12 Generative deep learning

This chapter covers

- Text generation
- DeepDream
- Neural style transfer
- Variational autoencoders
- Generative adversarial networks

The potential of artificial intelligence to emulate human thought processes goes beyond passive tasks such as object recognition and mostly reactive tasks such as driving a car. It extends well into creative activities. When I first made the claim that in a not-so-distant future, most of the cultural content that we consume will be created with substantial help from AIs, I was met with utter disbelief, even from long-time machine learning practitioners. That was in 2014. Fast-forward a few years, and the disbelief had receded at an incredible speed. In the summer of 2015, we were entertained by Google's DeepDream algorithm turning an image into a psychedelic mess of dog eyes and pareidolic artifacts; in 2016, we started using smartphone applications to turn photos into paintings of various styles. In the summer of 2016, an experimental short movie, *Sunspring*, was directed using a script written by

a Long Short-Term Memory. Maybe you've recently listened to music that was tentatively generated by a neural network.

Granted, the artistic productions we've seen from AI so far have been fairly low quality. AI isn't anywhere close to rivaling human screenwriters, painters, and composers. But replacing humans was always beside the point: artificial intelligence isn't about replacing our own intelligence with something else, it's about bringing into our lives and work *more* intelligence—intelligence of a different kind. In many fields, but especially in creative ones, AI will be used by humans as a tool to augment their own capabilities: more *augmented intelligence* than *artificial intelligence*.

A large part of artistic creation consists of simple pattern recognition and technical skill. And that's precisely the part of the process that many find less attractive or even dispensable. That's where AI comes in. Our perceptual modalities, our language, and our artwork all have statistical structure. Learning this structure is what deep learning algorithms excel at. Machine learning models can learn the statistical *latent space* of images, music, and stories, and they can then *sample* from this space, creating new artworks with characteristics similar to those the model has seen in its training data. Naturally, such sampling is hardly an act of artistic creation in itself. It's a mere mathematical operation: the algorithm has no grounding in human life, human emotions, or our experience of the world; instead, it learns from an experience that has little in common with ours. It's only our interpretation, as human spectators, that will give meaning to what the model generates. But in the hands of a skilled artist, algorithmic generation can be steered to become meaningful—and beautiful. Latent space sampling can become a brush that empowers the artist, augments our creative affordances, and expands the space of what we can imagine. What's more, it can make artistic creation more accessible by eliminating the need for technical skill and practice—setting up a new medium of pure expression, factoring art apart from craft.

Iannis Xenakis, a visionary pioneer of electronic and algorithmic music, beautifully expressed this same idea in the 1960s, in the context of the application of automation technology to music composition:[1]

> *Freed from tedious calculations, the composer is able to devote himself to the general problems that the new musical form poses and to explore the nooks and crannies of this form while modifying the values of the input data. For example, he may test all instrumental combinations from soloists, to chamber orchestras, to large orchestras. With the aid of electronic computers the composer becomes a sort of pilot: he presses the buttons, introduces coordinates, and supervises the controls of a cosmic vessel sailing in the space of sound, across sonic constellations and galaxies that he could formerly glimpse only as a distant dream.*

In this chapter, we'll explore from various angles the potential of deep learning to augment artistic creation. We'll review sequence data generation (which can be

[1] Iannis Xenakis, "Musiques formelles: nouveaux principes formels de composition musicale," special issue of *La Revue musicale*, nos. 253–254 (1963).

used to generate text or music), DeepDream, and image generation using both variational autoencoders and generative adversarial networks. We'll get your computer to dream up content never seen before; and maybe we'll get you to dream, too, about the fantastic possibilities that lie at the intersection of technology and art. Let's get started.

12.1 Text generation

In this section, we'll explore how recurrent neural networks can be used to generate sequence data. We'll use text generation as an example, but the exact same techniques can be generalized to any kind of sequence data: you could apply it to sequences of musical notes in order to generate new music, to timeseries of brushstroke data (perhaps recorded while an artist paints on an iPad) to generate paintings stroke by stroke, and so on.

Sequence data generation is in no way limited to artistic content generation. It has been successfully applied to speech synthesis and to dialogue generation for chatbots. The Smart Reply feature that Google released in 2016, capable of automatically generating a selection of quick replies to emails or text messages, is powered by similar techniques.

12.1.1 A brief history of generative deep learning for sequence generation

In late 2014, few people had ever seen the initials LSTM, even in the machine learning community. Successful applications of sequence data generation with recurrent networks only began to appear in the mainstream in 2016. But these techniques have a fairly long history, starting with the development of the LSTM algorithm in 1997 (discussed in chapter 10). This new algorithm was used early on to generate text character by character.

In 2002, Douglas Eck, then at Schmidhuber's lab in Switzerland, applied LSTM to music generation for the first time, with promising results. Eck is now a researcher at Google Brain, and in 2016 he started a new research group there, called Magenta, focused on applying modern deep learning techniques to produce engaging music. Sometimes good ideas take 15 years to get started.

In the late 2000s and early 2010s, Alex Graves did important pioneering work on using recurrent networks for sequence data generation. In particular, his 2013 work on applying recurrent mixture density networks to generate human-like handwriting using timeseries of pen positions is seen by some as a turning point.[2] This specific application of neural networks at that specific moment in time captured for me the notion of *machines that dream* and was a significant inspiration around the time I started developing Keras. Graves left a similar commented-out remark hidden in a 2013 LaTeX file uploaded to the preprint server arXiv: "Generating sequential data is

[2] Alex Graves, "Generating Sequences With Recurrent Neural Networks," arXiv (2013), https://arxiv.org/abs/1308.0850.

the closest computers get to dreaming." Several years later, we take a lot of these developments for granted, but at the time it was difficult to watch Graves's demonstrations and not walk away awe-inspired by the possibilities. Between 2015 and 2017, recurrent neural networks were successfully used for text and dialogue generation, music generation, and speech synthesis.

Then around 2017–2018, the Transformer architecture started taking over recurrent neural networks, not just for supervised natural language processing tasks, but also for generative sequence models—in particular *language modeling* (word-level text generation). The best-known example of a generative Transformer would be GPT-3, a 175 billion parameter text-generation model trained by the startup OpenAI on an astoundingly large text corpus, including most digitally available books, Wikipedia, and a large fraction of a crawl of the entire internet. GPT-3 made headlines in 2020 due to its capability to generate plausible-sounding text paragraphs on virtually any topic, a prowess that has fed a short-lived hype wave worthy of the most torrid AI summer.

12.1.2 How do you generate sequence data?

The universal way to generate sequence data in deep learning is to train a model (usually a Transformer or an RNN) to predict the next token or next few tokens in a sequence, using the previous tokens as input. For instance, given the input "the cat is on the," the model is trained to predict the target "mat," the next word. As usual when working with text data, tokens are typically words or characters, and any network that can model the probability of the next token given the previous ones is called a *language model*. A language model captures the *latent space* of language: its statistical structure.

Once you have such a trained language model, you can *sample* from it (generate new sequences): you feed it an initial string of text (called *conditioning data*), ask it to generate the next character or the next word (you can even generate several tokens at once), add the generated output back to the input data, and repeat the process many times (see figure 12.1). This loop allows you to generate sequences of arbitrary length that reflect the structure of the data on which the model was trained: sequences that look *almost* like human-written sentences.

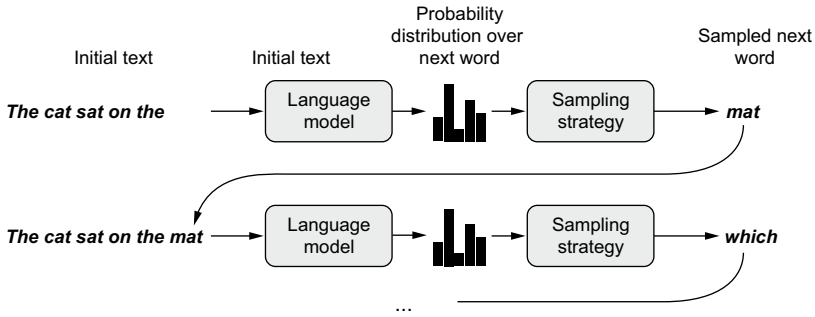

Figure 12.1 The process of word-by-word text generation using a language model

12.1.3 The importance of the sampling strategy

When generating text, the way you choose the next token is crucially important. A naive approach is *greedy sampling*, consisting of always choosing the most likely next character. But such an approach results in repetitive, predictable strings that don't look like coherent language. A more interesting approach makes slightly more surprising choices: it introduces randomness in the sampling process by sampling from the probability distribution for the next character. This is called *stochastic sampling* (recall that *stochasticity* is what we call *randomness* in this field). In such a setup, if a word has probability 0.3 of being next in the sentence according to the model, you'll choose it 30% of the time. Note that greedy sampling can also be cast as sampling from a probability distribution: one where a certain word has probability 1 and all others have probability 0.

Sampling probabilistically from the softmax output of the model is neat: it allows even unlikely words to be sampled some of the time, generating more interesting-looking sentences and sometimes showing creativity by coming up with new, realistic-sounding sentences that didn't occur in the training data. But there's one issue with this strategy: it doesn't offer a way to *control the amount of randomness* in the sampling process.

Why would you want more or less randomness? Consider an extreme case: pure random sampling, where you draw the next word from a uniform probability distribution, and every word is equally likely. This scheme has maximum randomness; in other words, this probability distribution has maximum entropy. Naturally, it won't produce anything interesting. At the other extreme, greedy sampling doesn't produce anything interesting, either, and has no randomness: the corresponding probability distribution has minimum entropy. Sampling from the "real" probability distribution—the distribution that is output by the model's softmax function—constitutes an intermediate point between these two extremes. But there are many other intermediate points of higher or lower entropy that you may want to explore. Less entropy will give the generated sequences a more predictable structure (and thus they will potentially be more realistic looking), whereas more entropy will result in more surprising and creative sequences. When sampling from generative models, it's always good to explore different amounts of randomness in the generation process. Because we—humans—are the ultimate judges of how interesting the generated data is, interestingness is highly subjective, and there's no telling in advance where the point of optimal entropy lies.

In order to control the amount of stochasticity in the sampling process, we'll introduce a parameter called the *softmax temperature*, which characterizes the entropy of the probability distribution used for sampling: it characterizes how surprising or predictable the choice of the next word will be. Given a `temperature` value, a new probability distribution is computed from the original one (the softmax output of the model) by reweighting it in the following way.

Listing 12.1 Reweighting a probability distribution to a different temperature

original_distribution is a 1D NumPy array of probability values that must sum to 1. temperature is a factor quantifying the entropy of the output distribution.

```
import numpy as np
def reweight_distribution(original_distribution, temperature=0.5):
    distribution = np.log(original_distribution) / temperature
    distribution = np.exp(distribution)
    return distribution / np.sum(distribution)
```

Returns a reweighted version of the original distribution. The sum of the distribution may no longer be 1, so you divide it by its sum to obtain the new distribution.

Higher temperatures result in sampling distributions of higher entropy that will generate more surprising and unstructured generated data, whereas a lower temperature will result in less randomness and much more predictable generated data (see figure 12.2).

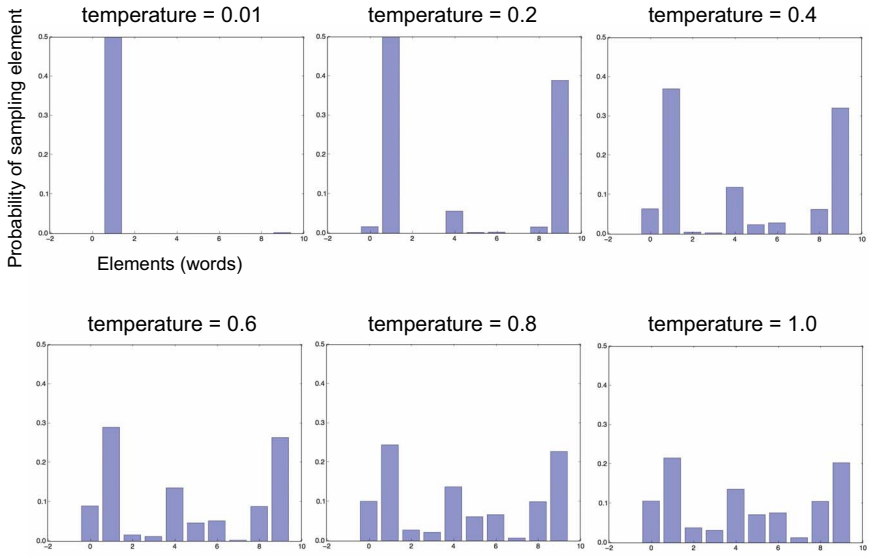

Figure 12.2 Different reweightings of one probability distribution. Low temperature = more deterministic, high temperature = more random.

