# Chapter 18. Autoencoders, GANs, and Diffusion Models

Autoencoders are artificial neural networks capable of learning dense representations of the input data, called *latent representations* or *codings*, without any supervision (i.e., the training set is unlabeled). These codings typically have a much lower dimensionality than the input data, making autoencoders useful for dimensionality reduction (see Chapter 7), especially for visualization purposes. Autoencoders also act as feature detectors, and they can be used for unsupervised pretraining of deep neural networks (as we discussed in Chapter 11). They are also commonly used for anomaly detection, as we will see. Lastly, some autoencoders are *generative models*: they are capable of randomly generating new data that looks very similar to the training data. For example, you could train an autoencoder on pictures of faces, and it would then be able to generate new faces.

*Generative adversarial networks* (GANs) are also neural nets capable of generating data. In fact, they can generate pictures of faces so convincing that it is hard to believe the people they represent do not exist. You can judge so for yourself by visiting *https://thispersondoesnotexist.com*, a website that shows faces generated by a GAN architecture called *StyleGAN*. You can also check out *https://thisrentaldoesnotexist.com* to see some generated Airbnb listings. GANs were widely used for super resolution (increasing the resolution of an image), colorization, powerful image editing (e.g., replacing photo bombers with realistic background), turning simple sketches into photorealistic images, predicting the next frames in a video, augmenting a dataset (to train other models), generating other types of data (such as text, audio, and time series), identifying the weaknesses in other models to strengthen them, and more.

However, since the early 2020s, GANs have largely been replaced by *diffusion models*, which can generate more diverse and higher-quality images than GANs, while also being much easier to train. However, diffusion models are much slower to run, so GANs are still useful when you need very fast generation.

Autoencoders, GANs, and diffusion models are all unsupervised, they all learn latent representations, they can all be used as generative models, and they have many similar applications. However, they work very differently:

*Autoencoders*

> Autoencoders simply learn to copy their inputs to their outputs. This may sound like a trivial task, but as you will see, constraining the network in various ways can make the task arbitrarily difficult. For example, you can limit the size of the latent representations, or you can add noise to the inputs and train the network to recover the original inputs. These constraints prevent the autoencoder from trivially copying the inputs directly to the outputs, which forces it to learn efficient ways of representing the data. In

short, the codings are byproducts of the autoencoder learning the identity function under some constraints.

*GANs*

GANs are composed of two neural networks: a *generator* that tries to generate data that looks similar to the training data, and a *discriminator* that tries to tell real data from fake data. This architecture is very original in deep learning in that the generator and the discriminator compete against each other during training: this is called *adversarial training*. The generator is often compared to a criminal trying to make realistic counterfeit money, while the discriminator is like the police investigator trying to tell real money from fake.

*Diffusion models*

A diffusion model is trained to gradually remove noise from an image. If you then take an image entirely full of random noise and repeatedly run the diffusion model on that image, a high-quality image will gradually emerge, similar to the training images (but not identical).

In this chapter we will start by exploring in more depth how autoencoders work and how to use them for dimensionality reduction, feature extraction, unsupervised pretraining, or as generative models. This will naturally lead us to GANs. We will build a simple GAN to generate fake images, but we will see that training is often quite difficult. We will discuss the main difficulties you will encounter with adversarial training, as well as some of the main techniques to work around these difficulties. And lastly, we will build and train a diffusion model—specifically a *denoising diffusion probabilistic model* (DDPM)—and use it to generate images. Let's start with autoencoders!

# Efficient Data Representations

Which of the following number sequences do you find the easiest to memorize?

- 40, 27, 25, 36, 81, 57, 10, 73, 19, 68
- 50, 48, 46, 44, 42, 40, 38, 36, 34, 32, 30, 28, 26, 24, 22, 20, 18, 16, 14

At first glance, it would seem that the first sequence should be easier, since it is much shorter. However, if you look carefully at the second sequence, you will notice that it is just the list of even numbers from 50 down to 14. Once you notice this pattern, the second sequence becomes much easier to memorize than the first because you only need to remember the pattern (i.e., decreasing even numbers) and the starting and ending numbers (i.e., 50 and 14). Note that if you could quickly and easily memorize very long sequences, you would not care much about the existence of a pattern in the second sequence. You would just learn every number by heart, and that would be that. The fact that it is hard to memorize long sequences is what makes it useful to recognize patterns, and hopefully this clarifies why constraining an autoencoder during training pushes it to discover and exploit patterns in the data.

The relationship between memory, perception, and pattern matching was famously studied by William Chase and Herbert Simon[1] in the early 1970s. They observed that expert chess players were able to memorize the positions of all the pieces in a game by looking at the board for just five seconds, a task that most people would find impossible. However, this was only the case when the pieces were placed in realistic positions (from actual games), not when the pieces were placed randomly. Chess experts don't have a much better memory than you and I; they just see chess patterns more easily, thanks to their experience with the game. Noticing patterns helps them store information efficiently.

Just like the chess players in this memory experiment, an autoencoder looks at the inputs, converts them to an efficient latent representation, and it is then capable of reconstructing something that (hopefully) looks very close to the inputs. An autoencoder is always composed of two parts: an *encoder* (or *recognition network*) that converts the inputs to a latent representation, followed by a *decoder* (or *generative network*) that converts the internal representation to the outputs.

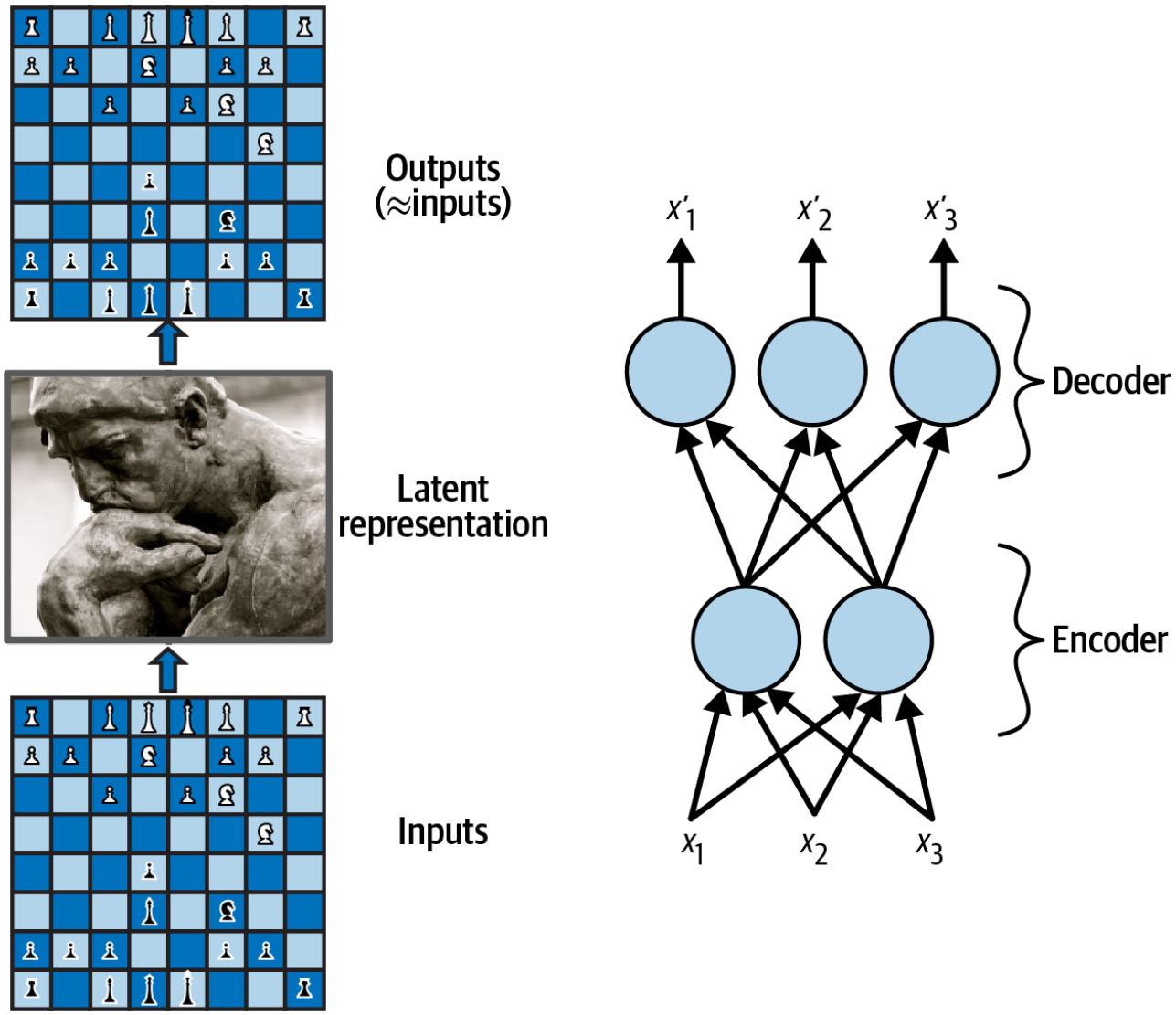



Figure 18-1. The chess memory experiment (left) and a simple autoencoder (right)

In the example shown in Figure 18-1, the autoencoder is a regular multilayer perceptron (MLP; see Chapter 9). Since it must reconstruct its inputs, the number of neurons in the output layer must be equal to the number of inputs (i.e. three in this example). The lower part of the network is the encoder (in this case it's a single layer with two neurons), and the upper part is the decoder. The outputs are often called the *reconstructions* because the autoencoder tries to re-

decoder. The outputs are often called the *reconstructions* because the autoencoder tries to reconstruct the inputs. The cost function always contains a *reconstruction loss* that penalizes the model when the reconstructions are different from the inputs.

Because the internal representation has a lower dimensionality than the input data (in this example, it is 2D instead of 3D), the autoencoder is said to be *undercomplete*. An undercomplete autoencoder cannot trivially copy its inputs to the codings, yet it must find a way to output a copy of its inputs. It is forced to compress the data, thereby learning the most important features in the input data (and dropping the unimportant ones).

Let's see how to implement a very simple undercomplete autoencoder for dimensionality reduction.

# Performing PCA with an Undercomplete Linear Autoencoder

If the autoencoder uses only linear activations and the cost function is the mean squared error (MSE), then it ends up performing principal component analysis (PCA; see Chapter 7).

The following code builds a simple linear autoencoder to perform PCA on a 3D dataset, projecting it to 2D:

```
import torch
import torch.nn as nn

torch.manual_seed(42)
encoder = nn.Linear(3, 2)
decoder = nn.Linear(2, 3)
autoencoder = nn.Sequential(encoder, decoder).to(device)
```

This code is really not very different from all the MLPs we built in past chapters, but there are a few things to note:

- We organized the autoencoder into two subcomponents: the encoder and the decoder, each composed of a single `Linear` layer in this example, and the autoencoder is a `Sequential` model containing the encoder followed by the decoder.
- The autoencoder's number of outputs is equal to the number of inputs (i.e., 3).
- To perform PCA, we do not use any activation function (i.e., all neurons are linear), and the cost function is the MSE. That's because PCA is a linear transformation. We will see more complex and nonlinear autoencoders shortly.

Now let's train the model on the same simple generated 3D dataset we used in Chapter 7 and use it to encode that dataset (i.e., project it to 2D):

```
from torch.utils.data import DataLoader, TensorDataset
```

```
X_train = [...]  # generate a 3D dataset, like in Chapter 7
train_set = TensorDataset(X_train, X_train)  # the inputs are also the targets
train_loader = DataLoader(train_set, batch_size=32, shuffle=True)
```

Note that `X_train` is used as both the inputs and the targets. Next, let's train the autoencoder, using the same `train()` function as in Chapter 10 (the notebook uses a slightly fancier function which prints some info and evaluates the model at each epoch):

```
import torchmetrics

optimizer = torch.optim.NAdam(autoencoder.parameters(), lr=0.2)
mse = nn.MSELoss()
rmse = torchmetrics.MeanSquaredError(squared=False).to(device)
train(autoencoder, optimizer, mse, train_loader, n_epochs=20)
```

Now that the autoencoder is trained, we can use its encoder to compress 3D inputs to 2D. For example, let's compress all the training set:

```
codings = encoder(X_train.to(device))
```

Figure 18-2 shows the original 3D dataset (on the left) and the output of the autoencoder's hidden layer (i.e., the coding layer, on the right). As you can see, the autoencoder found the best 2D plane to project the data onto, preserving as much variance in the data as it could (just like PCA).

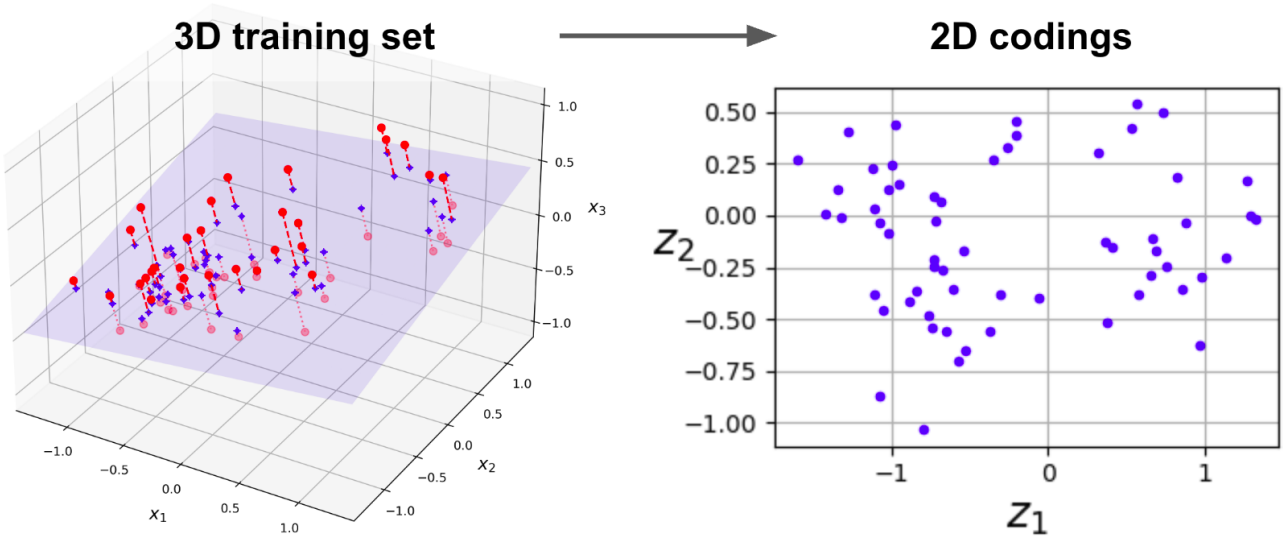



Figure 18-2. Approximate PCA performed by an undercomplete linear autoencoder

---

**NOTE**

You can think of an autoencoder as performing a form of self-supervised learning, since it is based on a supervised learning technique with automatically generated labels (in this case simply equal to the inputs).

---

# Stacked Autoencoders

Just like other neural networks we have discussed, autoencoders can have multiple hidden layers. In this case they are called *stacked autoencoders* (or *deep autoencoders*). Adding more layers helps the autoencoder learn more complex codings. That said, one must be careful not to make the autoencoder too powerful. Imagine an encoder so powerful that it just learns to map each input to a single arbitrary number (and the decoder learns the reverse mapping). Obviously such an autoencoder will reconstruct the training data perfectly, but it will not have learned any useful data representation in the process, and it is unlikely to generalize well to new instances.

