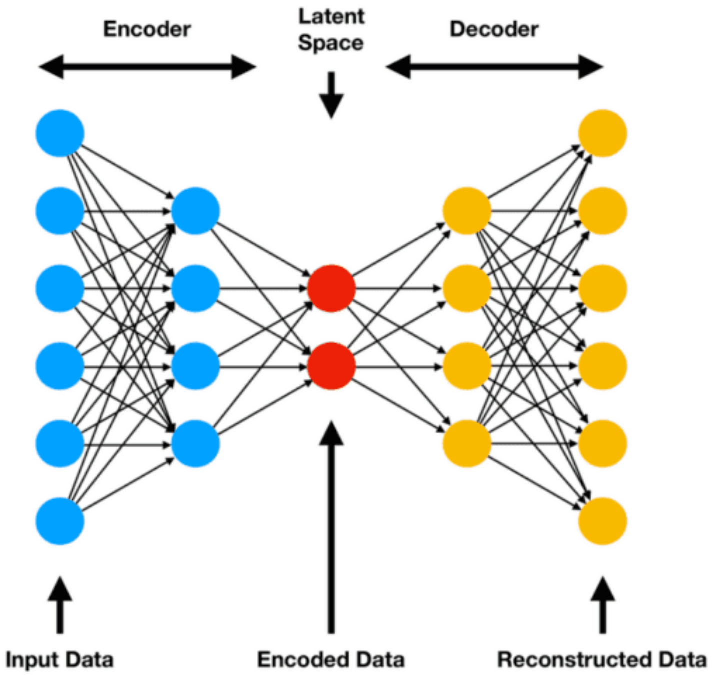
Hands-on Machine Learning

17. Autoencoders, GANs, and Diffusion Models

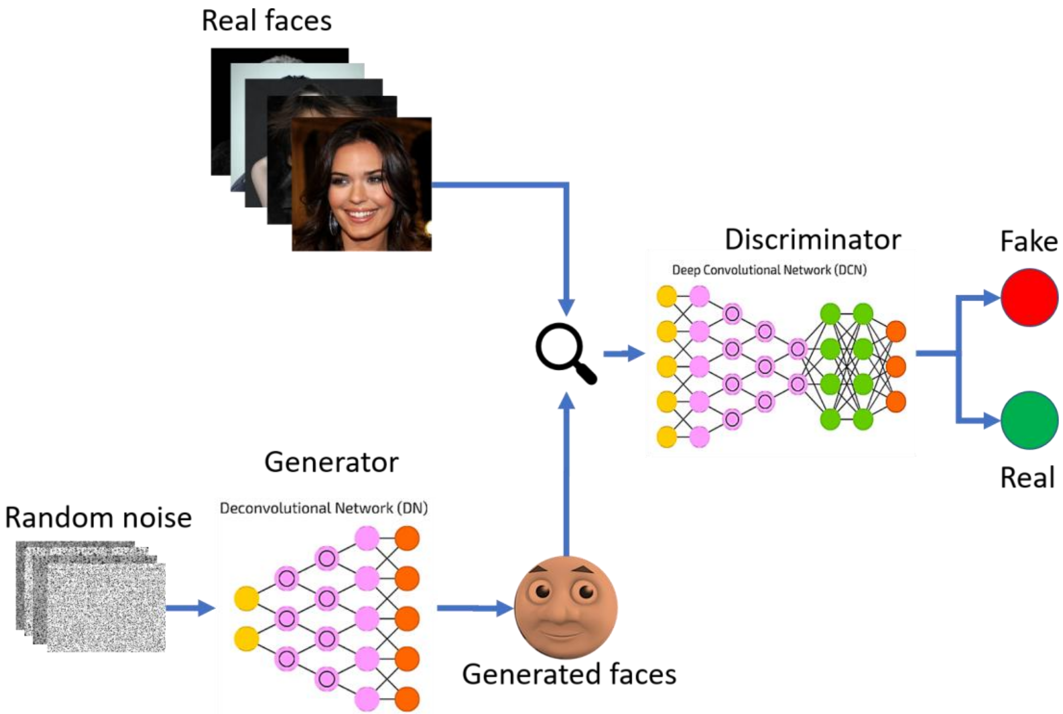
Autoencoders

- Autoencoders are neural networks capable of learning efficient representations of the unlabeled input data, called *codings* or *latent representations*.
  - The codings have a much lower dimensionality than the input data, making autoencoders useful for dimensionality reduction.
- Autoencoders also act as feature detectors, and can be used for unsupervised pretraining of deep neural networks.
- Some autoencoders are *generative models*: they can randomly generating new data that looks very similar to the training data.



