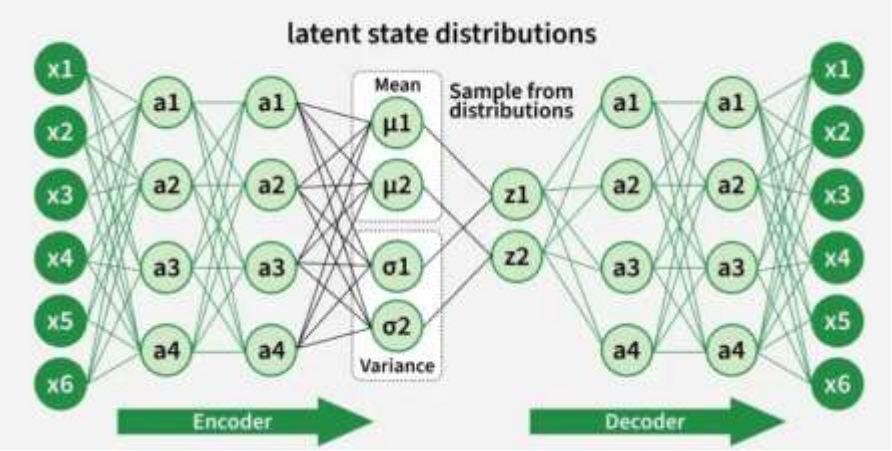
Deep learning for image generation

In this section we'll review some high-level concepts pertaining to image generation, alongside implementation details relative to two of the main techniques in this domain: *variational autoencoders* (VAEs) and *diffusion models*.

A **Variational Autoencoder (VAE)** is a generative model that learns to map input images into a continuous "latent space" and then reconstructs them. Unlike a standard autoencoder, a VAE ensures the latent space follows a specific distribution (usually Gaussian), allowing us to generate entirely new images by sampling random points from that space.

Core Architecture of VAE consists of three main parts:

- **Encoder:** Compresses the image into two vectors: a **mean (μ)** and a **log-variance (log σ^2)**.
- **Reparameterization Trick:** Samples a latent vector **z** from the distribution defined by the encoder. This step is critical because it allows the model to remain differentiable for training.
- **Decoder:** Takes the sampled vector **z** and maps it back into an image.



Sampling from latent spaces of images

The key idea of image generation is to develop a low-dimensional *latent space* of representations (which, like everything else in deep learning, is a vector space) where any point can be mapped to a "valid" image: an image that looks like the real thing. The module capable of realizing this mapping, taking as input a latent point and outputting an image (a grid of pixels), is usually called a *generator*, or sometimes a *decoder*. Once such a latent space has been learned, we can sample points from it, and, by mapping them back to image space, generate images that have never been seen before (see figure 1.1) — the in-betweens of the training images.

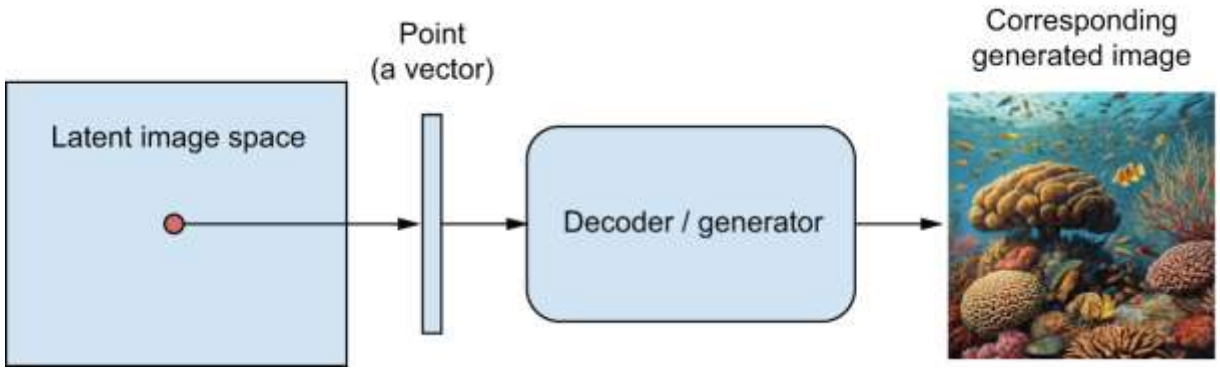


Figure 1.1: Using a latent vector space to sample new images

Further, *text-conditioning* makes it possible to map a space of prompts in natural language to the latent space (see figure 1.2), making it possible to do *language-guided image generation* — generating pictures that correspond to a text description. This category of models is called *text-to-image* models.