12.1.4 Implementing text generation with Keras

Let's put these ideas into practice in a Keras implementation. The first thing you need is a lot of text data that you can use to learn a language model. You can use any sufficiently large text file or set of text files—Wikipedia, *The Lord of the Rings*, and so on.

In this example, we'll keep working with the IMDB movie review dataset from the last chapter, and we'll learn to generate never-read-before movie reviews. As such, our language model will be a model of the style and topics of these movie reviews specifically, rather than a general model of the English language.

PREPARING THE DATA

Just like in the previous chapter, let's download and uncompress the IMDB movie reviews dataset.

Listing 12.2 Downloading and uncompressing the IMDB movie reviews dataset

```
!wget https://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz
!tar -xf aclImdb_v1.tar.gz
```

You're already familiar with the structure of the data: we get a folder named aclImdb containing two subfolders, one for negative-sentiment movie reviews, and one for positive-sentiment reviews. There's one text file per review. We'll call `text_dataset_from_directory` with `label_mode=None` to create a dataset that reads from these files and yields the text content of each file.

Listing 12.3 Creating a dataset from text files (one file = one sample)

```
import tensorflow as tf
from tensorflow import keras
dataset = keras.utils.text_dataset_from_directory(
    directory="aclImdb", label_mode=None, batch_size=256)
dataset = dataset.map(lambda x: tf.strings.regex_replace(x, "<br />", " "))
```

Strip the
 HTML tag that occurs in many of the reviews. This did not matter much for text classification, but we wouldn't want to generate
 tags in this example!

Now let's use a `TextVectorization` layer to compute the vocabulary we'll be working with. We'll only use the first `sequence_length` words of each review: our `TextVectorization` layer will cut off anything beyond that when vectorizing a text.

Listing 12.4 Preparing a `TextVectorization` layer

```
from tensorflow.keras.layers import TextVectorization

sequence_length = 100
vocab_size = 15000
text_vectorization = TextVectorization(
    max_tokens=vocab_size,
    output_mode="int",
    output_sequence_length=sequence_length,
)
text_vectorization.adapt(dataset)
```

We'll only consider the top 15,000 most common words—anything else will be treated as the out-of-vocabulary token, "[UNK]".

We want to return integer word index sequences.

We'll work with inputs and targets of length 100 (but since we'll offset the targets by 1, the model will actually see sequences of length 99).

Let's use the layer to create a language modeling dataset where input samples are vectorized texts, and corresponding targets are the same texts offset by one word.

Listing 12.5 Setting up a language modeling dataset

```
def prepare_lm_dataset(text_batch):
    vectorized_sequences = text_vectorization(text_batch)
```

Convert a batch of texts (strings) to a batch of integer sequences.

```
    x = vectorized_sequences[:, :-1]
    y = vectorized_sequences[:, 1:]
    return x, y

lm_dataset = dataset.map(prepare_lm_dataset, num_parallel_calls=4)
```

Create inputs by cutting off the last word of the sequences.

Create targets by offsetting the sequences by 1.

A Transformer-based sequence-to-sequence model

We'll train a model to predict a probability distribution over the next word in a sentence, given a number of initial words. When the model is trained, we'll feed it with a prompt, sample the next word, add that word back to the prompt, and repeat, until we've generated a short paragraph.

Like we did for temperature forecasting in chapter 10, we could train a model that takes as input a sequence of N words and simply predicts word N+1. However, there are several issues with this setup in the context of sequence generation.

First, the model would only learn to produce predictions when N words were available, but it would be useful to be able to start predicting with fewer than N words. Otherwise we'd be constrained to only use relatively long prompts (in our implementation, N=100 words). We didn't have this need in chapter 10.

Second, many of our training sequences will be mostly overlapping. Consider `N = 4`. The text "A complete sentence must have, at minimum, three things: a subject, verb, and an object" would be used to generate the following training sequences:

- "A complete sentence must"
- "complete sentence must have"
- "sentence must have at"
- and so on, until "verb and an object"

A model that treats each such sequence as an independent sample would have to do a lot of redundant work, re-encoding multiple times subsequences that it has largely seen before. In chapter 10, this wasn't much of a problem, because we didn't have that many training samples in the first place, and we needed to benchmark dense and convolutional models, for which redoing the work every time is the only option. We could try to alleviate this redundancy problem by using *strides* to sample our sequences—skipping a few words between two consecutive samples. But that would reduce our number of training samples while only providing a partial solution.

To address these two issues, we'll use a *sequence-to-sequence model*: we'll feed sequences of N words (indexed from 0 to N) into our model, and we'll predict the sequence offset by one (from 1 to `N+1`). We'll use causal masking to make sure that, for any `i`, the model will only be using words from 0 to `i` in order to predict the word `i + 1`. This means that we're simultaneously training the model to solve N mostly overlapping but different problems: predicting the next words given a sequence of `1 <= i <= N` prior words (see figure 12.3). At generation time, even if you only prompt the model with a single word, it will be able to give you a probability distribution for the next possible words.

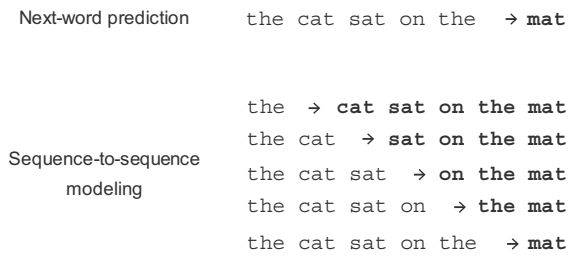

Figure 12.3 Compared to plain next-word prediction, sequence-to-sequence modeling simultaneously optimizes for multiple prediction problems.

Note that we could have used a similar sequence-to-sequence setup on our temperature forecasting problem in chapter 10: given a sequence of 120 hourly data points, learn to generate a sequence of 120 temperatures offset by 24 hours in the future. You'd be not only solving the initial problem, but also solving the 119 related problems of forecasting temperature in 24 hours, given `1 <= i < 120` prior hourly data points. If you try to retrain the RNNs from chapter 10 in a sequence-to-sequence setup, you'll find that you get similar but incrementally worse results, because the constraint of solving these additional 119 related problems with the same model interferes slightly with the task we actually do care about.

In the previous chapter, you learned about the setup you can use for sequence-to-sequence learning in the general case: feed the source sequence into an encoder, and then feed both the encoded sequence and the target sequence into a decoder that tries to predict the same target sequence offset by one step. When you're doing text generation, there is no source sequence: you're just trying to predict the next tokens in the target sequence given past tokens, which we can do using only the decoder. And thanks to causal padding, the decoder will only look at words `0…N` to predict the word `N+1`.

Let's implement our model—we're going to reuse the building blocks we created in chapter 11: `PositionalEmbedding` and `TransformerDecoder`.

Listing 12.6 A simple Transformer-based language model

```
from tensorflow.keras import layers
embed_dim = 256
latent_dim = 2048
num_heads = 2

inputs = keras.Input(shape=(None,), dtype="int64")
x = PositionalEmbedding(sequence_length, vocab_size, embed_dim)(inputs)
x = TransformerDecoder(embed_dim, latent_dim, num_heads)(x, x)
outputs = layers.Dense(vocab_size, activation="softmax")(x)
model = keras.Model(inputs, outputs)
model.compile(loss="sparse_categorical_crossentropy", optimizer="rmsprop")
```

Softmax over possible vocabulary words, computed for each output sequence timestep.

12.1.5 A text-generation callback with variable-temperature sampling

We'll use a callback to generate text using a range of different temperatures after every epoch. This allows you to see how the generated text evolves as the model begins to converge, as well as the impact of temperature in the sampling strategy. To

seed text generation, we'll use the prompt "this movie": all of our generated texts will start with this.

Listing 12.7 The text-generation callback

```
import numpy as np

tokens_index = dict(enumerate(text_vectorization.get_vocabulary()))

def sample_next(predictions, temperature=1.0):
    predictions = np.asarray(predictions).astype("float64")
    predictions = np.log(predictions) / temperature
    exp_preds = np.exp(predictions)
    predictions = exp_preds / np.sum(exp_preds)
    probas = np.random.multinomial(1, predictions, 1)
    return np.argmax(probas)

class TextGenerator(keras.callbacks.Callback):
    def __init__(self,
                 prompt,
                 generate_length,
                 model_input_length,
                 temperatures=(1.,),
                 print_freq=1):
        self.prompt = prompt
        self.generate_length = generate_length
        self.model_input_length = model_input_length
        self.temperatures = temperatures
        self.print_freq = print_freq

    def on_epoch_end(self, epoch, logs=None):
        if (epoch + 1) % self.print_freq != 0:
            return
        for temperature in self.temperatures:
            print("== Generating with temperature", temperature)
            sentence = self.prompt
            for i in range(self.generate_length):
                tokenized_sentence = text_vectorization([sentence])
                predictions = self.model(tokenized_sentence)
                next_token = sample_next(predictions[0, i, :])
                sampled_token = tokens_index[next_token]
                sentence += " " + sampled_token
            print(sentence)

prompt = "This movie"
text_gen_callback = TextGenerator(
    prompt,
    generate_length=50,
    model_input_length=sequence_length,
    temperatures=(0.2, 0.5, 0.7, 1., 1.5))
```

Dict that maps word indices back to strings, to be used for text decoding

Implements variable-temperature sampling from a probability distribution

Prompt that we use to seed text generation

How many words to generate

Range of temperatures to use for sampling

When generating text, we start from our prompt.

Feed the current sequence into our model.

Append the new word to the current sequence and repeat.

Retrieve the predictions for the last timestep, and use them to sample a new word.

We'll use a diverse range of temperatures to sample text, to demonstrate the effect of temperature on text generation.

Let's `fit()` this thing.

Listing 12.8 Fitting the language model

```
model.fit(lm_dataset, epochs=200, callbacks=[text_gen_callback])
```

Here are some cherrypicked examples of what we're able to generate after 200 epochs of training. Note that punctuation isn't part of our vocabulary, so none of our generated text has any punctuation:

- With `temperature=0.2`
  - "this movie is a [UNK] of the original movie and the first half hour of the movie is pretty good but it is a very good movie it is a good movie for the time period"
  - "this movie is a [UNK] of the movie it is a movie that is so bad that it is a [UNK] movie it is a movie that is so bad that it makes you laugh and cry at the same time it is not a movie i dont think ive ever seen"
- With `temperature=0.5`
  - "this movie is a [UNK] of the best genre movies of all time and it is not a good movie it is the only good thing about this movie i have seen it for the first time and i still remember it being a [UNK] movie i saw a lot of years"
  - "this movie is a waste of time and money i have to say that this movie was a complete waste of time i was surprised to see that the movie was made up of a good movie and the movie was not very good but it was a waste of time and"
- With `temperature=0.7`
  - "this movie is fun to watch and it is really funny to watch all the characters are extremely hilarious also the cat is a bit like a [UNK] [UNK] and a hat [UNK] the rules of the movie can be told in another scene saves it from being in the back of"
  - "this movie is about [UNK] and a couple of young people up on a small boat in the middle of nowhere one might find themselves being exposed to a [UNK] dentist they are killed by [UNK] i was a huge fan of the book and i havent seen the original so it"
- With `temperature=1.0`
  - "this movie was entertaining i felt the plot line was loud and touching but on a whole watch a stark contrast to the artistic of the original we watched the original version of england however whereas arc was a bit of a little too ordinary the [UNK] were the present parent [UNK]"
  - "this movie was a masterpiece away from the storyline but this movie was simply exciting and frustrating it really entertains friends like this the actors in this movie try to go straight from the sub thats image and they make it a really good tv show"
- With `temperature=1.5`
  - "this movie was possibly the worst film about that 80 women its as weird insightful actors like barker movies but in great buddies yes no decorated shield even [UNK] land dinosaur ralph ian was must make a play happened falls after miscast [UNK] bach not really not wrestlemania seriously sam didnt exist"

- "this movie could be so unbelievably lucas himself bringing our country wildly funny things has is for the garish serious and strong performances colin writing more detailed dominated but before and that images gears burning the plate patriotism we you expected dyan bosses devotion to must do your own duty and another"

As you can see, a low temperature value results in very boring and repetitive text and can sometimes cause the generation process to get stuck in a loop. With higher temperatures, the generated text becomes more interesting, surprising, even creative. With a very high temperature, the local structure starts to break down, and the output looks largely random. Here, a good generation temperature would seem to be about 0.7. Always experiment with multiple sampling strategies! A clever balance between learned structure and randomness is what makes generation interesting.

