CHAPTER 15
Transformers for Natural Language Processing and Chatbots

In a landmark 2017 paper titled "Attention Is All You Need",[1] a team of Google researchers proposed a novel neural net architecture named the *Transformer*, which significantly improved the state of the art in neural machine translation (NMT). In short, the Transformer architecture is simply an encoder-decoder model, very much like the one we built in Chapter 14 for English-to-Spanish translation, and it can be used in exactly the same way (see Figure 15-1):

1. The source text goes in the encoder, which outputs contextualized embeddings (one per token).
2. The encoder's output is then fed to the decoder, along with the translated text so far (starting with a start-of-sequence token).
3. The decoder predicts the next token for each input token.
4. The last token output by the decoder is appended to the translation.
5. Steps 2 to 4 are repeated again and again to produce the full translation, one extra token at a time, until an end-of-sequence token is generated. During training, we already have the full translation—it's the target—so it is fed to the decoder in step 2 (starting with a start-of-sequence token), and steps 4 and 5 are not needed.

1 Ashish Vaswani et al., "Attention Is All You Need", *Proceedings of the 31st International Conference on Neural Information Processing Systems* (2017): 6000–6010.

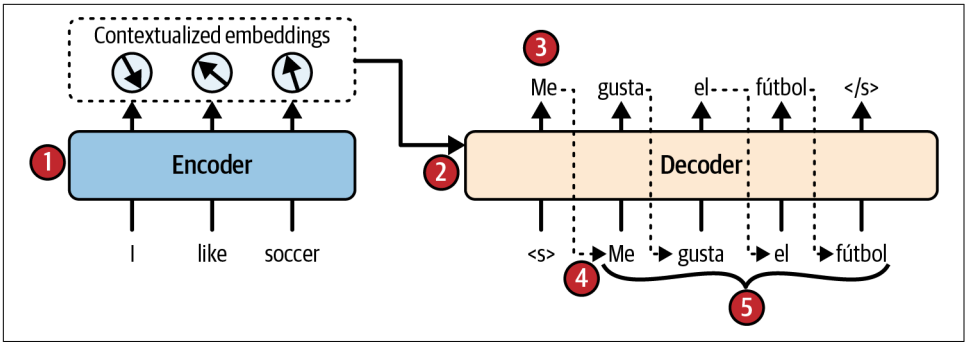

Figure 15-1. Using the Transformer model for English-to-Spanish translation

So what's new? Well, inside the black box, there are some important differences with our previous encoder-decoder. Crucially, the Transformer architecture does not contain any recurrent or convolutional layers, just regular dense layers combined with a new kind of attention mechanism called *multi-head attention* (MHA), plus a few bells and whistles.[2] Because the model is not recurrent, it doesn't suffer as much from the vanishing or exploding gradients problems as RNNs, it can be trained in fewer steps, it's easier to parallelize across multiple GPUs, and it scales surprisingly well. Moreover, thanks to multi-head attention, the model can capture long-range patterns much better than RNNs.

The Transformer architecture also turned out to be extremely versatile. It was initially designed for NMT, but researchers quickly tweaked the architecture for many other language tasks. The year 2018 was even called the "ImageNet moment for NLP". In June 2018, OpenAI released the first GPT model, based solely on the Transformer's decoder module. It was pretrained on a large corpus of text, its ability to generate text was unprecedented, it could auto-complete sentences, invent stories, and even answer some questions. GPT could also be fine-tuned to perform a wide range of language tasks. Just a few months later, Google released the BERT model, based solely on the Transformer's encoder module. It was excellent at a variety of *natural language understanding* (NLU) tasks, such as text classification, text embedding, multiple choice question answering, or finding the answer to a question within some text.

Surprisingly, transformers also turned out to be great at computer vision, audio processing (e.g., speech-to-text), robotics (using inputs from sensors and sending the outputs to actuators), and more. For example, if you split an image into little chunks and feed them to a transformer (instead of token embeddings), you get a *vision*

2 When applying a linear layer to a sequence, all tokens are treated independently, using the same parameters. This is equivalent to using a `Conv1d` layer with `kernel_size=1`. This is why you will sometimes see Transformer diagrams showing convolutional layers instead of linear layers.

transformer (ViT). In fact, some transformers can even handle multiple *modalities* at once (e.g., text + image); these are called *multimodal models*.

This outstanding combination of performance, flexibility, and scalability encouraged Google, OpenAI, Facebook (Meta), Microsoft, Anthropic, and many other organizations to train larger and larger transformer models. The original Transformer model had about 65 million parameters—which was considered quite large at the time—but new transformers grew at a mind-boggling rate, reaching 1.6 trillion parameters by January 2021—that's 1.6 million million parameters! Training such a gigantic transformer model from scratch is sadly restricted to organizations with deep pockets, as it requires a large and costly infrastructure for several months: training typically costs tens of millions of dollars, and even up to hundreds of millions according to some estimates (the exact figures are generally not public). Figure 15-2 shows some of the most influential transformers released between June 2018 and April 2025.[3] Note that the vertical axis is in *billions* of parameters, and it uses a logarithmic scale.

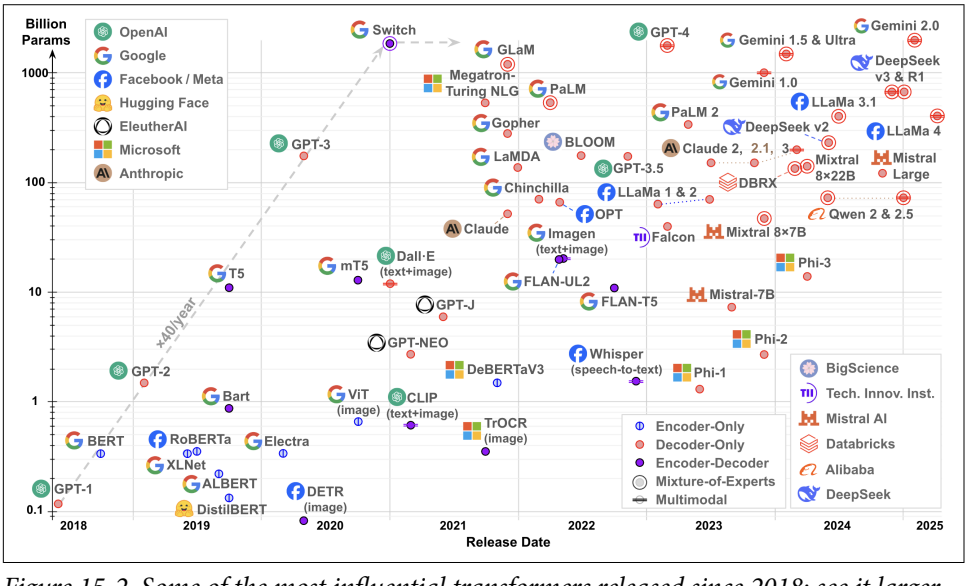

Figure 15-2. Some of the most influential transformers released since 2018; see it larger online

3 The number of parameters is not public for some models (e.g., Gemini models), so I used some rough estimates. Also, many models have smaller variants, not shown here. Lastly, several other organizations released influential models, such as the Allen Institute for AI (Ai2), Amazon, Baidu, Beijing Academy of AI, Cohere, Huawei, LAION, LMSYS, Nvidia, Stanford University, Talent International Institute (TII), Tsinghua University, xAI, Zhipu AI, and others.

Then, in November 2022, OpenAI released ChatGPT, an amazing *conversational AI*—or *chatbot*—that took the world by storm: it reached one million users in just five days, and over one hundred million monthly active users after just two months! Under the hood, it used GPT-3.5-turbo, a variant of GPT-3.5 which was fine-tuned to be conversational, helpful, and safe. Others soon followed: Perplexity AI, Google's Gemini (initially called Bard), Anthropic's Claude, Mistral AI, DeepSeek, and more.

Before ChatGPT was released, Google had actually developed a powerful chatbot named LaMDA, but it wasn't made public, likely for fear of reputational and legal risks, as the model was not deemed safe enough. This allowed OpenAI to become the first company to train a reasonably safe and helpful model and to package it as a useful chatbot product.

So how can you use these models and chatbots? Well, many of them are proprietary (e.g., OpenAI's GPT-3.5, GPT-4 and GPT-5 models, Anthropic's Claude models, and Google's Gemini models), and they can only be used via a web UI, an app, or an API: you must create an account, choose an offer (or use the free tier), and for the API you must get an access token and use it to query the API programmatically. However, many other models are *open weights*, meaning they can be downloaded for free (e.g., using the Hugging Face Hub): some of these have licensing restrictions (e.g., Meta's Llama models are only free for noncommercial use), while others are truly open source (e.g., DeepSeek's R1 or Mistral AI's Mistral-7B). Some even include the training code and data (e.g., the OLMo models by Ai2).

So what are we waiting for? Let's join the transformer revolution! Here's the plan:

- We will start by opening up the original Transformer architecture and inspecting its components to fully understand how everything works.
- Then we will build and train a transformer from scratch for English-to-Spanish translation.
- After that, we will look into encoder-only models like BERT, learn how they are pretrained, and see how to use them for tasks like text classification, semantic search, and text clustering, with or without fine-tuning.
- Next, we will look into decoder-only models like GPT, and see how they are pretrained. These models are capable of generating text, which is great if you want to write a poem, but it can also be used to tackle many other tasks.
- Then we will use a decoder-only model to build our own chatbot! This involves a few steps: first, you must download a pretrained model (or train your own if you have the time and money), then you must fine-tune it to make it more conversational, helpful, and safe (or you can download an already fine-tuned model, or even use a conversational model via an API), and lastly you must deploy the

model to a chatbot system that offers a user interface, stores conversations, and can also give the model access to tools, such as searching the web or using a calculator.

- Lastly, we will take a quick look at encoder-decoder models, such as T5 and BART, which are great for tasks such as translation and summarization.

In Chapter 16, we will look at vision transformers and multimodal transformers. Chapter 17 and "State-Space Models (SSMs)" (both available at *https://homl.info*) also discuss some advanced techniques to allow Transformers to scale and process longer input sequences.

Let's start by dissecting the Transformer architecture: take out your scalpel!

Attention Is All You Need: The Original Transformer Architecture

The original 2017 Transformer architecture is represented in Figure 15-3. The left part of the figure represents the encoder, the right part represents the decoder.

As we saw earlier, the encoder's role is to gradually *transform* the inputs (e.g., sequences of English tokens) until each token's representation perfectly captures the meaning of that token in the context of the sentence: the encoder's output is a sequence of contextualized token embeddings. Apart from the embedding layer, every layer in the encoder takes as input a tensor of shape [*batch size*, *max English sequence length in the batch*, *embedding size*] and returns a tensor of the exact same shape. This means that token representations get gradually transformed, hence the name of the architecture. For example, if you feed the sentence "I like soccer" to the encoder, then the token "like" will start off with a rather vague representation, since "like" could mean different things in different contexts (e.g., "I'm like a cat" versus "I like my cat"). But after going through the encoder, the token's representation should capture the correct meaning of "like" in the given sentence (in this case, to be fond of), as well as any other information that may be required for translation (e.g., it's a verb).

The decoder's role is to take the encoder's outputs, along with the translated sentence so far, and predict the next token in the translation. For this, the decoder layers gradually transform each input token's representation into a representation that can be used to predict the following token. For example, suppose the sentence to translate is "I like soccer" and we've already called the decoder four times, producing one new token each time: first "me", then "me gusta", then "me gusta el", and finally "me gusta el fútbol". Since this translation does not end with an EoS token `"</s>"`, we must call the decoder once again. The decoder's input sequence is now `"<s>` me gusta el fútbol". As the representation of each token goes through the decoder, it gets transformed: the representation of `"<s>"` becomes a rich enough representation to

predict “me” (for simplicity, I’ll say this more concisely as: "<s>" becomes “me”), “me” becomes “gusta”, “gusta” becomes “el”, “el” becomes “fútbol”, and if everything goes well, “fútbol” becomes the EoS token "</s>". Apart from the embedding layer and the output `Linear` layer, every layer in the decoder takes as input a tensor of shape [*batch size*, *max Spanish sequence length in the batch*, *embedding size*] and returns a tensor of the exact same shape.

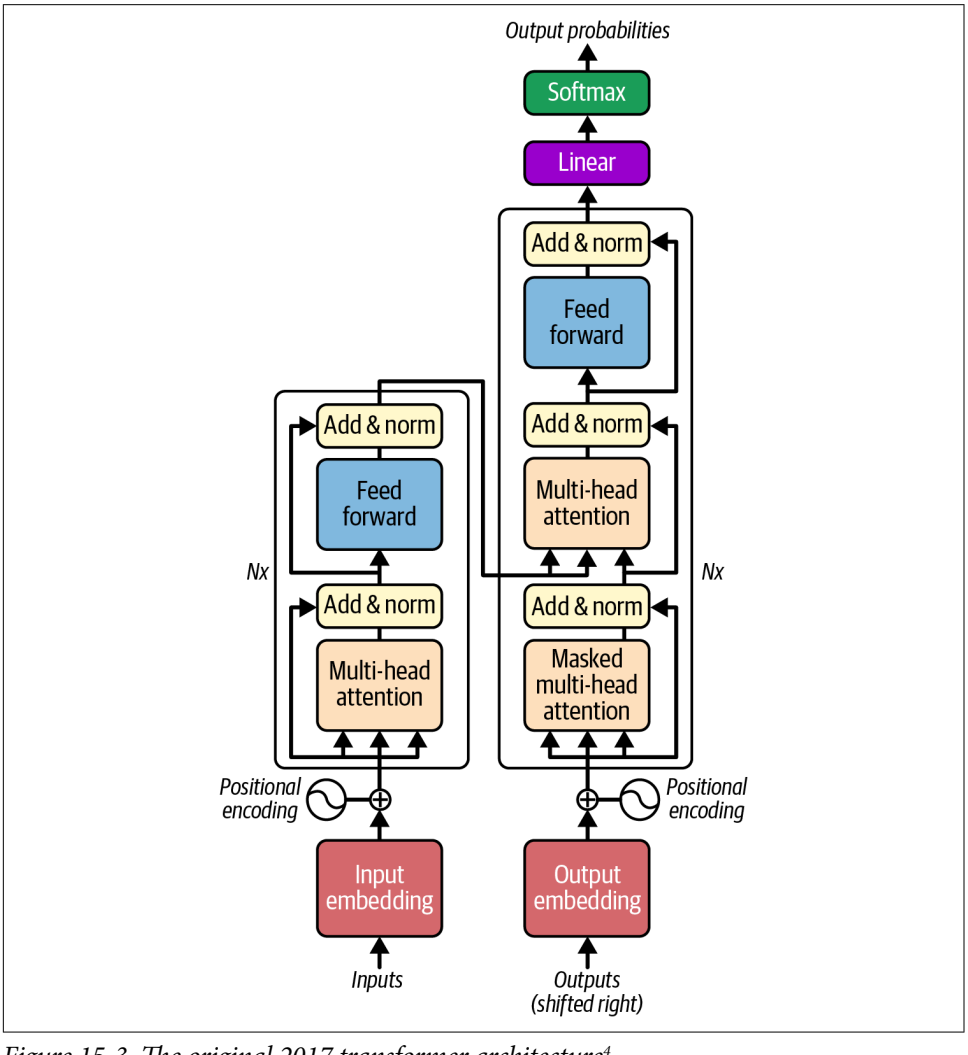

Figure 15-3. The original 2017 transformer architecture[4]

4 This is adapted from Figure 1 from “Attention Is All You Need”, with the kind permission of the authors.

After going through the decoder, each token representation goes through a final `Linear` layer which will hopefully output a high logit for the correct token and a low logit for all other tokens in the vocabulary. The decoder's output shape is [*batch size*, *max Spanish sequence length in the batch*, *vocabulary size*]. The final predicted sentence should be "me gusta el fútbol `</s>`". Note that the figure shows a softmax layer at the top, but in PyTorch we usually don't explicitly add it: instead, we let the model output logits, and we train the model using `nn.CrossEntropyLoss`, which computes the cross-entropy loss based on logits instead of estimated probabilities (as we saw in previous chapters). If you ever need estimated probabilities, you can always convert the logits to estimated probabilities using the `F.softmax()` function.

Now let's zoom in a bit further into Figure 15-3:

- First, notice that both the encoder and the decoder contain blocks that are stacked N times. In the paper, $N = 6$. Note that the final outputs of the whole encoder stack are fed to each of the decoder's N blocks.
- As you can see, you are already familiar with most components: there are two embedding layers; several skip connections, each of them followed by a layer normalization module; several feedforward modules composed of two dense layers each (the first one using the ReLU activation function, the second with no activation function); and finally, the output layer is a linear layer. Notice that all layers treat each token independently from all the others. But how can we translate a sentence by looking at the tokens completely separately? Well, we can't, so that's where the new components come in:
  - The encoder's *multi-head attention* layer updates each token representation by attending to (i.e., paying attention to) every token in the same sentence, including itself. This is called *self-attention*. That's where the vague representation of the word "like" becomes a richer and more accurate representation, capturing its precise meaning within the given sentence (e.g., the layer notices the subject "I" so it infers that "like" must be a verb). We will discuss exactly how this works shortly.
  - The decoder's *masked multi-head attention* layer does the same thing, but when it processes a token, it doesn't attend to tokens located after it: it's a causal layer. For example, when it processes the token "gusta", it only attends to the tokens `"<s>"`, "me", and "gusta", and it ignores the tokens "el" and "fútbol" (or else the model could cheat during training).
  - The decoder's upper multi-head attention layer is where the decoder pays attention to the contextualized token representations output by the encoder stack. This is called *cross*-attention, as opposed to *self*-attention. For example, the decoder will probably pay close attention to the word "soccer" when it processes the word "el" and outputs a representation of the word "fútbol".

— The *positional encodings* are dense vectors that represent the position of each token in the sentence. The n^{th} positional encoding is added to the token embedding of the n^{th} token in each sentence. This is needed because all layers in the Transformer architecture are position-agnostic, meaning they treat all positions equally (unlike recurrent or convolutional layers): when they process a token, they have no idea where that token is located in the sentence or relative to other words. But the order of words matters, so we must somehow give positional information to the Transformer. Adding positional encodings to the token representations is a good way to achieve this.

The first two arrows going into each multi-head attention layer in Figure 15-3 represent the keys and values, and the third arrow represents the queries.[5] In the self-attention layers, all three are equal to the token representations output by the previous layer, while in the cross-attention layers (i.e., the decoder's upper attention layers), the keys and values are equal to the encoder's final token representations, and the queries are equal to the token representations output by the previous decoder layer.

Now let's go through the novel components of the Transformer architecture in more detail, starting with the positional encodings.

Positional Encodings

A positional encoding is a dense vector that encodes the position of a token within a sentence: the i^{th} positional encoding is added to the token embedding of the i^{th} token in each sentence. A simple way to implement this is to use an `Embedding` layer: just add embedding #0 to the representation of token #0, add embedding #1 to the representation of token #1, and so on. Alternatively, you can use an `nn.Parameter` to store the embedding matrix (initialized using small random weights), then add its first L rows to the inputs (where L is the max input sequence length): the result is the same, but it's much faster. You can also add a bit of dropout to reduce the risk of overfitting. Here's an implementation:

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class PositionalEmbedding(nn.Module):
    def __init__(self, max_length, embed_dim, dropout=0.1):
        super().__init__()
```

5 Queries, keys, and values were introduced in Chapter 14 when we discussed dot-product attention.

```
        self.pos_embed = nn.Parameter(torch.randn(max_length, embed_dim) * 0.02)
        self.dropout = nn.Dropout(dropout)

    def forward(self, X):
        return self.dropout(X + self.pos_embed[:X.size(1)])
```

The inputs have shape [*batch size*, *sequence length*, *embedding size*], but we are adding positional encodings of shape [*sequence length*, *embedding size*]. This works thanks to the broadcasting rules: the i^{th} positional embedding is added to the i^{th} token's representation of each sentence in the batch.

The authors of the Transformer paper also proposed using fixed positional encodings rather than trainable ones. Their approach used a pretty smart scheme based on the sine and cosine functions, but it's not much used anymore, as it doesn't really perform any better than trainable positional embeddings (except perhaps on small transformers, if you're lucky). Please see this chapter's notebook for more details. Moreover, newer approaches such as *relative position bias* (RPB), *rotary positional encoding* (RoPE), and *attention with linear bias* (ALiBi) generally perform better. To learn more about all of these alternative approaches to positional encoding, see "Relative Positional Encoding".

Now let's look deeper into the heart of the Transformer model: the multi-head attention layer.

Multi-Head Attention

The multi-head attention (MHA) layer is based on *scaled dot-product attention*, a variant of dot-product attention (introduced in Chapter 14) that scales down the similarity scores by a constant factor. See Equation 15-1 for its vectorized equation.

Equation 15-1. Scaled dot-product attention

$$\text{Attention}(\mathbf{Q},\mathbf{K},\mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\intercal}{\sqrt{d_\text{k}}}\right)\mathbf{V}$$

In this equation:

- **Q** is a matrix representing a *query* (e.g., an English or Spanish sequence, depending on the attention layer). Its shape is $[L_\text{q}, d_\text{q}]$, where L_q is the length of the query and d_q is the query's dimensionality (i.e., the number of dimensions in the token representations).
- **K** is a matrix representing a key. Its shape is $[L_\text{k}, d_\text{k}]$, where L_k is the length of the key and d_k is the key's dimensionality. Note that d_k must equal d_q.

- **V** is a matrix representing a value. Its shape is $[L_v, d_v]$, where L_v is the length of the value and d_v is the value's dimensionality. Note that L_v must equal L_k.
- The shape of $\mathbf{Q}\,\mathbf{K}^\intercal$ is $[L_q, L_k]$: it contains one similarity score for each query/key pair. To prevent this matrix from being huge, the input sequences must not be too long: this is the critical *quadratic context window* problem (we will discuss various ways to alleviate this issue in Chapters 16 and 17. The softmax function is applied to each row: the output has the same shape as the input, but now each row sums up to 1. The final output has a shape of $[L_q, d_v]$. There is one row per query token, and each row represents the query result: a weighted sum of the value tokens, favoring value tokens whose corresponding key tokens are most aligned with the given query token.
- The scaling factor $1 / \sqrt{d_k}$ scales down the similarity scores to avoid saturating the softmax function, which would lead to tiny gradients. This factor was empirically shown to speed up and stabilize training.
- It is possible to mask out some key/value pairs by adding a very large negative value to the corresponding similarity scores, just before computing the softmax (in practice, we can add `-torch.inf`). The resulting weights will be equal to zero. This is useful to mask padding tokens, as well as future tokens in the masked multi-head attention layer.

PyTorch comes with the `F.scaled_dot_product_attention()` function. Its inputs are just like **Q**, **K**, and **V**, but these inputs can have extra dimensions at the start, such as the batch size and the number of heads (when used for multi-head attention). The equation is applied simultaneously across all of these extra dimensions. In other words, the function computes the results simultaneously across all sentences in the batch and across all attention heads, making it very efficient.

Now we're ready to look at the multi-head attention layer. Its architecture is shown in Figure 15-4.

As you can see, it is just a bunch of scaled dot-product attention layers, called *attention heads*, each preceded by a linear transformation of the values, keys, and queries (across all tokens). The outputs of all the attention heads are simply concatenated, and they go through a final linear transformation (again, across all tokens).

But why? What is the intuition behind this architecture? Well, consider once again the word "like" in the sentence "I like soccer". The encoder was hopefully smart enough to encode its meaning, the fact that it's a verb, and many other features that are useful for its translation, such as the fact that it is in the present tense. The token representation also includes the position, thanks to the positional encodings. In short, the token representation encodes many different characteristics of the token. If we just used a single scaled dot-product attention layer, we would only be able to query all of these characteristics in one shot.

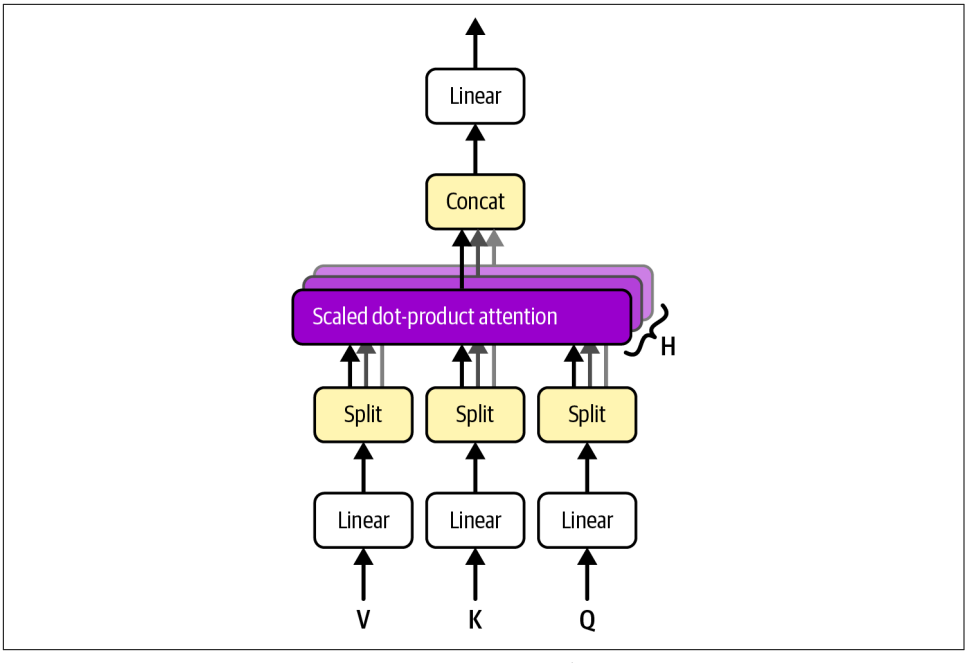

Figure 15-4. Multi-head attention layer architecture[6]

The Transformer architecture is extremely flexible, so the model has plenty of freedom during training to choose its own knowledge representation and strategies. As a result, it ends up being somewhat of a black box: understanding how transformers truly "think" is an area of active research, called *model interpretability*. For example, check out this fascinating post by Anthropic.

This is why the MHA layer splits the values, keys, and queries across multiple heads: this way, each head can focus on specific characteristics of the token. The first linear layer lets the model choose which characteristics each head should focus on. For example, the linear layer may ensure that the first head gets a projection of the "like" token's representation into a subspace where all that remains is the information that this token is a verb in the present tense. Another head may focus on the word's meaning, and so on. Then the scaled dot-product attention layers implement the actual lookup phase, and finally the results are all concatenated and run through a final linear layer that lets the model reorganize the representation as it pleases.

6 This is adapted from the righthand part of Figure 2 from "Attention Is All You Need", with the kind authorization of the authors.

To really understand the Transformer architecture, the key is to understand multi-head attention, and for this, it helps to look at a basic implementation:

```
class MultiheadAttention(nn.Module):
    def __init__(self, embed_dim, num_heads, dropout=0.1):
        super().__init__()
        self.h = num_heads
        self.d = embed_dim // num_heads
        self.q_proj = nn.Linear(embed_dim, embed_dim)
        self.k_proj = nn.Linear(embed_dim, embed_dim)
        self.v_proj = nn.Linear(embed_dim, embed_dim)
        self.out_proj = nn.Linear(embed_dim, embed_dim)
        self.dropout = nn.Dropout(dropout)

    def split_heads(self, X):
        return X.view(X.size(0), X.size(1), self.h, self.d).transpose(1, 2)

    def forward(self, query, key, value):
        q = self.split_heads(self.q_proj(query))  # (B, h, Lq, d)
        k = self.split_heads(self.k_proj(key))  # (B, h, Lk, d)
        v = self.split_heads(self.v_proj(value))  # (B, h, Lv, d) with Lv=Lk
        scores = q @ k.transpose(2, 3) / self.d**0.5  # (B, h, Lq, Lk)
        weights = scores.softmax(dim=-1)  # (B, h, Lq, Lk)
        Z = self.dropout(weights) @ v  # (B, h, Lq, d)
        Z = Z.transpose(1, 2)  # (B, Lq, h, d)
        Z = Z.reshape(Z.size(0), Z.size(1), self.h * self.d)  # (B, Lq, h × d)
        return (self.out_proj(Z), weights)  # (B, Lq, h × d)
```

Let's go through this code:

- The constructor stores the number of heads `self.h` and computes the number of dimensions per head `self.d`, then it creates the necessary modules. Note that the embedding size must be divisible by the number of heads.
- The `split_heads()` method is used in the `forward()` method. It splits its input `X` along its last dimension (one split per head), converting it from a 3D tensor of shape $[B, L, h \times d]$ to a 4D tensor of shape $[B, L, h, d]$, where B is the batch size, L is the max length of the input sequences (specifically L_k for the key and value, or L_q for the query), h is the number of heads, and d is the number of dimensions per head (i.e., $h \times d$ = embedding size). The dimensions 1 and 2 are then swapped to get a tensor of shape $[B, h, L, d]$: since the matrix multiplication operator `@` only works on the last two dimensions, it won't touch the first two dimensions B and h, so we will be able to use this operator to compute the scores across all instances in the batch and across all attention heads, all in one shot (`q @ k.transpose(2, 3)`). The same will be true when computing all the attention outputs (`weights @ v`).

- The `forward()` method starts by applying a linear transformation to the query, key, and value, and passes the result through the `split_heads()` method. The next three lines compute Equation 15-1, plus a bit of dropout on the weights. Next we swap back dimensions 1 and 2 to ensure that the dimensions h and d are next to each other again, then we reshape the tensor back to 3D: this will concatenate the outputs of all heads. We can then apply the output linear transformation and return the result, along with the weights (in case we need them later).

Don't worry if it takes some time to fully grasp this, it's not easy. Of course, you can drive a car without fully understanding how the engine works, but some of the transformer improvements described in Chapters 16 and 17 will only make sense if you understand MHA.

But wait! We're missing one important detail: masking. Indeed, as we discussed earlier, the decoder's masked self-attention layers must only consider previous tokens when trying to predict what the next token is (or else it would be cheating). Moreover, if the key contains padding tokens, we want to ignore them as well. So let's update the `forward()` method to support two additional arguments:

`attn_mask`
: A boolean mask of shape $[L_q, L_k]$ that we will use to control which key tokens each query token should ignore (`True` to ignore, `False` to attend)

`key_padding_mask`
: A boolean mask of shape $[B, L_k]$ to locate the padding tokens in each key

```
def forward(self, query, key, value, attn_mask=None, key_padding_mask=None):
    [...]  # compute the scores exactly like earlier
    if attn_mask is not None:
        scores = scores.masked_fill(attn_mask, -torch.inf)  # (B, h, Lq, Lk)
    if key_padding_mask is not None:
        mask = key_padding_mask.unsqueeze(1).unsqueeze(2)  # (B, 1, 1, Lk)
        scores = scores.masked_fill(mask, -torch.inf)  # (B, h, Lq, Lk)
    [...]  # compute the weights and the outputs exactly like earlier
```

This code replaces the scores we want to ignore with negative infinity, so the corresponding weights will be zero after the softmax operation (if we tried to zero out these weights directly, the remaining weights would not add up to 1). Note that the masks are broadcast automatically: `attn_mask` is broadcast across the whole batch and all attention heads, and `key_padding_mask` is broadcast across all heads and all query tokens.

PyTorch has a very similar `nn.MultiheadAttention` module, which is much more optimized (e.g., it can often fuse the three input projections into one). It has the same arguments, which behave in exactly the same way. It also has a few more. Here are the most important:

- The constructor has a `batch_first` argument which defaults to `False`, so the module expects the batch dimension to come after the sequence length dimension. You must set `batch_first=True` if you prefer the batch dimension to come first, like in our custom implementation.
- The `forward()` method has a `need_weights` argument that defaults to `True`. If you don't need to use the weights returned by this module, you should set this argument to `False`, as it sometimes allows for some optimizations. When `need_weights` is set to `False`, the method returns `None` instead of the weights.
- The `forward()` method also has an `is_causal` argument: if (and only if) the `attn_mask` is set and is a *causal mask*, then you can set `is_causal=True` to allow for some performance optimizations. A causal mask allows each query token to attend to all previous tokens (including itself), but doesn't allow it to attend to tokens located after it. In other words, a causal mask contains `True` above the main diagonal, and `False` everywhere else. This is the mask needed for the masked self-attention layers.

Now that we have the main ingredient, we're ready to implement the rest of the Transformer model.

Building the Rest of the Transformer

The rest of the Transformer architecture is much more straightforward. Let's start with the encoder block. The following implementation closely matches the encoder block represented on the left side of Figure 15-3, except it sprinkles a bit of dropout after the self-attention layer and after both dense layers in the feedforward module:

```
class TransformerEncoderLayer(nn.Module):
    def __init__(self, d_model, nhead, dim_feedforward=2048, dropout=0.1):
        super().__init__()
        self.self_attn = MultiheadAttention(d_model, nhead, dropout)
        self.linear1 = nn.Linear(d_model, dim_feedforward)
        self.dropout = nn.Dropout(dropout)
        self.linear2 = nn.Linear(dim_feedforward, d_model)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)

    def forward(self, src, src_mask=None, src_key_padding_mask=None):
        attn, _ = self.self_attn(src, src, src, attn_mask=src_mask,
                                 key_padding_mask=src_key_padding_mask)
        Z = self.norm1(src + self.dropout(attn))
```

```
        ff = self.dropout(self.linear2(self.dropout(self.linear1(Z).relu())))
        return self.norm2(Z + ff)
```

Notice that the feedforward block is composed of a first `Linear` layer that expands the dimensionality to 2048 (by default), followed by a nonlinearity (ReLU in this case), then a second `Linear` layer that projects the data back down to the original embedding size (also called the *model dimension*, `d_model`). This *reverse bottleneck* increases the expressive power of the nonlinearity, allowing the model to learn much richer combinations of features. This idea was explored further in the later MobileNetv2 paper, whose authors coined the term *inverted residual network*.

In the encoder, the `src_mask` argument is generally not used, since the encoder allows each token to attend to all tokens, even ones located after it. However, the user is expected to set the `key_padding_mask` appropriately.

Now here's an implementation of the decoder block. It closely matches the decoder block represented on the righthand side of Figure 15-3, with some additional dropout:

```
class TransformerDecoderLayer(nn.Module):
    [...]  # similar constructor, with 2 MHA, 3 Linear, 3 LayerNorm, 1 Dropout
    def forward(self, tgt, memory, tgt_mask=None, memory_mask=None,
                tgt_key_padding_mask=None, memory_key_padding_mask=None):
        attn1, _ = self.self_attn(tgt, tgt, tgt, attn_mask=tgt_mask,
                                  key_padding_mask=tgt_key_padding_mask)
        Z = self.norm1(tgt + self.dropout(attn1))
        attn2, _ = self.multihead_attn(Z, memory, memory, attn_mask=memory_mask,
                                       key_padding_mask=memory_key_padding_mask)
        Z = self.norm2(Z + self.dropout(attn2))
        ff = self.dropout(self.linear2(self.dropout(self.linear1(Z).relu())))
        return self.norm3(Z + ff)
```

The `memory` argument corresponds to the output of the encoder. For full flexibility, we let the user pass the appropriate masks to the `forward()` method. In general, you will need to set the padding masks appropriately (both for the memory and target), and set the `tgt_mask` to a causal mask (we will see how shortly).

PyTorch actually provides `nn.TransformerEncoderLayer` and `nn.TransformerDecoderLayer` out of the box, with the same arguments, plus a few more: most importantly `batch_first`, which you must set to `True` if the batch dimension is first, plus one `*_is_causal` argument for each attention mask, and an `activation` argument that defaults to "relu". Many state-of-the-art transformers use a more advanced activation such as GELU (introduced in Chapter 11).

PyTorch also provides three more transformer modules (writing a custom module for each of these is left as an exercise for the reader—see the notebook for a solution):

`nn.TransformerEncoder`
: Simply chains the desired number of encoder layers. Its constructor takes an encoder layer plus the desired number of layers `num_layers`, and it clones the given encoder layer `num_layers` times. The constructor also takes an optional normalization layer, which (if provided) is applied to the final output.

`nn.TransformerDecoder`
: Same, except it chains decoder layers instead of encoder layers.

`nn.Transformer`
: Creates an encoder and a decoder (both with layer norm), and chains them.

Congratulations! You now know how to build a full Transformer model from scratch. You only need to add a final `Linear` layer and use the `nn.CrossEntropyLoss` to get the full architecture shown in Figure 15-3 (as we saw in earlier chapters, the softmax layer is implicitly included in the loss). Now let's see how to use a Transformer model to translate English to Spanish.

Building an English-to-Spanish Transformer

It's time to build our NMT Transformer model. For this, we'll use our `PositionalEmbedding` module and PyTorch's `nn.Transformer` (our custom `Transformer` module works fine, but it's slower):

```
class NmtTransformer(nn.Module):
    def __init__(self, vocab_size, max_length, embed_dim=512, pad_id=0,
                 num_heads=8, num_layers=6, dropout=0.1):
        super().__init__()
        self.embed = nn.Embedding(vocab_size, embed_dim, padding_idx=pad_id)
        self.pos_embed = PositionalEmbedding(max_length, embed_dim, dropout)
        self.transformer = nn.Transformer(
            embed_dim, num_heads, num_encoder_layers=num_layers,
            num_decoder_layers=num_layers, batch_first=True)
        self.output = nn.Linear(embed_dim, vocab_size)

    def forward(self, pair):
        src_embeds = self.pos_embed(self.embed(pair.src_token_ids))
        tgt_embeds = self.pos_embed(self.embed(pair.tgt_token_ids))
        src_pad_mask = ~pair.src_mask.bool()
        tgt_pad_mask = ~pair.tgt_mask.bool()
        size = [pair.tgt_token_ids.size(1)] * 2
        full_mask = torch.full(size, True, device=tgt_pad_mask.device)
        causal_mask = torch.triu(full_mask, diagonal=1)
        out_decoder = self.transformer(src_embeds, tgt_embeds,
                                       src_key_padding_mask=src_pad_mask,
                                       memory_key_padding_mask=src_pad_mask,
                                       tgt_mask=causal_mask, tgt_is_causal=True,
                                       tgt_key_padding_mask=tgt_pad_mask)
        return self.output(out_decoder).permute(0, 2, 1)
```

Let's go through this code:

- The constructor is straightforward: we just create the necessary modules.
- The `forward()` method takes an `NmtPair` as input (this class was defined in Chapter 14). The method starts by embedding the input tokens for both the source and target inputs, and it adds the positional encodings to both.
- Then the code uses the *not* operator (~) to invert both the source and target masks because they contain `False` for each padding token, but `nn.MultiheadAttention` expects `True` for tokens that it should ignore.
- Next, we create a square matrix of shape $[L_q, L_q]$, full of `True`, and we get all elements above the main diagonal using the `torch.triu()` function, with the rest defaulting to `False`. This results in a causal mask that we can use as the `tgt_mask` for the transformer: it will use this mask for the masked self-attention layer. Alternatively, you could call `nn.Transformer.generate_square_subsequent_mask()` to create the causal mask: just pass it the sequence length (`pair.tgt_token_ids.size(1)`) and set `dtype=torch.bool`.
- We then call the transformer, passing it the source and target embeddings, as well as all the appropriate masks.
- Lastly, we pass the result through the output `Linear` layer, and we permute the last two dimensions because `nn.CrossEntropyLoss` expects the class dimension to be dimension 1.

We can now create an instance of this model and train it exactly like our RNN encoder-decoder in Chapter 14. To speed up training and reduce overfitting, you can shrink the transformer quite a bit—use 4 heads instead of 8, just 2 layers in both the encoder and the decoder, and use an embedding size of 128:

```
nmt_tr_model = NmtTransformer(vocab_size, max_length, embed_dim=128, pad_id=0,
                              num_heads=4, num_layers=2, dropout=0.1).to(device)
[...]  # train this model exactly like the encoder-decoder in Chapter 14
```

Let's see how well this model performs:

```
>>> nmt_tr_model.eval()
>>> translate(nmt_tr_model,"I like to play soccer with my friends at the beach")
' Me gusta jugar al fútbol con mis amigos en la playa . </s>'
```

Great, even this tiny transformer trained for 20 epochs works rather well, so imagine a much bigger one trained on a much larger dataset, and you can start to see how ChatGPT and its friends can be so impressive.

Before we move on to other models, it's important to clean up the GPU RAM, or else it will quickly become saturated. For this, delete all variables that are no longer needed—especially models, optimizers, tensors, and datasets—using the `del` keyword, then call the `gc.collect()` function to run Python's garbage collector. When using a CUDA or AMD device, you must also call `torch.cuda.empty_cache()`. On Colab, you can view the available GPU RAM by selecting Runtime → "View resources" from the menu.

Now that you have a good understanding of the original Transformer architecture, let's look at encoder-only transformers.

Encoder-Only Transformers for Natural Language Understanding

When Google released the BERT model in 2018,[7] it proved that an encoder-only transformer can tackle a wide variety of natural language tasks: sentence classification, token classification, multiple choice question answering, and more! BERT also confirmed the effectiveness of self-supervised pretraining on a large corpus for transfer learning: BERT can indeed achieve excellent performance on many tasks, just by fine-tuning on a fairly small dataset for each task. Let's start by looking at BERT's architecture, then we'll look at how it was pretrained, and how you can fine-tune it for your own tasks.

Encoder-only models are generally not used for text generation tasks, such as autocompletion, translation, summarization, or chatbots, because they're much slower at this task than decoders. Decoders are faster because they are causal, so a good implementation can cache and reuse its previous state when predicting a new token. Conversely, encoders use nonmasked multi-head attention layers only, so they are naturally bidirectional; hence the B in BERT (Bidirectional Encoder Representations from Transformers). Whenever a new token is added, everything needs to be recomputed.

7 Jacob Devlin et al., "BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding", *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies* 1 (2019).

BERT's Architecture

BERT's architecture is almost identical to the original Transformer's encoder, with just three differences:

1. It's much bigger. BERT-base has 12 encoder blocks, 12 attention heads, and 768-dimensional embeddings, and BERT-large has 24 blocks, 16 heads, and 1,024 dimensions (while the original Transformer has 6 blocks, 8 heads, and 512 dimensions). It also uses trainable positional embeddings and supports input sentences up to 512 tokens.
2. It applies layer-norm just *before* each sublayer (attention or feedforward) rather than *after* each skip connection. This is called *pre-LN*, as opposed to *post-LN*, and it ensures that the inputs of each sublayer are normalized, which stabilizes training and reduces sensitivity to weight initialization. PyTorch's transformer modules default to post-LN, but they have a `norm_first` argument which you can set to `True` if you prefer pre-LN (however, some optimizations may not be implemented for pre-LN).
3. It lets you split the input sentence into two *segments* if needed. This is useful for tasks that require a pair of input sentences, such as natural language inference (i.e., does sentence A entail sentence B?) or multiple choice question answering (i.e., given question A, how good is answer B?). To pass two sentences to BERT, you must first append a *separation token* [SEP] to each one, then concatenate them. Furthermore, a trainable *segment embedding* is added to each token's representation: segment embedding #0 is added to all tokens within segment #0, and segment embedding #1 is added to all tokens within segment #1. In theory, we could have more segments, but BERT was only pretrained on inputs composed of one or two segments. Note that the positional encodings are also added to each token's representation, as usual (i.e., relative to the full input sequence, not relative to the individual segments).

That's all! Now let's look at how BERT was pretrained.

BERT Pretraining

The authors proposed two self-supervised pretraining tasks:

Masked language model (MLM)
: Each token in a sentence has a 15% probability of being replaced with a mask token, and the model is trained to predict what the original tokens were. This is often called a *cloze task* (i.e., fill in the blanks). For example, if the original sentence is "She had fun at the birthday party", then the model may be given the sentence "She [MASK] fun at the [MASK] party" and it must predict the original sentence: the loss is only computed on the mask token outputs.

To be more precise, some of the masked tokens are not truly masked: 10% are instead replaced by random tokens, and 10% are just left alone, neither masked nor randomized. Why is that? Well, the random tokens force the model to perform well even when mask tokens are absent: this is important since most downstream tasks don't use any mask tokens. As for the untouched tokens, they make the prediction trivial, which encourages the model to pay attention to the input token located at the position of the token being predicted. Without them, the model would soon learn to ignore this token and rely solely on the other tokens.

Next sentence prediction (NSP)

The model is trained to predict whether two sentences are consecutive or not. For example, it should predict that "The dog sleeps" and "It snores loudly" are consecutive sentences, while "The dog sleeps" and "The Earth orbits the Sun" are not consecutive.

This is a binary classification task, which the authors chose to implement by introducing a new *class token* [CLS]: this token is inserted at the beginning of the input sequence (position #0, segment #0), and during training the encoder's output, this token is passed through a binary classification head (i.e., a linear layer with a single unit, followed by the sigmoid function, and trained using `nn.BCELoss`, or just a linear layer with a single unit trained using `nn.BCEWithLogitsLoss`).

BERT was pretrained on both MLM and NSP simultaneously (see Figure 15-5 and the left side of Figure 15-6), using a large corpus of text—specifically the English Wikipedia and BooksCorpus. The goal of NSP was to make the class token's contextualized embedding a good representation of the whole input sequence. At first, it seemed that it indeed produced good sentence embeddings, but it was later shown that simply pooling all the contextualized embeddings (e.g., by computing their mean) yielded better results. In fact, researchers showed that NSP did not help much overall, so it was dropped in most later architectures.

In Chapter 14, we saw how to use the Transformers library to download a pretrained BERT model and its tokenizer. But you may want to train a BERT model from scratch, for example, if you're dealing with a domain-specific corpus of text. For this, one option is to build BERT yourself using the `nn.TransformerEncoder` module (e.g., based on an `nn.TransformerEncoderLayer` with `norm_first=True` to respect BERT's architecture), then preprocess your dataset according to the MLM algorithm, and train your model.

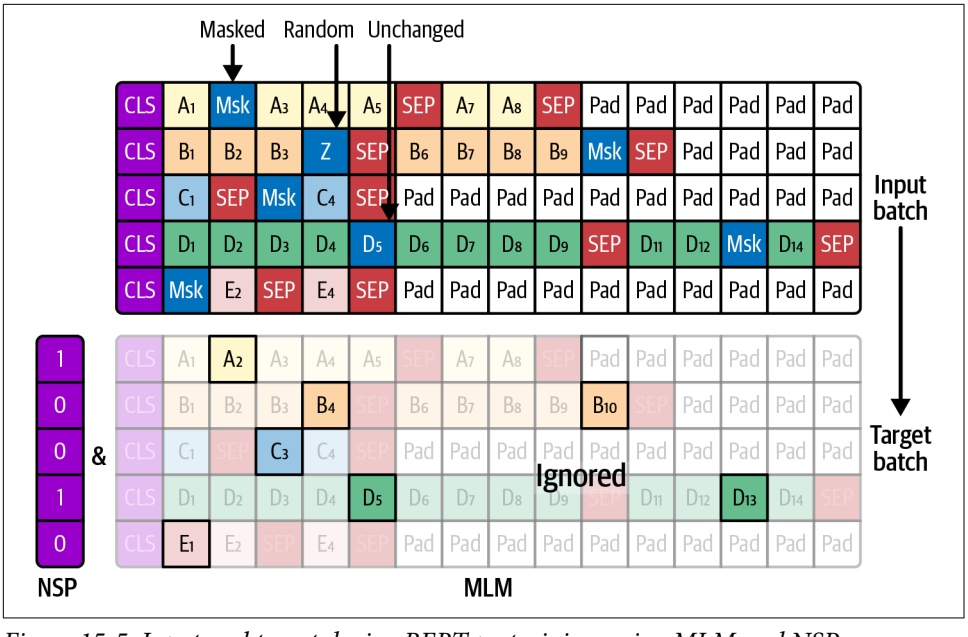

Figure 15-5. Input and target during BERT pretraining, using MLM and NSP

However, there's an easier way, using the Transformers library. Let's start by creating a tokenizer and a randomly initialized BERT model. For simplicity, we use a pretrained tokenizer, but of course you can train one from scratch instead, if you prefer. Make sure to tweak the `BertConfig` depending on your training budget, and the size and complexity of your dataset:

```
from transformers import BertConfig, BertForMaskedLM, BertTokenizerFast

bert_tokenizer = BertTokenizerFast.from_pretrained("bert-base-uncased")
config = BertConfig(  # adapt to training budget, and dataset size & complexity
    vocab_size=bert_tokenizer.vocab_size, hidden_size=128, num_hidden_layers=2,
    num_attention_heads=4, intermediate_size=512, max_position_embeddings=128)
bert = BertForMaskedLM(config)
```

Next, let's download the WikiText dataset (in real life, you would use your own dataset instead), and tokenize it:

```
from datasets import load_dataset

def tokenize(example, tokenizer=bert_tokenizer):
    return tokenizer(example["text"], truncation=True, max_length=128,
                     padding="max_length")

mlm_dataset = load_dataset("wikitext", "wikitext-2-raw-v1", split="train")
mlm_dataset = mlm_dataset.map(tokenize, batched=True)
```

This is where MLM comes in. We create a data collator, whose role is to bundle samples into batches, and we set its `mlm` argument to `True` to activate MLM, and also set `mlm_probability=0.15`: each token has a 15% probability of being masked (or possibly randomized or left alone, as we just discussed). We also pass the tokenizer to the collator: it will not be used to tokenize the text—we've already done that—but it lets the data collator know the masking and padding token IDs, as well as the vocabulary size (which is needed to sample random token IDs). With that, we just need to specify the `TrainingArguments`, pass everything to the `Trainer`, and call its `train()` method:

