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.

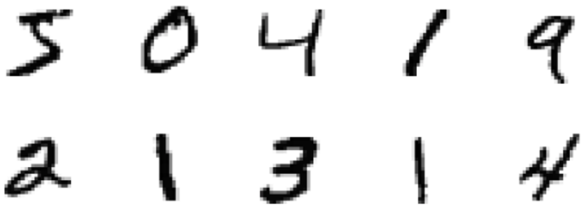

Figure 7-2. Examples of images from the MNIST dataset

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.

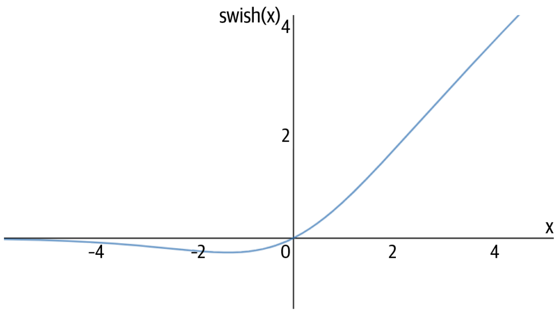

Figure 7-3. The swish activation function

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 $(-\infty, \infty)$. 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!

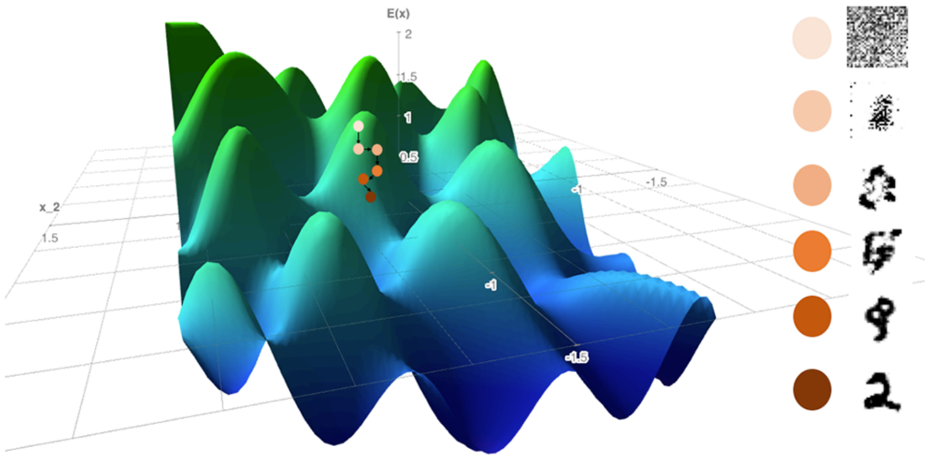

Figure 7-4. Gradient descent using Langevin dynamics

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.

Example 7-5. The `Buffer`

```
class Buffer:
    def __init__(self, model):
        super().__init__()
        self.model = model
        self.examples = [
            tf.random.uniform(shape = (1, 32, 32, 1)) * 2 - 1
            for _ in range(128)
        ] ❶

    def sample_new_exmps(self, steps, step_size, noise):
        n_new = np.random.binomial(128, 0.05) ❷
        rand_imgs = (
            tf.random.uniform((n_new, 32, 32, 1)) * 2 - 1
        )
        old_imgs = tf.concat(
            random.choices(self.examples, k=128-n_new), axis=0
        ) ❸
        inp_imgs = tf.concat([rand_imgs, old_imgs], axis=0)
        inp_imgs = generate_samples(
            self.model, inp_imgs, steps=steps, step_size=step_size, noise = noise
        ) ❹
        self.examples = tf.split(inp_imgs, 128, axis = 0) + self.examples ❺
        self.examples = self.examples[:8192]
        return inp_imgs
```

❶ The sampling buffer is initialized with a batch of random noise.

❷ On average, 5% of observations are generated from scratch (i.e., random noise) each time.

❸ The rest are pulled at random from the existing buffer.

❹ The observations are concatenated and run through the Langevin sampler.

❺ The resulting sample is added to the buffer, which is trimmed to a max length of 8,192 observations.

Figure 7-5 shows one training step of contrastive divergence. The scores of real observations are pushed down by the algorithm and the scores of fake observations are pulled up, without caring about normalizing these scores after each step.