Interpolating between many training images in the latent space enables such models to generate infinite combinations of visual concepts, including many that no one had explicitly come up with before. A horse riding a bike on the moon? You got it. This makes image generation a powerful brush for creative-minded people to play with.

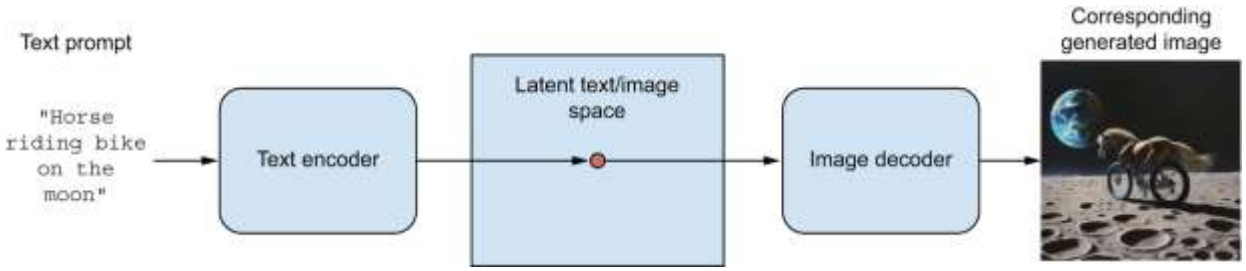


Figure 1.2: Language-guided image generation

There's a range of different strategies for learning such latent spaces of image representations, each with its own characteristics. The most common types of image generation models are

- Diffusion models
- Variational autoencoders (VAEs)
- Generative adversarial networks (GANs)

While previous editions of this book covered GANs, they have gradually fallen out of fashion in recent years and have been all but replaced by diffusion models. In this edition, we'll cover both VAEs and diffusion models. In the models we'll build ourselves, we'll focus on unconditioned image generation — sampling images from a latent space without text conditioning.

Variational autoencoders

VAEs, simultaneously discovered by Kingma and Welling in December 2013[1] and Rezende, Mohamed, and Wierstra in January 2014,[2] are a kind of generative model that's especially appropriate for the task of image editing via concept vectors. They're a kind of *autoencoder* — a type of network that aims to encode an input to a low dimensional latent space and then decode it back — that mixes ideas from deep learning with Bayesian inference.

A classical image autoencoder takes an image, maps it to a latent vector space via an encoder module, and then decodes it back to an output with the same dimensions as the original image, via a decoder module (see figure 1.3). It's then trained by using as target data the *same images* as the input images, meaning the autoencoder learns to reconstruct the original inputs. By imposing various constraints on the code (the output of the encoder), you can get the autoencoder to learn more or less interesting latent representations of the data. Most commonly, you'll constrain the code to be low-dimensional and sparse (mostly zeros), in which case the encoder acts as a way to compress the input data into fewer bits of information.

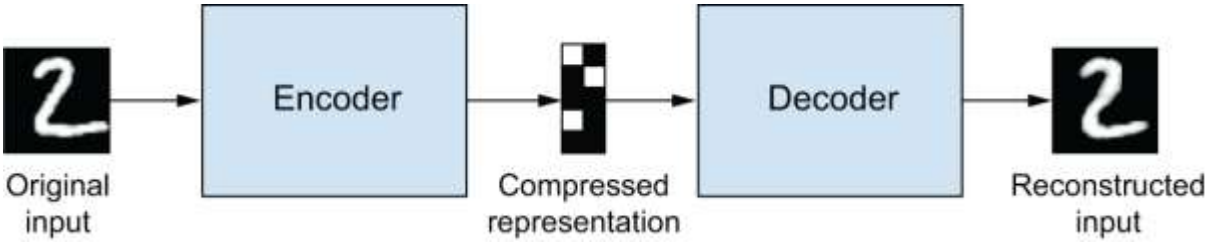


Figure 1.3: An autoencoder: mapping an input x to a compressed representation and then decoding it back as x'

In practice, such classical autoencoders don't lead to particularly useful or nicely structured latent spaces. They are not much good at compression either. For these reasons, they have largely fallen out of fashion. VAEs, however, augment autoencoders with a little bit of statistical magic that forces them to learn continuous, highly structured latent spaces. They have turned out to be a powerful tool for image generation.