Note that by training a bigger model, longer, on more data, you can achieve generated samples that look far more coherent and realistic than this one—the output of a model like GPT-3 is a good example of what can be done with language models (GPT-3 is effectively the same thing as what we trained in this example, but with a deep stack of Transformer decoders, and a much bigger training corpus). But don't expect to ever generate any meaningful text, other than through random chance and the magic of your own interpretation: all you're doing is sampling data from a statistical model of which words come after which words. Language models are all form and no substance.

Natural language is many things: a communication channel, a way to act on the world, a social lubricant, a way to formulate, store, and retrieve your own thoughts . . . These uses of languages are where its meaning originates. A deep learning "language model," despite its name, captures effectively none of these fundamental aspects of language. It cannot communicate (it has nothing to communicate about and no one to communicate with), it cannot act on the world (it has no agency and no intent), it cannot be social, and it doesn't have any thoughts to process with the help of words. Language is the operating system of the mind, and so, for language to be meaningful, it needs a mind to leverage it.

What a language model does is capture the statistical structure of the observable artifacts—books, online movie reviews, tweets—that we generate as we use language to live our lives. The fact that these artifacts have a statistical structure at all is a side effect of how humans implement language. Here's a thought experiment: what if our languages did a better job of compressing communications, much like computers do with most digital communications? Language would be no less meaningful and could still fulfill its many purposes, but it would lack any intrinsic statistical structure, thus making it impossible to model as you just did.

12.1.6 Wrapping up

- You can generate discrete sequence data by training a model to predict the next token(s), given previous tokens.
- In the case of text, such a model is called a *language model*. It can be based on either words or characters.
- Sampling the next token requires a balance between adhering to what the model judges likely, and introducing randomness.
- One way to handle this is the notion of softmax temperature. Always experiment with different temperatures to find the right one.

12.2 DeepDream

DeepDream is an artistic image-modification technique that uses the representations learned by convolutional neural networks. It was first released by Google in the summer of 2015 as an implementation written using the Caffe deep learning library (this was several months before the first public release of TensorFlow).[3] It quickly became an internet sensation thanks to the trippy pictures it could generate (see, for example, figure 12.4), full of algorithmic pareidolia artifacts, bird feathers, and dog eyes—a byproduct of the fact that the DeepDream convnet was trained on ImageNet, where dog breeds and bird species are vastly overrepresented.

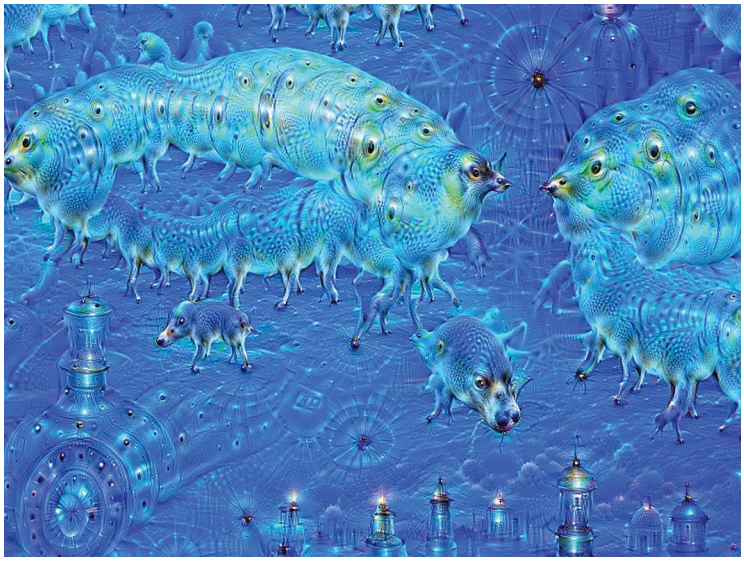

Figure 12.4 Example of a DeepDream output image

[3] Alexander Mordvintsev, Christopher Olah, and Mike Tyka, "DeepDream: A Code Example for Visualizing Neural Networks," Google Research Blog, July 1, 2015, http://mng.bz/xXlM.

The DeepDream algorithm is almost identical to the convnet filter-visualization technique introduced in chapter 9, consisting of running a convnet in reverse: doing gradient ascent on the input to the convnet in order to maximize the activation of a specific filter in an upper layer of the convnet. DeepDream uses this same idea, with a few simple differences:

- With DeepDream, you try to maximize the activation of entire layers rather than that of a specific filter, thus mixing together visualizations of large numbers of features at once.
- You start not from blank, slightly noisy input, but rather from an existing image—thus the resulting effects latch on to preexisting visual patterns, distorting elements of the image in a somewhat artistic fashion.
- The input images are processed at different scales (called *octaves*), which improves the quality of the visualizations.

Let's make some DeepDreams.

12.2.1 Implementing DeepDream in Keras

Let's start by retrieving a test image to dream with. We'll use a view of the rugged Northern California coast in the winter (figure 12.5).

Listing 12.9 Fetching the test image

```
from tensorflow import keras
import matplotlib.pyplot as plt

base_image_path = keras.utils.get_file(
    "coast.jpg", origin="https://img-datasets.s3.amazonaws.com/coast.jpg")

plt.axis("off")
plt.imshow(keras.utils.load_img(base_image_path))
```

Figure 12.5 Our test image

Next, we need a pretrained convnet. In Keras, many such convnets are available: VGG16, VGG19, Xception, ResNet50, and so on, all available with weights pretrained on ImageNet. You can implement DeepDream with any of them, but your base model of choice will naturally affect your visualizations, because different architectures result in different learned features. The convnet used in the original DeepDream release was an Inception model, and in practice, Inception is known to produce nice-looking DeepDreams, so we'll use the Inception V3 model that comes with Keras.

Listing 12.10 Instantiating a pretrained `InceptionV3` model

```
from tensorflow.keras.applications import inception_v3
model = inception_v3.InceptionV3(weights="imagenet", include_top=False)
```

We'll use our pretrained convnet to create a feature exactor model that returns the activations of the various intermediate layers, listed in the following code. For each layer, we pick a scalar score that weights the contribution of the layer to the loss we will seek to maximize during the gradient ascent process. If you want a complete list of layer names that you can use to pick new layers to play with, just use `model.summary()`.

Listing 12.11 Configuring the contribution of each layer to the DeepDream loss

```
layer_settings = {
    "mixed4": 1.0,
    "mixed5": 1.5,
    "mixed6": 2.0,
    "mixed7": 2.5,
}
outputs_dict = dict(
    [
        (layer.name, layer.output)
        for layer in [model.get_layer(name)
                      for name in layer_settings.keys()]
    ]
)
feature_extractor = keras.Model(inputs=model.inputs, outputs=outputs_dict)
```

Layers for which we try to maximize activation, as well as their weight in the total loss. You can tweak these setting to obtain new visual effects.

Symbolic outputs of each layer

Model that returns the activation values for every target layer (as a dict)

Next, we'll compute the *loss*: the quantity we'll seek to maximize during the gradient-ascent process at each processing scale. In chapter 9, for filter visualization, we tried to maximize the value of a specific filter in a specific layer. Here, we'll simultaneously maximize the activation of all filters in a number of layers. Specifically, we'll maximize a weighted mean of the L2 norm of the activations of a set of high-level layers. The exact set of layers we choose (as well as their contribution to the final loss) has a major influence on the visuals we'll be able to produce, so we want to make these parameters easily configurable. Lower layers result in geometric patterns, whereas higher layers result in visuals in which you can recognize some classes from ImageNet (for example, birds or dogs). We'll start from a somewhat arbitrary configuration involving four layers—but you'll definitely want to explore many different configurations later.

Listing 12.12 The DeepDream loss

```
def compute_loss(input_image):
    features = feature_extractor(input_image)
    loss = tf.zeros(shape=())
    for name in features.keys():
        coeff = layer_settings[name]
        activation = features[name]
        loss += coeff * tf.reduce_mean(tf.square(activation[:, 2:-2, 2:-2, :]))
    return loss
```

Extract activations.

Initialize the loss to 0.

We avoid border artifacts by only involving non-border pixels in the loss.

Now let's set up the gradient ascent process that we will run at each octave. You'll recognize that it's the same thing as the filter-visualization technique from chapter 9! The DeepDream algorithm is simply a multiscale form of filter visualization.

Listing 12.13 The DeepDream gradient ascent process

```
import tensorflow as tf

@tf.function
def gradient_ascent_step(image, learning_rate):
    with tf.GradientTape() as tape:
        tape.watch(image)
        loss = compute_loss(image)
    grads = tape.gradient(loss, image)
    grads = tf.math.l2_normalize(grads)
    image += learning_rate * grads
    return loss, image
```

We make the training step fast by compiling it as a tf.function.

Compute gradients of DeepDream loss with respect to the current image.

Normalize gradients (the same trick we used in chapter 9).

```
def gradient_ascent_loop(image, iterations, learning_rate, max_loss=None):
    for i in range(iterations):
        loss, image = gradient_ascent_step(image, learning_rate)
        if max_loss is not None and loss > max_loss:
            break
        print(f"... Loss value at step {i}: {loss:.2f}")
    return image
```

This runs gradient ascent for a given image scale (octave).

Repeatedly update the image in a way that increases the DeepDream loss.

Break out if the loss crosses a certain threshold (over-optimizing would create unwanted image artifacts).

Finally, the outer loop of the DeepDream algorithm. First, we'll define a list of *scales* (also called *octaves*) at which to process the images. We'll process our image over three different such "octaves." For each successive octave, from the smallest to the largest, we'll run 20 gradient ascent steps via `gradient_ascent_loop()` to maximize the loss we previously defined. Between each octave, we'll upscale the image by 40% (1.4x): we'll start by processing a small image and then increasingly scale it up (see figure 12.6).

We define the parameters of this process in the following code. Tweaking these parameters will allow you to achieve new effects!

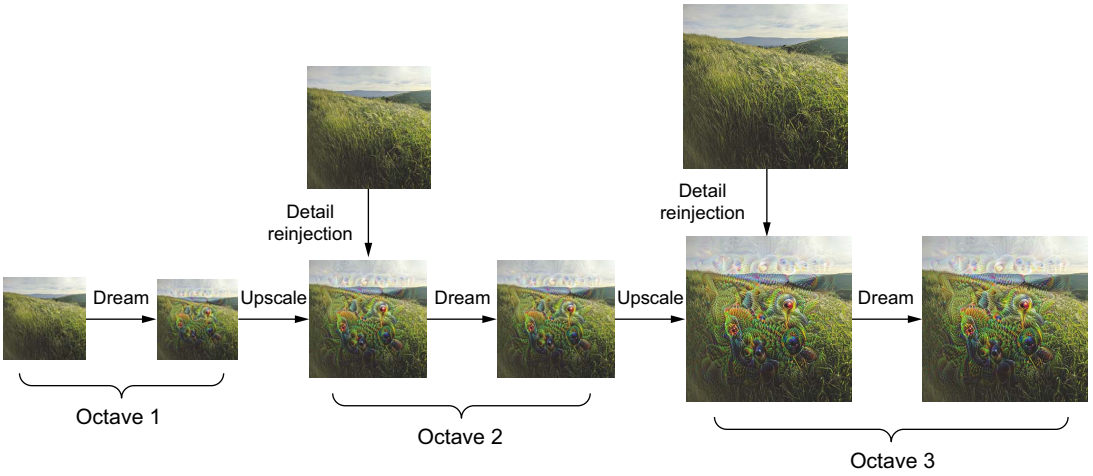

Figure 12.6 The DeepDream process: successive scales of spatial processing (octaves) and detail re-injection upon upscaling

```
step = 20.
num_octave = 3
octave_scale = 1.4
iterations = 30
max_loss = 15.
```

Gradient ascent step size

Number of scales at which to run gradient ascent

Size ratio between successive scales

Number of gradient ascent steps per scale

We'll stop the gradient ascent process for a scale if the loss gets higher than this.