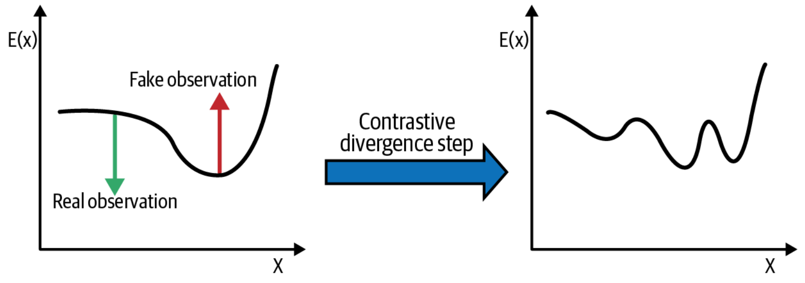

Figure 7-5. One step of contrastive divergence

We can code up the training step of the contrastive divergence algorithm within a custom Keras model as shown in Example 7-6.

Example 7-6. EBM trained using contrastive divergence

```
class EBM(models.Model):
    def __init__(self):
        super(EBM, self).__init__()
        self.model = model
        self.buffer = Buffer(self.model)
        self.alpha = 0.1
        self.loss_metric = metrics.Mean(name="loss")
        self.reg_loss_metric = metrics.Mean(name="reg")
        self.cdiv_loss_metric = metrics.Mean(name="cdiv")
        self.real_out_metric = metrics.Mean(name="real")
        self.fake_out_metric = metrics.Mean(name="fake")

    @property
    def metrics(self):
        return [
            self.loss_metric,
            self.reg_loss_metric,
            self.cdiv_loss_metric,
            self.real_out_metric,
            self.fake_out_metric
        ]

    def train_step(self, real_imgs):
        real_imgs += tf.random.normal(
            shape=tf.shape(real_imgs), mean = 0, stddev = 0.005
        ) ❶
        real_imgs = tf.clip_by_value(real_imgs, -1.0, 1.0)
        fake_imgs = self.buffer.sample_new_exmps(
            steps=60, step_size=10, noise = 0.005
        ) ❷
        inp_imgs = tf.concat([real_imgs, fake_imgs], axis=0)
        with tf.GradientTape() as training_tape:
            real_out, fake_out = tf.split(self.model(inp_imgs), 2, axis=0) ❸
            cdiv_loss = tf.reduce_mean(fake_out, axis = 0) - tf.reduce_mean(
                real_out, axis = 0
            ) ❹
            reg_loss = self.alpha * tf.reduce_mean(
                real_out ** 2 + fake_out ** 2, axis = 0
            ) ❺
            loss = reg_loss + cdiv_loss
        grads = training_tape.gradient(loss, self.model.trainable_variables) ❻
        self.optimizer.apply_gradients(
            zip(grads, self.model.trainable_variables)
        )
        self.loss_metric.update_state(loss)
        self.reg_loss_metric.update_state(reg_loss)
        self.cdiv_loss_metric.update_state(cdiv_loss)
        self.real_out_metric.update_state(tf.reduce_mean(real_out, axis = 0))
        self.fake_out_metric.update_state(tf.reduce_mean(fake_out, axis = 0))
        return {m.name: m.result() for m in self.metrics}

    def test_step(self, real_imgs): ❼
        batch_size = real_imgs.shape[0]
        fake_imgs = tf.random.uniform((batch_size, 32, 32, 1)) * 2 - 1
        inp_imgs = tf.concat([real_imgs, fake_imgs], axis=0)
        real_out, fake_out = tf.split(self.model(inp_imgs), 2, axis=0)
        cdiv = tf.reduce_mean(fake_out, axis = 0) - tf.reduce_mean(
            real_out, axis = 0
        )
        self.cdiv_loss_metric.update_state(cdiv)
        self.real_out_metric.update_state(tf.reduce_mean(real_out, axis = 0))
        self.fake_out_metric.update_state(tf.reduce_mean(fake_out, axis = 0))
        return {m.name: m.result() for m in self.metrics[2:]}

ebm = EBM()
ebm.compile(optimizer=optimizers.Adam(learning_rate=0.0001), run_eagerly=True)
ebm.fit(x_train, epochs=60, validation_data = x_test,)
```

❶ A small amount of random noise is added to the real images, to avoid the model overfitting to the training set.