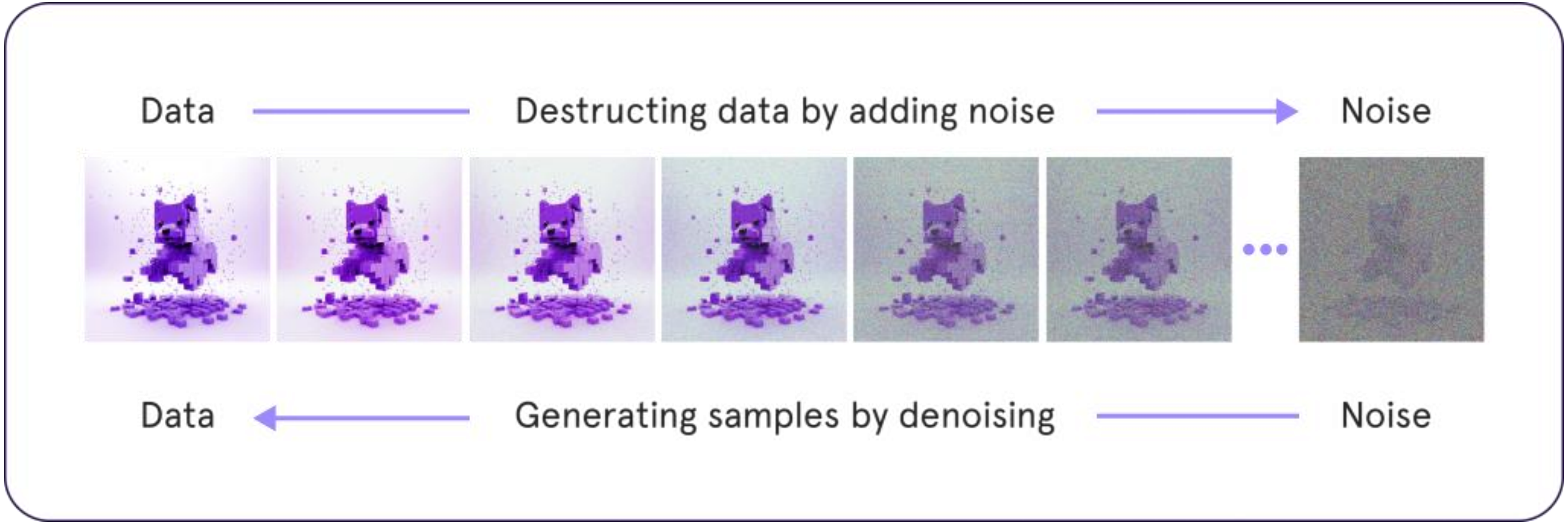
Generative Adversarial Networks

➢ *Generative adversarial networks* (GANs) are neural nets capable of generating data.



Diffusion Models

- Diffusion models are another approach to generative learning, uniquely capable of generating high-quality data by progressively adding noise to a dataset and then learning to reverse this process (denoising).



Comparing Generative Models

- Autoencoders, GANs, and diffusion models are all:
  - unsupervised
  - learn latent representations
  - can be used as generative models
  - have many similar applications

- However, they work very differently:
  - Autoencoders are designed to simply learn how to copy their inputs to their outputs.
  - GANs are composed of two competing neural networks that are trained adversarially.
  - A *denoising diffusion probabilistic model* (DDPM) is trained to iteratively remove small amounts of noise from data.

