CHAPTER 3

Variational Autoencoders

Chapter Goals

In this chapter you will:

- Learn how the architectural design of autoencoders makes them perfectly suited to generative modeling.
- Build and train an autoencoder from scratch using Keras.
- Use autoencoders to generate new images, but understand the limitations of this approach.
- Learn about the architecture of the variational autoencoder and how it solves many of the problems associated with standard autoencoders.
- Build a variational autoencoder from scratch using Keras.
- Use variational autoencoders to generate new images.
- Use variational autoencoders to manipulate generated images using latent space arithmetic.

In 2013, Diederik P. Kingma and Max Welling published a paper that laid the foundations for a type of neural network known as a *variational autoencoder* (VAE).[1] This is now one of the most fundamental and well-known deep learning architectures for generative modeling and an excellent place to start our journey into generative deep learning.

In this chapter, we shall start by building a standard autoencoder and then see how we can extend this framework to develop a variational autoencoder. Along the way, we will pick apart both types of models, to understand how they work at a granular level. By the end of the chapter you should have a complete understanding of how to

build and manipulate autoencoder-based models and, in particular, how to build a variational autoencoder from scratch to generate images based on your own dataset.

Introduction

Let's start with a simple story that will help to explain the fundamental problem that an autoencoder is trying to solve.

Brian, the Stitch, and the Wardrobe

Imagine that on the floor in front of you is a pile of all the clothing you own—trousers, tops, shoes, and coats, all of different styles. Your stylist, Brian, is becoming increasingly frustrated with how long it takes him to find the items you require, so he devises a clever plan.

He tells you to organize your clothes into a wardrobe that is infinitely high and wide (Figure 3-1). When you want to request a particular item, you simply need to tell Brian its location and he will sew the item from scratch using his trusty sewing machine. It soon becomes obvious that you will need to place similar items near to each other, so that Brian can accurately re-create each item given only its location.

Figure 3-1. A man standing in front of an infinite 2D wardrobe (created with Midjourney)

After several weeks of practice, you and Brian have adjusted to each other's understandings of the wardrobe layout. It is now possible for you to tell Brian the location of any item of clothing that you desire, and he can accurately sew it from scratch!

This gives you an idea—what would happen if you gave Brian a wardrobe location that was empty? To your amazement, you find that Brian is able to generate entirely

new items of clothing that haven't existed before! The process isn't perfect, but you now have limitless options for generating new clothing, just by picking an empty location in the infinite wardrobe and letting Brian work his magic with the sewing machine.

Let's now explore how this story relates to building autoencoders.

Autoencoders

A diagram of the process described by the story is shown in Figure 3-2. You play the part of the *encoder*, moving each item of clothing to a location in the wardrobe. This process is called *encoding*. Brian plays the part of the *decoder*, taking a location in the wardrobe and attempting to re-create the item. This process is called *decoding*.

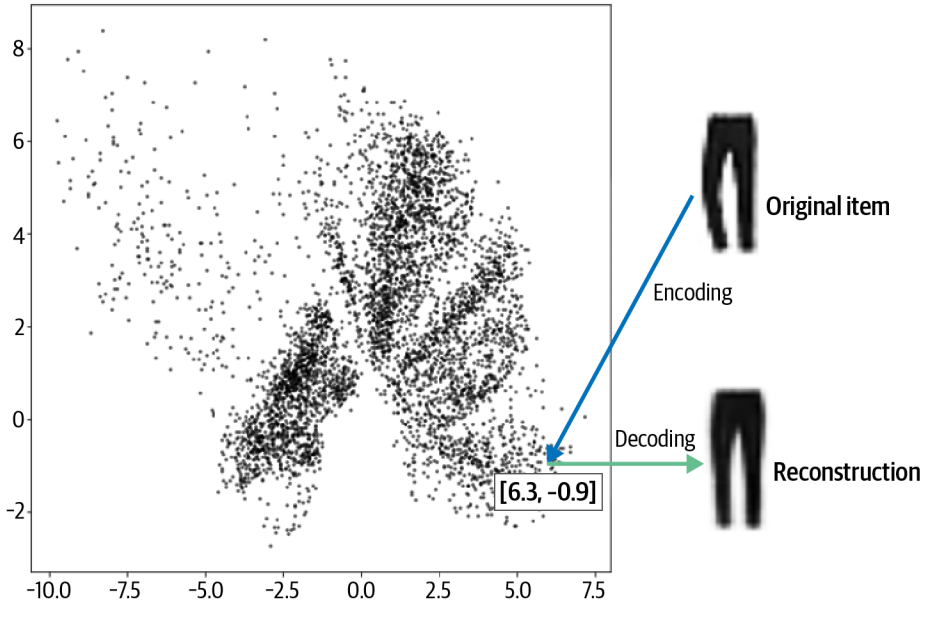

Figure 3-2. Items of clothing in the infinite wardrobe—each black dot represents an item of clothing

Each location in the wardrobe is represented by two numbers (i.e., a 2D vector). For example, the trousers in Figure 3-2 are encoded to the point [6.3, –0.9]. This vector is also known as an *embedding* because the encoder attempts to embed as much information into it as possible, so that the decoder can produce an accurate reconstruction.

An *autoencoder* is simply a neural network that is trained to perform the task of encoding and decoding an item, such that the output from this process is as close to the original item as possible. Crucially, it can be used as a generative model, because we can decode any point in the 2D space that we want (in particular, those that are not embeddings of original items) to produce a novel item of clothing.

Let's now see how we can build an autoencoder using Keras and apply it to a real dataset!

Running the Code for This Example

The code for this example can be found in the Jupyter notebook located at *notebooks/03_vae/01_autoencoder/autoencoder.ipynb* in the book repository.

The Fashion-MNIST Dataset

For this example, we'll be using the Fashion-MNIST dataset—a collection of grayscale images of clothing items, each of size 28 × 28 pixels. Some example images from the dataset are shown in Figure 3-3.

Figure 3-3. Examples of images from the Fashion-MNIST dataset

The dataset comes prepackaged with TensorFlow, so it can be downloaded as shown in Example 3-1.

Example 3-1. Loading the Fashion-MNIST dataset

```
from tensorflow.keras import datasets
(x_train,y_train), (x_test,y_test) = datasets.fashion_mnist.load_data()
```

These are 28 × 28 grayscale images (pixel values between 0 and 255) out of the box, which we need to preprocess to ensure that the pixel values are scaled between 0 and 1. We will also pad each image to 32 × 32 for easier manipulation of the tensor shape as it passes through the network, as shown in Example 3-2.

Example 3-2. Preprocessing the data

```
def preprocess(imgs):
    imgs = imgs.astype("float32") / 255.0
    imgs = np.pad(imgs, ((0, 0), (2, 2), (2, 2)), constant_values=0.0)
    imgs = np.expand_dims(imgs, -1)
    return imgs

x_train = preprocess(x_train)
x_test = preprocess(x_test)
```

Next, we need to understand the overall structure of an autoencoder, so that we can code it up using TensorFlow and Keras.

The Autoencoder Architecture

An *autoencoder* is a neural network made up of two parts:

- An *encoder* network that compresses high-dimensional input data such as an image into a lower-dimensional embedding vector
- A *decoder* network that decompresses a given embedding vector back to the original domain (e.g., back to an image)

A diagram of the network architecture is shown in Figure 3-4. An input image is encoded to a latent embedding vector z, which is then decoded back to the original pixel space.

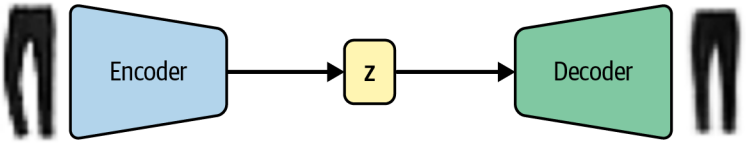

Figure 3-4. Autoencoder architecture diagram

The autoencoder is trained to reconstruct an image, after it has passed through the encoder and back out through the decoder. This may seem strange at first—why would you want to reconstruct a set of images that you already have available to you? However, as we shall see, it is the embedding space (also called the *latent space*) that is the interesting part of the autoencoder, as sampling from this space will allow us to generate new images.