We're also going to need a couple of utility functions to load and save images.

Listing 12.14 Image processing utilities

```
import numpy as np

def preprocess_image(image_path):
    img = keras.utils.load_img(image_path)
    img = keras.utils.img_to_array(img)
    img = np.expand_dims(img, axis=0)
    img = keras.applications.inception_v3.preprocess_input(img)
    return img

def deprocess_image(img):
    img = img.reshape((img.shape[1], img.shape[2], 3))
    img /= 2.0
    img += 0.5
    img *= 255.
    img = np.clip(img, 0, 255).astype("uint8")
    return img
```

Util function to open, resize, and format pictures into appropriate arrays

Util function to convert a NumPy array into a valid image

Undo inception v3 preprocessing.

Convert to uint8 and clip to the valid range [0, 255].

This is the outer loop. To avoid losing a lot of image detail after each successive scale-up (resulting in increasingly blurry or pixelated images), we can use a simple trick:

after each scale-up, we'll re-inject the lost details back into the image, which is possible because we know what the original image should look like at the larger scale. Given a small image size *S* and a larger image size *L*, we can compute the difference between the original image resized to size *L* and the original resized to size *S*—this difference quantifies the details lost when going from *S* to *L*.

Listing 12.15 Running gradient ascent over multiple successive "octaves"

```
original_img = preprocess_image(base_image_path)
original_shape = original_img.shape[1:3]

successive_shapes = [original_shape]
for i in range(1, num_octave):
    shape = tuple([int(dim / (octave_scale ** i)) for dim in original_shape])
    successive_shapes.append(shape)
successive_shapes = successive_shapes[::-1]

shrunk_original_img = tf.image.resize(original_img, successive_shapes[0])

img = tf.identity(original_img)
for i, shape in enumerate(successive_shapes):
    print(f"Processing octave {i} with shape {shape}")
    img = tf.image.resize(img, shape)
    img = gradient_ascent_loop(
        img, iterations=iterations, learning_rate=step, max_loss=max_loss
    )
    upscaled_shrunk_original_img = tf.image.resize(shrunk_original_img, shape)
    same_size_original = tf.image.resize(original_img, shape)
    lost_detail = same_size_original - upscaled_shrunk_original_img
    img += lost_detail
    shrunk_original_img = tf.image.resize(original_img, shape)

keras.utils.save_img("dream.png", deprocess_image(img.numpy()))
```

Load the test image.

Compute the target shape of the image at different octaves.

Iterate over the different octaves.

Make a copy of the image (we need to keep the original around).

Run gradient ascent, altering the dream.

Scale up the dream image.

Compute the high-quality version of the original image at this size.

Scale up the smaller version of the original image: it will be pixellated.

Save the final result.

Re-inject lost detail into the dream.

The difference between the two is the detail that was lost when scaling up.

NOTE Because the original Inception V3 network was trained to recognize concepts in images of size 299 × 299, and given that the process involves scaling the images down by a reasonable factor, the DeepDream implementation produces much better results on images that are somewhere between 300 × 300 and 400 × 400. Regardless, you can run the same code on images of any size and any ratio.

On a GPU, it only takes a few seconds to run the whole thing. Figure 12.7 shows the result of our dream configuration on the test image.

I strongly suggest that you explore what you can do by adjusting which layers you use in your loss. Layers that are lower in the network contain more-local, less-abstract representations and lead to dream patterns that look more geometric. Layers that are higher up lead to more-recognizable visual patterns based on the most common objects

Figure 12.7 Running the DeepDream code on the test image

found in ImageNet, such as dog eyes, bird feathers, and so on. You can use random generation of the parameters in the `layer_settings` dictionary to quickly explore many different layer combinations. Figure 12.8 shows a range of results obtained on an image of a delicious homemade pastry using different layer configurations.

Figure 12.8 Trying a range of DeepDream configurations on an example image

12.2.2 Wrapping up

- DeepDream consists of running a convnet in reverse to generate inputs based on the representations learned by the network.
- The results produced are fun and somewhat similar to the visual artifacts induced in humans by the disruption of the visual cortex via psychedelics.
- Note that the process isn't specific to image models or even to convnets. It can be done for speech, music, and more.

12.3 Neural style transfer

In addition to DeepDream, another major development in deep-learning-driven image modification is *neural style transfer*, introduced by Leon Gatys et al. in the summer of 2015.[4] The neural style transfer algorithm has undergone many refinements and spawned many variations since its original introduction, and it has made its way into many smartphone photo apps. For simplicity, this section focuses on the formulation described in the original paper.

Neural style transfer consists of applying the style of a reference image to a target image while conserving the content of the target image. Figure 12.9 shows an example.

Figure 12.9 A style transfer example

In this context, *style* essentially means textures, colors, and visual patterns in the image, at various spatial scales, and the content is the higher-level macrostructure of the image. For instance, blue-and-yellow circular brushstrokes are considered to be the style in figure 12.9 (using *Starry Night* by Vincent Van Gogh), and the buildings in the Tübingen photograph are considered to be the content.

The idea of style transfer, which is tightly related to that of texture generation, has had a long history in the image-processing community prior to the development of neural style transfer in 2015. But as it turns out, the deep-learning-based implementations of style transfer offer results unparalleled by what had been previously achieved with classical computer vision techniques, and they triggered an amazing renaissance in creative applications of computer vision.

[4] Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge, "A Neural Algorithm of Artistic Style," arXiv (2015), https://arxiv.org/abs/1508.06576.

The key notion behind implementing style transfer is the same idea that's central to all deep learning algorithms: you define a loss function to specify what you want to achieve, and you minimize this loss. We know what we want to achieve: conserving the content of the original image while adopting the style of the reference image. If we were able to mathematically define *content* and *style*, then an appropriate loss function to minimize would be the following:

```
loss = (distance(style(reference_image) - style(combination_image)) +
        distance(content(original_image) - content(combination_image)))
```

Here, `distance` is a norm function such as the L2 norm, `content` is a function that takes an image and computes a representation of its content, and `style` is a function that takes an image and computes a representation of its style. Minimizing this loss causes `style(combination_image)` to be close to `style(reference_image)`, and `content(combination_image)` is close to `content(original_image)`, thus achieving style transfer as we defined it.

A fundamental observation made by Gatys et al. was that deep convolutional neural networks offer a way to mathematically define the `style` and `content` functions. Let's see how.

12.3.1 The content loss

As you already know, activations from earlier layers in a network contain *local* information about the image, whereas activations from higher layers contain increasingly global, abstract information. Formulated in a different way, the activations of the different layers of a convnet provide a decomposition of the contents of an image over different spatial scales. Therefore, you'd expect the content of an image, which is more global and abstract, to be captured by the representations of the upper layers in a convnet.

A good candidate for content loss is thus the L2 norm between the activations of an upper layer in a pretrained convnet, computed over the target image, and the activations of the same layer computed over the generated image. This guarantees that, as seen from the upper layer, the generated image will look similar to the original target image. Assuming that what the upper layers of a convnet see is really the content of their input images, this works as a way to preserve image content.

12.3.2 The style loss

The content loss only uses a single upper layer, but the style loss as defined by Gatys et al. uses multiple layers of a convnet: you try to capture the appearance of the style-reference image at all spatial scales extracted by the convnet, not just a single scale. For the style loss, Gatys et al. use the *Gram matrix* of a layer's activations: the inner product of the feature maps of a given layer. This inner product can be understood as representing a map of the correlations between the layer's features. These feature

correlations capture the statistics of the patterns of a particular spatial scale, which empirically correspond to the appearance of the textures found at this scale.

Hence, the style loss aims to preserve similar internal correlations within the activations of different layers, across the style-reference image and the generated image. In turn, this guarantees that the textures found at different spatial scales look similar across the style-reference image and the generated image.

In short, you can use a pretrained convnet to define a loss that will do the following:

- Preserve content by maintaining similar high-level layer activations between the original image and the generated image. The convnet should "see" both the original image and the generated image as containing the same things.
- Preserve style by maintaining similar *correlations* within activations for both low-level layers and high-level layers. Feature correlations capture *textures*: the generated image and the style-reference image should share the same textures at different spatial scales.

Now let's look at a Keras implementation of the original 2015 neural style transfer algorithm. As you'll see, it shares many similarities with the DeepDream implementation we developed in the previous section.

12.3.3 Neural style transfer in Keras

Neural style transfer can be implemented using any pretrained convnet. Here, we'll use the VGG19 network used by Gatys et al. VGG19 is a simple variant of the VGG16 network introduced in chapter 5, with three more convolutional layers.

Here's the general process:

- Set up a network that computes VGG19 layer activations for the style-reference image, the base image, and the generated image at the same time.
- Use the layer activations computed over these three images to define the loss function described earlier, which we'll minimize in order to achieve style transfer.
- Set up a gradient-descent process to minimize this loss function.

Let's start by defining the paths to the style-reference image and the base image. To make sure that the processed images are a similar size (widely different sizes make style transfer more difficult), we'll later resize them all to a shared height of 400 px.

Listing 12.16 Getting the style and content images

```
from tensorflow import keras

base_image_path = keras.utils.get_file(
    "sf.jpg", origin="https://img-datasets.s3.amazonaws.com/sf.jpg")
style_reference_image_path = keras.utils.get_file(
    "starry_night.jpg",
    origin="https://img-datasets.s3.amazonaws.com/starry_night.jpg")
```

Path to the image we want to transform

Path to the style image

```
original_width, original_height = keras.utils.load_img(base_image_path).size
img_height = 400
img_width = round(original_width * img_height / original_height)
```

Dimensions of the generated picture

Our content image is shown in figure 12.10, and figure 12.11 shows our style image.

Figure 12.10 Content image: San Francisco from Nob Hill

Figure 12.11 Style image: Starry Night by Van Gogh

We also need some auxiliary functions for loading, preprocessing, and postprocessing the images that go in and out of the VGG19 convnet.

Listing 12.17 Auxiliary functions

```
import numpy as np

def preprocess_image(image_path):
    img = keras.utils.load_img(
        image_path, target_size=(img_height, img_width))
    img = keras.utils.img_to_array(img)
    img = np.expand_dims(img, axis=0)
    img = keras.applications.vgg19.preprocess_input(img)
    return img

def deprocess_image(img):
    img = img.reshape((img_height, img_width, 3))
    img[:, :, 0] += 103.939
    img[:, :, 1] += 116.779
    img[:, :, 2] += 123.68
    img = img[:, :, ::-1]
    img = np.clip(img, 0, 255).astype("uint8")
    return img
```

Util function to open, resize, and format pictures into appropriate arrays

Util function to convert a NumPy array into a valid image

Zero-centering by removing the mean pixel value from ImageNet. This reverses a transformation done by vgg19.preprocess_input.

Converts images from 'BGR' to 'RGB'. This is also part of the reversal of vgg19.preprocess_input.

Let's set up the VGG19 network. Like in the DeepDream example, we'll use the pretrained convnet to create a feature exactor model that returns the activations of intermediate layers—all layers in the model this time.

Listing 12.18 Using a pretrained VGG19 model to create a feature extractor

```
model = keras.applications.vgg19.VGG19(weights="imagenet", include_top=False)

outputs_dict = dict([(layer.name, layer.output) for layer in model.layers])
feature_extractor = keras.Model(inputs=model.inputs, outputs=outputs_dict)
```

Build a VGG19 model loaded with pretrained ImageNet weights.

Model that returns the activation values for every target layer (as a dict)

Let's define the content loss, which will make sure the top layer of the VGG19 convnet has a similar view of the style image and the combination image.

Listing 12.19 Content loss

```
def content_loss(base_img, combination_img):
    return tf.reduce_sum(tf.square(combination_img - base_img))
```

Next is the style loss. It uses an auxiliary function to compute the Gram matrix of an input matrix: a map of the correlations found in the original feature matrix.

Listing 12.20 Style loss

```
def gram_matrix(x):
    x = tf.transpose(x, (2, 0, 1))
    features = tf.reshape(x, (tf.shape(x)[0], -1))
    gram = tf.matmul(features, tf.transpose(features))
    return gram

def style_loss(style_img, combination_img):
    S = gram_matrix(style_img)
    C = gram_matrix(combination_img)
    channels = 3
    size = img_height * img_width
    return tf.reduce_sum(tf.square(S - C)) / (4.0 * (channels ** 2) * (size ** 2))
```

To these two loss components, you add a third: the *total variation loss*, which operates on the pixels of the generated combination image. It encourages spatial continuity in the generated image, thus avoiding overly pixelated results. You can interpret it as a regularization loss.