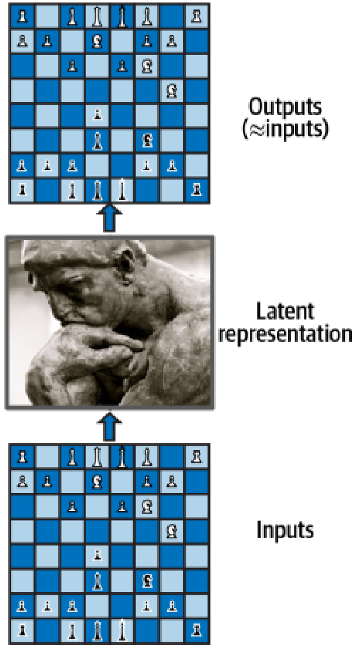
1.
Autoencoders

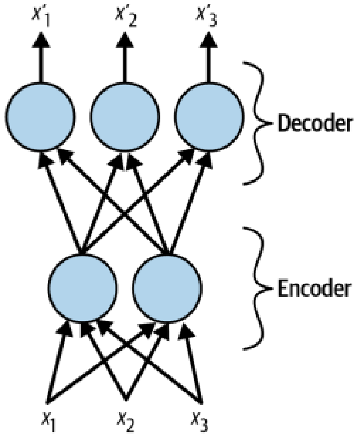
Efficient Data Representations

- Which of the following sequences do you find the easiest to memorize?
  - 40, 27, 25, 36, 81, 57, 10, 73, 19, 68
  - 50, 48, 46, 44, 42, 40, 38, 36, 34, 32, 30, 28, 26, 24, 22, 20, 18, 16, 14
- The relationship between memory, perception, and pattern matching was famously studied by William Chase and Herbert Simon in the early 1970s.
  - They noted that expert chess players were able to memorize the positions of all the pieces in a game by looking at the board for just five seconds.
  - This was only the case when the pieces were placed in realistic positions (from actual games), not when the pieces were placed randomly.
  - Chess experts don’t have a much better memory than you and I; they just see chess patterns more easily, thanks to their experience with the game.
  - **Noticing patterns helps them store information efficiently.**

Autoencoder Architecture

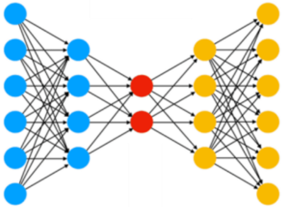
- An autoencoder consists of two parts:
  - an *encoder* (or *recognition network*) that converts the inputs to a latent representation.
  - a *decoder* (or *generative network*) that converts the internal representation to the outputs.
- The outputs are often called the *reconstructions*.





- The cost function contains a *reconstruction loss* that penalizes the model when the reconstructions are different from the inputs.

Undercomplete Autoencoders

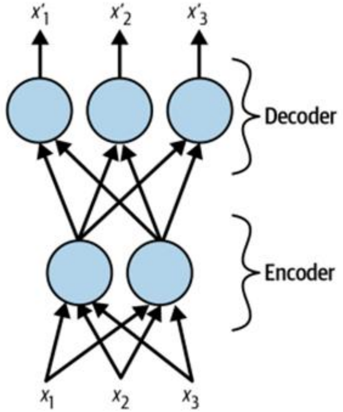


- Because the internal representation has a lower dimensionality than the input data, the autoencoder is said to be *undercomplete*.
  - It cannot trivially copy its inputs to the codings, yet it must find a way to output a copy of its inputs.
  - It is forced to learn the most important features in the input data and drop the unimportant ones.
- If the autoencoder uses only linear activations and the cost function is the mean squared error (MSE), then it ends up performing principal component analysis (PCA).
- An autoencoder performs a form of self-supervised learning, since it is based on a supervised learning technique with automatically generated labels (in this case simply equal to the inputs).

Performing PCA with Autoencoder

- Let's builds a linear undercomplete autoencoder to perform PCA on a 3D dataset, projecting it to 2D:

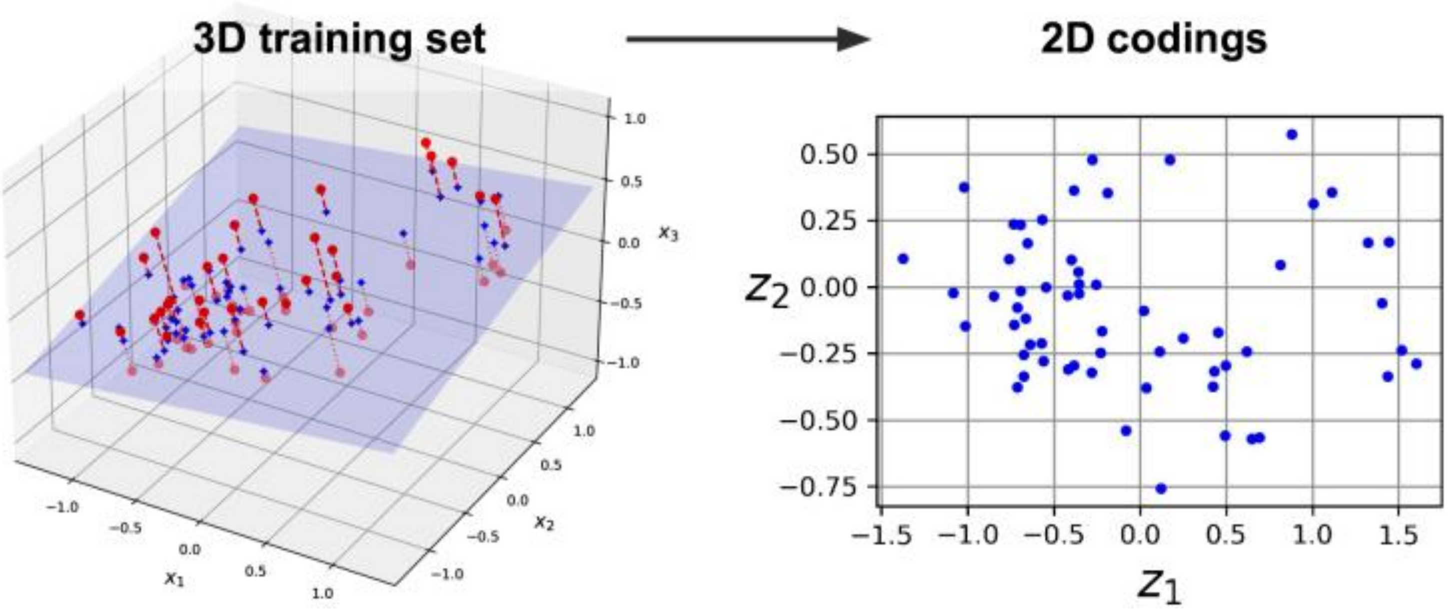
```
import tensorflow as tf

encoder = tf.keras.Sequential([tf.keras.layers.Dense(2)])
decoder = tf.keras.Sequential([tf.keras.layers.Dense(3)])
autoencoder = tf.keras.Sequential([encoder, decoder])

optimizer = tf.keras.optimizers.SGD(learning_rate=0.5)
autoencoder.compile(loss="mse", optimizer=optimizer)
```



- Train the model on the same 3D dataset we generated in Chapter 8 for PCA :

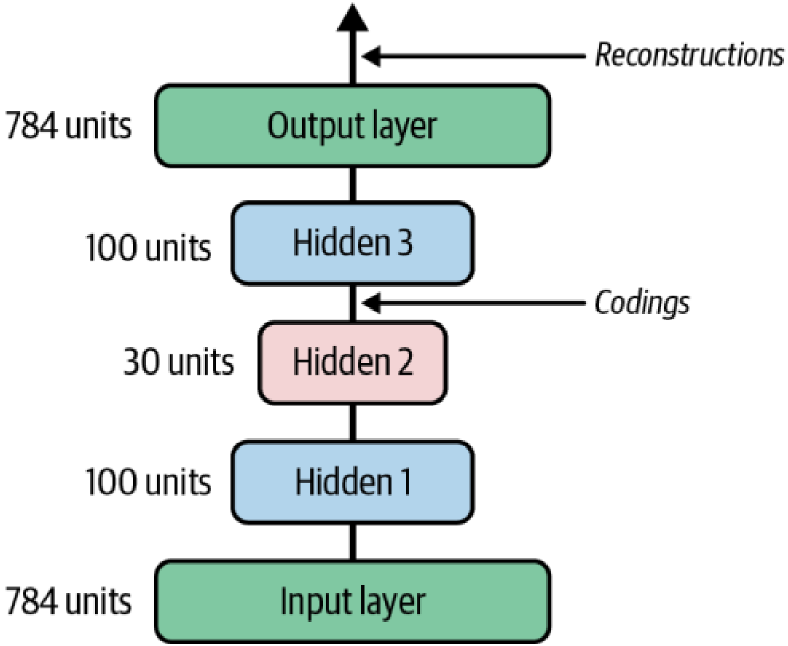
```
X_train = [...] # generate a 3D dataset, like in Chapter 8
history = autoencoder.fit(X_train, X_train, epochs=500, verbose=False)
codings = encoder.predict(X_train)
```