Let's first define what we mean by an embedding. The embedding (z) is a compression of the original image into a lower-dimensional latent space. The idea is that by choosing any point in the latent space, we can generate novel images by passing this point through the decoder, since the decoder has learned how to convert points in the latent space into viable images.

In our example, we will embed images into a two-dimensional latent space. This will help us to visualize the latent space, since we can easily plot points in 2D. In practice, the latent space of an autoencoder will usually have more than two dimensions in order to have more freedom to capture greater nuance in the images.

Autoencoders as Denoising Models

Autoencoders can be used to clean noisy images, since the encoder learns that it is not useful to capture the position of the random noise inside the latent space in order to reconstruct the original. For tasks such as this, a 2D latent space is probably too small to encode sufficient relevant information from the input. However, as we shall see, increasing the dimensionality of the latent space quickly leads to problems if we want to use the autoencoder as a generative model.

Let's now see how to build the encoder and decoder.

The Encoder

In an autoencoder, the encoder's job is to take the input image and map it to an embedding vector in the latent space. The architecture of the encoder we will be building is shown in Table 3-1.

Table 3-1. Model summary of the encoder

Layer (type)	Output shape	Param #
InputLayer	(None, 32, 32, 1)	0
Conv2D	(None, 16, 16, 32)	320
Conv2D	(None, 8, 8, 64)	18,496
Conv2D	(None, 4, 4, 128)	73,856
Flatten	(None, 2048)	0
Dense	(None, 2)	4,098

Total params	96,770
Trainable params	96,770
Non-trainable params	0

To achieve this, we first create an `Input` layer for the image and pass this through three `Conv2D` layers in sequence, each capturing increasingly high-level features. We use a stride of 2 to halve the size of the output of each layer, while increasing the number of channels. The last convolutional layer is flattened and connected to a `Dense` layer of size 2, which represents our two-dimensional latent space.

Example 3-3 shows how to build this in Keras.

Example 3-3. The encoder

```
encoder_input = layers.Input(
    shape=(32, 32, 1), name = "encoder_input"
) ❶
x = layers.Conv2D(32, (3, 3), strides = 2, activation = 'relu', padding="same")(
    encoder_input
) ❷
x = layers.Conv2D(64, (3, 3), strides = 2, activation = 'relu', padding="same")(x)
x = layers.Conv2D(128, (3, 3), strides = 2, activation = 'relu', padding="same")(x)
shape_before_flattening = K.int_shape(x)[1:]

x = layers.Flatten()(x) ❸
encoder_output = layers.Dense(2, name="encoder_output")(x) ❹

encoder = models.Model(encoder_input, encoder_output) ❺
```

❶ Define the `Input` layer of the encoder (the image).

❷ Stack `Conv2D` layers sequentially on top of each other.

❸ Flatten the last convolutional layer to a vector.

❹ Connect this vector to the 2D embeddings with a `Dense` layer.

❺ The Keras `Model` that defines the encoder—a model that takes an input image and encodes it into a 2D embedding.

I strongly encourage you to experiment with the number of convolutional layers and filters to understand how the architecture affects the overall number of model parameters, model performance, and model runtime.

The Decoder

The decoder is a mirror image of the encoder—instead of convolutional layers, we use *convolutional transpose* layers, as shown in Table 3-2.

Table 3-2. Model summary of the decoder

Layer (type)	Output shape	Param #
InputLayer	(None, 2)	0
Dense	(None, 2048)	6,144

Layer (type)	Output shape	Param #
Reshape	(None, 4, 4, 128)	0
Conv2DTranspose	(None, 8, 8, 128)	147,584
Conv2DTranspose	(None, 16, 16, 64)	73,792
Conv2DTranspose	(None, 32, 32, 32)	18,464
Conv2D	(None, 32, 32, 1)	289

Total params	246,273
Trainable params	246,273
Non-trainable params	0

Convolutional Transpose Layers

Standard convolutional layers allow us to halve the size of an input tensor in both dimensions (height and width), by setting `strides = 2`.

The convolutional transpose layer uses the same principle as a standard convolutional layer (passing a filter across the image), but is different in that setting `strides = 2` *doubles* the size of the input tensor in both dimensions.

In a convolutional transpose layer, the `strides` parameter determines the internal zero padding between pixels in the image, as shown in Figure 3-5. Here, a $3 \times 3 \times 1$ filter (gray) is being passed across a $3 \times 3 \times 1$ image (blue) with `strides = 2`, to produce a $6 \times 6 \times 1$ output tensor (green).

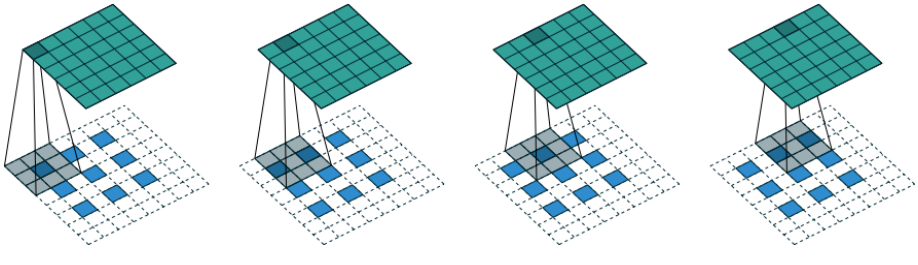

Figure 3-5. A convolutional transpose layer example (source: Dumoulin and Visin, 2018)[2]

In Keras, the `Conv2DTranspose` layer allows us to perform convolutional transpose operations on tensors. By stacking these layers, we can gradually expand the size of each layer, using strides of 2, until we get back to the original image dimension of 32×32.

Example 3-4 shows how we build the decoder in Keras.

Example 3-4. The decoder

```
decoder_input = layers.Input(shape=(2,), name="decoder_input") ❶
x = layers.Dense(np.prod(shape_before_flattening))(decoder_input) ❷
x = layers.Reshape(shape_before_flattening)(x) ❸
x = layers.Conv2DTranspose(
    128, (3, 3), strides=2, activation = 'relu', padding="same"
)(x) ❹
x = layers.Conv2DTranspose(
    64, (3, 3), strides=2, activation = 'relu', padding="same"
)(x)
x = layers.Conv2DTranspose(
    32, (3, 3), strides=2, activation = 'relu', padding="same"
)(x)
decoder_output = layers.Conv2D(
    1,
    (3, 3),
    strides = 1,
    activation="sigmoid",
    padding="same",
    name="decoder_output"
)(x)

decoder = models.Model(decoder_input, decoder_output) ❺
```

❶ Define the `Input` layer of the decoder (the embedding).

❷ Connect the input to a `Dense` layer.

❸ `Reshape` this vector into a tensor that can be fed as input into the first `Conv2DTranspose` layer.

❹ Stack `Conv2DTranspose` layers on top of each other.

❺ The Keras `Model` that defines the decoder—a model that takes an embedding in the latent space and decodes it into the original image domain.

Joining the Encoder to the Decoder

To train the encoder and decoder simultaneously, we need to define a model that will represent the flow of an image through the encoder and back out through the decoder. Luckily, Keras makes it extremely easy to do this, as you can see in Example 3-5. Notice the way in which we specify that the output from the autoencoder is simply the output from the encoder after it has been passed through the decoder.

Example 3-5. The full autoencoder

```
autoencoder = Model(encoder_input, decoder(encoder_output)) ❶
```

❶ The Keras `Model` that defines the full autoencoder—a model that takes an image and passes it through the encoder and back out through the decoder to generate a reconstruction of the original image.

Now that we've defined our model, we just need to compile it with a loss function and optimizer, as shown in Example 3-6. The loss function is usually chosen to be either the root mean squared error (RMSE) or binary cross-entropy between the individual pixels of the original image and the reconstruction.

Example 3-6. Compiling the autoencoder

```
# Compile the autoencoder
autoencoder.compile(optimizer="adam", loss="binary_crossentropy")
```

Choosing the Loss Function

Optimizing for RMSE means that your generated output will be symmetrically distributed around the average pixel values (because an overestimation is penalized equivalently to an underestimation).

On the other hand, binary cross-entropy loss is asymmetrical—it penalizes errors toward the extremes more heavily than errors toward the center. For example, if the true pixel value is high (say 0.7), then generating a pixel with value 0.8 is penalized more heavily than generating a pixel with value 0.6. If the true pixel value is low (say 0.3), then generating a pixel with value 0.2 is penalized more heavily than generating a pixel with value 0.4.