Listing 12.21 Total variation loss

```
def total_variation_loss(x):
    a = tf.square(
        x[:, : img_height - 1, : img_width - 1, :] - x[:, 1:, : img_width - 1, :]
    )
    b = tf.square(
        x[:, : img_height - 1, : img_width - 1, :] - x[:, : img_height - 1, 1:, :]
    )
    return tf.reduce_sum(tf.pow(a + b, 1.25))
```

The loss that you minimize is a weighted average of these three losses. To compute the content loss, you use only one upper layer—the `block5_conv2` layer—whereas for the style loss, you use a list of layers that spans both low-level and high-level layers. You add the total variation loss at the end.

Depending on the style-reference image and content image you're using, you'll likely want to tune the `content_weight` coefficient (the contribution of the content loss to the total loss). A higher `content_weight` means the target content will be more recognizable in the generated image.

Listing 12.22 Defining the final loss that you'll minimize

```
style_layer_names = [
    "block1_conv1",
    "block2_conv1",
    "block3_conv1",
    "block4_conv1",
    "block5_conv1",
]
content_layer_name = "block5_conv2"
total_variation_weight = 1e-6
```

List of layers to use for the style loss

The layer to use for the content loss

Contribution weight of the total variation loss

```
style_weight = 1e-6
content_weight = 2.5e-8

def compute_loss(combination_image, base_image, style_reference_image):
    input_tensor = tf.concat(
        [base_image, style_reference_image, combination_image], axis=0)
    features = feature_extractor(input_tensor)
    loss = tf.zeros(shape=())
    layer_features = features[content_layer_name]
    base_image_features = layer_features[0, :, :, :]
    combination_features = layer_features[2, :, :, :]
    loss = loss + content_weight * content_loss(
        base_image_features, combination_features
    )
    for layer_name in style_layer_names:
        layer_features = features[layer_name]
        style_reference_features = layer_features[1, :, :, :]
        combination_features = layer_features[2, :, :, :]
        style_loss_value = style_loss(
            style_reference_features, combination_features)
        loss += (style_weight / len(style_layer_names)) * style_loss_value

    loss += total_variation_weight * total_variation_loss(combination_image)
    return loss
```

Contribution weight of the content loss

Contribution weight of the style loss

Initialize the loss to 0.

Add the content loss.

Add the style loss.

Add the total variation loss.

Finally, let's set up the gradient-descent process. In the original Gatys et al. paper, optimization is performed using the L-BFGS algorithm, but that's not available in TensorFlow, so we'll just do mini-batch gradient descent with the `SGD` optimizer instead. We'll leverage an optimizer feature you haven't seen before: a learning-rate schedule. We'll use it to gradually decrease the learning rate from a very high value (100) to a much smaller final value (about 20). That way, we'll make fast progress in the early stages of training and then proceed more cautiously as we get closer to the loss minimum.

Listing 12.23 Setting up the gradient-descent process

```
import tensorflow as tf

@tf.function
def compute_loss_and_grads(
    combination_image, base_image, style_reference_image):
    with tf.GradientTape() as tape:
        loss = compute_loss(
            combination_image, base_image, style_reference_image)
    grads = tape.gradient(loss, combination_image)
    return loss, grads

optimizer = keras.optimizers.SGD(
    keras.optimizers.schedules.ExponentialDecay(
        initial_learning_rate=100.0, decay_steps=100, decay_rate=0.96
    )
)
```

We make the training step fast by compiling it as a tf.function.

We'll start with a learning rate of 100 and decrease it by 4% every 100 steps.

```
base_image = preprocess_image(base_image_path)
style_reference_image = preprocess_image(style_reference_image_path)
combination_image = tf.Variable(preprocess_image(base_image_path))

iterations = 4000
for i in range(1, iterations + 1):
    loss, grads = compute_loss_and_grads(
        combination_image, base_image, style_reference_image
    )
    optimizer.apply_gradients([(grads, combination_image)])
    if i % 100 == 0:
        print(f"Iteration {i}: loss={loss:.2f}")
        img = deprocess_image(combination_image.numpy())
        fname = f"combination_image_at_iteration_{i}.png"
        keras.utils.save_img(fname, img)
```

Use a Variable to store the combination image since we'll be updating it during training.

Update the combination image in a direction that reduces the style transfer loss.

Save the combination image at regular intervals.

Figure 12.12 shows what you get. Keep in mind that what this technique achieves is merely a form of image retexturing, or texture transfer. It works best with style-reference images that are strongly textured and highly self-similar, and with content targets that don't require high levels of detail in order to be recognizable. It typically can't achieve fairly abstract feats such as transferring the style of one portrait to another. The algorithm is closer to classical signal processing than to AI, so don't expect it to work like magic!

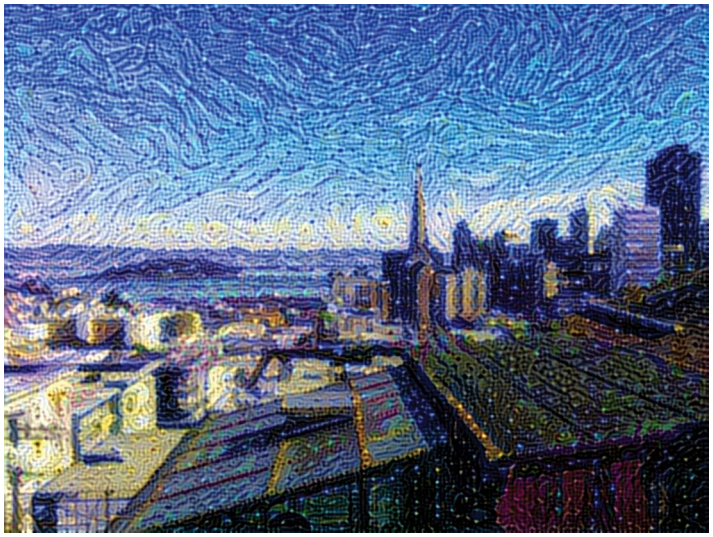

Figure 12.12 Style transfer result

Additionally, note that this style-transfer algorithm is slow to run. But the transformation operated by the setup is simple enough that it can be learned by a small, fast

feedforward convnet as well—as long as you have appropriate training data available. Fast style transfer can thus be achieved by first spending a lot of compute cycles to generate input-output training examples for a fixed style-reference image, using the method outlined here, and then training a simple convnet to learn this style-specific transformation. Once that's done, stylizing a given image is instantaneous: it's just a forward pass of this small convnet.

12.3.4 Wrapping up

- Style transfer consists of creating a new image that preserves the contents of a target image while also capturing the style of a reference image.
- Content can be captured by the high-level activations of a convnet.
- Style can be captured by the internal correlations of the activations of different layers of a convnet.
- Hence, deep learning allows style transfer to be formulated as an optimization process using a loss defined with a pretrained convnet.
- Starting from this basic idea, many variants and refinements are possible.

12.4 Generating images with variational autoencoders

The most popular and successful application of creative AI today is image generation: learning latent visual spaces and sampling from them to create entirely new pictures interpolated from real ones—pictures of imaginary people, imaginary places, imaginary cats and dogs, and so on.

In this section and the next, we'll review some high-level concepts pertaining to image generation, alongside implementation details relative to the two main techniques in this domain: *variational autoencoders* (VAEs) and *generative adversarial networks* (GANs). Note that the techniques I'll present here aren't specific to images—you could develop latent spaces of sound, music, or even text, using GANs and VAEs—but in practice, the most interesting results have been obtained with pictures, and that's what we'll focus on here.

12.4.1 Sampling from latent spaces of images

The key idea of image generation is to develop a low-dimensional *latent space* of representations (which, like everything else in deep learning, is a vector space), where any point can be mapped to a "valid" image: an image that looks like the real thing. The module capable of realizing this mapping, taking as input a latent point and outputting an image (a grid of pixels), is called a *generator* (in the case of GANs) or a *decoder* (in the case of VAEs). Once such a latent space has been learned, you can sample points from it, and, by mapping them back to image space, generate images that have never been seen before (see figure 12.13). These new images are the in-betweens of the training images.

GANs and VAEs are two different strategies for learning such latent spaces of image representations, each with its own characteristics. VAEs are great for learning

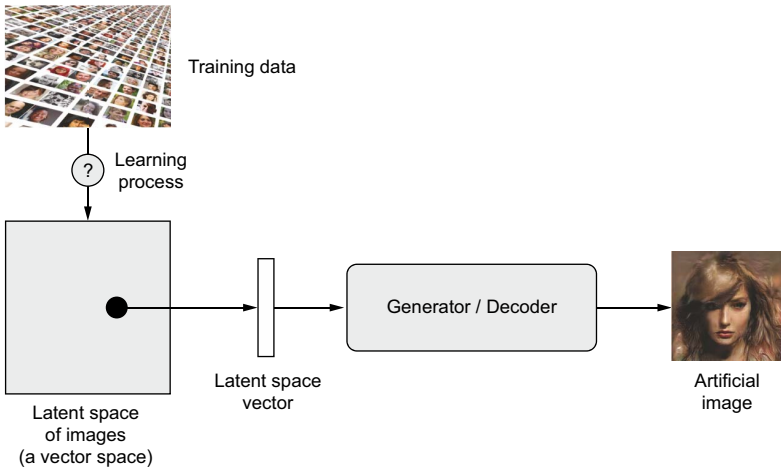

Figure 12.13 Learning a latent vector space of images and using it to sample new images

latent spaces that are well structured, where specific directions encode a meaningful axis of variation in the data (see figure 12.14). GANs generate images that can potentially be highly realistic, but the latent space they come from may not have as much structure and continuity.

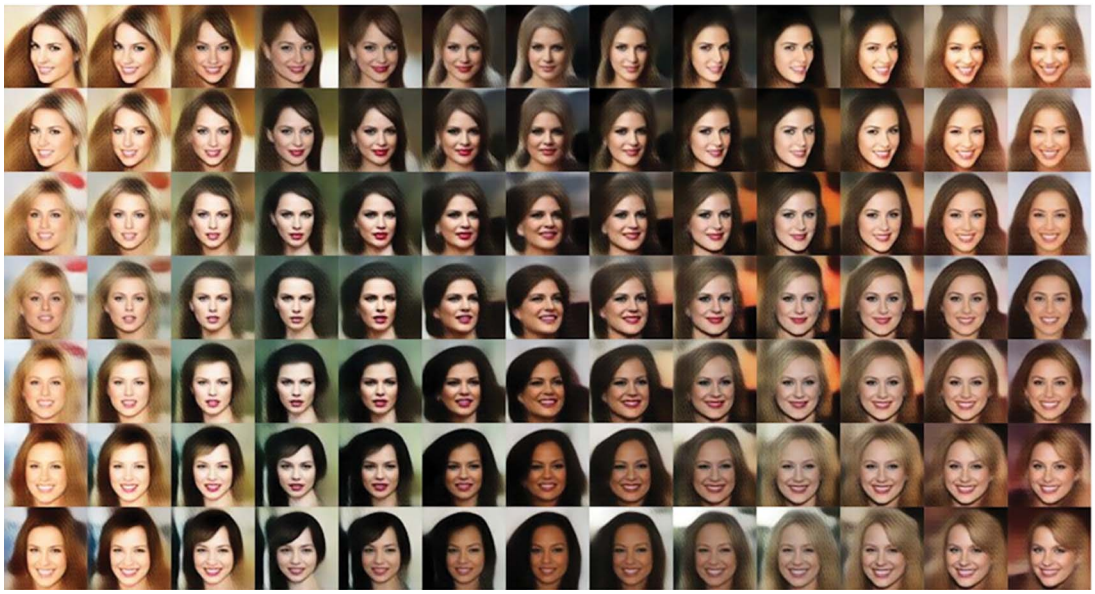

Figure 12.14 A continuous space of faces generated by Tom White using VAEs

12.4.2 Concept vectors for image editing

We already hinted at the idea of a *concept vector* when we covered word embeddings in chapter 11. The idea is still the same: given a latent space of representations, or an embedding space, certain directions in the space may encode interesting axes of variation in the original data. In a latent space of images of faces, for instance, there may be a *smile vector*, such that if latent point `z` is the embedded representation of a certain face, then latent point `z + s` is the embedded representation of the same face, smiling. Once you've identified such a vector, it then becomes possible to edit images by projecting them into the latent space, moving their representation in a meaningful way, and then decoding them back to image space. There are concept vectors for essentially any independent dimension of variation in image space—in the case of faces, you may discover vectors for adding sunglasses to a face, removing glasses, turning a male face into a female face, and so on. Figure 12.15 is an example of a smile vector, a concept vector discovered by Tom White, from the Victoria University School of Design in New Zealand, using VAEs trained on a dataset of faces of celebrities (the CelebA dataset).