Performing PCA with Autoencoder



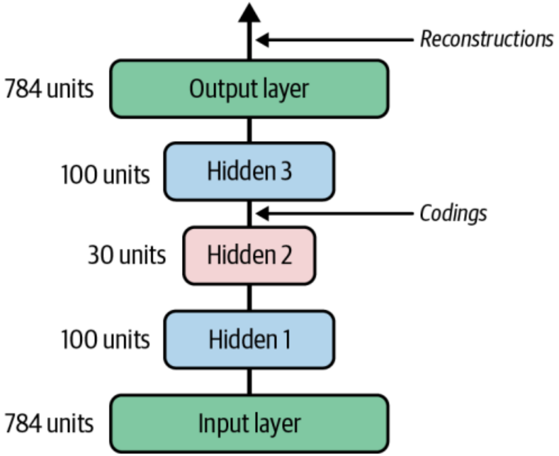
Stacked Autoencoders

- Autoencoders can have multiple hidden layers. In this case they are called *stacked autoencoders* (or *deep autoencoders*).



Implementing a Stacked Autoencoder

```
stacked_encoder = tf.keras.Sequential([
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(30, activation="relu"),
])
stacked_decoder = tf.keras.Sequential([
    tf.keras.layers.Dense(100, activation="relu"),
    tf.keras.layers.Dense(28 * 28),
    tf.keras.layers.Reshape([28, 28])
])
stacked_ae = tf.keras.Sequential([stacked_encoder, stacked_decoder])

stacked_ae.compile(loss="mse", optimizer="nadam")
history = stacked_ae.fit(X_train, X_train, epochs=20,
                         validation_data=(X_valid, X_valid))
```