A VAE, instead of compressing its input image into a fixed code in the latent space, turns the image into the parameters of a statistical distribution: a mean and a variance. Essentially, this means we're assuming the input image has been generated by a statistical process, and that the randomness of this process should be taken into account during encoding and decoding. The VAE then uses the mean and variance parameters to randomly sample one element of the distribution, and decodes that element back to the original input (see figure 1.4). The stochasticity of this process improves robustness and forces the latent space to encode meaningful representations everywhere: every point sampled in the latent space is decoded to a valid output.

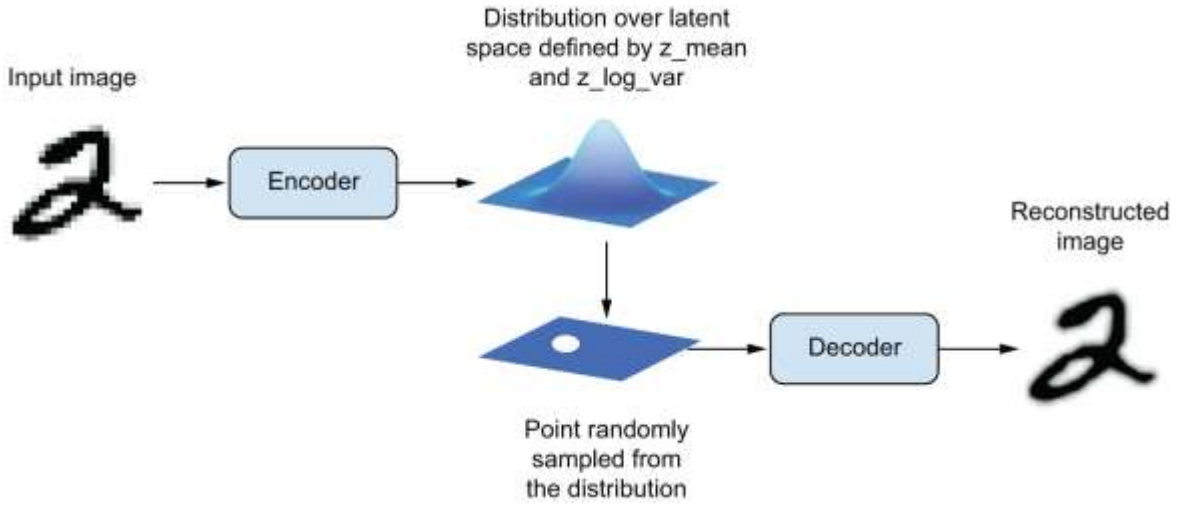


Figure 1.4: A VAE maps an image to two vectors, `z_mean` and `z_log_sigma`, which define a probability distribution over the latent space, used to sample a latent point to decode.

In technical terms, here's how a VAE works:

1. An **encoder** module turns the input sample `input_img` into two parameters in a latent space of representations, `z_mean` and `z_log_variance`.
2. A Randomly sampled point `z` from the latent normal distribution that's assumed to generate the input image, via `z = z_mean + exp(z_log_variance) * epsilon`, where `epsilon` is a random tensor of small values.
3. A **decoder** module maps this point in the latent space back to the original input image.

Because `epsilon` is random, the process ensures that every point that's close to the latent location where you encoded `input_img` (`z-mean`) can be decoded to something similar to `input_img`, thus forcing the latent space to be continuously meaningful. Any two close points in the latent space will decode to highly similar images. Continuity, combined with the low dimensionality of the latent space, forces every direction in the latent space to encode a meaningful axis of variation of the data, making the latent space very structured and thus highly suitable to manipulation via concept vectors.

The parameters of a VAE are trained via two loss functions: a ***reconstruction loss*** that forces the decoded samples to match the initial inputs, and a ***regularization loss*** that helps learn well-rounded latent distributions and reduces overfitting to the training data. Schematically, the process looks like this:

```
# Encodes the input into a mean and variance parameter
z_mean, z_log_variance = encoder(input_img)
# Draws a latent point using a small random epsilon
z = z_mean + exp(z_log_variance) * epsilon
# Decodes z back to an image
reconstructed_img = decoder(z)
# Instantiates the autoencoder model, which maps an input image to its
# reconstruction
model = Model(input_img, reconstructed_img)
```

we can then train the model using the reconstruction loss and the regularization loss. For the regularization loss, we typically use an expression (the Kullback–Leibler divergence) meant to nudge the distribution of the encoder output toward a well-rounded normal distribution centered around 0. This provides the encoder with a sensible assumption about the structure of the latent space it's modeling.