The architecture of a stacked autoencoder is typically symmetrical with regard to the central hidden layer (the coding layer). To put it simply, it looks like a sandwich. For example, an autoencoder for Fashion MNIST (introduced in Chapter 9) may have 784 inputs, followed by a hidden layer with 128 neurons, then a central hidden layer of 32 neurons, then another hidden layer with 128 neurons, and an output layer with 784 neurons. This stacked autoencoder is represented in Figure 18-3. Note that all hidden layers must have an activation function, such as ReLU.

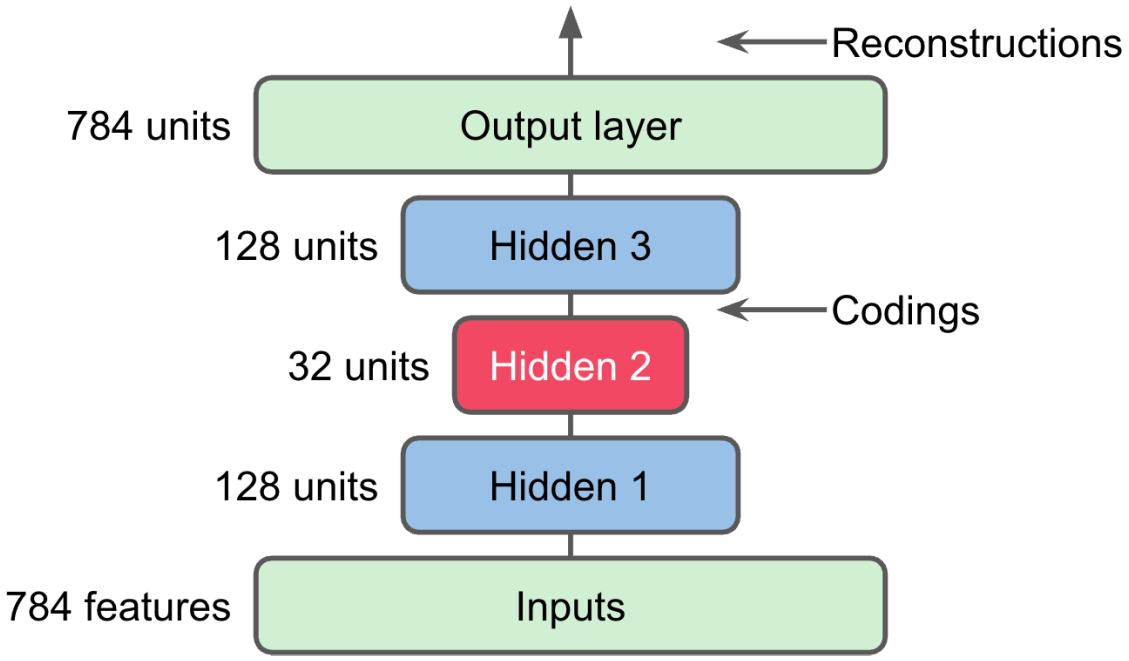



Figure 18-3. Stacked autoencoder

## Implementing a Stacked Autoencoder Using PyTorch

You can implement a stacked autoencoder very much like a regular deep MLP. For example, here is an autoencoder you can use to process Fashion MNIST images:

```
stacked_encoder = nn.Sequential(
    nn.Flatten(),
    nn.Linear(1 * 28 * 28, 128), nn.ReLU(),
    nn.Linear(128, 32), nn.ReLU(),
```

```
)
stacked_decoder = nn.Sequential(
    nn.Linear(32, 128), nn.ReLU(),
    nn.Linear(128, 1 * 28 * 28), nn.Sigmoid(),
    nn.Unflatten(dim=1, unflattened_size=(1, 28, 28))
)
stacked_ae = nn.Sequential(stacked_encoder, stacked_decoder).to(device)
```

Let's go through this code:

- Just like earlier, we split the autoencoder model into two submodels: the encoder and the decoder.
- The encoder takes 28 × 28–pixel grayscale images (i.e., with a single channel), flattens them so that each image is represented as a vector of size 784, then processes these vectors through two `Linear` layers of diminishing sizes (128 units then 32 units), each followed by the ReLU activation function. For each input image, the encoder outputs a vector of size 32.
- The decoder takes codings of size 32 (output by the encoder) and processes them through two `Linear` layers of increasing sizes (128 units then 784 units), and it reshapes the final vectors into 1 × 28 × 28 arrays so the decoder's outputs have the same shape as the encoder's inputs. Note that we use the sigmoid function for the output layer instead of ReLU, to ensure that the output pixel values range between 0 and 1.

We can now load the Fashion MNIST dataset and train the autoencoder exactly like the previous autoencoder, using the inputs as the targets and minimizing the MSE loss: give it a try!

## Visualizing the Reconstructions

One way to ensure that an autoencoder is properly trained is to compare the inputs and the outputs: the differences should not be too significant. Let's plot a few images from the validation set, as well as their reconstructions:

```
import matplotlib.pyplot as plt

def plot_image(image):
    plt.imshow(image.permute(1, 2, 0).cpu(), cmap="binary")
    plt.axis("off")

def plot_reconstructions(model, images=X_valid, n_images=5):
    images = images[:n_images]
    with torch.no_grad():
        y_pred = model(images.to(device))

    fig = plt.figure(figsize=(len(images) * 1.5, 3))
    for idx in range(len(images)):
        plt.subplot(2, len(images), 1 + idx)
        plot_image(images[idx])
        plt.subplot(2, len(images), 1 + len(images) + idx)
        plot_image(y_pred[idx])
```

```
plot_reconstructions(stacked_ae)
plt.show()
```

This code assumes that you have loaded the Fashion MNIST dataset, trained the autoencoder, and `X_valid` is a tensor containing the validation set. Figure 18-4 shows the resulting images.

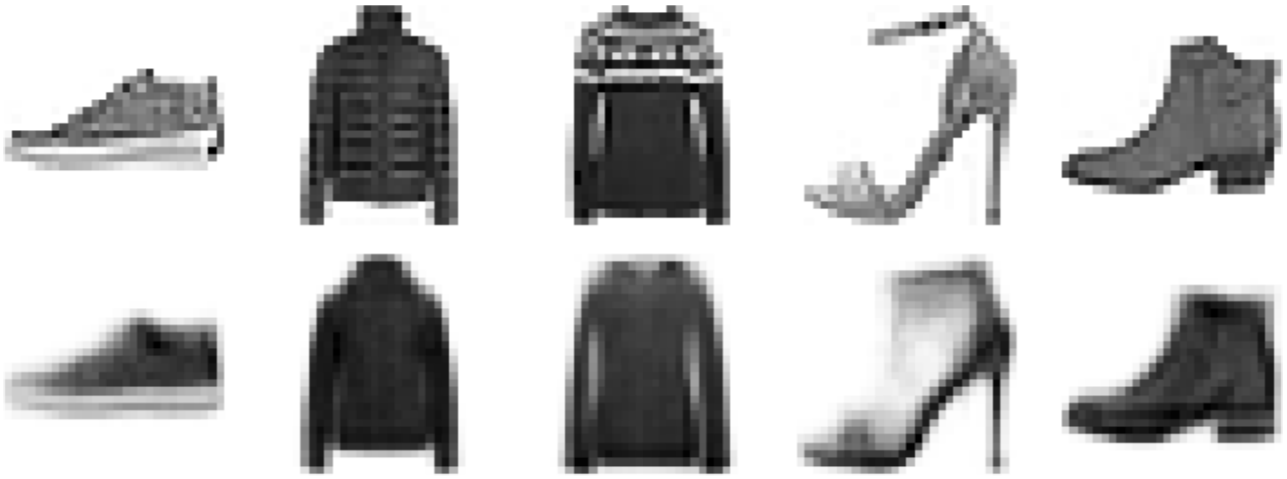


Figure 18-4. Original images (top) and their reconstructions (bottom)

The reconstructions are recognizable, but a bit too lossy. We may need to train the model for longer, or make the encoder and decoder more powerful, or make the codings larger. For now, let's go with this model and see how we can use it.

## Anomaly Detection Using AutoEncoders

One common use case for autoencoders is anomaly detection. Indeed, if an autoencoder is given an image that doesn't look like the images it was trained on (the image is said to be *out of distribution*), then the reconstruction will be terrible. For example, Figure 18-5 shows some MNIST digits and their reconstructions using the model we just trained on Fashion MNIST: as you can see, these reconstructions are very different from the inputs. If you compute the reconstruction loss (i.e., the MSE between the input and the output), it will be very high. To use the model for anomaly detection, you simply need to define a threshold, then any image whose reconstruction loss is greater than that threshold can be considered an anomaly.

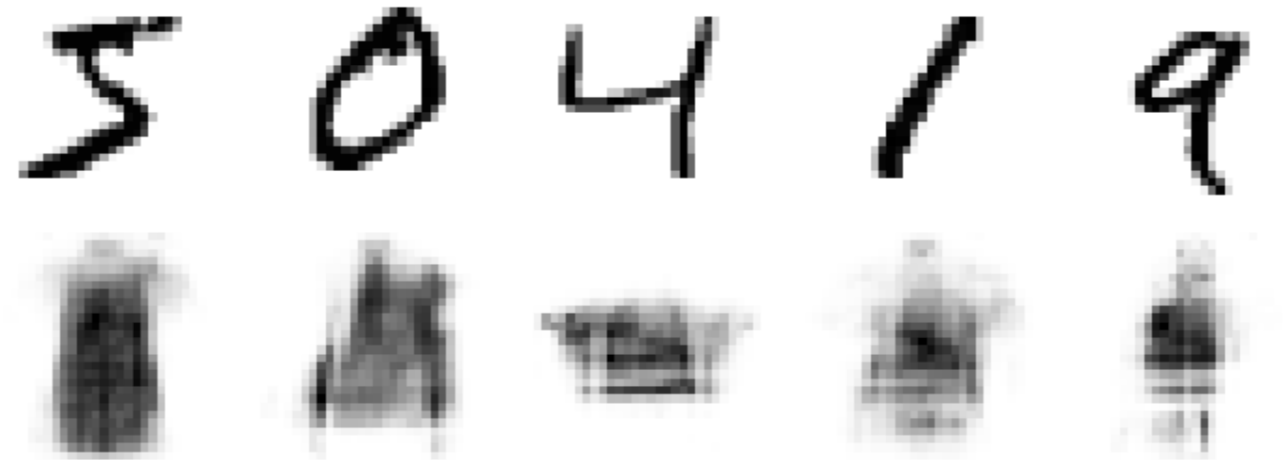


Figure 18-5. Out-of-distribution images are poorly reconstructed

That's all there is to it! Now let's look at another use case for autoencoders.

## Visualizing the Fashion MNIST Dataset

As we saw earlier in this chapter, undercomplete autoencoders can be used for dimensionality reduction. However, for most datasets they will not do a very good job at reducing the dimensionality down to 2 or 3 dimensions: they need enough dimensions to be able to properly reconstruct the inputs. As a result, they are generally not used directly for visualization. However, they are great at handling huge datasets, so one strategy is to use an autoencoder to reduce the dimensionality down to a reasonable level, then use another dimensionality reduction algorithm for visualization, such as those we discussed in Chapter 7.

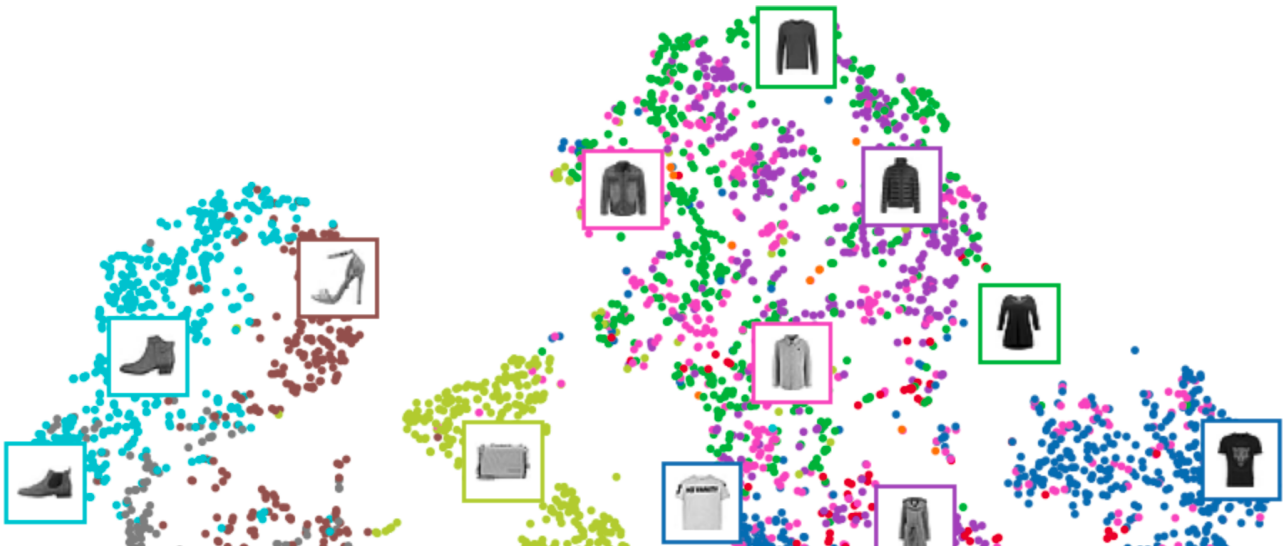
Let's use this strategy to visualize Fashion MNIST. First we'll use the encoder from our stacked autoencoder to reduce the dimensionality down to 32, then we'll use Scikit-Learn's implementation of the t-SNE algorithm to reduce the dimensionality down to 2 for visualization:

```
from sklearn.manifold import TSNE

with torch.no_grad():
    X_valid_compressed = stacked_encoder(X_valid.to(device))

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

Now we can plot the dataset:

```
plt.scatter(X_valid_2D[:, 0], X_valid_2D[:, 1], c=y_valid, s=10, cmap="tab10")
plt.show()
```

Figure 18-6 shows the resulting scatterplot, beautified a bit by displaying some of the images. The t-SNE algorithm identified several clusters that match the classes reasonably well (each class is represented by a different color).



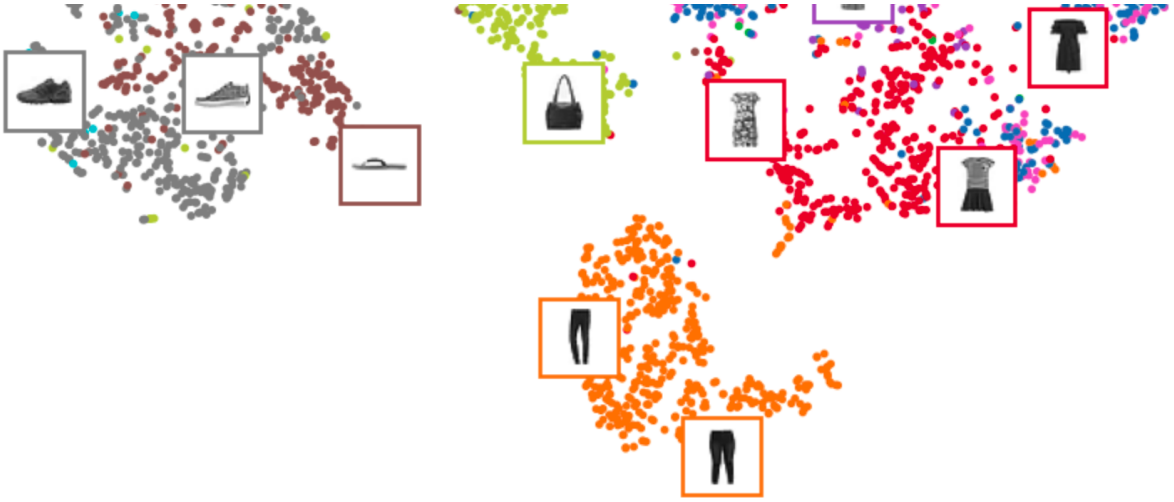


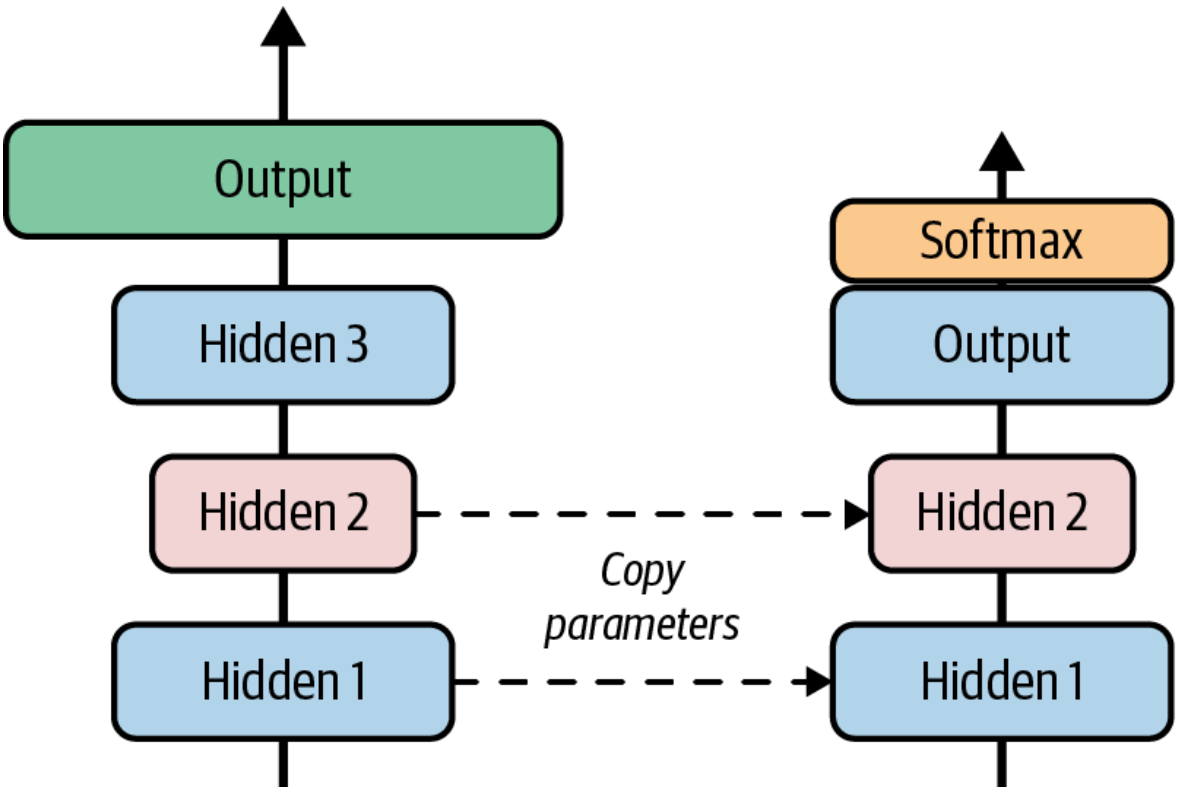
Figure 18-6. Fashion MNIST visualization using an autoencoder followed by t-SNE

Next let's look at how we can use autoencoders for unsupervised pretraining.

## Unsupervised Pretraining Using Stacked Autoencoders

As we discussed in Chapter 11, if you are tackling a complex supervised task but you do not have a lot of labeled training data, one solution is to find a neural network that performs a similar task and reuse its lower layers. This makes it possible to train a high-performance model using little training data because your neural network won't have to learn all the low-level features; it will just reuse the feature detectors learned by the existing network.

Similarly, if you have a large dataset but most of it is unlabeled, you can first train a stacked autoencoder using all the data, then reuse the lower layers to create a neural network for your actual task and train it using the labeled data. For example, Figure 18-7 shows how to use a stacked autoencoder to perform unsupervised pretraining for a classification neural network. When training the classifier, if you really don't have much labeled training data, you may want to freeze the pretrained layers (at least the lower ones).



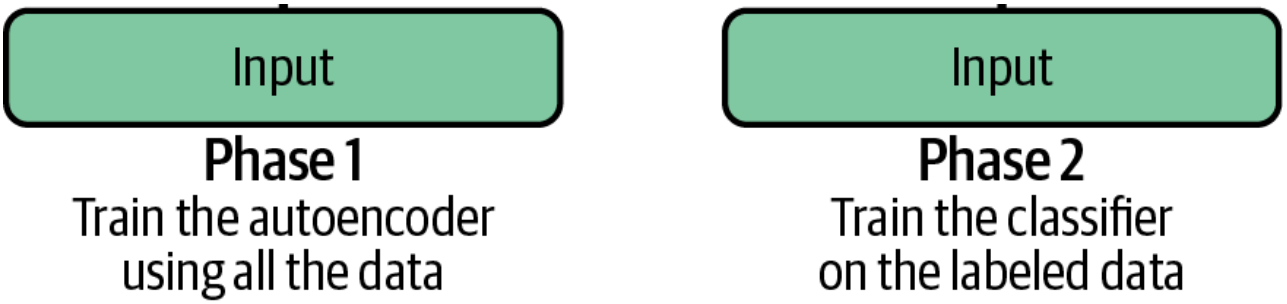



Figure 18-7. Unsupervised pretraining using autoencoders

---

**NOTE**

Having plenty of unlabeled data and little labeled data is common. Building a large unlabeled dataset is often cheap (e.g., a simple script can download millions of images off the internet), but labeling those images (e.g., classifying them as cute or not) can usually be done reliably only by humans. Labeling instances is time-consuming and costly, so it's normal to have only a few thousand human-labeled instances, or even less. That said, there is a growing trend toward using advanced AIs to label datasets.

---

There is nothing special about the implementation: just train an autoencoder using all the training data (labeled plus unlabeled), then reuse its encoder layers to create a new neural network and train it on the labeled instances (see the exercises at the end of this chapter for an example).

Let's now look at a few techniques for training stacked autoencoders.

## Tying Weights

When an autoencoder is neatly symmetrical, like the one we just built, a common technique is to *tie* the weights of the decoder layers to the weights of the encoder layers. This halves the number of weights in the model, speeding up training and limiting the risk of overfitting. Specifically, if the autoencoder has a total of $N$ layers (not counting the input layer), and $\mathbf{W}_L$ represents the connection weights of the $L^{\text{th}}$ layer (e.g., layer 1 is the first hidden layer, layer $N/2$ is the coding layer, and layer $N$ is the output layer), then the decoder layer weights can be defined as $\mathbf{W}_L = \mathbf{W}_{N-L+1}^{\intercal}$ (with $L = N / 2 + 1, \ldots, N$).

For example, below is the same autoencoder as the previous one, except the decoder weights are tied to the encoder weights:

```
import torch.nn.functional as F

class TiedAutoencoder(nn.Module):
    def __init__(self):
        super().__init__()
        self.enc1 = nn.Linear(1 * 28 * 28, 128)
        self.enc2 = nn.Linear(128, 32)
        self.dec1_bias = nn.Parameter(torch.zeros(128))
        self.dec2_bias = nn.Parameter(torch.zeros(1 * 28 * 28))
```

```
    def encode(self, X):
        Z = X.view(-1, 1 * 28 * 28)  # flatten
        Z = F.relu(self.enc1(Z))
        return F.relu(self.enc2(Z))

    def decode(self, X):
        Z = F.relu(F.linear(X, self.enc2.weight.t(), self.dec1_bias))
        Z = F.sigmoid(F.linear(Z, self.enc1.weight.t(), self.dec2_bias))
        return Z.view(-1, 1, 28, 28)  # unflatten

    def forward(self, X):
        return self.decode(self.encode(X))
```

This model achieves a smaller reconstruction error than the previous model, using about half the number of parameters.

## Training One Autoencoder at a Time

Rather than training the whole stacked autoencoder in one go like we just did, it is possible to train one shallow autoencoder at a time, then stack all of them into a single stacked autoencoder (hence the name), as shown in Figure 18-8. This technique is called "greedy layerwise training".

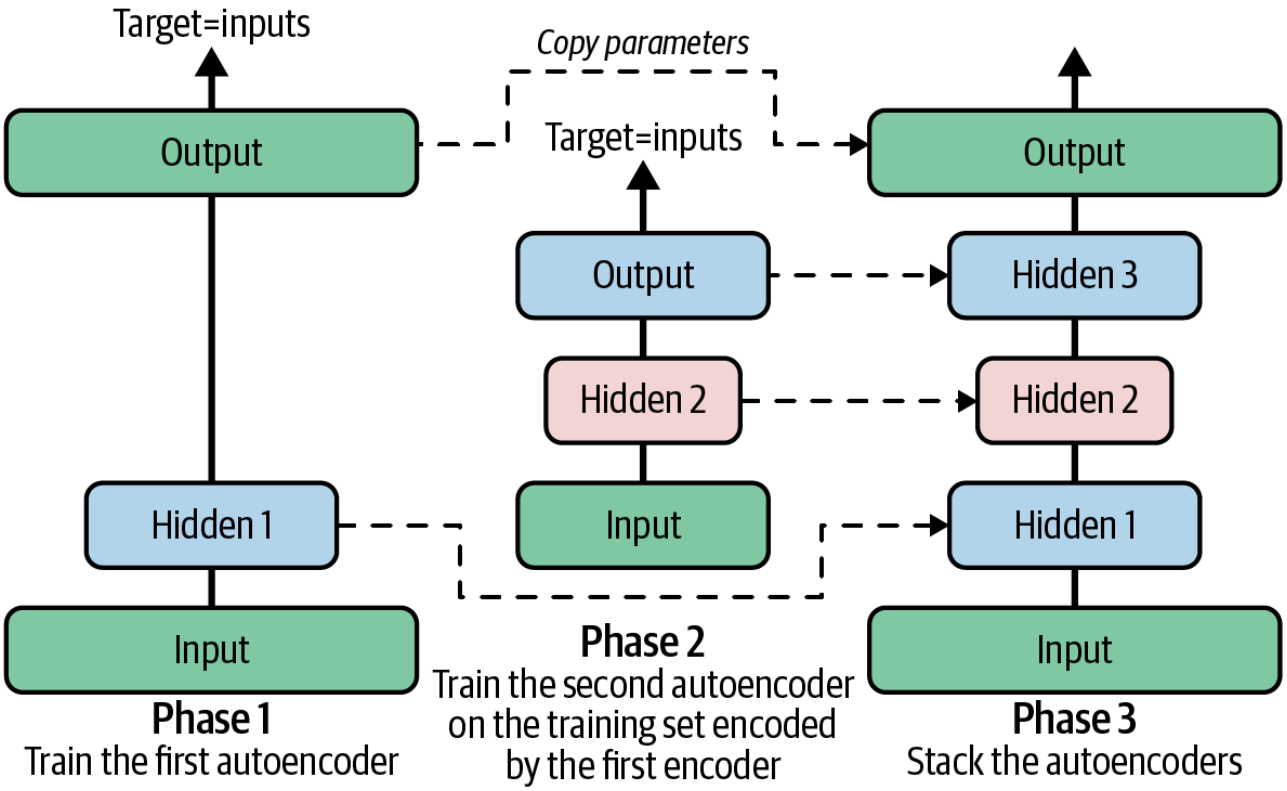



Figure 18-8. Training one autoencoder at a time

During the first phase of training, the first autoencoder learns to reconstruct the inputs. Then we encode the whole training set using this first autoencoder, and this gives us a new (compressed) training set. We then train a second autoencoder on this new dataset. This is the second phase of training. Finally, we build a big sandwich using all these autoencoders, as shown in Figure 18-8 (i.e., we first stack the encoder layers of each autoencoder, then the decoder lay-

ers in reverse order). This gives us the final stacked autoencoder. We could easily train more autoencoders this way, building a very deep stacked autoencoder.

As I mentioned in Chapter 11, one of the triggers of the deep learning tsunami was the discovery in 2006 by Geoffrey Hinton et al. that deep neural networks can be pretrained in an unsupervised fashion, using this greedy layerwise approach. They used restricted Boltzmann machines (RBMs; see *https://homl.info/extra-anns*) for this purpose, but in 2007 Yoshua Bengio et al.[2] showed that autoencoders worked just as well. For several years this was the only efficient way to train deep nets, until many of the techniques introduced in Chapter 11 made it possible to just train a deep net in one shot.

Autoencoders are not limited to dense networks: you can also build convolutional autoencoders. Let's look at these now.

## Convolutional Autoencoders

If you are dealing with images, then the autoencoders we have seen so far will not work well (unless the images are very small): as you saw in Chapter 12, convolutional neural networks are far better suited than dense networks to working with images. So if you want to build an autoencoder for images (e.g., for unsupervised pretraining or dimensionality reduction), you will need to build a *convolutional autoencoder*.[3] The encoder is a regular CNN composed of convolutional layers and pooling layers. It typically reduces the spatial dimensionality of the inputs (i.e., height and width) while increasing the depth (i.e., the number of feature maps). The decoder must do the reverse (upscale the image and reduce its depth back to the original dimensions), and for this you can use transpose convolutional layers (alternatively, you could combine upsampling layers with convolutional layers). Here is a basic convolutional autoencoder for Fashion MNIST:

```
conv_encoder = nn.Sequential(
    nn.Conv2d(1, 16, kernel_size=3, padding="same"), nn.ReLU(),
    nn.MaxPool2d(kernel_size=2),  # output: 16 × 14 × 14
    nn.Conv2d(16, 32, kernel_size=3, padding="same"), nn.ReLU(),
    nn.MaxPool2d(kernel_size=2),  # output: 32 × 7 × 7
    nn.Conv2d(32, 64, kernel_size=3, padding="same"), nn.ReLU(),
    nn.MaxPool2d(kernel_size=2),  # output: 64 × 3 × 3
    nn.Conv2d(64, 30, kernel_size=3, padding="same"), nn.ReLU(),
    nn.AdaptiveAvgPool2d((1, 1)), nn.Flatten())  # output: 30

conv_decoder = nn.Sequential(
    nn.Linear(30, 16 * 3 * 3),
    nn.Unflatten(dim=1, unflattened_size=(16, 3, 3)),
    nn.ConvTranspose2d(16, 32, kernel_size=3, stride=2), nn.ReLU(),
    nn.ConvTranspose2d(32, 16, kernel_size=3, stride=2, padding=1,
                       output_padding=1), nn.ReLU(),
    nn.ConvTranspose2d(16, 1, kernel_size=3, stride=2, padding=1,
                       output_padding=1), nn.Sigmoid())
```