Visualizing the Reconstructions

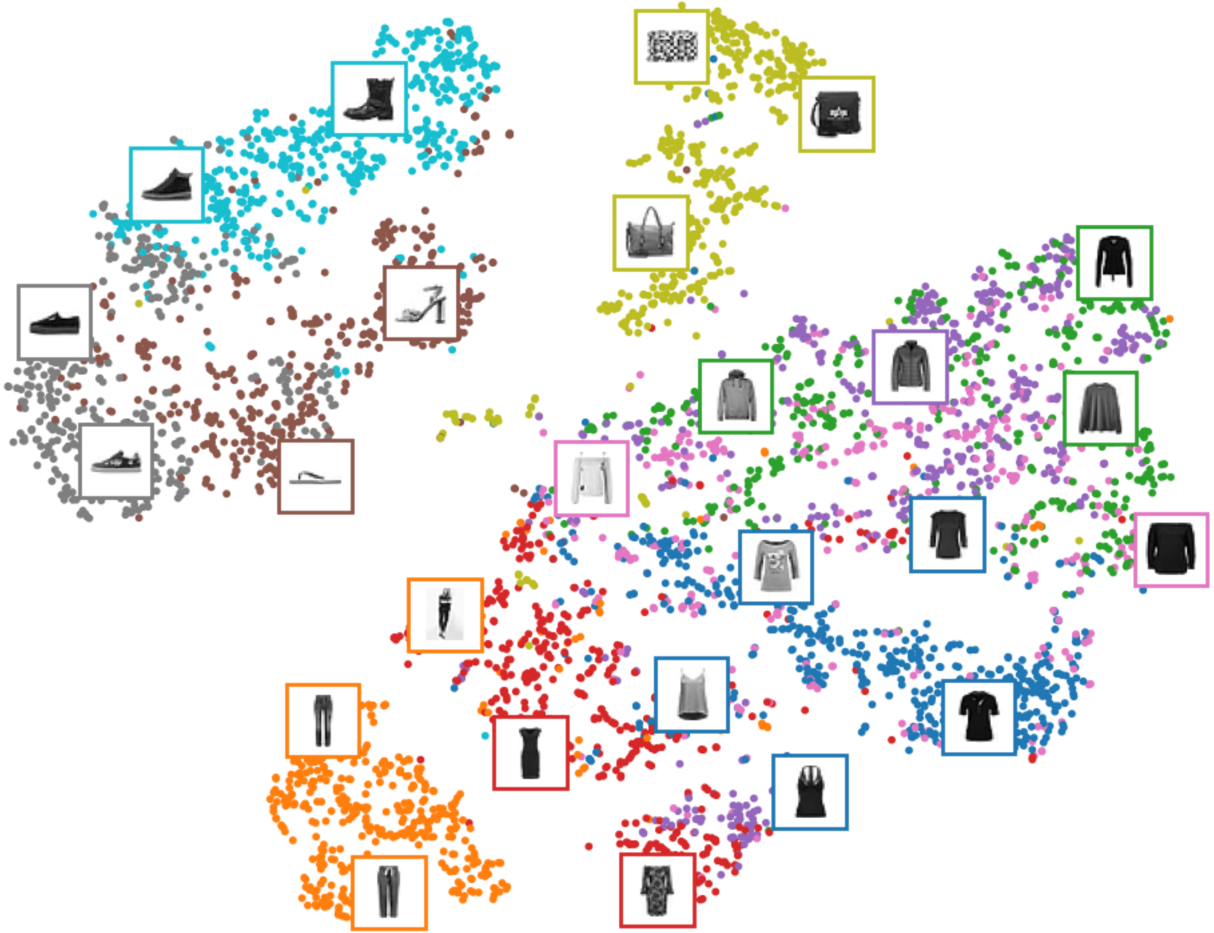
Visualizing the Fashion MNIST Dataset

- We can use the trained stacked autoencoder to reduce the dataset's dimensionality.
  - For visualization, this does not give great results compared to other dimensionality reduction algorithms.
- The big advantage of autoencoders is that they can handle large datasets with many instances and many features.
  - We can use an autoencoder to reduce the dimensionality down to a reasonable level, then use another dimensionality reduction algorithm for visualization.
- We use the encoder from our stacked autoencoder to reduce the dimensionality of Fashion MNIST dataset to 30, then we'll use the t-SNE algorithm to reduce the dimensionality down to 2 for visualization.