This has the effect of binary cross-entropy loss producing slightly blurrier images than RMSE loss (as it tends to push predictions toward 0.5), but sometimes this is desirable as RMSE can lead to obviously pixelized edges.

There is no right or wrong choice—you should choose whichever works best for your use case after experimentation.

We can now train the autoencoder by passing in the input images as both the input and output, as shown in Example 3-7.

Example 3-7. Training the autoencoder

```
autoencoder.fit(
    x_train,
    x_train,
    epochs=5,
    batch_size=100,
    shuffle=True,
    validation_data=(x_test, x_test),
)
```

Now that our autoencoder is trained, the first thing we need to check is that it is able to accurately reconstruct the input images.

Reconstructing Images

We can test the ability to reconstruct images by passing images from the test set through the autoencoder and comparing the output to the original images. The code for this is shown in Example 3-8.

Example 3-8. Reconstructing images using the autoencoder

```
example_images = x_test[:5000]
predictions = autoencoder.predict(example_images)
```

In Figure 3-6 you can see some examples of original images (top row), the 2D vectors after encoding, and the reconstructed items after decoding (bottom row).

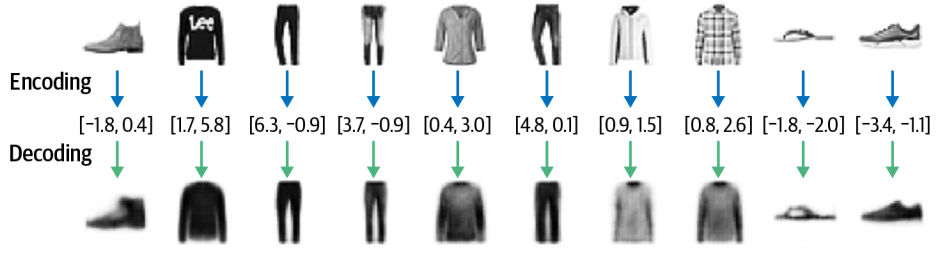

Figure 3-6. Examples of encoding and decoding items of clothing

Notice how the reconstruction isn't perfect—there are still some details of the original images that aren't captured by the decoding process, such as logos. This is because by reducing each image to just two numbers, we naturally lose some information.

Let's now investigate how the encoder is representing images in the latent space.

Visualizing the Latent Space

We can visualize how images are embedded into the latent space by passing the test set through the encoder and plotting the resulting embeddings, as shown in Example 3-9.

Example 3-9. Embedding images using the encoder

```
embeddings = encoder.predict(example_images)

plt.figure(figsize=(8, 8))
plt.scatter(embeddings[:, 0], embeddings[:, 1], c="black", alpha=0.5, s=3)
plt.show()
```

The resulting plot is the scatter plot shown in Figure 3-2—each black point represents an image that has been embedded into the latent space.

In order to better understand how this latent space is structured, we can make use of the labels that come with the Fashion-MNIST dataset, describing the type of item in each image. There are 10 groups altogether, shown in Table 3-3.

Table 3-3. The Fashion-MNIST labels

ID	Clothing label
0	T-shirt/top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle boot

We can color each point based on the label of the corresponding image to produce the plot in Figure 3-7. Now the structure becomes very clear! Even though the clothing labels were never shown to the model during training, the autoencoder has naturally grouped items that look alike into the same parts of the latent space. For example, the dark blue cloud of points in the bottom-right corner of the latent space are all different images of trousers and the red cloud of points toward the center are all ankle boots.

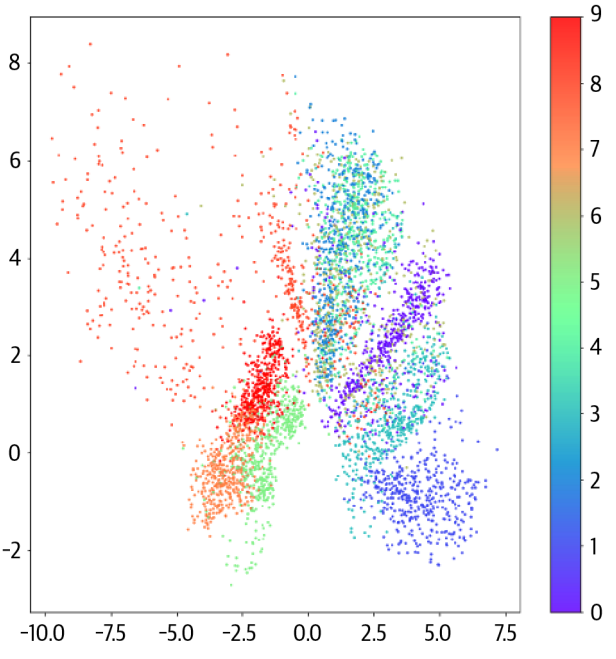

Figure 3-7. Plot of the latent space, colored by clothing label

Generating New Images

We can generate novel images by sampling some points in the latent space and using the decoder to convert these back into pixel space, as shown in Example 3-10.

Example 3-10. Generating novel images using the decoder

```
mins, maxs = np.min(embeddings, axis=0), np.max(embeddings, axis=0)
sample = np.random.uniform(mins, maxs, size=(18, 2))
reconstructions = decoder.predict(sample)
```

Some examples of generated images are shown in Figure 3-8, alongside their embeddings in the latent space.

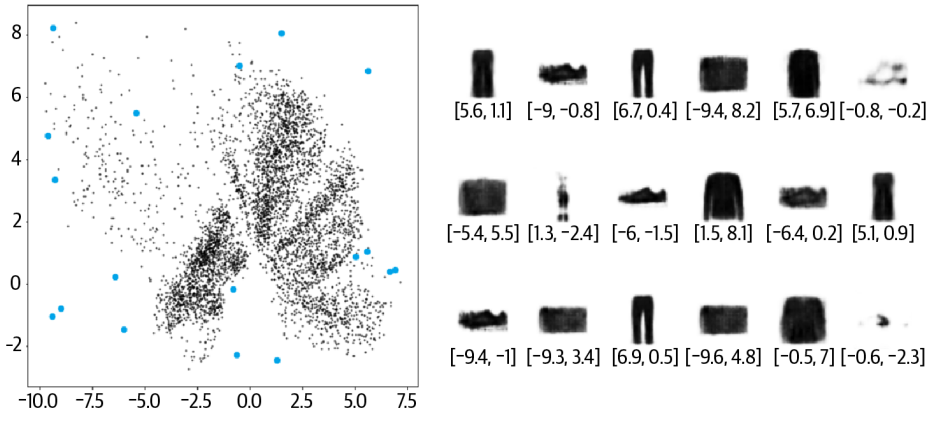

Figure 3-8. Generated items of clothing

Each blue dot maps to one of the images shown on the right of the diagram, with the embedding vector shown underneath. Notice how some of the generated items are more realistic than others. Why is this?

To answer this, let's first make a few observations about the overall distribution of points in the latent space, referring back to Figure 3-7:

- Some clothing items are represented over a very small area and others over a much larger area.
- The distribution is not symmetrical about the point (0, 0), or bounded. For example, there are far more points with positive y-axis values than negative, and some points even extend to a y-axis value > 8.
- There are large gaps between colors containing few points.

These observations actually make sampling from the latent space quite challenging. If we overlay the latent space with images of decoded points on a grid, as shown in Figure 3-9, we can begin to understand why the decoder may not always generate images to a satisfactory standard.