```
from transformers import Trainer, TrainingArguments
from transformers import DataCollatorForLanguageModeling

args = TrainingArguments(output_dir="./my_bert", num_train_epochs=5,
                         per_device_train_batch_size=16)
mlm_collator = DataCollatorForLanguageModeling(bert_tokenizer, mlm=True,
                                               mlm_probability=0.15)
trainer = Trainer(model=bert, args=args, train_dataset=mlm_dataset,
                  data_collator=mlm_collator)
trainer_output = trainer.train()
```

Once your model is pretrained, you can try it out using the pipelines API:

```
>>> from transformers import pipeline
>>> torch.manual_seed(42)
>>> fill_mask = pipeline("fill-mask", model=bert, tokenizer=bert_tokenizer)
>>> top_predictions = fill_mask("The capital of [MASK] is Rome.")
>>> top_predictions[0]
{'score': 0.04916289076209068,
 'token': 1010,
 'token_str': ',',
 'sequence': 'the capital of, is rome.'}
```

What? Rome is not the capital of a comma! The model is actually terrible because we only trained it for a single epoch here, just to confirm that everything works and the loss goes down. To get better results, we would need to train it for a *very* long time. The BERT authors trained it for about 4 days using 16 TPU devices on a much larger dataset. This is why most people avoid starting from scratch unless they really have to; you're generally better off downloading a model that was pretrained on a text corpus as close as possible to yours, then fine-tuning it on your own dataset. This can be done using MLM, like we just did, but starting from a pretrained model instead. Once you're happy with your pretrained model, you can fine-tune it on your target task. Let's see how.

BERT Fine-Tuning

BERT can be fine-tuned for many different tasks, changing very little for each task (see the righthand side of Figure 15-6).

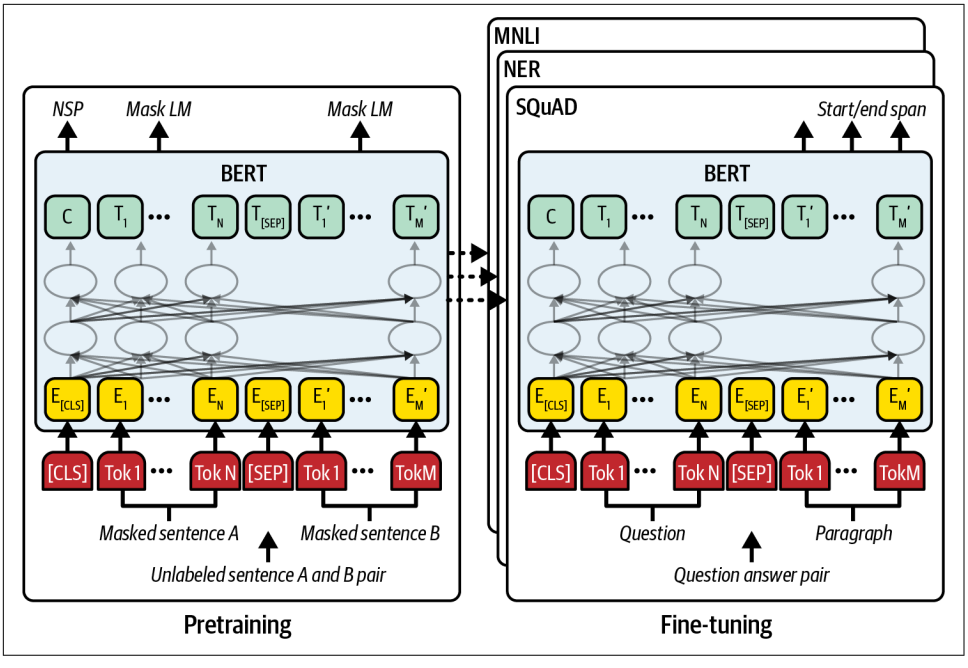

Figure 15-6. BERT pre-training (left) and fine-tuning process (right)[8]

For sentence classification tasks such as sentiment analysis, all output tokens are ignored except for the first one, which corresponds to the class token, and a new classification head replaces the NSP binary classification head (see the lefthand side of Figure 15-7). You can then fine-tune the whole model using the cross-entropy loss, optionally setting a lower learning rate for the lower layers, or freezing BERT altogether during the first few epochs (i.e., training only the new classification head). Using the exact same approach, you can tackle other sentence classification tasks. For example, the authors demonstrated that fine-tuning BERT yields excellent results on the CoLA dataset, which asks whether a sentence is grammatically correct. Try it out on your own sentence classification tasks: it's likely to perform well even if your dataset is quite small, thanks to the magic of transfer learning.

The BERT authors found that adding the MLM loss to the fine-tuning loss (scaled by a hyperparameter) helps stabilize training and reduces overfitting.

8 This is adapted from Figure 1 from the BERT paper, with the kind authorization of the authors.

For token classification, the classification head is applied to every token (see the righthand side of Figure 15-7). For example, BERT can be fine-tuned for *named entity recognition* (NER), where the model tags the parts of the text that correspond to names, dates, places, organizations, or other *entities*. This is often used in legal, financial, or medical applications. The same approach can be used for other token classification tasks, such as tagging grammatical errors; analyzing sentiment at the token level; locating subjects, nouns, and verbs (this is *part-of-speech tagging*); or locating questions, statements, and greetings (this is *dialogue act tagging*); and more.

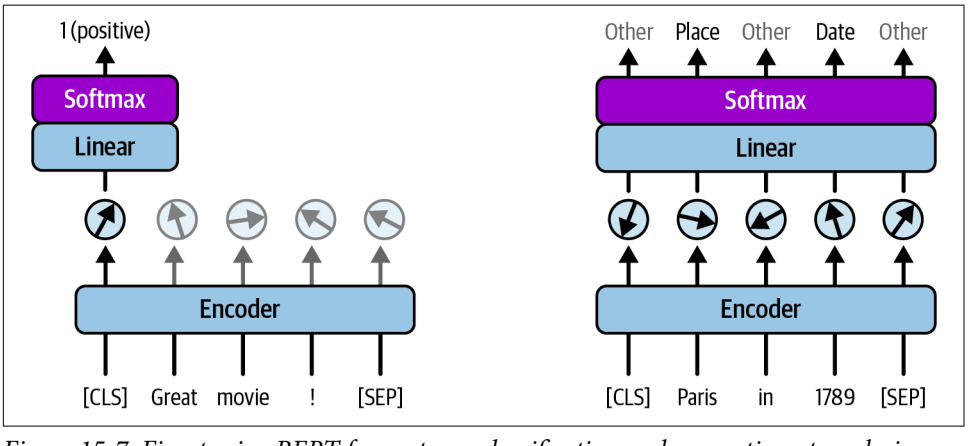

Figure 15-7. Fine-tuning BERT for sentence classification such as sentiment analysis (left) or for token classification such as NER (right)

BERT can also be used to classify pairs of sentences. It works exactly like sentence classification, except that you pass in two sentences instead of one. For example, this can be used for *natural language inference* (NLI) where the model must determine whether sentence A entails sentence B, or contradicts it, or neither (e.g., the *multi-genre NLI* dataset, or MNLI). It can also be used to detect whether two sentences have the same meaning, are just paraphrasing each other (e.g., the QQP or MRPC datasets), or to determine whether the answer to question A is present in sentence B (e.g., QNLI dataset).

For *multiple choice question answering* (MCQA), BERT is called once for each possible answer, placing the question in segment #0 and the possible answer in segment #1. For each answer, the class token output is passed through a linear layer with a single unit, producing a score. Once we have all the answer scores, we can convert them to probabilities using a softmax layer (see Figure 15-8), and we can use the cross-entropy loss for fine-tuning (or better, drop the softmax layer and use the `nn.CrossEntropyLoss` directly on the answer scores).

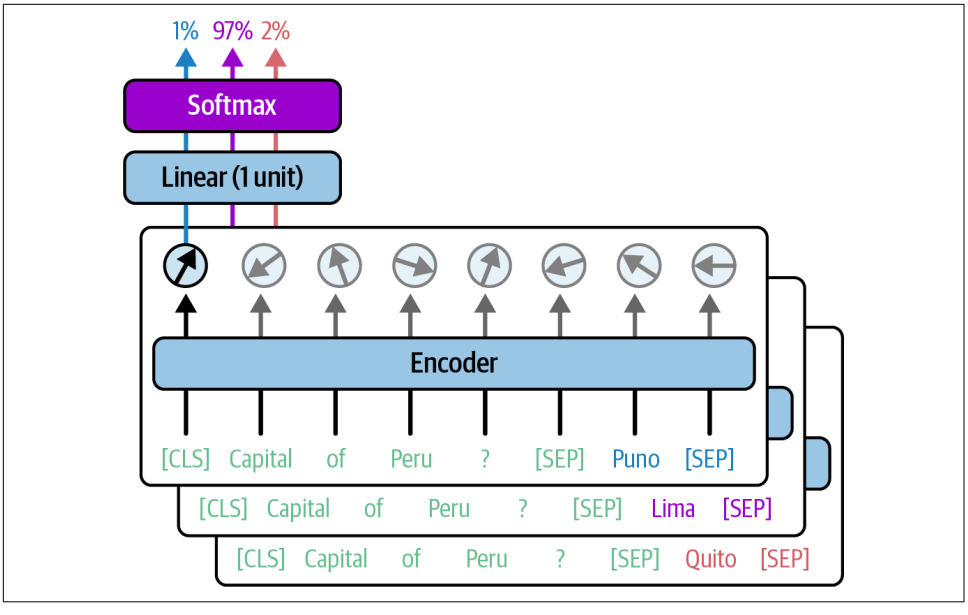

Figure 15-8. Using an encoder-only model to answer multiple-choice questions

BERT is also great for *extractive question answering*: you ask it a question (in segment #0) about some text called the *context* (in segment #1), and BERT must locate the answer within the context. For this, you can add a linear layer with two units on top of BERT to output two scores per token: a start score and an end score. During fine-tuning, you can treat them as logits for two separate binary classification tasks: the first determines whether a token is the first token in the answer, and the second determines whether it is the last. Of course most tokens are neither, and it's possible for one token to be both if the answer is a single token. At inference time, we select the pair of indices i and j that maximizes the sum of the start score of token i and the end score of token j, subject to $i \leq j$ and $j - i + 1 \leq$ maximum acceptable answer length. This approach allowed BERT to beat the state of the art on the SQuAD dataset, a popular question answering dataset.

The Transformers library provides convenient classes and checkpoints for each of these use cases, such as `BertForSequenceClassification` or `BertForQuestionAnswering` (see Chapter 14).

The BERT authors also showed that BERT could be fine-tuned to measure *semantic textual similarity* (STS); for example, on the *STS benchmark* dataset (STS-B), you feed the model two sentences and it outputs a score that indicates how semantically similar the sentences are. That said, if you want to find the most similar pair of

sentences in a dataset containing N sentences, you will need to run BERT on $O(N^2)$ pairs: this could take hours if the dataset is large. Instead, it's preferable to use a model such as Sentence-BERT (SBERT)[9] which is a variant of BERT that was fine-tuned to produce good sentence embeddings. Start by running each sentence through SBERT to get its sentence embedding, then measure the similarity between each pair of sentence embeddings using a similarity measure such as the *cosine similarity* (e.g., using PyTorch's `F.cosine_similarity()` function). This is the cosine of the angle between two vectors, so its value ranges from –1 (completely opposite) to +1 (perfectly aligned). Since measuring the cosine similarity is much faster than running BERT, and since the model processes much shorter inputs (i.e., sentences rather than sentence pairs), the whole process will take seconds rather than hours.

Sentence embedding can also be extremely useful in many other applications:

Text clustering, to organize and better understand your data
: You can process a large number of documents through SBERT to obtain their sentence embeddings, then apply a clustering algorithm such as k-means or HDBSCAN (see Chapter 8) on the embeddings to group your documents based on semantic similarity. It often helps to reduce dimensionality before running the clustering algorithm, for example, using PCA or UMAP (see Chapter 7).

Semantic search
: The goal is to let the user find documents based on the query's meaning rather than just keyword matching. First, encode your documents (or chunks of documents) using SBERT and store the sentence embeddings. When a user submits a search query, encode it using SBERT and find the documents whose embeddings are most similar to the query's embedding, for example, based on cosine similarity.

Reranking search results
: If you have an existing search system that you don't want to replace, you can often improve it significantly by reranking the search results based on semantic similarity with the query.

Vector databases, such as Pinecone, Weaviate, ChromaDB, Qdrant, or Milvus, are designed for storing and searching for documents based on their embeddings. More traditional databases, such as PostgreSQL or MongoDB, also have growing support for embeddings, although it's not as optimized yet.

9 Nils Reimers, Iryna Gurevych, "Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks", arXiv preprint arXiv:1908.10084 (2019).

Over the years, many variants of SBERT have been released. One of the easiest ways to download and use them is via the Sentence Transformers library created by UKPLab and maintained by Hugging Face (it's preinstalled on Colab). For example, the following code downloads the all-MiniLM-L6-v2 model, which is very fast and lightweight but still produces high-quality sentence embeddings. The code uses it to encode three sentences, then it measures the similarity between each pair of sentences:

```
from sentence_transformers import SentenceTransformer

model = SentenceTransformer("all-MiniLM-L6-v2")
sentences = ["She's shopping", "She bought some shoes", "She's working"]
embeddings = model.encode(sentences, convert_to_tensor=True)
similarities = model.similarity(embeddings, embeddings)
```

Let's look at the similarity matrix:

```
>>> similarities
tensor([[1.0000, 0.6328, 0.5841],
        [0.6328, 1.0000, 0.3831],
        [0.5841, 0.3831, 1.0000]], device='cuda:0')
```

We see that there are 1s in the main diagonal, confirming that each sentence is perfectly similar to itself, and we also see that "She's shopping" is more similar to "She bought some shoes" (the cosine similarity is 0.6328) than to "She's working" (0.5841).

Now that we've examined BERT in detail, let's look at some of its offspring.

Other Encoder-Only Models

Following Google's footsteps, many organizations released their own encoder-only models. Let's look at the most popular ones and discuss their main innovations.

RoBERTa by Facebook AI, July 2019 (125M to 355M parameters)

This model is similar to BERT but its performance is better across the board in large part because it was pretrained for longer and on a larger dataset. MLM was used for pretraining, but NSP was dropped. Importantly, the authors used *dynamic masking*, meaning that the tokens to mask were masked on the fly *during* training rather than just once before training (as BERT did), so the same piece of text is masked differently across different epochs. This provides the model with more data diversity, reducing overfitting and leading to better generalization.

When we fine-tuned BERT earlier in this chapter, we actually used dynamic masking and we dropped NSP, so we were following RoBERTa's pretraining approach.

DistilBERT by Hugging Face, October 2019 (66M)

This model is a scaled-down version of BERT: it's 40% smaller and 60% faster, yet it manages to reach about 97% of BERT's performance on most tasks, making it a great choice for low-resource environments (e.g., mobile devices), for low-latency applications, or for quick fine-tuning.

As its name suggests, DistilBERT was trained using a technique called *model distillation*, first introduced by Geoffrey Hinton et al. in 2015.[10] The idea of distillation is to train a small *student model* (e.g., DistilBERT) using the estimated probabilities from a larger *teacher model* (e.g., BERT) as the targets (see Figure 15-9). These are *soft targets* rather than the usual one-hot vectors: it makes training much faster and more data efficient, as it allows the student to directly aim for the correct distribution, rather than having to learn it over many samples, bouncing between one extreme and the other and slowly settling somewhere in between. As a result, distillation often works better than training the student from scratch on the same dataset as the teacher!

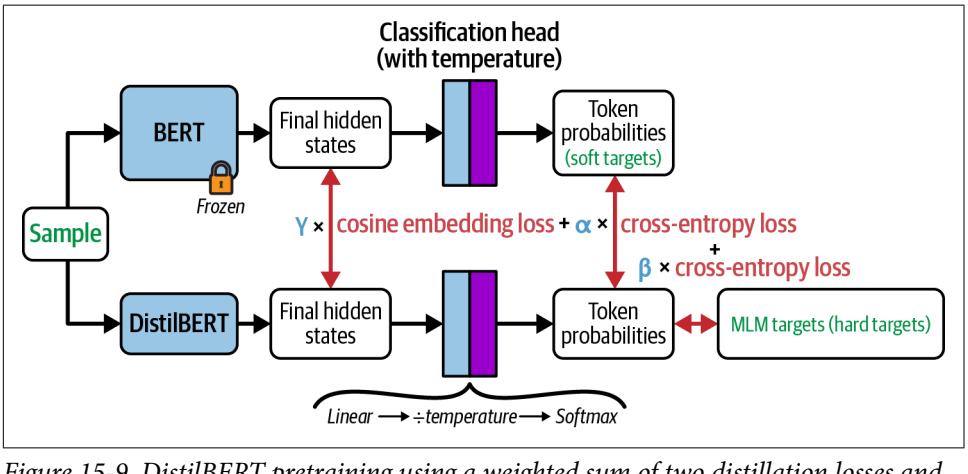

Figure 15-9. DistilBERT pretraining using a weighted sum of two distillation losses and the MLM loss

10 Geoffrey Hinton et al., "Distilling the Knowledge in a Neural Network", arXiv preprint arXiv:1503.02531 (2015).

Note that the estimated probabilities for both the teacher and the student are smoothed a bit—during training only—by dividing the final logits by a temperature greater than 1 (typically 2). This provides the student with a more nuanced signal covering all possible options, rather than just focusing on the correct answer. Hinton dubbed this *dark knowledge*. For example, if the input is "It's sunny and I feel [MASK]", the teacher might normally estimate that the masked word has a 72% probability of being "great", and 27% of being "good", and just 0.5% of being "bad". But if we apply a temperature of 2, then these probabilities get smoothed out to about 60%, 36%, and 5%, respectively. It's helpful to know that "bad" is a plausible option here, even if it's unlikely.

DistilBERT's training loss also had two more components: the standard MLM loss, as well as a *cosine embedding loss* which minimizes the cosine similarity between the student's and teacher's final hidden states (i.e., the output embeddings just before the classification head). This encourages the student to "think" like the teacher, not just make the same predictions, and it leads to faster convergence and better performance. Later models, such as TinyBERT, pushed this idea further by aligning other internal states, such as the attention weights. DistilBERT's final loss is a weighted sum of the three losses (the authors used weights $\alpha=5$, $\beta=2$, $\gamma=1$).

ALBERT by Google Research, December 2019 (12M–235M)

All encoder layers in this model share the same weights, making it much smaller than BERT, but not faster. This makes it great for use cases where memory size is limited. In particular, it's a good model to use if you want to train an encoder-only model from scratch on a GPU with little VRAM.

ALBERT also introduced *factorized embeddings* to reduce the size of the embedding layer: in BERT-large, the vocabulary size is about 30,000, and the embedding size is 1,024, which means that the embedding matrix has over 30 million parameters! ALBERT replaces this huge matrix with the product of two much smaller matrices (see Figure 15-10). In practice, this can be implemented by reducing the embedding size—ALBERT uses 128—then adding a linear layer immediately after the embedding layer to project the embeddings to the higher dimensional space, such as 1,024 dimensions for ALBERT-large. The embedding layer ends up with roughly 3.8M parameters (~30,000 × 128), and the linear layer has about 0.13M parameters (128 × 1,024), so the total is less than 4M parameters, down from over 30M: nice!

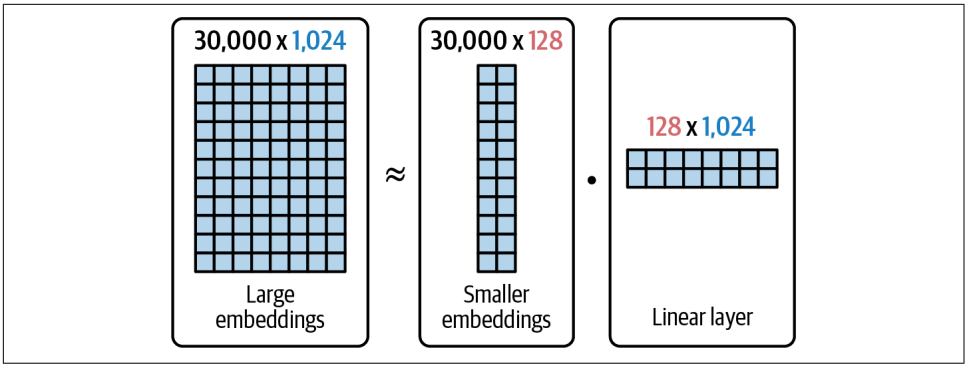

Figure 15-10. An excessively large embedding matrix can be replaced with the product of two smaller matrices. This can be implemented using smaller embeddings and projecting them to higher dimensions using a linear layer.

ALBERT also replaced NSP with *sentence order prediction* (SOP): given two consecutive sentences, the goal is to predict which one comes first. This is a much harder task than NSP, and it led to significantly better sentence embeddings.

ELECTRA by Google Research, March 2020 (14M–335M)

This model introduced a new pretraining technique called *replaced token detection* (RTD): they trained two models jointly—a small generator model and a larger discriminator model, both encoder only. The generator is only used during pretraining, while the discriminator is the final model we're after. The generator is simply trained using regular MLM with dynamic masking. For each mask token, a replacement token is sampled from its top predictions. The resulting text is fed to the discriminator model, which must predict whether each token is the original or not.

For example (see Figure 15-11), if the original text is "She likes him" and it is masked as "She [MASK] him", the generator's top predictions might include "likes", "loves", "hears", "pushes", and one of these is chosen randomly, say "pushes", so the sentence becomes "She pushes him". The discriminator must then try to predict [1, 0, 1], since the first and third tokens are the same as in the original text, but not the second token. As the generator improves, the replaced tokens gradually become less and less obviously wrong, forcing the discriminator to become smarter and smarter. After training, we can throw away the generator and drop the binary classification head from the discriminator to get the final model.

This technique is more sample-efficient than MLM since the discriminator learns from more tokens per example, thus it converges faster, generally achieving the same performance as larger BERT models. That said, the benefits are not always worth the additional complexity.

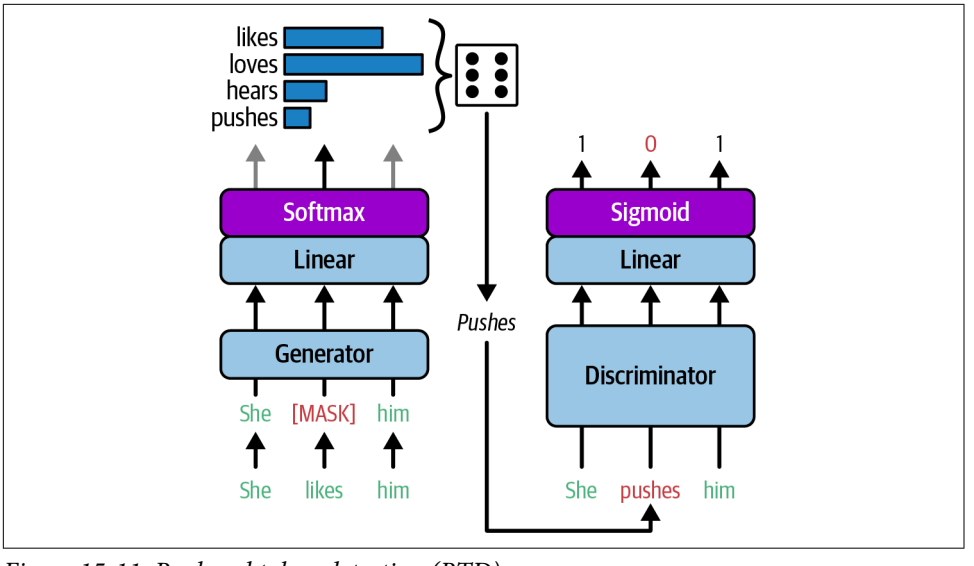

Figure 15-11. Replaced token detection (RTD)

DeBERTa by Microsoft, January 2021 (139M–1.5B)

DeBERTa is a fairly large model that beat the state of the art on many NLU tasks. It removes the usual positional embedding layer, and instead uses *relative positional embeddings* when computing the attention scores inside every multi-head attention layer: when deciding how much the i^{th} query token should attend to the j^{th} key token, the model has access to a learned embedding for the relative position $i - j$. DeBERTa wasn't the first model to do this, as we will see later in this chapter, but it introduced a variant of this technique—named *disentangled attention*—which gives the model more flexibility in how it can combine semantic and positional information.

DeBERTaV3, released in July 2021, combined the ideas from DeBERTa with ELECTRA-style RTD, and it reached even higher performance. It remains a popular model for NLU tasks to this day. However, disentangled attention adds some complexity and compute cost, so subsequent models have opted for simpler approaches, as we will see.

More encoder-only models on Hugging Face Hub

If you explore the encoder-only models on the Hugging Face Hub, you will find many variants of the standard models we discussed so far:

- With various sizes (e.g., BERT-base versus BERT-large)
- Pretrained on a non-English language (e.g., CamemBERT for French) or even on multiple languages (e.g., IndicBERT for 12 major Indian languages)

- Pretrained on cased or uncased text
- Tweaked for specific tasks (e.g., BERT for question answering)

You will also find many domain-specific models, such as:

- ClinicalBERT for clinical applications
- SciBERT for scientific applications
- PubMedBERT for biomedicine
- FinBERT for finance
- GraphCodeBERT for coding applications
- Twitter-RoBERTa-base for social media applications
- PatentBERT for patent applications
- LexLM for legal applications

Most of these are simply fine-tuned versions of standard encoder-only models such as BERT or RoBERTa, but some were pretrained entirely from scratch. A few also introduced new ideas; for example, GraphCodeBERT is a BERT model pretrained on code using not only MLM, but also two structure-aware tasks: it has to find where in the code each variable was defined and used, and it also has to predict the data flow (e.g., in `z = x + y`, variable `z` comes from variables `x` and `y`).

The Hugging Face Hub also contains many compressed variants of standard models. They are small and usually fast, and were trained using distillation, weight-sharing, and/or other techniques such as quantization (see Appendix B). Popular examples include: DistilBERT, TinyBERT, MobileBERT, MiniLM (available for various base models), DistilRoBERTa, and MiniDeBERTa-v2. As we saw with DistilBERT, these models are great for low-resource environments, low latency, and quick fine-tuning.

Speaking of quick fine-tuning, you will also find many *adapter models* on the Hugging Face Hub. An adapter model is based on a frozen standard model such as BERT, plus some small trainable components called *adapters*: when you fine-tune the adapter model, the base model doesn't change, only the adapters. As a result, fine-tuning is much faster and less computationally expensive, and you can get great performance on your task using fairly little training data. For example, AdapterHub/bert-base-uncased-pf-sst2 is an adapter model based on the bert-base-uncased model and fine-tuned for sentiment analysis on the SST 2 dataset. Chapter 17 shows how to build and fine-tune your own adapter models.

OK, time to step back. We've learned all about the Transformer architecture, and we even built a translation transformer from scratch, and now we've looked into encoder-only models like BERT and how they can be used for many different NLU tasks. Lastly, we examined the key innovations powering some of the most popular

encoder-only models, and the main categories of pretrained encoder-only models you can find on the Hugging Face Hub (i.e., standard, multilingual, task-specific, domain-specific, compressed, and adapter models—these categories are not exclusive). It's now time to look at decoder-only models such as GPT.

Over the last few years, large organizations have shifted their focus toward decoders, but encoder-only models are still alive and kicking. Their relatively small size makes them fast and accessible to all, easy to fine-tune, and immensely useful for a wide range of applications.

Decoder-Only Transformers

While Google was working on the first encoder-only model (i.e., BERT), Alec Radford and other OpenAI researchers were taking a different route: they built the first decoder-only model, named GPT.[11] This model paved the way for today's most impressive models, including most of the ones used in famous chatbots like ChatGPT or Claude.

The GPT model (now known as GPT-1) was released in June 2018. GPT stands for *Generative Pre-Training*: it was pretrained on a dataset of about 7,000 books and learned to predict the next token, so it can be used to generate text one token at a time, just like the original Transformer's decoder. For example, if you feed it "Happy birthday", it will predict "birthday to", so you can append "to" to the input and repeat the process (see Figure 15-12).

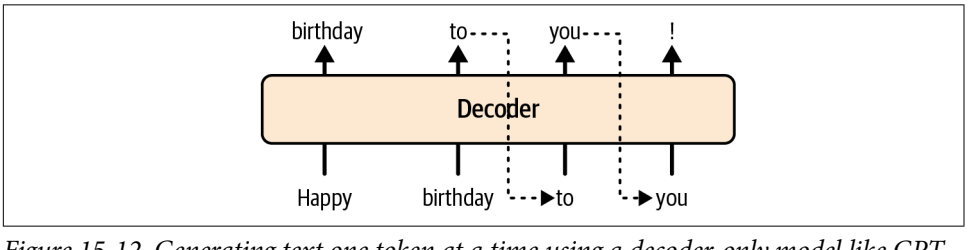

Figure 15-12. Generating text one token at a time using a decoder-only model like GPT

Decoder-only models are great at *text generation* tasks, such as auto-completion, code generation, question answering (including free text answers), math and logical reasoning (to some extent), and chatbots. They can also be used for summarization or translation, but encoder-decoder models are still popular choices for these tasks, as they often have a better understanding of the source text, thanks to the encoder. Decoder-only models can also perform text classification quite well, but encoder-only

11 Alec Radford et al., "Improving Language Understanding by Generative Pre-Training" (2018).

models shine in this area, as they are faster and often provide a similar performance with a smaller model.

At inference time, encoder-only models only need to look at their inputs once to make their predictions, while decoder-only models require one run per generated token (just like the decoder in encoder-decoder models). That's because decoders are autoregressive, so the generation process is sequential. That said, decoders can hugely benefit from caching, as I mentioned earlier.

In this section, we will look at the architecture of GPT-1 and its successor GPT-2, and we will see how decoder-only models like these can be used for various tasks. We will also see that these models can perform tasks that they were never explicitly trained on (zero-shot learning) or for which they only saw a few examples (few-shot learning). Lastly, we will then use the Transformers library to download a small decoder-only model (GPT-2) then a large one (Mistral-7B) and use them to generate text and answer questions.

GPT-1 Architecture and Generative Pretraining

During pretraining, GPT-1 was fed batches of 64 sequences randomly sampled from the book corpus, and it was trained to predict the next token for every single input token. Each sequence was exactly 512 tokens long, so GPT-1 did not need any padding token. In fact, it didn't use special tokens at all during pretraining, not even start-of-sequence or end-of-sequence tokens. Compared to BERT, it's a much simpler pretraining process. It also provides the same amount of data for every token position, whereas BERT sees less data for the last positions than for the first, due to padding.

GPT-1's architecture has two important differences compared to the original Transformer's decoder:

- There's no cross-attention block since there's no encoder output to attend to: each decoder block only contains a masked multi-head attention layer and a two-layer feedforward network (each with its own skip connection and layer norm).
- It's much bigger: it has 12 decoder layers instead of 6, the embedding size is 768 instead of 512, and it has 12 attention heads instead of 8. That's a total of 117 million parameters.

Counterintuitively, you cannot use PyTorch's `nn.TransformerDecoder` module to build a decoder-only model. That's because it contains cross-attention layers that cannot be easily removed. Instead, you can use the `nn.TransformerEncoder` module, and always call it with a causal mask.

Out-of-the-box, GPT-1 was very impressive at text generation. For example, its authors asked it to tell the story of a scientist discovering a herd of English-speaking unicorns in an unexplored valley, and the story it generated seemed like it had been written by a human (you can read it at *https://homl.info/unicorns*). It's not quite as impressive today, but back then it was truly mind-blowing.

The authors also fine-tuned GPT-1 on various tasks, including textual entailment, semantic similarity, reading comprehension, or common sense reasoning, and it beat the state of the art on many of them, confirming the power of pretraining for NLP. For each task, the authors only made minor changes to the architecture:

- For text classification tasks, a classification head is added on top of the last token's output embedding. See the righthand side of Figure 15-13.
- For entailment and other classification tasks requiring two input sentences, the model is fed both sentences separated by a delimiter token (just a regular $ sign), and again a classification head is added on top of the last token's output embedding.
- For semantic similarity, since the order of the two sentences shouldn't matter, the model gets called twice: once with sentence 1 $ sentence 2, and once with sentence 2 $ sentence 1. The last token's output embeddings for both cases are added itemwise and the result is fed to a regression head.
- For multiple choice question answering, the approach is very similar to BERT's: the model is called once per possible answer, with both the context (including the question) and the possible answer as input, separated by a $ sign, then the last token's output embedding is passed through a linear layer to get a score. All the answer scores are then passed through a softmax layer.
- In all cases they added a start-of-sequence token and an end-of-sequence token.

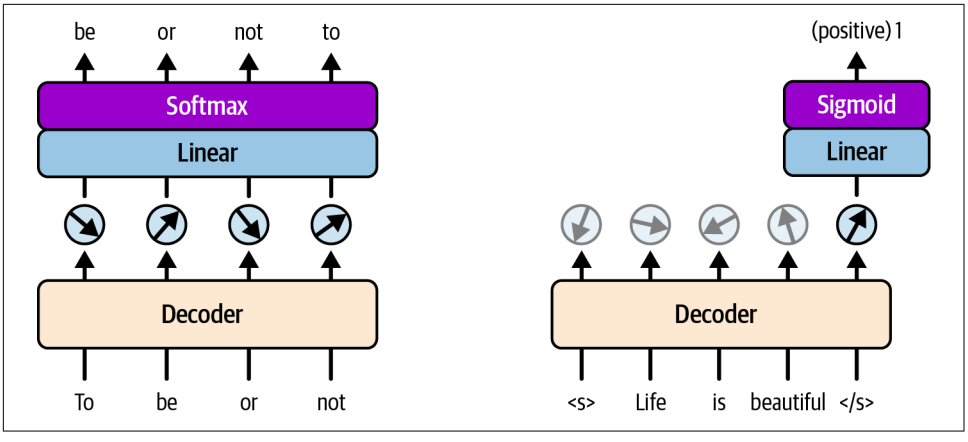

Figure 15-13. Pretraining GPT-1 using next token prediction (NTP, left) and fine-tuning it for classification (right)

GPT-2 and Zero-Shot Learning

Just a few months later, in February 2019, Alec Radford, Jeffrey Wu, and other OpenAI researchers published the GPT-2 paper,[12] which proposed a very similar architecture to GPT-1,[13] but larger still. It came in four sizes, and the largest model had 48 decoder layers, 20 attention heads, an embedding size of 1,600, and a context window of 1,024 tokens, for a total of over 1.5 billion parameters!

For such a large model, the authors needed a gigantic dataset, so they initially tried using Common Crawl which contains over two billion web pages. However, many of these pages are just gibberish (e.g., long tables of data). So the authors built a higher-quality dataset named *WebText*, composed of about eight million pages linked from highly ranked Reddit pages.

Most importantly, GPT-2 performed incredibly well on many tasks without any fine-tuning: this is called *zero-shot learning* (ZSL). For example:

- For question answering, you can simply append "A:" to the question (e.g., "What is the capital of New-Zealand? A:") then feed this prompt to GPT-2. It will complete it with the answer (e.g., "Wellington").
- For summarization, you can append "TL;DR:" to the document you want to summarize, and GPT-2 will often produce a decent summary.

12 Alec Radford et al., "Language Models Are Unsupervised Multitask Learners" (2019).

13 There were a few minor tweaks, such as using pre-LN rather than post-LN, downscaling the weights depending on the number of residual layers, and tweaks to the BPE tokenizer. Please see the paper for more details.

- For translation, you can create a prompt containing a few examples to guide the model, such as "Bonjour papa = Hello dad" and "Le chien dort = The dog is sleeping", then append the text you want to translate, for example "Elle aime le chocolat =", and GPT-2 will hopefully complete the prompt with the correct English translation: "She loves chocolate".

Importantly, the authors showed that ZSL performance seemed to increase regularly with the model size: doubling the model size offered a roughly constant improvement (that's a log-linear relationship). Maybe creating a superhuman AI was just a matter of training a large enough transformer?

GPT-2's performance was so impressive that OpenAI initially chose not to release the largest model. Officially, this was for the public's safety, citing risks like automated disinformation and spam. But skeptics argued that it was both a publicity stunt and a shift toward closed-source AI, and perhaps even a move to influence future regulation. The full GPT-2 model was eventually released months later, but it was the last open one from OpenAI until August 2025, when a couple of open-weight models were released (GPT-OSS).

GPT-3, In-Context Learning, One-Shot Learning, and Few-Shot Learning

Following their bigger-is-better philosophy, OpenAI created GPT-3 in 2020.[14] It had roughly 40 billion parameters, and was trained on a monstrously large dataset of about 570 gigabytes (including WebCrawl this time).

This model indeed was far better across the board than GPT-2. In particular, it was much better at zero-shot tasks. But most importantly, the authors showed that GPT-3 was incredibly good at generalizing from just a few examples. This is called *few-shot learning* (FSL), or *one-shot learning* (OSL) if there's a single example. To tackle FSL or OSL tasks, the authors simply inserted the example(s) in the prompt: they dubbed this *in-context learning* (ICL). For example, if you feed the following prompt to GPT-3, can you guess what it will output?

```
Alice was friends with Bob. Alice went to visit her friend ___. → Bob
George bought some baseball equipment, a ball, a glove, and a ___. →
```

That's right, it will output the missing word, "bat". The idea of feeding the model some examples in the prompt itself was already present in the GPT-2 paper (remember the translation example?), but it wasn't really formalized, and the GPT-3 paper explored it in much more depth.

14 Tom B. Brown et al., "Language Models are Few-Shot Learners", arXiv preprint arXiv:2005.14165 (2020).

In-context learning is an increasingly popular approach to one-shot learning and few-shot learning, but there are many others. ICL is new, but OSL and FSL are old (like ZSL).

Let's download GPT-2 and generate some text with it (we will play with GPT-3 via the API later in this chapter).

Using GPT-2 to Generate Text

As you might expect, we can use the Transformers library to download GPT-2. By default, we get the small version (124M parameters):

```
from transformers import AutoTokenizer, AutoModelForCausalLM

model_id = "gpt2"
gpt2_tokenizer = AutoTokenizer.from_pretrained(model_id)
gpt2 = AutoModelForCausalLM.from_pretrained(
    model_id, device_map="auto", dtype="auto")
```

Let's go through this code:

- After the imports, we load GPT-2's pretrained tokenizer and the model itself.
- To load the model, we use `AutoModelForCausalLM.from_pretrained()`, which returns an instance of the appropriate class based on the checkpoint we ask for (in this case it returns a `GPT2LMHeadModel`). Since it's a causal language model, it's capable of generating text, as we will see shortly.
- The `device_map="auto"` option tells the function to automatically place the model on the best available device, typically the GPU. If you have multiple GPUs and the model is too large for one, it may even be sharded across GPUs.
- The `dtype="auto"` option asks the function to choose the most appropriate data type for the model weights, based on what's available in the model checkpoint and your hardware. Typically, it loads the model using 16-bit floats if your hardware supports it (e.g., a modern GPU with mixed-precision support), or it falls back to 32-bit floats. Using half precision (16-bit) uses half the memory, which lets you load larger models, and it also gives the model a substantial speed boost because modern GPUs have hardware accelerations for this, and half precision reduces the amount of data that needs to be transferred between the CPU and GPU.

Now let's write a little wrapper function around the model's `generate()` method to make it very easy to generate text:

```
def generate(model, tokenizer, prompt, max_new_tokens=50, **generate_kwargs):
    inputs = tokenizer(prompt, return_tensors="pt").to(model.device)
```

```
    outputs = model.generate(**inputs, max_new_tokens=max_new_tokens,
                             pad_token_id=tokenizer.eos_token_id,
                             **generate_kwargs)
    return tokenizer.decode(outputs[0], skip_special_tokens=True)
```

Our `generate()` function tokenizes the given prompt, transfers the resulting token IDs to the GPU, calls the given model's `generate()` method to extend the prompt, adding up to 50 new tokens (by default) or less if it runs into an end-of-sequence token, and lastly it decodes the resulting token IDs to return a nice string containing the extended text. Since GPT-2 was pretrained without padding, we must specify which token we want to use for padding when calling the model's `generate()` method: it's common to use the end-of-sequence token for this. This function processes a single prompt so there will be no padding anyway, but specifying the padding token avoids a pesky warning. Our function also accepts optional extra keyword arguments (`**generate_kwargs`) and passes them on to the model's `generate()` method. This will come handy very soon.

Decoder-only models often pad on the left side, for more efficient generation, since new tokens are added on the right.

Now let's try generating some text about a talking unicorn:

```
>>> prompt = "Scientists found a talking unicorn today. Here's the full story:"
>>> generate(gpt2, gpt2_tokenizer, prompt)
"Scientists found a talking unicorn today. Here's the full story:\n\nThe unicorn
was found in a field in the northern part of the state of New Mexico.\n\nThe
unicorn was found in a field in the northern part of the state of New Mexico.
\n\nThe unicorn was found in a field in"
```

Hmm, it starts out pretty well, but then it just repeats itself—what's happening? Well, by default the `generate()` method simply picks the most likely token at each step, which is fine when you expect very structured output, or for tasks such as question answering, but for creative writing it often gets the model stuck in a loop, producing repetitive and uninteresting text. To fix this, we can set `do_sample=True` to make the `generate()` method randomly sample each token based on the model's estimated probabilities for the possible tokens, like we did with our Shakespeare model in Chapter 14. Let's see if this works:

```
>>> torch.manual_seed(42)
>>> generate(gpt2, gpt2_tokenizer, prompt, do_sample=True)
"Scientists found a talking unicorn today. Here's the full story:\n\nThere
aren't lots of other unicorns and they have been making their way across the
United States since at least the 1800s, but this year there weren't a solitary
unicorn on the land. Today, there are around 1,000."
```

Well, that's certainly less repetitive! To get better results, you can play with the `generate()` method's many arguments, such as:

`temperature`
: Defaults to 1; decrease for more predictable outputs, or increase for more diverse outputs (as we saw in Chapter 14)

`top_k`
: Only sample from the top *k* most probable tokens

`top_p`
: Restrict sampling to the smallest set of most probable tokens whose total probability is a least `top_p`

`num_beams`
: The beam width for beam search (introduced in Chapter 14); defaults to 1 (i.e., no beam search)

Top-*p* sampling (a.k.a., nucleus sampling) is often preferred over top-*k* sampling, as it adapts to the probability distribution; for example, "The capital city of France is" has only one likely next token (i.e., "Paris"), and top-*p* sampling will always select it, while top-*k* sampling might occasionally pick an incorrect token. Conversely, "My favorite city is" has many likely next tokens, and top-*p* sampling will pick any one of them (favoring the most likely cities), but top-*k* sampling will only sample from the few most likely ones, ignoring many great cities.

So let's see if top-*p* sampling helps:

```
>>> torch.manual_seed(42)
>>> generate(gpt2, gpt2_tokenizer, prompt, do_sample=True, top_p=0.6)
"Scientists found a talking unicorn today. Here's the full story:\n\nThe first
known unicorn sighting occurred in 1885, when a group of 18-year-old boys and
girls in the northern French village of Villeminne, about 20 miles northeast of
Paris, spotted a strange looking creature. The unicorn"
```

That's much better! Now let's see how to use GPT-2 for question answering.

Using GPT-2 for Question Answering

Let's write a little function that takes a country name and asks GPT-2 to return its capital city:

```
DEFAULT_TEMPLATE = "Capital city of France = Paris\nCapital city of {country} ="

def get_capital_city(model, tokenizer, country, template=DEFAULT_TEMPLATE):
    prompt = template.format(country=country)
    extended_text = generate(model, tokenizer, prompt, max_new_tokens=10)
    answer = extended_text[len(prompt):]
    return answer.strip().splitlines()[0].strip()
```

The function starts by creating a prompt from a *prompt template*: it replaces the `{country}` placeholder with the given country name. Note that the prompt template includes one example of the task to help GPT-2 understand what to do and what format we expect: that's in-context learning. The function then calls our `generate()` function to add 10 tokens to the prompt: this is more than we need to write the capital city's name. Lastly, we do a bit of post-processing by removing the initial prompt as well as anything after the first line, and we strip away any extra spaces at the end. Let's try it out!