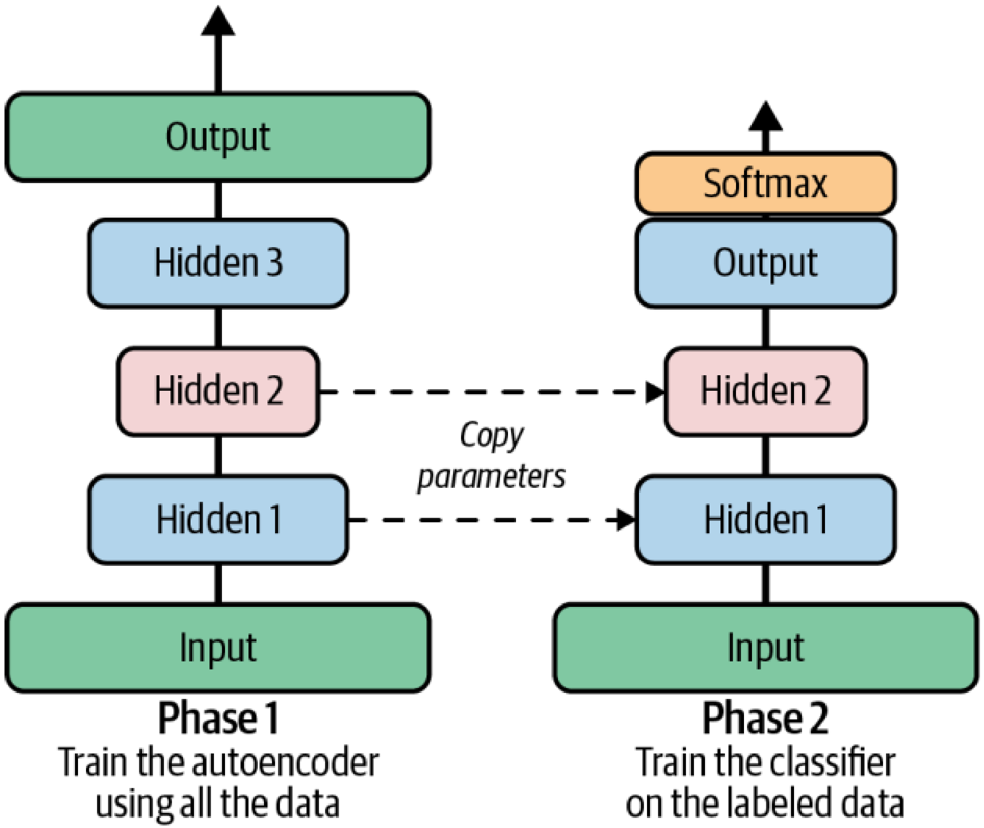
Visualizing the Fashion MNIST Dataset

- We can use the trained stacked autoencoder to reduce the dataset's dimensionality.
  - For visualization, this does not give great results compared to other dimensionality reduction algorithms.
- The big advantage of autoencoders is that they can handle large datasets with many instances and many features.
  - We can use an autoencoder to reduce the dimensionality down to a reasonable level, then use another dimensionality reduction algorithm for visualization.

```
from sklearn.manifold import TSNE

X_valid_compressed = stacked_encoder.predict(X_valid)
tsne = TSNE(init="pca", learning_rate="auto", random_state=42)
X_valid_2D = tsne.fit_transform(X_valid_compressed)
```



Autoencoders for Unsupervised Pretraining



Tying Weights

- When an autoencoder is symmetrical, a common technique is to *tie* the weights of the decoder layers to the weights of the encoder layers.
  - This halves the number of weights in the model, speeding up training and limiting the risk of overfitting.

- If the autoencoder has a total of N layers (not counting the input layer), and $\boldsymbol{W}_L$ represents the connection weights of the L-th layer, then the decoder layer weights can be defined as:

$$\boldsymbol{W}_L = \boldsymbol{W}_{N-L+1}^{\intercal}$$

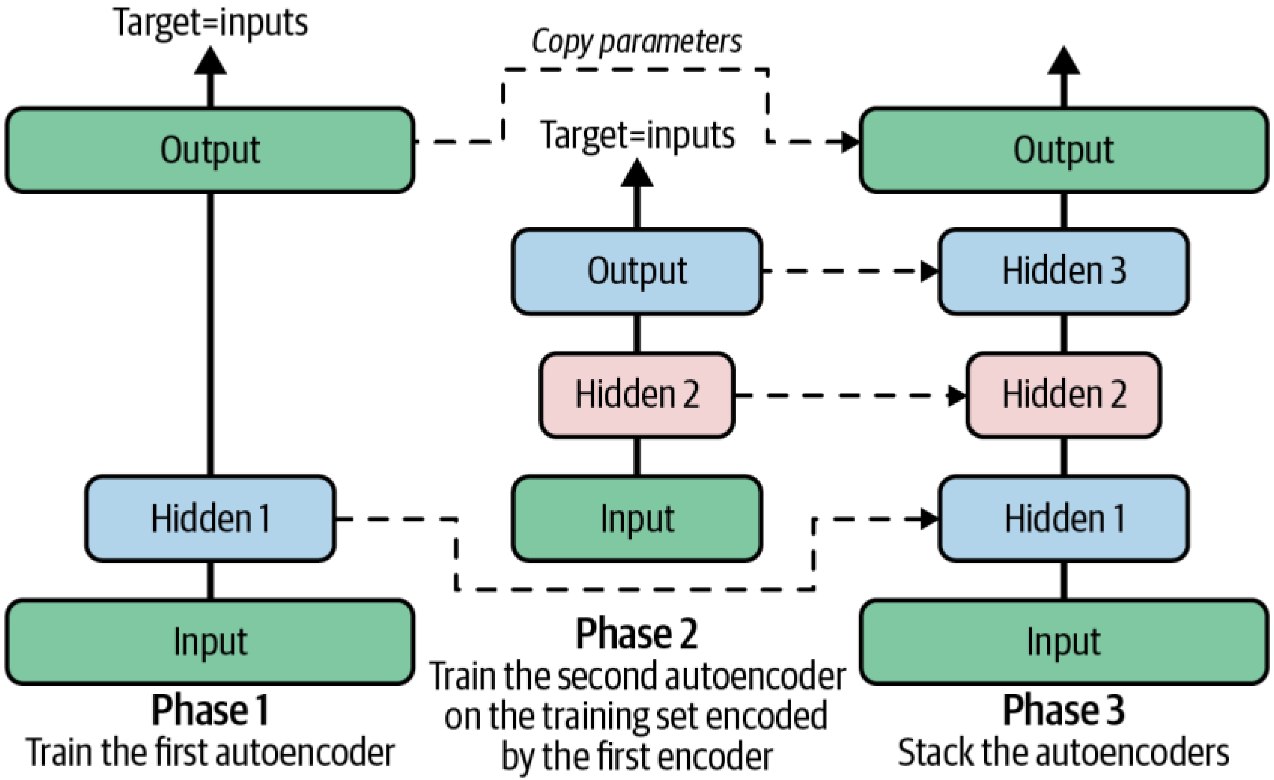
Building a Custom Layer

```
class DenseTranspose(tf.keras.layers.Layer):
    def __init__(self, dense, activation=None, **kwargs):
        super().__init__(**kwargs)
        self.dense = dense
        self.activation = tf.keras.activations.get(activation)

    def build(self, batch_input_shape):
        self.biases = self.add_weight(name="bias",
                                      shape=self.dense.input_shape[-1],
                                      initializer="zeros")
        super().build(batch_input_shape)

    def call(self, inputs):
        Z = tf.matmul(inputs, self.dense.weights[0], transpose_b=True)
        return self.activation(Z + self.biases)
```