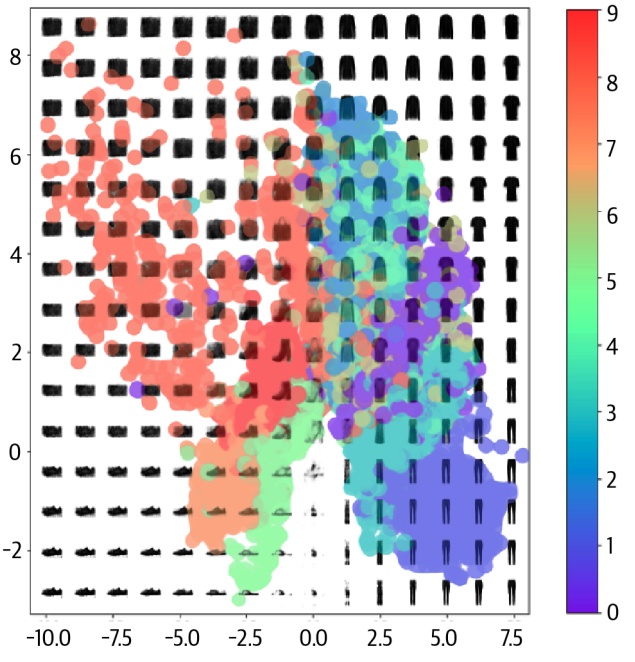

Figure 3-9. A grid of decoded embeddings, overlaid with the embeddings from the original images in the dataset, colored by item type

Firstly, we can see that if we pick points uniformly in a bounded space that we define, we're more likely to sample something that decodes to look like a bag (ID 8) than an ankle boot (ID 9) because the part of the latent space carved out for bags (orange) is larger than the ankle boot area (red).

Secondly, it is not obvious how we should go about choosing a *random* point in the latent space, since the distribution of these points is undefined. Technically, we would be justified in choosing any point in the 2D plane! It's not even guaranteed that points will be centered around (0, 0). This makes sampling from our latent space problematic.

Lastly, we can see holes in the latent space where none of the original images are encoded. For example, there are large white spaces at the edges of the domain—the autoencoder has no reason to ensure that points here are decoded to recognizable clothing items as very few images in the training set are encoded here.

Even points that are central may not be decoded into well-formed images. This is because the autoencoder is not forced to ensure that the space is continuous. For example, even though the point (–1, –1) might be decoded to give a satisfactory

image of a sandal, there is no mechanism in place to ensure that the point (–1.1, –1.1) also produces a satisfactory image of a sandal.

In two dimensions this issue is subtle; the autoencoder only has a small number of dimensions to work with, so naturally it has to squash clothing groups together, resulting in the space between clothing groups being relatively small. However, as we start to use more dimensions in the latent space to generate more complex images such as faces, this problem becomes even more apparent. If we give the autoencoder free rein over how it uses the latent space to encode images, there will be huge gaps between groups of similar points with no incentive for the spaces in between to generate well-formed images.

In order to solve these three problems, we need to convert our autoencoder into a *variational autoencoder*.

Variational Autoencoders

To explain, let's revisit the infinite wardrobe and make a few changes…

Revisiting the Infinite Wardrobe

Suppose now, instead of placing every item of clothing at a single point in the wardrobe, you decide to allocate a general area where the item is more likely to be found. You reason that this more relaxed approach to item location will help to solve the current issue around local discontinuities in the wardrobe.

Also, in order to ensure you do not become too careless with the new placement system, you agree with Brian that you will try to place the center of each item's area as close to the middle of the wardrobe as possible and that deviation of the item from the center should be as close to one meter as possible (not smaller and not larger). The further you stray from this rule, the more you have to pay Brian as your stylist.

After several months of operating with these two simple changes, you step back and admire the new wardrobe layout, alongside some examples of new clothing items that Brian has generated. Much better! There is plenty of diversity in the generated items, and this time there are no examples of poor-quality garments. It seems the two changes have made all the difference!

Let's now try to understand what we need to do to our autoencoder model to convert it into a variational autoencoder and thus make it a more sophisticated generative model.

The two parts that we need to change are the encoder and the loss function.

The Encoder

In an autoencoder, each image is mapped directly to one point in the latent space. In a variational autoencoder, each image is instead mapped to a multivariate normal distribution around a point in the latent space, as shown in Figure 3-10.

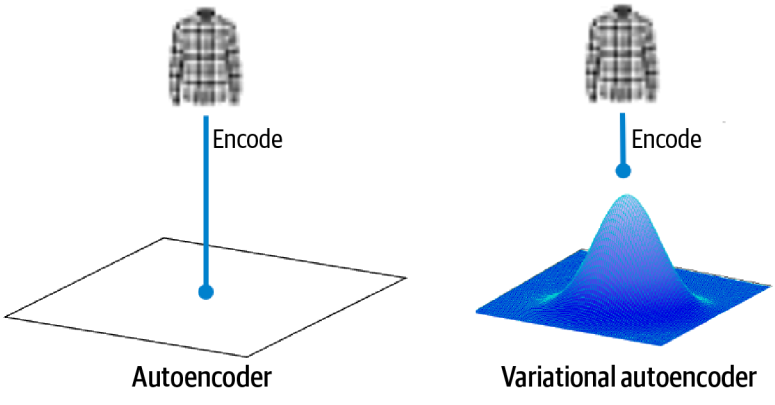

Figure 3-10. The difference between the encoders in an autoencoder and a variational autoencoder

The Multivariate Normal Distribution

A *normal distribution* (or *Gaussian distribution*) $\mathcal{N}(\mu, \sigma)$ is a probability distribution characterized by a distinctive *bell curve* shape, defined by two variables: the *mean* (μ) and the *variance* (σ^2). The *standard deviation* (\(\sigma\)) is the square root of the variance.

The probability density function of the normal distribution in one dimension is:

$$f\left(x \mid \mu, \sigma^2\right) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Figure 3-11 shows several normal distributions in one dimension, for different values of the mean and variance. The red curve is the *standard normal* (or *unit normal*) $\mathcal{N}(0, 1)$—the normal distribution with mean equal to 0 and variance equal to 1.

We can sample a point z from a normal distribution with mean μ and standard deviation σ using the following equation:

$$z = \mu + \sigma\epsilon$$

where ϵ is sampled from a standard normal distribution.

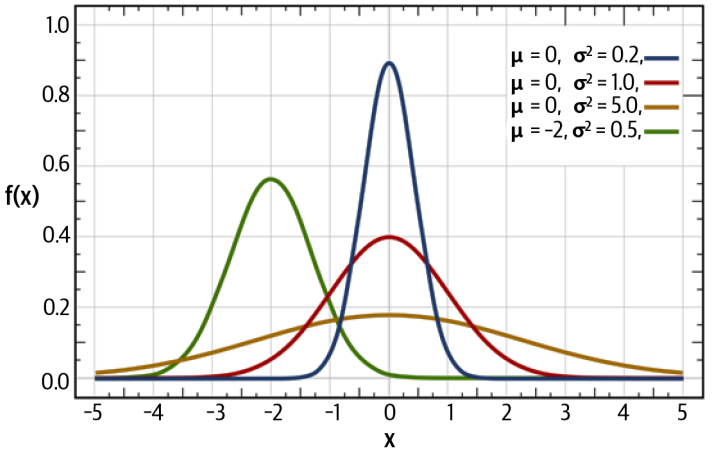

Figure 3-11. The normal distribution in one dimension (source: Wikipedia)

The concept of a normal distribution extends to more than one dimension—the probability density function for a *multivariate normal distribution* (or *multivariate Gaussian distribution*) $\mathcal{N}(\mu, \Sigma)$ in k dimensions with mean vector μ and symmetric covariance matrix Σ is as follows:

$$f(x_1, ..., x_k) = \frac{\exp\left(-\frac{1}{2}(\mathbf{x}-\mu)^{\mathrm{T}}\Sigma^{-1}(\mathbf{x}-\mu)\right)}{\sqrt{(2\pi)^k|\Sigma|}}$$

In this book, we will typically be using *isotropic* multivariate normal distributions, where the covariance matrix is diagonal. This means that the distribution is independent in each dimension (i.e., we can sample a vector where each element is normally distributed with independent mean and variance). This is the case for the multivariate normal distribution that we will use in our variational autoencoder.

A *multivariate standard normal distribution* $\mathcal{N}(0, \mathbf{I})$ is a multivariate distribution with a zero-valued mean vector and identity covariance matrix.

Normal Versus Gaussian

In this book, the terms *normal* and *Gaussian* are used interchangeably and the isotropic and multivariate nature of the distribution is usually implied. For example, "we sample from a Gaussian distribution" can be interpreted to mean "we sample from an isotropic, multivariate Gaussian distribution."

The encoder only needs to map each input to a mean vector and a variance vector and does not need to worry about covariance between dimensions. Variational autoencoders assume that there is no correlation between dimensions in the latent space.

