CHAPTER 7

Energy-Based Models

Chapter Goals

In this chapter you will:

- Understand how to formulate a deep energy-based model (EBM).
- See how to sample from an EBM using Langevin dynamics.
- Train your own EBM using contrastive divergence.
- Analyze the EBM, including viewing snapshots of the Langevin dynamics sampling process.
- Learn about other types of EBM, such as restricted Boltzmann machines.

Energy-based models are a broad class of generative model that borrow a key idea from modeling physical systems—namely, that the probability of an event can be expressed using a Boltzmann distribution, a specific function that normalizes a real-valued energy function between 0 and 1. This distribution was originally formulated in 1868 by Ludwig Boltzmann, who used it to describe gases in thermal equilibrium.

In this chapter, we will see how we can use this idea to train a generative model that can be used to produce images of handwritten digits. We will explore several new concepts, including contrastive divergence for training the EBM and Langevin dynamics for sampling.

Introduction

We will begin with a short story to illustrate the key concepts behind energy-based models.

The Long-au-Vin Running Club

Diane Mixx was head coach of the long-distance running team in the fictional French town of Long-au-Vin. She was well known for her exceptional abilities as a trainer and had acquired a reputation for being able to turn even the most mediocre of athletes into world-class runners (Figure 7-1).

Figure 7-1. A running coach training some elite athletes (created with Midjourney)

Her methods were based around assessing the energy levels of each athlete. Over years of working with athletes of all abilities, she had developed an incredibly accurate sense of just how much energy a particular athlete had left after a race, just by looking at them. The lower an athlete's energy level, the better—elite athletes always gave everything they had during the race!

To keep her skills sharp, she regularly trained herself by measuring the contrast between her energy sensing abilities on known elite athletes and the best athletes from her club. She ensured that the divergence between her predictions for these two groups was as large as possible, so that people would take her seriously if she said that she had found a true elite athlete within her club.

The real magic was her ability to convert a mediocre runner into a top-class runner. The process was simple—she measured the current energy level of the athlete and worked out the optimal set of adjustments the athlete needed to make to improve their performance next time. Then, after making these adjustments, she measured the athlete's energy level again, looking for it to be slightly lower than before, explaining the improved performance on the track. This process of assessing the optimal adjustments and taking a small step in the right direction would continue until eventually the athlete was indistinguishable from a world-class runner.

After many years Diane retired from coaching and published a book on her methods for generating elite athletes—a system she branded the “Long-au-Vin, Diane Mixx” technique.

The story of Diane Mixx and the Long-au-Vin running club captures the key ideas behind energy-based modeling. Let’s now explore the theory in more detail, before we implement a practical example using Keras.

Energy-Based Models

Energy-based models attempt to model the true data-generating distribution using a *Boltzmann distribution* (Equation 7-1) where $E(x)$ is know as the *energy function* (or *score*) of an observation x.

Equation 7-1. Boltzmann distribution

$$p(\mathbf{x}) = \frac{e^{-E(\mathbf{x})}}{\int_{\hat{\mathbf{x}} \in \mathbf{X}} e^{-E(\hat{\mathbf{x}})}}$$

In practice, this amounts to training a neural network $E(x)$ to output low scores for likely observations (so $p\mathbf{x}$ is close to 1) and high scores for unlikely observations (so $p\mathbf{x}$ is close to 0).

There are two challenges with modeling the data in this way. Firstly, it is not clear how we should use our model for sampling new observations—we can use it to generate a score given an observation, but how do we generate an observation that has a low score (i.e., a plausible observation)?

Secondly, the normalizing denominator of Equation 7-1 contains an integral that is intractable for all but the simplest of problems. If we cannot calculate this integral, then we cannot use maximum likelihood estimation to train the model, as this requires that $p\mathbf{x}$ is a valid probability distribution.

The key idea behind an energy-based model is that we can use approximation techniques to ensure we never need to calculate the intractable denominator. This is in contrast to, say, a normalizing flow, where we go to great lengths to ensure that the transformations that we apply to our standard Gaussian distribution do not change the fact that the output is still a valid probability distribution.

We sidestep the tricky intractable denominator problem by using a technique called contrastive divergence (for training) and a technique called Langevin dynamics (for sampling), following the ideas from Du and Mordatch’s 2019 paper “Implicit

Generation and Modeling with Energy-Based Models."[1] We shall explore these techniques in detail while building our own EBM later in the chapter.

First, let's get set up with a dataset and design a simple neural network that will represent our real-valued energy function $E(x)$.

Running the Code for This Example

The code for this example can be found in the Jupyter notebook located at *notebooks/07_ebm/01_ebm/ebm.ipynb* in the book repository.

The code is adapted from the excellent tutorial on deep energy-based generative models by Phillip Lippe.

The MNIST Dataset

We'll be using the standard MNIST dataset, consisting of grayscale images of handwritten digits. Some example images from the dataset are shown in Figure 7-2.