```
conv_ae = nn.Sequential(conv_encoder, conv_decoder).to(device)
```

It's also possible to create autoencoders with other architecture types, such as RNNs (see the notebook for an example).

OK, let's step back for a second. So far we have looked at various kinds of autoencoders (basic, stacked, and convolutional), and how to train them (either in one shot or layer by layer). We also looked at a few applications: dimensionality reduction (e.g., for data visualization), anomaly detection, and unsupervised pretraining.

Up to now, in order to force the autoencoder to learn interesting features, we have limited the size of the coding layer, making it undercomplete. There are actually many other kinds of constraints that can be used, including ones that allow the coding layer to be just as large as the inputs, or even larger, resulting in an *overcomplete autoencoder*. So in the following sections we'll look at a few more kinds of autoencoders: denoising autoencoders, sparse autoencoders, and variational autoencoders.

## Denoising Autoencoders

A simple way to force the autoencoder to learn useful features is to add noise to its inputs, training it to recover the original, noise-free inputs. This idea has been around since the 1980s (e.g., it is mentioned in Yann LeCun's 1987 master's thesis). In a 2008 paper,[4] Pascal Vincent et al. showed that autoencoders could also be used for feature extraction. In a 2010 paper,[5] Vincent et al. introduced *stacked denoising autoencoders*.

The noise can be pure Gaussian noise added to the inputs, or it can be randomly switched-off inputs, just like in dropout (introduced in Chapter 11). Figure 18-9 shows both options.

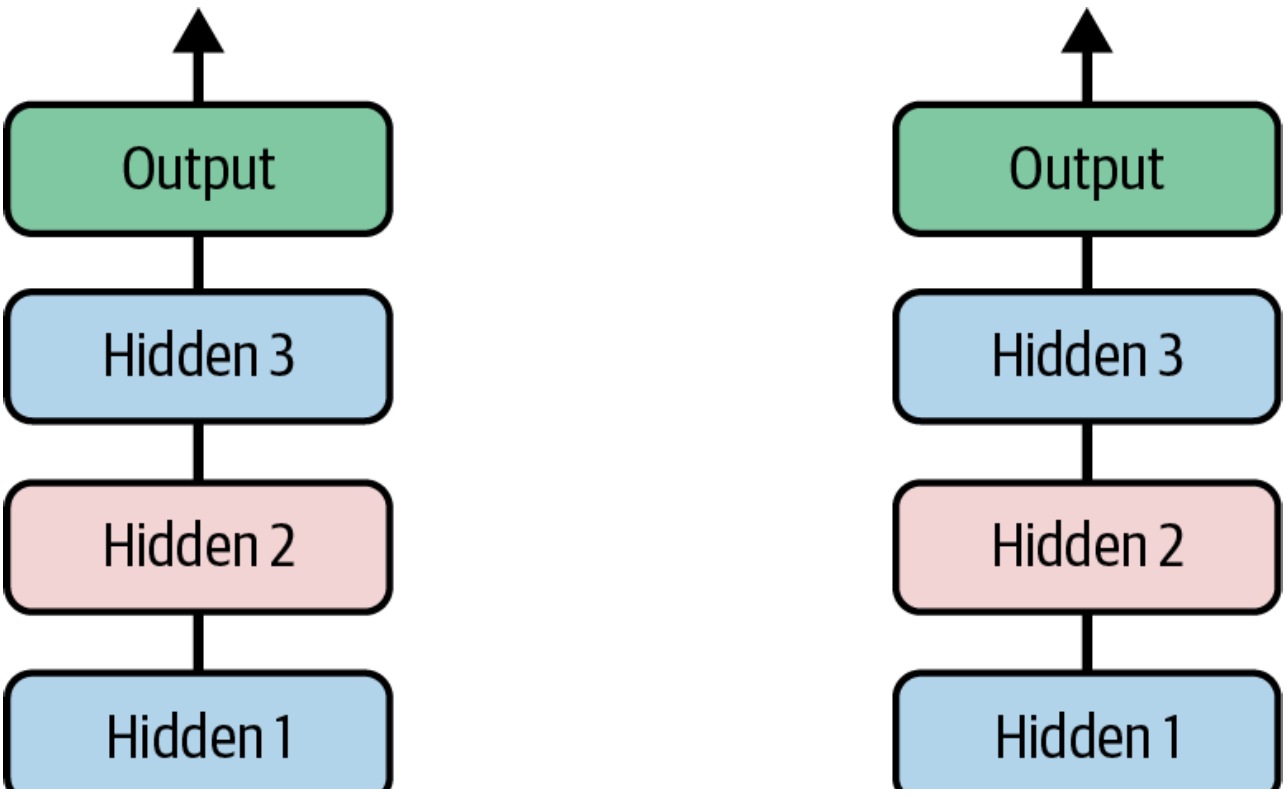



Figure 18-9. Denoising autoencoders, with Gaussian noise (left) or dropout (right)

The dropout implementation is straightforward: it is a regular stacked autoencoder with an additional `Dropout` layer applied to the encoder's inputs (recall that the `Dropout` layer is only active during training). Note that the coding layer does not need to compress the data as much since the noise already makes the reconstruction task non-trivial.

```
dropout_encoder = nn.Sequential(
    nn.Flatten(),
    nn.Dropout(0.5),
    nn.Linear(1 * 28 * 28, 128), nn.ReLU(),
    nn.Linear(128, 128), nn.ReLU(),
)
dropout_decoder = nn.Sequential(
    nn.Linear(128, 128), nn.ReLU(),
    nn.Linear(128, 1 * 28 * 28), nn.Sigmoid(),
    nn.Unflatten(dim=1, unflattened_size=(1, 28, 28))
)
dropout_ae = nn.Sequential(dropout_encoder, dropout_decoder).to(device)
```

---

**NOTE**

This may remind you of BERT's MLM pretraining task (see Chapter 15): reconstructing masked inputs (except BERT isn't split into an encoder and a decoder).

---

Figure 18-10 shows a few noisy images (with half of the pixels turned off), and the images reconstructed by the dropout-based denoising autoencoder, after training. Notice how the autoencoder guesses details that are actually not in the input, such as the top of the right-most shoe. As you can see, not only can denoising autoencoders be used for data visualization or unsupervised pretraining, like the other autoencoders we've discussed so far, but they can also be used quite simply and efficiently to remove noise from images.

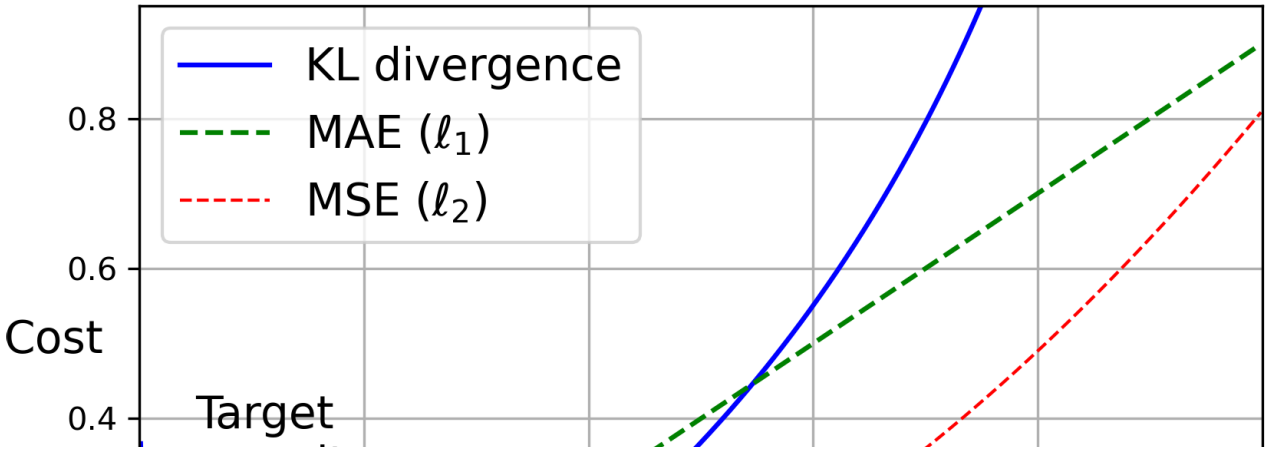
Figure 18-10. Noisy images (top) and their reconstructions (bottom)

## Sparse Autoencoders

Another kind of constraint that often leads to good feature extraction is *sparsity*: by adding an appropriate term to the cost function, the autoencoder is pushed to reduce the number of active neurons in the coding layer. This forces the autoencoder to represent each input as a combination of a small number of activations. As a result, each neuron in the coding layer typically ends up representing a useful feature (if you could speak only a few words per month, you would probably try to make them worth listening to).

A basic approach is to use the sigmoid activation function in the coding layer (to constrain the codings to values between 0 and 1), use a large coding layer (e.g., with 256 units), and add some $\ell_1$ regularization to the coding layer's activations. This means adding the $\ell_1$ norm of the codings (i.e., the sum of their absolute values) to the loss, weighted by a sparsity hyperparameter. This *sparsity loss* will encourage the neural network to produce codings close to 0. However, the total loss will still include the recontruction loss, so the model will be forced to output at least a few nonzero values to reconstruct the inputs correctly. Using the $\ell_1$ norm rather than the $\ell_2$ norm will push the neural network to preserve the most important codings while eliminating the ones that are not needed for the input image (rather than just reducing all codings).

Another approach—which often yields better results—is to measure the mean sparsity of each neuron in the coding layer, across each training batch, and penalize the model when the mean sparsity differs from the target sparsity (e.g., 10%). The batch size must not be too small, or else the mean will not be accurate. For example, if we measure that a neuron has an average activation of 0.3, but the target sparsity is 0.1, then this neuron must be penalized to activate less. One approach could be simply adding the squared error $(0.3 - 0.1)^2$ to the loss function, but in practice it's better to use the Kullback–Leibler (KL) divergence (briefly discussed in Chapter 4), since it has much stronger gradients than the mean squared error, as you can see in Figure 18-11.



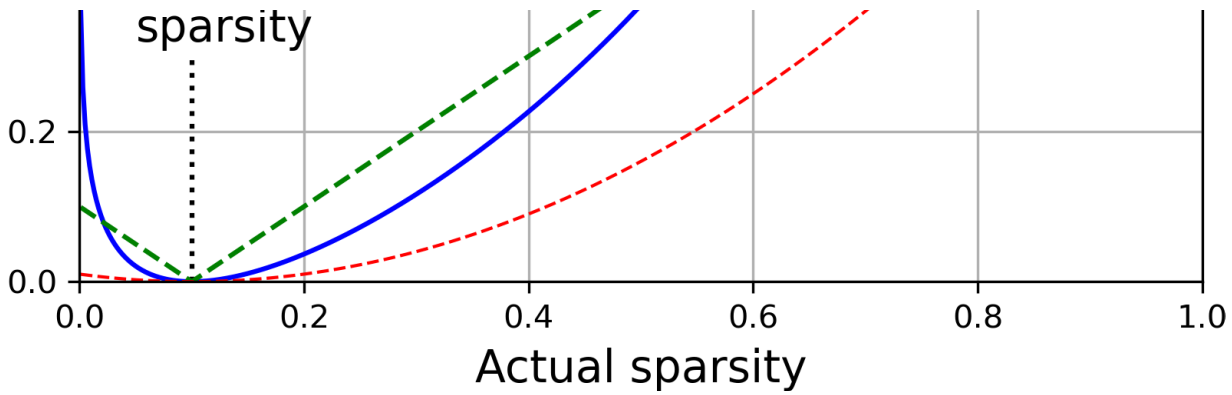



Figure 18-11. Sparsity loss

Given two discrete probability distributions $P$ and $Q$, the KL divergence between these distributions, noted $D_{\text{KL}}(P \parallel Q)$, can be computed using Equation 18-1.

**Equation 18-1. Kullback–Leibler divergence**

In our case, we want to measure the divergence between the target probability $p$ that a neuron in the coding layer will activate and the actual probability $q$, estimated by measuring the mean activation over the training batch. So, the KL divergence simplifies to Equation 18-2.

**Equation 18-2. KL divergence between the target sparsity $p$ and the actual sparsity $q$**

To implement this approach in PyTorch, we must first ensure that the autoencoder outputs both the reconstructions and the codings, since they are both needed to compute the loss. In the code below, the autoencoder's `forward()` method returns a `namedtuple` containing two fields: `output` (i.e., the reconstructions) and `codings`:

```
from collections import namedtuple

AEOutput = namedtuple("AEOutput", ["output", "codings"])

class SparseAutoencoder(nn.Module):
    def __init__(self):
        super().__init__()
        self.encoder = nn.Sequential(
            nn.Flatten(),
            nn.Linear(1 * 28 * 28, 128), nn.ReLU(),
            nn.Linear(128, 256), nn.Sigmoid())
        self.decoder = nn.Sequential(
            nn.Linear(256, 128), nn.ReLU(),
            nn.Linear(128, 1 * 28 * 28), nn.Sigmoid(),
            nn.Unflatten(dim=1, unflattened_size=(1, 28, 28)))

    def forward(self, X):
        codings = self.encoder(X)
```

```
        output = self.decoder(codings)
        return AEOutput(output, codings)
```

---

**NOTE**

You may need to tweak your training and evaluation functions to support these `namedtuple` predictions. For example, you can add `y_pred = y_pred.output` in the `evaluate_tm()` function, just after calling the model.

---

Next, we can define the loss function:

```
def mse_plus_sparsity_loss(y_pred, y_target, target_sparsity=0.1,
                           kl_weight=1e-3, eps=1e-8):
    p = torch.tensor(target_sparsity, device=y_pred.codings.device)
    q = torch.clamp(y_pred.codings.mean(dim=0), eps, 1 - eps)  # actual sparsity
    kl_div = p * torch.log(p / q) + (1 - p) * torch.log((1 - p) / (1 - q))
    return mse(y_pred.output, y_target) + kl_weight * kl_div.sum()
```

This function returns the reconstruction loss (MSE) plus a weighted sparsity loss. The sparsity loss is the KL divergence between the target sparsity and the mean sparsity across the batch. The `kl_weight` is a hyperparameter you can tune to control how much to encourage sparsity: if this hyperparameter is too high, the model will stick closely to the target sparsity, but it may not reconstruct the inputs properly, making the model useless. Conversely, if it is too low, the model will mostly ignore the sparsity objective and will not learn any interesting features. The `eps` argument is a smoothing term to avoid division by zero when computing the KL divergence.

Now we're ready to create the model and train it:

```
torch.manual_seed(42)
sparse_ae = SparseAutoencoder().to(device)
optimizer = torch.optim.NAdam(sparse_ae.parameters(), lr=0.002)
train(sparse_ae, optimizer, mse_plus_sparsity_loss, train_loader, n_epochs=10)
```

After training this sparse autoencoder on Fashion MNIST, the coding layer will have roughly 10% sparsity. Success!

---

**TIP**

Sparse autoencoders often produce fairly interpretable codings, where each component corresponds to an identifiable feature in the image. For example, you can plot all the images whose $n^{\text{th}}$ coding is larger than usual (e.g., above the $90^{\text{th}}$ percentile): you will often notice that all the images have something in common (e.g., they are all shoes).

---

Now let's move on to variational autoencoders!

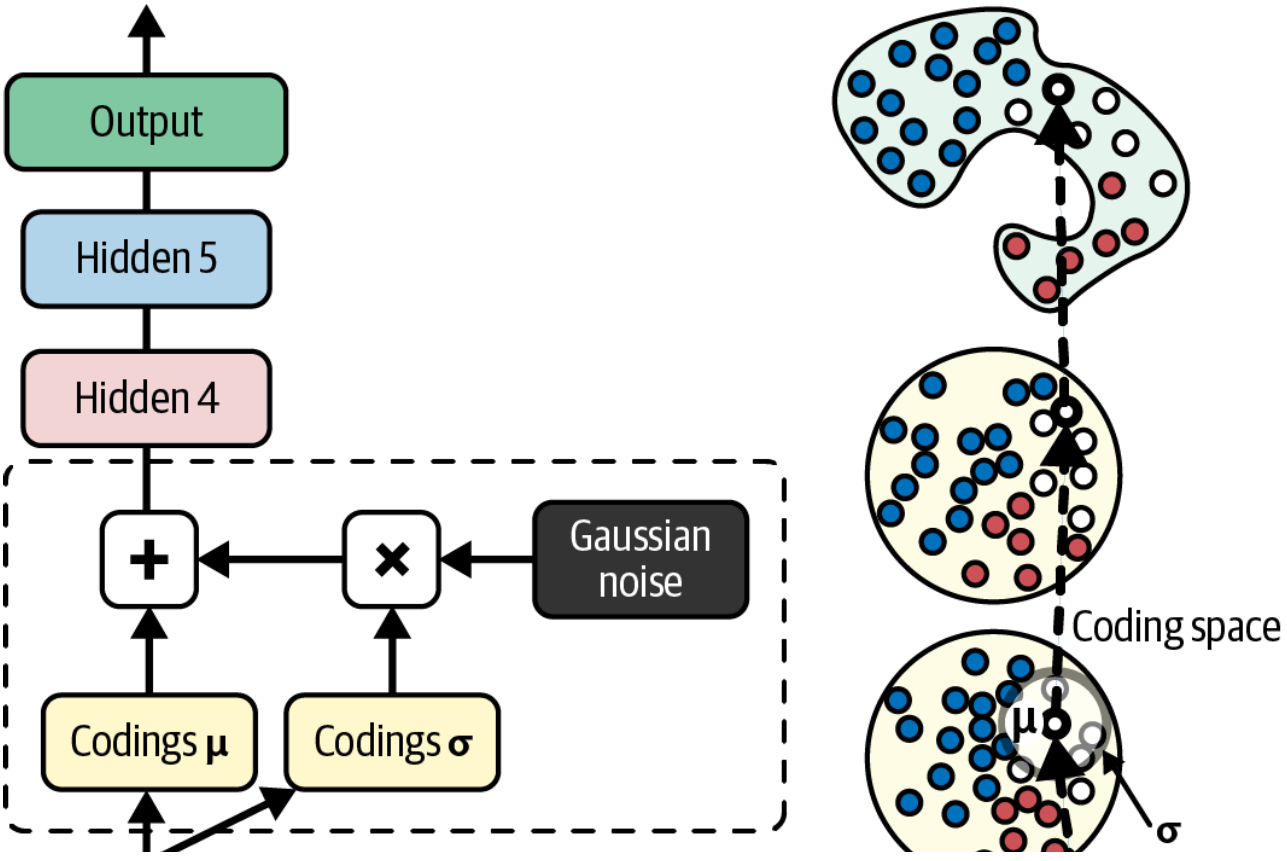
# Variational Autoencoders

An important category of autoencoders was introduced in 2013 by Diederik Kingma and Max Welling[6] and quickly became one of the most popular variants: *variational autoencoders* (VAEs).

VAEs are quite different from all the autoencoders we have discussed so far, in these particular ways:

- They are *probabilistic autoencoders*, meaning that their outputs are partly determined by chance, even after training (as opposed to denoising autoencoders, which use randomness only during training).
- Most importantly, they are *generative autoencoders*, meaning that they can generate new instances that look like they were sampled from the training set.[7]

Let's take a look at how VAEs work. Figure 18-12 (left) shows a variational autoencoder. You can recognize the basic structure of all autoencoders, with an encoder followed by a decoder (in this example, they both have two hidden layers), but there is a twist: instead of directly producing a coding for a given input, the encoder produces a *mean coding* **μ** and a standard deviation **σ**. The actual coding is then sampled randomly from a Gaussian distribution with mean **μ** and standard deviation **σ**. After that the decoder decodes the sampled coding normally. The right part of the diagram shows a training instance going through this autoencoder. First, the encoder produces **μ** and **σ**, then a coding is sampled randomly (notice that it is not exactly located at **μ**), and finally this coding is decoded; the final output resembles the training instance.



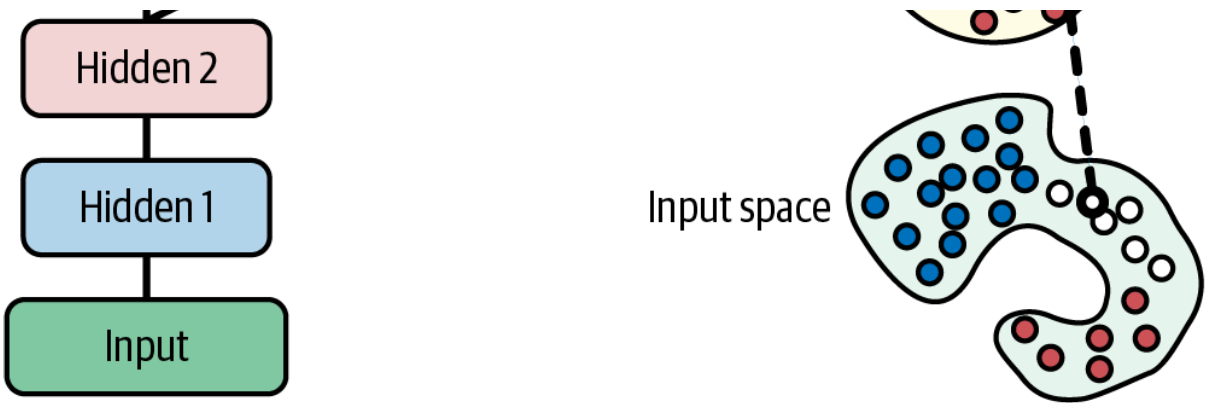



Figure 18-12. A variational autoencoder (left) and an instance going through it (right)

As you can see in the diagram, although the inputs may have a very convoluted distribution, a variational autoencoder tends to produce codings that look as though they were sampled from a simple Gaussian distribution during training, the cost function (discussed next) pushes the codings to gradually migrate within the coding space (also called the *latent space*) to end up looking like a cloud of Gaussian points. One great consequence is that after training a variational autoencoder, you can very easily generate a new instance: just sample a random coding from the Gaussian distribution, decode it, and voilà!

---

**NOTE**

Sampling a random vector from a Gaussian distribution is not differentiable, so when the VAE must sample codings from $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\sigma}^2)$, it actually starts by sampling $\boldsymbol{\epsilon}$ from $\mathcal{N}(0, 1)$, then computes $\boldsymbol{\mu} + \boldsymbol{\sigma}\boldsymbol{\epsilon}$. This *reparametrization trick* separates the deterministic and stochastic parts of the process, allowing the gradients to flow through the network.

---

Now, let's look at the cost function. It is composed of two parts. The first is the usual reconstruction loss that pushes the autoencoder to reproduce its inputs. We can use the MSE for this, as we did earlier. The second is the *latent loss* that pushes the autoencoder to have codings that look as though they were sampled from a simple Gaussian distribution: it is the KL divergence between the actual distribution of the codings and the desired latent distribution (i.e., the Gaussian distribution). The math is a bit more complex than with the sparse autoencoder, in particular because of the Gaussian noise, which limits the amount of information that can be transmitted to the coding layer. Luckily, the equations simplify, so the latent loss can be computed using Equation 18-3 (for the full mathematical details, check out the original paper on variational autoencoders, or Carl Doersch's great tutorial (2016).)

**Equation 18-3. Variational autoencoder's latent loss**

In this equation, $\mathcal{L}$ is the latent loss, $n$ is the codings' dimensionality, and $\mu_i$ and $\sigma_i$ are the mean and standard deviation of the $i^{\text{th}}$ component of the codings. The vectors $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ (which contain all the $\mu_i$ and $\sigma_i$) are output by the encoder, as shown in Figure 18-12 (left).

A common tweak to the variational autoencoder's architecture is to make the encoder output $\gamma$

A common tweak to the variational autoencoder's architecture is to make the encoder output $\gamma = \log(\sigma^2)$ rather than $\sigma$. The latent loss can then be computed as shown in Equation 18-4. This approach is more numerically stable and speeds up training.

**Equation 18-4. Variational autoencoder's latent loss, rewritten using $\gamma = \log(\sigma^2)$**

Let's build a variational autoencoder for Fashion MNIST, using the architecture shown in Figure 18-12, except using the $\gamma$ tweak:

```
VAEOutput = namedtuple("VAEOutput",
                       ["output", "codings_mean", "codings_logvar"])

class VAE(nn.Module):
    def __init__(self, codings_dim=32):
        super(VAE, self).__init__()
        self.codings_dim = codings_dim
        self.encoder = nn.Sequential(
            nn.Flatten(),
            nn.Linear(1 * 28 * 28, 128), nn.ReLU(),
            nn.Linear(128, 64), nn.ReLU(),
            nn.Linear(64, 2 * codings_dim))  # output both the mean and logvar
        self.decoder = nn.Sequential(
            nn.Linear(codings_dim, 64), nn.ReLU(),
            nn.Linear(64, 128), nn.ReLU(),
            nn.Linear(128, 1 * 28 * 28), nn.Sigmoid(),
            nn.Unflatten(dim=1, unflattened_size=(1, 28, 28)))

    def encode(self, X):
        return self.encoder(X).chunk(2, dim=-1)  # returns (mean, logvar)

    def sample_codings(self, codings_mean, codings_logvar):
        codings_std = torch.exp(0.5 * codings_logvar)
        noise = torch.randn_like(codings_std)
        return codings_mean + noise * codings_std

    def decode(self, Z):
        return self.decoder(Z)

    def forward(self, X):
        codings_mean, codings_logvar = self.encode(X)
        codings = self.sample_codings(codings_mean, codings_logvar)
        output = self.decode(codings)
        return VAEOutput(output, codings_mean, codings_logvar)
```

Let's go through this code:

- First, we define `VAEOutput`: this allows the model to output a `namedtuple` containing the reconstructions (`output`) as well as $\mu$ (`codings_mean`) and $\gamma$ (`codings_logvar`).

- The encoder and decoder architectures strongly resemble the previous autoencoders, but notice that the encoder's output is twice the size of the codings: this is because the encoder does not directly output the codings; instead, it outputs the parameters of the Gaussian distribution from which the codings will be sampled: the mean (**μ**) and the logarithm of the variance (**γ**).
- The `encode()` method calls the `encoder` model and splits the output in two using the `chunk()` method, to obtain **μ** and **γ**.
- The `sample_codings()` method takes **μ** and **γ** and samples the actual codings. For this, it first computes `torch.exp(0.5 * codings_logvar)` to get the codings' standard deviation **σ** (you can verify that this works mathematically). Then it uses the `torch.randn_like()` function to sample a random vector of the same shape as **σ** from the Gaussian distribution with mean 0 and standard deviation 1, on the same device and with the same data type. Lastly, it multiplies this Gaussian noise by **σ**, adds **μ**, and returns the result. This is the reparametrization trick we discussed earlier.
- The `decode()` method simply calls the decoder model to produce the reconstructions.
- The `forward()` method calls the encoder to get **μ** and **γ**, then it uses these parameters to sample the codings, which it decodes, and finally it returns a `VAEOutput` containing the reconstructions and the parameters **μ** and **γ**, which are all needed to compute the VAE loss.

Speaking of which, let's now define the loss function, which is the sum of the reconstruction loss (MSE) and the latent loss (KL divergence):

```
def vae_loss(y_pred, y_target, kl_weight=1.0):
    output, mean, logvar = y_pred
    kl_div = -0.5 * torch.sum(1 + logvar - logvar.exp() - mean.square(), dim=-1)
    return F.mse_loss(output, y_target) + kl_weight * kl_div.mean() / 784
```

The function first uses Equation 18-4 to compute the latent loss (`kl_div`) for each instance in the batch (by summing over the last dimension), then it computes the mean latent loss over all the instances in the batch (`kl_div.mean()`). Note that the reconstruction loss is the mean over all instances in the batch *and* all 784 pixels: this is why we divide the latent loss by 784 to ensure that the reconstruction loss and the latent loss have the same scale.

Finally, we can train the model on the Fashion MNIST dataset:

```
torch.manual_seed(42)
vae = VAE().to(device)
optimizer = torch.optim.NAdam(vae.parameters(), lr=1e-3)
train(vae, optimizer, vae_loss, train_loader, n_epochs=20)
```

# Generating Fashion MNIST Images

Now let's use this VAE to generate images that look like fashion items. All we need to do is sample random codings from a Gaussian distribution with mean 0 and variance 1, and decode

them:

```
torch.manual_seed(42)
vae.eval()
codings = torch.randn(3 * 7, vae.codings_dim, device=device)
with torch.no_grad():
    images = vae.decode(codings)
```

Figure 18-13 shows the 21 generated images.

Figure 18-13. Fashion MNIST images generated by the variational autoencoder

The majority of these images look fairly convincing, if a bit too fuzzy. The rest are not great, but don't be too harsh on the autoencoder—it only had a few minutes to learn, and you would get much better results by using convolutional layers!

Variational autoencoders make it possible to perform *semantic interpolation*: instead of interpolating between two images at the pixel level, which would look as if the two images were just overlaid, we can interpolate at the codings level. For example, if we sample two random codings and interpolate between them, then decode all of the interpolated codings, we get a sequence of images that gradually go from one fashion item to another (see Figure 18-14):

```
codings = torch.randn(2, vae.codings_dim)  # start and end codings
n_images = 7
weights = torch.linspace(0, 1, n_images).view(n_images, 1)
codings = torch.lerp(codings[0], codings[1], weights)  # linear interpolation
with torch.no_grad():
    images = vae.decode(codings.to(device))
```

Figure 18-14. Semantic interpolation

There are a few variants of VAEs, for example with different distributions for the latent variables. One important variant is discrete VAEs: let's discuss them now.

## Discrete Variational Autoencoders

A *discrete VAE* (dVAE) is much like a VAE, except the codings are discrete rather than continuous: each coding vector contains *latent codes* (also called *categories*), each of which is an integer between 0 and $k - 1$, where $k$ is the number of possible latent codes. The length of the coding vector is often noted $d$. For example, if you choose $k = 10$ and $d = 6$, then there are one million possible coding vectors ($10^6$), such as [3, 0, 3, 9, 1, 4]. Discrete VAEs are very useful for tokenizing continuous inputs for Transformers and other models. For example, they are at the core of models like BEiT and DALL·E (see Chapter 16).

The most natural way to make VAEs discrete is to use a categorical distribution instead of a Gaussian distribution. This implies a couple of changes:

- First, the encoder must output logits rather than means and variances: for each input image, it outputs a tensor of shape [$d$, $k$] containing logits.
- Second, since categorical sampling is not a differentiable operation, we must once again use a reparametrization trick, but we cannot reuse the same as for regular VAEs: we need one designed for categorical distributions. The most popular one is the Gumbel-softmax trick. Instead of directly sampling from the categorical distribution, we call the `F.gumble_softmax()` function: this implements a differentiable approximation of categorical sampling.

---

**NOTE**

The Gumbel distribution is used to model the maximum of a set of samples from another distribution: for example, it can be used to estimate the probability that a river will overflow within the next 10 years. If you add Gumbel noise to the logits, then take the argmax of the result, it is mathematically equivalent to categorical sampling. However, the argmax operation is not differentiable, so we replace it with the softmax during the backward pass: this gives us a differentiable approximation of categorical sampling.

---

This idea was proposed in 2016 almost simultaneously by two independent teams of researchers, one from DeepMind and Oxford University,[8] the other from Google, Cambridge University, and Stanford University.[9]

Let's implement a dVAE for Fashion MNIST:

```
DiscreteVAEOutput = namedtuple("DiscreteVAEOutput",
                               ["output", "logits", "codings_prob"])

class DiscreteVAE(nn.Module):
    def __init__(self, coding_length=32, n_codes=16, temperature=1.0):
        super().__init__()
```

```
        self.coding_length = coding_length
        self.n_codes = n_codes
        self.temperature = temperature
        self.encoder = nn.Sequential([...])  # outputs [coding_length, n_codes]
        self.decoder = nn.Sequential([...])  # outputs [1, 28, 28]

    def forward(self, X):
        logits = self.encoder(X)
        codings_prob = F.gumbel_softmax(logits, tau=self.temperature, hard=True)
        output = self.decoder(codings_prob)
        return DiscreteVAEOutput(output, logits, codings_prob)
```

As you can see, this code is very similar to the VAE code. Note that we set `hard=True` when calling the `F.gumbel_softmax()` function to ensure that the forward pass uses Gumbel-argmax (to obtain one-hot vectors of the sampled codes), while the backward pass uses the Gumbel-softmax approximation. Also note that we pass a temperature (a scalar) to this function: the logits will be divided by this temperature before calling the softmax function. The lower the temperature is, the closer the output distribution will be to one-hot vectors (this only affects the backward pass). In general, we use a temperature of 1 at the beginning of training, then gradually reduce it during training, down to a small value such as 0.1.

The loss function is also similar to the regular VAE loss: it's the sum of a reconstruction loss (MSE) and a weighted latent loss (KL divergence). However, the KL divergence equation is a bit different since the latent distribution has changed: it's now a uniform categorical distribution, where all possible codes are equally likely, so they each have a probability of 1 / $k$. Since $\log(1 / k) = -\log(k)$, we can add $\log(k)$ instead of subtracting $\log(1 / k)$ in the KL divergence equation.