Implementing a VAE with Keras

We're going to be implementing a VAE that can generate MNIST digits. It's going to have three parts:

- An encoder network that turns a real image into a mean and a variance in the latent space
- A sampling layer that takes such a mean and variance and uses them to sample a random point from the latent space
- A decoder network that turns points from the latent space back into images

The following listing shows the encoder network you'll use, mapping images to the parameters of a probability distribution over the latent space. It's a simple ConvNet that maps the input image `x` to two vectors, `z_mean` and `z_log_var`. One important detail is that we use strides for downsampling feature maps, instead of max pooling. The last time we did this was in the image segmentation example of chapter 11. Recall that, in general, strides are preferable to max pooling for any model that cares about *information location* — that is, *where* stuff is in the image — and this one does, since it will have to produce an image encoding that can be used to reconstruct a valid image.

```
import keras
from keras import layers

# Dimensionality of the latent space: a 2D plane
latent_dim = 2

image_inputs = keras.Input(shape=(28, 28, 1))
x = layers.Conv2D(32, 3, activation="relu", strides=2, padding="same")(
    image_inputs
)
x = layers.Conv2D(64, 3, activation="relu", strides=2, padding="same")(x)
x = layers.Flatten()(x)
x = layers.Dense(16, activation="relu")(x)
# The input image ends up being encoded into these two parameters.
z_mean = layers.Dense(latent_dim, name="z_mean")(x)
z_log_var = layers.Dense(latent_dim, name="z_log_var")(x)
encoder = keras.Model(image_inputs, [z_mean, z_log_var], name="encoder")
```

Its summary looks like this:

```
>>> encoder.summary()
Model: "encoder"
```

Layer (type)	Output Shape	Param #	Connected to
input_layer (InputLayer)	(None, 28, 28, 1)	0	-
conv2d (Conv2D)	(None, 14, 14, 32)	320	input_layer[0][0]
conv2d_1 (Conv2D)	(None, 7, 7, 64)	18,496	conv2d[0][0]
flatten (Flatten)	(None, 3136)	0	conv2d_1[0][0]
dense (Dense)	(None, 16)	50,192	flatten[0][0]
z_mean (Dense)	(None, 2)	34	dense[0][0]
z_log_var (Dense)	(None, 2)	34	dense[0][0]

```
Total params: 69,076 (269.83 KB)
Trainable params: 69,076 (269.83 KB)
Non-trainable params: 0 (0.00 B)
```

Next is the code for using `z_mean` and `z_log_var`, the parameters of the statistical distribution assumed to have produced `input_img`, to generate a latent space point `z`.

```
from keras import ops

class Sampler(keras.Layer):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        # We need a seed generator to use functions from keras.random
        # in call().
        self.seed_generator = keras.random.SeedGenerator()
        self.built = True

    def call(self, z_mean, z_log_var):
        batch_size = ops.shape(z_mean)[0]
        z_size = ops.shape(z_mean)[1]
        epsilon = keras.random.normal(
            # Draws a batch of random normal vectors
            (batch_size, z_size), seed=self.seed_generator
        )
        # Applies the VAE sampling formula
        return z_mean + ops.exp(0.5 * z_log_var) * epsilon
```

Listing 1.2: Latent space sampling layer

The following listing shows the decoder implementation. We reshape the vector `z` to the dimensions of an image and then use a few convolution layers to obtain a final image output that has the same dimensions as the original `input_img`.

```
# Input where we'll feed z
latent_inputs = keras.Input(shape=(latent_dim,))
# Produces the same number of coefficients we had at the level of the
# Flatten layer in the encoder
x = layers.Dense(7 * 7 * 64, activation="relu")(latent_inputs)
# Reverts the Flatten layer of the encoder
x = layers.Reshape((7, 7, 64))(x)
# Reverts the Conv2D layers of the encoder
x = layers.Conv2DTranspose(64, 3, activation="relu", strides=2,
padding="same")(x)
x = layers.Conv2DTranspose(32, 3, activation="relu", strides=2,
padding="same")(x)
# The output ends up with shape (28, 28, 1).
decoder_outputs = layers.Conv2D(1, 3, activation="sigmoid",
padding="same")(x)
decoder = keras.Model(latent_inputs, decoder_outputs, name="decoder")
```