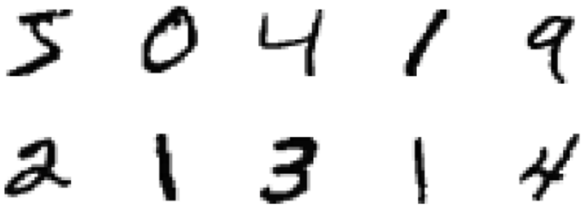
Figure 7-2. Examples of images from the MNIST dataset

The dataset comes prepackaged with TensorFlow, so it can be downloaded as shown in Example 7-1.

Example 7-1. Loading the MNIST dataset

```
from tensorflow.keras import datasets
(x_train, _), (x_test, _) = datasets.mnist.load_data()
```

As usual, we'll scale the pixel values to the range [-1, 1] and add some padding to make the images 32 × 32 pixels in size. We also convert it to a TensorFlow Dataset, as shown in Example 7-2.

Example 7-2. Preprocessing the MNIST dataset

```
def preprocess(imgs):
    imgs = (imgs.astype("float32") - 127.5) / 127.5
    imgs = np.pad(imgs , ((0,0), (2,2), (2,2)), constant_values= -1.0)
    imgs = np.expand_dims(imgs, -1)
    return imgs

x_train = preprocess(x_train)
x_test = preprocess(x_test)
x_train = tf.data.Dataset.from_tensor_slices(x_train).batch(128)
x_test = tf.data.Dataset.from_tensor_slices(x_test).batch(128)
```

Now that we have our dataset, we can build the neural network that will represent our energy function $E(x)$.

The Energy Function

The energy function $E_\theta(x)$ is a neural network with parameters θ that can transform an input image x into a scalar value. Throughout this network, we make use of an activation function called *swish*, as described in the following sidebar.

Swish Activation

Swish is an alternative to ReLU that was introduced by Google in 2017[2] and is defined as follows:

$$\text{swish}(x) = x \cdot \text{sigmoid}(x) = \frac{x}{e^{-x} + 1}$$

Swish is visually similar to ReLU, with the key difference being that it is smooth, which helps to alleviate the vanishing gradient problem. This is particularly important for energy-based models. A plot of the swish function is shown in Figure 7-3.

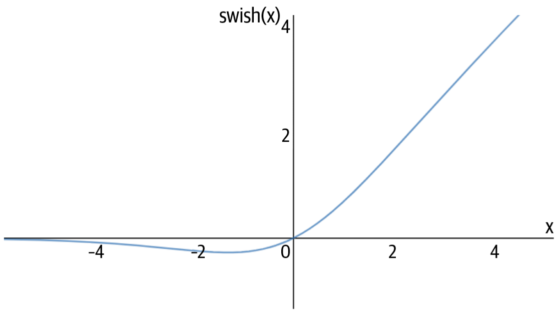

Figure 7-3. The swish activation function

The network is a set of stacked `Conv2D` layers that gradually reduce the size of the image while increasing the number of channels. The final layer is a single fully connected unit with linear activation, so the network can output values in the range ($-\infty$, ∞). The code to build it is given in Example 7-3.

Example 7-3. Building the energy function E(x) neural network

```
ebm_input = layers.Input(shape=(32, 32, 1))
x = layers.Conv2D(
    16, kernel_size=5, strides=2, padding="same", activation = activations.swish
)(ebm_input) ❶
x = layers.Conv2D(
    32, kernel_size=3, strides=2, padding="same", activation = activations.swish
)(x)
x = layers.Conv2D(
    64, kernel_size=3, strides=2, padding="same", activation = activations.swish
)(x)
x = layers.Conv2D(
    64, kernel_size=3, strides=2, padding="same", activation = activations.swish
)(x)
x = layers.Flatten()(x)
x = layers.Dense(64, activation = activations.swish)(x)
ebm_output = layers.Dense(1)(x) ❷
model = models.Model(ebm_input, ebm_output) ❸
```

❶ The energy function is a set of stacked `Conv2D` layers, with swish activation.

❷ The final layer is a single fully connected unit, with a linear activation function.

❸ A Keras `Model` that converts the input image into a scalar energy value.

Sampling Using Langevin Dynamics

The energy function only outputs a score for a given input—how can we use this function to generate new samples that have a low energy score?

We will use a technique called *Langevin dynamics*, which makes use of the fact that we can compute the gradient of the energy function with respect to its input. If we start from a random point in the sample space and take small steps in the opposite direction of the calculated gradient, we will gradually reduce the energy function. If our neural network is trained correctly, then the random noise should transform into an image that resembles an observation from the training set before our eyes!

Stochastic Gradient Langevin Dynamics

Importantly, we must also add a small amount of random noise to the input as we travel across the sample space; otherwise, there is a risk of falling into local minima. The technique is therefore known as stochastic gradient Langevin dynamics.[3]

We can visualize this gradient descent as shown in Figure 7-4, for a two-dimensional space with the energy function value on the third dimension. The path is a noisy descent downhill, following the negative gradient of the energy function $E(x)$ with respect to the input x. In the MNIST image dataset, we have 1,024 pixels so are navigating a 1,024-dimensional space, but the same principles apply!