```
def d_vae_loss(y_pred, y_target, kl_weight=1.0):
    output, logits, _ = y_pred
    codings_prob = F.softmax(logits, -1)
    k = logits.new_tensor(logits.size(-1))  # same device and dtype as logits
    kl_div = (codings_prob * (codings_prob.log() + k.log())).sum(dim=(1, 2))
    return F.mse_loss(output, y_target) + kl_weight * kl_div.mean() / 784
```

You can now train the model. Remember to update your training loop to reduce the temperature gradually during training, for example:

```
model.temperature = 1 - 0.9 * epoch / n_epochs
```

Once the model is trained, you can generate new images by sampling random codings from a uniform distribution, one-hot encoding them, then decoding the resulting one-hot distribution:

```
codings = torch.randint(0, d_vae.n_codes,  # from 0 to k - 1
                        (n_images, d_vae.coding_length), device=device)
codings_prob = F.one_hot(codings, num_classes=d_vae.n_codes).float()
with torch.no_grad():
```

```
images = d_vae.decoder(codings_prob)
```

Another popular approach to discrete VAEs is called *vector quantization* (VQ-VAE), proposed by DeepMind researchers in 2017.[10] Instead of producing logits, the encoder outputs *d* embeddings, each of dimensionality *e*. Then instead of sampling from a categorical distribution, the VQ-VAE maps each embedding to the index of the nearest embedding in a trainable embedding matrix of shape [*k*, *e*], called the *codebook*. This produces the integer codes. Finally, these codes are embedded using the embedding matrix and passed on to the decoder.

Since replacing an embedding with the nearest codebook embedding is not a differentiable operation, the backward pass pretends that the codebook lookup step is the identity function, so the gradients just go straight through this operation: this is why this trick is called the *straight-through estimator* (STE). It's an approximation that assumes that the gradients around the encoder embeddings are similar to the gradients around the nearest codebook embeddings.

---

**TIP**

VQ-VAEs can be a bit tricky to implement correctly, but you can use a library like *https://github.com/lucidrains/vector-quantize-pytorch*. On the positive side, training is more stable, and the codes are a bit easier to interpret.

---

Discrete VAEs work pretty well for small images, but not so much for large images: the small scale features may look good, but there will often be large scale inconsistencies. To improve on this, you can use the trained dVAE to encode your whole training set (so each instance becomes a sequence of integers), then use this new training set to train a transformer: just treat the codes as tokens, and train the transformer using next token prediction. Intuitively, the dVAE learns the vocabulary, while the transformer learns the grammar. Once the transformer is trained, you can generate a new image by first generating a sequence of codes using the transformer, then passing this sequence to the dVAE's decoder.

This two-stage approach also makes it easier to control the image generation process: when training the transformer, a textual description of the image can be fed to the transformer, for example as a prefix to the sequence of codes. We say that the transformer is *conditioned* on the description, which helps it predict the correct next code. This way, after training, we can guide the image generation process by providing a description of the image we desire: the transformer will use this description to generate the appropriate sequence of codes. This is exactly how the first DALL·E system worked.

In practice, the encoder and decoder are usually convolutional networks, so the latent representation is often organized as a grid (but it's still flattened to a sequence to train the transformer). For example, the encoder may output a tensor of shape [256, 32, 32]: that's a 32 × 32 grid containing 256-dimensional embeddings in each cell (or 256 logits in the case of Gumbel-Softmax dVAEs). After mapping these embeddings to the indices of the nearest embeddings in the codebook (or after categorical sampling), each image is represented as a 32 × 32 grid of integers (codes, with codes ranging between 0 and 255). To generate a new image, you use the

tegers (codes), with codes ranging between 0 and 255. To generate a new image, you use the transformer to predict a sequence of 1,024 codes, organize them into a 32 × 32 grid, replace each code with its codebook vector, then pass the result to the decoder to generate the final image.

---

TIP

To improve the image quality, you can also stack two or more dVAEs, each producing a smaller grid than the previous one: this is called a *hierarchical VAE* (HVAE). The encoders are stacked, followed by the decoders in reverse order, and all are trained jointly: the loss is the sum of a single reconstruction loss plus multiple KL divergence losses (one per dVAE).

---

Let's now turn our attention to GANs: they are harder to train, but when you manage to get them to work, they produce pretty amazing images.

## Generative Adversarial Networks

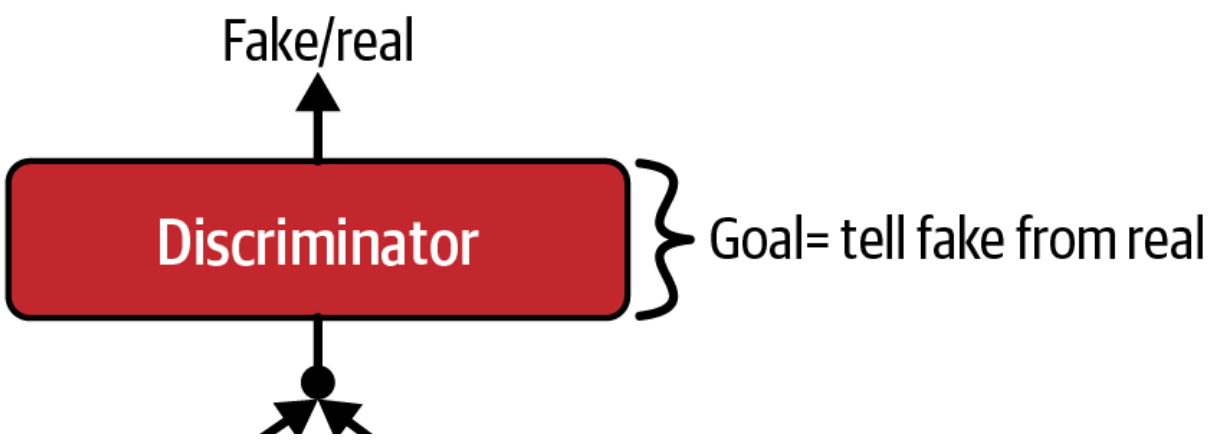
Generative adversarial networks were proposed in a 2014 paper[11] by Ian Goodfellow et al., and although the idea got researchers excited almost instantly, it took a few years to overcome some of the difficulties of training GANs. Like many great ideas, it seems simple in hindsight: make neural networks compete against each other in the hope that this competition will push them to excel. As shown in Figure 18-15, a GAN is composed of two neural networks:

*Generator*

Takes a random distribution as input (typically Gaussian) and outputs some data—typically, an image. You can think of the random inputs as the latent representations (i.e., codings) of the image to be generated. So, as you can see, the generator offers the same functionality as a decoder in a variational autoencoder, and it can be used in the same way to generate new images: just feed it some Gaussian noise, and it outputs a brand-new image. However, it is trained very differently, as you will soon see.

*Discriminator*

Takes either a fake image from the generator or a real image from the training set as input, and must guess whether the input image is fake or real.



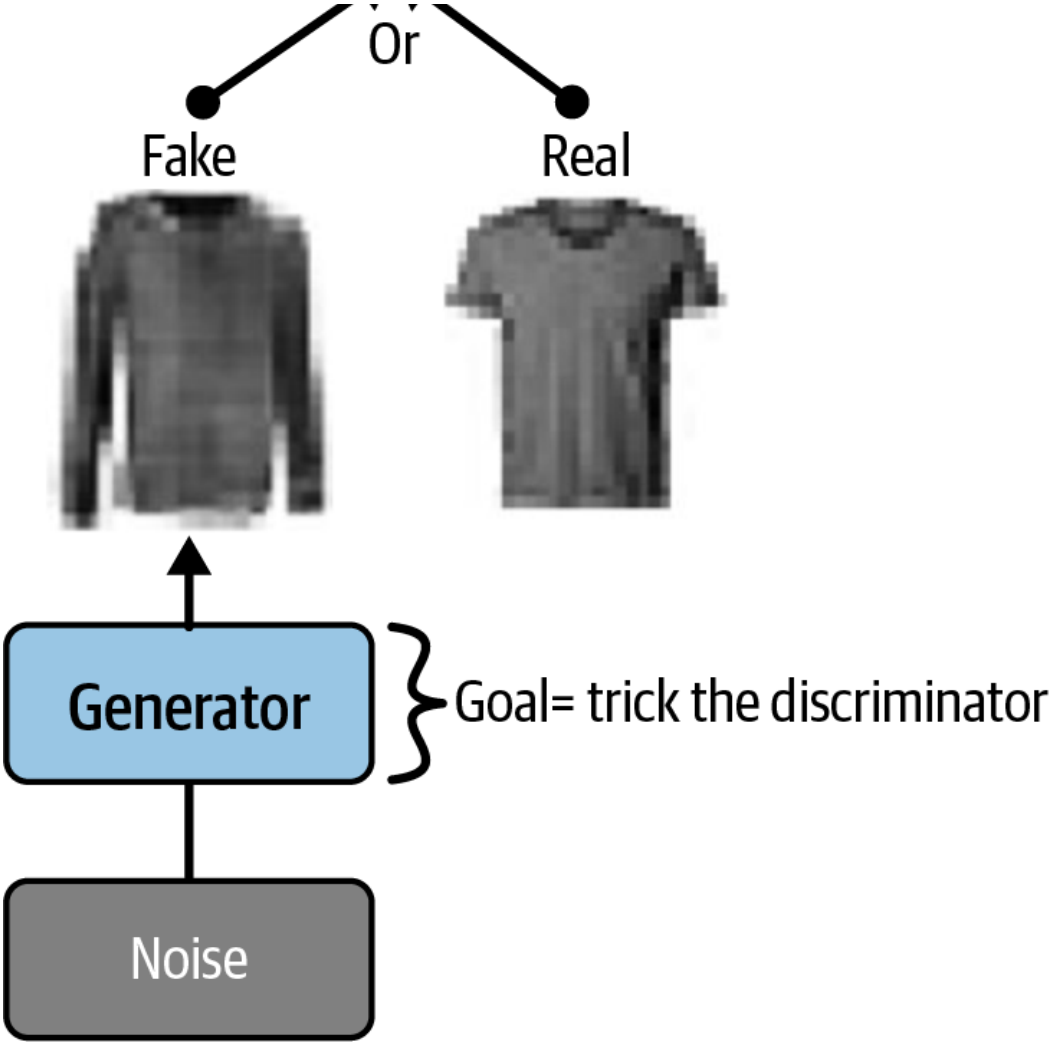



Figure 18-15. A generative adversarial network

During training, the generator and the discriminator have opposite goals: the discriminator tries to tell fake images from real images, while the generator tries to produce images that look real enough to trick the discriminator. Because the GAN is composed of two networks with different objectives, it cannot be trained like a regular neural network. Each training iteration is divided into two phases:

*First phase: train the discriminator*

A batch of real images is sampled from the training set and is completed with an equal number of fake images produced by the generator. The labels are set to 0 for fake images and 1 for real images, and the discriminator is trained on this labeled batch for one step, using the binary cross-entropy loss. Importantly, backpropagation only optimizes the weights of the discriminator during this phase.

*Second phase: train the generator*

We first use the generator to produce another batch of fake images, and once again the discriminator is used to tell whether the images are fake or real. This time we do not add real images in the batch, and all the labels are set to 1 (real): in other words, we want the generator to produce images that the discriminator will (wrongly) believe to be real! Crucially, the weights of the discriminator are frozen during this step, so backpropagation only affects the weights of the generator.

NOTE

The generator never actually sees any real images, yet it gradually learns to produce convincing fake images! All it gets is the gradients flowing back through the discriminator. Fortunately, the better the discriminator gets, the more information about the real images is contained in these secondhand gradients, so the generator can make significant progress.

---

Let's go ahead and build a simple GAN for Fashion MNIST.

First, we need to build the generator and the discriminator. The generator is similar to an autoencoder's decoder, and the discriminator is a regular binary classifier: it takes an image as input and ends with a `Dense` layer containing a single unit and using the sigmoid activation function. For the second phase of each training iteration, we also need the full GAN model containing the generator followed by the discriminator:

```
codings_dim = 30
generator = nn.Sequential(
    nn.Linear(codings_dim, 100), nn.ReLU(),
    nn.Linear(100, 150), nn.ReLU(),
    nn.Linear(150, 1 * 28 * 28), nn.Sigmoid(),
    nn.Unflatten(dim=1, unflattened_size=(1, 28, 28))).to(device)
discriminator = nn.Sequential(
    nn.Flatten(),
    nn.Linear(1 * 28 * 28, 150), nn.ReLU(),
    nn.Linear(150, 100), nn.ReLU(),
    nn.Linear(100, 1), nn.Sigmoid()).to(device)
```

Since the training loop is unusual, we need a new training function:

```
def train_gan(generator, discriminator, train_loader, codings_dim, n_epochs=20,
              g_lr=1e-3, d_lr=5e-4):
    criterion = nn.BCELoss()
    generator_opt = torch.optim.NAdam(generator.parameters(), lr=g_lr)
    discriminator_opt = torch.optim.NAdam(discriminator.parameters(), lr=d_lr)
    for epoch in range(n_epochs):
        for real_images, _ in train_loader:
            real_images = real_images.to(device)
            pred_real = discriminator(real_images)
            batch_size = real_images.size(0)
            ones = torch.ones(batch_size, 1, device=device)
            real_loss = criterion(pred_real, ones)
            noise = torch.randn(batch_size, codings_dim, device=device)
            fake_images = generator(noise).detach()
            pred_fake = discriminator(fake_images)
            zeros = torch.zeros(batch_size, 1, device=device)
            fake_loss = criterion(pred_fake, zeros)
            discriminator_loss = real_loss + fake_loss
            discriminator_opt.zero_grad()
            discriminator_loss.backward()
```