Variance values are always positive, so we actually choose to map to the *logarithm* of the variance, as this can take any real number in the range $(-\infty, \infty)$. This way we can use a neural network as the encoder to perform the mapping from the input image to the mean and log variance vectors.

To summarize, the encoder will take each input image and encode it to two vectors that together define a multivariate normal distribution in the latent space:

`z_mean`
: The mean point of the distribution

`z_log_var`
: The logarithm of the variance of each dimension

We can sample a point `z` from the distribution defined by these values using the following equation:

```
z = z_mean + z_sigma * epsilon
```

where:

```
z_sigma = exp(z_log_var * 0.5)
epsilon ~ N(0,I)
```

The derivation of the relationship between `z_sigma` (σ) and `z_log_var` ($\log\left(\sigma^2\right)$) is as follows:

$$\sigma = \exp\left(\log\left(\sigma\right)\right) = \exp\left(2\log\left(\sigma\right)/2\right) = \exp\left(\log\left(\sigma^2\right)/2\right)$$

The decoder of a variational autoencoder is identical to the decoder of a plain autoencoder, giving the overall architecture shown in Figure 3-12.

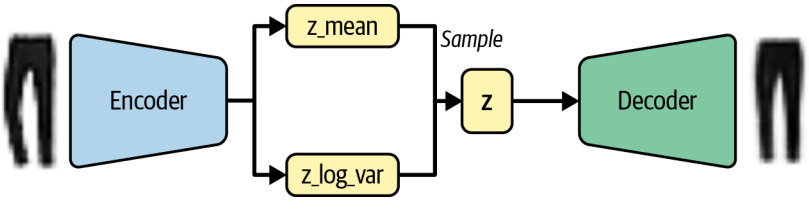

Figure 3-12. VAE architecture diagram

Why does this small change to the encoder help?

Previously, we saw that there was no requirement for the latent space to be continuous—even if the point (–2, 2) decodes to a well-formed image of a sandal, there's no requirement for (–2.1, 2.1) to look similar. Now, since we are sampling a random point from an area around `z_mean`, the decoder must ensure that all points in the same neighborhood produce very similar images when decoded, so that the reconstruction loss remains small. This is a very nice property that ensures that even when we choose a point in the latent space that has never been seen by the decoder, it is likely to decode to an image that is well formed.

Building the VAE encoder

Let's now see how we build this new version of the encoder in Keras.

Running the Code for This Example

The code for this example can be found in the Jupyter notebook located at *notebooks/03_vae/02_vae_fashion/vae_fashion.ipynb* in the book repository.

The code has been adapted from the excellent VAE tutorial created by Francois Chollet, available on the Keras website.

First, we need to create a new type of `Sampling` layer that will allow us to sample from the distribution defined by `z_mean` and `z_log_var`, as shown in Example 3-11.

Example 3-11. The `Sampling` layer

```
class Sampling(layers.Layer): ❶
    def call(self, inputs):
        z_mean, z_log_var = inputs
        batch = tf.shape(z_mean)[0]
        dim = tf.shape(z_mean)[1]
        epsilon = K.random_normal(shape=(batch, dim))
        return z_mean + tf.exp(0.5 * z_log_var) * epsilon ❷
```

❶ We create a new layer by subclassing the Keras base `Layer` class (see the "Subclassing the Layer Class" sidebar).

❷ We use the reparameterization trick (see "The Reparameterization Trick" sidebar) to build a sample from the normal distribution parameterized by `z_mean` and `z_log_var`.

Subclassing the Layer Class

You can create new layers in Keras by subclassing the abstract `Layer` class and defining the `call` method, which describes how a tensor is transformed by the layer.

For example, in the variational autoencoder, we can create a `Sampling` layer that can handle the sampling of `z` from a normal distribution with parameters defined by `z_mean` and `z_log_var`.

This is useful when you want to apply a transformation to a tensor that isn't already included as one of the out-of-the-box Keras layer types.

The Reparameterization Trick

Rather than sample directly from a normal distribution with parameters `z_mean` and `z_log_var`, we can sample `epsilon` from a standard normal and then manually adjust the sample to have the correct mean and variance.

This is known as the *reparameterization trick*, and it's important as it means gradients can backpropagate freely through the layer. By keeping all of the randomness of the layer contained within the variable `epsilon`, the partial derivative of the layer output with respect to its input can be shown to be deterministic (i.e., independent of the random `epsilon`), which is essential for backpropagation through the layer to be possible.

The complete code for the encoder, including the new `Sampling` layer, is shown in Example 3-12.

Example 3-12. The encoder

```
encoder_input = layers.Input(
    shape=(32, 32, 1), name="encoder_input"
)
x = layers.Conv2D(32, (3, 3), strides=2, activation="relu", padding="same")(
    encoder_input
)
x = layers.Conv2D(64, (3, 3), strides=2, activation="relu", padding="same")(x)
x = layers.Conv2D(128, (3, 3), strides=2, activation="relu", padding="same")(x)
shape_before_flattening = K.int_shape(x)[1:]

x = layers.Flatten()(x)
z_mean = layers.Dense(2, name="z_mean")(x) ❶
z_log_var = layers.Dense(2, name="z_log_var")(x)
z = Sampling()([z_mean, z_log_var]) ❷
```

```
encoder = models.Model(encoder_input, [z_mean, z_log_var, z], name="encoder") ❸
```

❶ Instead of connecting the `Flatten` layer directly to the 2D latent space, we connect it to layers `z_mean` and `z_log_var`.

❷ The `Sampling` layer samples a point `z` in the latent space from the normal distribution defined by the parameters `z_mean` and `z_log_var`.

❸ The Keras `Model` that defines the encoder—a model that takes an input image and outputs `z_mean`, `z_log_var`, and a sampled point `z` from the normal distribution defined by these parameters.

A summary of the encoder is shown in Table 3-4.

Table 3-4. Model summary of the VAE encoder

Layer (type)	Output shape	Param #	Connected to
InputLayer (input)	(None, 32, 32, 1)	0	[]
Conv2D (conv2d_1)	(None, 16, 16, 32)	320	[input]
Conv2D (conv2d_2)	(None, 8, 8, 64)	18,496	[conv2d_1]
Conv2D (conv2d_3)	(None, 4, 4, 128)	73,856	[conv2d_2]
Flatten (flatten)	(None, 2048)	0	[conv2d_3]
Dense (z_mean)	(None, 2)	4,098	[flatten]
Dense (z_log_var)	(None, 2)	4,098	[flatten]
Sampling (z)	(None, 2)	0	[z_mean, z_log_var]

Total params	100,868
Trainable params	100,868
Non-trainable params	0

The only other part of the original autoencoder that we need to change is the loss function.

The Loss Function

Previously, our loss function only consisted of the *reconstruction loss* between images and their attempted copies after being passed through the encoder and decoder. The reconstruction loss also appears in a variational autoencoder, but we now require one extra component: the *Kullback–Leibler (KL) divergence* term.

KL divergence is a way of measuring how much one probability distribution differs from another. In a VAE, we want to measure how much our normal distribution with parameters `z_mean` and `z_log_var` differs from a standard normal distribution. In this special case, it can be shown that the KL divergence has the following closed form:

```
kl_loss = -0.5 * sum(1 + z_log_var - z_mean ^ 2 - exp(z_log_var))
```

or in mathematical notation:

$$D_{KL}[N(\mu, \sigma \parallel N(0,1)] = -\frac{1}{2}\sum\left(1 + log\left(\sigma^2\right) - \mu^2 - \sigma^2\right)$$

The sum is taken over all the dimensions in the latent space. `kl_loss` is minimized to 0 when `z_mean = 0` and `z_log_var = 0` for all dimensions. As these two terms start to differ from 0, `kl_loss` increases.

In summary, the KL divergence term penalizes the network for encoding observations to `z_mean` and `z_log_var` variables that differ significantly from the parameters of a standard normal distribution, namely `z_mean = 0` and `z_log_var = 0`.

Why does this addition to the loss function help?

Firstly, we now have a well-defined distribution that we can use for choosing points in the latent space—the standard normal distribution. Secondly, since this term tries to force all encoded distributions toward the standard normal distribution, there is less chance that large gaps will form between point clusters. Instead, the encoder will try to use the space around the origin symmetrically and efficiently.