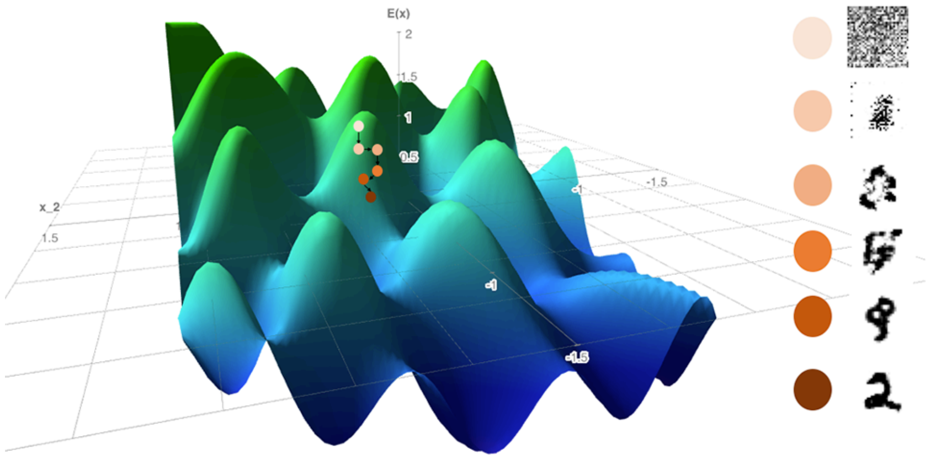

Figure 7-4. Gradient descent using Langevin dynamics

It is worth noting the difference between this kind of gradient descent and the kind of gradient descent we normally use to train a neural network.

When training a neural network, we calculate the gradient of the *loss function* with respect to the *parameters* of the network (i.e., the weights) using backpropagation. Then we update the parameters a small amount in the direction of the negative gradient, so that over many iterations, we gradually minimize the loss.

With Langevin dynamics, we keep the neural network weights *fixed* and calculate the gradient of the *output* with respect to the *input*. Then we update the input a small amount in the direction of the negative gradient, so that over many iterations, we gradually minimize the output (the energy score).

Both processes utilize the same idea (gradient descent), but are applied to different functions and with respect to different entities.

Formally, Langevin dynamics can be described by the following equation:

$$x^k = x^{k-1} - \eta \nabla_x E_\theta\left(x^{k-1}\right) + \omega$$

where $\omega \sim \mathcal{N}(0, \sigma)$ and $x^0 \sim \mathcal{U}(-1,1)$. η is the step size hyperparameter that must be tuned—too large and the steps jump over minima, too small and the algorithm will be too slow to converge.

$x^0 \sim \mathcal{U}(-1,1)$ is the uniform distribution on the range [–1, 1].

We can code up our Langevin sampling function as illustrated in Example 7-4.

Example 7-4. The Langevin sampling function

```
def generate_samples(model, inp_imgs, steps, step_size, noise):
    imgs_per_step = []
    for _ in range(steps): ❶
        inp_imgs += tf.random.normal(inp_imgs.shape, mean = 0, stddev = noise) ❷
        inp_imgs = tf.clip_by_value(inp_imgs, -1.0, 1.0)
        with tf.GradientTape() as tape:
            tape.watch(inp_imgs)
            out_score = -model(inp_imgs) ❸
        grads = tape.gradient(out_score, inp_imgs) ❹
        grads = tf.clip_by_value(grads, -0.03, 0.03)
        inp_imgs += -step_size * grads ❺
        inp_imgs = tf.clip_by_value(inp_imgs, -1.0, 1.0)
        return inp_imgs
```

❶ Loop over given number of steps.

❷ Add a small amount of noise to the image.

❸ Pass the image through the model to obtain the energy score.

❹ Calculate the gradient of the output with respect to the input.

❺ Add a small amount of the gradient to the input image.

Training with Contrastive Divergence

Now that we know how to sample a novel low-energy point from the sample space, let's turn our attention to training the model.

We cannot apply maximum likelihood estimation, because the energy function does not output a probability; it outputs a score that does not integrate to 1 across the sample space. Instead, we will apply a technique first proposed in 2002 by Geoffrey Hinton, called *contrastive divergence*, for training unnormalized scoring models.[4]

The value that we want to minimize (as always) is the negative log-likelihood of the data:

$$\mathscr{L} = -\mathbb{E}_{x \sim \text{data}}\left[\log p_\theta(\mathbf{x})\right]$$

When $p_\theta(\mathbf{x})$ has the form of a Boltzmann distribution, with energy function $E_\theta(\mathbf{x})$, it can be shown that the gradient of this value can be written as follows (Oliver Woodford's "Notes on Contrastive Divergence" for the full derivation):[5]

$$\nabla_\theta \mathscr{L} = \mathbb{E}_{x \sim \text{data}}\left[\nabla_\theta E_\theta(\mathbf{x})\right] - \mathbb{E}_{x \sim \text{model}}\left[\nabla_\theta E_\theta(\mathbf{x})\right]$$

This intuitively makes a lot of sense—we want to train the model to output large negative energy scores for real observations and large positive energy scores for generated fake observations so that the contrast between these two extremes is as large as possible.