```
            discriminator_loss.backward()
            discriminator_opt.step()

            noise = torch.randn(batch_size, codings_dim, device=device)
            fake_images = generator(noise)
            for p in discriminator.parameters():
                p.requires_grad = False
            pred_fake = discriminator(fake_images)
            generator_loss = criterion(pred_fake, ones)
            generator_opt.zero_grad()
            generator_loss.backward()
            generator_opt.step()
            for p in discriminator.parameters():
                p.requires_grad = True
```

As discussed earlier, you can see the two phases at each iteration: first the discriminator makes a gradient descent step, then it's the generator's turn. We use a separate optimizer for each. Let's look in more detail:

*Phase one*

We feed a batch of real images to the discriminator and we compute the loss given targets equal to one: indeed, we want the discriminator to predict that these images are real. We then generate some noise and feed it to the generator to produce some fake images. Note that we call `detach()` on these images because we don't want gradient descent to affect the generator in this phase. Then we pass these fake images to the discriminator and we compute the loss given targets equal to zero: we want the discriminator to predict that these images are fake. The total discriminator loss is the `real_loss` plus the `fake_loss`. Finally, we perform the gradient descent step, improving the discriminator.

*Phase two*

We generate some fake images using the generator, and we pass them to the discriminator, like we just did. However, this time we don't call `detach()` on the fake images since we want to train the generator. Moreover, we make the discriminator non-trainable by setting `p.required_grad = False` for each parameter `p`. We then compute the loss using targets equal to one: indeed, we want the generator to fool the discriminator, so we want the discriminator to wrongly predict that these are real images. And finally we perform a gradient descent step for the generator, and we make the discriminator trainable again.

That's it! After training, you can randomly sample some codings from a Gaussian distribution, and feed them to the generator to produce new images:

```
generator.eval()
codings = torch.randn(n_images, codings_dim, device=device)
with torch.no_grad():
    generated_images = generator(codings)
```

If you display the generated images (see Figure 18-16), you will see that at the end of the first epoch, they already start to look like (very noisy) Fashion MNIST images.

Figure 18-16. Images generated by the GAN after one epoch of training

Unfortunately, the images never really get much better than that, and you may even find epochs where the GAN seems to be forgetting what it learned. Why is that? Well, it turns out that training a GAN can be challenging. Let's see why.

## The Difficulties of Training GANs

During training, the generator and the discriminator constantly try to outsmart each other, in a zero-sum game. As training advances, the game may end up in a state that game theorists call a *Nash equilibrium*, named after the mathematician John Nash: this is when no player would be better off changing their own strategy, assuming the other players do not change theirs. For example, a Nash equilibrium is reached when everyone drives on the left side of the road: no driver would be better off being the only one to switch sides. Of course, there is a second possible Nash equilibrium: when everyone drives on the *right* side of the road. Different initial states and dynamics may lead to one equilibrium or the other. In this example, there is a single optimal strategy once an equilibrium is reached (i.e., driving on the same side as everyone else), but a Nash equilibrium can involve multiple competing strategies (e.g., a predator chases its prey, the prey tries to escape, and neither would be better off changing their strategy).

So how does this apply to GANs? Well, the authors of the GAN paper demonstrated that a GAN can only reach a single Nash equilibrium: that's when the generator produces perfectly realistic images, and the discriminator is forced to guess (50% real, 50% fake). This fact is very encouraging: it would seem that you just need to train the GAN for long enough, and it will eventually reach this equilibrium, giving you a perfect generator. Unfortunately, it's not that simple: nothing guarantees that the equilibrium will ever be reached.

The biggest difficulty is called *mode collapse*: this is when the generator's outputs gradually become less diverse. How can this happen? Suppose that the generator gets better at producing convincing shoes than any other class. It will fool the discriminator a bit more with shoes, and this will encourage it to produce even more images of shoes. Gradually, it will forget how to

produce anything else. Meanwhile, the only fake images that the discriminator will see will be shoes, so it will also forget how to discriminate fake images of other classes. Eventually, when the discriminator manages to discriminate the fake shoes from the real ones, the generator will be forced to move to another class. It may then become good at shirts, forgetting about shoes, and the discriminator will follow. The GAN may gradually cycle across a few classes, never really becoming very good at any of them.

Moreover, because the generator and the discriminator are constantly pushing against each other, their parameters may end up oscillating and becoming unstable. Training may begin properly, then suddenly diverge for no apparent reason, due to these instabilities. And since many factors affect these complex dynamics, GANs are very sensitive to the hyperparameters: you may have to spend a lot of effort fine-tuning them.

These problems have kept researchers very busy since 2014: many papers have been published on this topic, some proposing new cost functions[12] (though a 2018 paper[13] by Google researchers questions their efficiency) or techniques to stabilize training or to avoid the mode collapse issue. For example, a popular technique called *experience replay* consists of storing the images produced by the generator at each iteration in a replay buffer (gradually dropping older generated images) and training the discriminator using real images plus fake images drawn from this buffer (rather than just fake images produced by the current generator). This reduces the chances that the discriminator will overfit the latest generator's outputs. Another common technique is called *mini-batch discrimination*: it measures how similar images are across the batch and provides this statistic to the discriminator, so it can easily reject a whole batch of fake images that lack diversity. This encourages the generator to produce a greater variety of images, reducing the chance of mode collapse. Other papers simply propose specific architectures that happen to perform well.

In short, this was a very active field of research, and much progress was made until quite recently: from *deep Convolutional GANs* (DCGAN) based on convolutional layers (see the notebook for an example), to *progressively growing GANs* which could produce high resolution images, or *StyleGANs* which gave the user fine-grained control over the image generation process, it seemed like GANs had a bright future ahead of them. But when diffusion models started to produce amazing images as well, with a much more stable training process and more diverse images, GANs were quickly sidelined. So let's now turn our attention to diffusion models.

# Diffusion Models

The ideas behind diffusion models have been around for many years, but they were first formalized in their modern form in a 2015 paper[14] by Jascha Sohl-Dickstein et al. from Stanford University and UC Berkeley. The authors applied tools from thermodynamics to model a diffusion process, similar to a drop of milk diffusing in a cup of tea. The core idea is to train a model to learn the reverse process: start from the completely mixed state, and gradually "unmix" the milk from the tea. Using this idea, they obtained promising results in image generation, but since GANs produced more convincing images back then, and they did so much faster, diffusion models did not get as much attention

Then, in 2020, Jonathan Ho et al., also from UC Berkeley, managed to build a diffusion model capable of generating highly realistic images, which they called a *denoising diffusion probabilistic model* (DDPM).[15] A few months later, a 2021 paper[16] by OpenAI researchers Alex Nichol and Prafulla Dhariwal analyzed the DDPM architecture and proposed several improvements that allowed DDPMs to finally beat GANs: not only are DDPMs much easier to train than GANs, but the generated images are more diverse and of even higher quality. The main downside of DDPMs, as you will see, is that they take a very long time to generate images, compared to GANs or VAEs.

So how exactly does a DDPM work? Well, suppose you start with a picture of a cat (like the one in Figure 18-17), noted $\mathbf{x}_0$, and at each time step $t$ you add a little bit of Gaussian noise to the image, with mean 0 and variance $\beta_t$ (a scalar). This noise is independent for each pixel: we call it *isotropic*. You first obtain the image $\mathbf{x}_1$, then $\mathbf{x}_2$, and so on, until the cat is completely hidden by the noise, impossible to see. The last time step is noted $T$. In the original DDPM paper, the authors used $T$ = 1,000, and they scheduled the variance $\beta_t$ in such a way that the cat signal fades linearly between time steps 0 and $T$. In the improved DDPM paper, $T$ was bumped up to 4,000, and the variance schedule was tweaked to change more slowly at the beginning and at the end. In short, we're gradually drowning the cat in noise: this is called the *forward process*.

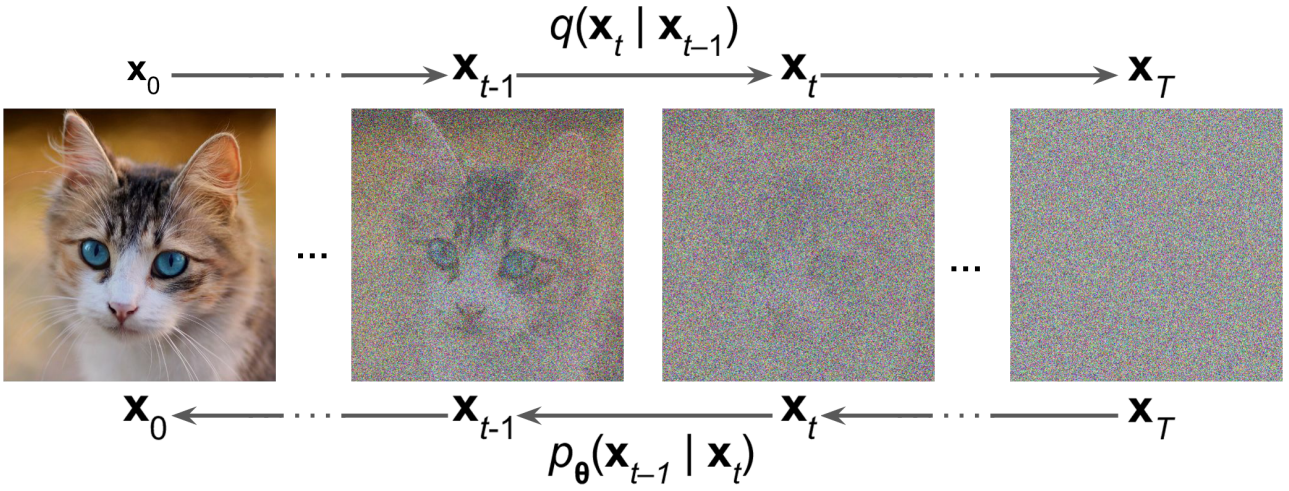



Figure 18-17. The forward process $q$ and reverse process $p$

As we add more and more Gaussian noise in the forward process, the distribution of pixel values becomes more and more Gaussian. One important detail I left out is that the pixel values get rescaled slightly at each step, by a factor of . This ensures that the mean of the pixel values gradually approaches 0, since the scaling factor is a bit smaller than 1 (imagine repeatedly multiplying a number by 0.99). It also ensures that the variance will gradually converge to 1. This is because the standard deviation of the pixel values also gets scaled by , so the variance gets scaled by $1 - \beta_t$ (i.e., the square of the scaling factor). But the variance cannot shrink to 0 since we're adding Gaussian noise with variance $\beta_t$ at each step. And since variances add up when you sum Gaussian distributions, the variance must converge to $1 - \beta_t + \beta_t = 1$.

The forward diffusion process is summarized in Equation 18-5. This equation won't teach you anything new about the forward process, but it's useful to understand this type of mathemati-

cal notation, as it's often used in ML papers. This equation defines the probability distribution $q$ of $\mathbf{x}_t$ given $\mathbf{x}_{t-1}$ as a Gaussian distribution with mean $\mathbf{x}_{t-1}$ times the scaling factor, and with a covariance matrix equal to $\beta_t\mathbf{I}$. This is the identity matrix $\mathbf{I}$ multiplied by $\beta_t$, which means that the noise is isotropic with variance $\beta_t$.

**Equation 18-5. Probability distribution $q$ of the forward diffusion process**

Interestingly, there's a shortcut for the forward process: it's possible to sample an image $\mathbf{x}_t$ given $\mathbf{x}_0$ without having to first compute $\mathbf{x}_1$, $\mathbf{x}_2$, ..., $\mathbf{x}_{t-1}$. Indeed, since the sum of multiple independent Gaussian distributions is also a Gaussian distribution, all the noise can be added in just one shot. If we define $\alpha_t = 1 - \beta_t$, and $\bar{\alpha}_t = \alpha_1 \times \alpha_2 \times \ldots \times \alpha_t =$ , then we can compute $\mathbf{x}_t$ using Equation 18-6. This is the equation we will be using, as it is much faster.

**Equation 18-6. Shortcut for the forward diffusion process**

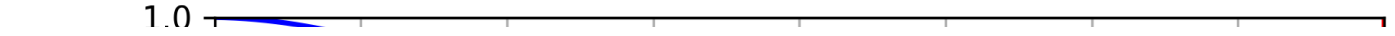
Our goal, of course, is not to drown cats in noise. On the contrary, we want to create many new cats! We can do so by training a model that can perform the *reverse process*: going from $\mathbf{x}_t$ to $\mathbf{x}_{t-1}$. We can then use it to remove a tiny bit of noise from an image, and repeat the operation many times until all the noise is gone. If we train the model on a dataset containing many cat images, then we can give it a picture entirely full of Gaussian noise, and the model will gradually make a brand new cat appear (see Figure 18-17).

OK, so let's start coding! The first thing we need to do is to code the forward process. For this, we will first need to implement the variance schedule. How can we control how fast the cat disappears? Initially, 100% of the variance comes from the original cat image. Then at each time step $t$, the variance gets multiplied by $\alpha_t = 1 - \beta_t$ (as explained earlier), and noise gets added. So, the part of the variance that comes from the initial distribution shrinks by a factor of $\alpha_t$ at each step. Therefore, after $t$ time steps, the cat signal will have been multiplied by $\bar{\alpha}_t = \alpha_1 \times \alpha_2 \times \ldots \times \alpha \sim t$. It's this "cat signal" factor $\bar{\alpha}_t$ that we want to schedule so it shrinks down from 1 to 0 gradually between time steps 0 and $T$. In the improved DDPM paper, the authors schedule $\bar{\alpha}_t$ according to Equation 18-7. This schedule is represented in Figure 18-18.

**Equation 18-7. Variance schedule equation for the forward diffusion process**

In this equation:

- $s$ is a tiny value which prevents $\beta_t$ from being too small near $t = 0$. In the paper, the authors used $s = 0.008$.
- $\beta_t$ is clipped to be no larger than 0.999, to avoid instabilities near $t = T$.



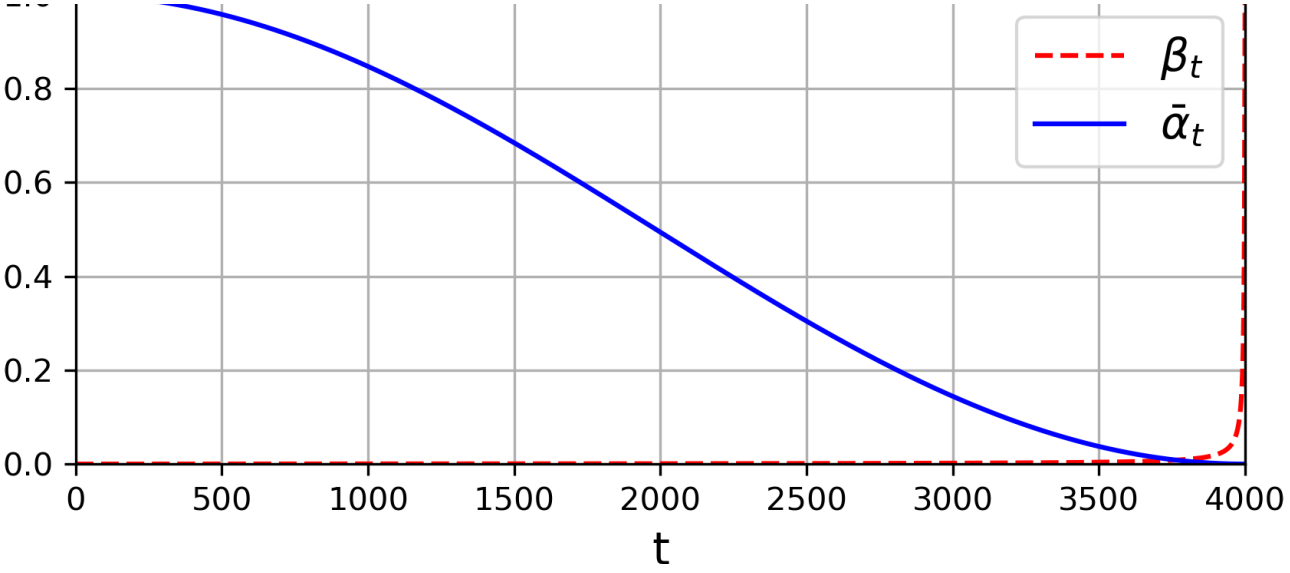



Figure 18-18. Noise variance schedule $\beta_t$, and the remaining signal variance $\bar{\alpha}_t$

Let's create a small function to compute $\alpha_t$, $\beta_t$, and $\bar{\alpha}_t$, using Equation 18-7, and call this function with $T$ = 4,000:

```
def variance_schedule(T, s=0.008, max_beta=0.999):
    t = torch.linspace(0, T, T + 1)
    f = torch.cos((t / T + s) / (1 + s) * torch.pi / 2) ** 2
    alpha_bars = f / f[0]
    betas = (1 - (f[1:] / f[:-1])).clamp(max=max_beta)
    betas = torch.cat([torch.zeros(1), betas])  # for easier indexing
    alphas = 1 - betas
    return alphas, betas, alpha_bars

T = 4000
alphas, betas, alpha_bars = variance_schedule(T)
```

To train our model to reverse the diffusion process, we will need noisy images from different time steps of the forward process. For this, let's create a function that will take an image $\mathbf{x}_0$ and a time step $t$ using Equation 18-6, and return a noisy image $\mathbf{x}_t$.