Tying Weights

```
dense_1 = tf.keras.layers.Dense(100, activation="relu")
dense_2 = tf.keras.layers.Dense(30, activation="relu")

tied_encoder = tf.keras.Sequential([
    tf.keras.layers.Flatten(),
    dense_1,
    dense_2
])

tied_decoder = tf.keras.Sequential([
    DenseTranspose(dense_2, activation="relu"),
    DenseTranspose(dense_1),
    tf.keras.layers.Reshape([28, 28])
])

tied_ae = tf.keras.Sequential([tied_encoder, tied_decoder])
```

Greedy Layerwise Training



Convolutional Autoencoders

- If you want to build an autoencoder for images (e.g., for unsupervised pretraining or dimensionality reduction), you will need to build a *convolutional autoencoder*.

- The encoder is a regular CNN composed of convolutional layers and pooling layers.
  - It typically reduces the spatial dimensionality of the inputs (i.e., height and width) while increasing the depth (i.e., the number of feature maps).

- The decoder must do the reverse (upscale the image and reduce its depth back to the original dimensions), and for this you can use transpose convolutional layers.

Convolutional Autoencoders

```
conv_encoder = tf.keras.Sequential([
    tf.keras.layers.Reshape([28, 28, 1]),
    tf.keras.layers.Conv2D(16, 3, padding="same", activation="relu"),
    tf.keras.layers.MaxPool2D(pool_size=2),  # output: 14 × 14 x 16
    tf.keras.layers.Conv2D(32, 3, padding="same", activation="relu"),
    tf.keras.layers.MaxPool2D(pool_size=2),  # output: 7 × 7 x 32
    tf.keras.layers.Conv2D(64, 3, padding="same", activation="relu"),
    tf.keras.layers.MaxPool2D(pool_size=2),  # output: 3 × 3 x 64
    tf.keras.layers.Conv2D(30, 3, padding="same", activation="relu"),
    tf.keras.layers.GlobalAvgPool2D()  # output: 30
])
conv_decoder = tf.keras.Sequential([
    tf.keras.layers.Dense(3 * 3 * 16),
    tf.keras.layers.Reshape((3, 3, 16)),
    tf.keras.layers.Conv2DTranspose(32, 3, strides=2, activation="relu"),
    tf.keras.layers.Conv2DTranspose(16, 3, strides=2, padding="same", activation="relu"),
    tf.keras.layers.Conv2DTranspose(1, 3, strides=2, padding="same"),
    tf.keras.layers.Reshape([28, 28])
])
conv_ae = tf.keras.Sequential([conv_encoder, conv_decoder])
```