In other words, we can calculate the difference between the energy scores of real and fake samples and use this as our loss function.

To calculate the energy scores of fake samples, we would need to be able to sample exactly from the distribution $p_\theta(\mathbf{x})$, which isn't possible due to the intractable denominator. Instead, we can use our Langevin sampling procedure to generate a set of observations with low energy scores. The process would need to run for infinitely many steps to produce a perfect sample (which is obviously impractical), so instead we run for some small number of steps, on the assumption that this is good enough to produce a meaningful loss function.

We also maintain a buffer of samples from previous iterations, so that we can use this as the starting point for the next batch, rather than pure random noise. The code to produce the sampling buffer is shown in Example 7-5.

CHAPTER 8

Diffusion Models

Chapter Goals

In this chapter you will:

- Learn the underlying principles and components that define a diffusion model.
- See how the forward process is used to add noise to the training set of images.
- Understand the reparameterization trick and why it is important.
- Explore different forms of forward diffusion scheduling.
- Understand the reverse diffusion process and how it relates to the forward noising process.
- Explore the architecture of the U-Net, which is used to parameterize the reverse diffusion process.
- Build your own denoising diffusion model (DDM) using Keras to generate images of flowers.
- Sample new images of flowers from your model.
- Explore the effect of the number of diffusion steps on image quality and interpolate between two images in the latent space.

Alongside GANs, diffusion models are one of the most influential and impactful generative modeling techniques for image generation to have been introduced over the last decade. Across many benchmarks, diffusion models now outperform previously state-of-the-art GANs and are quickly becoming the go-to choice for generative modeling practitioners, particularly for visual domains (e.g., OpenAI's DALL.E 2 and Google's ImageGen for text-to-image generation). Recently, there has been an explosion of diffusion models being applied across wide range of tasks, reminiscent of the GAN proliferation that took place between 2017–2020.

Many of the core ideas that underpin diffusion models share similarities with earlier types of generative models that we have already explored in this book (e.g., denoising autoencoders, energy-based models). Indeed, the name *diffusion* takes inspiration from the well-studied property of thermodynamic diffusion: an important link was made between this purely physical field and deep learning in 2015.[1]

Important progress was also being made in the field of score-based generative models,[2,3] a branch of energy-based modeling that directly estimates the gradient of the log distribution (also known as the score function) in order to train the model, as an alternative to using contrastive divergence. In particular, Yang Song and Stefano Ermon used multiple scales of noise perturbations applied to the raw data to ensure the model—a *noise conditional score network* (NCSN)—performs well on regions of low data density.

The breakthrough diffusion model paper came in the summer of 2020.[4] Standing on the shoulders of earlier works, the paper uncovers a deep connection between diffusion models and score-based generative models, and the authors use this fact to train a diffusion model that can rival GANs across several datasets, called the *Denoising Diffusion Probabilistic Model* (DDPM).

This chapter will walk through the theoretical requirements for understanding how a denoising diffusion model works. You will then learn how to build your own denoising diffusion model using Keras.

Introduction

To help explain the key ideas that underpin diffusion models, let's begin with a short story!

DiffuseTV

You are standing in an electronics store that sells television sets. However, this store is clearly very different from ones you have visited in the past. Instead of a wide variety of different brands, there are hundreds of identical copies of the same TV connected together in sequence, stretching into the back of the shop as far as you can see. What's more, the first few TV sets appear to be showing nothing but random static noise (Figure 8-1).

The shopkeeper comes over to ask if you need assistance. Confused, you ask her about the odd setup. She explains that this is the new DiffuseTV model that is set to revolutionize the entertainment industry and immediately starts telling you how it works, while walking deeper into the shop, alongside the line of TVs.

Figure 8-1. A long line of connected television sets stretching out along an aisle of a shop (created with Midjourney)

She explains that during the manufacturing process, the DiffuseTV is exposed to thousands of images of previous TV shows—but each of those images has been gradually corrupted with random static, until it is indistinguishable from pure random noise. The TVs are then designed to *undo* the random noise, in small steps, essentially trying to predict what the images looked like before the noise was added. You can see that as you walk further into the shop the images on each television set are indeed slightly clearer than the last.

You eventually reach the end of the long line of televisions, where you can see a perfect picture on the last set. While this is certainly clever technology, you are curious to understand how this is useful to the viewer. The shopkeeper continues with her explanation.

Instead of choosing a channel to watch, the viewer chooses a random initial configuration of static. Every configuration will lead to a different output image, and in some models can even be guided by a text prompt that you choose to input. Unlike a normal TV, with a limited range of channels to watch, the DiffuseTV gives the viewer unlimited choice and freedom to generate whatever they would like to appear on the screen!

You purchase a DiffuseTV right away and are relieved to hear that the long line of TVs in the shop is for demonstration purposes only, so you won't have to also buy a warehouse to store your new device!