❷ A set of fake images are sampled from the buffer.

❸ The real and fake images are run through the model to produce real and fake scores.

❹ The contrastive divergence loss is simply the difference between the scores of real and fake observations.

❺ A regularization loss is added to avoid the scores becoming too large.

❻ Gradients of the loss function with respect to the weights of the network are calculated for backpropagation.

❼ The `test_step` is used during validation and calculates the contrastive divergence between the scores of a set of random noise and data from the training set. It can be used as a measure for how well the model is training (see the following section).

Analysis of the Energy-Based Model

The loss curves and supporting metrics from the training process are shown in Figure 7-6.

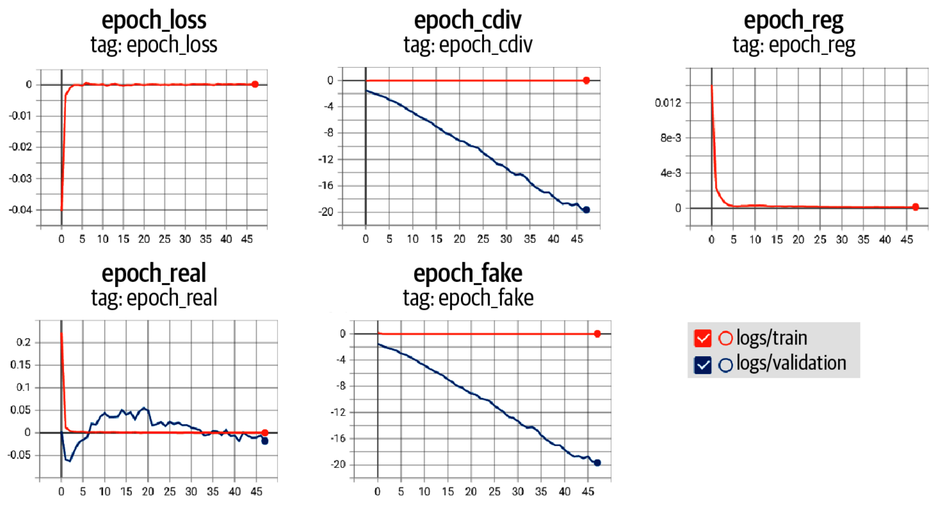

Figure 7-6. Loss curves and metrics for the training process of the EBM

Firstly, notice that the loss calculated during the training step is approximately constant and small across epochs. While the model is constantly improving, so is the quality of generated images in the buffer that it is required to compare against real images from the training set, so we shouldn't expect the training loss to fall significantly.

Therefore, to judge model performance, we also set up a validation process that doesn't sample from the buffer, but instead scores a sample of random noise and compares this against the scores of examples from the training set. If the model is improving, we should see that the contrastive divergence falls over the epochs (i.e., it is getting better at distinguishing random noise from real images), as can be seen in Figure 7-6.

Generating new samples from the EBM is simply a case of running the Langevin sampler for a large number of steps, from a standing start (random noise), as shown in Example 7-7. The observation is forced *downhill*, following the gradients of the scoring function with respect to the input, so that out of the noise, a plausible observation appears.

Example 7-7. Generating new observations using the EBM

```
start_imgs = np.random.uniform(size = (10, 32, 32, 1)) * 2 - 1
gen_img = generate_samples(
    ebm.model,
    start_imgs,
    steps=1000,
    step_size=10,
    noise = 0.005,
    return_img_per_step=True,
)
```

Some examples of observations produced by the sampler after 50 epochs of training are shown in Figure 7-7.

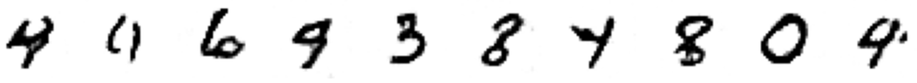

Figure 7-7. Examples produced by the Langevin sampler using the EBM model to direct the gradient descent

We can even show a replay of how a single observation is generated by taking snapshots of the current observations during the Langevin sampling process—this is shown in Figure 7-8.