Listing 1.3: VAE decoder network, mapping latent space points to images

```
>>> decoder.summary()
Model: "decoder"
```

Layer (type)	Output Shape	Param #
input_layer_1 (InputLayer)	(None, 2)	0
dense_1 (Dense)	(None, 3136)	9,408
reshape (Reshape)	(None, 7, 7, 64)	0
conv2d_transpose (Conv2DTranspose)	(None, 14, 14, 64)	36,928
conv2d_transpose_1 (Conv2DTranspose)	(None, 28, 28, 32)	18,464
conv2d_2 (Conv2D)	(None, 28, 28, 1)	289

```
Total params: 65,089 (254.25 KB)
Trainable params: 65,089 (254.25 KB)
Non-trainable params: 0 (0.00 B)
```

Now, let's create the VAE model itself. This is your first example of a model that isn't doing supervised learning (an autencoder is an example of *self-supervised* learning because it uses its inputs as targets). Whenever you depart from classic supervised learning, it's common to subclass the `Model` class and implement a custom `train_step()` to specify the new training logic, a familiar workflow. We could easily do that here, but a downside of this technique is that the `train_step()` contents must be backend specific — you'd use `GradientTape` with TensorFlow, you'd use `loss.backward()` with PyTorch, and so on. A simpler way to customize your training logic is to just implement the `compute_loss()` method instead and keep the default `train_step()`. `compute_loss()` is the key bit of differentiable logic called by the built-in

`train_step()`. Since it doesn't involve direct manipulation of gradients, it's easy to keep it backend agnostic.

Its signature is as follows:

```
compute_loss(x, y, y_pred, sample_weight=None, training=True)
```

where `x` is the model's input; `y` is the model's target (in our case, it is `None` since the dataset we use only has inputs, no targets); and `y_pred` is the output of `call()` — the model's predictions. In any supervised training workflow, you'd compute the loss based on `y` and `y_pred`. In our case, since `y` is `None` and `y_pred` contains the latent parameters, we'll compute the loss using `x` (the original input) and the `reconstruction` derived from `y_pred`.

The method must return a scalar, the loss value to be minimized. You can also use `compute_loss()` to update the state of your metrics, which is something we'll want to do in our case.

Now, let's write our VAE with a custom `compute_loss()` method. It works with all backends with no code changes!

```
class VAE(keras.Model):
    def __init__(self, encoder, decoder, **kwargs):
        super().__init__(**kwargs)
        self.encoder = encoder
        self.decoder = decoder
        self.sampler = Sampler()
        # We'll use these metrics to keep track of the loss averages
        # over each epoch.
        self.reconstruction_loss_tracker = keras.metrics.Mean(
            name="reconstruction_loss"
        )
        self.kl_loss_tracker = keras.metrics.Mean(name="kl_loss")

    def call(self, inputs):
        return self.encoder(inputs)

    def compute_loss(self, x, y, y_pred, sample_weight=None, training=True):
        # Argument x is the model's input.
        original = x
        # Argument y_pred is the output of call().
        z_mean, z_log_var = y_pred
        # This is our reconstructed image.
        reconstruction = self.decoder(self.sampler(z_mean, z_log_var))

        # We sum the reconstruction loss over the spatial dimensions
        # (axes 1 and 2) and take its mean over the batch dimension.
        reconstruction_loss = ops.mean(
            ops.sum(
                keras.losses.binary_crossentropy(x, reconstruction), axis=(1,
2)
            )
        )
```

```
        # Adds the regularization term (Kullback-Leibler divergence)
        kl_loss = -0.5 * (
            1 + z_log_var - ops.square(z_mean) - ops.exp(z_log_var)
        )
        total_loss = reconstruction_loss + ops.mean(kl_loss)

        # Updates the state of our loss-tracking metrics
        self.reconstruction_loss_tracker.update_state(reconstruction_loss)
        self.kl_loss_tracker.update_state(kl_loss)
        return total_loss
```