In the original VAE paper, the loss function for a VAE was simply the addition of the reconstruction loss and the KL divergence loss term. A variant on this (the β-VAE) includes a factor that weights the KL divergence to ensure that it is well balanced with the reconstruction loss. If we weight the reconstruction loss too heavily, the KL loss will not have the desired regulatory effect and we will see the same problems that we experienced with the plain autoencoder. If the KL divergence term is weighted too heavily, the KL divergence loss will dominate and the reconstructed images will be poor. This weighting term is one of the parameters to tune when you're training your VAE.

Training the Variational Autoencoder

Example 3-13 shows how we build the overall VAE model as a subclass of the abstract Keras `Model` class. This allows us to include the calculation of the KL divergence term of the loss function in a custom `train_step` method.

Example 3-13. Training the VAE

```
class VAE(models.Model):
    def __init__(self, encoder, decoder, **kwargs):
        super(VAE, self).__init__(**kwargs)
        self.encoder = encoder
        self.decoder = decoder
        self.total_loss_tracker = metrics.Mean(name="total_loss")
        self.reconstruction_loss_tracker = metrics.Mean(
            name="reconstruction_loss"
        )
        self.kl_loss_tracker = metrics.Mean(name="kl_loss")

    @property
    def metrics(self):
        return [
            self.total_loss_tracker,
            self.reconstruction_loss_tracker,
            self.kl_loss_tracker,
        ]

    def call(self, inputs): ❶
        z_mean, z_log_var, z = encoder(inputs)
        reconstruction = decoder(z)
        return z_mean, z_log_var, reconstruction

    def train_step(self, data): ❷
        with tf.GradientTape() as tape:
            z_mean, z_log_var, reconstruction = self(data)
            reconstruction_loss = tf.reduce_mean(
                500
                * losses.binary_crossentropy(
                    data, reconstruction, axis=(1, 2, 3)
                )
            ) ❸
            kl_loss = tf.reduce_mean(
                tf.reduce_sum(
                    -0.5
                    * (1 + z_log_var - tf.square(z_mean) - tf.exp(z_log_var)),
                    axis = 1,
                )
            )
            total_loss = reconstruction_loss + kl_loss ❹

        grads = tape.gradient(total_loss, self.trainable_weights)
```

```
        self.optimizer.apply_gradients(zip(grads, self.trainable_weights))

        self.total_loss_tracker.update_state(total_loss)
        self.reconstruction_loss_tracker.update_state(reconstruction_loss)
        self.kl_loss_tracker.update_state(kl_loss)

        return {m.name: m.result() for m in self.metrics}

vae = VAE(encoder, decoder)
vae.compile(optimizer="adam")
vae.fit(
    train,
    epochs=5,
    batch_size=100
)
```

❶ This function describes what we would like returned what we call the VAE on a particular input image.

❷ This function describes one training step of the VAE, including the calculation of the loss function.

❸ A beta value of 500 is used in the reconstruction loss.

❹ The total loss is the sum of the reconstruction loss and the KL divergence loss.

Gradient Tape

TensorFlow's *Gradient Tape* is a mechanism that allows the computation of gradients of operations executed during a forward pass of a model. To use it, you need to wrap the code that performs the operations you want to differentiate in a `tf.GradientTape()` context. Once you have recorded the operations, you can compute the gradient of the loss function with respect to some variables by calling `tape.gradient()`. The gradients can then be used to update the variables with the optimizer.

This mechanism is useful for calculating the gradient of custom loss functions (as we have done here) and also for creating custom training loops, as we shall see in Chapter 4.

Analysis of the Variational Autoencoder

Now that we have trained our VAE, we can use the encoder to encode the images in the test set and plot the `z_mean` values in the latent space. We can also sample from a standard normal distribution to generate points in the latent space and use the decoder to decode these points back into pixel space to see how the VAE performs.

Figure 3-13 shows the structure of the new latent space, alongside some sampled points and their decoded images. We can immediately see several changes in how the latent space is organized.

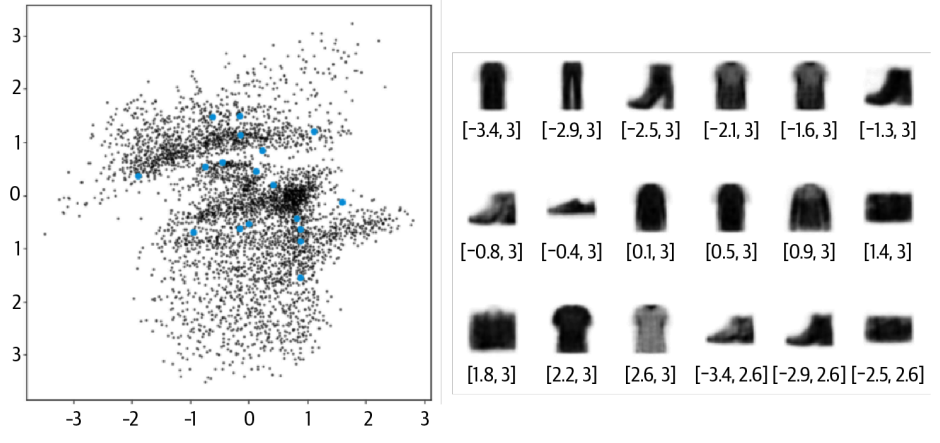

Figure 3-13. The new latent space: the black dots show the `z_mean` value of each encoded image, while blue dots show some sampled points in the latent space (with their decoded images on the right)

Firstly, the KL divergence loss term ensures that the `z_mean` and `z_log_var` values of the encoded images never stray too far from a standard normal distribution. Secondly, there are not so many poorly formed images as the latent space is now much more continuous, due to fact that the encoder is now stochastic, rather than deterministic.

Finally, by coloring points in the latent space by clothing type (Figure 3-14), we can see that there is no preferential treatment of any one type. The righthand plot shows the space transformed into *p*-values—we can see that each color is approximately equally represented. Again, it's important to remember that the labels were not used at all during training; the VAE has learned the various forms of clothing by itself in order to help minimize reconstruction loss.

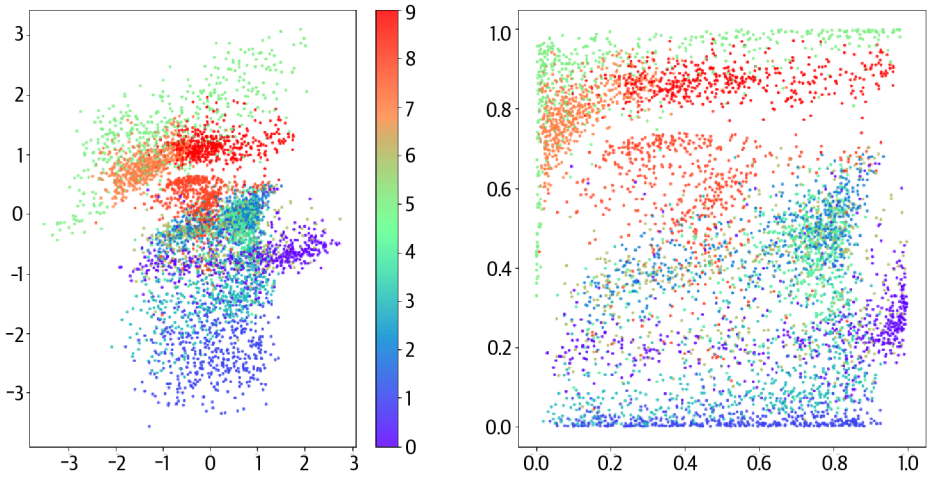

Figure 3-14. The latent space of the VAE colored by clothing type

Exploring the Latent Space

So far, all of our work on autoencoders and variational autoencoders has been limited to a latent space with two dimensions. This has helped us to visualize the inner workings of a VAE on the page and understand why the small tweaks that we made to the architecture of the autoencoder helped transform it into a more powerful class of network that can be used for generative modeling.

Let's now turn our attention to a more complex dataset and see the amazing things that variational autoencoders can achieve when we increase the dimensionality of the latent space.

Running the Code for This Example

The code for this example can be found in the Jupyter notebook located at *notebooks/03_vae/03_faces/vae_faces.ipynb* in the book repository.