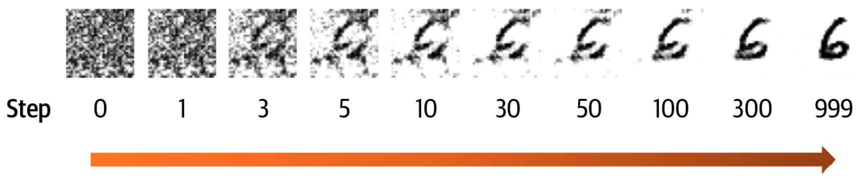

Figure 7-8. Snapshots of an observation at different steps of the Langevin sampling process

Other Energy-Based Models

In the previous example we made use of a deep EBM trained using contrastive divergence with a Langevin dynamics sampler. However, early EBM models did not make use of Langevin sampling, but instead relied on other techniques and architectures.

One of the earliest examples of an EBM was the *Boltzmann machine*.[6] This is a fully connected, undirected neural network, where binary units are either *visible* (v) or *hidden* (h). The energy of a given configuration of the network is defined as follows:

$$E_\theta(v, h) = -\frac{1}{2}\left(v^T L v + h^T J h + v^T W h\right)$$

where W, L, J are the weights matrices that are learned by the model. Training is achieved by contrastive divergence, but using Gibbs sampling to alternate between the visible and hidden layers until an equilibrium is found. In practice this is very slow and not scalable to large numbers of hidden units.

See Jessica Stringham's blog post "Gibbs Sampling in Python" for an excellent simple example of Gibbs sampling.

An extension to this model, the *restricted Boltzmann machine* (RBM), removes the connections between units of the same type, therefore creating a two-layer bipartite graph. This allows RBMs to be stacked into *deep belief networks* to model more complex distributions. However, modeling high-dimensional data with RBMs remains impractical, due to the fact that Gibbs sampling with long mixing times is still required.

It was only in the late 2000s that EBMs were shown to have potential for modeling more high-dimensional datasets and a framework for building deep EBMs was established.[7] Langevin dynamics became the preferred sampling method for EBMs, which later evolved into a training technique known as *score matching*. This further developed into a model class known as *Denoising Diffusion Probabilistic Models*, which power state-of-the-art generative models such as DALL.E 2 and ImageGen. We will explore diffusion models in more detail in Chapter 8.

Summary

Energy-based models are a class of generative model that make use of an energy scoring function—a neural network that is trained to output low scores for real observations and high scores for generated observations. Calculating the probability distribution given by this score function would require normalizing by an intractable denominator. EBMs avoid this problem by utilizing two tricks: contrastive divergence for training the network and Langevin dynamics for sampling new observations.

The energy function is trained by minimizing the difference between the generated sample scores and the scores of the training data, a technique known as contrastive divergence. This can be shown to be equivalent to minimizing the negative log-likelihood, as required by maximum likelihood estimation, but does not require us to calculate the intractable normalizing denominator. In practice, we approximate the sampling process for the fake samples to ensure the algorithm remains efficient.

Sampling of deep EBMs is achieved through Langevin dynamics, a technique that uses the gradient of the score with respect to the input image to gradually transform random noise into a plausible observation by updating the input in small steps, following the gradient downhill. This improves upon earlier methods such as Gibbs sampling, which is utilized by restricted Boltzmann machines.

References

1. Yilun Du and Igor Mordatch, "Implicit Generation and Modeling with Energy-Based Models," March 20, 2019, *https://arxiv.org/abs/1903.08689*.

2. Prajit Ramachandran et al., "Searching for Activation Functions," October 16, 2017, *https://arxiv.org/abs/1710.05941v2*.

3. Max Welling and Yee Whye Teh, "Bayesian Learning via Stochastic Gradient Langevin Dynamics," 2011, *https://www.stats.ox.ac.uk/~teh/research/compstats/WelTeh2011a.pdf*

4. Geoffrey E. Hinton, "Training Products of Experts by Minimizing Contrastive Divergence," 2002, *https://www.cs.toronto.edu/~hinton/absps/tr00-004.pdf*.

5. Oliver Woodford, "Notes on Contrastive Divergence," 2006, *https://www.robots.ox.ac.uk/~ojw/files/NotesOnCD.pdf*.

6. David H. Ackley et al., "A Learning Algorithm for Boltzmann Machines," 1985, *Cognitive Science* 9(1), 147-165.

7. Yann Lecun et al., "A Tutorial on Energy-Based Learning," 2006, *https://www.researchgate.net/publication/200744586_A_tutorial_on_energy-based_learning*.