Listing 1.4: VAE model with custom `compute_loss()` method

Finally, you're ready to instantiate and train the model on MNIST digits. Because `compute_loss()` already takes care of the loss, you don't specify an external loss at compile time (`loss=None`), which, in turn, means you won't pass target data during training (as you can see, you only pass `x_train` to the model in `fit`).

```
import numpy as np

(x_train, _), (x_test, _) = keras.datasets.mnist.load_data()
# We train on all MNIST digits, so we concatenate the training and test
# samples.
mnist_digits = np.concatenate([x_train, x_test], axis=0)
mnist_digits = np.expand_dims(mnist_digits, -1).astype("float32") / 255

vae = VAE(encoder, decoder)
# We don't pass a loss argument in compile(), since the loss is already
# part of the train_step().
vae.compile(optimizer=keras.optimizers.Adam())
# We don't pass targets in fit(), since train_step() doesn't expect
# any.
vae.fit(mnist_digits, epochs=30, batch_size=128)
```

Listing 1.5: Training the VAE

Once the model is trained, you can use the `decoder` network to turn arbitrary latent space vectors into images.

```
import matplotlib.pyplot as plt

# We'll display a grid of 30 × 30 digits (900 digits total).
n = 30
digit_size = 28
figure = np.zeros((digit_size * n, digit_size * n))

# Samples points linearly on a 2D grid
grid_x = np.linspace(-1, 1, n)
grid_y = np.linspace(-1, 1, n)[::-1]

# Iterates over grid locations
for i, yi in enumerate(grid_y):
    for j, xi in enumerate(grid_x):
        # For each location, samples a digit and adds it to our figure
        z_sample = np.array([[xi, yi]])
        x_decoded = vae.decoder.predict(z_sample)
```

```
        digit = x_decoded[0].reshape(digit_size, digit_size)
        figure[
            i * digit_size : (i + 1) * digit_size,
            j * digit_size : (j + 1) * digit_size,
        ] = digit

plt.figure(figsize=(15, 15))
start_range = digit_size // 2
end_range = n * digit_size + start_range
pixel_range = np.arange(start_range, end_range, digit_size)
sample_range_x = np.round(grid_x, 1)
sample_range_y = np.round(grid_y, 1)
plt.xticks(pixel_range, sample_range_x)
plt.yticks(pixel_range, sample_range_y)
plt.xlabel("z[0]")
plt.ylabel("z[1]")
plt.axis("off")
plt.imshow(figure, cmap="Greys_r")
```

Listing 1.6: Sampling a grid of points from the 2D latent space and decoding them to images

The grid of sampled digits (see figure 1.5) shows a completely continuous distribution of the different digit classes, with one digit morphing into another as you follow a path through latent space. Specific directions in this space have a meaning: for example, there's a direction for "four-ness," "one-ness," and so on.

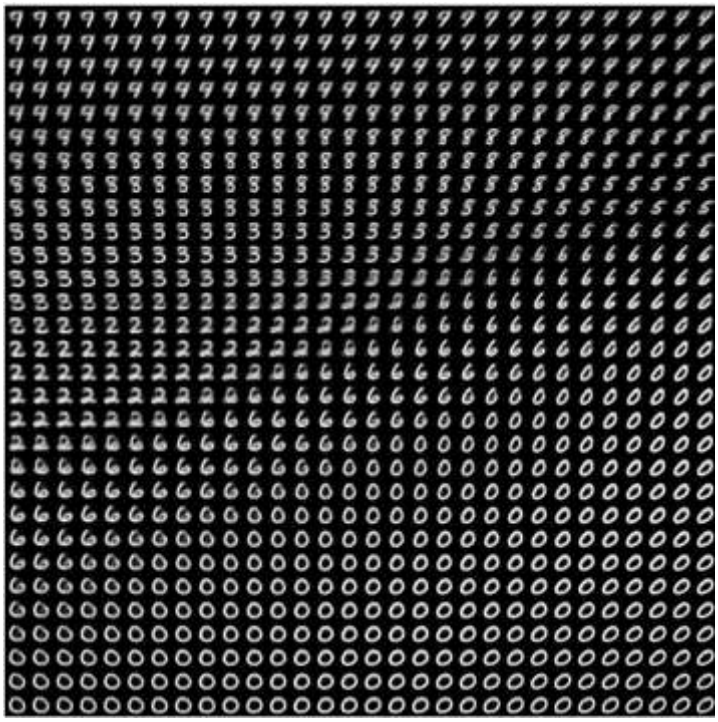


Figure 1.5: Grid of digits decoded from the latent space