The CelebA Dataset

We shall be using the CelebFaces Attributes (CelebA) dataset to train our next variational autoencoder. This is a collection of over 200,000 color images of celebrity faces, each annotated with various labels (e.g., *wearing hat*, *smiling*, etc.). A few examples are shown in Figure 3-15.

Figure 3-15. Some examples from the CelebA dataset (source: Liu et al., 2015)[3]

Of course, we don't need the labels to train the VAE, but these will be useful later when we start exploring how these features are captured in the multidimensional latent space. Once our VAE is trained, we can sample from the latent space to generate new examples of celebrity faces.

The CelebA dataset is also available through Kaggle, so you can download the dataset by running the Kaggle dataset downloader script in the book repository, as shown in Example 3-14. This will save the images and accompanying metadata locally to the */data* folder.

Example 3-14. Downloading the CelebA dataset

```
bash scripts/download_kaggle_data.sh jessicali9530 celeba-dataset
```

We use the Keras function `image_dataset_from_directory` to create a TensorFlow Dataset pointed at the directory where the images are stored, as shown in Example 3-15. This allows us to read batches of images into memory only when required (e.g., during training), so that we can work with large datasets and not worry about having to fit the entire dataset into memory. It also resizes the images to 64 × 64, interpolating between pixel values.

Example 3-15. Preprocessing the CelebA dataset

```
train_data = utils.image_dataset_from_directory(
    "/app/data/celeba-dataset/img_align_celeba/img_align_celeba",
    labels=None,
    color_mode="rgb",
    image_size=(64, 64),
    batch_size=128,
    shuffle=True,
    seed=42,
    interpolation="bilinear",
)
```

The original data is scaled in the range [0, 255] to denote the pixel intensity, which we rescale to the range [0, 1] as shown in Example 3-16.

Example 3-16. Preprocessing the CelebA dataset

```
def preprocess(img):
    img = tf.cast(img, "float32") / 255.0
    return img

train = train_data.map(lambda x: preprocess(x))
```

Training the Variational Autoencoder

The network architecture for the faces model is similar to the Fashion-MNIST example, with a few slight differences:

- Our data now has three input channels (RGB) instead of one (grayscale). This means we need to change the number of channels in the final convolutional transpose layer of the decoder to 3.
- We shall be using a latent space with 200 dimensions instead of 2. Since faces are much more complex than the Fashion-MNIST images, we increase the dimensionality of the latent space so that the network can encode a satisfactory amount of detail from the images.
- There are batch normalization layers after each convolutional layer to stabilize training. Even though each batch takes a longer time to run, the number of batches required to reach the same loss is greatly reduced.
- We increase the β factor for the KL divergence to 2,000. This is a parameter that requires tuning; for this dataset and architecture this value was found to generate good results.

The full architectures of the encoder and decoder are shown in Tables 3-5 and 3-6, respectively.

Table 3-5. Model summary of the VAE faces encoder

Layer (type)	Output shape	Param #	Connected to
InputLayer (input)	(None, 32, 32, 3)	0	[]
Conv2D (conv2d_1)	(None, 16, 16, 128)	3,584	[input]
BatchNormalization (bn_1)	(None, 16, 16, 128)	512	[conv2d_1]
LeakyReLU (lr_1)	(None, 16, 16, 128)	0	[bn_1]
Conv2D (conv2d_2)	(None, 8, 8, 128)	147,584	[lr_1]
BatchNormalization (bn_2)	(None, 8, 8, 128)	512	[conv2d_2]
LeakyReLU (lr_2)	(None, 8, 8, 128)	0	[bn_2]
Conv2D (conv2d_3)	(None, 4, 4, 128)	147,584	[lr_2]
BatchNormalization (bn_3)	(None, 4, 4, 128)	512	[conv2d_3]
LeakyReLU (lr_3)	(None, 4, 4, 128)	0	[bn_3]
Conv2D (conv2d_4)	(None, 2, 2, 128)	147,584	[lr_3]
BatchNormalization (bn_4)	(None, 2, 2, 128)	512	[conv2d_4]
LeakyReLU (lr_4)	(None, 2, 2, 128)	0	[bn_4]
Flatten (flatten)	(None, 512)	0	[lr_4]
Dense (z_mean)	(None, 200)	102,600	[flatten]
Dense (z_log_var)	(None, 200)	102,600	[flatten]
Sampling (z)	(None, 200)	0	[z_mean, z_log_var]

Total params	653,584
Trainable params	652,560
Non-trainable params	1,024

Table 3-6. Model summary of the VAE faces decoder

Layer (type)	Output shape	Param #
InputLayer	(None, 200)	0
Dense	(None, 512)	102,912
BatchNormalization	(None, 512)	2,048
LeakyReLU	(None, 512)	0
Reshape	(None, 2, 2, 128)	0
Conv2DTranspose	(None, 4, 4, 128)	147,584
BatchNormalization	(None, 4, 4, 128)	512
LeakyReLU	(None, 4, 4, 128)	0
Conv2DTranspose	(None, 8, 8, 128)	147,584

Layer (type)	Output shape	Param #
BatchNormalization	(None, 8, 8, 128)	512
LeakyReLU	(None, 8, 8, 128)	0
Conv2DTranspose	(None, 16, 16, 128)	147,584
BatchNormalization	(None, 16, 16, 128)	512
LeakyReLU	(None, 16, 16, 128)	0
Conv2DTranspose	(None, 32, 32, 128)	147,584
BatchNormalization	(None, 32, 32, 128)	512
LeakyReLU	(None, 32, 32, 128)	0
Conv2DTranspose	(None, 32, 32, 3)	3,459

Total params	700,803
Trainable params	698,755
Non-trainable params	2,048

After around five epochs of training, our VAE should be able to produce novel images of celebrity faces!

Analysis of the Variational Autoencoder

First, let's take a look at a sample of reconstructed faces. The top row in Figure 3-16 shows the original images and the bottom row shows the reconstructions once they have passed through the encoder and decoder.

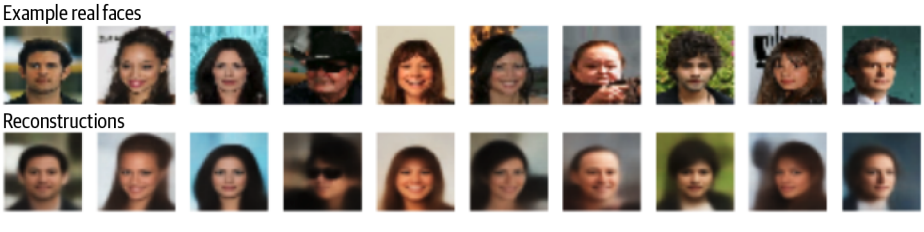

Figure 3-16. Reconstructed faces, after passing through the encoder and decoder

We can see that the VAE has successfully captured the key features of each face—the angle of the head, the hairstyle, the expression, etc. Some of the fine detail is missing, but it is important to remember that the aim of building variational autoencoders isn't to achieve perfect reconstruction loss. Our end goal is to sample from the latent space in order to generate new faces.

For this to be possible we must check that the distribution of points in the latent space approximately resembles a multivariate standard normal distribution. If we see any dimensions that are significantly different from a standard normal distribution,

we should probably reduce the reconstruction loss factor, since the KL divergence term isn't having enough effect.

The first 50 dimensions in our latent space are shown in Figure 3-17. There aren't any distributions that stand out as being significantly different from the standard normal, so we can move on to generating some faces!

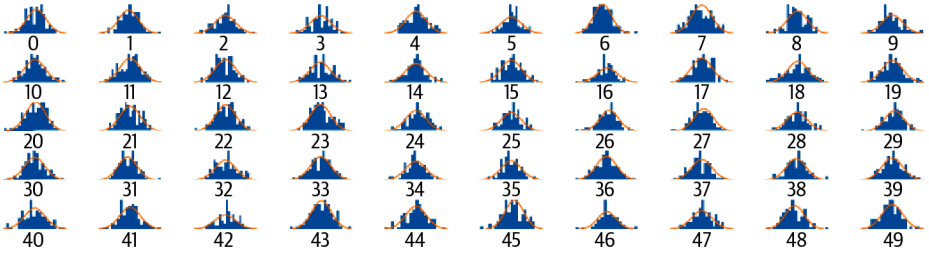

Figure 3-17. Distributions of points for the first 50 dimensions in the latent space

Generating New Faces

To generate new faces, we can use the code in Example 3-17.