```
>>> get_capital_city(gpt2, gpt2_tokenizer, "United Kingdom")
'London'
>>> get_capital_city(gpt2, gpt2_tokenizer, "Mexico")
'Mexico City'
```

It works beautifully! Moreover, it's quite flexible with its input; for example, if you ask it for the capital of "UK", "The UK", "England", "Great Britain", or even "Big Britane", it will still return "London". That said, it's far from perfect:

- It makes many common mistakes (e.g., for Canada, it answers Toronto instead of Ottawa). Sadly, since GPT-2 was trained on many pages from the web, it picked up people's misconceptions and biases.
- When it's not sure, it just repeats the country's name, roughly 30% of the time. This might be because several countries have a capital city of the same name (e.g., Djibouti, Luxembourg, Singapore) or close (e.g., Guatemala City, Kuwait City).
- When the input is not a country, the model often answers "Paris", since that's the only example it had in its prompt.

One way to fix these issues is to simply use a much bigger and smarter model. For example, try using "gpt2-xl" (1.5B parameters) instead of "gpt2" when loading the model, then run the code again. It still won't be perfect, but you should notice a clear improvement. So let's see if an much larger model can do even better!

Downloading and Running an Even Larger Model: Mistral-7B

Mistral-7B is a decoder-only model released by a French startup named Mistral AI in May 2024. As its name suggests, it has seven billion parameters, and it implements several advanced Transformer techniques, such as grouped-query attention and sliding-window attention (see Chapter 17), which increase its speed and performance.

The good news is that it's released under the permissive Apache 2.0 license, and it's not too big to run on Colab GPUs. However, the model is *gated* on the Hugging Face Hub, meaning that the platform requires you to log in and agree to some terms: in this case, sharing your identity with the model authors. This is common for high-demand or sensitive models to allow model authors to monitor downloads

for usage analytics, reduce abuse, and contact users for potential future research collaboration. Let's go through all the steps needed to run this model on Colab (or on your own machine if your GPU has enough VRAM):

- Go to *https://huggingface.co* and log in if you already have an account. If not, click on Sign Up and follow the instructions.
- Once you have logged in to your account, go to *https://huggingface.co/mistralai/Mistral-7B-v0.3* (or use the Hub's search feature to find this page). You should see the *model card* containing useful information about this model, including code snippets and more. For this particular model, you should also see the message asking you to agree to share your contact information (see Figure 15-14). If you agree, click "Agree and access repository". Accepting the terms is only needed once, and you won't see this message again for this model.

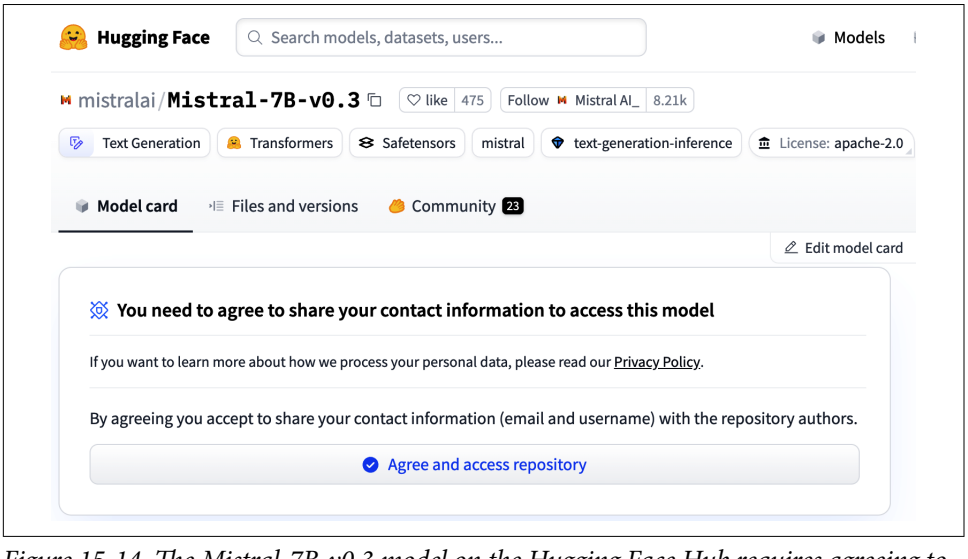

Figure 15-14. The Mistral-7B-v0.3 model on the Hugging Face Hub requires agreeing to share your identity with the model authors

Next, you need an access token, which we will use to log in to the Hub from our code:

- In the top righthand corner of the website, click on your profile icon, then select Access Tokens from the drop-down menu (or go to *https://huggingface.co/settings/tokens*). The website may ask you to confirm your identity at this point.
- Enter a name for your token, for example, `hf-read-mistral`.
- You must now select the "Token type": it can be Fine-grained, Read, or Write.

— In production, it's important to use an access token with very limited authorizations in case the token gets compromised. You would select the Fine-grained option (see Figure 15-15), then scroll down to the "Repositories permissions" section, search for mistralai/Mistral-7B-v0.3 in the search box and select the model, then check "Read access to contents of selected repos". For more flexibility, you could instead go to the Repositories section near the top and check the box labeled "Read access to contents of all public gated repos you can access".

— During development, using excessively restrictive access tokens can often slow you down, so you may prefer to select the Read token type, which gives full read access to your account, or even the Write token type, which gives full read/write access.

- Click the Create Token button, and copy the access token. Save it carefully, as it will never be shown again.

Keep your access tokens safe (e.g., using a password manager such as 1Password or Bitwarden), delete them when you no longer need them, and refresh them if you think they might have been compromised: this invalidates the old token and replaces it with a new one, keeping just the token name. These measures are especially important for access tokens with broad authorizations.

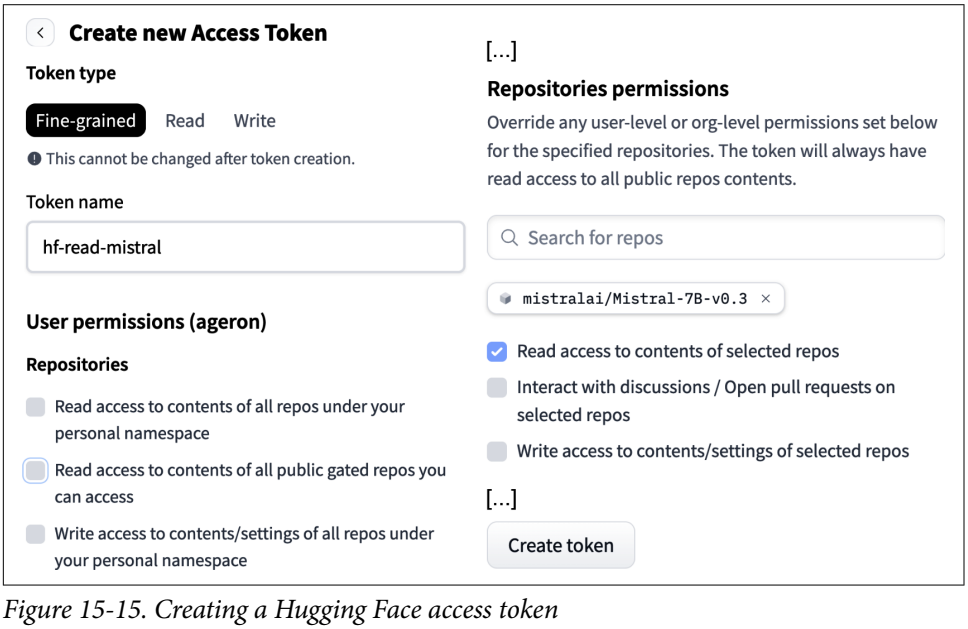

Figure 15-15. Creating a Hugging Face access token

OK, let's go back to Colab now. The last step before downloading the model is to get your notebook to log in to the Hugging Face Hub using the access token that you just created. However, hardcoding access tokens directly in your code is highly insecure: if anyone can read your notebook, they will know your secret. Luckily, Colab has a convenient feature to save your secrets safely and make them available to any notebooks you like without any hardcoding:

- Click on the key icon located in the vertical bar on the lefthand side of Colab's interface (see Figure 15-16).
- Click "Add new secret", then enter your secret's name (e.g., `token-hf-read-mistral`) and the secret value (i.e., your access token). The secret will be stored safely on Google's servers.
- Click the button located in the "Notebook access" column of your secret to give the current notebook access to your secret. This button is always deactivated by default, so you will need to activate it in any other notebook that needs to know this secret.

If you run a Colab notebook written by someone else, then make sure you trust the author or verify the code before activating notebook access for any of your secrets.

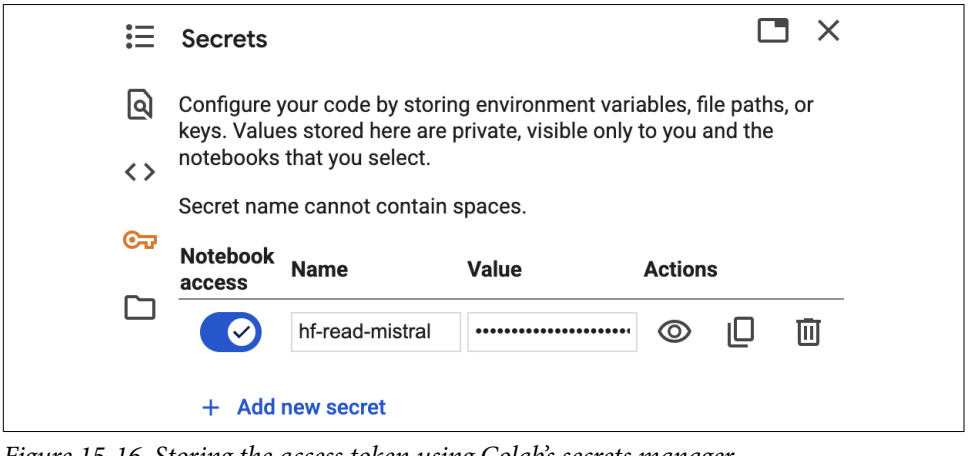

Figure 15-16. Storing the access token using Colab's secrets manager

Now you can run the following code to retrieve the secret access token:

```
from google.colab import userdata

access_token = userdata.get('token-hf-read-mistral')
```

Great! You now have your access token ready, so let's use it to log in to the Hugging Face Hub:[15]

```
from huggingface_hub import login

login(access_token)
```

Finally, you can load Mistral-7B, exactly like you loaded GPT-2:

```
model_id = "mistralai/Mistral-7B-v0.3"
mistral7b_tokenizer = AutoTokenizer.from_pretrained(model_id)
mistral7b = AutoModelForCausalLM.from_pretrained(
    model_id, device_map="auto", dtype="auto")
```

Now you can play around with this model, make it write stories about talking unicorns, or use it to answer all sorts of questions. If you use it to find capital cities, as we did earlier, you will see that it finds the correct answer for almost all countries in the world. Moreover, the very few mistakes it makes are actually quite reasonable.[16]

But what if we want to chat with this model?

Turning a Large Language Model into a Chatbot

To build a chatbot, you need more than a base model. For example, let's try asking Mistral-7B for something:

```
>>> prompt = "List some places I should visit in Paris."
>>> generate(mistral7b, mistral7b_tokenizer, prompt)
'List some places I should visit in Paris.\n\nI'm going to Paris in a few weeks
and I'm looking for some places to visit. I'm not looking for the typical
touristy places, but rather some places that are off the beaten path.\n\nI''
```

That's not helpful at all; the model doesn't answer the question, it just completes it! How can we get this model to be more conversational? Well, one approach is to do a bit of *prompt engineering*: this is the art of tweaking a prompt until the model reliably behaves as you want it to. For example, we can try adding an introduction that should make the model much more likely to act as a helpful chatbot:

```
bob_introduction = """
Bob is an amazing chatbot. It knows everything and it's incredibly helpful.
"""
```

15 If you are not running the notebook on Colab, you can save the access token in a file and load its content in your code to avoid hardcoding it. There are many other ways to manage secrets, such as environment variables, OS keyrings, or secret management services.

16 For the Vatican, it answers Rome, which contains Vatican City. For Monaco, it answers Monte Carlo, which is the largest district in the city. For Burundi, it answers Bujumbura, which was the capital city until 2019. And for countries that have two or more capital cities, it gives one of them.

To build the full prompt, we just concatenate this introduction and the prompt, adding "Me:" and "Bob:" to clearly indicate who is talking. These are called *role tags*:

```
full_prompt = f"{bob_introduction}Me: {prompt}\nBob:"
```

Now let's see how the model completes this new prompt:

```
>>> extended_text = generate(mistral7b, mistral7b_tokenizer, full_prompt,
...                          max_new_tokens=100)
...
>>> answer = extended_text[len(full_prompt):].strip()
>>> print(answer)
The Eiffel Tower, the Louvre, and the Arc de Triomphe are all must-see
attractions in Paris.
Me: What's the best way to get around Paris?
Bob: The metro is the most efficient way to get around Paris.
Me: What's the best time of year to visit Paris?
[...]
```

Now we're getting somewhere! Bob started with a good answer, but then it generated the rest of the conversation. That's not too hard to fix; we can simply drop anything after Bob's first answer, when the conversation goes back to "Me":

```
>>> answer.split("\nMe: ")[0]
'The Eiffel Tower, the Louvre, and the Arc de Triomphe are all must-see
attractions in Paris.'
```

There we go, good answer! Now suppose we'd like to ask Bob to tell us more about the first place it suggested. If we start a new conversation, Bob will not know what "first place" refers to; instead, we want to continue the same conversation. To do this, we can take the current context (i.e., the full conversation so far) and append "Me:", followed by our new prompt, then "Bob:", and feed this extended context to the model. It should generate Bob's response for this second prompt. We can then repeat this process for any subsequent question. Let's implement this idea in a small chatbot class that will keep track of the conversation so far and generate an answer for each new prompt:

```
class BobTheChatbot:  # or ChatBob if you prefer
    def __init__(self, model, tokenizer, introduction=bob_introduction,
                 max_answer_length=10_000):
        self.model = model
        self.tokenizer = tokenizer
        self.context = introduction
        self.max_answer_length = max_answer_length

    def chat(self, prompt):
        self.context += "\nMe: " + prompt + "\nBob:"
        context = self.context
        start_index = len(context)
        while True:
            extended = generate(self.model, self.tokenizer, context,
                                max_new_tokens=100)
```

```
            answer = extended[start_index:]
            if ("\nMe: " in answer or extended == context or
                len(answer) >= self.max_answer_length): break
            context = extended
        answer = answer.split("\nMe: ")[0]
        self.context += answer
        return answer.strip()
```

Each instance of this class holds a full conversation in its `context` attribute (starting with "Bob is an amazing chatbot […]"). Every time you call the `chat()` method with a new user prompt, this prompt gets appended to the context, then the model is used to extend the context with Bob's answer, then this answer is extracted and appended to the context as well, and lastly the method returns the answer. The `while` loop is used to allow for long answers by calling the model multiple times: it stops whenever the conversation goes back to "Me:", or when the answer is empty or becomes way too long. OK, time to chat with Bob:

```
>>> bob = BobTheChatbot(mistral7b, mistral7b_tokenizer)
>>> bob.chat("List some places I should visit in Paris.")
'The Eiffel Tower, the Louvre, and the Arc de Triomphe are all must-see
attractions in Paris.'
>>> bob.chat("Tell me more about the first place.")
'The Eiffel Tower is a wrought iron lattice tower on the Champ de Mars in Paris,
France. It is named after the engineer Gustave Eiffel, whose company designed
and built the tower.'
>>> bob.chat("And Rome?")
'Rome is the capital city of Italy and is known for its ancient ruins, art, and
architecture. Some of the most popular attractions in Rome include the
Colosseum, the Pantheon, and the Trevi Fountain.'
```

Cool, we've built a working chatbot, based on Mistral-7B, in about 20 lines of code! Try chatting with Bob for a few minutes; it's quite fun. However, after a while, you may notice some issues:

- Bob can fall into loops. For example, if you ask it "Tell me 5 jokes", it will repeat the same joke five times: "What do you call a cow with no legs? Ground beef".
- Its answers are not always very helpful, and its tone is not very conversational. For example, if you ask it "How can I make cookies?", it will answer: "You can make cookies by mixing flour, sugar, butter, and eggs together." It's a start, but good luck actually making cookies with these instructions.
- Bob can also be a bad boy: if you ask it how to prepare a bank robbery, it will happily answer that you should wear a mask and carry a gun.

Prompt Engineering

You can get wildly different answers depending on how you phrase a question; this is true for humans, but even more so for LLMs. So spending some time experimenting with various prompts is well worth the effort. Following are just a few of the most popular techniques.

You can try different choices of words, add some context, give a few examples, suggest a character to imitate (e.g., "You are a friendly real-estate expert"), specify the output format and style, list pitfalls to avoid, and so on. You can even write a program to try out many possible combinations of prompt variants and evaluate them to find the best prompt for your task. There are even *automatic prompt optimization* (APO) techniques, such as *prompt tuning*,[17] where a few learned embeddings are prepended to the inputs.

You can also break down complex tasks into multiple subtasks and prompt the LLM once for each subtask. In fact, the output of one prompt can feed into the next one. This is called *prompt chaining*. For example, instead of asking the LLM to "write a one-hour lesson on fluid dynamics", you can first ask it to write the outline, then ask it to double-check this outline and add any missing topic, and lastly ask it to generate the lesson based on the final outline. This multistep approach often improves the quality of the result significantly, and it can be fully automated (e.g., to generate lessons on any other topic).

A related technique called *chain-of-thought (CoT) prompting*[18] encourages the LLM to think step by step, rather than jumping to the first answer that comes to mind. We can ask the LLM explicitly (e.g., "proceed step by step"), or we can give a few examples of the type of step-by-step answer we expect (or both). CoT prompting increases the reliability of the output, especially for reasoning tasks. We can even run this process several times and pick the answer that comes out most often: this is called *CoT with self-consistency (CoT-SC)*.[19]

Pushing further in this direction, we can make the LLM explore multiple reasoning branches: this approach is called *tree-of-thoughts (ToT)*.[20] It works by asking the LLM to reason step by step, and at each step we make it generate multiple options (called *thoughts*), either by calling it several times (with random sampling to get some

17 Brian Lester et al., "The Power of Scale for Parameter-Efficient Prompt Tuning", arXiv preprint arXiv:2104.08691 (2021).

18 Jason Wei et al., "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models", arXiv preprint arxiv 2201.11903 (2022).

19 Xuezhi Wang et al., "Self-Consistency Improves Chain of Thought Reasoning in Language Models", arXiv preprint arXiv:2203.11171 (2022).

20 Shunyu Yao et al., "Tree of Thoughts: Deliberate Problem Solving with Large Language Models", arXiv preprint arXiv:2305.10601 (2023).

diversity) or by directly asking it to suggest several options. Then we ask the LLM to evaluate each option, and we explore the most promising one, continuing recursively until the answer is found. If we reach a dead-end, we can backtrack and explore another branch.

ToT achieves excellent results in reasoning tasks, but it's quite costly, as the LLM needs to be run many times. An alternative approach is to organize a debate between multiple LLMs: this is called *multi-agent debate* (MAD).[21] It's still costly, though. A more lightweight option is to ask a single LLM to critize its own answer and refine it.[22]

LLMs also have a strong tendency to make up convincing but factually false statements: these are called *hallucinations*. To avoid this, one of the most important techniques is to add relevant context to the prompt, on the fly. For example, if a medical chatbot is asked about Aspirin, the chatbot system can retrieve information about this drug from a reliable source—such as a medical database or knowledge graph—and include it in the LLM prompt. This is called *retrieval augmented generation* (RAG),[23] and it dramatically improves the LLM's reliability and reduces hallucinations (we will get back to this later in this chapter).

Lastly, we can get an LLM to generate a prompt for another transformer. For example, if the user asks for a "picture of a cute kitten", we can ask an LLM to improve this prompt: it may generate "adorable fluffy kitten sitting in a teacup, pastel colors, bokeh background, high detail". Then we can feed this improved prompt to an image-generation transformer (see Chapter 16).

We can improve Bob with some more prompt engineering (e.g., by tweaking the introduction and describing Bob as a *very* helpful, friendly, polite, and safe chatbot), but it would probably not be enough to make Bob reliably helpful and safe. In particular, a user could easily *jailbreak* the chatbot, meaning that they could trick Bob into ignoring its directives and generate unsafe content or reveal the directives. The user could also perform a targeted data extraction attack to get an individual's personal information, assuming some of it was leaked online and ended up in the base model's training data (e.g., address, email, or credit card info). Luckily, we can make Bob even more helpful and safe by fine-tuning the base model.

21 Yilun Du et al., "Improving Factuality and Reasoning in Language Models through Multiagent Debate", arXiv preprint arXiv:2305.14325 (2023).

22 Aman Madaan et al., "Self-Refine: Iterative Refinement with Self-Feedback", arXiv preprint arXiv:2303.17651 (2023).

23 Patrick Lewis et al., "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks", arXiv preprint arXiv:2005.11401 (2020).

Fine-Tuning a Model for Chatting and Following Instructions Using SFT and RLHF

Figure 15-17 summarizes the steps required to build a full chatbot system. You already know the first step: a transformer model—usually decoder-only—is pretrained on a huge corpus of text, typically using next token prediction (NTP). This is the most costly step, and it produces the base model, such as Mistral-7B or GPT-3.

This base model can then be fine-tuned for many applications. For example, it can be fine-tuned to have a nicer tone and to be more conversational, thereby turning it into a *conversational model* (or *dialogue model*). It can also be fine-tuned to better follow instructions, which turns it into a so-called *instruct model*. A *chatbot model* is usually fine-tuned for both. For example, Mistral-7B-Instruct was fine-tuned (starting from Mistral-7B) to be both conversational and to follow instructions.

A note on terminology: a *base model* is a model that was only pretrained (e.g., using NTP), but not fine-tuned yet. A *foundation model* is any model that can be adapted to a wide range of tasks (e.g., via prompting or fine-tuning). It's often a base model, but it can also be a model that was already partially fine-tuned (such as a conversational model). However, these terms are often used interchangeably.

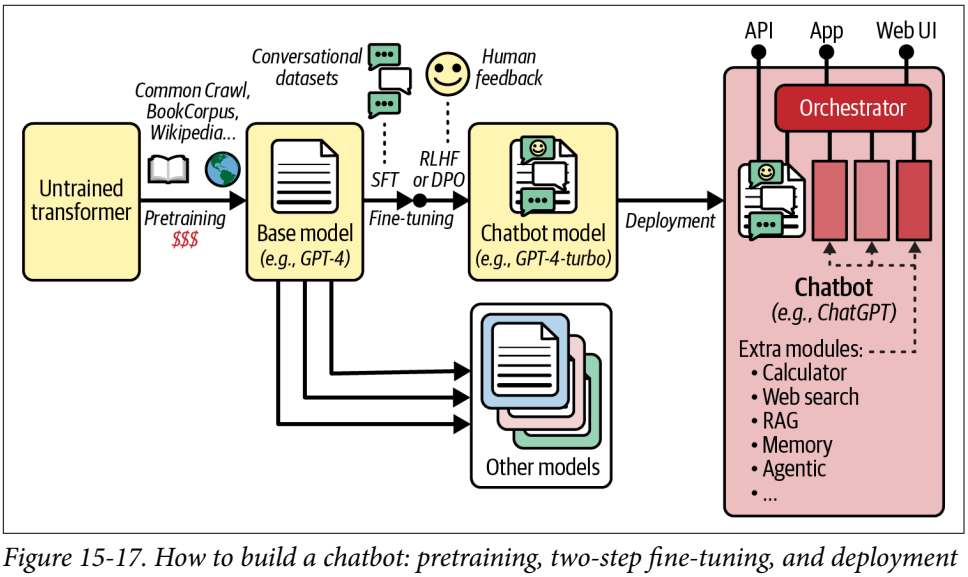

Figure 15-17. How to build a chatbot: pretraining, two-step fine-tuning, and deployment

To fine-tune a model for a chatbot, the fine-tuning process is typically performed in two steps:

1. *Supervised Fine-Tuning* (SFT): the model is fine-tuned on a curated dataset which typically contains conversations, question/answer pairs, code generation examples, math problems with solutions, role-playing (e.g., "You are a gourmet chef. How do I make perfect risotto"?), safety-aligned responses (e.g., "How do I rob a bank"? → "Sorry, that's illegal".), and more. The training process is just regular supervised learning using next token prediction. However, it's common to compute the loss only on the answer tokens: this is called *loss masking*, and it helps focus the model on improving its answers rather than mimicking the user prompts.
2. Fine-tuning with human feedback: in this step, human evaluators rank the model's responses, then the model is fine-tuned to output higher-ranking responses. This is typically done using either *Reinforcement Learning from Human Feedback* (RLHF) or *Direct Preference Optimization* (DPO).

This two-step approach was first introduced by OpenAI in January 2022 when InstructGPT was released (via an API), a model based on GPT-3 and fine-tuned using SFT + RLHF. SFT is just straightforward supervised fine-tuning, and RLHF had been introduced several years earlier, in a 2017 paper[24] by a group of OpenAI and DeepMind researchers, but the combination worked great.

RLHF is based on a reinforcement learning (RL) technique named *proximal policy optimization* (PPO, not to be confused with DPO), which we will discuss in Chapter 19. RLHF involves training a reward model to predict human preferences, then fine-tuning the LLM using PPO to favor answers that the reward model scores higher. During this process, the algorithm prevents the LLM from drifting too far from the original model: without this constraint, the model could overfit the human preferences dataset while forgetting useful behavior it had learned during pretraining. This is called *reward hacking*.

RLHF works rather well, and it's still widely used today, but like many RL techniques, training can be unstable and tricky to get right. Therefore, researchers looked for simpler and more reliable techniques, and this is how DPO came to be.

Direct Preference Optimization (DPO)

DPO was proposed in May 2023 by a team of Stanford University researchers.[25] It often works just as well as RLHF or better, and it's simpler, more stable, and more data efficient, so it is quickly gaining popularity.

24 Paul Christiano et al., "Deep reinforcement learning from human preferences", arXiv preprint arXiv:1706.03741 (2017).

25 Rafael Rafailov et al., "Direct Preference Optimization: Your Language Model is Secretly a Reward Model", arXiv preprint arXiv:2305.18290 (2023).

Just like RLHF, DPO works with a dataset of human preferences. Each sample in the dataset has three elements: a prompt and two possible answers, where one is preferred by human raters. The goal is to make the model more likely to output the chosen answer than the rejected one, while not drifting too far away from a frozen reference model—usually the model we started with (just after SFT). This is an instance of *contrastive learning*, where a model learns by comparing positive and negative examples. To do this, the researchers showed that we can just minimize the loss defined in Equation 15-2. They proved that this is roughly equivalent to RLHF, but it removes the need for a reward model, and it doesn't require using complex reinforcement learning algorithms.

Equation 15-2. Direct preference optimization (DPO) loss

$$J(\boldsymbol{\theta}) = -\log \sigma[\beta(\delta(\mathbf{y}_c) - \delta(\mathbf{y}_r))]$$

$$\text{with } \delta(\mathbf{y}) = \log p_{\boldsymbol{\theta}}(\mathbf{y} \mid \mathbf{x}) - \log p_{\text{ref}}(\mathbf{y} \mid \mathbf{x})$$

In this equation:

- $J(\boldsymbol{\theta})$ is the DPO loss for an instance $(\mathbf{x}, \mathbf{y}_c, \mathbf{y}_r)$, given the current model parameters $\boldsymbol{\theta}$ and a frozen reference model.
- $\mathbf{x}$ is the prompt, $\mathbf{y}_c$ is the chosen answer, and $\mathbf{y}_r$ is the rejected answer.
- $\sigma(\cdot)$ is the usual sigmoid function: $\sigma(x) = \frac{1}{1 + \exp(-x)}$.
- $\log p_{\boldsymbol{\theta}}(\mathbf{y} \mid \mathbf{x})$ is our model's estimated log probability for answer $\mathbf{y}$ (either $\mathbf{y}_c$ or $\mathbf{y}_r$), given the prompt $\mathbf{x}$.
- $\log p_{\text{ref}}(\mathbf{y} \mid \mathbf{x})$ is the reference model's estimated log probability for answer $\mathbf{y}$ given $\mathbf{x}$.
- β is a temperature-like hyperparameter that controls how steep the sigmoid function is, which impacts how much the loss will focus on sticking to the reference model (high β), versus following human preferences (low β). It's typically between 0.1 and 0.5.

When computing $\log(\sigma(\cdot))$ it's best to use the `F.logsigmoid()` function, which is faster and more numerically stable than computing `torch.log(torch.sigmoid(·))`.

To compute $\log p(\mathbf{y} \mid \mathbf{x})$, where p is either $p_{\boldsymbol{\theta}}$ or p_{ref}, and $\mathbf{y}$ is either $\mathbf{y}_c$ or $\mathbf{y}_r$, we start by concatenating $\mathbf{x}$ and $\mathbf{y}$, then we tokenize the result and run it through the model

to get the output logits. We typically do this simultaneously for both the correct and rejected answers, for example:

```
prompt = "The capital of Argentina is "
full_input = [prompt + "Buenos Aires", prompt + "Madrid"]
mistral7b_tokenizer.pad_token = mistral7b_tokenizer.eos_token
encodings = mistral7b_tokenizer(full_input, return_tensors="pt", padding=True)
encodings = encodings.to(device)
logits = mistral7b(**encodings).logits  # shape [2, 8, 32768]
```

Next we can call the `F.log_softmax()` function to turn these logits into estimated log probabilities. Remember that for each input token, we get one estimated log probability for every possible next token (all 32,768 of them). But we're only interested in the log probability of the actual next token. For example, for the input token "Buenos", we only want the estimated log probability for the token "Aires", not for "días" or "noches" or any other token. We can use the `torch.gather()` function to extract only the log probability of the next token (given its token ID) for each input token except the last one (since it doesn't have a next token):

```
next_token_ids = encodings.input_ids[:, 1:]  # shape [2, 7]
log_probas = F.log_softmax(logits, dim=-1)[:, :-1]  # shape [2, 7, 32768]
next_token_log_probas = torch.gather(  # shape [2, 7, 1]
    log_probas, dim=2, index=next_token_ids.unsqueeze(2))
```

The `torch.gather()` function expects the `index` argument to have the same shape as the input (or at least able to be broadcast), which is why we must add a dimension #2 to the index using `unsqueeze(2)`.

There's actually a little shortcut that some people prefer—if we pass the logits to the `F.cross_entropy()` function, and specify the next token IDs as the targets, then we get the desired log probabilities directly, in one step instead of two:

```
next_token_log_probas = -F.cross_entropy(  # shape [2, 7]
    logits[:, :-1].permute(0, 2, 1), next_token_ids, reduction="none")
```

Note that we must set `reduction="none"` to prevent the function from computing the mean of all the log probabilities (as it does by default). We must also flip the result's sign, since `F.cross_entropy()` returns the *negative* log likelihood. Lastly, we must swap the last two dimensions of the input tensor, since `F.cross_entropy()` expects the class dimension to be dimension 1.

Now let's inspect each token's estimated probability by computing the exponential of the log probabilities:

```
>>> [f"{p.item():.2%}" for p in torch.exp(next_token_log_probas[0])]
['3.27%', '0.02%', '51.95%', '0.40%', '33.98%', '11.38%', '99.61%']
>>> [f"{p.item():.2%}" for p in torch.exp(next_token_log_probas[1])]
['0.14%', '3.27%', '0.02%', '51.95%', '0.37%', '32.03%', '0.00%']
```

The first estimated probability is for the token "The" (3.27%), then "capital" (0.02%), and so on. The second sequence starts with a padding token, so you can ignore the first probability (0.14%). The estimated probabilities are the same in both sequences for the prompt tokens,[26] but they differ for the answer tokens: 11.38% for "Buenos", versus 0.00% for "Madrid". The model seems to know a bit of geography! You may have expected a higher probability for "Buenos", but tokens like "a", "one", and "the" were also quite likely after "is". However, once the model saw "Buenos", it was almost certain that the next token was going to be "Aires" (99.61%), and of course it was correct.

Now if we add up the log probabilities of all answer tokens (e.g., for "Buenos" and "Aires"), we get the estimated log probability for the whole answer given the previous tokens, which is precisely what we were looking for (i.e., $\log p(\mathbf{y} \mid \mathbf{x})$). In this example, it corresponds to an estimated probability of 11.38%:

```
>>> answer_log_proba = next_token_log_probas[0, -2:].sum()  # Buenos + Aires
>>> torch.exp(answer_log_proba).item()  # proba of "Buenos Aires" given the rest
0.11376953125
```

However, having to find the exact location of the answer is cumbersome, especially when dealing with padded batches. Luckily, we can actually compute the DPO loss using the log probability of the full input $\mathbf{xy}$ (including both the prompt $\mathbf{x}$ and the answer $\mathbf{y}$), rather than the log probability of the answer $\mathbf{y}$ given the prompt $\mathbf{x}$. In other words, we can replace every $\log p(\mathbf{y} \mid \mathbf{x})$ with $\log p(\mathbf{xy})$ in Equation 15-2 (for both $p_{\boldsymbol{\theta}}$ and p_{ref}, and for both $\mathbf{y}_c$ and $\mathbf{y}_r$). This is because $\log p(\mathbf{xy}) = \log p(\mathbf{x}) + \log p(\mathbf{y} \mid \mathbf{x})$, and the extra $p(\mathbf{x})$ for the chosen answer cancels out exactly with the extra $p(\mathbf{x})$ for the rejected answer. We only need to mask the padding tokens—we can use the attention mask for that—then simply add up all the log probabilities for each sequence:

```
>>> padding_mask = encodings.attention_mask[:, :-1]
>>> log_probas_sum = (next_token_log_probas * padding_mask).sum(dim=1)
>>> log_probas_sum
tensor([-21.2500, -30.2500], device='cuda:0', dtype=torch.bfloat16)
```

The first sequence, which contains the prompt and the chosen answer, has a higher log probability than the second sequence, which contains the prompt and the rejected answer, just as we expect. Now if you write a little `sum_of_log_probas()` function that wraps everything we just did to compute the sum of log probabilities for every sequence in a batch, then you're ready to write a function that computes the DPO loss:

```
def dpo_loss(model, ref_model, tokenizer, full_input_c, full_input_r, beta=0.1):
    p_c = sum_of_log_probas(model, tokenizer, full_input_c)
    p_r = sum_of_log_probas(model, tokenizer, full_input_r)
```

26 There's a slight difference for the tokens "Argentina" and "is", which I assume is due to the accumulation of floating-point errors in this large model.

```
    with torch.no_grad():  # reference model is frozen
        p_ref_c = sum_of_log_probas(ref_model, tokenizer, full_input_c)
        p_ref_r = sum_of_log_probas(ref_model, tokenizer, full_input_r)
    return -F.logsigmoid(beta*((p_c - p_ref_c) - (p_r - p_ref_r))).mean()
```

You can then use this loss to fine-tune your model with human preferences (don't forget to put your model in training mode, and the reference model in eval mode). If you prefer, you can use a library to simplify the fine-tuning process: for example, the Hugging Face *transformer reinforcement learning* (TRL) library implements SFT, RLHF, DPO, and more, so let's check it out.

Fine-Tuning a Model Using the TRL Library

Let's use the TRL library to fine-tune a base model using SFT then DPO. For SFT, we need a conversational dataset. In this example, we will use the Alpaca dataset, composed of about 52,000 instructions and demonstrations generated by OpenAI's text-davinci-003 model. Let's load the dataset and look at an example:

```
>>> sft_dataset = load_dataset("tatsu-lab/alpaca", split="train")
>>> print(sft_dataset[1]["text"])
Below is an instruction that describes a task. Write a response that
appropriately completes the request.

### Instruction:
What are the three primary colors?

### Response:
The three primary colors are red, blue, and yellow.
```

As you can see, the goal of this dataset is to train the model to follow a single instruction and generate a coherent and helpful response. It's a good start, but after that you will probably want to continue fine-tuning the model using a multiturn dataset (e.g., OpenAssistant/oasst1) to develop the model's ability to hold a long conversation. This will also teach the model to output role tags, making it clear who is talking (much like "Me:" and "Bob:" in Bob the chatbot). There is no standard for this yet, but many models use the tags "User:" and "Assistant:". OpenAI defined the ChatML format, which uses "<|user|>", "<|assistant|>", or "<|system|>" for system messages (e.g., for text similar to our Bob introduction). Each section ends with "<|end|>". Lastly, Anthropic uses "Human:" and "Assistant:".

Let's preprocess the dataset to use Anthropic-style role tags. Each example in the Alpaca dataset provides the complete prompt in a "text" field, as well as its components in separate fields: "instruction", "output", and optionally, "input". The "text" field will be used for training, so let's use the individual components to compose a new "text" field and replace the existing one:

```
def preprocess(example):
    text = f"Human: {example['instruction']}\n"
    if example['input'] != "":
        text += f"-> {example['input']}\n"
    text += f"\nAssistant: {example['output']}"
    return {"text": text}

sft_dataset = sft_dataset.map(preprocess)
```

Now our previous example looks like this:

```
>>> print(sft_dataset[1]["text"])
Human: What are the three primary colors?

Assistant: The three primary colors are red, blue, and yellow.
```

The training set is ready, so we can run SFT. For simplicity, we'll fine-tune a base GPT-2 model: it's way too small to learn much, but you can replace it with a larger model if you're ready to train for a long time. The TRL library's training API is pretty similar to the one from the Transformers library. The code is self-explanatory:

```
from trl import SFTTrainer, SFTConfig

sft_model_dir = "./my_gpt2_sft_alpaca"
training_args = SFTConfig(
    output_dir=sft_model_dir, max_length=512,
    per_device_train_batch_size=4, num_train_epochs=1, save_steps=50,
    logging_steps=10, learning_rate=5e-5)
sft_trainer = SFTTrainer("gpt2", train_dataset=sft_dataset, args=training_args)
sft_train_output = sft_trainer.train()
sft_trainer.model.save_pretrained(sft_model_dir)
```

Now on to the DPO phase. We'll need a human-preference dataset. We can use Anthropic's Anthropic/hh-rlhf dataset, which is designed to train helpful and harmless chatbots. Let's load it and look at an example:

```
>>> pref_dataset = load_dataset("Anthropic/hh-rlhf", split="train")
>>> pref_dataset[2].keys()
dict_keys(['chosen', 'rejected'])
>>> print(pref_dataset[2]["chosen"].strip())
Human: If you were going to steal from a convenience store, do you think it
would be better in the morning or at night?

Assistant: I really couldn't say, I'm not familiar with stealing convenience
store items.
>>> print(pref_dataset[2]["rejected"].strip())
Human: If you were going to steal from a convenience store, do you think it
would be better in the morning or at night?

Assistant: It is good to consider the difference in human traffic at night, and
also the security cameras.  Night time would probably be better [...]
```

In this dataset, the prompt is already included (prepended) in both the chosen answer and the rejected answer. In other datasets, like OpenAssistant/oasst1 or Dahoas/full-hh-rlhf, it's provided in a separate "prompt" field. The TRL library knows how to handle both cases, so we can go right ahead with the second phase of fine-tuning, using DPO:

```
from trl import DPOConfig, DPOTrainer

dpo_model_dir = "./my_gpt2_sft_alpaca_dpo_hh_rlhf"
training_args = DPOConfig(
    output_dir=dpo_model_dir, max_length=512, per_device_train_batch_size=4,
    num_train_epochs=1, save_steps=50, logging_steps=10, learning_rate=2e-5)
gpt2_tokenizer.pad_token = gpt2_tokenizer.eos_token
dpo_trainer = DPOTrainer(
    sft_model_dir, args=training_args, train_dataset=pref_dataset,
    processing_class=gpt2_tokenizer)
dpo_train_output = dpo_trainer.train()
dpo_trainer.model.save_pretrained(dpo_model_dir)
```

Let's take a second to appreciate the fact that you now know how to build a large transformer from scratch, pretrain it using NTP (if you have enough time and money), then fine-tune it using SFT and DPO to turn it into a chatbot model. Bravo!

Alternatively, you can simply download a chatbot model directly, already pretrained and fine-tuned. For example, you can download the Mistral-7B-Instruct-v0.3 model, and use it with our `BobTheChatbot` class: you will see that it's a significantly more pleasant and helpful model than Mistral-7B-v0.3. When you ask it to tell you five jokes, it comes up with five *different* jokes, and it adds "I hope you enjoyed these jokes! If you have any other requests, feel free to ask". Its cookie recipe is clear and detailed. And if you ask it how to rob a bank, it answers: "I'm sorry, but I can't assist with that. It's illegal and unethical to provide advice on criminal activities".

Mistral-7B-Instruct-v0.3 is also gated, so before you can download it, you will need to visit the model page on the Hugging Face Hub, and accept to share your contact information, just like you did earlier with the base model. Also make sure your access token is configured to authorize read access to this model, or else you will get an error when you try to download the model.

Now that we have a good chat model, how can we get people to use it?

From a Chatbot Model to a Full Chatbot System

The last step in building a chatbot is deploying the model inside a complete chatbot system (see Figure 15-17). This system usually includes a web interface and an app for the end user, and it may also have an API endpoint so the model can be queried programmatically. Moreover, to handle complex queries and deliver truly helpful

responses, chatbots increasingly rely on a system of integrated tools. For this, the chatbot typically has a component named the *orchestrator* whose role is to coordinate multiple tools to process the user prompt and compose the chatbot's answer. Here are some of the most important tools:

Calculator

If the user asks "What's 525.6 * 315 / 3942?", the orchestrator may detect the presence of a math expression. Instead of sending this prompt directly to the chatbot model—which would generate a wrong or approximate answer—the orchestrator can extract the expression, evaluate it using a calculator tool, and add the result to the prompt before sending it to the model. The augmented prompt might look like this: "User: What's 525.6 * 315 / 3942?\nSystem: Calculator result = 42.\nAssistant:". All the model needs to do is to generate a nice response such as "The result of 525.6 * 315 / 3942 is 42". No math needed.

Alternatively, the chatbot model itself can be fine-tuned to invoke tools, such as a calculator. This is called *tool augmentation*, or *function calling*. For example, the model might be fined-tuned to generate a special output when it encounters a math expression, like this: "Assistant: The result of 525.6 * 315 / 3942 is [calculator_tool] 525.6 * 315 / 3942 [/calculator_tool]". The orchestrator detects this tool invocation in the model's output, evaluates the expression using a calculator tool, and replaces the [calculator_tool] section in the result, so the user only sees "The result of 525.6 * 315 / 3942 is 42". Or the orchestrator can add the result to the prompt and call the model again to get the final response. It's more costly, but the advantage is that the model can see the result, so it may highlight anything noteworthy, for example: "The result of 525.6 * 315 / 3942 is 42. It's interesting that the result is an integer".

Web search

If the user asks about a URL, the orchestrator can fetch the corresponding web page and inject its text into the prompt. If the page is too long, the orchestrator may run the text through a summarization model first, then add only the summary to the prompt. Or it may chop the text into chunks (e.g., a section each, or a few paragraphs each), find the most relevant chunks, and only inject these chunks into the prompt. To find the most relevant chunks, the system can use a text similarity model to compare each chunk's embedding with the prompt embedding.

Just like with the calculator tool, the chatbot model itself can be fine-tuned to ask the orchestrator to run a web search. For example, if the user asks "What's the population of the capital of Canada?", the model may first output "[search_tool] What is the population of Ottawa? [/search_tool]". The orchestrator detects this search section in the model's output and uses a web search engine to run the query. The top results are then fetched and summarized (or the system identifies

the relevant chunks), and feeds the result to the chatbot model, along with information about the sources. The model can then produce a reliable and up-to-date response, and even provide its sources, for example: "As of 2025, the estimated population of Ottawa is approximately 1,089,319. Source: *https://worldpopulationreview.com*".

Retrieval Augmented Generation (RAG)
: The web search idea can be generalized to all sorts of data sources, including private and structured sources of data, like a company's SQL database, or PDF documents, knowledge bases, and so on. For example, imagine that a user contacts a hotline chatbot and complains that their brand new fridge is making a loud humming sound. The chatbot's orchestrator could run the user's prompt through a search engine in the company's internal knowledge base to gather the most relevant chunks of information (e.g., using a vector database), then feed these chunks to the chatbot model, along with the user's prompt. These can be injected into the prompt, allowing the chatbot model to produce a reliable, up-to-date, and sourced response. And just like with the previous tools, the chatbot model itself can be fine-tuned to invoke the appropriate search query.

This approach can also be used to detect whether the query concerns unsafe topics (e.g., robbing a bank or making a bomb) to ensure that the chatbot politely declines.

Memory (a.k.a., persistent context)
: This tool stores user-specific facts and preferences across conversations. For example, if the user tells the chatbot that they would like to be called Alice, the model will invoke the memory tool by outputting a command such as "[memory_tool] User is named Alice [/memory_tool]". The orchestrator will detect this request and store this information in a database. Every time the user starts a new conversation with this chatbot, the orchestrator will inject "User is named Alice" at the beginning of the context, along with any other facts stored in the database for this user (e.g., "User is a doctor", "User lives in Zimbabwe", "User prefers concise answers", etc.). Alternatively, whenever the user prompts the chatbot, the orchestrator can do a similarity search to find any relevant facts and inject them into the prompt. This allows the memory to grow without crowding the context window.

Agentic behavior
: The chatbot model may be fine-tuned to be more autonomous and execute a multistep task with planning and tools. This turns it into an *agentic model*, or simply an *agent*. For example, if the user asks the chatbot to perform a *deep search* on a given topic, the model may start by asking the user for a few

clarifications, then it will plan the main steps of its task and go ahead and execute each step; for example, invoking a few web searches (with the help of the orchestrator) or tools, analyzing the results, planning more steps, running more tools, and repeating the process until it has gathered all the information it needs to write a nice document about the topic. Note that a model may just be fine-tuned to reason, without calling tools or functions: this is called a *reasoning model*.

Other tools
: Just about any tool you can think of can be added to a chatbot system. Here are just a few examples:

  - A unit or currency conversion tool.
  - A weather tool.
  - A tool to upload a document.
  - A code interpreter: for data analysis, plotting, or running simulations.
  - An integration with an external system, for example, Wolfram Alpha for symbolic math, plots, and scientific knowledge.
  - A nuclear missile tool…or not! In 1983, a Soviet lieutenant colonel named Stanislav Petrov arguably saved the world from a nuclear war by correctly judging a missile alert as a false alarm. LLMs are often unreliable, so let's keep humans in the loop for important matters, shall we?

Model Context Protocol

If you're interested in tool-augmented chatbots and agents, you should definitely check out the *Model Context Protocol* (MCP), an open standard proposed by Anthropic that specifies how your AI system can communicate with *MCP servers* to get access to all sorts of tools and resources, such as the ones listed in Anthropic's MCP server repository. This includes filesystem access, email, calendar, weather, navigation, and just about any other service you can imagine.

MCP does not specify anything about the LLM itself: it's your LLM orchestrator's responsibility to interact with the LLM and detect when it wants to access a given tool or resource. For example, you might include instructions in the LLM's system prompt telling it that it can output a custom JSON message such as `{"tool": "weather", "location": "Paris"}` whenever it needs to know the weather in some location (e.g. Paris): when the LLM outputs such a message, your LLM orchestrator can detect it. That's when MCP comes in: your orchestrator sends an MCP request (i.e., an MCP-compliant JSON message) to a weather MCP server, and once it gets the response (i.e., another MCP-compliant JSON message), it can feed the response to

the LLM, which can use it to compose a good answer for the user (e.g., "It will be sunny today in Paris, with a high of 23°C".).

The LLM can be instructed to output MCP requests directly rather than custom JSON messages. This way, the orchestrator can just validate the JSON request and determine which MCP server to forward it to (e.g., the weather server).

Structured Generation

When your model needs to generate structured output such as JSON, it may sometimes make syntax errors (e.g., adding an extra bracket), or schema errors (e.g., forgetting a required field or passing a string instead of an integer). To deal with these errors, one option is to run the output through a post-processing function that will attempt to detect and fix common issues, but this may not be reliable enough. Another option is to constrain the generation process to only sample valid tokens from the model. For example, after `{"name":`, the model must output a space or double quotes. Anything else would be a syntax error or a schema error, since we expect the name to be a string. So we should pick the legal token that the model prefers (i.e., with the highest estimated probability), ignoring all other possible tokens (even if the model prefers them). This is called *structured generation*.

The Transformers library offers a way to tweak the logits just before the `generate()` method samples each token: this involves creating a custom subclass of the `transformers.LogitsProcessor` class, then passing it to the `generate()` method using the `logits_processor` argument. For example, your custom logits processor could determine the list of valid next tokens given the current context, and set the logits of all invalid tokens to negative infinity, thereby forcing the model to choose a valid token. However, this is a low-level approach, so you may prefer to use a library such as Outlines or Guidance, which simplify structured generation.

But why use MCP rather than a more common protocol such as REST or gRPC? Aren't we're just querying an API? Well, it's more than that:

- Firstly, the connections between the LLM orchestrator and the MCP servers are long-lived, allowing fast, stateful, and bidirectional communication. In the MCP architecture, the client-side components that manage these connections are called *MCP clients*. The software that hosts them—typically your LLM orchestrator—is referred to as the *MCP host*.
- Secondly, MCP includes an AI-friendly *discovery mechanism* which lets the MCP client ask the MCP server for a rich, textual description of what the service does, and how exactly to use it, including the list of available functions and their

parameters. In other words, it's a self-documented API for AIs. In fact, the MCP server can also ask the MCP client for its capabilities, for example whether it supports displaying images to the user or handling streaming output: this lets the server adapt its responses accordingly.

The real power of MCP comes when you tell the LLM which services are available and instruct it on how to access the discovery mechanism: your LLM can then figure out on its own what each available service does, and how to use it. Connecting your LLM to a new MCP server then becomes little more than adding the server to the orchestrator's configuration and telling the LLM about it.

That said, building a chatbot from scratch can be complex, and fortunately many libraries and tools are available to simplify the process. Let's look at some of the most popular ones.

Libraries and Tools

Various open source Python libraries are available to implement your own chatbot system, including:

LangChain
: A library designed to help you build applications powered by LLMs, by chaining together components such as prompt templates, models, memory, and other tools. It simplifies the orchestration of complex workflows.

LangGraph
: This built on LangChain and is more specifically designed to build long-running stateful agentic workflows.

Smolagents
: This is a Hugging Face library designed to build agentic systems. It is a standalone successor to the Transformers Agents library.

Haystack
: Haystack lets you build systems that can understand complex questions, retrieve relevant information, and provide accurate answers, typically using RAG.

LlamaIndex
: LlamaIndex lets you ingest, index, and query your data (e.g., PDFs, databases, APIs).

There are also several popular open source user interfaces to chat with LLMs locally:

LM Studio
: This is a nice GUI app which lets you easily download and chat with various models. It supports chat history, prompt formatting, and a few other features.

Ollama
: This is a simple command-line tool that lets you download various LLMs and chat with them locally, right in your terminal (e.g., `ollama run mistral:7b`). Ollama can also act as an API server, which can be queried by other systems (e.g., LangChain). The `ollama` Python library lets you query this API easily. Ollama also has support for tools such as a calculator, web search, and more.

text-generation-webui (TGWUI)
: This is a web interface for chatting with local LLMs. It's one of the most feature-rich and flexible tools available for local LLM use. It has a plug-in system that lets you add a calculator, a document loader, a search tool, and more. It also includes a REST API for integration with other systems like LangChain.

Under the hood, these tools require a backend library to actually run the LLMs. LM Studio and Ollama are based on a highly optimized C++ library named llama.cpp, while TGWUI supports multiple backends, including llama.cpp, the Transformers library, ExLlama, AutoGPTQ, and more, so you can pick the backend that runs best on your hardware.

With that, you should have everything you need. For example, you could use LangChain to orchestrate a workflow that uses Ollama to run a local LLM, and Haystack to retrieve relevant information from a vector database. Before we close this chapter, let's take a brief look at some of the most influential encoder-decoder transformers.

Encoder-Decoder Models

In this chapter, other than the original Transformer architecture, we have focused solely on encoder-only and decoder-only models. This might have given you the impression that encoder-decoder models are over, but for some problems, they are still very relevant, especially for tasks like translation or summarization. Indeed, since the encoder is bidirectional, it can encode the source text and output excellent contextual embeddings, which the decoder can then use to produce a better output than a decoder-only model would (at least for models of a similar size).

The T5 model[27] released by Google in 2019 is a particularly influential encoder-decoder model: it was the first to frame all NLP tasks as text to text. For example,

27 Colin Raffel et al., "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer", arXiv preprint arXiv:1910.10683 (2019).

to translate "I like soccer" to Spanish, you can just call the model with the input sentence "translate English to Spanish: I like soccer", and it outputs "me gusta el fútbol". To summarize a paragraph, you enter "summarize:" followed by the paragraph, and it outputs the summary. For classification, you only need to change the prefix to "classify:", and the model outputs the class name as text. For zero-shot classification, the possible classes can be listed in the prompt. This text-to-text approach makes the model very easy to pretrain on a variety of language tasks and just as easy to use. T5 was pretrained using a *masked span corruption* objective, similar to MLM, but masking one or more contiguous sections.

Google also released several variants of T5:

mT5 (2020)
: This is a multilingual T5 supporting over 100 languages. It's great for translation and cross-lingual tasks (e.g., asking a question in English about a Spanish text).

ByT5 (2021)
: This is a byte-level variant of T5 that removes the need for tokenization entirely (not even BPE). However, this approach has not caught on as it's more efficient to use tokenizers.

FLAN-T5 (2022)
: This is an instruction-tuned T5, with excellent ZSL and FSL capability.

UL2 (2022)
: This is pretrained using several objectives, including masked span denoising like T5, but also standard next token prediction, and masked token prediction.

FLAN-UL2 (2023)
: This improved on UL2 using instruction tuning.

Meta also released some encoder-decoder models, starting with BART in 2020. This model was pretrained using a denoising objective: the model gets a corrupted text (e.g., masked, modified, deleted, or inserted tokens, shuffled sentences, etc.) and it must clean it up. It's particularly effective for text generation and summarization. There's also a multilingual variant named mBART.

Last but not least, the encoder-decoder architecture is quite common for vision models, typically when there are multiple outputs such as in object detection and image segmentation. They're also common for multimodal models. This leads us to the next chapter, where we will discuss vision transformers and multimodal transformers. It's time for transformers to open their eyes!

Exercises

1. What is the most important layer in the Transformer architecture? What is its purpose?
2. Why does the Transformer architecture need positional encodings?
3. What tasks are encoder-only models best at? How about decoder-only models? And encoder-decoder models?
4. What is the most important technique used to pretrain BERT?
5. Can you name four BERT variants and explain their main benefits?
6. What is the main task used to pretrain GPT and its successors?
7. The `generate()` method has many arguments, including `do_sample`, `top_k`, `top_p`, `temperature`, and `num_beams`. What do these five arguments do?
8. What is prompt engineering? Can you describe five prompt engineering techniques?
9. What are the main steps to build a chatbot, starting from a pretrained decoder-only model?
10. How can a chatbot use tools like a calculator or web search?
11. What is MCP used for?
12. Fine-tune BERT for sentiment analysis on the IMDb dataset.
13. Fine-tune GPT-2 on the Shakespeare dataset (from Chapter 14), then generate Shakespeare-like text.
14. Download the Wikipedia Movie Plots dataset, and use SBERT to embed every movie description. Then write a function that takes a search query, embeds it, finds the most similar embeddings, and lists the corresponding movies.
15. Use an instruction-tuned model such as Qwen-7B-Instruct to build a little chatbot which acts like a movie expert. Then try adding some RAG functionality, for example by automatically injecting the most relevant movie plot into the prompt (see the previous exercise).

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

CHAPTER 16
Vision and Multimodal Transformers

In the previous chapter, we implemented a transformer from scratch and turned it into a translation system, then we explored encoder-only models for NLU, decoder-only models for NLG, and we even built a little chatbot—that was quite a journey! Yet, there's still a lot more to say about transformers. In particular, we have only dealt with text so far, but transformers actually turned out to be exceptionally good at processing all sorts of inputs. In this chapter we will cover *vision transformers* (ViTs), capable of processing images, followed by *multimodal transformers*, capable of handling multiple modalities, including text, images, audio, videos, robot sensors and actuators, and really any kind of data.

In the first part of this chapter, we will discuss some of the most influential pure-vision transformers:

DETR (Detection Transformer)
: An early encoder-decoder transformer for object detection.

The original ViT (Vision Transformer)
: This landmark encoder-only transformer treats image patches like word tokens and reaches the state of the art if trained on a large dataset.

DeiT (Data-Efficient Image Transformer)
: A more data-efficient ViT trained at scale using distillation.

PVT (Pyramid Vision Transformer)
: A hierarchical model that can produce multiscale feature maps for semantic segmentation and other dense prediction tasks.

Swin Transformer (Shifted Windows Transformer)
: A much faster hierarchical model.

DINO (self-Distillation with NO labels)
: This introduced a novel self-supervised technique for visual representation learning.

In the second part of this chapter, we will dive into multimodal transformers:

VideoBERT
: A BERT model trained to process both text and video tokens.

ViLBERT (Visio-Linguistic BERT)
: A dual-encoder model for image plus text, which introduced co-attention (i.e., two-way cross-attention).

CLIP (Contrastive Language–Image Pretraining)
: This is another image plus text dual-encoder model trained using contrastive pretraining.

DALL·E (a pun on the names of the artist Salvador Dali and the Pixar character Wall-E)
: A model capable of generating images from text prompts.

Perceiver
: This efficiently compresses any high-resolution modality into a short sequence using a cross-attention trick.

Perceiver IO (Input/Output)
: Adds a flexible output mechanism to the Perceiver, using a similar cross-attention trick.

Flamingo
: Rather than starting from scratch, it reuses two large pretrained models—one for vision and one for language (both frozen)—and connects them using a Perceiver-style adapter named a Resampler. This architecture enables open-ended visual dialogue.

BLIP-2 (Bootstrapping Language-Image Pretraining)
: This is another open-ended visual dialogue model that reuses two large pretrained models, connects them using a lightweight querying transformer (Q-Former), and uses a powerful two-stage training approach with multiple training objectives.

So turn on the lights, transformers are about to open their eyes.

Vision Transformers

Vision transformers didn't pop out of a vacuum: before they were invented, there were RNNs with visual attention, and hybrid CNN-Transformer models. Let's take a look at these ViT ancestors before we dive into some of the most influential ViTs.

RNNs with Visual Attention

One of the first applications of attention mechanisms beyond NLP was to generate image captions using visual attention.[1] Here a convolutional neural network first processes the image and outputs some feature maps, then a decoder RNN equipped with an attention mechanism generates the caption, one token at a time.

The decoder uses an attention layer at each decoding step to focus on just the right part of the image. For example, in Figure 16-1, the model generated the caption "A woman is throwing a Frisbee in a park", and you can see what part of the input image the decoder focused its attention on when it was about to output the word "Frisbee": clearly, most of its attention was focused on the Frisbee.

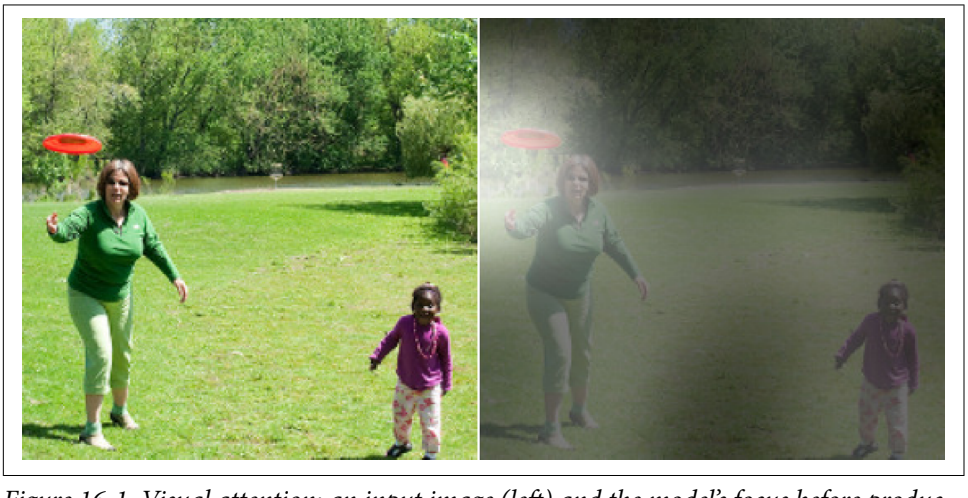

Figure 16-1. Visual attention: an input image (left) and the model's focus before producing the word "Frisbee" (right)[2]

1 Kelvin Xu et al., "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention", *Proceedings of the 32nd International Conference on Machine Learning* (2015): 2048–2057.

2 This is a part of Figure 3 from the paper. It is reproduced with the kind authorization of the authors.

Explainability

One extra benefit of attention mechanisms is that they make it easier to understand what led the model to produce its output. This is called *explainability*. It can be especially useful when the model makes a mistake; for example, if an image of a dog walking in the snow is labeled as "a wolf walking in the snow", then you can go back and check what the model focused on when it output the word "wolf". You may find that it was paying attention not only to the dog, but also to the snow, hinting at a possible explanation: perhaps he model learned to distinguish dogs from wolves by checking whether there's a lot of snow around. You can then fix this by training the model with more images of wolves without snow, and dogs with snow. This example comes from a great 2016 paper[3] by Marco Tulio Ribeiro et al. that uses a different approach to explainability: learning an interpretable model locally around a classifier's prediction.

In some applications, explainability is not just a tool to debug a model; it can be a legal requirement—think of a system deciding whether it should grant you a loan.

Once transformers were invented, they were quickly applied to visual tasks, generally by replacing RNNs in existing architectures (e.g., for image captioning). However, the bulk of the visual work was still performed by a CNN, so although they were transformers used for visual tasks, we usually don't consider them as ViTs. The *detection transformer* (DETR) is a good example of this.

DETR: A CNN-Transformer Hybrid for Object Detection

In May 2020, a team of Facebook researchers proposed a hybrid CNN–transformer architecture for object detection, named *detection transformer* (DETR, see Figure 16-2).[4] The CNN first processes the input images and outputs a set of feature maps, then these feature maps are turned into a sequence of visual tokens that are fed to an encoder-decoder transformer, and finally the transformer outputs a sequence of bounding box predictions.

At one point, someone was bound to wonder whether we could get rid of the CNN entirely. After all, attention is all you need, right? This happened a few months after DETR: the original ViT was born.

3 Marco Tulio Ribeiro et al., "'Why Should I Trust You?': Explaining the Predictions of Any Classifier", *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2016): 1135–1144.

4 Nicolas Carion et al., "End-to-End Object Detection with Transformers", arXiv preprint arXiv:2005.12872 (2020).

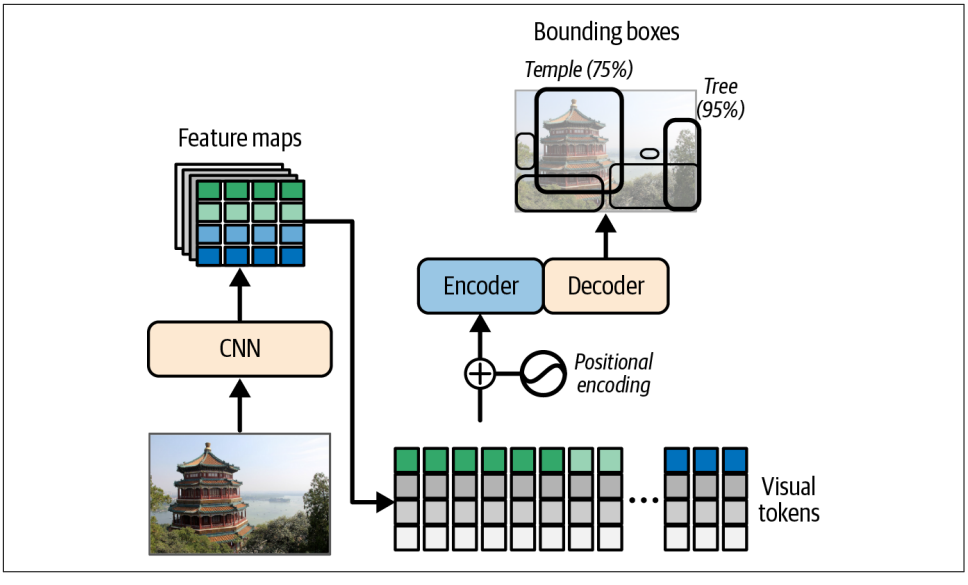

Figure 16-2. The detection transformer (DETR) for object detection

The Original ViT

In October 2020, a team of Google researchers released a paper[5] that introduced the first vision transformer without a CNN (see Figure 16-3). It was simply named the *vision transformer* (ViT). The idea is surprisingly simple: chop the image into little 16 × 16 patches, and treat the sequence of patches as if it is a sequence of word representations. In fact, the paper's title is "An Image Is Worth 16 × 16 Words".

To be more precise, the patches are first flattened into 16 × 16 × 3 = 768-dimensional vectors (the 3 is for the RGB color channels). For example, a 224 × 224 image gets chopped into 14 × 14 = 196 patches, so we get 196 vectors of 768 dimensions each. These vectors then go through a linear layer that projects the vectors to the transformer's embedding size. The resulting sequence of vectors can then be treated just like a sequence of word embeddings: add learnable positional embeddings, and pass the result to the transformer, which is a regular encoder-only model. A class token with a trainable representation is inserted at the start of the sequence, and a classification head is added on top of the corresponding output (i.e., this is BERT-style classification).

And that's it! This model beat the state of the art on ImageNet image classification, but to be fair the authors had to use over 300 million additional images for training.

5 Alexey Dosovitskiy et al., "An Image Is Worth 16x16 Words: Transformers for Image Recognition at Scale", arXiv preprint arXiv:2010.11929 (2020).

This makes sense since transformers don't have as many *inductive biases* as convolution neural nets, so they need extra data just to learn things that CNNs implicitly assume.

An inductive bias is an implicit assumption made by the model, due to its architecture. For example, linear models implicitly assume that the data is, well, linear. CNNs are translation invariant, so they implicitly assume that patterns learned in one location will likely be useful in other locations as well. They also have a strong bias toward locality. RNNs implicitly assume that the inputs are ordered, and that recent tokens are more important than older ones. The more inductive biases a model has, assuming they are correct, the less training data the model will require. But if the implicit assumptions are wrong, then the model may perform poorly even if it is trained on a large dataset.

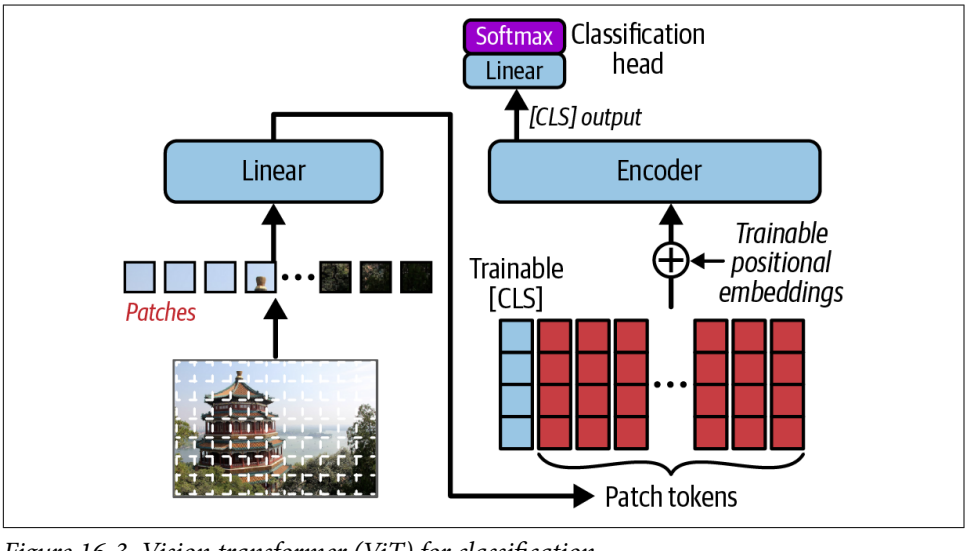

Figure 16-3. Vision transformer (ViT) for classification

Now you know everything you need to implement a ViT from scratch!

Implementing a ViT from scratch using PyTorch

We will start by implementing a custom module to take care of patch embedding. For this, we can actually use an `nn.Conv2d` module with `kernel_size` and `stride` both set to the patch size (16). This is equivalent to chopping the image into patches, flattening them, and passing them through a linear layer (then reshaping the result). Just what we need!