```
def forward_diffusion(x0, t):
    eps = torch.randn_like(x0)  # this unscaled noise will be the target
    xt = alpha_bars[t].sqrt() * x0 + (1 - alpha_bars[t]).sqrt() * eps
    return xt, eps
```

The model will need both the noisy image $\mathbf{x}_t$ and the time step $t$, so let's create a small class that will hold both. We'll give it a handy `to()` method to move both $\mathbf{x}_t$ and $t$ to the GPU:

```
class DiffusionSample(namedtuple("DiffusionSampleBase", ["xt", "t"])):
    def to(self, device):
        return DiffusionSample(self.xt.to(device), self.t.to(device))
```

Next, let's create a dataset wrapper class: it takes an image dataset—Fashion MNIST in our case

—and preprocesses the images so their pixel values range between –1 and +1 (this is optional but usually works better), and it uses the `forward_diffusion()` function to add noise to the image, then it wraps the resulting noisy image as well as the time step in a `DiffusionSample` object, and returns it along with the target, which is the unscaled noise `eps`, before it was scaled by and added to the image.

```
class DiffusionDataset:
    def __init__(self, dataset):
        self.dataset = dataset

    def __getitem__(self, i):
        x0, _ = self.dataset[i]
        x0 = (x0 * 2) - 1  # scale from -1 to +1
        t = torch.randint(1, T + 1, size=[1])
        xt, eps = forward_diffusion(x0, t)
        return DiffusionSample(xt, t), eps

    def __len__(self):
        return len(self.dataset)

train_set = DiffusionDataset(train_data)  # wrap Fashion MNIST
train_loader = DataLoader(train_set, batch_size=32, shuffle=True)
```

You may be wondering: why not predict the original image directly, rather than the unscaled noise? One reason is empirical: the authors tried both approaches, and they observed that predicting the noise rather than the image led to more stable training and better results. The other reason is that the noise is Gaussian, which allows for some mathematical simplifications: in particular, the KL divergence between two Gaussian distributions is proportional to the squared distance between their means, so we can use the MSE loss, which is simple, fast, and quite stable.

Now we're ready to build the actual diffusion model itself. It can be any model you want, as long as it takes a `DiffusionSample` as input and outputs images of the same shape as the input images. The DDPM authors used a modified U-Net architecture,[17] which has many similarities with the FCN architecture we discussed in Chapter 12 for semantic segmentation: it's a convolutional neural network that gradually downsamples the input images, then gradually upsamples them again, with skip connections crossing over from each level of the downsampling part to the corresponding level in the upsampling part. To take into account the time steps, they were encoded using a fixed sinusoidal encoding (i.e., the same technique as the positional encodings in the original Transformer architecture, see [Link to Come]). At every level in the U-Net architecture, they passed these time encodings through `Linear` layers and fed them to the U-Net. Lastly, they also used multi-head attention layers at various levels. See this chapter's notebook for a basic implementation (it's too long to copy here, and the details don't matter: many other model architectures would work just fine).

```
class DiffusionModel(nn.Module):  # see the notebook for full details
```

```
    def __init__(self, T=T, embed_dim=64):
        [...]  # create all the required modules to build the U-Net

    def forward(self, sample):
        [...]  # process the sample and predict the noise for each image
```

For training, the authors noted that using the MAE loss worked better than the MSE. You can also use the Huber loss:

```
diffusion_model = DiffusionModel().to(device)
huber = nn.HuberLoss()
optimizer = torch.optim.NAdam(diffusion_model.parameters(), lr=3e-3)
train(diffusion_model, optimizer, huber, train_loader, n_epochs=20)
```

Once the model is trained, you can use it to generate new images: you sample $\mathbf{x}_T$ randomly from a Gaussian distribution with mean 0 and variance 1, then you use Equation 18-8 to get $\mathbf{x}_{T-1}$. Then use this equation 3,999 more times until you get $\mathbf{x}_0$: if all went well, $\mathbf{x}_0$ should look like a regular Fashion MNIST image!

**Equation 18-8. Going one step in reverse in the DDPM diffusion process**

In this equation, $\boldsymbol{\epsilon}_{\boldsymbol{\theta}}(\mathbf{x}_t, t)$ represents the noise predicted by the model given the input image $\mathbf{x}_t$ and the time step $t$. The $\boldsymbol{\theta}$ represents the model parameters. Moreover, $\mathbf{z}$ is Gaussian noise with mean 0 and variance 1. This makes the reverse process stochastic: if you run it multiple times, you will get different images.

This works well, but it requires 4,000 iterations to generate an image! That's too slow. Luckily, just a few months after the DDPM paper, researchers from Stanford University proposed a technique named DDIM[18] to generate images in much fewer steps: instead of going from $t$ = 4,000 down to 0 one step at a time, DDIM can go down any number of time steps at a time, using Equation 18-9. Moreover, the training process is exactly the same as for DDPM, so we can simply reuse our trained DDPM model.

**Equation 18-9. Going multiple steps in reverse with DDIM**

In this equation:

- $\boldsymbol{\epsilon}_{\boldsymbol{\theta}}(\mathbf{x}_t, t)$, $\boldsymbol{\theta}$, and $\mathbf{z}$ have the same meanings as in Equation 18-8.
- $n$ represents any time step before $t$. For example, it could be $n = t - 50$

- $p$ represents any time step before $t$. For example, it could be $p = t - 50$.
- $\eta$ is a hyperparameter that controls how much randomness should be used during generation, from 0 (no randomness, fully deterministic) to 1 (just like DDPM).

Let's write a function that implements this reverse process, and call it to generate a few images:

```
def generate_ddim(model, batch_size=32, num_steps=50, eta=0.85):
    model.eval()
    with torch.no_grad():
        xt = torch.randn([batch_size, 1, 28, 28], device=device)
        times = torch.linspace(T - 1, 0, steps=num_steps + 1).long().tolist()
        for t, t_prev in zip(times[:-1], times[1:]):
            t_batch = torch.full((batch_size, 1), t, device=device)
            sample = DiffusionSample(xt, t_batch)
            eps_pred = model(sample)
            x0 = ((xt - (1 - alpha_bars[t]).sqrt() * eps_pred)
                  / (alpha_bars[t].sqrt()))
            abar_t_prev = alpha_bars[t_prev]
            variance = eta * (1 - abar_t_prev) / (1 - alpha_bars[t]) * betas[t]
            sigma_t = variance.sqrt()
            pred_dir = (1 - abar_t_prev - sigma_t**2).sqrt() * eps_pred
            noise = torch.randn_like(xt)
            xt = abar_t_prev.sqrt() * x0 + pred_dir + sigma_t * noise

        return torch.clamp((xt + 1) / 2, 0, 1)  # from [-1, 1] range to [0, 1]

X_gen_ddim = generate_ddim(diffusion_model, num_steps=500)
```

This time the generation will only take a few seconds, and it will produce images such as the ones shown in Figure 18-19. Granted, they're not very impressive, but we've only trained the model for a few minutes on Fashion MNIST: give it a try on a larger dataset and train it for a few hours to get more impressive results.

Figure 18-19. Images generated by DDIM accelerated diffusion

Diffusion models have made tremendous progress since 2020. In particular, a paper published in December 2021 by Robin Rombach, Andreas Blattmann, et al.,[19] introduced *latent diffusion models*, where the diffusion process takes place in latent space, rather than in pixel space. To achieve this, a powerful autoencoder is used to compress each training image into a much smaller latent space, where the diffusion process takes place, then the autoencoder is used to decompress the final latent representation, generating the output image. This considerably speeds up image generation, and reduces training time and cost dramatically. Importantly, the quality of the generated images is outstanding.

Moreover, the researchers also adapted various conditioning techniques to guide the diffusion process using text prompts, images, or any other inputs. This makes it possible to quickly produce any image you might fancy. You can also condition the image generation process using an input image. This enables many applications, such as outpainting—where an input image is extended beyond its borders—or inpainting—where holes in an image are filled in.

Lastly, a powerful pretrained latent diffusion model named *Stable Diffusion* (SD) was open sourced in August 2022 by a collaboration between LMU Munich and a few companies, including StabilityAI, and Runway, with support from EleutherAI and LAION. Now anyone can generate mindblowing images in seconds, for free, even on a regular laptop. For example, you can use Hugging Face's *Diffusers* library to load SD (e.g., the turbo variant), create an image generation pipeline for text-to-image, and generate an image of an orangutan reading a book:

```
from diffusers import AutoPipelineForText2Image

pipe = AutoPipelineForText2Image.from_pretrained(
    "stabilityai/sd-turbo", variant="fp16", torch_dtype=torch.float16)
pipe.to(device)
prompt = "A closeup photo of an orangutan reading a book"
torch.manual_seed(26)
image = pipe(prompt=prompt, num_inference_steps=1, guidance_scale=0.0).images[0]
```

Figure 18-20. A picture generated by Stable Diffusion using the Diffusers library

The possibilities are endless!

In the next chapter we will move on to an entirely different branch of deep learning: deep reinforcement learning.

# Exercises

1. What are the main tasks that autoencoders are used for?
2. Suppose you want to train a classifier, and you have plenty of unlabeled training data but only a few thousand labeled instances. How can autoencoders help? How would you proceed?
3. If an autoencoder perfectly reconstructs the inputs, is it necessarily a good autoencoder? How can you evaluate the performance of an autoencoder?
4. What are undercomplete and overcomplete autoencoders? What is the main risk of an excessively undercomplete autoencoder? What about the main risk of an overcomplete autoencoder?
5. How do you tie weights in a stacked autoencoder? What is the point of doing so?
6. What is a generative model? Can you name a type of generative autoencoder?
7. What is a GAN? Can you name a few tasks where GANs can shine?
8. What are the main difficulties when training GANs?
9. What are diffusion models good at? What is their main limitation?
10. Try using a denoising autoencoder to pretrain an image classifier. You can use MNIST (the simplest option), or a more complex image dataset such as CIFAR10 if you want a bigger challenge. Regardless of the dataset you're using, follow these steps:
    a. Split the dataset into a training set and a test set. Train a deep denoising autoencoder on the full training set.
    b. Check that the images are fairly well reconstructed. Visualize the images that most activate each neuron in the coding layer.
    c. Build a classification DNN, reusing the lower layers of the autoencoder. Train it using only 500 images from the training set. Does it perform better with or without pretraining?
11. Train a variational autoencoder on the image dataset of your choice, and use it to generate images. Alternatively, you can try to find an unlabeled dataset that you are interested in and see if you can generate new samples.
12. Train a DCGAN to tackle the image dataset of your choice, and use it to generate images. Add experience replay and see if this helps.
13. Train a diffusion model on your preferred image dataset (e.g., `tf_flowers` or `cifar10`), and generate nice images. Next, add the image

`torchvision.datasets.Flowers102`), and generate nice images. Next, add the image class as an extra input to the model, and retrain it: you should now be able to control the class of the generated image.

Solutions to these exercises are available at the end of this chapter's notebook, at *https://homl.info/colab-p*.

1 William G. Chase and Herbert A. Simon, "Perception in Chess", *Cognitive Psychology* 4, no. 1 (1973): 55–81.

2 Yoshua Bengio et al., "Greedy Layer-Wise Training of Deep Networks", *Proceedings of the 19th International Conference on Neural Information Processing Systems* (2006): 153–160.

3 Jonathan Masci et al., "Stacked Convolutional Auto-Encoders for Hierarchical Feature Extraction", *Proceedings of the 21st International Conference on Artificial Neural Networks* 1 (2011): 52–59.

4 Pascal Vincent et al., "Extracting and Composing Robust Features with Denoising Autoencoders", *Proceedings of the 25th International Conference on Machine Learning* (2008): 1096–1103.

5 Pascal Vincent et al., "Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion", *Journal of Machine Learning Research* 11 (2010): 3371–3408.

6 Diederik Kingma and Max Welling, "Auto-Encoding Variational Bayes", arXiv preprint arXiv:1312.6114 (2013).

7 Both these properties make VAEs rather similar to RBMs, but they are easier to train, and the sampling process is much faster (with RBMs you need to wait for the network to stabilize into a "thermal equilibrium" before you can sample a new instance).

8 Chris J. Maddison et al., "The Concrete Distribution: A Continuous Relaxation of Discrete Random Variables", arXiv preprint arXiv:1611.00712 (2016).

9 Eric Jang et al., "Categorical Reparameterization with Gumbel-Softmax", arXiv preprint arXiv:1611.01144 (2016).

10 Aaron van den Oord et al., "Neural Discrete Representation Learning", arXiv preprint arXiv:1711.00937 (2017).

11 Ian Goodfellow et al., "Generative Adversarial Nets", *Proceedings of the 27th International Conference on Neural Information Processing Systems* 2 (2014): 2672–2680.

12 For a nice comparison of the main GAN losses, check out this great GitHub project by Hwalsuk Lee.

13 Mario Lucic et al., "Are GANs Created Equal? A Large-Scale Study", Proceedings of the 32nd International Conference on Neural Information Processing Systems (2018): 698–707.

14 Jascha Sohl-Dickstein et al., "Deep Unsupervised Learning using Nonequilibrium Thermodynamics", arXiv preprint arXiv:1503.03585 (2015).

15 Jonathan Ho et al., "Denoising Diffusion Probabilistic Models" (2020).

16 Alex Nichol and Prafulla Dhariwal, “Improved Denoising Diffusion Probabilistic Models” (2021).

17 Olaf Ronneberger et al., “U-Net: Convolutional Networks for Biomedical Image Segmentation”, arXiv preprint arXiv:1505.04597 (2015).

18 Jiaming Song et al., “Denoising Diffusion Implicit Models”, arXiv preprint arXiv:2010.02502 (2020).

19 Robin Rombach, Andreas Blattmann, et al., “High-Resolution Image Synthesis with Latent Diffusion Models”, arXiv preprint arXiv:2112.10752 (2021).