The DiffuseTV story describes the general idea behind a diffusion model. Now let's dive into the technicalities of how we build such a model using Keras.

Denoising Diffusion Models (DDM)

The core idea behind a denoising diffusion model is simple—we train a deep learning model to denoise an image over a series of very small steps. If we start from pure random noise, in theory we should be able to keep applying the model until we obtain an image that looks as if it were drawn from the training set. What's amazing is that this simple concept works so well in practice!

Let's first get set up with a dataset and then walk through the forward (noising) and backward (denoising) diffusion processes.

Running the Code for This Example

The code for this example can be found in the Jupyter notebook located at *notebooks/08_diffusion/01_ddm/ddm.ipynb* in the book repository.

The code is adapted from the excellent tutorial on denoising diffusion implicit models created by András Béres available on the Keras website.

The Flowers Dataset

We'll be using the Oxford 102 Flower dataset that is available through Kaggle. This is a set of over 8,000 color images of a variety of flowers.

You can download the dataset by running the Kaggle dataset downloader script in the book repository, as shown in Example 8-1. This will save the flower images to the */data* folder.

Example 8-1. Downloading the Oxford 102 Flower dataset

```
bash scripts/download_kaggle_data.sh nunenuh pytorch-challange-flower-dataset
```

As usual, we'll load the images in using the Keras `image_dataset_from_directory` function, resize the images to 64 × 64 pixels, and scale the pixel values to the range [0, 1]. We'll also repeat the dataset five times to increase the epoch length and batch the data into groups of 64 images, as shown in Example 8-2.

Example 8-2. Loading the Oxford 102 Flower dataset

```
train_data = utils.image_dataset_from_directory(
    "/app/data/pytorch-challange-flower-dataset/dataset",
    labels=None,
    image_size=(64, 64),
    batch_size=None,
    shuffle=True,
    seed=42,
    interpolation="bilinear",
) ❶

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

train = train_data.map(lambda x: preprocess(x)) ❷
train = train.repeat(5) ❸
train = train.batch(64, drop_remainder=True) ❹
```

❶ Load dataset (when required during training) using the Keras `image_dataset_from_directory` function.

❷ Scale the pixel values to the range [0, 1].

❸ Repeat the dataset five times.

❹ Batch the dataset into groups of 64 images.

Example images from the dataset are shown in Figure 8-2.

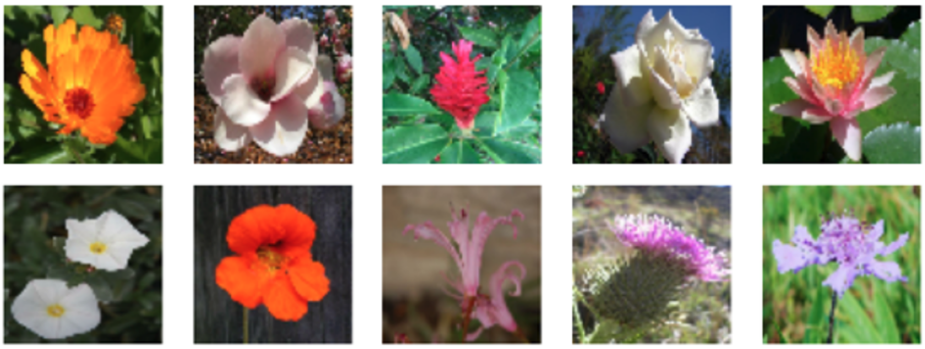

Figure 8-2. Example images from the Oxford 102 Flower dataset

Now that we have our dataset we can explore how we should add noise to the images, using a forward diffusion process.

The Forward Diffusion Process

Suppose we have an image $\mathbf{x}_0$ that we want to corrupt gradually over a large number of steps (say, $T = 1,000$), so that eventually it is indistinguishable from standard Gaussian noise (i.e., $\mathbf{x}_T$ should have zero mean and unit variance). How should we go about doing this?

We can define a function q that adds a small amount of Gaussian noise with variance β_t to an image $\mathbf{x}_{t-1}$ to generate a new image $\mathbf{x}_t$. If we keep applying this function, we will generate a sequence of progressively noisier images $(\mathbf{x}_0, ..., \mathbf{x}_T)$, as shown in Figure 8-3.

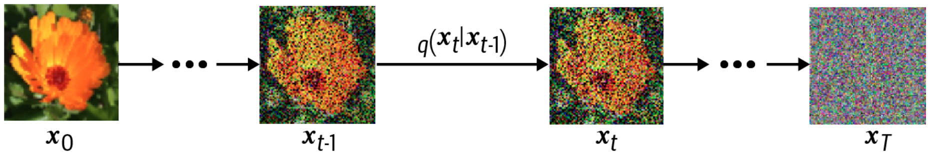

Figure 8-3. The forward diffusion process q

We can write this update process mathematically as follows (here, ϵ_{t-1} is a standard Gaussian with zero mean and unit variance):

$$\mathbf{x}_t = \sqrt{1-\beta_t}\mathbf{x}_{t-1} + \sqrt{\beta_t}\epsilon_{t-1}$$

Note that we also scale the input image $\mathbf{x}_{t-1}$, to ensure that the variance of the output image $\mathbf{x}_t$ remains constant over time. This way, if we normalize our original image $\mathbf{x}_0$ to have zero mean and unit variance, then $\mathbf{x}_T$ will approximate a standard Gaussian distribution for large enough T, by induction, as follows.

