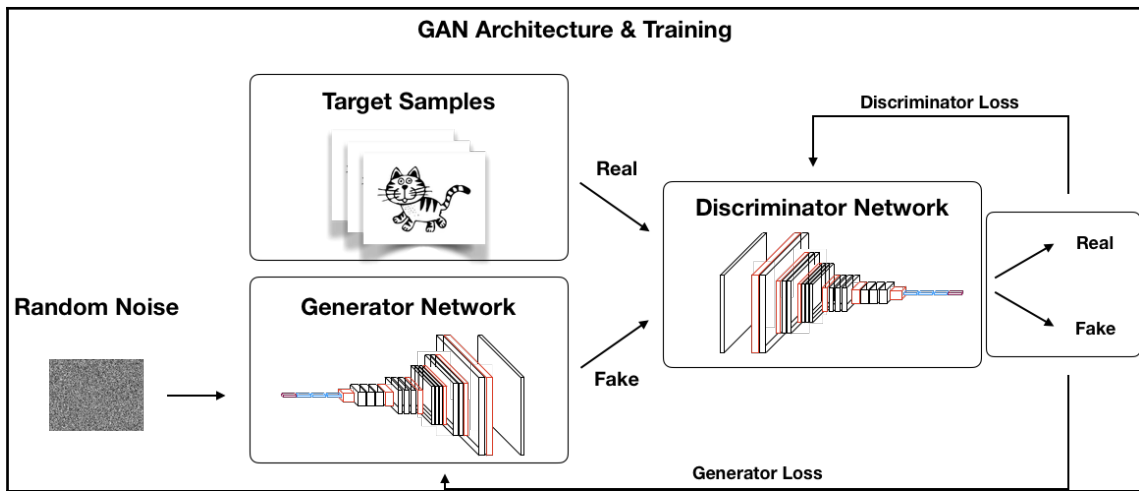
## How adversarial training works

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

The learning objective of the generator is to create output from random input, for example, images of faces. The discriminator, in turn, aims to differentiate between the generator's output and a set of training data that reflects the target output, for example, a database of celebrities as in a popular application. The overall purpose is that both networks get better at their respective tasks while they compete so that the generator ends up producing output that a machine can no longer distinguish from the originals.

The following diagram illustrates a generic GAN architecture and how training works. We assume the generator uses a deep CNN architecture (using the VGG example from [Chapter 18, \*Recurrent Neural Networks\*](#)) that's reversed, just like the decoder part of the convolutional autoencoder we discussed in the previous section. The generator receives a random input and produces a *fake* output image that's 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:





The recent GAN Lab is a great interactive tool inspired by TensorFlow Playground that allows the user to design GANs and visualize various aspects of the learning process and performance over time (see references on GitHub: <https://github.com/PacktPublishing/Hands-On-Machine-Learning-for-Algorithmic-Trading>).

## How GAN architectures are evolving

Since publication in 2014, GANs have experienced an enormous amount of interest, as evidenced by Yann LeCun's quote in the introduction, and have triggered a flurry of research.

The bulk of this work has refined the original architecture to adapt it to different domains and tasks, and expand it to include additional information, creating a **conditional GAN (cGAN)**. Additional research has focused on improving methods for the challenging training process that 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, but you can find additional references to both surveys and individual milestones on GitHub.

## Deep Convolutional GAN (DCGAN)

**Deep Convolutional GAN (DCGAN)** was motivated by the successful application of CNN to supervised learning for grid-like data. The architecture pioneered the use of GANs for unsupervised learning by developing a feature extractor based on adversarial training. It's 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: <https://github.com/PacktPublishing/Hands-On-Machine-Learning-for-Algorithmic-Trading>).

The DCGAN network takes uniformly-distributed random numbers as input and outputs a color image with a resolution of 64 x 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 preceding autoencoder example 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 Keras implementation layer.

## 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 precedingly by adding a third input value to the discriminator, which 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)**, which uses bounding-box information to not only generate synthetic images but also place objects at a given location.

## Successful and emerging GAN applications

Alongside a large variety of extensions and modifications of the original architecture, numerous applications have emerged. We list a few examples here and then dive into more detailed an application to time-series data that may become particularly relevant to algorithmic trading and investment (see GitHub references for additional examples).

## 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 Impressionist print from an arbitrary landscape photo.

## 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, shorted to StackGAN, uses a sentence as input and generates multiple images that match the description.

The architecture operates in two stages: 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 photo-realistic details.

## Photo-realistic image super-resolution

Super-resolution aims at producing photo-realistic higher-resolution 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](#), *Recurrent Neural Networks*) to produce impressive results that are likely to find commercial application.

## Synthetic time series with recurrent cGANs

Recurrent (conditional) GANs are two model architectures that aim to synthesize realistic real-valued multivariate time series. 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 RNNs (see the last chapter) for the generator and the discriminator. RCGANs further add auxiliary information in the spirit of cGANs (see the previous section).

The authors succeeded in generating visually and quantitatively compelling realistic samples. Furthermore, they evaluated 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 demonstrated 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 illustrated that RCGANs are capable of generating time-series data useful for supervised training. It could be a worthwhile endeavor to apply this approach to financial market data.