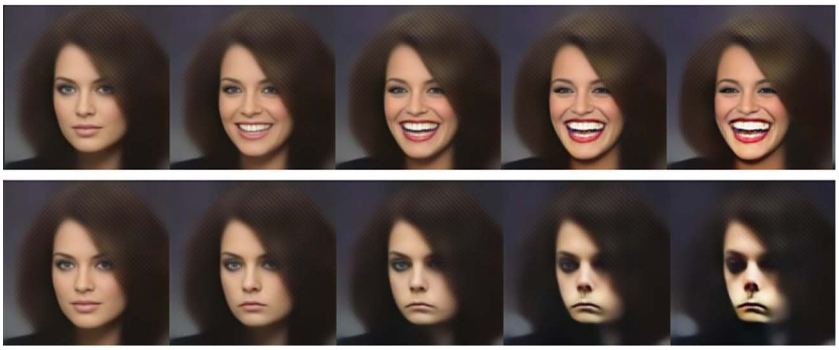

Figure 12.15 The smile vector

12.4.3 Variational autoencoders

Variational autoencoders, simultaneously discovered by Kingma and Welling in December 2013[5] and Rezende, Mohamed, and Wierstra in January 2014,[6] are a kind of generative model that's especially appropriate for the task of image editing via concept vectors. They're a modern take on autoencoders (a type of network that aims to encode an input to a low-dimensional latent space and then decode it back) that mixes ideas from deep learning with Bayesian inference.

---

[5] Diederik P. Kingma and Max Welling, "Auto-Encoding Variational Bayes," arXiv (2013), https://arxiv.org/abs/1312.6114.

[6] Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra, "Stochastic Backpropagation and Approximate Inference in Deep Generative Models," arXiv (2014), https://arxiv.org/abs/1401.4082.

A classical image autoencoder takes an image, maps it to a latent vector space via an encoder module, and then decodes it back to an output with the same dimensions as the original image, via a decoder module (see figure 12.16). It's then trained by using as target data the *same images* as the input images, meaning the autoencoder learns to reconstruct the original inputs. By imposing various constraints on the code (the output of the encoder), you can get the autoencoder to learn more- or less-interesting latent representations of the data. Most commonly, you'll constrain the code to be low-dimensional and sparse (mostly zeros), in which case the encoder acts as a way to compress the input data into fewer bits of information.

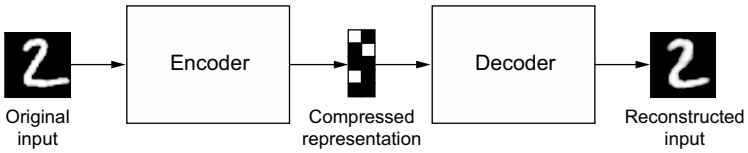

Figure 12.16 An autoencoder mapping an input *x* to a compressed representation and then decoding it back as *x'*

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

A VAE, instead of compressing its input image into a fixed code in the latent space, turns the image into the parameters of a statistical distribution: a mean and a variance. Essentially, this means we're assuming the input image has been generated by a statistical process, and that the randomness of this process should be taken into account during encoding and decoding. The VAE then uses the mean and variance parameters to randomly sample one element of the distribution, and decodes that element back to the original input (see figure 12.17). The stochasticity of this process improves robustness and forces the latent space to encode meaningful representations everywhere: every point sampled in the latent space is decoded to a valid output.

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

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

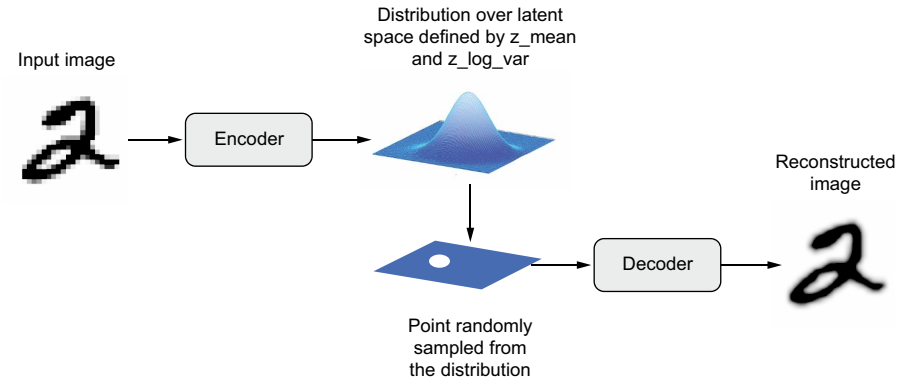

Figure 12.17 A VAE maps an image to two vectors, `z_mean` and `z_log_sigma`, which define a probability distribution over the latent space, used to sample a latent point to decode.

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

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

```
z_mean, z_log_variance = encoder(input_img)
z = z_mean + exp(z_log_variance) * epsilon
reconstructed_img = decoder(z)
model = Model(input_img, reconstructed_img)
```

Encodes the input into mean and variance parameters

Draws a latent point using a small random epsilon

Decodes z back to an image

Instantiates the autoencoder model, which maps an input image to its reconstruction

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

Now let's see what implementing a VAE looks like in practice!

12.4.4 Implementing a VAE with Keras

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

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

The following listing shows the encoder network we'll use, mapping images to the parameters of a probability distribution over the latent space. It's a simple convnet that maps the input image `x` to two vectors, `z_mean` and `z_log_var`. One important detail is that we use strides for downsampling feature maps instead of max pooling. The last time we did this was in the image segmentation example in chapter 9. Recall that, in general, strides are preferable to max pooling for any model that cares about *information location*—that is to say, *where* stuff is in the image—and this one does, since it will have to produce an image encoding that can be used to reconstruct a valid image.

Listing 12.24 VAE encoder network

```
from tensorflow import keras
from tensorflow.keras import layers

latent_dim = 2

encoder_inputs = keras.Input(shape=(28, 28, 1))
x = layers.Conv2D(
    32, 3, activation="relu", strides=2, padding="same")(encoder_inputs)
x = layers.Conv2D(64, 3, activation="relu", strides=2, padding="same")(x)
x = layers.Flatten()(x)
x = layers.Dense(16, activation="relu")(x)
z_mean = layers.Dense(latent_dim, name="z_mean")(x)
z_log_var = layers.Dense(latent_dim, name="z_log_var")(x)
encoder = keras.Model(encoder_inputs, [z_mean, z_log_var], name="encoder")
```

Dimensionality of the latent space: a 2D plane

The input image ends up being encoded into these two parameters.

Its summary looks like this:

```
>>> encoder.summary()
Model: "encoder"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_1 (InputLayer)            [(None, 28, 28, 1)]  0
__________________________________________________________________________________________________
conv2d (Conv2D)                 (None, 14, 14, 32)   320         input_1[0][0]
__________________________________________________________________________________________________
conv2d_1 (Conv2D)               (None, 7, 7, 64)     18496       conv2d[0][0]
__________________________________________________________________________________________________
```

```
flatten (Flatten)            (None, 3136)        0           conv2d_1[0][0]
__________________________________________________________________________________________________
dense (Dense)                (None, 16)          50192       flatten[0][0]
__________________________________________________________________________________________________
z_mean (Dense)               (None, 2)           34          dense[0][0]
__________________________________________________________________________________________________
z_log_var (Dense)            (None, 2)           34          dense[0][0]
==================================================================================================
Total params: 69,076
Trainable params: 69,076
Non-trainable params: 0
__________________________________________________________________________________________________
```

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

Listing 12.25 Latent-space-sampling layer

```
import tensorflow as tf

class Sampler(layers.Layer):
    def call(self, z_mean, z_log_var):
        batch_size = tf.shape(z_mean)[0]
        z_size = tf.shape(z_mean)[1]
        epsilon = tf.random.normal(shape=(batch_size, z_size))
        return z_mean + tf.exp(0.5 * z_log_var) * epsilon
```

Draw a batch of random normal vectors.

Apply the VAE sampling formula.

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

Listing 12.26 VAE decoder network, mapping latent space points to images

Input where we'll feed z

Produce the same number of coefficients that we had at the level of the Flatten layer in the encoder.

Revert the Flatten layer of the encoder.

Revert the Conv2D layers of the encoder.

```
latent_inputs = keras.Input(shape=(latent_dim,))
x = layers.Dense(7 * 7 * 64, activation="relu")(latent_inputs)
x = layers.Reshape((7, 7, 64))(x)
x = layers.Conv2DTranspose(64, 3, activation="relu", strides=2, padding="same")(x)
x = layers.Conv2DTranspose(32, 3, activation="relu", strides=2, padding="same")(x)
decoder_outputs = layers.Conv2D(1, 3, activation="sigmoid", padding="same")(x)
decoder = keras.Model(latent_inputs, decoder_outputs, name="decoder")
```

The output ends up with shape (28, 28, 1).

Its summary looks like this:

```
>>> decoder.summary()
Model: "decoder"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 2)]               0
_________________________________________________________________
dense_1 (Dense)              (None, 3136)              9408
```

```
_________________________________________________________________
reshape (Reshape)            (None, 7, 7, 64)          0
_________________________________________________________________
conv2d_transpose (Conv2DTran (None, 14, 14, 64)        36928
_________________________________________________________________
conv2d_transpose_1 (Conv2DTr (None, 28, 28, 32)        18464
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 28, 28, 1)         289
=================================================================
Total params: 65,089
Trainable params: 65,089
Non-trainable params: 0
_________________________________________________________________
```

Now let's create the VAE model itself. This is your first example of a model that isn't doing supervised learning (an autoencoder is an example of *self-supervised* learning, because it uses its inputs as targets). Whenever you depart from classic supervised learning, it's common to subclass the `Model` class and implement a custom `train_step()` to specify the new training logic, a workflow you learned about in chapter 7. That's what we'll do here.

Listing 12.27 VAE model with custom `train_step()`

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

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

    def train_step(self, data):
        with tf.GradientTape() as tape:
            z_mean, z_log_var = self.encoder(data)
            z = self.sampler(z_mean, z_log_var)
            reconstruction = decoder(z)
            reconstruction_loss = tf.reduce_mean(
                tf.reduce_sum(
                    keras.losses.binary_crossentropy(data, reconstruction),
                    axis=(1, 2)
                )
            )
            kl_loss = -0.5 * (1 + z_log_var - tf.square(z_mean) -
                tf.exp(z_log_var))
            total_loss = reconstruction_loss + tf.reduce_mean(kl_loss)
```

We use these metrics to keep track of the loss averages over each epoch.

We list the metrics in the metrics property to enable the model to reset them after each epoch (or between multiple calls to fit()/evaluate()).

We sum the reconstruction loss over the spatial dimensions (axes 1 and 2) and take its mean over the batch dimension.

Add the regularization term (Kullback–Leibler divergence).

```
        grads = tape.gradient(total_loss, self.trainable_weights)
        self.optimizer.apply_gradients(zip(grads, self.trainable_weights))
        self.total_loss_tracker.update_state(total_loss)
        self.reconstruction_loss_tracker.update_state(reconstruction_loss)
        self.kl_loss_tracker.update_state(kl_loss)
        return {
            "total_loss": self.total_loss_tracker.result(),
            "reconstruction_loss": self.reconstruction_loss_tracker.result(),
            "kl_loss": self.kl_loss_tracker.result(),
        }
```

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

Listing 12.28 Training the VAE

```
import numpy as np

(x_train, _), (x_test, _) = keras.datasets.mnist.load_data()
mnist_digits = np.concatenate([x_train, x_test], axis=0)
mnist_digits = np.expand_dims(mnist_digits, -1).astype("float32") / 255

vae = VAE(encoder, decoder)
vae.compile(optimizer=keras.optimizers.Adam(), run_eagerly=True)
vae.fit(mnist_digits, epochs=30, batch_size=128)
```

We train on all MNIST digits, so we concatenate the training and test samples.

Note that we don't pass targets in fit(), since train_step() doesn't expect any.

Note that we don't pass a loss argument in compile(), since the loss is already part of the train_step().

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

Listing 12.29 Sampling a grid of images from the 2D latent space

```
import matplotlib.pyplot as plt

n = 30
digit_size = 28
figure = np.zeros((digit_size * n, digit_size * n))

grid_x = np.linspace(-1, 1, n)
grid_y = np.linspace(-1, 1, n)[::-1]

for i, yi in enumerate(grid_y):
    for j, xi in enumerate(grid_x):
        z_sample = np.array([[xi, yi]])
        x_decoded = vae.decoder.predict(z_sample)
        digit = x_decoded[0].reshape(digit_size, digit_size)
        figure[
            i * digit_size : (i + 1) * digit_size,
```

We'll display a grid of 30 × 30 digits (900 digits total).

Sample points linearly on a 2D grid.

Iterate over grid locations.

For each location, sample a digit and add it to our figure.