If we assume that $\mathbf{x}_{t-1}$ has zero mean and unit variance then $\sqrt{1-\beta_t}\mathbf{x}_{t-1}$ will have variance $1-\beta_t$ and $\sqrt{\beta_t}\epsilon_{t-1}$ will have variance β_t, using the rule that $Var(aX) = a^2Var(X)$. Adding these together, we obtain a new distribution $\mathbf{x}_t$ with zero mean and variance $1-\beta_t+\beta_t = 1$, using the rule that $Var(X+Y) = Var(X) + Var(Y)$ for independent X and Y. Therefore, if $\mathbf{x}_0$ is normalized to a zero mean and unit variance, then we guarantee that this is also true for all $\mathbf{x}_t$, including the final image $\mathbf{x}_T$, which will approximate a standard Gaussian distribution. This is exactly what we need, as we want to be able to easily sample $\mathbf{x}_T$ and then apply a reverse diffusion process through our trained neural network model!

In other words, our forward noising process q can also be written as follows:

$$q\left(\mathbf{x}_t \middle| \mathbf{x}_{t-1}\right) = \mathcal{N}\left(\mathbf{x}_t; \sqrt{1-\beta_t}\mathbf{x}_{t-1}, \beta_t\mathbf{I}\right)$$

The Reparameterization Trick

It would also be useful to be able to jump straight from an image $\mathbf{x}_0$ to any noised version of the image $\mathbf{x}_t$ without having to go through t applications of q. Luckily, there is a reparameterization trick that we can use to do this.

If we define $\alpha_t = 1 - \beta_t$ and $\bar{\alpha}_t = \prod_{i=1}^{t} \alpha_i$, then we can write the following:

$$\begin{aligned} \mathbf{x}_t &= \sqrt{\alpha_t}\mathbf{x}_{t-1} + \sqrt{1-\alpha_t}\epsilon_{t-1} \\ &= \sqrt{\alpha_t \alpha_{t-1}}\mathbf{x}_{t-2} + \sqrt{1-\alpha_t \alpha_{t-1}}\epsilon \\ &= \cdots \\ &= \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\epsilon \end{aligned}$$

Note that the second line uses the fact that we can add two Gaussians to obtain a new Gaussian. We therefore have a way to jump from the original image $\mathbf{x}_0$ to any step of the forward diffusion process $\mathbf{x}_t$. Moreover, we can define the diffusion schedule using the $\bar{\alpha}_t$ values, instead of the original β_t values, with the interpretation that $\bar{\alpha}_t$ is the variance due to the signal (the original image, $\mathbf{x}_0$) and $1 - \bar{\alpha}_t$ is the variance due to the noise (ϵ).

The forward diffusion process q can therefore also be written as follows:

$$q\left(\mathbf{x}_t \mid \mathbf{x}_0\right) = \mathcal{N}\left(\mathbf{x}_t; \sqrt{\bar{\alpha}_t}\mathbf{x}_0, \left(1 - \bar{\alpha}_t\right)\mathbf{I}\right)$$

Diffusion Schedules

Notice that we are also free to choose a different β_t at each timestep—they don't all have be the same. How the β_t (or $\bar{\alpha}_t$) values change with t is called the *diffusion schedule*.

In the original paper (Ho et al., 2020), the authors chose a *linear diffusion schedule* for β_t—that is, β_t increases linearly with t, from $\beta_1 = 0.0001$ to $\beta_T = 0.02$. This ensures that in the early stages of the noising process we take smaller noising steps than in the later stages, when the image is already very noisy.

We can code up a linear diffusion schedule as shown in Example 8-3.

Example 8-3. The linear diffusion schedule

```
def linear_diffusion_schedule(diffusion_times):
    min_rate = 0.0001
    max_rate = 0.02
    betas = min_rate + tf.convert_to_tensor(diffusion_times) * (max_rate - min_rate)
    alphas = 1 - betas
    alpha_bars = tf.math.cumprod(alphas)
    signal_rates = alpha_bars
    noise_rates = 1 - alpha_bars
    return noise_rates, signal_rates
```

```
T = 1000
diffusion_times = [x/T for x in range(T)] ❶
linear_noise_rates, linear_signal_rates = linear_diffusion_schedule(
    diffusion_times
) ❷
```

❶ The diffusion times are equally spaced steps between 0 and 1.

❷ The linear diffusion schedule is applied to the diffusion times to produce the noise and signal rates.