## Building GANs using Python

To illustrate the implementation of a GAN using Python, we use the **Deep Convolutional GAN (DCGAN)** example discussed precedingly to synthesize images from the fashion MNIST dataset. See the `deep_convolutional_generative_adversarial_network` notebook for implementation details and references.

## Defining the discriminator network

Both the discriminator and generator use a deep CNN architecture, wrapped in a function:

```
def build_discriminator():
    model = Sequential([
        Conv2D(32, kernel_size=3, strides=2,
              input_shape=img_shape, padding='same'),
        LeakyReLU(alpha=0.2),
        Dropout(0.25),
        Conv2D(64, kernel_size=3, strides=2,
              padding='same'),
        ZeroPadding2D(padding=((0, 1), (0, 1))),
        BatchNormalization(momentum=0.8),
        LeakyReLU(alpha=0.2),
        Dropout(0.25),
        Conv2D(128, kernel_size=3, strides=2,
              padding='same'),
        BatchNormalization(momentum=0.8),
        LeakyReLU(alpha=0.2),
        Dropout(0.25),
        Conv2D(256, kernel_size=3, strides=1,
              padding='same'),
        BatchNormalization(momentum=0.8),
        LeakyReLU(alpha=0.2),
        Dropout(0.25),
```

```
        Flatten(),
        Dense(1, activation='sigmoid')
    ])

    model.summary()
    img = Input(shape=img_shape)
    validity = model(img)
    return Model(img, validity)
```

A call to this function and subsequent compilation shows that this network has over 393,000 parameters.

## Defining the generator network

The generator network is slightly shallower but has more than twice as many parameters:

```
def build_generator():
    model = Sequential([
        Dense(128 * 7 * 7, activation='relu', input_dim=latent_dim),
        Reshape((7, 7, 128)),
        UpSampling2D(),
        Conv2D(128, kernel_size=3, padding='same'),
        BatchNormalization(momentum=0.8),
        Activation('relu'),
        UpSampling2D(),
        Conv2D(64, kernel_size=3, padding='same'),
        BatchNormalization(momentum=0.8),
        Activation('relu'),
        Conv2D(channels, kernel_size=3, padding='same'),
        Activation('tanh')])

    model.summary()

    noise = Input(shape=(latent_dim,))
    img = model(noise)

    return Model(noise, img)
```

## Combining both networks to define the GAN

The combined model consists of the stacked generator and discriminator and trains the former to fool the latter:

```
# The generator takes noise as input and generates imgs
z = Input(shape=(latent_dim,))
img = generator(z)
# For the combined model we will only train the generator
discriminator.trainable = False
# discriminator determines validity for generated images
valid = discriminator(img)
combined = Model(z, valid)
combined.compile(loss='binary_crossentropy', optimizer=optimizer)
```

## Adversarial training

Adversarial training iterates over the epochs, generates random image and noise input, and trains both the discriminator and the generator (as part of the combined model):

```
valid = np.ones((batch_size, 1))

fake = np.zeros((batch_size, 1))

for epoch in range(epochs):
    # Select a random half of images
    idx = np.random.randint(0, X_train.shape[0], batch_size)
    imgs = X_train[idx]

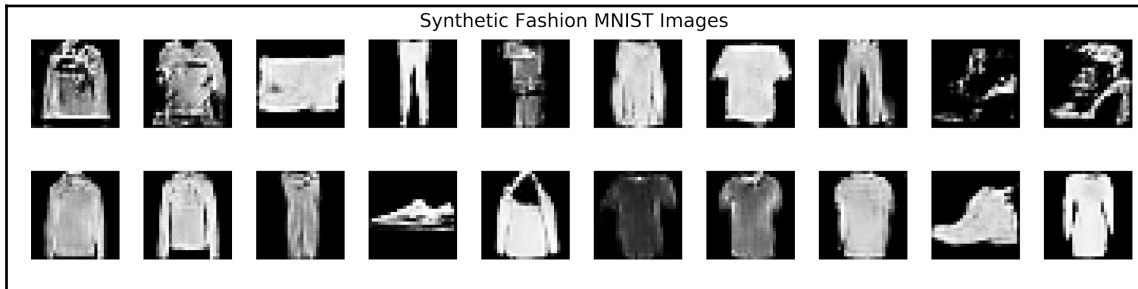
    # Sample noise and generate a batch of new images
    noise = np.random.normal(0, 1, (batch_size, latent_dim))
    gen_imgs = generator.predict(noise)

    # Train the discriminator
    d_loss_real = discriminator.train_on_batch(imgs, valid)
    d_loss_fake = discriminator.train_on_batch(gen_imgs, fake)
    d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)

    # Train the generator (wants discriminator to mistake images as real)
    g_loss = combined.train_on_batch(noise, valid)
```

## Evaluating the results

After 4,000 epochs, which only takes a few minutes, the synthetic images created from random noise clearly resemble the originals:



## Summary

In this chapter, we introduced two unsupervised learning methods that leverage deep learning. Autoencoders learn sophisticated, nonlinear feature representations that are capable of significantly compressing complex data while losing little information. As a result, they are very useful to counter the curse of dimensionality associated with rich datasets that have many features, which is especially common in alternative data. We also saw how to implement various types of autoencoders using Keras.

Then, we covered GANs, which learn a probability distribution over the input data and are hence 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. Finally, we learned how to set up adversarial training using Keras.

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