Example 3-17. Generating new faces from the latent space

```
grid_width, grid_height = (10,3)
z_sample = np.random.normal(size=(grid_width * grid_height, 200)) ❶

reconstructions = decoder.predict(z_sample) ❷

fig = plt.figure(figsize=(18, 5))
fig.subplots_adjust(hspace=0.4, wspace=0.4)
for i in range(grid_width * grid_height):
    ax = fig.add_subplot(grid_height, grid_width, i + 1)
    ax.axis("off")
    ax.imshow(reconstructions[i, :, :]) ❸
```

❶ Sample 30 points from a standard multivariate normal distribution with 200 dimensions.

❷ Decode the sampled points.

❸ Plot the images!

The output is shown in Figure 3-18.

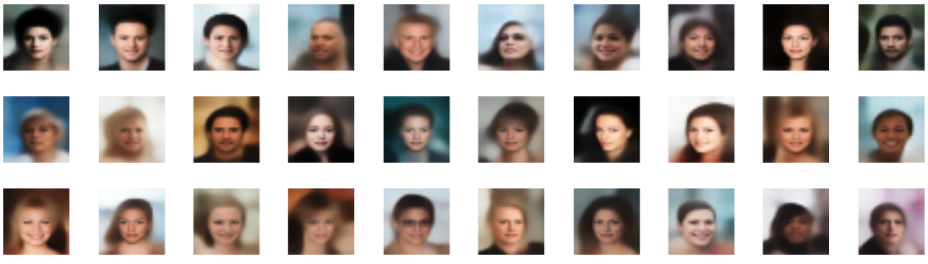

Figure 3-18. New generated faces

Amazingly, the VAE is able to take the set of points that we sampled from a standard normal distribution and convert each into a convincing image of a person's face. This is our first glimpse of the true power of generative models!

Next, let's see if we can start to use the latent space to perform some interesting operations on generated images.

Latent Space Arithmetic

One benefit of mapping images into a lower-dimensional latent space is that we can perform arithmetic on vectors in this latent space that has a visual analogue when decoded back into the original image domain.

For example, suppose we want to take an image of somebody who looks sad and give them a smile. To do this we first need to find a vector in the latent space that points in the direction of increasing smile. Adding this vector to the encoding of the original image in the latent space will give us a new point which, when decoded, should give us a more smiley version of the original image.

So how can we find the *smile* vector? Each image in the CelebA dataset is labeled with attributes, one of which is `Smiling`. If we take the average position of encoded images in the latent space with the attribute `Smiling` and subtract the average position of encoded images that do not have the attribute `Smiling`, we will obtain the vector that points in the direction of `Smiling`, which is exactly what we need.

Conceptually, we are performing the following vector arithmetic in the latent space, where `alpha` is a factor that determines how much of the feature vector is added or subtracted:

```
z_new = z + alpha * feature_vector
```

Let's see this in action. Figure 3-19 shows several images that have been encoded into the latent space. We then add or subtract multiples of a certain vector (e.g., `Smiling`, `Black_Hair`, `Eyeglasses`, `Young`, `Male`, `Blond_Hair`) to obtain different versions of the image, with only the relevant feature changed.

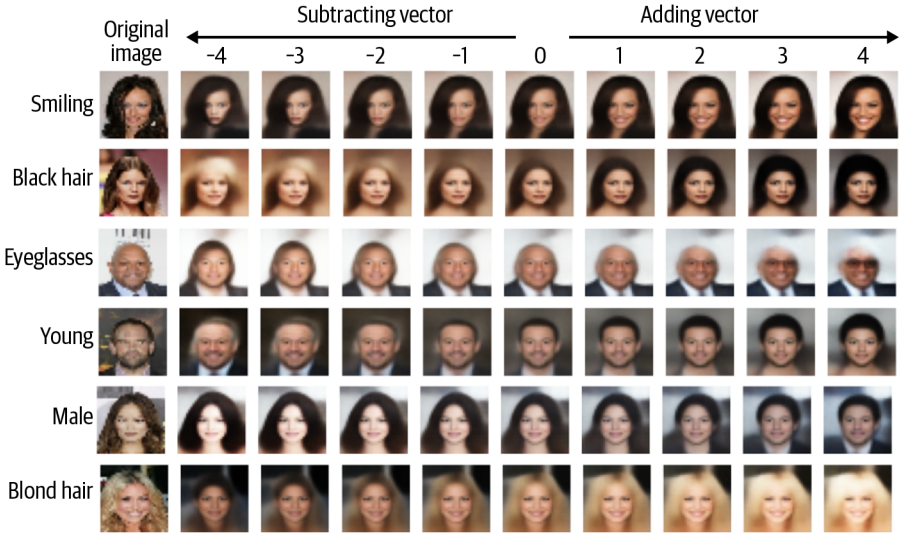

Figure 3-19. Adding and subtracting features to and from faces

It is remarkable that even though we are moving the point a significantly large distance in the latent space, the core image remains approximately the same, except for the one feature that we want to manipulate. This demonstrates the power of variational autoencoders for capturing and adjusting high-level features in images.

Morphing Between Faces

We can use a similar idea to morph between two faces. Imagine two points in the latent space, A and B, that represent two images. If you started at point A and walked toward point B in a straight line, decoding each point on the line as you went, you would see a gradual transition from the starting face to the end face.

Mathematically, we are traversing a straight line, which can be described by the following equation:

```
z_new = z_A * (1- alpha) + z_B * alpha
```

Here, `alpha` is a number between 0 and 1 that determines how far along the line we are, away from point A.

Figure 3-20 shows this process in action. We take two images, encode them into the latent space, and then decode points along the straight line between them at regular intervals.

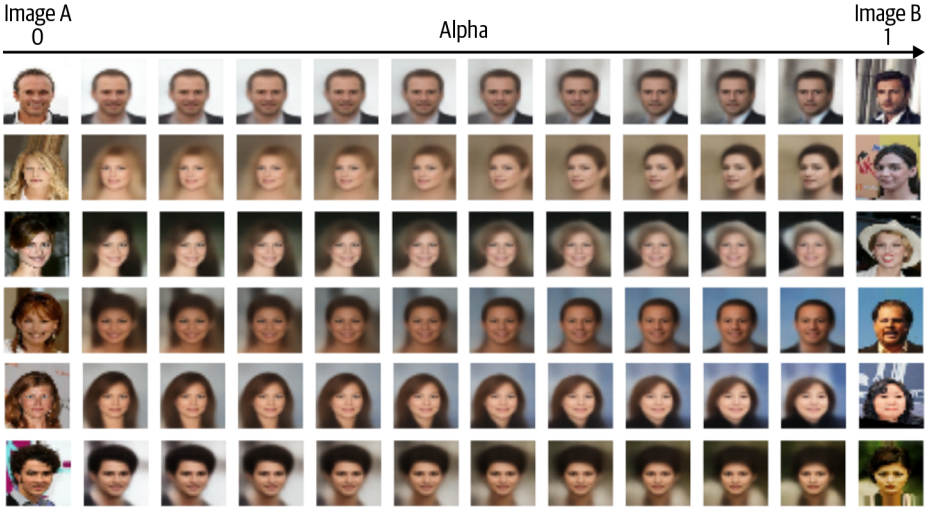

Figure 3-20. Morphing between two faces

It is worth noting the smoothness of the transition—even where there are multiple features to change simultaneously (e.g., removal of glasses, hair color, gender), the VAE manages to achieve this fluidly, showing that the latent space of the VAE is truly a continuous space that can be traversed and explored to generate a multitude of different human faces.

Summary

In this chapter we have seen how variational autoencoders are a powerful tool in the generative modeling toolbox. We started by exploring how plain autoencoders can be used to map high-dimensional images into a low-dimensional latent space, so that high-level features can be extracted from the individually uninformative pixels. However, we quickly found that there were some drawbacks to using plain autoencoders as a generative model—sampling from the learned latent space was problematic, for example.

Variational autoencoders solve these problems by introducing randomness into the model and constraining how points in the latent space are distributed. We saw that with a few minor adjustments, we can transform our autoencoder into a variational autoencoder, thus giving it the power to be a true generative model.