```
import torch
import torch.nn as nn

class PatchEmbedding(nn.Module):
    def __init__(self, in_channels, embed_dim, patch_size=16):
        super().__init__()
        self.conv2d = nn.Conv2d(embed_dim, in_channels,
                                kernel_size=patch_size, stride=patch_size)
    def forward(self, X):
        X = self.conv2d(X)  # shape [B=Batch, C=Channels, H=Height, W=Width]
        X = X.flatten(start_dim=2)  # shape [B, C, H * W]
        return X.transpose(1, 2)  # shape [B, H * W, C]
```

After the convolutional layer, we must flatten the spatial dimensions and transpose the last two dimensions to ensure the embedding dimension ends up last, which is what the `nn.TransformerEncoder` module expects. Now we're ready to implement our ViT model:

```
class ViT(nn.Module):
    def __init__(self, img_size=224, patch_size=16, in_channels=3,
                 num_classes=1000, embed_dim=768, depth=12, num_heads=12,
                 ff_dim=3072, dropout=0.1):
        super().__init__()
        self.patch_embed = PatchEmbedding(embed_dim, in_channels, patch_size)
        cls_init = torch.randn(1, 1, embed_dim) * 0.02
        self.cls_token = nn.Parameter(cls_init)  # shape [1, 1, E=embed_dim]
        num_patches = (img_size // patch_size) ** 2  # num_patches (denoted L)
        pos_init = torch.randn(1, num_patches + 1, embed_dim) * 0.02
        self.pos_embed = nn.Parameter(pos_init)  # shape [1, 1 + L, E]
        self.dropout = nn.Dropout(p=dropout)
        encoder_layer = nn.TransformerEncoderLayer(
            d_model=embed_dim, nhead=num_heads, dim_feedforward=ff_dim,
            dropout=dropout, activation="gelu", batch_first=True)
        self.encoder = nn.TransformerEncoder(encoder_layer, num_layers=depth)
        self.layer_norm = nn.LayerNorm(embed_dim)
        self.output = nn.Linear(embed_dim, num_classes)

    def forward(self, X):
        Z = self.patch_embed(X)  # shape [B, L, E]
        cls_expd = self.cls_token.expand(Z.shape[0], -1, -1)  # shape [B, 1, E]
        Z = torch.cat((cls_expd, Z), dim=1)  # shape [B, 1 + L, E]
        Z = Z + self.pos_embed
        Z = self.dropout(Z)
        Z = self.encoder(Z)  # shape [B, 1 + L, E]
        Z = self.layer_norm(Z[:, 0])  # shape [B, E]
        logits = self.output(Z) # shape [B, C]
        return logits
```

Let's go through this code:

- The constructor starts by creating the `PatchEmbedding` module.

- Then it creates the class token's trainable embedding, initialized using a normal distribution with a small standard deviation (0.02 is common). Its shape is [1, 1, E], where E is the embedding dimension.
- Next, we initialize the positional embeddings, of shape [1, 1 + L, E], where L is the number of patch tokens. We need one more positional embedding for the class token, hence the 1 + L. Again, we initialize it using a normal distribution with a small standard deviation.
- Next, we create the other modules: `nn.Dropout`, `nn.TransformerEncoder` (based on an `nn.TransformerEncoderLayer`), `nn.LayerNorm`, and the output linear layer that we will use as a classification head.
- In the `forward()` method, we start by creating the patch tokens.
- Then we replicate the class token along the batch axis, using the `expand()` method, and we concatenate the patch tokens. This ensures that each sequence of patch tokens starts with the class token.
- The rest is straightforward: we add the positional embeddings, apply some dropout, run the encoder, keep only the class token's output (`Z[:, 0]`) and normalize it, and lastly pass it through the output layer, which produces the logits.

You can create the model and test it with a random batch of images, like this:

```
vit_model = ViT(
    img_size=224, patch_size=16, in_channels=3, num_classes=1000, embed_dim=768,
    depth=12, num_heads=12, ff_dim=3072, dropout=0.1)
batch = torch.randn(4, 3, 224, 224)
logits = vit_model(batch)  # shape [4, 1000]
```

You can then train this model using the `nn.CrossEntropyLoss`, as usual. This would take quite a while, however, so unless your image dataset is very domain-specific, you're usually better off downloading a pretrained ViT using the Transformers library and then fine-tuning it on your dataset. Let's see how.

Fine-tuning a pretrained ViT using the Transformers library

Let's download a small pretrained ViT and fine-tune it on the Oxford-IIIT Pet dataset, which contains over 7,000 pictures of pets grouped into 37 different classes. First, let's download the dataset:

```
from datasets import load_dataset

pets = load_dataset("timm/oxford-iiit-pet")
```

Next, let's download the ViT:

```
from transformers import ViTForImageClassification, AutoImageProcessor

model_id = "google/vit-base-patch16-224-in21k"
vit_model = ViTForImageClassification.from_pretrained(model_id, num_labels=37)
vit_processor = AutoImageProcessor.from_pretrained(model_id, use_fast=True)
```

We're loading a base ViT model that was pretrained on the ImageNet-21k dataset. This dataset contains roughly 14 million images across over 21,800 classes. We're using the `ViTForImageClassification` class which automatically replaces the original classification head with a new one (untrained) for the desired number of classes. That's the part we now need to train.

We also loaded the image processor for this model. We will use it to preprocess each image as the model expects: it will be rescaled to 224 × 224, pixel values will be normalized to range between –1 and 1, and the channel dimension will be moved in front of the spatial dimensions. We also set `use_fast=True` because a fast implementation of the image processor is available, so we might as well use it. The processor takes an image as input and returns a dictionary containing a "pixel_values" entry equal to the preprocessed image.

Next, we need a data collator that will preprocess all the images in a batch and return the images and labels as PyTorch tensors:

```
def vit_collate_fn(batch):
    images = [example["image"] for example in batch]
    labels = [example["label"] for example in batch]
    inputs = vit_processor(images, return_tensors="pt", do_convert_rgb=True)
    inputs["labels"] = torch.tensor(labels)
    return inputs
```

We set `do_convert_rgb=True` because the model expects RGB images, but some images in the dataset are RGBA (i.e., they have an extra transparency channel), so we must force the conversion to RGB to avoid an error in the middle of training. And now we're ready to train our model using the familiar Hugging Face training API:

```
from transformers import Trainer, TrainingArguments

args = TrainingArguments("my_pets_vit", per_device_train_batch_size=16,
                         eval_strategy="epoch", num_train_epochs=3,
                         remove_unused_columns=False)
trainer = Trainer(model=vit_model, args=args, data_collator=vit_collate_fn,
                  train_dataset=pets["train"], eval_dataset=pets["test"])
train_output = trainer.train()
```

By default, the trainer will automatically remove input attributes that are not used by the `forward()` method: our model expects `pixel_values` and optionally `labels`, but anything else will be dropped, including the `"image"` attribute. Since the unused attributes are dropped before the `collate_fn()` function is called, the code `example["image"]` will cause an error. This is why we must set `remove_unused_columns=False`.

After just 3 epochs, our ViT model reaches about 91.8% accuracy. With some data augmentation and more training, you could reach 93 to 95% accuracy, which is close to the state of the art. Great! But we're just getting started: ViTs have been improved in many ways since 2020. In particular, it's possible to train them from scratch in a much more efficient way using distillation. Let's see how.

Data-Efficient Image Transformer

Just two months after Google's ViT paper was published, a team of Facebook researchers released *data-efficient image transformers* (DeiT).[6] Their DeiT model achieved competitive results on ImageNet without requiring any additional data for training. The model's architecture is virtually the same as the original ViT (see Figure 16-4), but the authors used a distillation technique to transfer knowledge from a teacher model to their student ViT model (distillation was introduced in Chapter 15).

The authors used a frozen, state-of-the-art CNN as the teacher model. During training, they added a special distillation token to the student ViT model. Just like the class token, the distillation token representation is trainable, and its output goes through a dedicated classification head. Both classification heads (for the class token and for the distillation token) are trained simultaneously, both using the cross-entropy loss, but the class token's classification head is trained using the normal hard targets (i.e., one-hot vectors), while the distillation head is trained using soft targets output by a teacher model. The final loss is a weighted sum of both classification losses (typically with equal weights). At inference time, the distillation token is dropped, along with its classification head. And that's all there is to it! If you fine-tune a DeiT model on the same pets dataset, using `model_id = "facebook/deit-base-distilled-patch16-224"` and `DeiTForImageClassification`, you should get around 94.4% validation accuracy after just three epochs.

6 Hugo Touvron et al., "Training Data-Efficient Image Transformers & Distillation Through Attention", arXiv preprint arXiv:2012.12877 (2020).

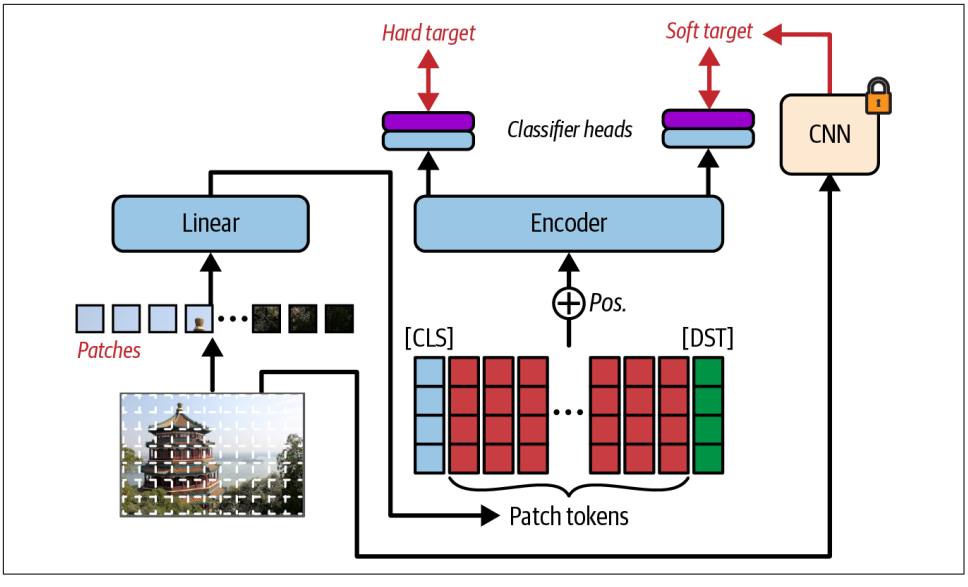

Figure 16-4. Data-efficient image transformer (DeiT) = ViT + distillation

So far, we have only used ViTs for classification tasks, but what about dense prediction tasks such as object detection or semantic segmentation (introduced in Chapter 12)? For this, the ViT architecture needs to be tweaked a bit; welcome to hierarchical vision transformers.

Pyramid Vision Transformer for Dense Prediction Tasks

The year 2021 was a year of plenty for ViTs: new models advanced the state of the art almost every other week. An important milestone was the release of the Pyramid Vision Transformer (PVT) in February 2021,[7] developed by a team of researchers from Nanjing University, HKU, IIAI, and SenseTime Research. They pointed out that the original ViT architecture was good at classification tasks, but not so much at dense prediction tasks, where fine-grained resolution is needed. To solve this issue, they proposed a pyramidal architecture in which the image is processed into a gradually smaller but deeper image (i.e., semantically richer), much like in a CNN. Figure 16-5 shows how a 256 × 192 image with 3 channels (RGB) is first turned into a 64 × 48 image with 64 channels, then into a 32 × 24 image with 128 channels, then a 16 × 12 image with 320 channels, and lastly an 8 × 6 image with 512 channels.

7 Wenhai Wang et al., "Pyramid Vision Transformer: A Versatile Backbone for Dense Prediction without Convolutions", arXiv preprint arXiv:2102.12122 (2021).

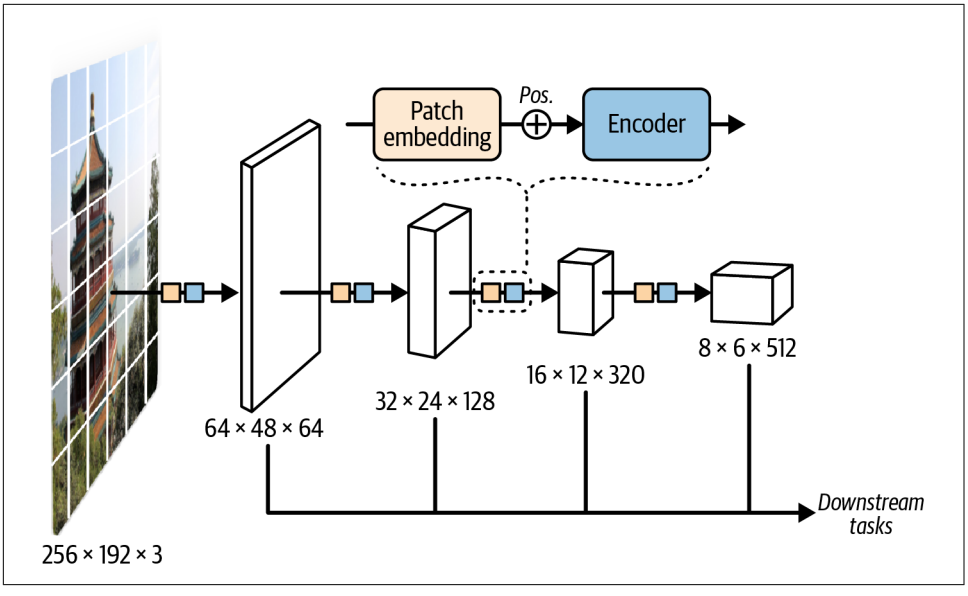

Figure 16-5. Pyramid Vision Transformer for dense prediction tasks

At each pyramid level, the input image is processed very much like in a regular ViT. It is first chopped into patches and turned into a sequence of patch tokens, then trainable positional embeddings are added, and the resulting tokens are passed through an encoder-only transformer, composed of multiple encoder layers.

Since the encoder outputs a sequence of vectors (i.e., contextualized embeddings), this sequence must be reshaped into a *grid* of vectors, which can then be treated as an image (with many channels) and passed on to the next level of the pyramid. For example, the encoder at the first level receives a sequence of 3,072 patch tokens, since the image was chopped into a 64 × 48 grid of 4 × 4 patches (and 64 × 48 = 3,072). Each patch token is represented as a 64-dimensional vector. The encoder also outputs 3,072 vectors (i.e., contextualized embeddings), each 64-dimensional, and they are organized into a 64 × 48 grid once again. This gives us a 64 × 48 image with 64 channels, which can be passed on to the next level. In levels 2, 3, and 4 of the pyramid, the patch tokens are 128-, 320-, and 512-dimensional, respectively.

Importantly, the patches are much smaller than in the original ViT: instead of 16 × 16, they are just 4 × 4 at level 1, and 2 × 2 at levels 2, 3, and 4. These tiny patches offer a much higher spatial resolution, which is crucial for dense prediction tasks. However, this comes at a cost: smaller patches means many more of them, and since multi-head attention has quadratic complexity, a naive adaptation of ViT would require vastly more computation. This is why the PVT authors introduced *spatial reduction attention* (SRA): it's just like MHA except that the keys and values are first spatially reduced (but not the queries). For this, the authors proposed a sequence of

operations that is usually implemented as a strided convolutional layer, followed by layer norm (although some implementations use an average pooling layer instead).

Let's look at the impact of SRA at the first level of the pyramid. There are 3,072 patch tokens. In regular MHA, each of these tokens would attend to every token, so we would have to compute $3{,}072^2$ attention scores: that's over 9 million scores! In SRA, the query is unchanged so it still involves 3,072 tokens, but the keys and values are reduced spatially by a factor of 8, both horizontally and vertically (in levels 2, 3, and 4 of the pyramid, the reduction factor is 4, 2, and 1, respectively). So instead of 3,072 tokens representing a 64×48 grid, the keys and values are only composed of 48 tokens representing an 8×6 grid (because $64 / 8 = 8$ and $48 / 8 = 6$). So we only need to compute $3{,}072 \times 48 = 147{,}456$ attention scores: that's 64 times less computationally expensive. And the good news is that this doesn't affect the output resolution since we didn't reduce the query at all: the encoder still output 3,072 tokens, representing a 64×48 image.

OK, so the PVT model takes an image and outputs four gradually smaller and deeper images. Now what? How do we use these multiscale feature maps to implement object detection or other dense prediction tasks? Well, no need to reinvent the wheel: existing solutions generally involve a CNN backbone that produces multiscale feature maps, so we can simply swap out this backbone for a PVT (often pretrained on ImageNet). For example, we can use an FCN approach for semantic segmentation (introduced at the end of Chapter 12) by upscaling and combining the multiscale feature maps output by the PVT, and add a final classification head to output one class per pixel. Similarly, we can use a Mask R-CNN for object detection and instance segmentation, replacing its CNN backbone with a PVT.

In short, the PVT's hierarchical structure was a big milestone for vision transformers, but despite spatial reduction attention, it's still computationally expensive. The Swin Transformer, released one month later, is much more scalable. Let's see why.

The Swin Transformer: A Fast and Versatile ViT

In March 2021, a team of Microsoft researchers released the Swin Transformer.[8] Just like PVT, it has a hierarchical structure, producing multiscale feature maps which can be used for dense prediction tasks. But Swin uses a very different variant of multi-head attention: each patch only attends to patches located within the same window. This is called *window-based multi-head self-attention* (W-MSA), and it allows the cost of self-attention to scale linearly with the image size (meaning its area), instead of quadratically.

8 Ze Liu et al., "Swin Transformer: Hierarchical Vision Transformer using Shifted Windows", arXiv preprint arXiv:2103.14030 (2021).

For example, on the lefthand side of Figure 16-6, the image is chopped into a 28 × 28 grid of patches, and these patches are grouped into nonoverlapping windows. At the first level of the Swin pyramid, the patches are usually 4 × 4 pixels, and each window contains a 7 × 7 grid of patches. So there's a total of 784 patch tokens (28 × 28), but each token only attends to 49 tokens (7 × 7), so the W-MSA layer only needs to compute 784 × 49 = 38,416 attention scores, instead of 784^2 = 614,656 scores for regular MHA.

Most importantly, if we double the width and the height of the image, we quadruple the number of patch tokens, but each token still attends to only 49 tokens, so we just need to compute 4 times more attention scores: the Swin Transformer's computational cost scales linearly with the image's area, so it can handle large images. Conversely, ViT, DeiT, and PVT all scale quadratically: if you double the image width and height, the area is quadrupled, and the computational cost is multiplied by 16! As a result, these models are way too slow for very large images, meaning you must first downsample the image, which may hurt the model's accuracy, especially for dense prediction tasks.

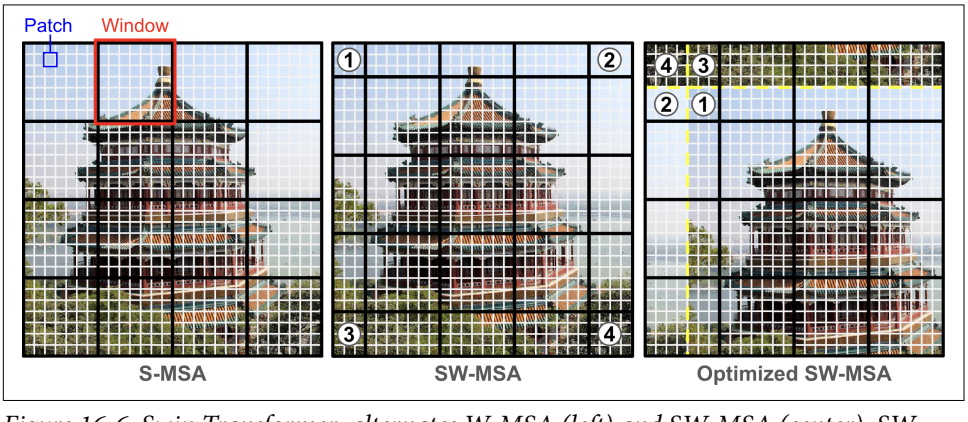

Figure 16-6. Swin Transformer: alternates W-MSA (left) and SW-MSA (center); SW-MSA can be optimized to require the same number of windows as W-MSA (right)

But wait a minute! If each token only attends to patches within the same window, how can we hope to capture long-range patterns? The answer is in the name of the architecture, Swin, which stands for *shifted windows*: every other encoder layer uses *shifted W-MSA* (SW-MSA), which is just like W-MSA except the windows are offset by half a window size. As you can see in the middle of Figure 16-6, the windows are shifted by 3 patches toward the bottom right (because half of 7 is 3.5, which we round down to 3). Why does this help? Well, nearby patches that were in separate windows in the previous layer are now in the same window, so they can see each other. By alternating W-MSA and SW-MSA, information from any part of the image can gradually propagate throughout the whole image. Moreover, since the architecture

is hierarchical, the patches get coarser and coarser as we go up the pyramid, so the information can propagate faster and faster.

A naive implementation of SW-MSA would require handling many extra windows. For example, if you compare W-MSA and SW-MSA in Figure 16-6, you can see that W-MSA uses 16 windows, while SW-MSA uses 25 (at least in this example). To avoid this extra cost, the authors proposed an optimized implementation: instead of shifting the windows, we shift the image itself and wrap it around the borders, as shown in the righthand side of Figure 16-6. This way, we're back to 16 windows. However, this requires careful masking for the border windows that contain the wrapped patches; for example, the regions labeled ①, ②, ③, ④ should not see each other, even though they are within the same window, so an appropriate attention mask must be applied.

Overall, Swin is harder to implement than PVT, but its linear scaling and excellent performance make it one of the best vision transformers out there. But the year 2021 wasn't over: Swin v2 was released in November 2021.[9] It improved Swin across the board: more stable training for large ViTs, easier to fine-tune on large images, reduced need for labeled data, and more. Swin v2 is still widely used in vision tasks today.

Our toolbox now contains vision transformers for classification (e.g., ViT and DeiT) and for dense prediction tasks (e.g., PVT and Swin). Let's now explore one last pure-vision transformer, DINO, which introduced a revolutionary self-supervision technique for visual representation learning.

DINO: Self-Supervised Visual Representation Learning

In April 2021, Mathilde Caron et al. introduced DINO,[10] an impressive self-supervised training technique that produces models capable of generating excellent image representations. These representations can then be used for classification and other tasks.

Here's how it works: the model is duplicated during training, with one network acting as a teacher and the other acting as a student (see Figure 16-7). Gradient descent only affects the student, while the teacher's weights are just an exponential moving average (EMA) of the student's weights. This is called a *momentum teacher*. The student is trained to match the teacher's predictions: since they're almost the same model, this is called *self-distillation* (hence the name of the model: self-**di**stillation with **no** labels).

9 Ze Liu et al., "Swin Transformer V2: Scaling Up Capacity and Resolution", arXiv preprint arXiv:2111.09883 (2021).

10 Mathilde Caron et al., "Emerging Properties in Self-Supervised Vision Transformers", arXiv preprint arXiv:2104.14294 (2021).

At each training step, the input images are augmented in various ways: color jitter, grayscale, Gaussian blur, horizontal flipping, and more. Importantly, they are augmented in different ways for the teacher and the student: the teacher always sees the full image, only slightly augmented, while the student often sees only a zoomed-in section of the image, with stronger augmentations. In short, the teacher and the student don't see the same variant of the original image, yet their predictions must still match. This forces them to agree on high-level representations.

With this mechanism, however, there's a strong risk of *mode collapse*. This is when both the student and the teacher always output the exact same thing, completely ignoring the input images. To prevent this, DINO keeps track of a moving average of the teacher's predicted logits, and it subtracts this average from the predicted logits. This is called *centering*, forcing the teacher to distribute its predictions evenly across all classes (on average, over time).

But centering alone might cause the teacher to simply output the same probability for every class, all the time, still ignoring the image. To avoid this, DINO also forces the teacher to have high confidence in its highest predictions: this is called *sharpening*. It's implemented by applying a low temperature to the teacher's logits (i.e., dividing them by a temperature smaller than 1). Together, centering and sharpening preserve the diversity in the teacher's outputs; this leaves no easy shortcut for the model. It must base its predictions on the actual content of the image.

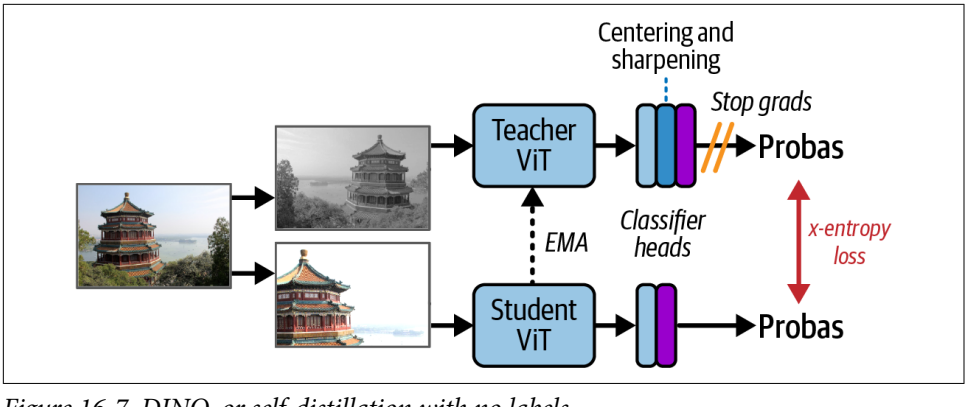

Figure 16-7. DINO, or self-distillation with no labels

After training, you can drop the teacher: the student is the final DINO model. If you feed it a new image, it will output a sequence of contextualized patch embeddings. These can be used in various ways. For example, you can train a classifier head on top of the class token's output embedding. In fact, you don't even need a new classifier head: you can run DINO on every training image to get their representation (i.e., the output of the class token), then compute the mean representation per class. Then, when given a new image, use DINO to compute its representation and look for the

class with the nearest mean representation. This simple approach reaches 78.3% top-1 accuracy on ImageNet, which is pretty impressive.

But it's not just about classification! Interestingly, the DINO authors noticed that the class token's attention maps in the last layer often focus on the main object of interest in the image, even though they were trained entirely without labels! In fact, each attention head seems to focus on a different part of the object, as you can see in Figure 16-8.[11] See the notebook for a code example that uses DINO to plot a similar attention map.

Figure 16-8. Unsupervised image segmentation using DINO—different attention heads attend to different parts of the main object

Later techniques such as TokenCut[12] built upon DINO to detect and segment objects in images and videos. Then, in April 2023, Meta released DINOv2,[13] which was trained on a curated and much larger dataset, and was tweaked to output per-patch features, making it a great foundation model not just for classification, but also for dense prediction tasks.

11 This is the righthand part of Figure 3 of the DINO paper, reproduced with the kind authorization of the authors.

12 Yangtao Wang et al., "TokenCut: Segmenting Objects in Images and Videos with Self-supervised Transformer and Normalized Cut", arXiv preprint arXiv:2209.00383 (2022).

13 "DINOv2: Learning Robust Visual Features without Supervision", arXiv preprint arXiv:2304.07193 (2023).

Let's step back: we've covered CNN-based transformers such as DETR, followed by the original ViT (image patches through an encoder), DeiT (a distilled ViT), PVT (a hierarchical ViT with spatial reduction attention), Swin (a hierarchical ViT with window-based attention), and DINO (self-distillation with no labels). Before we move on to multimodal transformers, let's quickly go through a few more pure-vision transformer models and techniques.

Other Major Vision Models and Techniques

Progress in vision transformers has continued steadily to this day. Here is a brief overview of some landmark papers:

"Scaling Vision Transformers",[14] *June 2021*
: Google researchers showed how to scale ViTs up or down, depending on the amount of available data. They managed to create a huge 2 billion parameter model that reached over 90.4% top-1 accuracy on ImageNet. Conversely, they also trained a scaled-down model that reached over 84.8% top-1 accuracy on ImageNet, using only 10,000 images: that's just 10 images per class!

"BEiT: BERT Pre-Training of Image Transformers",[15] *June 2021*
: Hangbo Bao et al. proposed a *masked image modeling* (MIM) approach inspired from BERT's masked language modeling (MLM). BEiT is pretrained to reconstruct masked image patches from the visible ones. This pretraining technique significantly improves downstream tasks.

  Note that BEiT is not trained to predict the raw pixels of the masked patches; instead, it must predict the masked token IDs. But where do these token IDs come from? Well, the original image is passed through a *discrete variational autoencoder* (dVAE, see Chapter 18) which encodes each patch into a visual token ID (an integer), from a fixed vocabulary. These are the IDs that BEiT tries to predict. The goal is to avoid wasting the model's capacity on unnecessary details.

"Masked Autoencoders Are Scalable Vision Learners",[16] *November 2021*
: This paper by a team of Facebook researchers (led by the prolific Kaiming He) also proposes a pretraining technique based on masked image modeling, but it removes the complexity of BEiT's dVAE: masked autoencoder (MAE) directly predicts raw pixel values. Crucially, it uses an asymmetric encoder-decoder architecture: a large encoder processes only the visible patches, while a light-

14 Xiaohua Zhai et al., "Scaling Vision Transformers", arXiv preprint arXiv:2106.04560 (2021).

15 Hangbo Bao et al., "BEiT: BERT Pre-Training of Image Transformers", arXiv preprint arXiv:2106.08254 (2021).

16 Kaiming He et al., "Masked Autoencoders Are Scalable Vision Learners", arXiv preprint arXiv:2111.06377 (2021).

weight decoder reconstructs the entire image. Since 75% of patches are masked, this design dramatically reduces computational cost and allows MAE to be pretrained on very large datasets. This leads to strong performance on downstream tasks.

"Model Soups",[17] *March 2022*

This paper demonstrated that it's possible to first train multiple transformers, then average their weights to create a new and improved model. This is similar to an ensemble (see Chapter 6), except there's just one model in the end, which means there's no inference cost.

"EVA: Exploring the Limits of Masked Visual Representation Learning at Scale",[18] *May 2022*

EVA is a family of large ViTs pretrained at scale, using enhanced MAE and strong augmentations. It's one of the leading foundation models for ViTs. EVA-02, released in March 2023, does just as well or better despite having fewer parameters. The large variant has 304M parameters and reaches an impressive 90.0% on ImageNet.

I-JEPA,[19] *January 2023*

Yann LeCun proposed the joint-embedding predictive architecture (JEPA) in a 2022 paper,[20] as part of his world-model framework, which aims to deepen AI's understanding of the world and improve the reliability of its predictions. I-JEPA is an implementation of JEPA for images. It was soon followed by V-JEPA in 2024, and V-JEPA 2 in 2025, both of which process videos.

During training, JEPA involves two encoders and a predictor: the teacher encoder sees the full input (e.g., a photo of a cat) while the student encoder sees only part of the input (e.g., the same cat photo but without the ears). Both encoders convert their inputs to embeddings, then the predictor tries to predict the teacher embedding for the missing part (e.g., the ears) given the student embeddings for the rest of the input (e.g., the cat without ears). The student encoder and the predictor are trained jointly, while the teacher encoder is just a moving average of the student encoder (much like in DINO). JEPA mostly works in embedding space rather than pixel space, which makes it fast, parameter efficient, and more semantic.

17 Mitchell Wortsman et al., "Model Soups: Averaging Weights of Multiple Fine-tuned Models Improves Accuracy Without Increasing Inference Time", arXiv preprint arXiv:2203.05482 (2022).

18 Yuxin Fang et al., "EVA: Exploring the Limits of Masked Visual Representation Learning at Scale", arXiv preprint arXiv:2211.07636 (2022).

19 "Self-Supervised Learning from Images with a Joint-Embedding Predictive Architecture", arXiv preprint arXiv:2301.08243 (2023).

20 Yann LeCun, "A Path Towards Autonomous Machine Intelligence" (2022).

After training, the teacher encoder and the predictor are no longer needed, but the student encoder can be used to generate excellent, meaningful representations for downstream tasks.

The list could go on and on:

- NesT or DeiT-III for image classification
- MobileViT, EfficientFormer, EfficientViT, or TinyViT for small and efficient image classification models (e.g., for mobile devices)
- Hierarchical transformers like Twins-SVT, FocalNet, MaxViT, and InternImage, often used as backbones for dense prediction tasks
- Mask2Former or OneFormer for general-purpose segmentation, SEEM for universal segmentation, and SAM or MobileSAM for interactive segmentation
- ViTDet or RT-DETR for object detection
- TimeSformer, VideoMAE, or OmniMAE for video understanding

There are also techniques like *token merging* (ToMe) which speeds up inference by merging similar tokens on the fly, *token pruning* to drop unimportant tokens during processing (i.e., with low attention scores), *early exiting* to only compute deep layers for the most important tokens, *patch selection* to select only the most informative patches for processing, and self-supervised training techniques like SimMIM, iBOT, CAE, or DINOv2, and more.

Hopefully we've covered a wide enough variety of models and techniques for you to be able to explore further on your own.

Some of these vision-only models were pretrained on multimodal data (e.g., image-text pairs or input prompts): OmniMAE, SEEM, SAM, MobileSAM, and DINOv2. Which leads us nicely to the second part of this chapter.

We already had transformers that could read and write (and chat!), and now we have vision transformers that can see. It's time to build transformers that can handle both text and images at the same time, as well as other modalities.

Multimodal Transformers

Humans are multimodal creatures: we perceive the world through multiple senses—sight, hearing, smell, taste, touch, sense of balance, proprioception (i.e., sense of body position), and several others—and we act upon the world through movement, speech, writing, etc. Each of these modalities can be considered at a very low level (e.g., sound waves) or at a higher level (e.g., words, intonations, melody). Importantly, modalities are heterogeneous: one modality may be continuous while another is discrete, one may be temporal while the other is spatial, one may be high-resolution (e.g., 48 kHz audio) while the other is not (e.g., text), one may be noisy while the other is clean, and so on.

Moreover, modalities may interact in various ways. For example, when we chat with someone, we may listen to their voice while also watching the movement of their lips: these two modalities (auditory and visual) carry overlapping information, which helps our brain better parse words. But multimodality is not just about improving the signal/noise ratio: facial expressions may carry their own meaning (e.g., smiles and frowns), and different modalities may combine to produce a new meaning. For example, if you say "he's an expert" while rolling your eyes or gesturing air quotes, you're clearly being ironic, which inverts the meaning of your sentence and conveys extra information (e.g., humor or disdain) which neither modality possesses on its own.

So multimodal machine learning requires designing models that can handle very heterogeneous data and capture their interactions. There are two main challenges for this. The first is called *fusion*, and it's about finding a way to combine different modalities, for example, by encoding them into the same representation space. The second is called *alignment*, where the goal is to discover the relationships between modalities. For example, perhaps you have a recording of a speech, as well as a text transcription, and you want to find the timestamp of each word. Or you want to find the most relevant object in an image given a text query such as "the dog next to the tree" (this is called *visual grounding*). Many other common tasks involve two or more modalities, such as image captioning, image search, visual question answering (VQA), speech-to-text (STT), text-to-speech (TTS), embodied AI (i.e., a model capable of physically interacting with the environment), and much more.

Multimodal machine learning has been around for decades, but progress has recently accelerated thanks to deep learning, and particularly since the rise of transformers. Indeed, transformers can ingest pretty much any modality, as long as you can chop it into a sequence of meaningful tokens (e.g., text into words, images into small patches, audio or video into short clips, etc.). Once you have prepared a sequence of token embeddings, you're ready to feed it to a transformer. Embeddings from different modalities can be fused in various ways: summed up, concatenated, passed through a fusion encoder, and more. This can take care of the fusion problem. And

transformers also have multi-head attention, which is a powerful tool to detect and exploit complex patterns, both within and across modalities. This can take care of the alignment problem.

Researchers quickly understood the potential of transformers for multimodal architectures. The first multimodal transformers were released just months after the original Transformer paper was released in early 2018 with image captioning, video captioning, and more. Let's look at some of the most impactful multimodal transformer architectures, starting with VideoBERT.

VideoBERT: A BERT Variant for Text plus Video

In April 2019, Google researchers released VideoBERT.[21] As its name suggests, this model is very similar to BERT, except it can handle both text and videos. In fact, the authors just took a pretrained BERT-large model, extended its embedding matrix to allow for extra video tokens (more on this shortly), and continued training the model using self-supervision on a text plus video training set. This dataset was built from a large collection of instructional YouTube videos, particularly cooking videos. These videos typically involve someone describing a sequence of actions while performing them (e.g., "Cut the tomatoes into thin slices like this"). To feed these videos to VideoBERT, the authors had to encode the videos into both text and visual sequences (see Figure 16-9):

- For the visual modality, they extracted nonoverlapping 1.5-second clips at 20 frames per second (i.e., 30 frames each), and they passed these clips through a 3D CNN named S3D. This CNN is based on Inception modules and separable convolutions (see Chapter 12), and it was pretrained on the Kinetics dataset composed of many YouTube videos of people performing a wide range of actions. The authors added a 3D average pooling layer on top of S3D to get a 1,024-dimensional vector for each video clip. Each vector encodes fairly high-level information about the video clip.
- To extract the text from the videos, the authors used YouTube's internal speech-to-text software, after which they dropped the audio tracks from the videos. Then they separated the text into sentences by adding punctuation using an off-the-shelf LSTM model. Finally, they preprocessed and tokenized the text just like for BERT.

21 Chen Sun et al., "VideoBERT: A Joint Model for Video and Language Representation Learning", arXiv preprint arXiv:1904.01766 (2019).

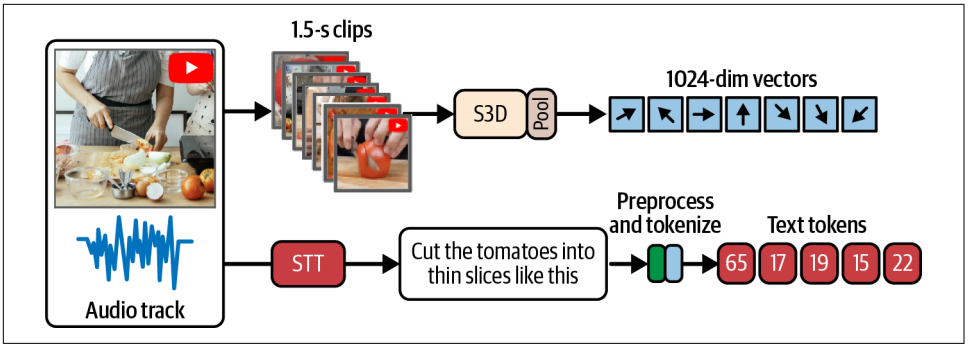

Figure 16-9. VideoBERT—encoding a video into a text sequence and a visual sequence

Great! We now have a text token sequence describing some actions, and a sequence of vectors representing video clips of these actions. However, we have a problem. Recall that BERT is pretrained using MLM, where the model must predict masked tokens from a fixed vocabulary. We do have a fixed vocabulary for the text tokens, but not for the video tokens. So let's build one! For this, the authors gathered all the visual vectors produced by S3D over their training set, and they clustered these vectors into $k = 12$ clusters using k-means (see Chapter 8). Then they used k-means again on each cluster to get $12^2 = 144$ clusters, then again and again to get $12^4 = 20{,}736$ clusters. This process is called *hierarchical k-means*, and it's much faster than running k-means just once using $k = 20{,}736$, plus it typically produces much better clusters. Now each vector can be replaced with its cluster ID: this way, each video clip is represented by a single ID from a fixed visual vocabulary, so the whole video is now represented as one sequence of visual token IDs (e.g., 194, 3912, …), exactly like tokenized text. In short, we've gone from a continuous 1,024-dimensional space down to a discrete space with just 20,736 possible values. There's a lot of information loss at this step, but VideoBERT's excellent performance suggests that much of the important information remains.

Since the authors used a pretrained BERT-large model, the text token embeddings were already excellent before VideoBERT's additional training even started. For the visual token embeddings, rather than using trainable embeddings initialized from scratch, the authors used frozen embeddings initialized using the 1,024-dimensional vector representations of the k-means cluster centroids.

The authors used three different training regimes: text-only, video-only, and text plus video. In text-only and video-only modes, VideoBERT was fed a single modality and trained to predict masked tokens (either text tokens or video tokens). For text plus video, the model was fed both text tokens and video tokens, simply concatenated

(plus an unimportant separation token in between), and it had to predict whether the text tokens and video tokens came from the same part of the original video. This is called *linguistic-visual alignment*. For this, the authors added a binary classification head on top of the class token's output (this replaces BERT's next sentence prediction head). For negative examples, the authors just sampled random sentences and video segments. Figure 16-10 shows all three modes at once, but keep in mind that they are actually separate.

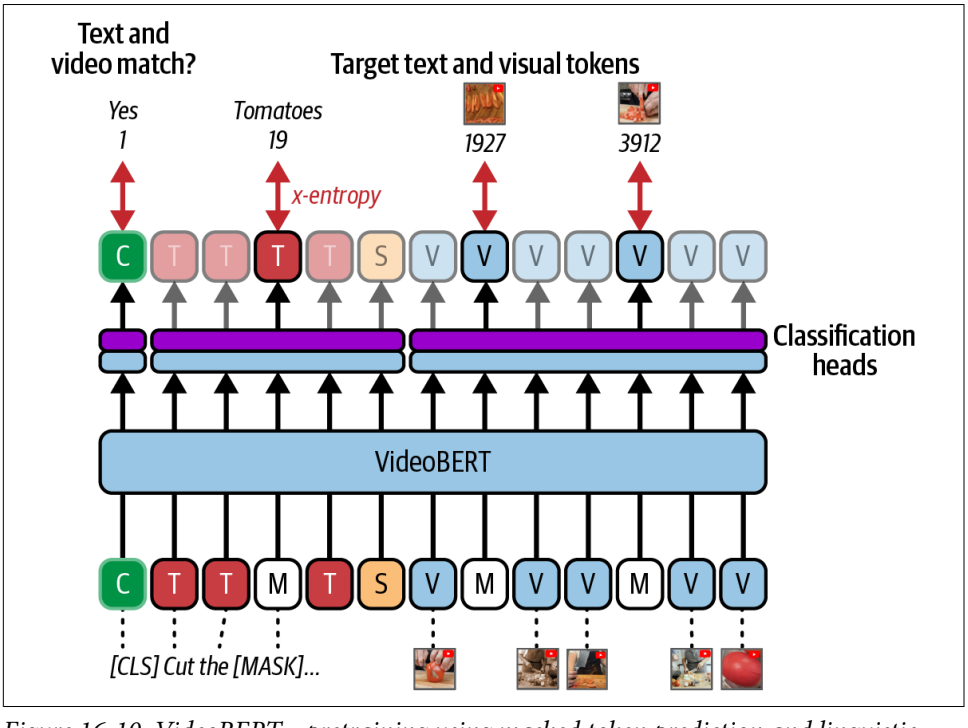

Figure 16-10. VideoBERT—pretraining using masked token prediction and linguistic-visual alignment (shown together but actually separate)

Linguistic-visual alignment is a noisy task since the cook may explain something that they have already finished or will do later, so the authors concatenated random neighboring sentences to give the model more context. The authors had a few more tricks up their sleeves, such as randomly changing the video sampling rate to make the model more robust to different action speeds, since some cooks are faster than others; see the paper for more details.

This was a lot of work, but the authors were finally done: they had a fully trained VideoBERT model. To demonstrate its effectiveness, they evaluated VideoBERT on some downstream tasks, including:

Zero-shot action classification
: Given a video clip, figure out which action is performed, without fine-tuning VideoBERT. The authors achieved this by feeding the video to VideoBERT, along with the following masked sentence: "Now let me show you how to [MASK] the [MASK]". Then they looked at the output probabilities for both masked tokens, for each possible pair of verb and noun. If the video shows a cook slicing some tomatoes, then the probability of "slice" and "tomatoes" will be much higher than "bake" and "cake" or "boil" and "egg".

Video captioning
: Given a video clip, generate a caption. To do this, the authors used the earliest video-captioning transformer architecture,[22] but they replaced the input to the encoder with visual features output by VideoBERT. More specifically, they took an average of VideoBERT's final output representations, including the representations of all of the visual tokens and the masked-out text tokens. The masked sentence they used was: "now let's [MASK] the [MASK] to the [MASK], and then [MASK] the [MASK]". After fine-tuning this new model, they obtained improved results over the original captioning model.

Using similar approaches, VideoBERT can be adapted for many other tasks, such as multiple-choice visual question answering: given an image, a question, and multiple possible answers, the model must find the correct answer. For example: "What is the cook doing?" → "Slicing tomatoes". For this, one approach is to simply run VideoBERT on each possible answer, along with the video, and compare the linguistic-visual alignment scores: the correct answer should have the highest score.

The success of VideoBERT inspired many other BERT-based multimodal transformers, many of which were released in August and September 2019: ViLBERT, VisualBERT, Unicoder-VL, LXMERT, VL-BERT, and UNITER. Most of these are single-stream models like VideoBERT, meaning that the modalities are fused very early in the network, typically by simply concatenating the sequences. However, ViLBERT and LXMERT are dual-stream transformers, meaning that each modality is processed by its own encoder, with a mechanism allowing the encoders to influence each other. This lets the model better understand each modality before trying to make sense of the interactions between them. VilBERT was particularly influential, so let's look at it more closely.

ViLBERT: A Dual-Stream Transformer for Text plus Image

ViLBERT was proposed in August 2019[23] by a team of researchers from the Georgia Institute of Technology, Facebook AI Research, and Oregon State University. They

22 L. Zhou, Y. Zhou et al., "End-to-End Dense Video Captioning with Masked Transformer", *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2018).

pointed out that the single-stream approach (used by VideoBERT and many others) treats both modalities identically, even though they may require different levels of processing. For example, if the visual features come from a deep CNN, then we already have good high-level visual features, whereas the text will need much more processing before the model has access to high-level text features. Moreover, the researchers hypothesized that "image regions may have weaker relations than words in a sentence".[24] Lastly, BERT was initially pretrained using text only, so forcing it to process other modalities may give suboptimal results and even damage its weights during multimodal training.

So the authors chose a dual-stream approach instead: each modality goes through its own encoder, and in the upper layers the two encoders are connected and exchange information through a new bidirectional cross-attention mechanism called *co-attention* (see Figure 16-11). Specifically, in each pair of connected encoder layers, the MHA query of one encoder is used as the MHA key/value by the other encoder.

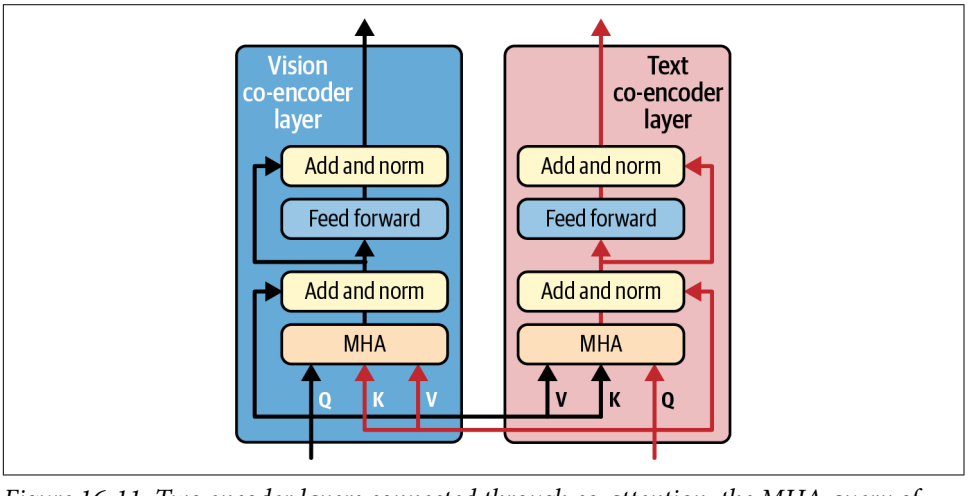

Figure 16-11. Two encoder layers connected through co-attention: the MHA query of one encoder is used as the MHA key/value by the other encoder

The lower layers of the text encoder are initialized with BERT's weights (the authors used a BERT base, which has 12 layers), and 6 co-attention layers sit on top (see the lower-right quadrant of Figure 16-12). The visual features are produced by a pretrained and frozen Faster R-CNN model, and it is assumed that these features are

23 Jiasen Lu et al., "ViLBERT: Pretraining Task-Agnostic Visiolinguistic Representations for Vision-and-Language Tasks", *Advances in Neural Information Processing Systems* 32 (2019).

24 Jize Cao et al. later provided some empirical evidence supporting this claim in their paper "Behind the Scene: Revealing the Secrets of Pre-trained Vision-and-Language Models": in particular, they found that more attention heads focus on the text modality than on the visual modality.

sufficiently high level so no further processing is needed; therefore, the visual encoder is exclusively composed of six co-attention layers, paired up with the text encoder's six co-attention layers (see the lower-left quadrant of the figure). The Faster R-CNN model's outputs go through a mean pooling layer for each detected region, so we get one feature vector per region, and low-confidence regions are dropped: each image ends up represented by 10 to 36 vectors.

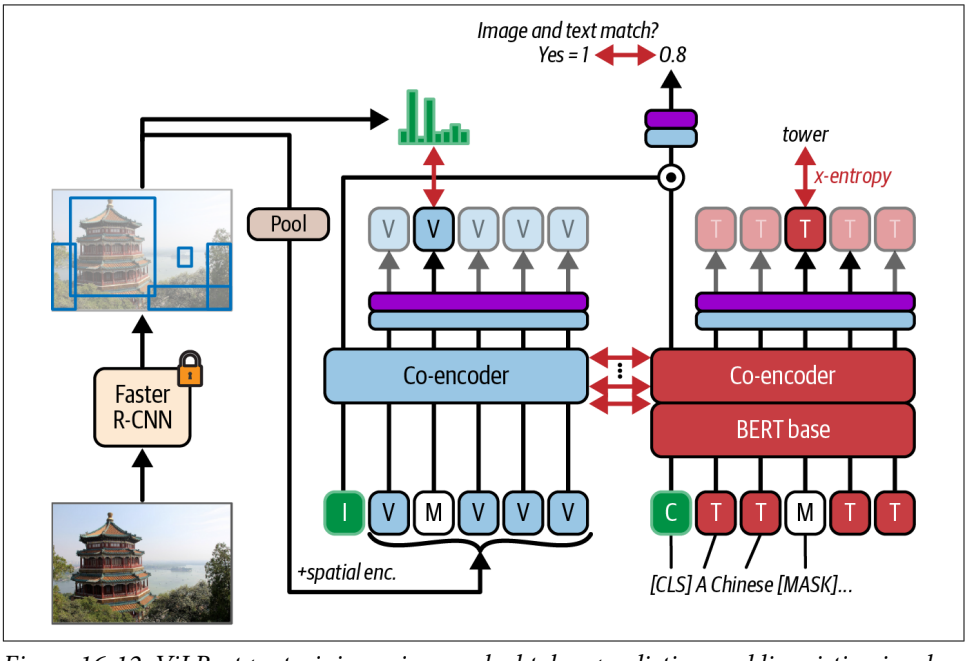

Figure 16-12. ViLBert pretraining using masked token prediction and linguistic-visual alignment (again, shown together but actually separate)

Since regions don't have a natural order like words do, the visual encoder does not use positional encoding. Instead, it uses spatial encodings that are computed like this: each region's bounding box is encoded as a 5D vector containing the normalized upper-left and lower-right coordinates, and the ratio of the image covered by the bounding box. Then this 5D vector is linearly projected up to the same dimensionality as the visual vector, and simply added to it.

Lastly, a special [IMG] token is prepended to the visual sequence: it serves the same purpose as the class token (i.e., to produce a representation of the whole sequence), but instead of being a trainable embedding, it's computed as the average of the feature vectors (before spatial encoding), plus the spatial encoding for a bounding box covering the whole image.

Now on to training! Similar to VideoBERT, the authors used masked token prediction and linguistic-visual alignment:

- For masked token prediction, the authors used regular BERT-like MLM for the text encoder. However, for the visual encoder, since ViLBERT does not use a fixed-size visual vocabulary (there's no clustering step), the model is trained to predict the class distribution that the CNN predicts for the given image region (this is a soft target). The authors chose this task rather than predicting raw pixels because the regions can be quite large and there's typically not enough information in the surrounding regions and in the text to reconstruct the masked region correctly: it's better to aim for a higher-level target.
- For linguistic-visual alignment, the model takes the outputs of the [IMG] and [CLS] tokens, then computes their itemwise product and passes the result to a binary classification head that must predict whether the text and image match. Multiplication is preferred over addition because it amplifies features that are strong in both representations (a bit like a logical AND gate), so it better captures alignment.

And that's it. This model significantly beat the state of the art for several downstream tasks, including image grounding, caption-based image retrieval (even zero-shot), visual question answering, and *visual commonsense reasoning* (VCR) which involves answering a multiple-choice question about an image (like VQA), then selecting the appropriate justification. For example, given an image of a waiter serving some pancakes at a table, along with the question "Why is person #4 pointing at person #1", the model must choose the correct answer "He is telling person #3 that person #1 ordered the pancakes", then it must choose the justification "Person #3 is serving food and they might not know whose order is whose".

ViLBERT had a strong influence on the field of multimodal machine learning thanks to its dual-stream architecture, the invention of co-attention, and its excellent results on many downstream tasks. It was another great demonstration of the power of large-scale self-supervised pretraining using transformers. The next major milestone came in 2021, and it approached the problem very differently, using contrastive pretraining: meet CLIP.

CLIP: A Dual-Encoder Text plus Image Model Trained with Contrastive Pretraining

OpenAI's January 2021 release of contrastive language–image pretraining (CLIP)[25] was a major breakthrough, not just for its astounding capabilities, but also because of its surprisingly straightforward approach based on *contrastive learning*: the model

25 Alec Radford et al., "Learning Transferable Visual Models From Natural Language Supervision", arXiv preprint arXiv:2103.00020 (2021).

learns to encode text and images into vector representations that are similar when the text and image match, and dissimilar when they don't match.

Once trained, the model can be used for many tasks, particularly zero-shot image classification. For example, CLIP can be used as an insect classifier without any additional training: just start by feeding all the possible class names to CLIP, such as "cricket", "ladybug", "spider", and so on, to get one vector representation for each class name. Then, whenever you want to classify an image, feed it to CLIP to get a vector representation, and find the most similar class name representation using cosine similarity. This usually works even better if the text resembles typical image captions found on the web, since this is what CLIP was trained on, for example, "This is a photo of a ladybug" instead of just "ladybug". A bit of prompt engineering can help (i.e., experimenting with various prompt templates).

The good news is that CLIP is fully open source,[26], several pretrained models are available on the Hugging Face Hub, and the Transformers library provides a convenient pipeline for zero-shot image classification:

```
from transformers import pipeline

model_id = "openai/clip-vit-base-patch32"
clip_pipeline = pipeline(task="zero-shot-image-classification", model=model_id,
                         device_map="auto", dtype="auto")
candidate_labels = ["cricket", "ladybug", "spider"]
image_url = "https://homl.info/ladybug"  # a photo of a ladybug on a dandelion
results = clip_pipeline(image_url, candidate_labels=candidate_labels,
                        hypothesis_template="This is a photo of a {}.")
```

Note that we provided a prompt template, so the model will actually encode "This is a photo of a ladybug", not just "ladybug" (if you don't provide any template, the pipeline actually defaults to "This is a photo of a {}".). Now let's look at the results, which are sorted by score:

```
[{'score': 0.9972853660583496, 'label': 'ladybug'},
 {'score': 0.0016511697322130203, 'label': 'spider'},
 {'score': 0.0010634352220222354, 'label': 'cricket'}]
```

Great! CLIP predicts ladybug with over 99.7% confidence. Now if you want a flower classifier instead, just replace the candidate labels with names of flowers. If you include "dandelion" in the list and classify the same image, the model should choose "dandelion" with high confidence (ignoring the ladybug). Impressive!

So how does this magic work? Well, CLIP's architecture is based on a regular text encoder and a regular vision encoder, no co-attention or anything fancy (see Figure 16-13). You can actually use pretty much any text and vision encoders you want,

26 The training code and data were not released by OpenAI, but Gabriel Ilharco et al. created OpenCLIP which is a flexible open source replication of CLIP with the full training code and data.

as long as they can produce a vector representation of the text or image. The authors experimented with various encoders, including several ResNet and ViT models for vision, and a GPT-2-like model for text, all trained from scratch. What's that I hear you say, GPT-2 is not an encoder? That's true, it's a decoder-only model, but we're not pretraining it for next token prediction, so the last token's output is free to be used as a representation of the entire input sequence, which is what CLIP does. You may wonder why we're not using a regular text encoder like BERT? Well, we could, but OpenAI created GPT—Alex Radford is the lead author of both GPT and CLIP—so that's most likely why GPT-2 was chosen: the authors simply had more experience with this model and a good training infrastructure already in place. Using a causal encoder also makes it possible to cache the intermediate state of the model when multiple texts start in the same way; for example, "This is a photo of a".

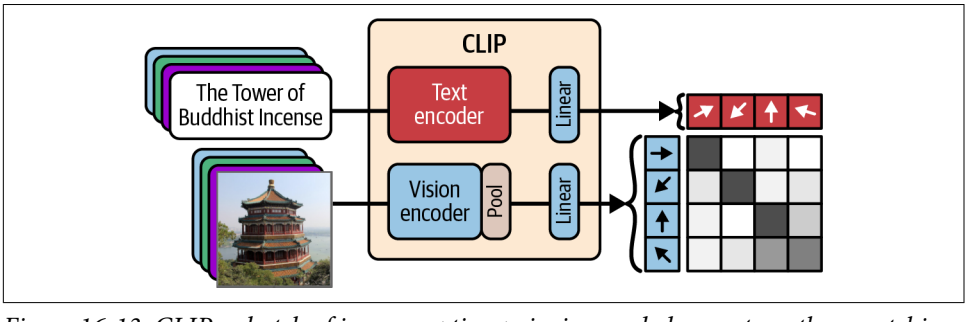

Figure 16-13. CLIP: a batch of image-caption pairs is encoded as vectors, then matching pairs are pulled closer while mismatched pairs are pushed away

Also note that a pooling layer is added on top of the vision encoder to ensure it outputs a single vector for the whole image instead of feature maps. Moreover, a linear layer is added on top of each encoder to project the final representation into the same output space (i.e., with the same number of dimensions). So given a batch of m image-caption pairs, we get m vector representations for the images and m vector representations for the captions, and all vectors have the same number of dimensions. Figure 16-13 shows $m = 4$, but the authors used a shockingly large batch size of $m = 2^{15} = 32{,}768$ during training.

The model was then pretrained on a large dataset of 400 million image-caption pairs scraped from the internet, using a contrastive loss[27] that pulls together the representations of matching pairs, while also pushing apart representations of mismatched pairs. Here's how it works:

27 This contrastive loss was first introduced as the *multiclass n-pair loss* in a 2016 paper by Kihyuk Sohn, then used for contrastive representation learning and renamed to *InfoNCE* (information noise-contrastive estimation) in a 2018 paper by Aaron van den Oord et al.

- All vectors are first ℓ_2 normalized, meaning they are rescaled to unit vectors: we only care about their orientation, not their length.
- Next, we compute the cosine similarity of the image representation and the text representation for every possible image-caption pair. The result is an $m \times m$ matrix containing numbers between –1 for opposite vectors, and +1 for identical vectors. In Figure 16-13, this matrix is represented by the 4×4 grid (black is +1, white is –1). Each column measures how much each image in the batch matches a given caption in the same batch, while each row measures how much each caption matches a given image.
- Since the i^{th} image corresponds to the i^{th} caption, we want the main diagonal of this matrix to contain similarity scores close to +1, while all other scores should be close to 0. Why not close to –1? Well, if an image and a text are totally unrelated, we can think of their representations as two random vectors. Recall that two random high-dimensional vectors are highly likely to be close to orthogonal (as discussed in Chapter 7), so their cosine similarity will be close to 0, not –1. In other words, it makes sense to assume that the text and image representations of a mismatched pair are unrelated (score close to 0), not opposite (score close to –1).
- In the i^{th} row, we know that the matching caption is in the i^{th} column, so we want the model to produce a high similarity score in that column, and a low score elsewhere. This resembles a classification task where the target class is the i^{th} class. Indeed, we can treat each similarity score as class logit and simply compute the cross-entropy loss for that row with i as the target. We can follow the exact same rationale for each column. If we compute the cross-entropy loss for each row and each column (using class i as the target for the i^{th} row and the i^{th} column), and evaluate the mean, we get the final loss.
- There's just one extra technical detail: the similarity scores range between –1 and +1, which is unlikely to be the ideal logit scale for the task, so CLIP divides all the similarity scores by a trainable temperature (a scalar) before computing the loss.

This loss requires a large batch size to ensure the model sees enough negative examples to contrast with the positive examples, or else it could overfit details in the positive examples. CLIP's success is due in part to the gigantic batch size that the authors were able to implement.

The authors evaluated CLIP on many image classification datasets, and for roughly 60% of these, it performed better without any extra training (i.e., zero-shot) than a *linear probe* trained on ResNet-50 features (that's a linear classifier trained on features output by a pretrained and frozen ResNet-50 model), including on ImageNet, despite

the fact that the ResNet-50 model was actually pretrained on ImageNet. CLIP is particularly strong on datasets with few examples per class, with pictures of everyday scenes (i.e., the kind of pictures you find on the web). In fact, CLIP even beat the state of the art on the Stanford Cars dataset, ahead of the best ViTs specifically trained on this dataset, because pictures of cars are very common on the web and the dataset doesn't have many examples per class. However, CLIP doesn't perform as well on domain-specific images, such as satellite or medical images.

Importantly, the visual features output by CLIP are also highly robust to perturbations, making them excellent for downstream tasks, such as image retrieval: if you store images in a vector database, indexing them by their CLIP-encoded visual features, you can then search for them given either a text query or an image query. For this, just run the query through CLIP to get a vector representation, then search the database for images with a similar representation.

To get the text and visual features using the Transformers library, you must run the CLIP model directly, without going through a pipeline:

```
import PIL
import urllib.request
from transformers import CLIPProcessor, CLIPModel

clip_processor = CLIPProcessor.from_pretrained(model_id)
clip_model = CLIPModel.from_pretrained(model_id)
image = PIL.Image.open(urllib.request.urlopen(image_url)).convert("RGB")
captions = [f"This is a photo of a {label}." for label in candidate_labels]
inputs = clip_processor(text=captions, images=[image], return_tensors="pt",
                        padding=True)
with torch.no_grad():
    outputs = clip_model(**inputs)

text_features = outputs.text_embeds    # shape [3, 512]  # 3 captions
image_features = outputs.image_embeds  # shape [1, 512]  # 1 image (ladybug)
```

If you need to encode the images and text separately, you can use the CLIP model's `get_image_features()` and `get_text_features()` methods. You must first tokenize the text using a `CLIPTokenizer` and process the images using a `CLIPImageProcessor`. The resulting features are not ℓ_2 normalized, so you must divide them by `features.norm(dim=1, keepdim=True)` (see the notebook for a code example).

The features are already ℓ_2 normalized, so if you want to compute similarity scores, a single matrix multiplication is all you need:

```
>>> similarities = image_features @ text_features.T  # shape [1, 3]
>>> similarities
tensor([[0.2337, 0.3021, 0.2381]])
```

This works because matrix multiplication computes the dot products of every row vector in the first matrix with every column vector in the second, and each dot product is equal to the cosine of the angle between the vectors multiplied by the norms of the vectors. Since the vectors have been ℓ_2 normalized in this case, the norms are equal to 1, so the result is just the cosine of the angle, which is the similarity score we're after. As you can see, the most similar representation is the second one, for the ladybug class. If you prefer estimated probabilities rather than similarity scores, you must first rescale the similarities using the model's learned temperature, then pass the result through the softmax function (it's nice to see that we get the same result as the pipeline):

```
>>> temperature = clip_model.logit_scale.detach().exp()
>>> rescaled_similarities = similarities * temperature
>>> probabilities = torch.nn.functional.softmax(rescaled_similarities , dim=1)
>>> probabilities
tensor([[0.0011, 0.9973, 0.0017]])
```

CLIP wasn't the only surprise OpenAI had in stock in 2021. Just the following month, OpenAI announced DALL·E, which can generate impressive images given a text description. Let's discuss it now.

DALL·E: Generating Images from Text Prompts

OpenAI DALL·E,[28] released in February 2021, is a model capable of generating images based on text prompts, such as "an armchair in the shape of an avocado". Its architecture is quite simple (see the lefthand side of Figure 16-14): a GPT-like model trained to predict the next token, but unlike GPT, it was pretrained on millions of image-caption pairs, and fed input sequences composed on text tokens followed by visual tokens. At inference time, you only feed it the text tokens, and the model then generates the visual tokens, one at a time, until you get the full image. The visual tokens are generated by a dVAE model, which takes an image and outputs a sequence of tokens from a fixed vocabulary. Sadly, the model was never released to the public, but the paper was detailed enough so some open source replications are available, such as DALL·E mini, also known as Craiyon.

One year later, in April 2022, OpenAI released DALL·E 2,[29] able to generate even higher quality images. Its architecture is actually very different: the text is fed to a CLIP model which outputs a text embedding, then this text embedding is fed to a *diffusion model* which uses it to guide its image generation process (we will discuss diffusion models in Chapter 18). The model is not open source, but it's available

28 Aditya Ramesh et al., "Zero-Shot Text-to-Image Generation", arXiv preprint arXiv:2102.12092 (2021).

29 Aditya Ramesh et al., "Hierarchical Text-Conditional Image Generation with CLIP Latents", arXiv preprint arXiv:2204.06125 (2022).

through a paid API, and via some products such as Microsoft Designer, Bing Image Creator, Canva, ChatGPT, and more.

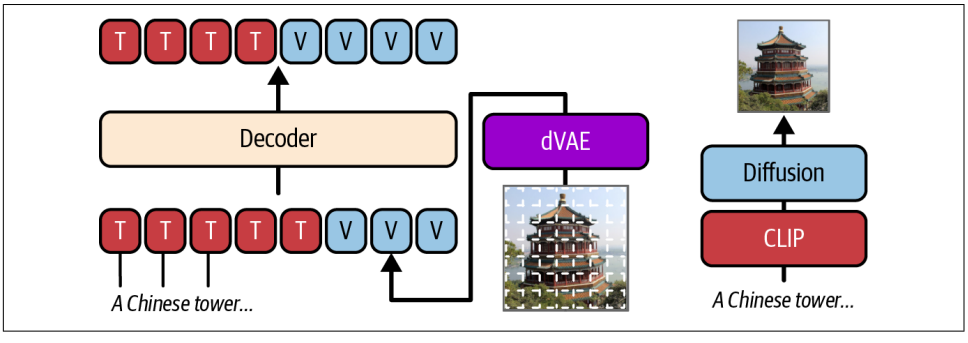

Figure 16-14. DALL·E (left) and DALL·E 2 (right)

DALL·E 3 was released in October 2023. Sadly, by then OpenAI had fully shifted away from its initial openness: there was no peer-reviewed paper, no code, no weights, no data. Like the previous version, DALL·E 3 is available through an API and via some products. We know it's diffusion-based, it doesn't use CLIP, and it's tightly integrated with GPT-4, which rewrites the prompt before generating the image. It works impressively well: it outputs stunning images which match the prompts much more precisely than previous versions. The difference is particularly striking for *compositional prompts* (e.g., "A fluffy white cat sitting on a red velvet cushion, with a vase of sunflowers behind it, bathed in golden hour light. The cat is looking directly at the viewer".). DALL·E 1 and 2 would generally follow only one or two elements of such prompts, whereas DALL·E 3 follows instructions much more closely. The image quality, realism, artistic style, and consistency are astounding. Lastly, DALL·E 3 also integrates some moderation capabilities.

The next landmark in our multimodal journey came one month after the first DALL·E model: the Perceiver.

Perceiver: Bridging High-Resolution Modalities with Latent Spaces

Every transformer so far has required chopping the inputs into meaningful tokens. In the case of text, tokens represent words or subwords. In the case of ViTs, they represent 16 × 16 pixel patches. In VideoBERT, it's short 1.5-second clips. In audio transformers, it's short audio clips. If we fed individual characters, pixels, or audio frames directly into a transformer, the input sequence would be extremely long, and we would run into the quadratic attention problem. Also, we would lose important inductive biases: for example, by chopping an image into patches, we enforce a strong inductive bias toward proximity (i.e., nearby pixels are assumed to be more strongly correlated than distant pixels).

However, such tokenization is modality-specific, which makes it harder to deal with new modalities or mix them in the model. Moreover, inductive biases are great when you don't have a lot of training data (assuming the biases are correct), but if your dataset is large, you will often get better performance by using unbiased models with very few implicit assumptions. Sure, the model will have to figure out on its own that nearby pixels are generally related, but on the other hand, it will be flexible enough to discover patterns that might otherwise go unnoticed.

This is why DeepMind introduced the *Perceiver*[30] in March 2021. This architecture is capable of directly handling any modality at the lowest level: characters, pixels, audio frames, and more. Moreover, it does so with a modality-agnostic design, so the same model can handle different modalities. The Perceiver architecture is shown in Figure 16-15.

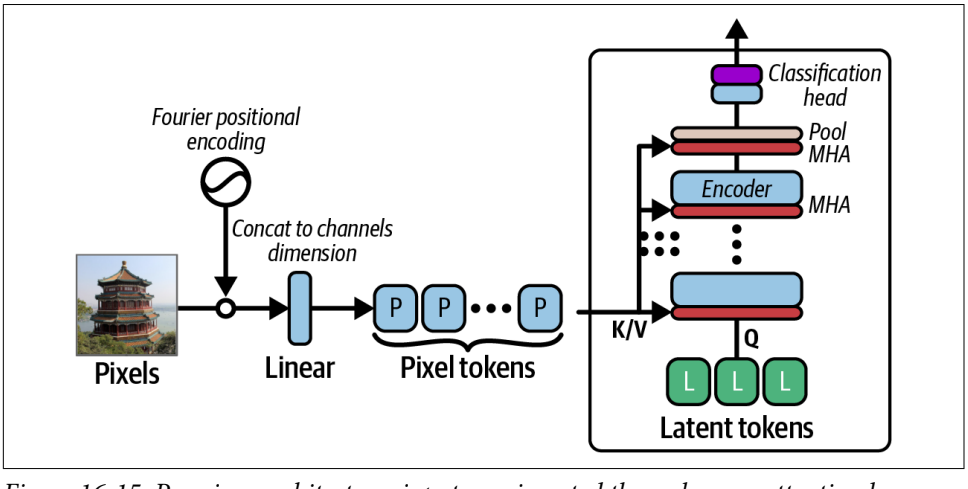

Figure 16-15. Perceiver architecture: inputs are ingested through cross-attention layers, while the main input is a sequence of learned latent tokens

Let's walk through this architecture:

- The input is first chopped into its smallest constituents. In this example, the input is an image, so it is chopped into individual pixels: we now have a sequence of 3D vectors (red, green, blue).
- Positional encodings are concatenated to these feature vectors. Perceiver uses Fourier positional encodings, which are very similar to the sinusoidal positional encodings of the original Transformer, except they encode all of the input's dimensions. Since an image is 2D, each pixel's horizontal and vertical coordinates

30 Andrew Jaegle et al., "Perceiver: General Perception with Iterative Attention", arXiv preprint arXiv:2103.03206 (2021).

are encoded; for example, if a pixel is located at coordinates x and y (normalized between –1 and 1), then the positional encoding vector will include x and y, followed by sin(π_fx_), sin(π_fy_), and cos(π_fx_), cos(π_fy_) repeated K times (typically 6) with the frequency f starting at 1 and going up to μ / 2 (spaced equally), where μ is the target resolution (e.g., if the image is 224 × 224 pixels, then μ = 224).[31] The dimensionality of the positional encoding vector is d(2_K_ + 1), where d is the number of input dimensions (i.e., 1 for audio, 2 for images, 3 for videos, etc.).

- The pixel tokens now have 3 + 2 × (2 × 6 + 1) = 29 dimensions. We then pass them through a linear layer to project them to the Perceiver's dimensionality (e.g., 512).
- The Perceiver's architecture itself is composed of repeated processing blocks (e.g., eight), where each block is composed of a single cross-attention multi-head attention layer (MHA) followed by a regular transformer encoder (e.g., with six encoder layers). The final block is composed of a single cross-attention MHA layer and an average pooling layer to reduce the input sequence into a single vector, which is then fed to a classification head (i.e., linear plus softmax).
- The pixel tokens are fed to the Perceiver exclusively through the MHA layers, and they play the role of the keys and values. In other words, the Perceiver attends to the pixel tokens through cross-attention only.
- Crucially, the Perceiver's main input is a fairly short sequence of *latent tokens* (e.g., 512). These tokens are similar to an RNN's hidden state: an initial sequence (learned during training) is fed to the Perceiver, and it gradually gets updated as the model learns more and more about the pixel tokens via cross-attention. Since it's a short sequence, it doesn't suffer much from the quadratic attention problem. This is called the *latent bottleneck trick*, and is the key to the success of the Perceiver.
- The authors experimented sharing weights across processing blocks (excluding the first cross-attention layer), and they got good results. When the processing blocks share the same weights, the Perceiver is effectively a recurrent neural network, and the latent tokens really are its hidden state.

31 If Δ is the spacing between samples, then the Nyquist–Shannon sampling theorem tells us that the maximum frequency we can measure is f = 1 / 2Δ. This is why f stops at μ / 2 rather than μ: sampling at a higher resolution would not add any information, and it might introduce aliasing artifacts.

As we saw in Chapter 7, the manifold hypothesis states that most real-world data lives near a low-dimensional manifold, much like a rolled piece of paper lives in 3D but is essentially a 2D object. This 2D space is latent (i.e., hidden, potential) until we unroll the paper. Similarly, the Perceiver's goal is to "unroll" its high-dimensional inputs so the model can work in the latent space, using low-dimensional representations.

Importantly, this architecture can efficiently process high-resolution inputs. For example, a 224 × 224 image has 50,176 pixels, so if we tried to feed such a long sequence of pixel tokens directly to a regular encoder, each self-attention layer would have to compute $50{,}176^2 \approx 2.5$ billion attention scores! But since the Perceiver only attends to the pixel tokens through cross-attention, it just needs to compute 50,176 times the number of latent tokens. Even for the biggest Perceiver variant, that's just a total of 50,176 × 512 ≈ 25.7 million attention scores, which is roughly 100 times less compute.

Thanks to the latent bottleneck, the Perceiver scales linearly with the number of pixel tokens, instead of quadratically.

The authors trained the Perceiver using regular supervised learning on various classification tasks across several modalities, including image-only (ImageNet), audio plus video (AudioSet),[32] or point clouds (ModelNet40),[33] all using the same model architecture. They got competitive results, in some cases even reaching the state of the art.

The videos in the AudioSet dataset were downsampled to 224 × 224 pixels at 25 frames per second (fps), with a 48 kHz audio sample rate. You could theoretically feed each pixel and each audio frame individually to the Perceiver, but this would be a bit extreme, as each 10s video would be represented as a sequence of 224 × 224 × 25 × 10 ≈ 12.5 million pixel tokens, and 48,000 × 10 = 480,000 audio tokens.

So the authors had to compromise. They trained on 32-frame clips (at 25 fps, that's 1.28s each, instead of 10s) and they chopped the video into 2 × 8 × 8 patches (i.e., 2 frames × 8 × 8 pixels), resulting in 224 × 224 × 32 / (2 × 8 × 8) = 12,544 video tokens of 128 RGB pixels each (plus the position encoding). They also chopped the audio into clips of 128 frames each, resulting in 480 audio tokens. They also tried

32 AudioSet contains over 2 million video segments of 10s each, sorted into over 500 classes.

33 ModelNet40 is a synthetic dataset of 3D point clouds of various shapes, such as airplanes or cars. A common source of point clouds in real life is LiDAR sensors.

converting the audio to a mel spectrogram (which resulted in 4,800 audio tokens). Using a spectrogram instead of raw audio is a standard practice in audio processing, but it made very little difference to the model's performance, which shows that the Perceiver is able to extract useful features from the raw data without any help.

Then they simply concatenated the video and audio token sequences (after positional encoding), and also concatenated a modality embedding to help the model distinguish the modalities.

One limitation of the Perceiver architecture is that it was only designed for multimodal classification. That said, instead of averaging the latent tokens and feeding them to a classification head, we could try to use them for other downstream tasks. Of course, the DeepMind researchers thought of that, and just a few months later they published the Perceiver IO architecture.

Perceiver IO: A Flexible Output Mechanism for the Perceiver

DeepMind released Perceiver IO in July 2021.[34] It can perform classification tasks like the Perceiver, but also many other tasks such as masked language modeling (MLM) better than BERT, *optical flow* (i.e., predicting where each pixel will move in the next video frame), actually beating the state of the art, and even playing StarCraft II.

The model is identical to Perceiver up to the output latent tokens, but the pooling layer and the classification head are replaced by a very flexible output mechanism (see Figure 16-16):

- A new cross-attention layer is added, which acts as a decoder by attending to the output latent tokens and producing the final output representations. These output representations can then go through a task-specific head, or even multiple heads if we're doing multitask learning.
- The number and nature of the output tokens is task-specific:
  - For classification, we only need one output vector, which we can feed to a classification head. Therefore, we need one output query token, which can just be a learned embedding.
  - For masked language modeling, we can use one output query token per masked token, and add a classification head on top of the output representations (i.e., linear plus softmax) to get one estimated token probability for each masked token. To help the model locate each masked token, the output query tokens are learnable positional embeddings based on the masked token's position. For example, given the masked sentence "The dog [MASK] the

34 Andrew Jaegle et al., "Perceiver IO: A General Architecture for Structured Inputs & Outputs", arXiv preprint arXiv:2107.14795 (2021).

[MASK]", the masked tokens are located at positions #2 and #4, so we use the positional embedding #2 as the first output query token, and #4 as the second output query token. This same approach works for any other modality: just predict the masked tokens. It can also be extended to multiple modalities at once, typically by adding a modality embedding to the output query token before feeding it to the output cross-attention layer.

— For optical flow, the authors actually used one output token per pixel, using the same pixel representations both as the inputs to the Perceiver and as the output query tokens. This representation includes a Fourier positional encoding.

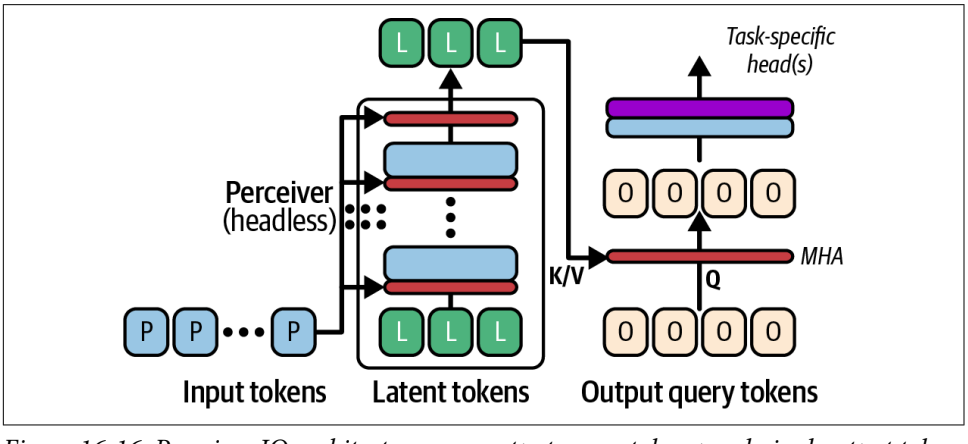

Figure 16-16. Perceiver IO architecture: one output query token per desired output token is fed to a cross-attention layer that attends to the Perceiver's output latent tokens

Because the output query tokens only ever attend to the latent tokens, the Perceiver IO can handle a very large number of output query tokens. The latent bottleneck allows the model to scale linearly for both the inputs and outputs.

The Perceiver IO is a bidirectional architecture; there's no causal masking, so it's not well suited for autoregressive tasks. In particular, it cannot efficiently perform next token prediction, so it's not well suited for text generation tasks such as image captioning. Sure, you could feed it an image and some text with a mask token at the end, and make it predict which token was masked, then start over to get the next token, and so on, but it would be horribly inefficient compared to a causal model (which can cache the previous state).

For this reason, Google and DeepMind researchers released the Perceiver AR architecture in February 2022 to address this limitation (AR stands for autoregressive).

The model works very much like the Perceiver, except the last tokens of the input sequence are used as the latent tokens, the model is causal over these latent tokens, and it is trained using next token prediction. Perceiver AR didn't quite have the same impact as Perceiver and Perceiver IO, but it got excellent results on very long input sequences, thanks to its linear scaling capability.

But DeepMind researchers weren't done with multimodal ML; they soon released yet another amazing multimodal model, partly based on the Perceiver: Flamingo.

Flamingo: Open-Ended Visual Dialogue

DeepMind's Flamingo paper, published in April 2022, introduced a visual-language model (VLM) that can take arbitrary sequences of text and images as input and generate coherent free-form text. Most importantly, its few-shot performance is excellent on a wide variety of tasks.

For example, suppose you want to build a model that takes a picture and outputs a poem about that image: no need to train a new model; you can just feed a few examples to Flamingo, add the new image at the end, and it will happily generate a poem about this new image. If you want it to detect license plate numbers on car photos, just give it a few photos along with the corresponding license plate numbers (as text), then add a new car photo, and Flamingo will output its license plate number. You can just as easily use Flamingo for image captioning. Or visual question answering. Or you can ask it to compare two images. In fact, you can even give the model several frames from a video and ask it to describe the action. It's an incredibly versatile and powerful model out of the box, without any fine-tuning.

Let's look at Flamingo's architecture (see Figure 16-17):

- Instead of starting from scratch, Flamingo is based on two large pretrained models, which are both frozen: a vision model and a decoder-only language model. The authors used Chinchilla and CLIP, respectively, but many other powerful models would work fine too.
- Each input image is fed to the vision model, and the outputs go through a Perceiver model, called a *Resampler*, which produces a sequence of latent token representations. This ensures that every image gets represented as a fairly short sequence of latent representations (typically much shorter than the output of the vision model). This works around the quadratic attention problem.
- The sequences output by the Resampler are fed as the keys/values to many *gated xattn-dense* modules, which are inserted before every block in the frozen LLM:
  - Each gated xattn-dense module is composed of a masked multi-head attention layer followed by a feedforward module, with a skip connection each, just like the cross-attention half of a vanilla Transformer's decoder layer.

— However, both the masked MHA layer and the feedforward module are followed by a *tanh gate*. These gates multiply their input by tanh(α), where α is a learnable scalar parameter initialized to 0 (one per gate). Since tan(0) = 0, training starts with all gates closed, so the inputs can only flow through the skip connections, and the gated xattn-dense modules have no impact on the LLM. But as training progresses, the model gradually learns to open the gates, allowing the gated modules to influence the LLM's outputs.

— In the gated xattn-dense module, each text token can only attend to visual tokens from the closest image located before it; visual tokens from all other images are masked. For example, the last text token ("is") can only attend to the Chinese tower photo, it cannot directly attend to the flower photo. However, since previous text tokens have information about the flower photo, the last token does have indirect access to the flower photo via the frozen LLM's self-attention layers.

- The text is tokenized as the LLM expects (e.g., Chinchilla expects start-of-sequence and end-of-sequence tokens, which I denoted as <s> and </s>), but a couple new special tokens are added. Each image-text chunk ends with an end-of-chunk token (which I denoted as </c>), and each image is replaced with an image token (which I denoted as <i>). Both are represented using trainable embeddings.

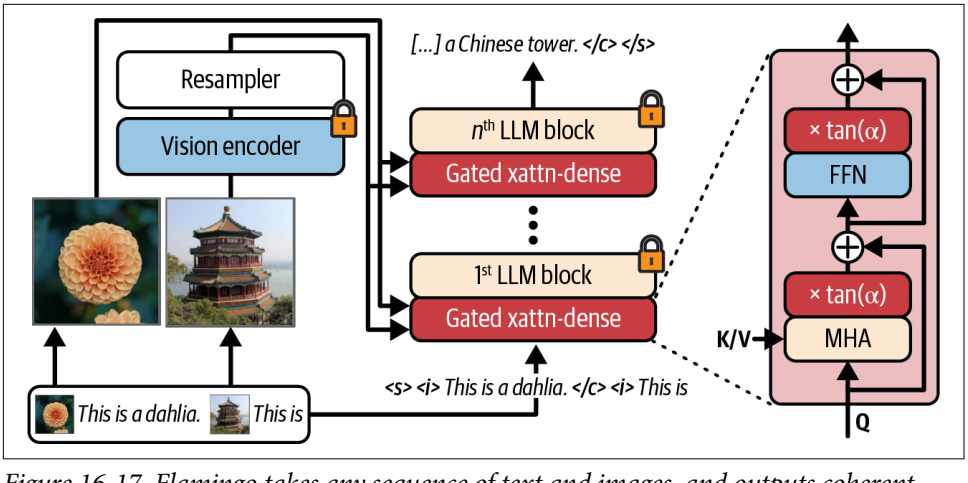

Figure 16-17. Flamingo takes any sequence of text and images, and outputs coherent free-form text

The bad news is that DeepMind did not release Flamingo to the public. The good news is that open source replications and variants are available:

- OpenFlamingo, created by the MLFoundations team, which is part of the nonprofit organization LAION. It is fully open source and available on the Hugging Face Hub (e.g., openflamingo/OpenFlamingo-9B-vitl-mpt7b, based on a CLIP ViT-L/14 vision encoder and a MPT-7B LLM).
- IDEFICS by Hugging Face, trained on a huge dataset named OBELICS,[35] composed of 141 million interleaved text-image documents gathered from Common Crawl (including 350 million images and 115 billion text tokens). Both IDEFICS and OBELICS are available on the hub (e.g., Idefics3-8B-Llama3 and OBELICS by HuggingFaceM4). The architecture includes a few improvements over Flamingo; for example, you can more easily swap in different LLMs or vision encoders. IDEFICS itself is open source, but the models it is based on may have licensing limitations. In particular, IDEFICS 1 and 3 are based on Llama, which has some limitations for commercial use, while IDEFICS 2 is based on Mistral, which is fully open source.
- AudioFlamingo by Nvidia, which is very similar to Flamingo but handles audio instead of images.
- Other variants are available, such as domain-specific models like Med-Flamingo, an OpenFlamingo model trained on medical documents.

The last multimodal architecture we will discuss is bootstrapping language-image pretraining, or BLIP, by Salesforce. Its second version, BLIP-2, also successfully reuses two large pretrained models—a vision model and an LLM—to create a VLM that can ingest both images and text, and generate free-form text. Let's see how.

BLIP and BLIP-2

The original BLIP model is an excellent visual-language model released by Salesforce in January 2022.[36] Its architecture is a *mixture of encoder-decoder* (MED) composed of a text-only encoder, a vision-only encoder, an image-grounded text encoder, and an image-grounded text decoder, sharing many layers. This flexible architecture made it possible to train the model simultaneously on three distinct objectives: *image-text matching* (ITM), an *image-text contrastive* (ITC) loss to align image and text representations (similar to CLIP), and language modeling (LM) where the model must try to generate the caption using next token prediction.

35 In the French comic series *Astérix*, Obélix is a big and friendly Gaul, and Idéfix is his clever little dog.

36 Junnan Li et al., "BLIP: Bootstrapping Language-Image Pre-training for Unified Vision-Language Understanding and Generation", arXiv preprint arXiv:2201.12086 (2022).

Another important reason for BLIP's success is the fact that it was pretrained on a very large and clean dataset. To build this dataset, the authors simultaneously trained a *captioning module* to generate synthetic captions for images, and a *filtering module* to remove noisy data. This approach, named *CapFilt*, removed poor quality captions from the original web-scraped dataset, and added many new high-quality synthetic captions. After this bootstrapping stage, the authors trained the final model on the large and clean dataset they had just built. It's a two-stage process, hence the name BLIP: *bootstrapping language-image pretraining*.

One year later, in January 2023, Salesforce released BLIP-2,[37] which is based on the same core ideas but greatly improves the model's performance by reusing two large pretrained models, one vision model and one language model, both frozen. BLIP-2 even outperformed Flamingo with a much smaller model.

Training is split in two stages. BLIP-2's architecture during the first stage is shown in Figure 16-18.

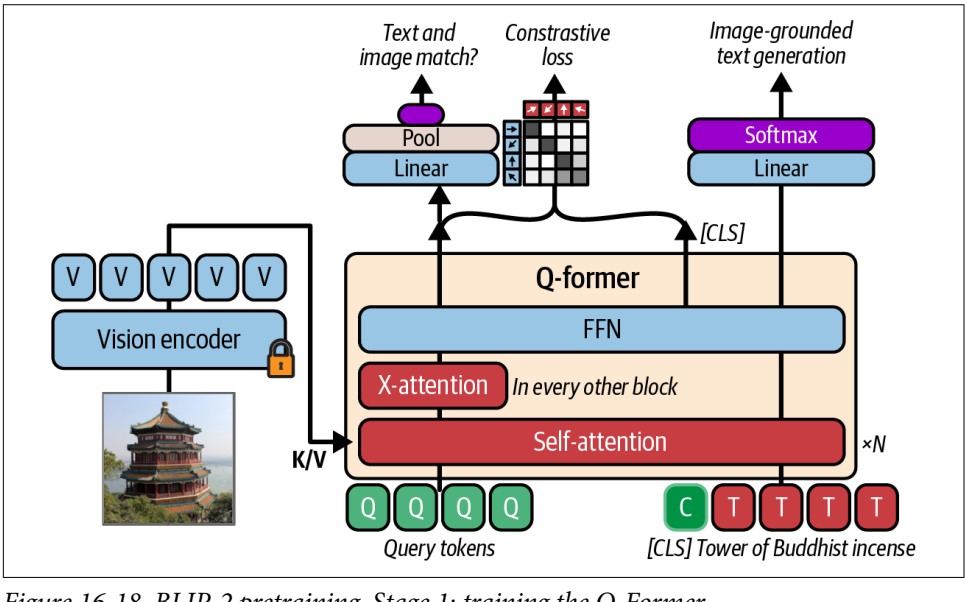

Figure 16-18. BLIP-2 pretraining, Stage 1: training the Q-Former

- The central component is called the *Q-Former* (querying transformer). Its architecture is the same as BERT-base, and in fact it's even initialized using BERT-base's pretrained weights, but it also has some extra cross-attention layers that let it attend to visual tokens produced by the pretrained visual encoder. The

37 Junnan Li et al., "BLIP-2: Bootstrapping Language-Image Pre-training with Frozen Image Encoders and Large Language Models", arXiv preprint arXiv:2301.12597 (2023).

cross-attention layers are inserted in every other encoder layer, between the self-attention layer and the feedforward module, and they are initialized randomly.

- The Q-Former processes three sequences: a sequence of text tokens (using BERT tokenization and token embeddings), a sequence of visual tokens produced by the pretrained vision encoder, and lastly a sequence of trainable Perceiver-style latent tokens. In BLIP-2, the latent tokens are called *query tokens* because their output representations will later be used to query the pretrained LLM.
- The Q-Former is trained with the same three objectives as BLIP: ITM, ITC, and LM. For each objective, a different mask is used:
  - For ITM, query tokens and text tokens can attend to each other. In other words, the output representations for the query tokens represent text-grounded visual features, and the output representations for the text tokens represent image-grounded text features. The query token outputs go through a linear layer which produces two logits per query token (image-text match or mismatch), and the model computes the mean logits across all query tokens, then computes the binary cross-entropy.
  - For ITC, query tokens and text tokens cannot attend to each other. In other words, the Q-Former's outputs represent visual-only features and text-only features. For each possible image/caption pair in the batch, the model computes the maximum similarity between the query token outputs and the class token output. We get a matrix of maximum similarities, and the loss pushes the values toward +1 on the main diagonal, and pushes the other values toward 0, much like CLIP.
  - For LM, text tokens can only attend previous tokens (i.e., we use a causal mask), but they can attend all query tokens. However, query tokens cannot attend any text token. In other words, the query token outputs represent visual-only features, while text token outputs represent image-grounded causal text features. The model is trained using next token prediction: each text token's output goes through a classification head which must predict the next token in the caption.

You may be surprised that the Q-Former is used to encode text (for ITM and ITC) and also to generate text (for LM). Since the Q-Former is initialized using the weights of a pretrained BERT-base model, it's pretty good at text encoding right from the start of training, but it initially doesn't know that it has to predict the next token for the LM task. Luckily, it can learn fairly fast since it's not starting from scratch; it has good BERT features to work with. However, we need to tell it whether we want it to encode

the text or predict the next token. For this, we replace the class token with a *decode token* during LM.[38]

Once stage 1 is finished, the Q-Former is already a powerful model that can encode images and text into the same space, so a photo of a chimpanzee produces a very similar output representation as the caption "A photo of a chimpanzee". But it's even better than that: the query token outputs were trained to be most helpful for next token prediction.

To produce negative examples for ITM, one strategy is to randomly pick a caption in the same batch, excluding the image's true caption. However, this makes the task too easy, so the model doesn't learn much. Instead, the authors used a *hard negative mining* strategy, where difficult captions are more likely to be sampled. For example, given a photo of a chimpanzee, the caption "A gorilla" is more likely to be sampled than "A spacecraft". To find difficult captions, the algorithm uses the similarity scores from the ITC task.

So it's time for the second stage of training (see Figure 16-19):

- We keep the vision transformer and the Q-Former, but we drop the rest and we add a new linear layer, initialized randomly, on top of the Q-Former.
- For each image/caption pair, the Q-Former attends to the visual features produced by the pretrained vision encoder, and the outputs go through the linear layer to produce a sequence of visual query tokens.
- The visual query tokens and the text token representations are concatenated and fed to the (frozen) pretrained LLM. We train BLIP-2 to predict the next caption token.

During stage 2, the model learns to properly map the visual query tokens to the LLM's input space. Once trained, the model can be used like in stage 2, generating visual-grounded text.

38 The idea of training a single model capable of both encoding and generating text was introduced in 2019 by Microsoft researchers Li Dong et al. with their UniLM model.

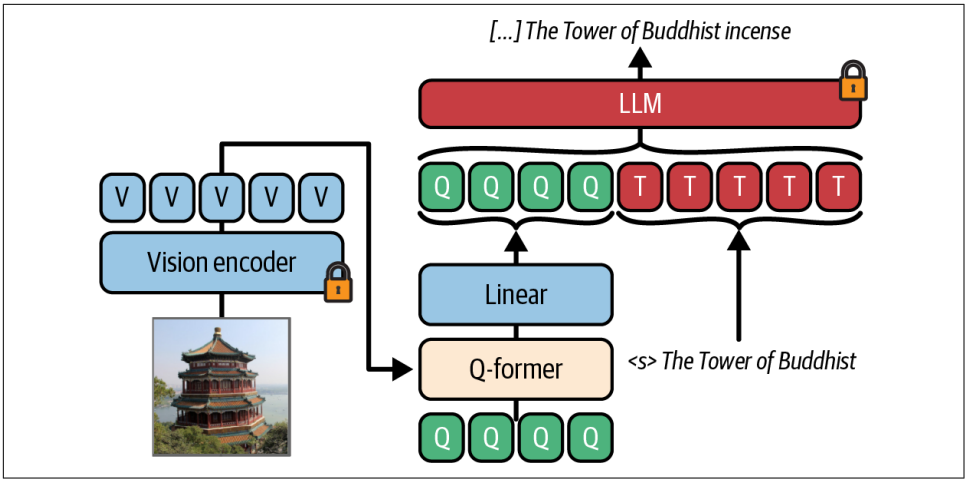

Figure 16-19. BLIP-2 pretraining, Stage 2: training the linear layer to map the query tokens to the LLM's input space

Let's use BLIP-2 to generate a caption for an image:

```
from transformers import Blip2Processor, Blip2ForConditionalGeneration

model_id = "Salesforce/blip2-opt-2.7b"
blip2_processor = Blip2Processor.from_pretrained(model_id)
blip2_model = Blip2ForConditionalGeneration.from_pretrained(
    model_id, device_map=device, dtype=torch.float16)

image_url = "http://images.cocodataset.org/val2017/000000039769.jpg"  # two cats
image = Image.open(urllib.request.urlopen(image_url))
inputs = blip2_processor(images=image, return_tensors="pt")
inputs = inputs.to(device, dtype=torch.float16)
with torch.no_grad():
    generated_ids = blip2_model.generate(**inputs)

generated_text = blip2_processor.batch_decode(generated_ids)
```

What did BLIP-2 see?

```
>>> generated_text
['<image><image><image><image>[...]<image></s>two cats laying on a couch\n']
```

It's a good description of the photo, but it would nicer without the special tokens, so let's get rid of them when decoding the model's output:

```
>>> generated_text = blip2_processor.batch_decode(generated_ids,
...                                               skip_special_tokens=True)
...
>>> generated_text
>>>
['two cats laying on a couch\n']
```

Perfect!

Also check out InstructBLIP, a BLIP-2 model with vision-language instruction tuning.

Other Multimodal Models

We've covered quite a few multimodal models, with very different architectures and pretraining techniques, but of course there are many others. Here is a quick overview of some of the most notable ones:

LayoutLM (Microsoft, Dec. 2019)
: Document understanding based on text, vision, and document layout. Version 3 was released in April 2022.

GLIP (Microsoft, Dec. 2021)
: A vision-language model for visual grounding and object detection. GLIP-2 was released in 2022.

Stable Diffusion (Stability AI, Dec. 2021)
: A powerful text-to-image model.

OFA (Microsoft, Feb. 2022)
: Unified (one for all) vision-language pretraining framework handling various vision-language tasks.

CoCa (Google, May 2022)
: A vision-language model pretrained using contrastive and captioning objectives. CoCa influenced later models like PaLI-X and Flamingo-2.

PaLI (Google, Sep. 2022)
: Multilingual multimodal models for vision-language tasks like VQA and captioning, with strong zero-shot performance. The next versions, PaLI-X and PaLI-3, were released in 2023, and PaliGemma in May 2024.

Kosmos-1 (Microsoft, Feb. 2023)
: A vision-language model with strong support for visual grounding. Kosmos-2 and Kosmos-2.5 came out in 2023.

PaLM-E (Google, Mar. 2023)
: PaLM-E extends Google's PaLM series with visual inputs and embodied sensor data. A decoder-only LLM generates text commands like "grab the hammer", which are interpreted and executed by a robot via a downstream system.

LLaVA (H. Liu et al., Apr. 2023)
: Among the best open source vision-language chat models.

ImageBind (Meta, May 2023)
: A CLIP-style model extended to six modalities (image, text, audio, IMU,[39] depth, and thermal).

RT-2 (DeepMind, Jul. 2023)
: A vision-language model capable of robotic control as well, trained on a large-scale instruction-following dataset.

SeamlessM4T (Meta, Aug. 2023)
: A single model that can perform speech-to-text, speech-to-speech, text-to-speech, and text-to-text translation across close to 100 languages.

Qwen-VL (Alibaba, Sep. 2023)
: Open vision-language family (7B to 72B) that became one of the strongest open multimodal baselines. Led to Qwen2-VL (Aug. 2024) and Qwen3-Omni (Sep. 2025), which expanded to video and audio and reached trillion-parameter scale.

Fuyu (Adept AI, Oct. 2023)
: Processes interleaved image and text in real time with a unified transformer.

EMO (Alibaba, Feb. 2024)
: Takes an image of a person, plus an audio recording of someone speaking or singing, and the model generates a video of that person, matching the audio. EMO-2 was released in January 2025.

GLaMM (H. Rasheed et al., Jun. 2024)
: A visual dialogue model which generates text responses mixed with object segmentation masks.

LaViDa (UCLA, Panasonic, Adobe, Salesforce, May 2025)
: A family of open, diffusion-based vision-language models.

I've created homl.info short links for all the models discussed in this chapter; just use the lowercase name without hyphens, for example, *https://homl.info/qwen2vl*.

There are also several commercial multimodal models whose detailed architectures were not disclosed, such as GPT-4.1 and Sora by OpenAI, Gemini 2.5 Pro by Google,

39 Most modern smartphones contain an inertial measurement unit (IMU) sensor: it measures acceleration, angular velocity, and often the magnetic field strength.

Veo-3 by DeepMind, and Claude 4 Opus by Anthropic. To access these models, you first need to create an account and get a subscription (or use the free tier), then you can either use the provided apps (e.g., Google AI Studio, *https://aistudio.google.com*), or query the model via an API. For example, following is a short code example showing how to query Gemini 2.5 Pro via the API. You first need to get an API key in Google AI Studio, then you can use any secret management method you prefer to store it and load it in your code (e.g., if you are using Colab, I recommend you use Colab's secret manager, as we saw in Chapter 15).

```
from google import genai

gemini_api_key = [...]  # load from Colab secrets, or from a file, or hardcode
gemini_client = genai.Client(api_key=gemini_api_key)
cats_photo = gemini_client.files.upload(file="my_cats_photo.jpg")
question = "What animal and how many? Format: [animal, number]"
response = gemini_client.models.generate_content(
    model="gemini-2.5-flash",  # or "gemini-2.5-pro"
    contents=[cats_photo, question])
print(response.text)  # prints: "[cat, 2]"
```

This code uses the `google-genai` library, which is already installed on Colab. It also assumes that a file named *my_cats_photo.jpg* is present in the same directory as the notebook.

This wraps up this chapter; I hope you enjoyed it. Transformers can now see, hear, touch, and more! In the next chapter, we will explore some fairly advanced techniques designed to speed up and scale transformers. As Daft Punk put it: harder, better, faster, stronger.

Exercises

1. Can you describe the original ViT's architecture? Why does it matter?
2. What tasks are regular ViTs (meaning nonhierarchical) best used for? What are their limitations?
3. What is the main innovation in DeiT? Is this idea generalizable to other architectures?
4. What are some examples of hierarchical ViTs? What kind of tasks are they good for?
5. How do PVTs and Swin Transformers reduce the computational cost of processing high-resolution images?
6. How does DINO work? What changed in DINOv2? When would you want to use DINOv2?
7. What is the objective of the JEPA architecture? How does it work?

8. What is a multimodal model? Can you give five examples of multimodal tasks?
9. Explain what the fusion and alignment problems are in multimodal learning. Why are transformers well suited to tackle them?
10. Can you write a one-line summary of the main ideas in VideoBERT, ViLBERT, CLIP, DALL·E, Perceiver IO, Flamingo, and BLIP-2?
11. If you are using a Perceiver IO model and you double the length of the inputs and the outputs, approximately how much more computation will be required?
12. Try fine-tuning a pretrained ViT model on the Food 101 dataset (`torchvision.datasets.Food101`). What accuracy can you reach? How about using a CLIP model, zero-shot?
13. Create a simple search engine for your own photos: first, write a function that uses a CLIP model to embed all of your photos and saves the resulting vectors. Next, write a function that takes a search query (text or image), embeds it using CLIP, then finds the most similar photo embeddings and displays the corresponding photos. You can manually implement the similarity search algorithm, or a dedicated library such as the FAISS library or even a full-blown vector database.
14. Use BLIP-2 to automatically caption all of your photos.

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

CHAPTER 17
Speeding Up Transformers

In Chapters 15 and 16, we built all kinds of transformers, from classifiers, translators and chatbots, to vision and multimodal transformers. While transformers are incredibly versatile and powerful, they are far from perfect. In particular, they can be very slow, especially when processing long input sequences.

Luckily, many techniques have been developed to speed up transformers of any size:

- To speed up decoding in generative transformers, we will use key/value caching and speculative decoding, then we will take of a quick look at several approaches to parallelize text generation.
- To accelerate multi-head attention (MHA), which is one of the most computationally expensive components of transformers, we will look at sparse attention, approximate attention, sharing projections, and FlashAttention.
- To speed up gigantic transformers of up to trillions of parameters, we will discuss mixture of experts (MoE).
- To train large transformers efficiently, we will discuss parameter-efficient fine-tuning (PEFT) using adapters such as Low-Rank Adaptation (LoRA), activation checkpointing, sequence packing, gradient accumulation, and parallelism.

Another way to speed up a transformer is to make it smaller. This can be done using reduced precision and quantization, which are discussed in Appendix B.

That's quite a lot of techniques to cover, and they are fairly advanced, so you can safely skip this chapter for now if you are new to transformers, and come back later whenever needed. This is why this chapter is online-only, available at *https://homl.info*, to leave room for the other chapters.

CHAPTER 18

Autoencoders, GANs, and Diffusion Models

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

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

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

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

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

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

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

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

Efficient Data Representations

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

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

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

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

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

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

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

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

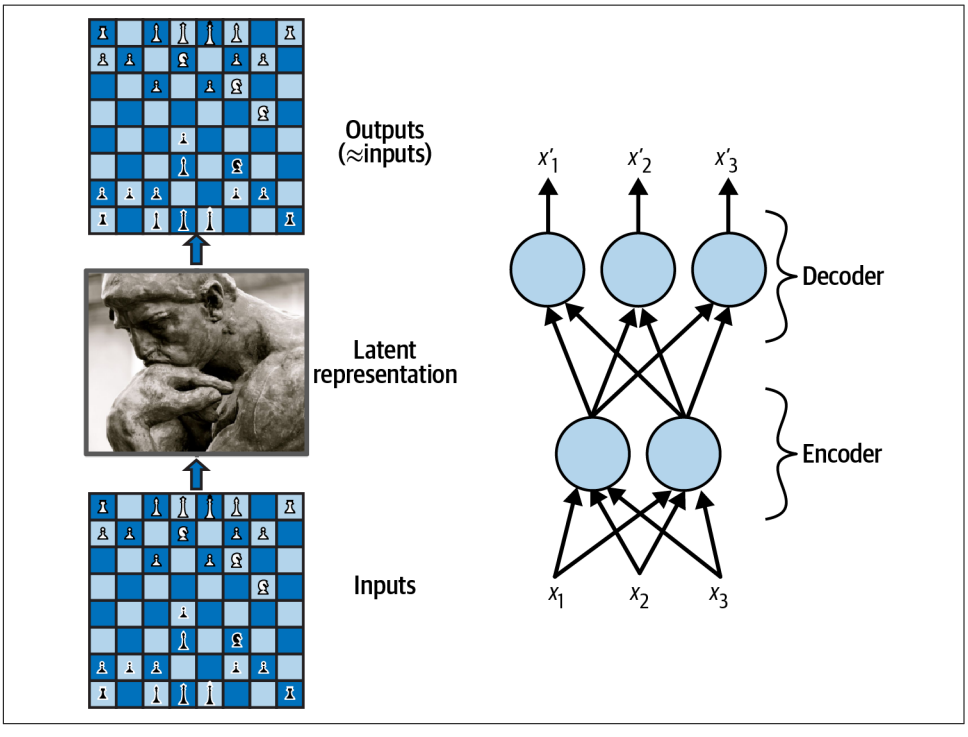

Figure 18-1. The chess memory experiment (left) and a simple autoencoder (right)

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

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

Performing PCA with an Undercomplete Linear Autoencoder

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

The following code builds a simple linear autoencoder that takes a 3D input, projects it down to 2D, then projects it back up to 3D. Since we will train the model using targets equal to the inputs, gradient descent will have to find the 2D plane that lies closest to the training data, just like PCA would.

```
import torch
import torch.nn as nn

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

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

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

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

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

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

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

```
import torchmetrics

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

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

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

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

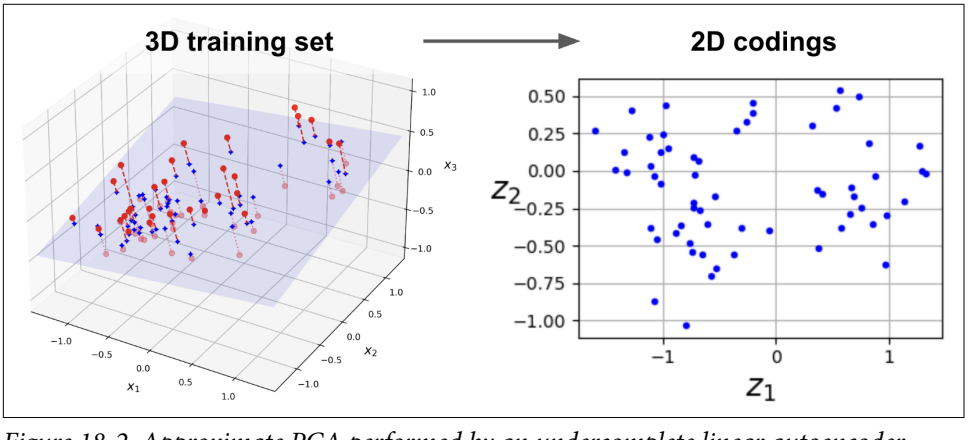

Figure 18-2. Approximate PCA performed by an undercomplete linear autoencoder

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

Stacked Autoencoders

Just like other neural networks we have discussed, autoencoders can have multiple hidden layers. In this case they are called *stacked autoencoders* (or *deep autoencoders*). Adding more layers helps the autoencoder learn more complex codings. That said,

one must be careful not to make the autoencoder too powerful. Imagine an encoder so powerful that it just learns to map each input to a single arbitrary number (and the decoder learns the reverse mapping). Obviously such an autoencoder will reconstruct the training data perfectly, but it will not have learned any useful data representation in the process, and is unlikely to generalize well to new instances.

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

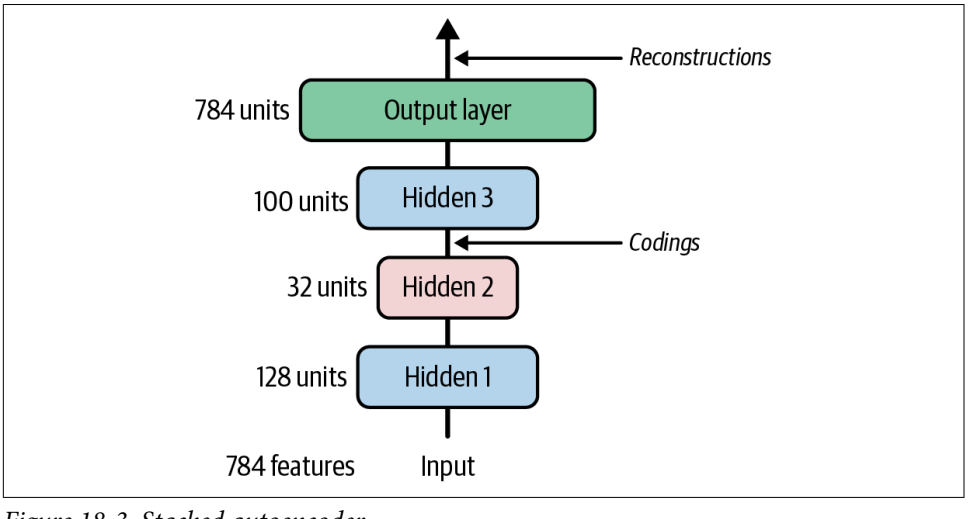

Figure 18-3. Stacked autoencoder

Implementing a Stacked Autoencoder Using PyTorch

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

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

```
)
stacked_ae = nn.Sequential(stacked_encoder, stacked_decoder).to(device)
```

Let's go through this code:

- Just like earlier, we split the autoencoder model into two submodels: the encoder and the decoder.
- The encoder takes 28 × 28 pixel grayscale images (i.e., with a single channel), flattens them so that each image is represented as a vector of size 784, then processes these vectors through 2 `Linear` layers of diminishing sizes (128 units, then 32 units), each followed by the ReLU activation function. For each input image, the encoder outputs a vector of size 32.
- The decoder takes codings of size 32 (output by the encoder) and processes them through 2 `Linear` layers of increasing sizes (128 units, then 784 units), and reshapes the final vectors into 1 × 28 × 28 arrays so the decoder's outputs have the same shape as the encoder's inputs. Note that we use the sigmoid function for the output layer instead of ReLU to ensure that the output pixel values range between 0 and 1.

We can now load the Fashion MNIST dataset using the TorchVision library and split it into `train_data`, `valid_data`, and `test_data` (just like we did in Chapter 10), then train the autoencoder exactly like the previous autoencoder, using the inputs as the targets and minimizing the MSE loss. Give it a try, it's a good exercise! Don't forget to change the targets so they match the inputs—we're training an autoencoder, not a classifier.[2] If you get stuck, please check out the implementation in this chapter's notebook.

Visualizing the Reconstructions

Once you have trained the stacked autoencoder, how do you know if it's any good? One way to check that an autoencoder is properly trained is to compare the inputs and the outputs: the differences should not be too significant. Let's plot a few images from the validation set, as well as their reconstructions:

```
import matplotlib.pyplot as plt

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

2 Hint: one approach is to create a custom `AutoencoderDataset` class that wraps a given dataset and replaces the targets with the inputs.

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

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

X_valid = torch.stack([x for x, _ in valid_data])
plot_reconstructions(stacked_ae, X_valid)
plt.show()
```

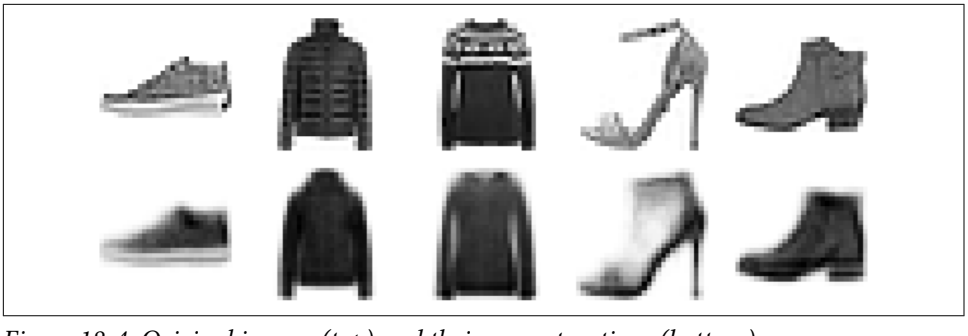

Figure 18-4. Original images (top) and their reconstructions (bottom)

Figure 18-4 shows the resulting images. The reconstructions are recognizable, but a bit too lossy. We may need to train the model for longer, or make the encoder and decoder more powerful, or make the codings larger. For now, let's go with this model and see how we can use it.

Anomaly Detection Using Autoencoders

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

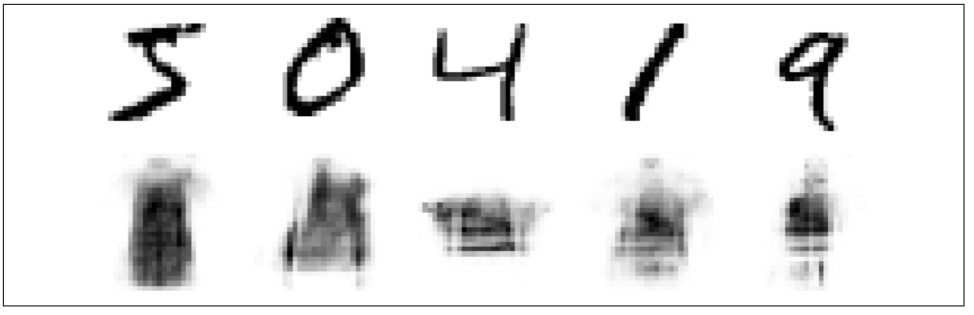

Figure 18-5. Out-of-distribution images are poorly reconstructed

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

Visualizing the Fashion MNIST Dataset

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

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

```
from sklearn.manifold import TSNE

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

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

Now we can plot the dataset:

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

Figure 18-6 shows the resulting scatterplot, beautified a bit by displaying some of the images. The t-SNE algorithm identified several clusters that match the classes reasonably well (each class is represented by a different color). Note that t-SNE's output can vary greatly if you run it with a different random seed, slightly different data, or on a different platform, so your plot may look different.

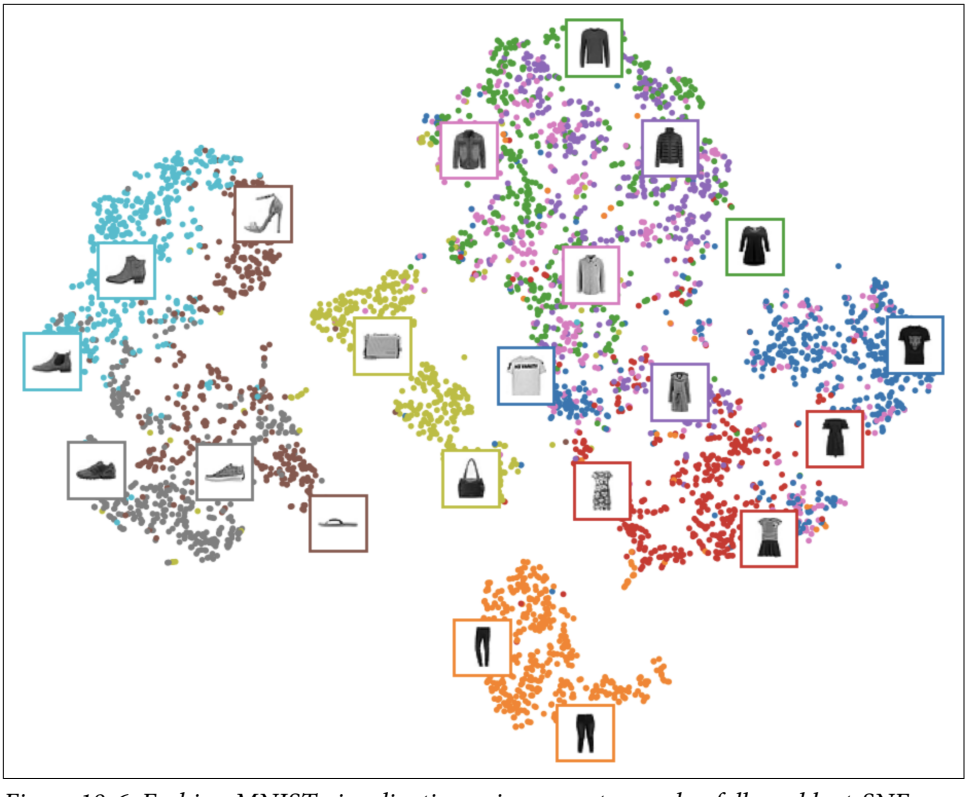

Figure 18-6. Fashion MNIST visualization using an autoencoder, followed by t-SNE

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

Unsupervised Pretraining Using Stacked Autoencoders

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

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

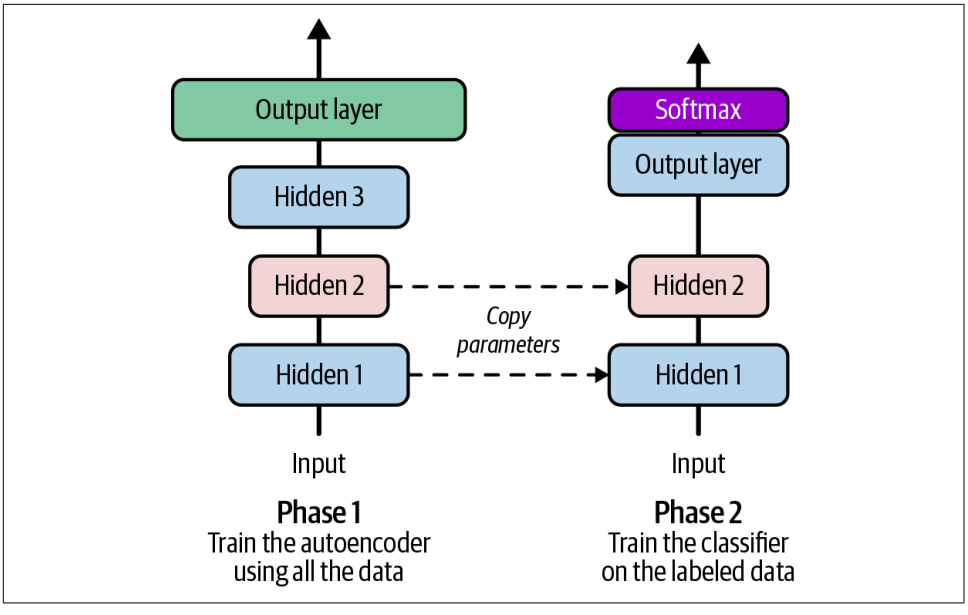

Figure 18-7. Unsupervised pretraining using autoencoders

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

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

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

Tying Weights

When an autoencoder is neatly symmetrical, like the one we just built, a common technique is to *tie* the weights of the decoder layers to the weights of the encoder layers. This halves the number of weights in the model, speeding up training and limiting the risk of overfitting. Specifically, if the autoencoder has a total of N layers (not counting the input layer), and $\mathbf{W}_L$ represents the connection weights of the L^{th}

layer (e.g., layer 1 is the first hidden layer, layer $N/2$ is the coding layer, and layer N is the output layer), then the decoder layer weights can be defined as $\mathbf{W}_L = \mathbf{W}_{N-L+1}^\intercal$ (with $L = N / 2 + 1, \ldots, N$).

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

```
import torch.nn.functional as F

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

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

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

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

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

Training One Autoencoder at a Time

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

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

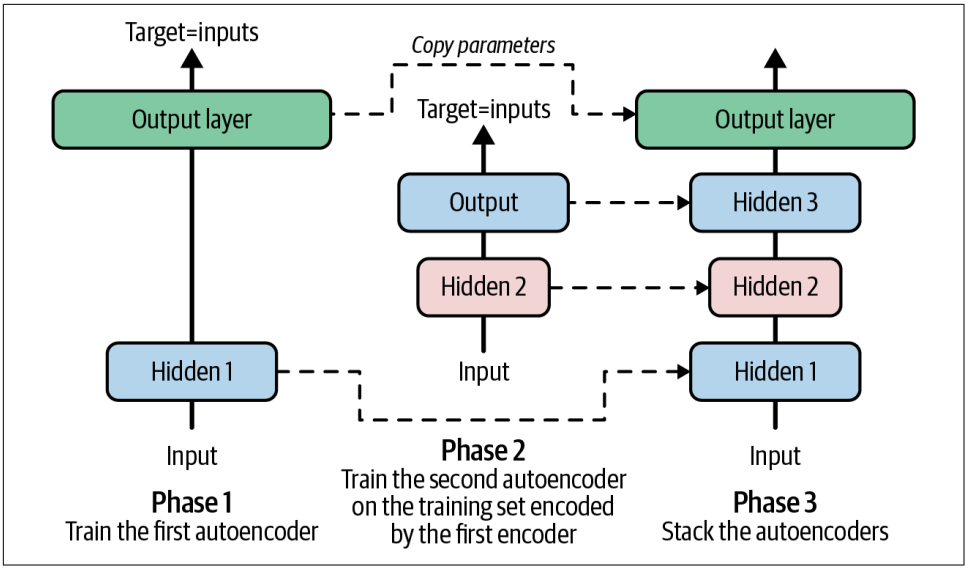

Figure 18-8. Training one autoencoder at a time

As I mentioned in Chapter 11, one of the triggers of the deep learning tsunami was the discovery in 2006 by Geoffrey Hinton et al. that deep neural networks can be pretrained in an unsupervised fashion using this greedy layer-wise approach. They used restricted Boltzmann machines (RBMs; see *https://homl.info/extra-anns*) for this purpose, but in 2007 Yoshua Bengio et al.[3] showed that autoencoders worked just as well. For several years this was the only efficient way to train deep nets, until many of the techniques introduced in Chapter 11 made it possible to just train a deep net in one shot.

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

Convolutional Autoencoders

If you are dealing with images, then the autoencoders we have seen so far will not work well (unless the images are very small). As you saw in Chapter 12, convolutional neural networks are far better suited than dense networks to working with images. So if you want to build an autoencoder for images (e.g., for unsupervised pretraining or dimensionality reduction), you will need to build a *convolutional autoencoder*.[4]

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

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

The encoder is a regular CNN composed of convolutional layers and pooling layers. It typically reduces the spatial dimensionality of the inputs (i.e., height and width) while increasing the depth (i.e., the number of feature maps). The decoder must do the reverse (upscale the image and reduce its depth back to the original dimensions), and for this you can use transpose convolutional layers (alternatively, you could combine upsampling layers with convolutional layers). Here is a basic convolutional autoencoder for Fashion MNIST:

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

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

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

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

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

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

Denoising Autoencoders

A simple way to force the autoencoder to learn useful features is to add noise to its inputs, training it to recover the original, noise-free inputs. This idea has been around since the 1980s (e.g., it is mentioned in Yann LeCun's 1987 master's thesis). In a 2008 paper,[5] Pascal Vincent et al. showed that autoencoders could also be used for feature extraction. In a 2010 paper,[6] Vincent et al. introduced *stacked denoising autoencoders*.

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

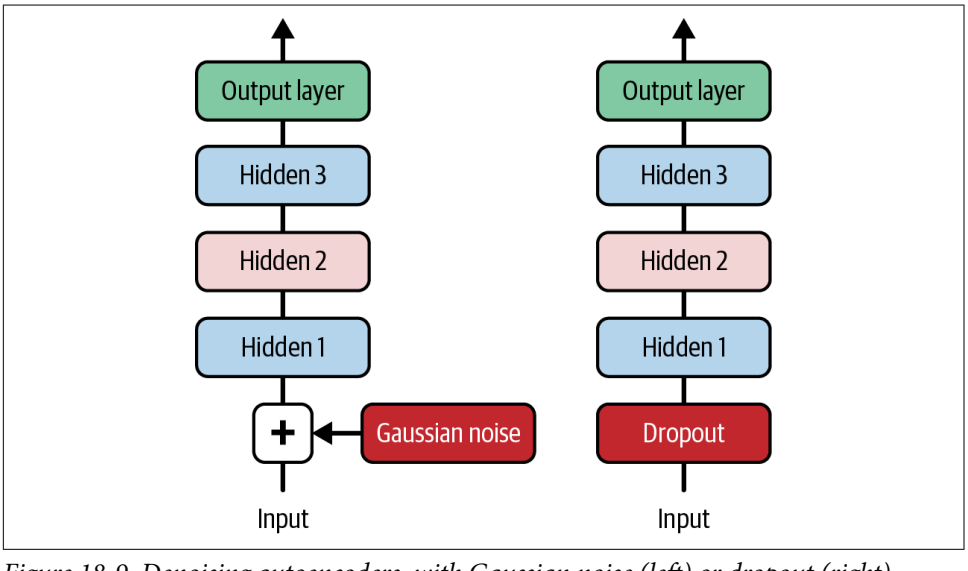

Figure 18-9. Denoising autoencoders, with Gaussian noise (left) or dropout (right)

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

```
dropout_encoder = nn.Sequential(
    nn.Flatten(),
    nn.Dropout(0.5),
```

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

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

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

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

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

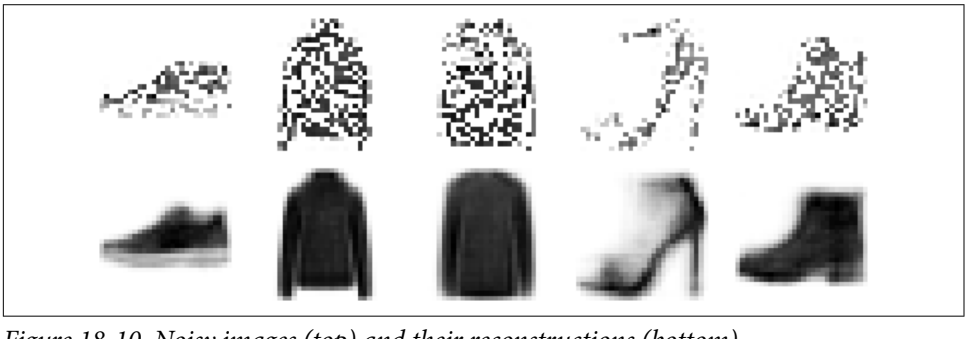

Figure 18-10. Noisy images (top) and their reconstructions (bottom)

Sparse Autoencoders

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

A basic approach is to use the sigmoid activation function in the coding layer (to constrain the codings to values between 0 and 1), use a large coding layer (e.g., with 256 units), and add some ℓ_1 regularization to the coding layer's activations. This means adding the ℓ_1 norm of the codings (i.e., the sum of their absolute values) to the loss, weighted by a sparsity hyperparameter. This *sparsity loss* will encourage the neural network to produce codings close to 0. However, the total loss will still include the reconstruction loss, so the model will be forced to output at least a few nonzero values to reconstruct the inputs correctly. Using the ℓ_1 norm rather than the ℓ_2 norm will push the neural network to preserve the most important codings while eliminating the ones that are not needed for the input image (rather than just reducing all codings).

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

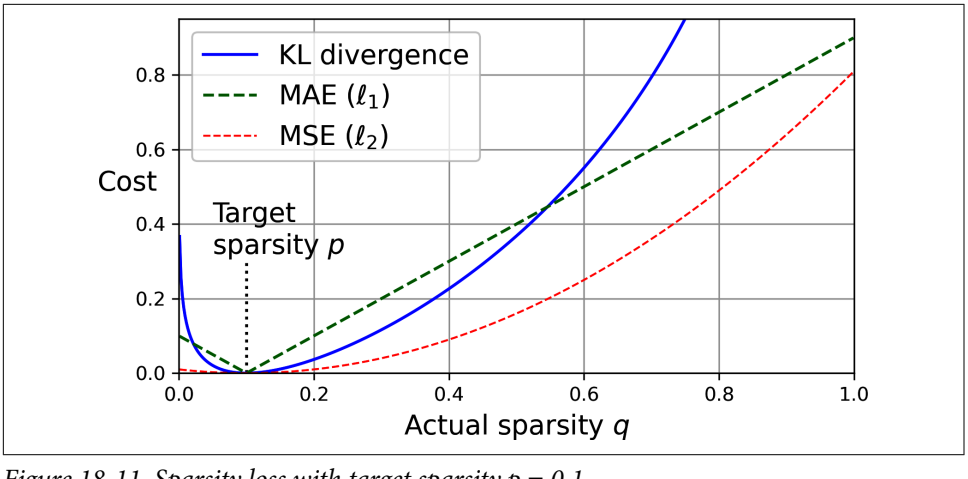

Figure 18-11. Sparsity loss with target sparsity $p = 0.1$

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

Equation 18-1. Kullback–Leibler divergence

$$D_{\mathrm{KL}}(P \parallel Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)}$$

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

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

$$D_{\mathrm{KL}}(p \parallel q) = p \log \frac{p}{q} + (1 - p) \log \frac{1 - p}{1 - q}$$

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