```
            j * digit_size : (j + 1) * digit_size,
        ] = digit

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

The grid of sampled digits (see figure 12.18) shows a completely continuous distribution of the different digit classes, with one digit morphing into another as you follow a path through latent space. Specific directions in this space have a meaning: for example, there are directions for "five-ness," "one-ness," and so on.

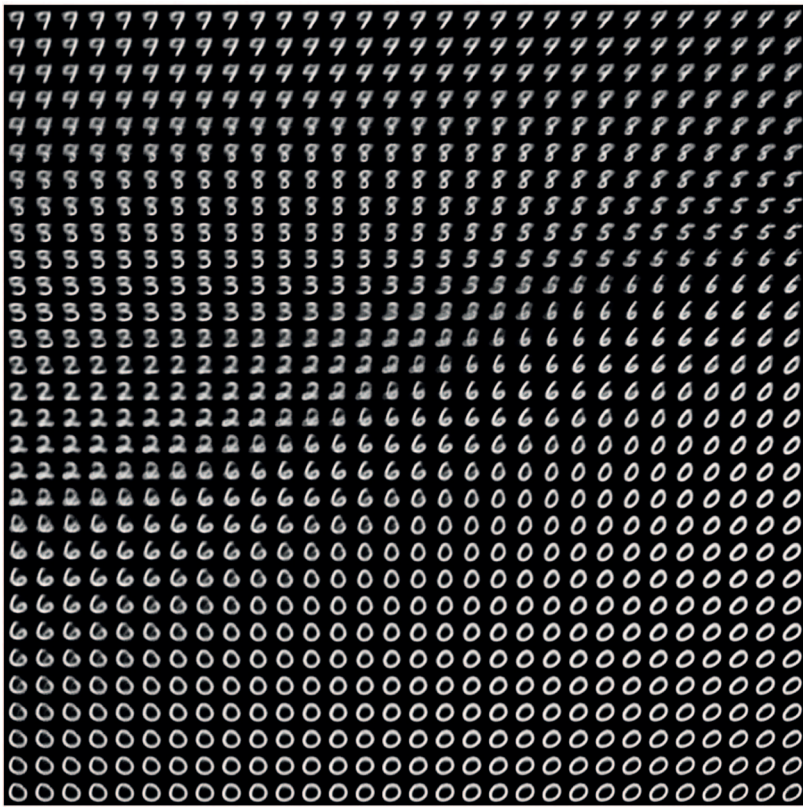

Figure 12.18 Grid of digits decoded from the latent space

In the next section, we'll cover in detail the other major tool for generating artificial images: generative adversarial networks (GANs).

12.4.5 Wrapping up

- Image generation with deep learning is done by learning latent spaces that capture statistical information about a dataset of images. By sampling and decoding points from the latent space, you can generate never-before-seen images. There are two major tools to do this: VAEs and GANs.
- VAEs result in highly structured, continuous latent representations. For this reason, they work well for doing all sorts of image editing in latent space: face swapping, turning a frowning face into a smiling face, and so on. They also work nicely for doing latent-space-based animations, such as animating a walk along a cross section of the latent space or showing a starting image slowly morphing into different images in a continuous way.
- GANs enable the generation of realistic single-frame images but may not induce latent spaces with solid structure and high continuity.

Most successful practical applications I have seen with images rely on VAEs, but GANs have enjoyed enduring popularity in the world of academic research. You'll find out how they work and how to implement one in the next section.

12.5 Introduction to generative adversarial networks

Generative adversarial networks (GANs), introduced in 2014 by Goodfellow et al.,[7] are an alternative to VAEs for learning latent spaces of images. They enable the generation of fairly realistic synthetic images by forcing the generated images to be statistically almost indistinguishable from real ones.

An intuitive way to understand GANs is to imagine a forger trying to create a fake Picasso painting. At first, the forger is pretty bad at the task. He mixes some of his fakes with authentic Picassos and shows them all to an art dealer. The art dealer makes an authenticity assessment for each painting and gives the forger feedback about what makes a Picasso look like a Picasso. The forger goes back to his studio to prepare some new fakes. As time goes on, the forger becomes increasingly competent at imitating the style of Picasso, and the art dealer becomes increasingly expert at spotting fakes. In the end, they have on their hands some excellent fake Picassos.

That's what a GAN is: a forger network and an expert network, each being trained to best the other. As such, a GAN is made of two parts:

- *Generator network*—Takes as input a random vector (a random point in the latent space), and decodes it into a synthetic image
- *Discriminator network (or adversary)*—Takes as input an image (real or synthetic), and predicts whether the image came from the training set or was created by the generator network

[7] Ian Goodfellow et al., "Generative Adversarial Networks," arXiv (2014), https://arxiv.org/abs/1406.2661.

The generator network is trained to be able to fool the discriminator network, and thus it evolves toward generating increasingly realistic images as training goes on: artificial images that look indistinguishable from real ones, to the extent that it's impossible for the discriminator network to tell the two apart (see figure 12.19). Meanwhile, the discriminator is constantly adapting to the gradually improving capabilities of the generator, setting a high bar of realism for the generated images. Once training is over, the generator is capable of turning any point in its input space into a believable image. Unlike VAEs, this latent space has fewer explicit guarantees of meaningful structure; in particular, it isn't continuous.

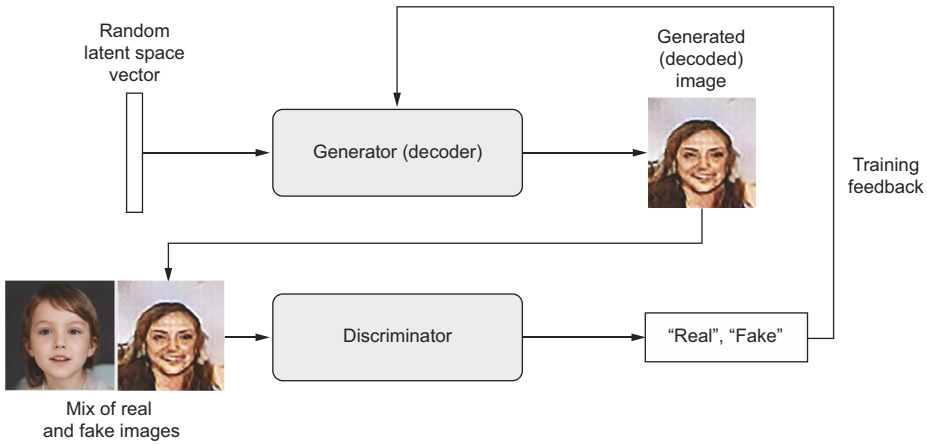

Figure 12.19 A generator transforms random latent vectors into images, and a discriminator seeks to tell real images from generated ones. The generator is trained to fool the discriminator.

Remarkably, a GAN is a system where the optimization minimum isn't fixed, unlike in any other training setup you've encountered in this book. Normally, gradient descent consists of rolling down hills in a static loss landscape. But with a GAN, every step taken down the hill changes the entire landscape a little. It's a dynamic system where the optimization process is seeking not a minimum, but an equilibrium between two forces. For this reason, GANs are notoriously difficult to train—getting a GAN to work requires lots of careful tuning of the model architecture and training parameters.

12.5.1 A schematic GAN implementation

In this section, we'll explain how to implement a GAN in Keras in its barest form. GANs are advanced, so diving deeply into the technical details of architectures like that of the StyleGAN2 that generated the images in figure 12.20 would be out of scope for this book. The specific implementation we'll use in this demonstration is a *deep convolutional GAN* (DCGAN): a very basic GAN where the generator and discriminator are deep convnets.

Figure 12.20 Latent space dwellers. Images generated by https://thispersondoesnotexist.com using a StyleGAN2 model. (Image credit: Phillip Wang is the website author. The model used is the StyleGAN2 model from Karras et al., https://arxiv.org/abs/1912.04958.)

We'll train our GAN on images from the Large-scale CelebFaces Attributes dataset (known as CelebA), a dataset of 200,000 faces of celebrities (http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html) To speed up training, we'll resize the images to 64 × 64, so we'll be learning to generate 64 × 64 images of human faces.

Schematically, the GAN looks like this:

- A `generator` network maps vectors of shape `(latent_dim,)` to images of shape `(64, 64, 3)`.
- A `discriminator` network maps images of shape `(64, 64, 3)` to a binary score estimating the probability that the image is real.
- A `gan` network chains the generator and the discriminator together: `gan(x) = discriminator(generator(x))`. Thus, this `gan` network maps latent space vectors to the discriminator's assessment of the realism of these latent vectors as decoded by the generator.
- We train the discriminator using examples of real and fake images along with "real"/"fake" labels, just as we train any regular image-classification model.
- To train the generator, we use the gradients of the generator's weights with regard to the loss of the `gan` model. This means that at every step, we move the weights of the generator in a direction that makes the discriminator more likely to classify as "real" the images decoded by the generator. In other words, we train the generator to fool the discriminator.

12.5.2 A bag of tricks

The process of training GANs and tuning GAN implementations is notoriously difficult. There are a number of known tricks you should keep in mind. Like most things in deep learning, it's more alchemy than science: these tricks are heuristics, not theory-backed guidelines. They're supported by a level of intuitive understanding of the phenomenon at hand, and they're known to work well empirically, although not necessarily in every context.

Here are a few of the tricks used in the implementation of the GAN generator and discriminator in this section. It isn't an exhaustive list of GAN-related tips; you'll find many more across the GAN literature:

- We use strides instead of pooling for downsampling feature maps in the discriminator, just like we did in our VAE encoder.
- We sample points from the latent space using a *normal distribution* (Gaussian distribution), not a uniform distribution.
- Stochasticity is good for inducing robustness. Because GAN training results in a dynamic equilibrium, GANs are likely to get stuck in all sorts of ways. Introducing randomness during training helps prevent this. We introduce randomness by adding random noise to the labels for the discriminator.
- Sparse gradients can hinder GAN training. In deep learning, sparsity is often a desirable property, but not in GANs. Two things can induce gradient sparsity: max pooling operations and `relu` activations. Instead of max pooling, we recommend using strided convolutions for downsampling, and we recommend using a `LeakyReLU` layer instead of a `relu` activation. It's similar to `relu`, but it relaxes sparsity constraints by allowing small negative activation values.
- In generated images, it's common to see checkerboard artifacts caused by unequal coverage of the pixel space in the generator (see figure 12.21). To fix this, we use a kernel size that's divisible by the stride size whenever we use a strided `Conv2DTranspose` or `Conv2D` in both the generator and the discriminator.

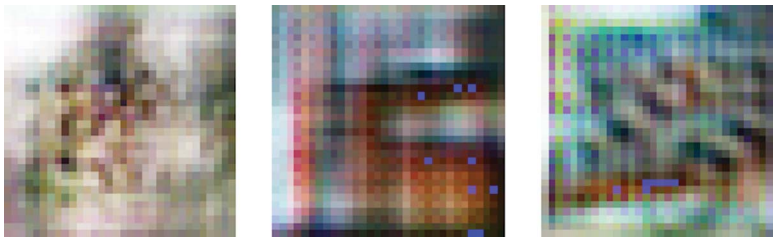

Figure 12.21 Checkerboard artifacts caused by mismatching strides and kernel sizes, resulting in unequal pixel-space coverage: one of the many gotchas of GANs

12.5.3 Getting our hands on the CelebA dataset

You can download the dataset manually from the website: http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html. If you're using Colab, you can run the following to download the data from Google Drive and uncompress it.