In a later paper it was found that a *cosine diffusion schedule* outperformed the linear schedule from the original paper.[5] A cosine schedule defines the following values of $\bar{\alpha}_t$:

$$\bar{\alpha}_t = \cos^2\left(\frac{t}{T} \cdot \frac{\pi}{2}\right)$$

The updated equation is therefore as follows (using the trigonometric identity $\cos^2(x) + \sin^2(x) = 1$):

$$\mathbf{x}_t = \cos\left(\frac{t}{T} \cdot \frac{\pi}{2}\right)\mathbf{x}_0 + \sin\left(\frac{t}{T} \cdot \frac{\pi}{2}\right)\epsilon$$

This equation is a simplified version of the actual cosine diffusion schedule used in the paper. The authors also add an offset term and scaling to prevent the noising steps from being too small at the beginning of the diffusion process. We can code up the cosine and offset cosine diffusion schedules as shown in Example 8-4.

Example 8-4. The cosine and offset cosine diffusion schedules

```
def cosine_diffusion_schedule(diffusion_times): ❶
    signal_rates = tf.cos(diffusion_times * math.pi / 2)
    noise_rates = tf.sin(diffusion_times * math.pi / 2)
    return noise_rates, signal_rates

def offset_cosine_diffusion_schedule(diffusion_times): ❷
    min_signal_rate = 0.02
    max_signal_rate = 0.95
    start_angle = tf.acos(max_signal_rate)
    end_angle = tf.acos(min_signal_rate)

    diffusion_angles = start_angle + diffusion_times * (end_angle - start_angle)

    signal_rates = tf.cos(diffusion_angles)
    noise_rates = tf.sin(diffusion_angles)
    return noise_rates, signal_rates
```

❶ The pure cosine diffusion schedule (without offset or rescaling).

❷ The offset cosine diffusion schedule that we will be using, which adjusts the schedule to ensure the noising steps are not too small at the start of the noising process.

We can compute the $\bar{\alpha}_t$ values for each t to show how much signal ($\bar{\alpha}_t$) and noise ($1 - \bar{\alpha}_t$) is let through at each stage of the process for the linear, cosine, and offset cosine diffusion schedules, as shown in Figure 8-4.

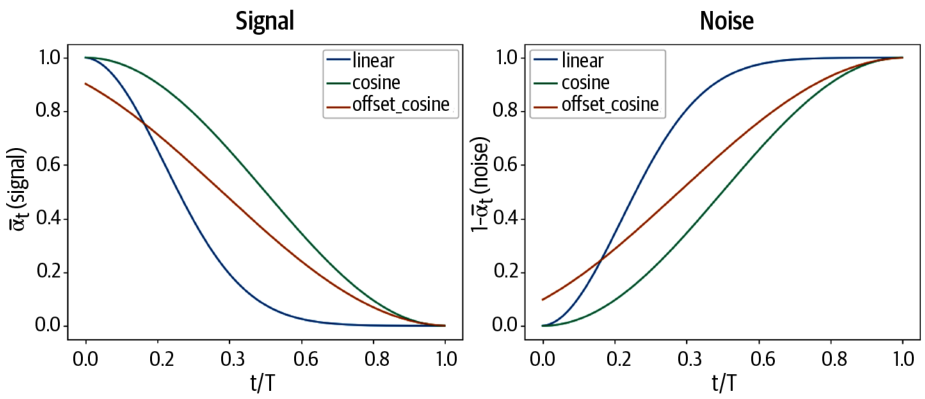

Figure 8-4. The signal and noise at each step of the noising process, for the linear, cosine, and offset cosine diffusion schedules

Notice how the noise level ramps up more slowly in the cosine diffusion schedule. A cosine diffusion schedule adds noise to the image more gradually than a linear diffusion schedule, which improves training efficiency and generation quality. This can also be seen in images that have been corrupted by the linear and cosine schedules (Figure 8-5).

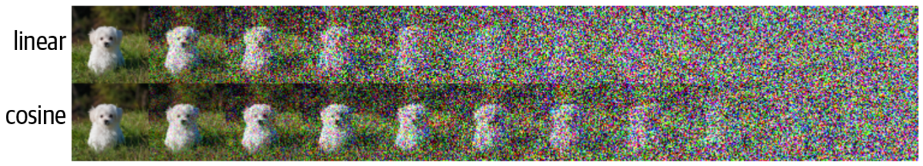

Figure 8-5. An image being corrupted by the linear (top) and cosine (bottom) diffusion schedules, at equally spaced values of t from 0 to T (source: Ho et al., 2020)

The Reverse Diffusion Process

Now let's look at the reverse diffusion process. To recap, we are looking to build a neural network $p_\theta(\mathbf{x}_{t-1} | \mathbf{x}_t)$ that can *undo* the noising process—that is, approximate the reverse distribution $q(\mathbf{x}_{t-1} | \mathbf{x}_t)$. If we can do this, we can sample random noise from $\mathcal{N}(0, \mathbf{I})$ and then apply the reverse diffusion process multiple times in order to generate a novel image. This is visualized in Figure 8-6.