Listing 12.30 Getting the CelebA data

```
!mkdir celeba_gan
!gdown --id 1O7m1010EJjLE5QxLZiM9Fpjs7Oj6e684 -O celeba_gan/data.zip
!unzip -qq celeba_gan/data.zip -d celeba_gan
```

Create a working directory.

Download the compressed data using gdown (available by default in Colab; install it otherwise).

Uncompress the data.

Once you've got the uncompressed images in a directory, you can use `image_dataset_from_directory` to turn it into a dataset. Since we just need the images—there are no labels—we'll specify `label_mode=None`.

Listing 12.31 Creating a dataset from a directory of images

```
from tensorflow import keras
dataset = keras.utils_dataset_from_directory(
    "celeba_gan",
    label_mode=None,
    image_size=(64, 64),
    batch_size=32,
    smart_resize=True)
```

Only the images will be returned—no labels.

We will resize the images to 64 × 64 by using a smart combination of cropping and resizing to preserve aspect ratio. We don't want face proportions to get distorted!

Finally, let's rescale the images to the `[0-1]` range.

Listing 12.32 Rescaling the images

```
dataset = dataset.map(lambda x: x / 255.)
```

You can use the following code to display a sample image.

Listing 12.33 Displaying the first image

```
import matplotlib.pyplot as plt
for x in dataset:
    plt.axis("off")
    plt.imshow((x.numpy() * 255).astype("int32")[0])
    break
```

12.5.4 The discriminator

First, we'll develop a `discriminator` model that takes as input a candidate image (real or synthetic) and classifies it into one of two classes: "generated image" or "real image that comes from the training set." One of the many issues that commonly arise with GANs is that the generator gets stuck with generated images that look like noise. A possible solution is to use dropout in the discriminator, so that's what we will do here.

Listing 12.34 The GAN discriminator network

```
from tensorflow.keras import layers

discriminator = keras.Sequential(
    [
        keras.Input(shape=(64, 64, 3)),
        layers.Conv2D(64, kernel_size=4, strides=2, padding="same"),
        layers.LeakyReLU(alpha=0.2),
        layers.Conv2D(128, kernel_size=4, strides=2, padding="same"),
        layers.LeakyReLU(alpha=0.2),
        layers.Conv2D(128, kernel_size=4, strides=2, padding="same"),
        layers.LeakyReLU(alpha=0.2),
        layers.Flatten(),
        layers.Dropout(0.2),
        layers.Dense(1, activation="sigmoid"),
    ],
    name="discriminator",
)
```

One dropout layer: an important trick!

Here's the discriminator model summary:

```
>>> discriminator.summary()
Model: "discriminator"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 32, 32, 64)        3136
_________________________________________________________________
leaky_re_lu (LeakyReLU)      (None, 32, 32, 64)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 16, 16, 128)       131200
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 16, 16, 128)       0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 8, 8, 128)         262272
_________________________________________________________________
leaky_re_lu_2 (LeakyReLU)    (None, 8, 8, 128)         0
_________________________________________________________________
flatten (Flatten)            (None, 8192)              0
_________________________________________________________________
dropout (Dropout)            (None, 8192)              0
_________________________________________________________________
dense (Dense)                (None, 1)                 8193
=================================================================
Total params: 404,801
Trainable params: 404,801
Non-trainable params: 0
_________________________________________________________________
```

12.5.5 The generator

Next, let's develop a `generator` model that turns a vector (from the latent space—during training it will be sampled at random) into a candidate image.

Listing 12.35 GAN generator network

```
latent_dim = 128

generator = keras.Sequential(
    [
        keras.Input(shape=(latent_dim,)),
        layers.Dense(8 * 8 * 128),
        layers.Reshape((8, 8, 128)),
        layers.Conv2DTranspose(128, kernel_size=4, strides=2, padding="same"),
        layers.LeakyReLU(alpha=0.2),
        layers.Conv2DTranspose(256, kernel_size=4, strides=2, padding="same"),
        layers.LeakyReLU(alpha=0.2),
        layers.Conv2DTranspose(512, kernel_size=4, strides=2, padding="same"),
        layers.LeakyReLU(alpha=0.2),
        layers.Conv2D(3, kernel_size=5, padding="same", activation="sigmoid"),
    ],
    name="generator",
)
```

The latent space will be made of 128-dimensional vectors.

Produce the same number of coefficients we had at the level of the Flatten layer in the encoder.

Revert the Flatten layer of the encoder.

Revert the Conv2D layers of the encoder.

The output ends up with shape (28, 28, 1).

We use LeakyReLU as our activation.

This is the generator model summary:

```
>>> generator.summary()
Model: "generator"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 8192)              1056768
_________________________________________________________________
reshape (Reshape)            (None, 8, 8, 128)         0
_________________________________________________________________
conv2d_transpose (Conv2DTran (None, 16, 16, 128)       262272
_________________________________________________________________
leaky_re_lu_3 (LeakyReLU)    (None, 16, 16, 128)       0
_________________________________________________________________
conv2d_transpose_1 (Conv2DTr (None, 32, 32, 256)       524544
_________________________________________________________________
leaky_re_lu_4 (LeakyReLU)    (None, 32, 32, 256)       0
_________________________________________________________________
conv2d_transpose_2 (Conv2DTr (None, 64, 64, 512)       2097664
_________________________________________________________________
leaky_re_lu_5 (LeakyReLU)    (None, 64, 64, 512)       0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 64, 64, 3)         38403
=================================================================
Total params: 3,979,651
Trainable params: 3,979,651
Non-trainable params: 0
_________________________________________________________________
```

12.5.6 The adversarial network

Finally, we'll set up the GAN, which chains the generator and the discriminator. When trained, this model will move the generator in a direction that improves its ability to fool the discriminator. This model turns latent-space points into a classification decision—"fake" or "real"—and it's meant to be trained with labels that are always "these are real images." So training `gan` will update the weights of `generator` in a way that makes `discriminator` more likely to predict "real" when looking at fake images.

To recapitulate, this is what the training loop looks like schematically. For each epoch, you do the following:

1. Draw random points in the latent space (random noise).
2. Generate images with `generator` using this random noise.
3. Mix the generated images with real ones.
4. Train `discriminator` using these mixed images, with corresponding targets: either "real" (for the real images) or "fake" (for the generated images).
5. Draw new random points in the latent space.
6. Train `generator` using these random vectors, with targets that all say "these are real images." This updates the weights of the generator to move them toward getting the discriminator to predict "these are real images" for generated images: this trains the generator to fool the discriminator.

Let's implement it. Like in our VAE example, we'll use a `Model` subclass with a custom `train_step()`. Note that we'll use two optimizers (one for the generator and one for the discriminator), so we will also override `compile()` to allow for passing two optimizers.

Listing 12.36 The GAN `Model`

```
import tensorflow as tf
class GAN(keras.Model):
    def __init__(self, discriminator, generator, latent_dim):
        super().__init__()
        self.discriminator = discriminator
        self.generator = generator
        self.latent_dim = latent_dim
        self.d_loss_metric = keras.metrics.Mean(name="d_loss")
        self.g_loss_metric = keras.metrics.Mean(name="g_loss")

    def compile(self, d_optimizer, g_optimizer, loss_fn):
        super(GAN, self).compile()
        self.d_optimizer = d_optimizer
        self.g_optimizer = g_optimizer
        self.loss_fn = loss_fn

    @property
    def metrics(self):
        return [self.d_loss_metric, self.g_loss_metric]
```

Sets up metrics to track the two losses over each training epoch

```
    def train_step(self, real_images):
        batch_size = tf.shape(real_images)[0]
        random_latent_vectors = tf.random.normal(
            shape=(batch_size, self.latent_dim))
        generated_images = self.generator(random_latent_vectors)
        combined_images = tf.concat([generated_images, real_images], axis=0)
        labels = tf.concat(
            [tf.ones((batch_size, 1)), tf.zeros((batch_size, 1))],
            axis=0
        )
        labels += 0.05 * tf.random.uniform(tf.shape(labels))

        with tf.GradientTape() as tape:
            predictions = self.discriminator(combined_images)
            d_loss = self.loss_fn(labels, predictions)
        grads = tape.gradient(d_loss, self.discriminator.trainable_weights)
        self.d_optimizer.apply_gradients(
            zip(grads, self.discriminator.trainable_weights)
        )

        random_latent_vectors = tf.random.normal(
            shape=(batch_size, self.latent_dim))

        misleading_labels = tf.zeros((batch_size, 1))

        with tf.GradientTape() as tape:
            predictions = self.discriminator(
                self.generator(random_latent_vectors))
            g_loss = self.loss_fn(misleading_labels, predictions)
        grads = tape.gradient(g_loss, self.generator.trainable_weights)
        self.g_optimizer.apply_gradients(
            zip(grads, self.generator.trainable_weights))

        self.d_loss_metric.update_state(d_loss)
        self.g_loss_metric.update_state(g_loss)
        return {"d_loss": self.d_loss_metric.result(),
                "g_loss": self.g_loss_metric.result()}
```

Decodes them to fake images

Combines them with real images

Samples random points in the latent space

Assembles labels, discriminating real from fake images

Adds random noise to the labels—an important trick!

Trains the discriminator

Samples random points in the latent space

Assembles labels that say "these are all real images" (it's a lie!)

Trains the generator

Before we start training, let's also set up a callback to monitor our results: it will use the generator to create and save a number of fake images at the end of each epoch.

Listing 12.37 A callback that samples generated images during training

```
class GANMonitor(keras.callbacks.Callback):
    def __init__(self, num_img=3, latent_dim=128):
        self.num_img = num_img
        self.latent_dim = latent_dim

    def on_epoch_end(self, epoch, logs=None):
        random_latent_vectors = tf.random.normal(
            shape=(self.num_img, self.latent_dim))
        generated_images = self.model.generator(random_latent_vectors)
        generated_images *= 255
        generated_images.numpy()
```

```
        for i in range(self.num_img):
            img = keras.utils.array_to_img(generated_images[i])
            img.save(f"generated_img_{epoch:03d}_{i}.png")
```

Finally, we can start training.

Listing 12.38 Compiling and training the GAN

```
epochs = 100                                    ◁ You'll start getting interesting results after epoch 20.

gan = GAN(discriminator=discriminator, generator=generator,
          latent_dim=latent_dim)
gan.compile(
    d_optimizer=keras.optimizers.Adam(learning_rate=0.0001),
    g_optimizer=keras.optimizers.Adam(learning_rate=0.0001),
    loss_fn=keras.losses.BinaryCrossentropy(),
)

gan.fit(
    dataset, epochs=epochs,
    callbacks=[GANMonitor(num_img=10, latent_dim=latent_dim)]
)
```

When training, you may see the adversarial loss begin to increase considerably, while the discriminative loss tends to zero—the discriminator may end up dominating the generator. If that's the case, try reducing the discriminator learning rate, and increase the dropout rate of the discriminator.

Figure 12.22 shows what our GAN is capable of generating after 30 epochs of training.

Figure 12.22 Some generated images around epoch 30

12.5.7 Wrapping up

- A GAN consists of a generator network coupled with a discriminator network. The discriminator is trained to differentiate between the output of the generator and real images from a training dataset, and the generator is trained to fool

the discriminator. Remarkably, the generator never sees images from the training set directly; the information it has about the data comes from the discriminator.

- GANs are difficult to train, because training a GAN is a dynamic process rather than a simple gradient descent process with a fixed loss landscape. Getting a GAN to train correctly requires using a number of heuristic tricks, as well as extensive tuning.
- GANs can potentially produce highly realistic images. But unlike VAEs, the latent space they learn doesn't have a neat continuous structure and thus may not be suited for certain practical applications, such as image editing via latent-space concept vectors.

These few techniques cover only the basics of this fast-expanding field. There's a lot more to discover out there—generative deep learning is deserving of an entire book of its own.

Summary

- You can use a sequence-to-sequence model to generate sequence data, one step at a time. This is applicable to text generation, but also to note-by-note music generation or any other type of timeseries data.
- DeepDream works by maximizing convnet layer activations through gradient ascent in input space.
- In the style-transfer algorithm, a content image and a style image are combined together via gradient descent to produce an image with the high-level features of the content image and the local characteristics of the style image.
- VAEs and GANs are models that learn a latent space of images and can then dream up entirely new images by sampling from the latent space. *Concept vectors* in the latent space can even be used for image editing.