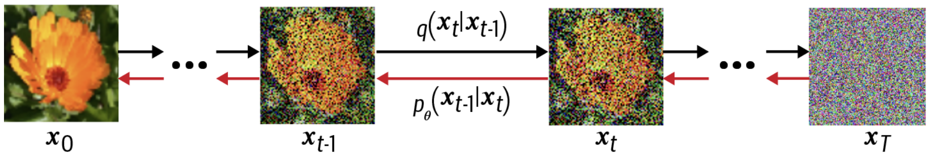

Figure 8-6. The reverse diffusion process $p_\theta.(\mathbf{x}_{t-1} | \mathbf{x}_t)$ tries to undo the noise produced by the forward diffusion process

There are many similarities between the reverse diffusion process and the decoder of a variational autoencoder. In both, we aim to transform random noise into meaningful output using a neural network. The difference between diffusion models and VAEs is that in a VAE the forward process (converting images to noise) is part of the model (i.e., it is learned), whereas in a diffusion model it is unparameterized.

Therefore, it makes sense to apply the same loss function as in a variational autoencoder. The original DDPM paper derives the exact form of this loss function and shows that it can be optimized by training a network ϵ_θ to predict the noise ϵ that has been added to a given image $\mathbf{x_0}$ at timestep t.

In other words, we sample an image $\mathbf{x_0}$ and transform it by t noising steps to get the image $\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1 - \bar{\alpha}_t}\epsilon$. We give this new image and the noising rate $\bar{\alpha}_t$ to the neural network and ask it to predict ϵ, taking a gradient step against the squared error between the prediction $\epsilon_\theta(\mathbf{x}_t)$ and the true ϵ.

We'll take a look at the structure of the neural network in the next section. It is worth noting here that the diffusion model actually maintains two copies of the network: one that is actively trained used gradient descent and another (the EMA network) that is an exponential moving average (EMA) of the weights of the actively trained network over previous training steps. The EMA network is not as susceptible to short-term fluctuations and spikes in the training process, making it more robust for generation than the actively trained network. We therefore use the EMA network whenever we want to produce generated output from the network.

The training process for the model is shown in Figure 8-7.

```
Algorithm 1 Training
1: repeat
2:   $\mathbf{x}_0 \sim q(\mathbf{x}_0)$
3:   $t \sim \mathrm{Uniform}(\{1, \ldots, T\})$
4:   $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
5:   Take gradient descent step on
         $\nabla_\theta \left\| \boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t) \right\|^2$
6: until converged
```

Figure 8-7. The training process for a denoising diffusion model (source: Ho et al., 2020)

In Keras, we can code up this training step as illustrated in Example 8-5.

Example 8-5. The `train_step` function of the Keras diffusion model

```
class DiffusionModel(models.Model):
    def __init__(self):
        super().__init__()
        self.normalizer = layers.Normalization()
        self.network = unet
        self.ema_network = models.clone_model(self.network)
        self.diffusion_schedule = cosine_diffusion_schedule

    ...

    def denoise(self, noisy_images, noise_rates, signal_rates, training):
        if training:
            network = self.network
        else:
            network = self.ema_network
        pred_noises = network(
            [noisy_images, noise_rates**2], training=training
        )
        pred_images = (noisy_images - noise_rates * pred_noises) / signal_rates

        return pred_noises, pred_images

    def train_step(self, images):
        images = self.normalizer(images, training=True) ❶
        noises = tf.random.normal(shape=tf.shape(images)) ❷
        batch_size = tf.shape(images)[0]
        diffusion_times = tf.random.uniform(
            shape=(batch_size, 1, 1, 1), minval=0.0, maxval=1.0
        ) ❸
        noise_rates, signal_rates = self.cosine_diffusion_schedule(
            diffusion_times
        ) ❹
        noisy_images = signal_rates * images + noise_rates * noises ❺
        with tf.GradientTape() as tape:
            pred_noises, pred_images = self.denoise(
                noisy_images, noise_rates, signal_rates, training=True
            ) ❻
            noise_loss = self.loss(noises, pred_noises)  ❼
        gradients = tape.gradient(noise_loss, self.network.trainable_weights)
        self.optimizer.apply_gradients(
            zip(gradients, self.network.trainable_weights)
        ) ❽
        self.noise_loss_tracker.update_state(noise_loss)

        for weight, ema_weight in zip(
            self.network.weights, self.ema_network.weights
        ):
            ema_weight.assign(0.999 * ema_weight + (1 - 0.999) * weight) ❾

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

    ...
```

❶ We first normalize the batch of images to have zero mean and unit variance.

❷ Next, we sample noise to match the shape of the input images.

❸ We also sample random diffusion times…

❹ …and use these to generate the noise and signal rates according to the cosine diffusion schedule.

❺ Then we apply the signal and noise weightings to the input images to generate the noisy images.

❻ Next, we denoise the noisy images by asking the network to predict the noise and then undoing the noising operation, using the provided `noise_rates` and `signal_rates`.

❼ We can then calculate the loss (mean absolute error) between the predicted noise and the true noise…

❽ …and take a gradient step against this loss function.

❾ The EMA network weights are updated to a weighted average of the existing EMA weights and the trained network weights after the gradient step.