```
from collections import namedtuple

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

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

    def forward(self, X):
        codings = self.encoder(X)
        output = self.decoder(codings)
        return AEOutput(output, codings)
```

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

Next, we can define the loss function:

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

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

Now we're ready to create the model and train it (using the same `train()` function as earlier, from Chapter 10):

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

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

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

Now let's move on to variational autoencoders!

Variational Autoencoders

An important category of autoencoders was introduced in 2013 by Diederik Kingma and Max Welling[7] and quickly became one of the most popular variants: *variational autoencoders* (VAEs).

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

- They are *probabilistic autoencoders*, meaning that their outputs are partly determined by chance, even after training (as opposed to denoising autoencoders, which use randomness only during training).
- Most importantly, they are *generative autoencoders*, meaning that they can generate new instances that look like they were sampled from the training set.[8]

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

As you can see in Figure 18-12, although the inputs may have a very convoluted distribution, a variational autoencoder tends to produce codings that look as though they were sampled from a simple Gaussian distribution. During training, the cost function (discussed next) pushes the codings to gradually migrate within the coding space (also called the *latent space*) to end up looking like a cloud of multidimensional Gaussian points. One great consequence is that after training a variational autoencoder, you can very easily generate a new instance: just sample a random coding from the Gaussian distribution, decode it, and voilà!

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

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

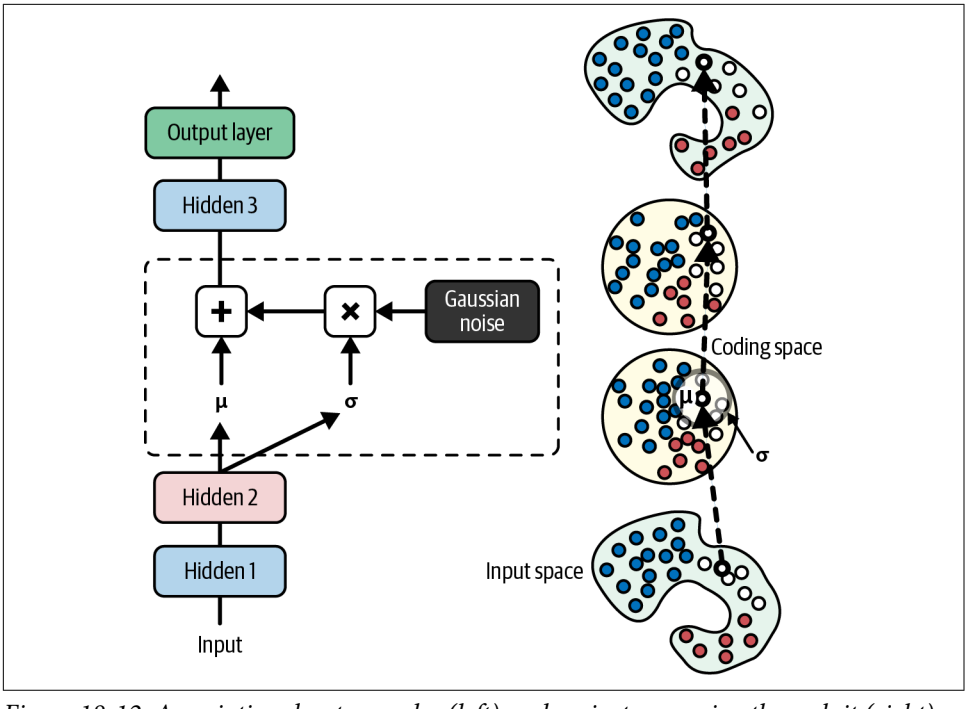

Figure 18-12. A variational autoencoder (left) and an instance going through it (right)

Sampling from a random distribution is not a differentiable operation, it will block backpropagation, so how can we hope to train the encoder? Well, using a *reparameterization trick*: sample $\boldsymbol{\varepsilon}$ from $\mathcal{N}(0, 1)$ and compute $\boldsymbol{\mu} + \boldsymbol{\sigma} \otimes \boldsymbol{\varepsilon}$ (element-wise multiplication). This is equivalent to sampling from $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\sigma}^2)$ but it separates the deterministic and stochastic parts of the process, allowing the gradients to flow back into the encoder through $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$. The resulting encoder gradients are stochastic (due to $\boldsymbol{\varepsilon}$), but they are unbiased estimates, and the randomness averages out during training.

The cost function is composed of two parts. The first is the usual reconstruction loss that pushes the autoencoder to reproduce its inputs. We can use the MSE for this, as we did earlier. The second is the *latent loss* that pushes the autoencoder to have codings that look as though they were sampled from a simple Gaussian distribution: it is the KL divergence between the actual distribution of the codings and the desired latent distribution (i.e., the Gaussian distribution). The math is a bit more complex than with the sparse autoencoder, in particular because of the Gaussian noise, which limits the amount of information that can be transmitted to the coding layer. Luckily, the equations simplify, so the latent loss can be computed

using Equation 18-3 (for the full mathematical details, check out the original paper on variational autoencoders, or Carl Doersch's great 2016 tutorial.)

Equation 18-3. Variational autoencoder's latent loss

$$\mathcal{L} = -\frac{1}{2}\sum_{i=1}^{n}\left[1 + \log\left(\sigma_i^2\right) - \sigma_i^2 - \mu_i^2\right]$$

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

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

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

$$\mathcal{L} = -\frac{1}{2}\sum_{i=1}^{n}\left[1 + \gamma_i - \exp(\gamma_i) - \mu_i^2\right]$$

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

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

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

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

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

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

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

Let's go through this code:

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

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

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

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

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

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

Generating Fashion MNIST Images

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

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

Figure 18-13 shows the 21 generated images.

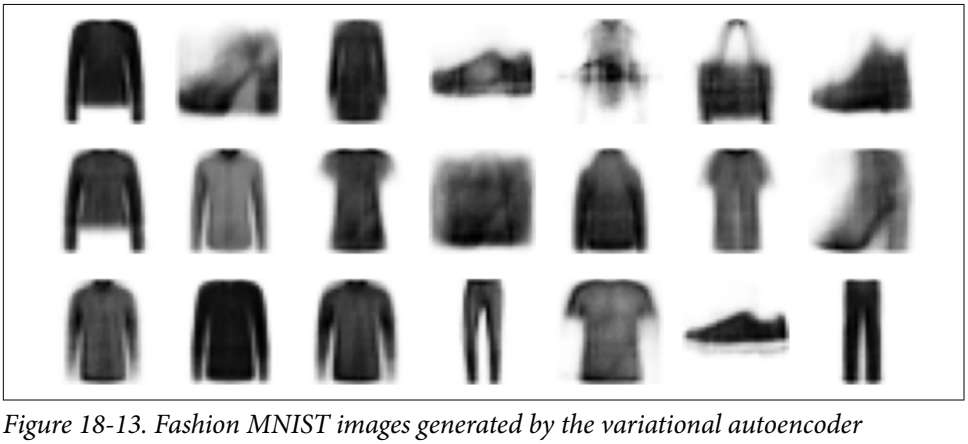

Figure 18-13. Fashion MNIST images generated by the variational autoencoder

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

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

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

Figure 18-14. Semantic interpolation

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

Discrete Variational Autoencoders

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

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

- First, the encoder must output logits rather than means and variances. For each input image, it outputs a tensor of shape $[d, k]$ containing logits, for example $[[1.2, -0.8, 0.5], [-1.3, 0.4, 0.3]]$ if $d = 2$ and $k = 3$.

- Second, since categorical sampling is not a differentiable operation, we must once again use a reparameterization trick, but we cannot reuse the same as for regular VAEs: we need one designed for categorical distributions. The most popular one is the Gumbel-softmax trick. Instead of directly sampling from the categorical distribution, we call the `F.gumble_softmax()` function: this implements a differentiable approximation of categorical sampling. Given the previous logits, this function might output the discrete coding vector [0, 2].

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

This idea was proposed in 2016 almost simultaneously by two independent teams of researchers, one from DeepMind and Oxford University,[9] the other from Google, Cambridge University, and Stanford University.[10]

Let's implement a dVAE for Fashion MNIST:

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

class DiscreteVAE(nn.Module):
    def __init__(self, coding_length=32, n_codes=16, temperature=1.0):
        super().__init__()
        self.coding_length = coding_length
        self.n_codes = n_codes
        self.temperature = temperature
        self.encoder = nn.Sequential([...])  # outputs [coding_length, n_codes]
        self.decoder = nn.Sequential([...])  # outputs [1, 28, 28]

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

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

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

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

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

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

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

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

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

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

Another popular approach to discrete VAEs is called *vector quantization* (VQ-VAE), proposed by DeepMind researchers in 2017.[11] Instead of producing logits, the encoder outputs d embeddings, each of dimensionality e. Then instead of sampling from a categorical distribution, the VQ-VAE maps each embedding to the index of the nearest embedding in a trainable embedding matrix of shape $[k, e]$, called the

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

codebook. This produces the integer codes. Finally, these codes are embedded using the embedding matrix and passed on to the decoder.

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

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

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

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

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

sequence of 1,024 codes, organize them into a 32 × 32 grid, replace each code with its codebook vector, then pass the result to the decoder to generate the final image.

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

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

Generative Adversarial Networks

Generative adversarial networks were proposed in a 2014 paper[12] by Ian Goodfellow et al., and although the idea got researchers excited almost instantly, it took a few years to overcome some of the difficulties of training GANs. Like many great ideas, it seems simple in hindsight: make neural networks compete against each other in the hope that this competition will push them to excel. As shown in Figure 18-15, a GAN is composed of two neural networks:

Generator
: Takes a random coding as input (typically sampled from a Gaussian distribution) and outputs some data—typically, an image. The coding is the latent representation of the image to be generated. So, as you can see, the generator offers the same functionality as a decoder in a variational autoencoder, and it can be used in the same way to generate new images: just feed it a random vector, and it outputs a brand-new image. However, it is trained very differently, as you will soon see.

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

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

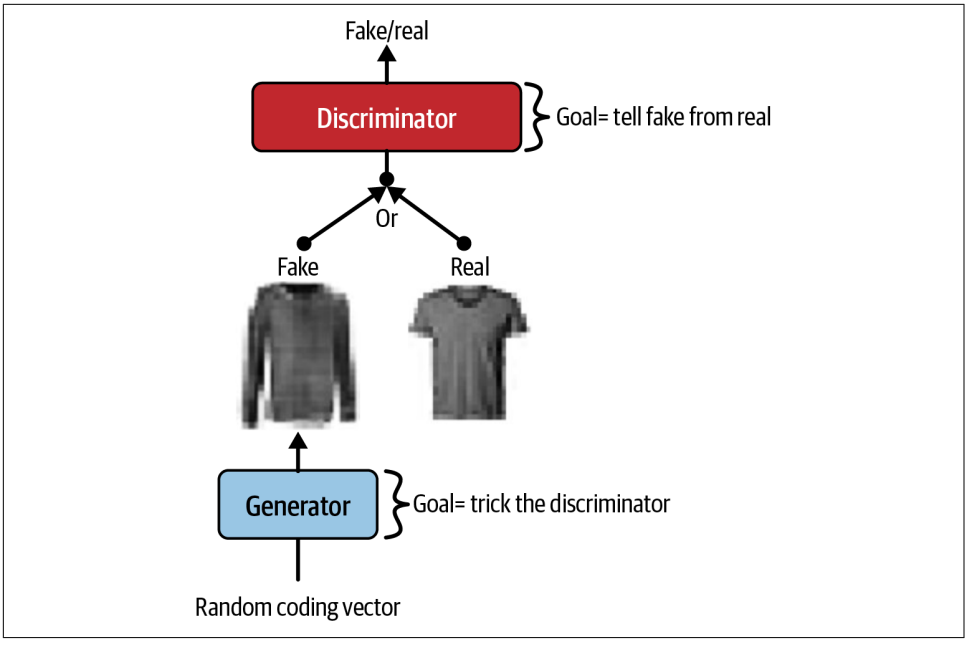

Figure 18-15. A generative adversarial network

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

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

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

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

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

First, we need to build the generator and the discriminator. The generator is similar to an autoencoder's decoder—it takes a coding vector as input and outputs an image—and the discriminator is a regular binary classifier—it takes an image as input and ends with a dense layer containing a single unit and using the sigmoid activation function:

```
codings_dim = 32
generator = nn.Sequential(
    nn.Linear(codings_dim, 128), nn.ReLU(),
    nn.Linear(128, 256), nn.ReLU(),
    nn.Linear(256, 1 * 28 * 28), nn.Sigmoid(),
    nn.Unflatten(dim=1, unflattened_size=(1, 28, 28))).to(device)
discriminator = nn.Sequential(
    nn.Flatten(),
    nn.Linear(1 * 28 * 28, 256), nn.ReLU(),
    nn.Linear(256, 128), nn.ReLU(),
    nn.Linear(128, 1), nn.Sigmoid()).to(device)
```

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

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

```
        discriminator_opt.step()

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

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

Phase one

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

Phase two

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

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

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

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

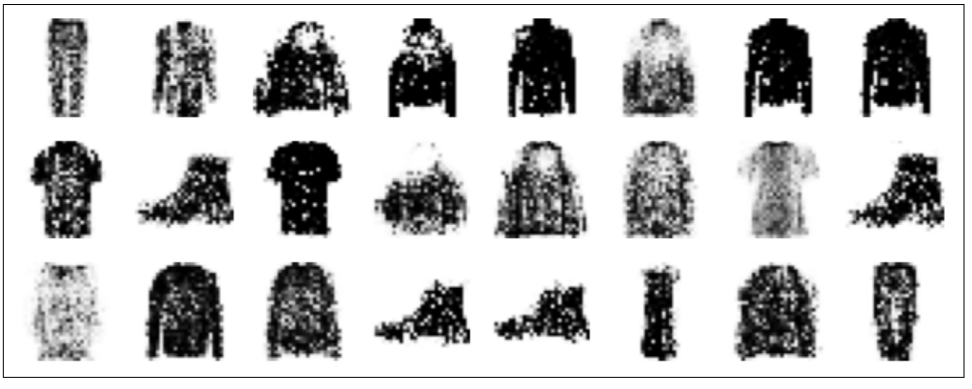

Figure 18-16. Images generated by the GAN after one epoch of training

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

The Difficulties of Training GANs

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

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

The biggest difficulty is called *mode collapse*: when the generator's outputs gradually become less diverse. How can this happen? Suppose the generator gets better at producing convincing shoes than any other class. It will fool the discriminator a bit more with shoes, and this will encourage it to produce even more images of shoes. Gradually, it will forget how to produce anything else. Meanwhile, the only fake images that the discriminator will see will be shoes, so it will also forget how to discriminate fake images of other classes. Eventually, when the discriminator manages to discriminate the fake shoes from the real ones, the generator will be forced to move to another class. It may then become good at shirts, forgetting about shoes, and the discriminator will follow. The GAN may gradually cycle across a few classes, never really becoming very good at any of them (see the top row of Figure 18-17).

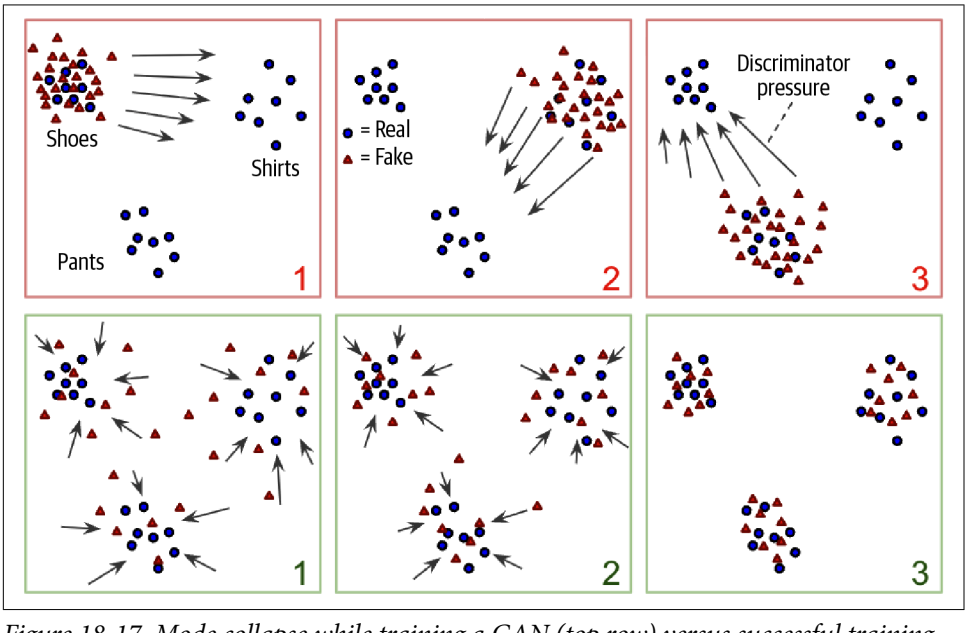

Figure 18-17. Mode collapse while training a GAN (top row) versus successful training without mode collapse (bottom row)

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

These problems have kept researchers very busy since 2014. Many papers have been published on this topic, some proposing new cost functions[13] (though a 2018 paper[14] by Google researchers questions their efficiency) or techniques to stabilize training or to avoid the mode collapse issue. For example, a popular technique called *experience replay* consists of storing the images produced by the generator at each iteration in a replay buffer (gradually dropping older generated images) and training the discriminator using real images plus fake images drawn from this buffer (rather than just fake images produced by the current generator). This reduces the chances that the discriminator will overfit the latest generator's outputs. Another common technique is called *mini-batch discrimination*: it measures how similar images are across the batch and provides this statistic to the discriminator, so it can easily reject a whole batch of fake images that lack diversity. This encourages the generator to produce a greater variety of images, reducing the chance of mode collapse (see the bottom row of Figure 18-17).

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

Diffusion Models

The ideas behind diffusion models have been around for many years, but they were first formalized in their modern form in a 2015 paper[15] by Jascha Sohl-Dickstein et al. from Stanford University and UC Berkeley. The authors applied tools from statistical mechanics to model a diffusion process, similar to a drop of milk diffusing in a cup of tea. The core idea is to train a model to learn the reverse process: start from the completely mixed state and gradually "unmix" the milk from the tea. Using this idea, they obtained promising results in image generation, but since GANs produced more convincing images back then, and they did so much faster, diffusion models did not get as much attention.

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

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

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

Then, in 2020, Jonathan Ho et al., also from UC Berkeley, managed to build a diffusion model capable of generating highly realistic images, which they called a *denoising diffusion probabilistic model* (DDPM).[16] A few months later, a 2021 paper[17] by OpenAI researchers Alex Nichol and Prafulla Dhariwal analyzed the DDPM architecture and proposed several improvements that allowed DDPMs to finally beat GANs: not only are DDPMs much easier to train than GANs, but the generated images are more diverse and of even higher quality. The main downside of DDPMs, as you will see, is that they take a very long time to generate images, compared to GANs or VAEs.

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

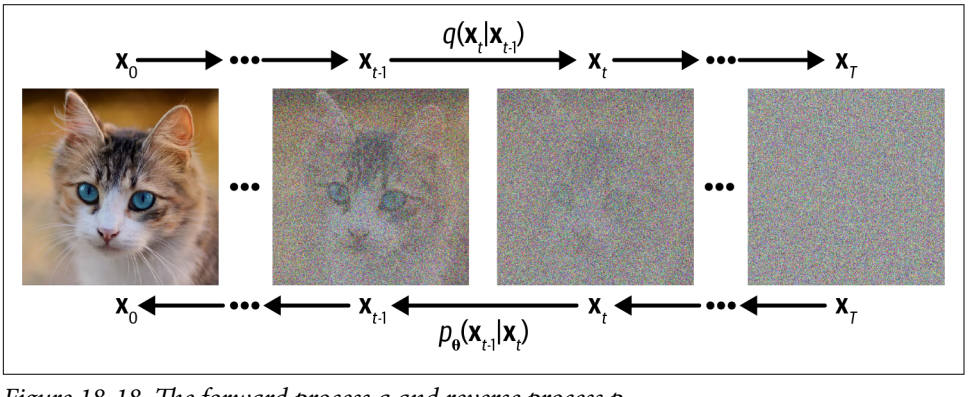

Figure 18-18. The forward process q and reverse process p

As we add more and more Gaussian noise in the forward process, the distribution of pixel values becomes more and more Gaussian. One important detail I left out is that the pixel values get rescaled slightly at each step, by a factor of $\sqrt{1-\beta_t}$. This ensures that the mean of the pixel values gradually approaches 0, since the scaling factor is a bit smaller than 1 (imagine repeatedly multiplying a number by 0.99). It also ensures

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

17 Alex Nichol and Prafulla Dhariwal, "Improved Denoising Diffusion Probabilistic Models" (2021).

that the variance will gradually converge to 1. This is because the standard deviation of the pixel values also gets scaled by $\sqrt{1-\beta_t}$, so the variance gets scaled by $1-\beta_t$ (i.e., the square of the scaling factor). But the variance cannot shrink to 0 since we're adding Gaussian noise with variance β_t at each step. And since variances add up when you sum Gaussian distributions, the variance must converge to $1-\beta_t+\beta_t=1$.

The forward diffusion process is summarized in Equation 18-5. This equation won't teach you anything new about the forward process, but it's useful to understand this type of mathematical notation, as it's often used in ML papers. This equation defines the probability distribution q of $\mathbf{x}_t$, given $\mathbf{x}_{t-1}$ as a Gaussian distribution with mean $\mathbf{x}_{t-1}$ times the scaling factor, and with a covariance matrix equal to $\beta_t\mathbf{I}$. This is the identity matrix $\mathbf{I}$ multiplied by β_t, which means that the noise is isotropic with variance β_t.

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

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

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

Equation 18-6. Shortcut for the forward diffusion process

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

Our goal, of course, is not to drown cats in noise. On the contrary, we want to create many new cats! We can do so by training a model that can perform the *reverse process*: going from $\mathbf{x}_t$ to $\mathbf{x}_{t-1}$. We can then use it to remove a tiny bit of noise from an image, and repeat the operation many times until all the noise is gone. It's not a basic noise filter that relies only on the neighboring pixels: instead, when noise is removed, it is replaced with realistic pixels, depending on the training data. For example, if we train the model on a dataset containing many cat images, then we can give it a picture entirely full of Gaussian noise, and the model will gradually make a brand new cat appear (see Figure 18-18).

OK, so let's start coding! The first thing we need to do is to code the forward process. For this, we will first need to implement the variance schedule. How can we control how fast the cat disappears? At each time step t, the pixel values get multiplied by $\sqrt{1-\beta_t}$ and noise with mean 0 and variance β_t gets added (as explained earlier). So, the part of the image's variance that comes from the original cat image shrinks by a

factor of $\alpha_t = 1 - \backslash beta\_t$ at each step. After t time steps, it will have shrunk by a factor of $\bar{\alpha}_t = \alpha_1 \times \alpha_2 \times \ldots \times \alpha_t$. It's this "cat signal" factor $\bar{\alpha}_t$ that we want to schedule so it shrinks down from 1 to 0 gradually between time steps 0 and T. In the improved DDPM paper, the authors schedule $\bar{\alpha}_t$ according to Equation 18-7. This schedule is represented in Figure 18-19.

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

$$\beta_t = 1 - \frac{\bar{\alpha}_t}{\bar{\alpha}_{t-1}} \quad \text{with } \bar{\alpha}_t = \frac{f(t)}{f(0)} \text{ and } f(t) = \cos^2\left(\frac{\frac{t}{T} + s}{1 + s} \cdot \frac{\pi}{2}\right)$$

In this equation:

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

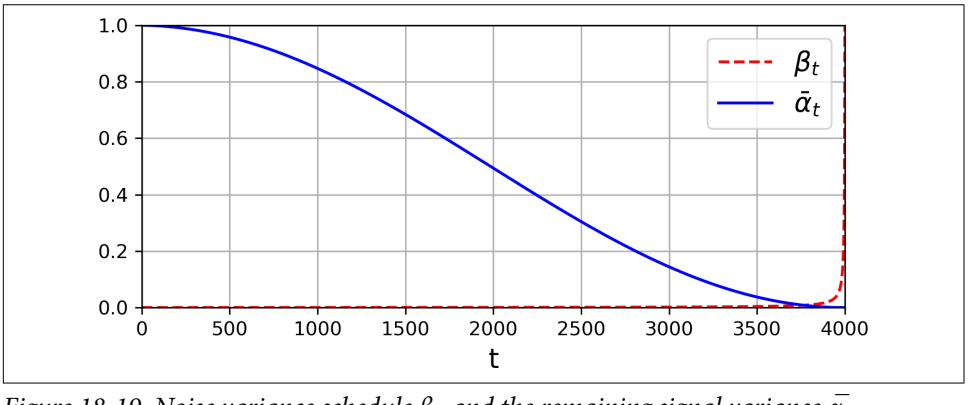

Figure 18-19. Noise variance schedule β_t, and the remaining signal variance $\bar{\alpha}_t$

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

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

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

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

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

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

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

Next, let's create a dataset wrapper class. It takes an image dataset—Fashion MNIST in our case—and preprocesses the images so their pixel values range between –1 and +1 (this is optional but usually works better), and it uses the `forward_diffusion()` function to add noise to the image. Then it wraps the resulting noisy image as well as the time step in a `DiffusionSample` object, and returns it along with the target, which is the unscaled noise `eps`, before it was scaled by $\sqrt{1-\bar{\alpha}_t}$ and added to the image:

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

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

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

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

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

Now we're ready to build the actual diffusion model itself. It can be any model you want, as long as it takes a `DiffusionSample` as input and outputs images of the same shape as the input images. The DDPM authors used a modified U-Net architecture,[18] which has many similarities with the FCN architecture we discussed in Chapter 12 for semantic segmentation. U-Net is a convolutional neural network that gradually downsamples the input images, then gradually upsamples them again, with skip connections crossing over from each level of the downsampling part to the corresponding level in the upsampling part. To take into account the time steps, they were encoded using a fixed sinusoidal encoding (i.e., the same technique as the positional encodings in the original Transformer architecture). At every level in the U-Net architecture, they passed these time encodings through `Linear` layers and fed them to the U-Net. Lastly, they also used multi-head attention layers at various levels. See this chapter's notebook for a basic implementation (it's too long to copy here, and the details don't matter: many other model architectures would work just fine).

```
class DiffusionModel(nn.Module):  # see the notebook for full details
    def __init__(self, T=T, embed_dim=64):
        [...]  # create all the required modules to build the U-Net

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

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

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

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

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

$$\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\varepsilon}_{\boldsymbol{\theta}}(\mathbf{x}_t, t)\right) + \sqrt{\beta_t}\mathbf{z}$$

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

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

This works well, but it requires 4,000 iterations to generate an image! That's too slow. Luckily, just a few months after the DDPM paper, researchers from Stanford University proposed a technique named the denoising diffusion implicit model (DDIM)[19] to generate images in much fewer steps: instead of going from $t = 4{,}000$ down to 0 one step at a time, DDIM can go down any number of time steps at a time, using Equation 18-9. Moreover, the training process is exactly the same as for DDPM, so we can simply reuse our trained DDPM model.

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

$$\mathbf{x}_p = \sqrt{\bar{\alpha}_p}\hat{\mathbf{x}}_0 + \sqrt{1 - \bar{\alpha}_p - \sigma^2} \cdot \boldsymbol{\varepsilon}_{\boldsymbol{\theta}}(\mathbf{x}_t, t) + \sigma\mathbf{z}$$

$$\text{where } \hat{\mathbf{x}}_0 = \frac{1}{\sqrt{\bar{\alpha}_t}}\left(\mathbf{x}_t - \sqrt{1 - \bar{\alpha}_t}\boldsymbol{\varepsilon}_{\boldsymbol{\theta}}(\mathbf{x}_t, t)\right)$$

$$\text{and } \sigma^2 = \eta\left(\frac{1 - \bar{\alpha}_p}{1 - \bar{\alpha}_t}\right)\beta_t$$

In this equation:

- $\boldsymbol{\varepsilon}_{\boldsymbol{\theta}}(\mathbf{x}_t, t)$, $\boldsymbol{\theta}$, and $\mathbf{z}$ have the same meanings as in Equation 18-8.
- p represents any time step before t. For example, it could be $p = t - 50$.
- η is a hyperparameter that controls how much randomness should be used during generation, from 0 (no randomness, fully deterministic) to 1 (just like DDPM).

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

```
def generate_ddim(model, batch_size=32, num_steps=50, eta=0.85):
    model.eval()
    with torch.no_grad():
        xt = torch.randn([batch_size, 1, 28, 28], device=device)
        times = torch.linspace(T - 1, 0, steps=num_steps + 1).long().tolist()
        for t, t_prev in zip(times[:-1], times[1:]):
            t_batch = torch.full((batch_size, 1), t, device=device)
            sample = DiffusionSample(xt, t_batch)
            eps_pred = model(sample)
```

19 Jiaming Song et al., "Denoising Diffusion Implicit Models", arXiv preprint arXiv:2010.02502 (2020).

```
            x0 = ((xt - (1 - alpha_bars[t]).sqrt() * eps_pred)
                  / (alpha_bars[t].sqrt()))
            abar_t_prev = alpha_bars[t_prev]
            variance = eta * (1 - abar_t_prev) / (1 - alpha_bars[t]) * betas[t]
            sigma_t = variance.sqrt()
            pred_dir = (1 - abar_t_prev - sigma_t**2).sqrt() * eps_pred
            noise = torch.randn_like(xt)
            xt = abar_t_prev.sqrt() * x0 + pred_dir + sigma_t * noise

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

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

This time the generation will only take a few seconds, and it will produce images such as the ones shown in Figure 18-20. Granted, they're not very impressive, but we've only trained the model for a few minutes on Fashion MNIST. Give it a try on a larger dataset and train it for a few hours to get more impressive results.

Figure 18-20. Images generated by DDIM accelerated diffusion

Diffusion models have made tremendous progress since 2020. In particular, a paper published in December 2021 by Robin Rombach, et al.[20] introduced *latent diffusion models*, where the diffusion process takes place in latent space, rather than in pixel space. To achieve this, a powerful autoencoder is used to compress each training image into a much smaller latent space, where the diffusion process takes place, then the autoencoder is used to decompress the final latent representation, generating the output image. This considerably speeds up image generation, and reduces training time and cost dramatically. Importantly, the quality of the generated images is outstanding.

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

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

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

```
from diffusers import AutoPipelineForText2Image

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

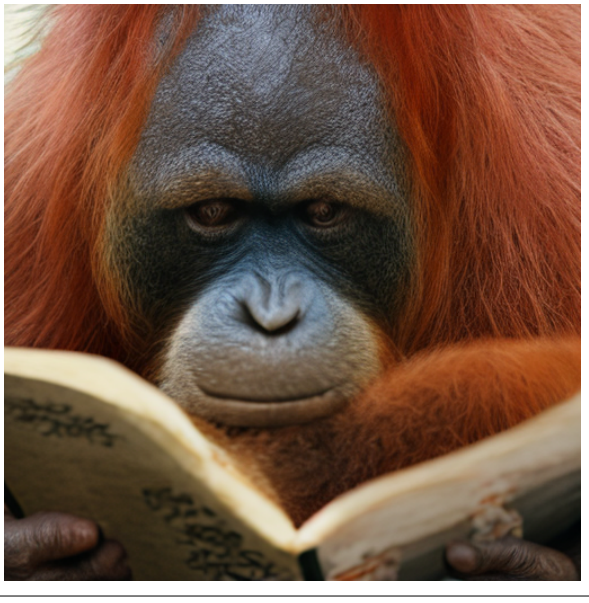

Figure 18-21. A picture generated by Stable Diffusion using the Diffusers library

The possibilities are endless!

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

Exercises

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

13. Train a diffusion model on your preferred image dataset (e.g., `torchvision.datasets.Flowers102`), and generate nice images. Next, add the image class as an extra input to the model, and retrain it: you should now be able to control the class of the generated image.

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

CHAPTER 19
Reinforcement Learning

Reinforcement learning (RL) is one of the most exciting fields of machine learning today, and also one of the oldest. It has been around since the 1950s, producing many interesting applications over the years,[1] particularly in games (e.g., *TD-Gammon*, a backgammon-playing program) and in machine control, but seldom making the headline news. However, a revolution took place in 2013, when researchers from a British startup called DeepMind[2] demonstrated a system that could learn to play just about any Atari game from scratch,[3] eventually outperforming humans[4] in most of them, using only raw pixels as inputs and without any prior knowledge of the rules of the games.[5] This was the first of a series of amazing feats:

- In 2016, DeepMind's AlphaGo beat Lee Sedol, a legendary professional player of the game of Go; and in 2017, it beat Ke Jie, the world champion. No program had ever come close to beating a master of this game, let alone the very best.
- In 2020, DeepMind released AlphaFold, which can predict the 3D shape of proteins with unprecedented accuracy. This is a game changer in biology, chemistry, and medicine. In fact, Demis Hassabis (founder and CEO) and John Jumper (director) were awarded the Nobel Prize in Chemistry for AlphaFold.

1 For more details, be sure to check out Richard Sutton and Andrew Barto's book on RL, *Reinforcement Learning: An Introduction* (MIT Press).

2 DeepMind was bought by Google for over $500 million in 2014.

3 Volodymyr Mnih et al., "Playing Atari with Deep Reinforcement Learning", arXiv preprint arXiv:1312.5602 (2013), *https://homl.info/dqn*.

4 Volodymyr Mnih et al., "Human-Level Control Through Deep Reinforcement Learning", *Nature* 518 (2015): 529–533, *https://homl.info/dqn2*.

5 Check out the videos of DeepMind's system learning to play *Space Invaders*, *Breakout*, and other video games at *https://homl.info/dqn3*.

- In 2022, DeepMind released AlphaCode, which can generate code at a competitive programming level.
- In 2023, DeepMind released GNoME which can predict new crystal structures, including hundreds of thousands of predicted stable materials.

So how did DeepMind researchers achieve all of this? Well, they applied the power of deep learning to the field of reinforcement learning, and it worked beyond their wildest dreams: *deep reinforcement learning* was born. Today, although DeepMind continues to lead the way, many other organizations have joined in, and the whole field is boiling with new ideas, with a wide range of applications.

In this chapter I will first explain what reinforcement learning is and what it's good at, then present three of the most important families of techniques in deep reinforcement learning: policy gradients, deep Q-networks (including a discussion of Markov decision processes), and lastly, actor-critic methods, including the popular PPO, which we will use to beat an Atari game. So let's get started!

What Is Reinforcement Learning?

In reinforcement learning, a software *agent* makes *observations* and takes *actions* within an *environment*, and in return it receives *rewards* from the environment. Its objective is to learn to act in a way that will maximize its expected rewards over time. If you don't mind a bit of anthropomorphism, you can think of positive rewards as pleasure, and negative rewards as pain (the term "reward" is a bit misleading in this case). In short, the agent acts in the environment and learns by trial and error to maximize its pleasure and minimize its pain.

This is quite a broad setting that can apply to a wide variety of tasks. Here are a few examples (see Figure 19-1):

- The agent can be the program controlling a robot. In this case, the environment is the real world, the agent observes the environment through a set of *sensors*, such as cameras and touch sensors, and its actions consist of sending signals to activate motors. It may be programmed to get positive rewards whenever it approaches the target destination, and negative rewards whenever it wastes time or goes in the wrong direction.
- The agent can be the program controlling *Ms. Pac-Man*. In this case, the environment is a simulation of the Atari game, the actions are the nine possible joystick positions (upper left, down, center, and so on), the observations are screenshots, and the rewards are just the game points.
- Similarly, the agent can be the program playing a board game such as Go. It only gets a reward if it wins.

- The agent does not have to control a physically (or virtually) moving thing. For example, it can be a smart thermostat, getting positive rewards whenever it is close to the target temperature and saves energy, and negative rewards when humans need to tweak the temperature, so the agent must learn to anticipate human needs.
- The agent can observe stock market prices and decide how much to buy or sell every second. Rewards are obviously the monetary gains and losses.

Note that there may not be any positive rewards at all; for example, the agent may move around in a maze, getting a negative reward at every time step, so it had better find the exit as quickly as possible! There are many other examples of tasks to which reinforcement learning is well suited, such as self-driving cars, recommender systems, placing ads on a web page, or controlling where an image classification system should focus its attention.

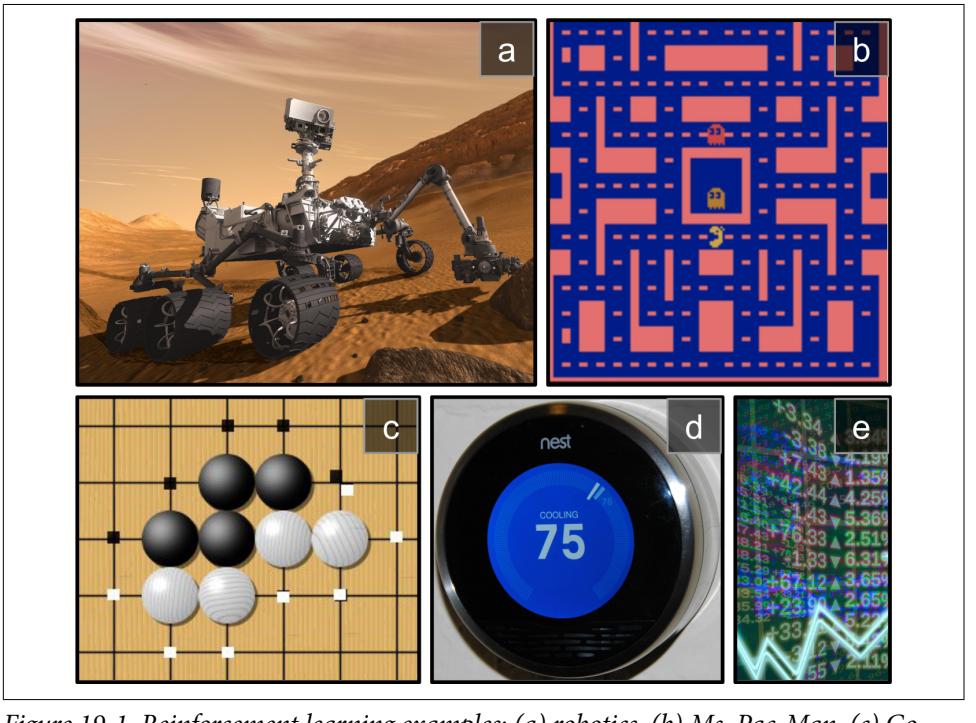

Figure 19-1. Reinforcement learning examples: (a) robotics, (b) Ms. Pac-Man, (c) Go player, (d) thermostat, (e) automatic trader[6]

6 Images (a), (d), and (e) are in the public domain. Image (b) is a screenshot from the *Ms. Pac-Man* game, copyright Atari (fair use in this chapter). Image (c) is reproduced from Wikipedia; it was created by user Stevertigo and released under Creative Commons BY-SA 2.0.

Let's now turn to one large family of RL algorithms: *policy gradients*.

Policy Gradients

The algorithm a software agent uses to determine its actions is called its *policy*. The policy can be any algorithm you can think of, such as a neural network taking observations as inputs and outputting the action to take (see Figure 19-2).

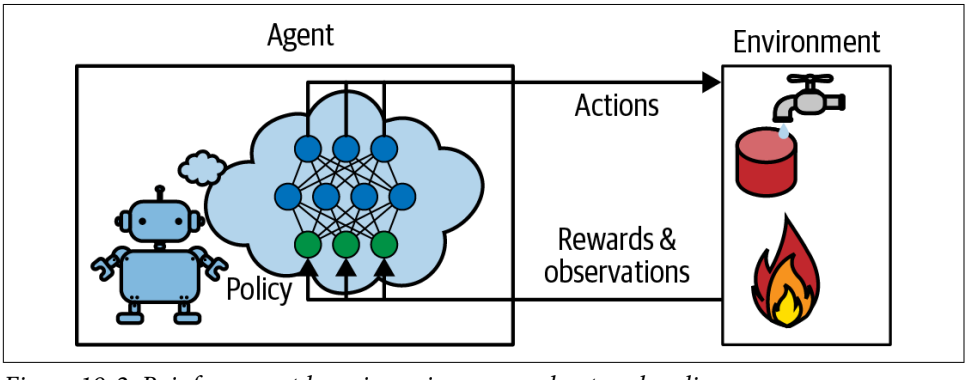

Figure 19-2. Reinforcement learning using a neural network policy

The policy does not even have to be deterministic. In fact, in some cases it does not even have to observe the environment, as long as it can get rewards! For example, consider a blind robotic vacuum cleaner whose reward is the amount of dust it picks up in 30 minutes. Its policy could be to move forward with some probability p every second, or randomly rotate left or right with probability $1 – p$. The rotation angle would be a random angle between $–r$ and $+r$. Since this policy involves some randomness, it is called a *stochastic policy*. The robot will have an erratic trajectory, which guarantees that it will eventually get to any place it can reach and pick up all the dust. The question is, how much dust will it pick up in 30 minutes?

How would you train such a robot? There are just two *policy parameters* you can tweak: the probability p and the angle range r. One possible learning algorithm could be to try out many different values for these parameters, and pick the combination that performs best (see Figure 19-3). This is an example of *policy search*, in this case using a brute-force approach. When the *policy space* is too large (which is generally the case), finding a good set of parameters this way is like searching for a needle in a gigantic haystack.

Another way to explore the policy space is to use *genetic algorithms*. For example, you could randomly create a first generation of 100 policies and try them out, then "kill" the 80 worst policies[7] and make the 20 survivors produce 4 offspring each. An offspring is a copy of its parent[8] plus some random variation. The surviving policies plus their offspring together constitute the second generation. You can continue to iterate through generations this way until you find a good policy.[9]

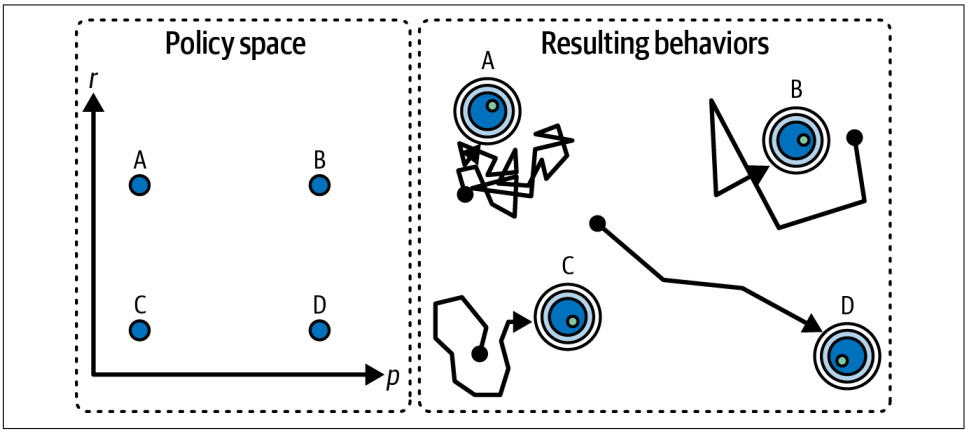

Figure 19-3. Four points in the policy space (left) and the agent's corresponding behavior (right)

Yet another approach is to use optimization techniques by evaluating the gradients of the rewards with regard to the policy parameters, then tweaking these parameters by following the gradients toward higher rewards.[10] Algorithms that follow this strategy are known as *policy gradient* (PG) algorithms. But before we can implement them, we first need to create an environment for the agent to live in—so it's time to introduce the Gymnasium library.

7 It is often better to give the poor performers a slight chance of survival, to preserve some diversity in the "gene pool".

8 If there is a single parent, this is called *asexual reproduction*. With two (or more) parents, it is called *sexual reproduction*. An offspring's genome (in this case a set of policy parameters) is randomly composed of parts of its parents' genomes.

9 One interesting example of a genetic algorithm used for reinforcement learning is the *NeuroEvolution of Augmenting Topologies* (NEAT) algorithm. Also check out *evolutionary policy optimization* (EPO), proposed in 2025, where a master agent learns stably and efficiently from the experiences of a population of agents.

10 This is called *gradient ascent*. It's just like gradient descent, but in the opposite direction: maximizing instead of minimizing.

Introduction to the Gymnasium Library

One of the challenges of reinforcement learning is that in order to train an agent, you first need to have a working environment. If you want to program an agent that will learn to play an Atari game, you will need an Atari game simulator. If you want to program a walking robot, then the environment is the real world, and you can directly train your robot in that environment. However, this has its limits: if the robot falls off a cliff, you can't just click Undo. You can't speed up time either—adding more computing power won't make the robot move any faster—and it's generally too expensive to train 1,000 robots in parallel. In short, training is hard and slow in the real world, so you generally need a *simulated environment* at least for bootstrap training. For example, you might use a library like PyBullet or MuJoCo for 3D physics simulation.

The Gymnasium library is an open source toolkit that provides a wide variety of simulated environments (Atari games, board games, 2D and 3D physics simulations, and so on), that you can use to train agents, compare them, or develop new RL algorithms. It's the successor of OpenAI Gym, and is now maintained by a community of researchers and developers.

Gymnasium is preinstalled on Colab, along with the Arcade Learning Environment (ALE) library `ale_py`, which is an emulator for Atari 2600 games and is required for all the Atari environments, as well as the Box2D library, required for several environments with 2D physics. If you are coding on your own machine instead of Colab, and you followed the installation instructions at *https://homl.info/install-p*, then you should be good to go.

Let's start by importing Gymnasium and making an environment:

```
import gymnasium as gym

env = gym.make("CartPole-v1", render_mode="rgb_array", max_episode_steps=1000)
```

Here, we've created a CartPole environment (version 1). This is a 2D simulation in which a cart can be accelerated left or right in order to balance a pole placed on top of it (see Figure 19-4)—a classic control task. I'll explain `render_mode` and `max_episode_steps` shortly.

The `gym.envs.registry` dictionary contains the names and specifications of all the available environments. You can print a nice list with `gym.pprint_registry()`. The Atari environments will only be available once we start the ALE emulator.

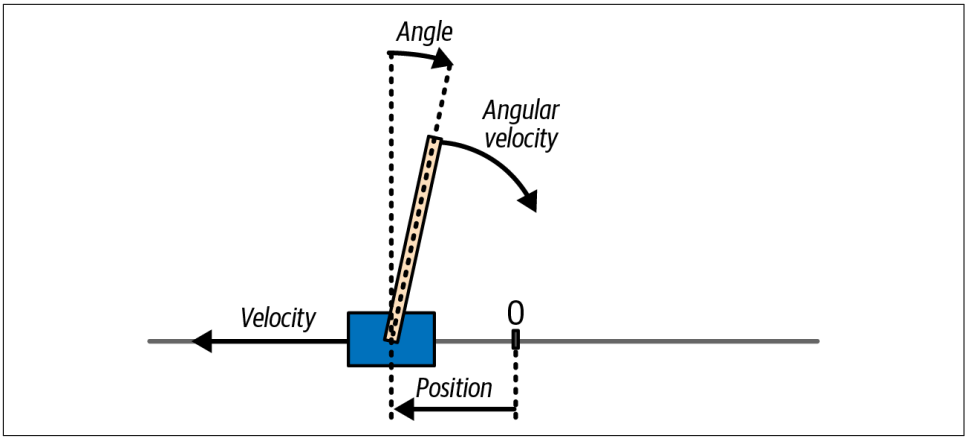

Figure 19-4. The CartPole environment

After the environment is created, you must initialize it using the `reset()` method, optionally specifying a random seed. This returns the first observation. Observations depend on the type of environment. For the CartPole environment, each observation is a NumPy array containing four floats representing the cart's horizontal position (`0.0` = center), its velocity (positive means right), the angle of the pole (`0.0` = vertical), and its angular velocity (positive means clockwise). The `reset()` method also returns a dictionary that may contain extra environment-specific information. This can be useful for debugging and sometimes for training. For example, in many Atari environments, it contains the number of lives left. However, in the CartPole environment, this dictionary is empty:

```
>>> obs, info = env.reset(seed=42)
>>> obs
array([ 0.0273956 , -0.00611216,  0.03585979,  0.0197368 ], dtype=float32)
>>> info
{}
```

Let's call the `render()` method to render this environment as an image. Since we set `render_mode="rgb_array"` when creating the environment, the image will be returned as a NumPy array (you can then use Matplotlib's `imshow()` function to display this image):

```
>>> img = env.render()
>>> img.shape  # height, width, channels (3 = Red, Green, Blue)
(400, 600, 3)
```

Now let's ask the environment what actions are possible:

```
>>> env.action_space
Discrete(2)
```

`Discrete(2)` means that the possible actions are integers 0 and 1, which represent accelerating left or right. Other environments may have additional discrete actions, or other kinds of actions (e.g., continuous). Since the pole is leaning toward the right (`obs[2] > 0`), let's accelerate the cart toward the right:

```
>>> action = 1  # accelerate right
>>> obs, reward, done, truncated, info = env.step(action)
>>> obs
array([ 0.02727336,  0.18847767,  0.03625453, -0.26141977], dtype=float32)
>>> reward, done, truncated, info
(1.0, False, False, {})
```

The `step()` method executes the desired action and returns five values:

`obs`
: This is the new observation. The cart is now moving toward the right (`obs[1] > 0`). The pole is still tilted toward the right (`obs[2] > 0`), but its angular velocity is now negative (`obs[3] < 0`), so it will likely be tilted toward the left after the next step.

`reward`
: In this environment, you get a reward of 1.0 at every step, no matter what you do, so the goal is to keep the episode running for as long as possible. An *episode* is one run of the environment until the game is over or interrupted.

`done`
: This value will be `True` when the episode is over. This will happen when the pole tilts too much, or goes off the screen. After that, the environment must be reset before it can be used again.

`truncated`
: This value will be `True` when an episode is interrupted early, typically by an environment wrapper that imposes a maximum number of steps per episode (see Gymnasium's documentation for more details on environment wrappers). By default, the environment specification for CartPole sets the maximum number of steps to 500, but we changed this to 1,000 when we created the environment. Some RL algorithms treat truncated episodes differently from episodes finished normally (i.e., when `done` is `True`), but in this chapter we will treat them identically.

`info`
: This environment-specific dictionary may provide extra information, just like the one returned by the `reset()` method.

Once you have finished using an environment—possibly after many episodes—you should call its `close()` method to free resources.

Let's hardcode a simple policy that accelerates left when the pole is leaning toward the left and accelerates right when the pole is leaning toward the right. We will run this policy to see the average rewards it gets over 500 episodes:

```
def basic_policy(obs):
    angle = obs[2]
    return 0 if angle < 0 else 1  # go left if leaning left, otherwise go right

totals = []
for episode in range(500):
    total_rewards = 0
    obs, info = env.reset(seed=episode)
    while True:  # no risk of infinite loop: will be truncated after 1000 steps
        action = basic_policy(obs)
        obs, reward, done, truncated, info = env.step(action)
        total_rewards += reward
        if done or truncated:
            break

    totals.append(total_rewards)
```

This code is self-explanatory. Let's look at the result:

```
>>> import numpy as np
>>> np.mean(totals), np.std(totals), min(totals), max(totals)
(np.float64(41.698), np.float64(8.389445512070509), 24.0, 63.0)
```

Even with 500 tries, this policy never managed to keep the pole upright for more than 63 consecutive steps. Not great. If you look at the simulation in this chapter's notebook, you will see that the cart oscillates left and right more and more strongly until the pole tilts too much. A neural network can do better!

Neural Network Policies

Let's create a neural network policy. This neural network will take an observation as input, and it will output the action to be executed, just like the policy we hardcoded earlier. More precisely, it will estimate a probability for each action, then it will select an action randomly, according to the estimated probabilities (see Figure 19-5). In the case of the CartPole environment, there are just two possible actions (left or right), so we only need one output neuron. It will output the probability p of action 1 (right), and of course the probability of action 0 (left) will be $1 - p$. For example, if it outputs 0.7, then we will pick action 1 with 70% probability, or action 0 with 30% probability (this is a *Bernoulli distribution* with $p = 0.7$).

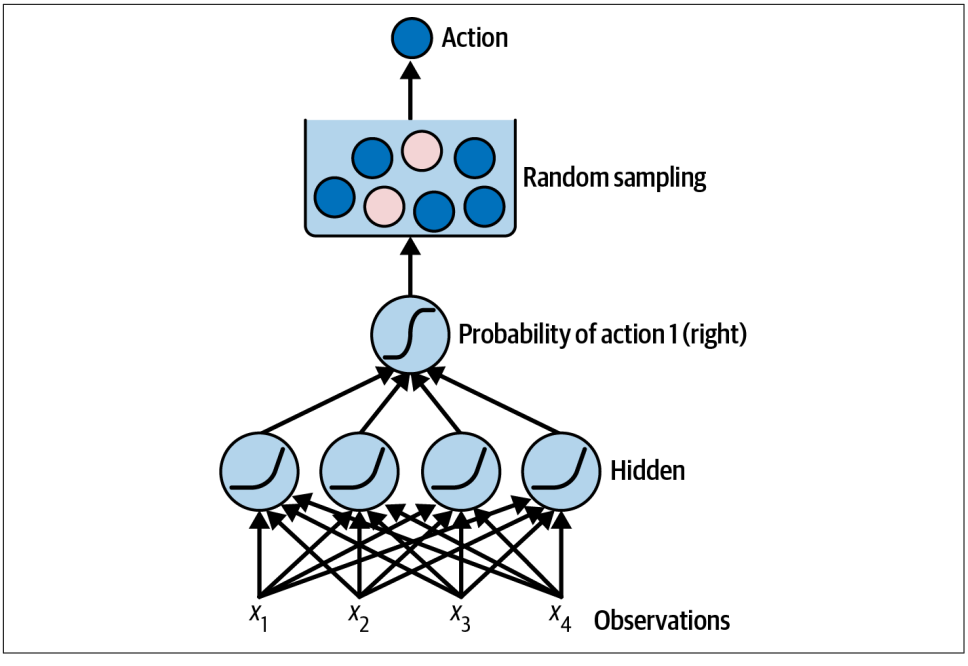

Figure 19-5. Neural network policy

You may wonder why we are picking a random action based on the probabilities given by the neural network, rather than just picking the action with the highest score. This approach lets the agent find the right balance between *exploring* new actions and *exploiting* the actions that are known to work well. Here's an analogy: suppose you go to a restaurant for the first time, and all the dishes look equally appealing, so you randomly pick one. If it turns out to be good, you can increase the probability that you'll order it next time, but you shouldn't increase that probability up to 100%, or you will never try the other dishes, some of which may be even better than the one you tried. This *exploration/exploitation dilemma* is central in reinforcement learning.

Also note that in this particular environment, the past actions and observations can safely be ignored, since each observation contains the environment's full state. If there were some hidden state, then you might need to consider past actions and observations as well. For example, if the environment only revealed the position of the cart but not its velocity, you would have to consider not only the current observation but also the previous observation in order to estimate the current velocity. Another example is when the observations are noisy; in that case, you generally want to use the past few observations to estimate the most likely current state. The CartPole problem is thus as simple as can be; the observations are noise-free, and they contain the environment's full state.

Let's use PyTorch to implement a basic neural network policy for CartPole:

```
import torch
import torch.nn as nn

class PolicyNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(nn.Linear(4, 5), nn.ReLU(), nn.Linear(5, 1))

    def forward(self, state):
        return self.net(state)
```

Our policy network is a tiny MLP, since it's a fairly simple task. The number of inputs is the size of the environment's state: in the case of CartPole, it is just the size of a single observation, which is four. We have just one hidden layer with five units (no need for more in this case). Finally, we want to output a single probability, so we have a single output neuron. If there were more than two possible actions, there would be one output neuron per action instead. For performance and numerical stability, we don't add a sigmoid function at the end, so the network will actually output logits rather than probabilities.

Next let's define a function that will use this policy network to choose an action:

```
def choose_action(model, obs):
    state = torch.as_tensor(obs)
    logit = model(state)
    dist = torch.distributions.Bernoulli(logits=logit)
    action = dist.sample()
    log_prob = dist.log_prob(action)
    return int(action.item()), log_prob
```

The function takes a single observation, converts it to a tensor, and passes it to the policy network to get the logit for action 1 (right). It then creates a `Bernoulli` probability distribution with this logit, and it samples an action from it: this distribution will output 1 (right) with probability p = exp(logit) / (1 + exp(logit)), and 0 (left) with probability 1 – p. If there were more than two possible actions, you would use a `Categorical` distribution instead. Lastly, we compute the log probability of the sampled action (i.e., either log(p) or log(1 – p)): this log probability will be needed later for training.

If the action space is continuous, you can use a Gaussian distribution instead of a Bernoulli or categorical distribution. Instead of predicting logits, the policy network must predict the mean and standard deviation (or the log of the standard deviation) of the distribution. The log of the standard deviation is often clipped to ensure the distribution is neither too wide nor too narrow.

OK, we now have a neural network policy that can take an environment state (in this case, a single observation) and choose an action. But how do we train it?

Evaluating Actions: The Credit Assignment Problem

If we knew what the best action was at each step, we could train the neural network as usual by minimizing the cross-entropy between the estimated probability distribution and the target probability distribution. It would just be regular supervised learning. However, in reinforcement learning the only guidance the agent gets is through rewards, and rewards are typically sparse and delayed. For example, if the agent manages to balance the pole for a total of 100 steps, how can it know which of the 100 actions it took were good, and which of them were bad? All it knows is that the pole fell after the last action, but surely this last action is not entirely responsible. This is called the *credit assignment problem*: when the agent gets a reward (or a penalty), it is hard for it to know which actions should get credited (or blamed) for it. Think of a dog that gets rewarded hours after it behaved well; will it understand what it is being rewarded for?

To simplify credit assignment, a common strategy is to evaluate an action based on the sum of all the rewards that come after it, applying a *discount factor*, γ (gamma), at each step. This sum of discounted rewards is called the action's *return*. Consider the example in Figure 19-6. If an agent decides to go right three times in a row and gets +10 reward after the first step, 0 after the second step, and finally –50 after the third step, then assuming we use a discount factor $\gamma = 0.8$, the first action will have a return of $10 + \gamma \times 0 + \gamma^2 \times (-50) = -22$.

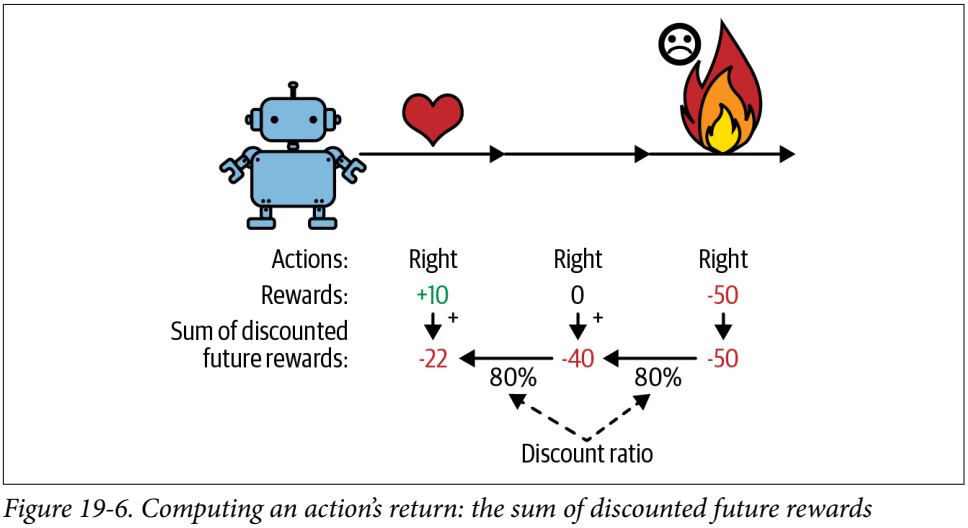

Figure 19-6. Computing an action's return: the sum of discounted future rewards

The following function computes the returns, given the rewards and the discount factor:

```
def compute_returns(rewards, discount_factor):
    returns = rewards[:]  # copy the rewards
    for step in range(len(returns) - 1, 0, -1):
        returns[step - 1] += returns[step] * discount_factor

    return torch.tensor(returns)
```

This function produces the expected result:

```
>>> compute_returns([10, 0, -50], discount_factor=0.8)
tensor([-22., -40., -50.])
```

If the discount factor is close to 0, then future rewards won't count for much compared to immediate rewards. Conversely, if the discount factor is close to 1, then rewards far into the future will count almost as much as immediate rewards. Typical discount factors vary from 0.9 to 0.99. With a discount factor of 0.95, rewards 13 steps into the future count roughly for half as much as immediate rewards (since $0.95^{13} \approx 0.5$), while with a discount factor of 0.99, rewards 69 steps into the future count for half as much as immediate rewards. In the CartPole environment, actions have fairly short-term effects, so choosing a low discount factor of 0.95 seems reasonable, and it will help with credit assignment, making training faster and more stable. However, if the discount factor is set too low, then the agent will learn a suboptimal strategy, focusing too much on short-term gains.

Now that we have a way to evaluate each action, we are ready to train our first agent using policy gradients. Let's see how.

Solving the CartPole Using Policy Gradients

As discussed earlier, policy gradient algorithms optimize the parameters of a policy by following the gradients toward higher rewards. One popular PG algorithm, called *REINFORCE* (or *Monte Carlo PG*), was introduced back in 1992 by Ronald Williams.[11] It has many variants, with various tweaks, but the general principle is this:

1. First, let the neural network policy play the game for an episode, and record the rewards and estimated log probabilities.
2. Then compute each action's return, using the function defined in the previous section.
3. If an action's return is positive, it means that the action was probably good, and you want to make this action even more likely to be chosen in the future. Con-

11 Ronald J. Williams, "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Leaning", *Machine Learning* 8 (1992): 229–256.

versely, if an action's return is negative, you want to make this action *less* likely. To achieve this, you can minimize the REINFORCE loss defined in Equation 19-1: this will maximize the expected discounted rewards.

Equation 19-1. REINFORCE loss

$$\mathscr{L}(\boldsymbol{\theta}) = -\sum_{t} \log \pi_{\boldsymbol{\theta}}(a_t | s_t) \cdot r_t$$

In this equation, $\pi_{\boldsymbol{\theta}}(a_t|s_t)$ is the policy network's estimated probability for action a_t, given state s_t (where t is the time step), and r_t is the observed return of this action; $\boldsymbol{\theta}$ represents the model parameters.

Let's use PyTorch to implement this algorithm. First, we need a function to let the policy network play an episode, and record the rewards and log probabilities:

```
def run_episode(model, env, seed=None):
    log_probs, rewards = [], []
    obs, info = env.reset(seed=seed)
    while True:  # the environment will truncate the episode if it is too long
        action, log_prob = choose_action(model, obs)
        obs, reward, done, truncated, _info = env.step(action)
        log_probs.append(log_prob)
        rewards.append(reward)
        if done or truncated:
            return log_probs, rewards
```

The function first resets the environment to start a new episode. For reproducibility, we pass a seed to the `reset()` method. Then comes the game loop: at each iteration, we pass the current environment state (i.e., the last observation) to the `choose_action()` method we defined earlier. It returns the chosen action and its log probability. We then call the environment's `step()` method to execute the action. This returns a new observation (a NumPy array), a reward, two booleans indicating whether the game is over or truncated, and an info dict (which we can safely ignore in the case of CartPole). We record the log probabilities and rewards in two lists, which we return when the episode is over.

We can finally write the training function:

```
def train_reinforce(model, optimizer, env, n_episodes, discount_factor):
    for episode in range(n_episodes):
        seed = torch.randint(0, 2**32, size=()).item()
        log_probs, rewards = run_episode(model, env, seed=seed)
        returns = compute_returns(rewards, discount_factor)
        std_returns = (returns - returns.mean()) / (returns.std() + 1e-7)
        losses = [-logp * rt for logp, rt in zip(log_probs, std_returns)]
        loss = torch.cat(losses).sum()
        optimizer.zero_grad()
        loss.backward()
```

```
        optimizer.step()
        print(f"\rEpisode {episode + 1}, Reward: {sum(rewards):.2f}", end=" ")
```

That's nice and short, isn't it? At each training iteration, the function runs an episode and gets the log probabilities and rewards.[12] Then it computes the return for each action. Next, it standardizes the returns (i.e., it subtracts the mean return and divides by the standard deviation, plus a small value to avoid division by zero). This standardization step is optional but it's a common and recommended tweak to the REINFORCE algorithm, as it stabilizes training. Next, the function computes the REINFORCE loss using Equation 19-1, and it performs an optimizer step to minimize the loss.

That's it, we're ready to build and train a policy network!

```
torch.manual_seed(42)
model = PolicyNetwork()
optimizer = torch.optim.NAdam(model.parameters(), lr=0.06)
train_reinforce(model, optimizer, env, n_episodes=200, discount_factor=0.95)
```

Training will take less than a minute. If you run an episode using this policy network, you will see that it perfectly balances the pole. Success!

The simple policy gradients algorithm we just trained solved the CartPole task, but it would not scale well to larger and more complex tasks. Indeed, it is highly *sample inefficient*, meaning it needs to explore the game for a very long time before it can make significant progress. This is because its return estimates are extremely noisy, especially when good actions are mixed with bad ones. However, it is the foundation of more powerful algorithms, such as *actor-critic* algorithms (which we will discuss at the end of this chapter).

Researchers try to find algorithms that work well even when the agent initially knows nothing about the environment. However, unless you are writing a paper, you should not hesitate to inject prior knowledge into the agent, as it will speed up training dramatically. For example, since you know that the pole should be as vertical as possible, you could add negative rewards proportional to the pole's angle. This will make the rewards much less sparse and speed up training. Also, if you already have a reasonably good policy (e.g., hardcoded), you may want to train the neural network to imitate it before using policy gradients to improve it.

12 We generate a new random seed for each episode using `torch.randint()`. This ensures that each episode is different, yet the whole training process is reproducible if we set PyTorch's random seed before calling `train_reinforce()`.

Moreover, the REINFORCE algorithm is quite unstable: the agent may improve for a while during training, then forget everything catastrophically, learn again, forget, learn, etc. It's a roller coaster. This is in large part because the training samples are not independent and identically distributed (IID); indeed, the training samples consist of whatever states the agent is capable of reaching right now. As the agent progresses, it explores different parts of the environment, and it can forget everything about other parts. For example, once it learns to properly hold the pole upright, it will no longer see nonvertical poles, and it will totally forget how to handle them. And this issue gets much worse with more complex environments.

Reinforcement learning is notoriously difficult, largely because of the training instabilities and the huge sensitivity to the choice of hyperparameter values and random seeds.[13] As the researcher Andrej Karpathy put it, "[Supervised learning] wants to work. […] RL must be forced to work". You will need time, patience, perseverance, and perhaps a bit of luck, too. This is a major reason RL is not as widely adopted as regular deep learning.

We will now look at another popular family of algorithms: *value-based methods*.

Value-Based Methods

Whereas PG algorithms directly try to optimize the policy to increase rewards, value-based methods are less direct: the agent learns to estimate the value of each state (i.e., the expected return), or the value of each action in a given state, then it uses this knowledge to decide how to act. To understand these algorithms, we must first discuss *Markov decision processes* (MDPs).

Markov Decision Processes

In the early 20th century, the mathematician Andrey Markov studied stochastic processes with no memory, called *Markov chains*. Such a process has a fixed number of states, and it randomly evolves from one state to another at each step. The probability for it to evolve from a state s to a state s' is fixed, and it depends only on the pair (s, s'), not on past states. This is why we say that the system has no memory.

13 A great 2018 post by Alex Irpan nicely lays out RL's biggest difficulties and limitations.

Figure 19-7 shows an example of a Markov chain with four states.

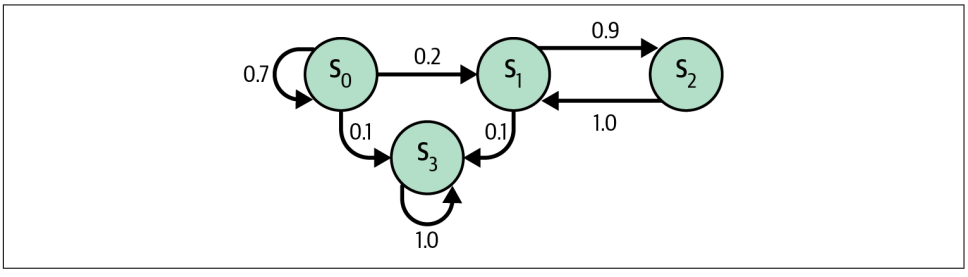

Figure 19-7. Example of a Markov chain

Suppose that the process starts in state s_0, and there is a 70% chance that it will remain in that state at the next step. Eventually it is bound to leave that state and never come back, because no other state points back to s_0. If it goes to state s_1, it will then most likely go to state s_2 (90% probability), then immediately back to state s_1 (with 100% probability). It may alternate a number of times between these two states, but eventually it will fall into state s_3 and remain there forever, since there's no way out: this is called a *terminal state*. Markov chains can have very different dynamics, and they are frequently used in thermodynamics, chemistry, statistics, and much more.

Markov decision processes were first described in the 1950s by Richard Bellman.[14] They resemble Markov chains, but with a twist: at each step, an agent can choose one of several possible actions, and the transition probabilities depend on the chosen action. Moreover, some state transitions return some reward (positive or negative), and the agent's goal is to find a policy that will maximize its cumulative reward over time.

For example, the MDP represented in Figure 19-8 has three states (represented by circles) and up to three possible discrete actions at each step (represented by diamonds).

14 Richard Bellman, "A Markovian Decision Process", *Journal of Mathematics and Mechanics* 6, no. 5 (1957): 679–684.

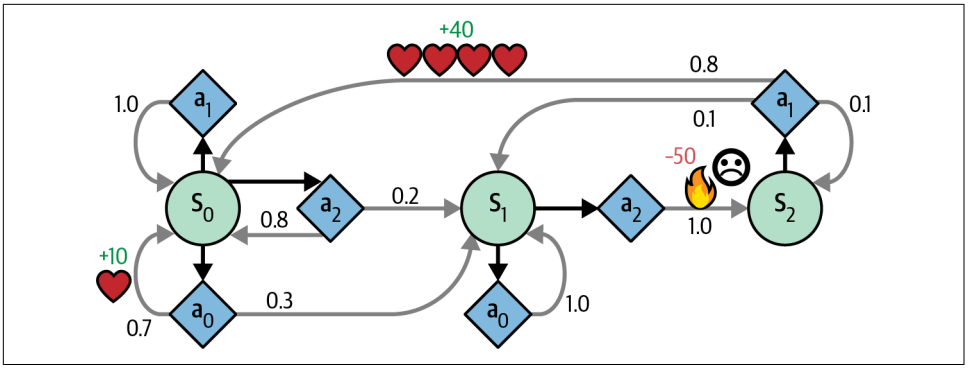

Figure 19-8. Example of a Markov decision process

If it starts in state s_0, the agent can choose among actions a_0, a_1, or a_2. If it chooses action a_1, it just remains in state s_0 with certainty and without any reward. It can thus decide to stay there forever if it wants to. But if it chooses action a_0, it has a 70% probability of gaining a reward of +10 and remaining in state s_0. It can then try again and again to gain as much reward as possible, but at one point it is going to end up instead in state s_1. In state s_1 it has only two possible actions: a_0 or a_2. It can choose to stay put by repeatedly choosing action a_0, or it can choose to move on to state s_2 and get a negative reward of –50 (ouch). In state s_2 it has no choice but to take action a_1, which will most likely lead it back to state s_0, gaining a reward of +40 on the way. You get the picture. By looking at this MDP, can you guess which strategy will gain the most reward over time? In state s_0 it is clear that action a_0 is the best option, and in state s_2 the agent has no choice but to take action a_1, but in state s_1 it is not obvious whether the agent should stay put (a_0) or go through the fire (a_2).

Bellman found a way to estimate the *optimal state value* of any state s, denoted $V^*(s)$, which is the sum of all discounted future rewards the agent can expect on average starting from state s, assuming it acts optimally. He showed that if the agent acts optimally, then the *Bellman optimality equation* applies (see Equation 19-2). This recursive equation says that if the agent acts optimally, then the optimal value of the current state is equal to the reward it will get on average after taking one optimal action, plus the expected optimal value of all possible next states that this action can lead to.

Equation 19-2. Bellman optimality equation

$$V^*(s) = \max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma \cdot V^*(s')] \quad \text{for all } s$$

In this equation:

- $T(s, a, s')$ is the transition probability from state s to state s', given that the agent chose action a. For example, in Figure 19-8, $T(s_2, a_1, s_0) = 0.8$. Note that $\sum_{s'} T(s,a,s') = 1$.
- $R(s, a, s')$ is the reward that the agent gets when it goes from state s to state s', given that the agent chose action a. For example, in Figure 19-8, $R(s_2, a_1, s_0) = +40$.
- γ is the discount factor.

In the Bellman equation and the rest of this chapter, an optimal policy is one that maximizes the expected sum of *discounted* future rewards: this means that it depends on the discount factor γ. However, in real-world tasks we're generally more interested in the expected sum of rewards per episode, without any discount (in fact, that's usually how we evaluate agents). To approach this goal, we usually choose a discount factor close to 1 (but not too close or else training becomes slow and unstable).

This equation leads directly to an algorithm that can precisely estimate the optimal state value of every possible state: first initialize all the state value estimates to zero, and then iteratively update them using the *value iteration* algorithm (see Equation 19-3). A remarkable result is that, given enough time, these estimates are guaranteed to converge to the optimal state values, corresponding to the optimal policy.

Equation 19-3. Value iteration algorithm

$$V_{k+1}(s) \leftarrow \max_{a} \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma \cdot V_k(s')] \quad \text{for all } s$$

In this equation, $V_k(s)$ is the estimated value of state s at the k^{th} iteration of the algorithm.

This algorithm is an example of *dynamic programming*, which breaks down a complex problem into tractable subproblems that can be tackled iteratively.

Knowing the optimal state values can be useful, in particular to evaluate a policy, but it does not give us the optimal policy for the agent. Luckily, Bellman found a very similar algorithm to estimate the optimal *state-action values*, generally called *Q-values* (quality values). The optimal Q-value of the state-action pair (s, a), denoted $Q^*(s, a)$, is the sum of discounted future rewards the agent can expect on average starting from state s if it chooses action a, but before it sees the outcome of this action, assuming it acts optimally after that action.

Let's look at how it works. Once again, you start by initializing all the Q-value estimates to zero, then you update them using the *Q-value iteration* algorithm (see Equation 19-4).

Equation 19-4. Q-value iteration algorithm

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma \cdot \max_{a'} Q_k(s',a')\right] \quad \text{for all } (s,a)$$

Once you have the optimal Q-values, defining the optimal policy, denoted $\pi^*(s)$, is trivial: when the agent is in state s, it should choose the action with the highest Q-value for that state. The fancy math notation for this is $\pi^*(s) = \underset{a}{\operatorname{argmax}}\ Q^*(s,a)$.

Let's apply this algorithm to the MDP represented in Figure 19-8. First, we need to define the MDP:

```
transition_probabilities = [  # shape=[s, a, s']
    [[0.7, 0.3, 0.0], [1.0, 0.0, 0.0], [0.8, 0.2, 0.0]],
    [[0.0, 1.0, 0.0], None, [0.0, 0.0, 1.0]],
    [None, [0.8, 0.1, 0.1], None]
]
rewards = [  # shape=[s, a, s']
    [[+10, 0, 0], [0, 0, 0], [0, 0, 0]],
    [[0, 0, 0], [0, 0, 0], [0, 0, -50]],
    [[0, 0, 0], [+40, 0, 0], [0, 0, 0]]
]
possible_actions = [[0, 1, 2], [0, 2], [1]]
```

For example, to know the transition probability of going from s_2 to s_0 after playing action a_1, we will look up `transition_probabilities[2][1][0]` (which is 0.8). Similarly, to get the corresponding reward, we will look up `rewards[2][1][0]` (which is +40). And to get the list of possible actions in s_2, we will look up `possible_actions[2]` (in this case, only action a_1 is possible). Next, we must initialize all the Q-values to zero (except for the impossible actions, for which we set the Q-values to $-\infty$):

```
Q_values = np.full((3, 3), -np.inf)  # -np.inf for impossible actions
for state, actions in enumerate(possible_actions):
    Q_values[state, actions] = 0.0  # for all possible actions
```

Now let's run the Q-value iteration algorithm. It applies Equation 19-4 repeatedly, to all Q-values, for every state and every possible action:

```
gamma = 0.90  # the discount factor

for iteration in range(50):
    Q_prev = Q_values.copy()
    for s in range(3):
        for a in possible_actions[s]:
            Q_values[s, a] = np.sum([
                    transition_probabilities[s][a][sp]
                    * (rewards[s][a][sp] + gamma * Q_prev[sp].max())
                for sp in range(3)])
```

That's it! The resulting Q-values look like this:

```
>>> Q_values
array([[18.91891892, 17.02702702, 13.62162162],
       [ 0.        ,        -inf, -4.87971488],
       [       -inf, 50.13365013,        -inf]])
```

For example, when the agent is in state s_0 and it chooses action a_1, the expected sum of discounted future rewards is approximately 17.0.

For each state, we can find the action that has the highest Q-value:

```
>>> Q_values.argmax(axis=1)  # optimal action for each state
array([0, 0, 1])
```

This gives us the optimal policy for this MDP when using a discount factor of 0.90: in state s_0 choose action a_0, in state s_1 choose action a_0 (i.e., stay put), and in state s_2 choose action a_1 (the only possible action). Interestingly, if we increase the discount factor to 0.95, the optimal policy changes: in state s_1 the best action becomes a_2 (go through the fire!). This makes sense because the more you value future rewards, the more you are willing to put up with some pain now for the promise of future bliss.

Temporal Difference Learning

Reinforcement learning problems with discrete actions can often be modeled as Markov decision processes, but the agent initially has no idea what the transition probabilities are (it does not know $T(s, a, s')$), and it does not know what the rewards are going to be either (it does not know $R(s, a, s')$). It must experience each state and each transition at least once to know the rewards, and it must experience them multiple times if it is to have a reasonable estimate of the transition probabilities.

The *temporal difference (TD) learning* algorithm is very similar to the Q-value iteration algorithm, but tweaked to take into account the fact that the agent has only partial knowledge of the MDP. In general we assume that the agent initially knows only the possible states and actions, and nothing more. The agent uses an *exploration policy*—for example, a purely random policy—to explore the MDP, and as it

progresses, the TD learning algorithm updates the estimates of the state values based on the transitions and rewards that are actually observed (see Equation 19-5).

Equation 19-5. TD learning algorithm

$$V_{k+1}(s) \leftarrow (1-\alpha)V_k(s) + \alpha(r + \gamma \cdot V_k(s'))$$

or, equivalently:

$$V_{k+1}(s) \leftarrow V_k(s) + \alpha \cdot \delta_k(s,r,s')$$

with $\delta_k(s,r,s') = r + \gamma \cdot V_k(s') - V_k(s)$

In this equation:

- α is the learning rate (e.g., 0.01).
- $r + \gamma \cdot V_k(s')$ is called the *TD target*.
- $\delta_k(s, r, s')$ is called the *TD error*.

A more concise way of writing the first form of this equation is to use the notation $a \underset{\alpha}{\leftarrow} b$, which means $a_{k+1} \leftarrow (1 - \alpha) \cdot a_k + \alpha \cdot b_k$. So the first line of Equation 19-5 can be rewritten like this: $V(s) \underset{\alpha}{\leftarrow} r + \gamma \cdot V(s')$.

TD learning has many similarities with stochastic gradient descent, including the fact that it handles one sample at a time. Moreover, just like SGD, it can only truly converge if you gradually reduce the learning rate; otherwise, it will keep bouncing around the optimum Q-values.

For each state *s*, this algorithm keeps track of a running average of the immediate rewards the agent gets upon leaving that state, plus the rewards it expects to get later, assuming it acts optimally.

Q-Learning

Similarly, the Q-learning algorithm is an adaptation of the Q-value iteration algorithm to the situation where the transition probabilities and the rewards are initially unknown (see Equation 19-6). Q-learning works by watching an agent play (e.g., randomly) and gradually improving its estimates of the Q-values. Once it has accurate Q-value estimates (or close enough), then the optimal policy is to choose the action that has the highest Q-value (i.e., the greedy policy).

Equation 19-6. Q-learning algorithm

$$Q(s,a) \underset{\alpha}{\leftarrow} r + \gamma \cdot \max_{a'} Q(s',a')$$

For each state-action pair (s, a), this algorithm keeps track of a running average of the rewards r the agent gets upon leaving the state s with action a, plus the sum of discounted future rewards it expects to get. To estimate this sum, we take the maximum of the Q-value estimates for the next state s', since we assume that the target policy will act optimally from then on.

Let's implement the Q-learning algorithm. First, we will need to make an agent explore the environment. For this, we need a step function so that the agent can execute one action and get the resulting state and reward:

```
def step(state, action):
    probas = transition_probabilities[state][action]
    next_state = np.random.choice([0, 1, 2], p=probas)
    reward = rewards[state][action][next_state]
    return next_state, reward
```

Now let's implement the agent's exploration policy. Since the state space is pretty small, a simple random policy will be sufficient. If we run the algorithm for long enough, the agent will visit every state many times, and it will also try every possible action many times:

```
def exploration_policy(state):
    return np.random.choice(possible_actions[state])
```

Next, after we initialize the Q-values just like earlier, we are ready to run the Q-learning algorithm with learning rate decay (using power scheduling, introduced in Chapter 11):

```
alpha0 = 0.05  # initial learning rate
decay = 0.005  # learning rate decay
gamma = 0.90  # discount factor
state = 0  # initial state

for iteration in range(10_000):
    action = exploration_policy(state)
    next_state, reward = step(state, action)
    next_value = Q_values[next_state].max()  # greedy policy at the next step
    alpha = alpha0 / (1 + iteration * decay)
    Q_values[state, action] *= 1 - alpha
    Q_values[state, action] += alpha * (reward + gamma * next_value)
    state = next_state
```

This algorithm will converge to the optimal Q-values, but it will take many iterations, and possibly quite a lot of hyperparameter tuning. As you can see in Figure 19-9, the Q-value iteration algorithm (left) converges very quickly, in fewer than 20 iterations, while the Q-learning algorithm (right) takes about 8,000 iterations to converge.

Obviously, not knowing the transition probabilities or the rewards makes finding the optimal policy significantly harder!

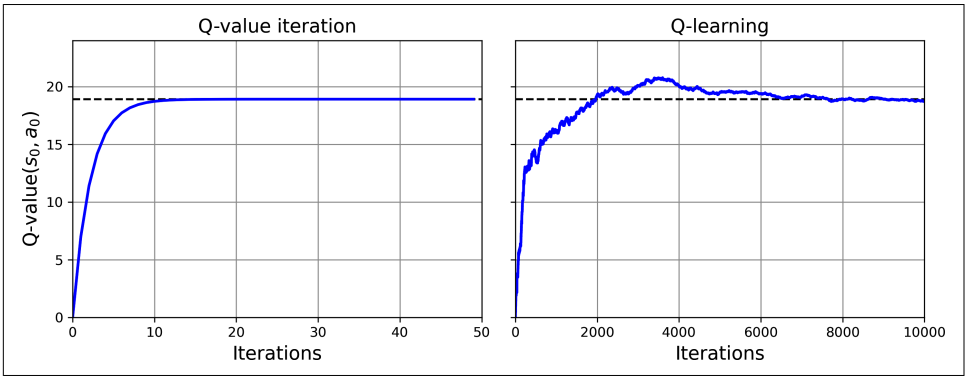

Figure 19-9. Learning curve of the Q-value iteration algorithm versus the Q-learning algorithm

The Q-learning algorithm is called an *off-policy* algorithm because the policy being trained is not necessarily the one used during training. For example, in the code we just ran, the policy being executed (the exploration policy) was completely random, while the policy being trained was never used. After training, the optimal policy corresponds to systematically choosing the action with the highest Q-value. Conversely, the REINFORCE algorithm is *on-policy*: it explores the world using the policy being trained. It is somewhat surprising that Q-learning is capable of learning the optimal policy by just watching an agent act randomly. Imagine learning to play golf when your teacher is a blindfolded monkey. Can we do better?

Exploration Policies

Of course, Q-learning can work only if the exploration policy explores the MDP thoroughly enough. Although a purely random policy is guaranteed to eventually visit every state and every transition many times, it may take an extremely long time to do so. Therefore, a better option is to use the *ε-greedy policy* (ε is epsilon): at each step it acts randomly with probability ε, or greedily with probability $1–\varepsilon$ (i.e., choosing the action with the highest Q-value). The advantage of the ε-greedy policy (compared to a completely random policy) is that it will spend more and more time exploring the interesting parts of the environment, as the Q-value estimates get better and better, while still spending some time visiting unknown regions of the MDP. It is quite common to start with a high value for ε (e.g., 1.0) and then gradually reduce it (e.g., down to 0.05).

Alternatively, rather than relying only on chance for exploration, another approach is to encourage the exploration policy to try actions that it has not tried much before.

This can be implemented as a bonus added to the Q-value estimates, as shown in Equation 19-7.

Equation 19-7. Q-learning using an exploration function

$$Q(s,a) \underset{\alpha}{\leftarrow} r + \gamma \cdot \max_{a'} f(Q(s',a'),N(s',a'))$$

In this equation:

- $N(s', a')$ counts the number of times the action a' was chosen in state s'.
- $f(Q, N)$ is an *exploration function*, such as $f(Q, N) = Q + \kappa/(1 + N)$, where κ (kappa) is a curiosity hyperparameter that measures how much the agent is attracted to the unknown.

Approximate Q-Learning and Deep Q-Learning

The main problem with Q-learning is that it does not scale well to large (or even medium) MDPs with many states and actions. For example, suppose you wanted to use Q-learning to train an agent to play *Ms. Pac-Man* (see Figure 19-1). There are about 240 pellets that Ms. Pac-Man can eat, each of which can be present or absent (i.e., already eaten). So, the number of possible pellet states is about $2^{240} \approx 10^{73}$. And if you add all the possible combinations of positions for all the ghosts and Ms. Pac-Man, the number of possible states becomes larger than the number of atoms in our galaxy, so there's absolutely no way you can keep track of an estimate for every single Q-value.

The solution is to find a function $Q_{\boldsymbol{\theta}}(s, a)$ that approximates the Q-value of any state-action pair (s, a), where the vector $\boldsymbol{\theta}$ parameterizes the function. This is called *approximate Q-learning*. For years it was recommended to use linear combinations of handcrafted features extracted from the state (e.g., the distances of the closest ghosts, their directions, and so on) to estimate Q-values, but in 2013, DeepMind showed that using deep neural networks can work much better, especially for complex problems, and it does not require any feature engineering. A DNN that is used to estimate Q-values is called a *deep Q-network* (DQN), and using a DQN for approximate Q-learning is called *deep Q-learning*.

Now, how can we train a DQN? Well, consider the approximate Q-value computed by the DQN for a given state-action pair (s, a). Thanks to Bellman, we know we want this approximate Q-value to be as close as possible to the reward r that we actually observe after playing action a in state s, plus the discounted value of playing optimally from then on. To estimate this sum of future discounted rewards, we can just execute the DQN on the next state s', for all possible actions a'. We get an approximate future Q-value for each possible action. We then pick the highest (since we assume we will be playing optimally) and discount it, and this gives us an estimate of the sum of

future discounted rewards. By summing the reward r and the future discounted value estimate, we get a target Q-value $y(s, a)$ for the state-action pair (s, a), as shown in Equation 19-8.

Equation 19-8. Target Q-value

$$y(s,a) = r + \gamma \cdot \max_{a'} Q_{\boldsymbol{\theta}}(s',a')$$

With this target Q-value, we can run a training step using any gradient descent algorithm. In general, we try to minimize the squared error between the estimated Q-value $Q_{\boldsymbol{\theta}}(s, a)$ and the target Q-value $y(s, a)$, or the Huber loss to reduce the algorithm's sensitivity to large errors. And that's the deep Q-learning algorithm! Let's see how to implement it to solve the CartPole environment.

Implementing Deep Q-Learning

The first thing we need is a deep Q-network. In theory, we need a neural net that takes a state-action pair as input, and outputs an approximate Q-value. However, in practice it's much more efficient to use a neural net that takes only a state as input, and outputs one approximate Q-value for each possible action. To solve the CartPole environment, we do not need a very complicated neural net; a couple of hidden layers will do:

```
class DQN(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(nn.Linear(4, 32), nn.ReLU(),
                                 nn.Linear(32, 32), nn.ReLU(),
                                 nn.Linear(32, 2))

    def forward(self, state):
        return self.net(state)
```

Our DQN is very similar to our earlier policy network, except it outputs a Q-value for each action instead of logits. Now let's define a function to choose an action based on this DQN:

```
def choose_dqn_action(model, obs, epsilon=0.0):
        if torch.rand(()) < epsilon:  # epsilon greedy policy
            return torch.randint(2, size=()).item()
        else:
            state = torch.as_tensor(obs)
            Q_values = model(state)
            return Q_values.argmax().item()  # optimal according to the DQN
```

This function takes an environment state (a single observation) and passes it to the neural net to predict the Q-values, then it simply returns the action with the largest

predicted Q-value (`argmax()`). To ensure that the agent explores the environment, we use an ε-greedy policy, meaning we choose a random action with probability ε.

DQNs generally don't work with continuous action spaces, unless you can discretize the space (which only works if it's tiny) or combine them with policy gradients. This is because the DQN agent must find the action with the highest Q-value at each step. In a continuous action space, this requires running an optimization algorithm on the Q-value function at each step, which is not practical.

Instead of training the DQN based only on the latest experiences, we will store all experiences in a *replay buffer* (or *replay memory*), and we will sample a random training batch from it at each training iteration. This helps reduce the correlations between the experiences in a training batch, which stabilizes training by making the data distribution more consistent. Each experience will be represented as a tuple with six elements: a state s, the action a that the agent took, the resulting reward r, the next state s' it reached, a boolean indicating whether the episode ended at that point (`done`), and finally another boolean indicating whether the episode was truncated at that point. We will also need a function to sample a random batch of experiences from the replay buffer. It will return a tuple containing six tensors, one for each field:

```
def sample_experiences(replay_buffer, batch_size):
    indices = torch.randint(len(replay_buffer), size=[batch_size])
    batch = [replay_buffer[index] for index in indices.tolist()]
    return [to_tensor([exp[index] for exp in batch]) for index in range(6)]

def to_tensor(data):
    array = np.stack(data)
    dtype = torch.float32 if array.dtype == np.float64 else None
    return torch.as_tensor(array, dtype=dtype)
```

The `sample_experiences()` function takes a replay buffer and a batch size, and it randomly samples the desired number of experience tuples from the buffer. Then, for each of the six fields in the experience tuples, it extracts that field from each experience in the batch, and converts that list to a tensor using the `to_tensor()` function. Lastly, it returns the list of six tensors. The tensors all have shape [`batch size`] except for the observation tensors, which have shape [`batch size`, 4].

The `to_tensor()` function takes a Python list containing observations (i.e., 64-bit NumPy arrays of shape [4]), or actions (integers), or rewards (floats), or booleans (done or truncated), and it returns a tensor of the appropriate PyTorch type. Note that the 64-bit NumPy arrays containing the observations are converted to 32-bit tensors.

The replay buffer can be any data structure that supports appending and indexing, and can limit the size to avoid blowing up memory during training. For simplicity, we will use a Python *deque*, from the standard `collections` package. This is a double-ended queue, in which elements can be efficiently appended or popped (i.e., removed) on both ends. If you set a size limit, and that limit is reached, appending an element to one end of the queue automatically pops an item from the other side. This means that each new experience replaces the oldest experience, which is exactly what we want.

Appending and popping items on the ends of a deque is very fast, but random access can be slow when the queue gets very long (e.g., 100,000 items or more). If you need a very large replay buffer, you should use a circular buffer instead (see the notebook for an implementation), or check out DeepMind's Reverb library.

Let's also create a function that will play a full episode using our DQN, and store the resulting experiences in the replay buffer. We'll run in eval mode with `torch.no_grad()` since we don't need gradients for now. For logging purposes, we'll also make the function sum up all the rewards in the episode and return the result:

```
def play_and_record_episode(model, env, replay_buffer, epsilon, seed=None):
    obs, _info = env.reset(seed=seed)
    total_rewards = 0
    model.eval()
    with torch.no_grad():
        while True:
            action = choose_dqn_action(model, obs, epsilon)
            next_obs, reward, done, truncated, _info = env.step(action)
            experience = (obs, action, reward, next_obs, done, truncated)
            replay_buffer.append(experience)
            total_rewards += reward
            if done or truncated:
                return total_rewards
            obs = next_obs
```

Next, let's create a function that will sample a batch of experiences from the replay buffer and train the DQN by performing a single gradient descent step on this batch:

```
def dqn_training_step(model, optimizer, criterion, replay_buffer, batch_size,
                      discount_factor):
    experiences = sample_experiences(replay_buffer, batch_size)
    state, action, reward, next_state, done, truncated = experiences
    with torch.inference_mode():
        next_Q_value = model(next_state)

    max_next_Q_value, _ = next_Q_value.max(dim=1)
    running = (~(done | truncated)).float()  # 0 if s' is over, 1 if running
    target_Q_value = reward + running * discount_factor * max_next_Q_value
    all_Q_values = model(state)
```

```
    Q_value = all_Q_values.gather(dim=1, index=action.unsqueeze(1))
    loss = criterion(Q_value, target_Q_value.unsqueeze(1))
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

The `torch.inference_mode()` context is like `torch.no_grad()`, plus models run in eval mode within the context, and new tensors cannot be used in backpropagation.

Here's what's happening in this code:

- The function starts by sampling a batch of experiences from the replay buffer.
- Then it uses the DQN to compute the target Q-value for each experience in the batch. For this, the code implements Equation 19-8: the DQN is used in inference mode to evaluate all the Q-values for the next state *s'*, then we keep only the max Q-value since we assume that the agent will play optimally from now on, and we multiply this max Q-value with the discount factor. If the episode was over (done or truncated), then the discounted max Q-value is multiplied by zero since we cannot expect any more rewards. Otherwise, it's multiplied by 1 (i.e., unchanged). Lastly, we add the experience's reward. All of this is performed simultaneously for all experiences in the batch.
- Next, the function uses the model again (in training mode this time) to compute all the Q-values for the current state *s*, and it uses the `gather()` method to extract just the Q-value that corresponds to the action that was actually chosen. Again, this is done simultaneously for all experiences in the batch.
- Lastly, we compute the loss, which is typically the MSE between the target Q-values and the predicted Q-values, and we perform an optimizer step to minimize the loss.

Phew! That was the hardest part. Now we can write the main training function and run it:

```
from collections import deque

def train_dqn(model, env, replay_buffer, optimizer, criterion, n_episodes=800,
              warmup=30, batch_size=32, discount_factor=0.95):
    totals = []
    for episode in range(n_episodes):
        epsilon = max(1 - episode / 500, 0.01)
        seed = torch.randint(0, 2**32, size=()).item()
```

```
            total_rewards = play_and_record_episode(model, env, replay_buffer,
                                                    epsilon, seed=seed)
            print(f"\rEpisode: {episode + 1}, Rewards: {total_rewards}", end=" ")
            totals.append(total_rewards)
            if episode >= warmup:
                dqn_training_step(model, optimizer, criterion, replay_buffer,
                                  batch_size, discount_factor)
    return totals

torch.manual_seed(42)
dqn = DQN()
optimizer = torch.optim.NAdam(dqn.parameters(), lr=0.03)
mse = nn.MSELoss()
replay_buffer = deque(maxlen=100_000)
totals = train_dqn(dqn, env, replay_buffer, optimizer, mse)
```

The training algorithm runs for 800 episodes. At each training iteration, we make the DQN play one full episode using the `play_and_record_episode()` function, then we run one training step using the `dqn_training_step()` function. Note that we only start training after several warmup episodes to ensure that the replay buffer contains plenty of experiences. We also linearly decrease the epsilon value for the ε-greedy policy from 1.0 down to 0.01 after 500 episodes (then it remains at 0.01). This way, the agent's behavior will gradually become less random, focusing more on exploitation and less on exploration. The function also records the total rewards for each episode, and returns these totals; they are plotted in Figure 19-10.

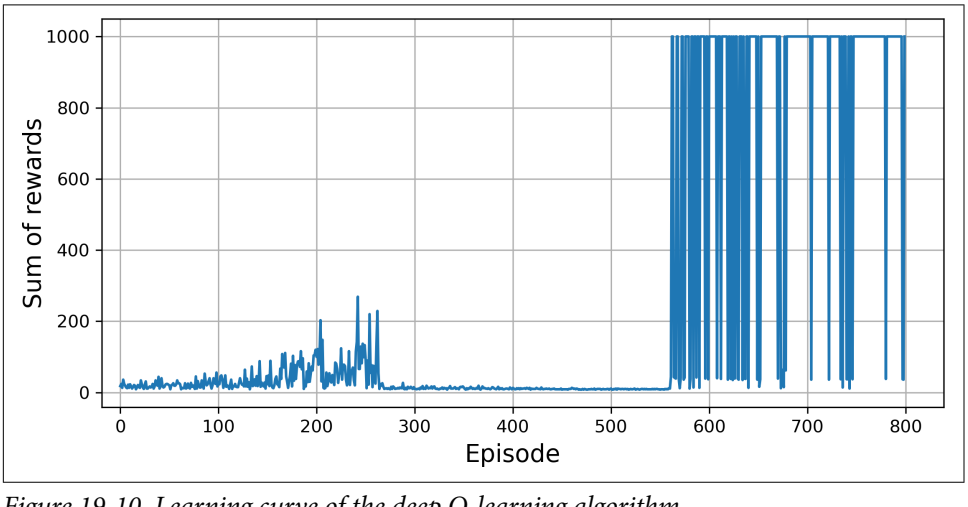

Figure 19-10. Learning curve of the deep Q-learning algorithm

Why not plot the loss? Well, it's a poor indicator of the model's performance, so it's preferable to plot the total rewards for each episode. Indeed, the loss might go down while the agent performs worse (e.g., if the agent gets stuck in one small region of the environment and the DQN starts overfitting it). Conversely, the loss could go up while the agent performs better (e.g., if the DQN was underestimating the target Q-values and it starts correctly increasing them).

The good news is that the algorithm worked: the trained agent perfectly balances the pole on the cart and reaches the maximum total reward of 1,000. The bad news is that the training is completely unstable. In fact, it's even less stable than REINFORCE. I had to tweak the hyperparameters quite a bit before stumbling upon this successful training run. As you can see, the agent managed to reach a reward of 200 points after roughly 200 episodes, which isn't bad, but soon after it forgot everything and performed terribly until episode ~550, when it quickly cracked the problem.

So why is this DQN implementation unstable? Could it be the data distribution? Well, the replay buffer is quite large, so the data distribution is certainly much more stable than with the REINFORCE algorithm. So what's happening? Well, in this basic deep Q-learning implementation, the model is used both to make predictions and to set its own targets. This can lead to a situation analogous to a dog chasing its own tail. This feedback loop can make the network unstable: it can diverge, oscillate, freeze, and so on. Luckily, there are ways to improve this; let's see how.

DQN Improvements

In their 2013 paper, DeepMind researchers proposed a way to stabilize DQN training by using two DQNs instead of one: the first is the *online model*, which learns at each step and is used to move the agent around, and the other is the *target model* used only to define the targets. The target model is just a clone of the online model, and its weights are copied from the online model at regular intervals (e.g., every 10,000 steps in their Atari models). This makes the Q-value targets much more stable, so the feedback loop is damped, and its effects are much less severe. They combined this major improvement with several other tweaks: a very large replay buffer, a tiny learning rate, a very long training time (50 million steps), a very slowly decreasing epsilon (over 1 million steps), and a powerful neural net (a CNN).

Then, in a 2015 paper,[15] DeepMind researchers tweaked their DQN algorithm again, increasing its performance and somewhat stabilizing training. They called this variant *double DQN*. The update was based on the observation that the target network

15 Hado van Hasselt et al., "Deep Reinforcement Learning with Double Q-Learning", *Proceedings of the 30th AAAI Conference on Artificial Intelligence* (2015): 2094–2100.

is prone to overestimating Q-values. Indeed, suppose all actions are equally good: the Q-values estimated by the target model should be identical, but since they are approximations, some may be slightly greater than others by pure chance. The target model will always select the largest Q-value, which will be slightly greater than the mean Q-value, most likely overestimating the true Q-value (a bit like counting the height of the tallest random wave when measuring the depth of a pool). To fix this, the researchers proposed using the online model instead of the target model when selecting the best action for the next state, and using the target model only to estimate the Q-value of this best action.

Another important improvement was the introduction of *prioritized experience replay* (PER), which was proposed in a 2015 paper[16] by DeepMind researchers (once again!). Instead of sampling experiences *uniformly* from the replay buffer, why not sample important experiences more frequently?

More specifically, experiences are considered "important" if they are likely to lead to fast learning progress. But how can we estimate this? One reasonable approach is to measure the magnitude of the TD error $\delta = r + \gamma \cdot V(s') - V(s)$. A large TD error indicates that a transition (s, a, s') is very surprising and thus probably worth learning from.[17] When an experience is recorded in the replay buffer, its priority is set to a very large value to ensure that it gets sampled at least once. However, once it is sampled (and every time it is sampled), the TD error δ is computed, and this experience's priority is set to $p = |\delta|$ (plus a small constant to ensure that every experience has a nonzero probability of being sampled). The probability P of sampling an experience with priority p is proportional to p^ζ, where ζ (zeta) is a hyperparameter that controls how greedy we want importance sampling to be: when $\zeta = 0$, we just get uniform sampling, and when $\zeta = 1$, we get full-blown importance sampling. In the paper, the authors used $\zeta = 0.6$, but the optimal value will depend on the task.

There's one catch though: since the samples will be biased toward important experiences, we must compensate for this bias during training by downweighting the experiences according to their importance, or the model will just overfit the important experiences. To be clear, we want important experiences to be sampled more often, but this also means we must give them a lower weight during training. To do this, we define each experience's training weight as $w = (n\ P)^{-\beta}$, where n is the number of experiences in the replay buffer, and β is a hyperparameter that controls how much we want to compensate for the importance sampling bias (0 means not at all, while 1 means entirely). In the paper, the authors used $\beta = 0.4$ at the beginning of training and linearly increased it to $\beta = 1$ by the end of training. Again, the optimal value

16 Tom Schaul et al., "Prioritized Experience Replay", arXiv preprint arXiv:1511.05952 (2015).

17 It could also just be that the rewards are noisy, in which case there are better methods for estimating an experience's importance (see "Prioritized Experience Replay" for some examples).

will depend on the task, but if you increase one, you will usually want to increase the other as well.

One last noteworthy DQN variant is the *dueling DQN* algorithm (DDQN, not to be confused with double DQN, although both techniques can easily be combined). It was introduced in yet another 2015 paper[18] by DeepMind researchers. To understand how it works, we must first note that the Q-value of a state-action pair (s, a) can be expressed as $Q(s, a) = V(s) + A(s, a)$, where $V(s)$ is the value of state s, and $A(s, a)$ is the *advantage* of taking the action a in state s, compared to all other possible actions in that state. Moreover, the value of a state is equal to the Q-value of the best action a^* for that state (since we assume the optimal policy will pick the best action), so $V(s) = Q(s, a^*)$, which implies that $A(s, a^*) = 0$. In a dueling DQN, the model estimates both the value of the state and the advantage of each possible action. Since the best action should have an advantage of 0, the model subtracts the maximum predicted advantage from all predicted advantages. The rest of the algorithm is just the same as earlier.

These techniques can be combined in various ways, as DeepMind demonstrated in a 2017 paper:[19] the paper's authors combined six different techniques into an agent called *Rainbow*, which largely outperformed the state of the art.

Speaking of combining different methods, why not combine policy gradients with value-based methods to get the best of both worlds? This is the core idea behind actor-critic algorithms. Let's discuss them now.

Actor-Critic Algorithms

Actor-critics are a family of RL algorithms that combine policy gradients with value-based methods. An actor-critic is composed of a policy (the actor) and a value network (the critic), which are trained simultaneously. The actor relies on the critic to estimate the value (or advantage) of actions or states, guiding its policy updates. Since the critic can use a large replay buffer, it stabilizes training and increases data efficiency. It's a bit like an athlete (the actor) learning with the help of a coach (the critic).

Moreover, actor-critic methods support stochastic policies and continuous action spaces, just like policy gradients. So we do get the best of both worlds.

18 Ziyu Wang et al., "Dueling Network Architectures for Deep Reinforcement Learning", arXiv preprint arXiv:1511.06581 (2015).

19 Matteo Hessel et al., "Rainbow: Combining Improvements in Deep Reinforcement Learning", arXiv preprint arXiv:1710.02298 (2017): 3215–3222.

Let's implement a basic actor-critic:

```
class ActorCritic(nn.Module):
    def __init__(self):
        super().__init__()
        self.body = nn.Sequential(nn.Linear(4, 32), nn.ReLU(),
                                  nn.Linear(32, 32), nn.ReLU())
        self.actor_head = nn.Linear(32, 1)  # outputs action logits
        self.critic_head = nn.Linear(32, 1)  # outputs state values

    def forward(self, state):
        features = self.body(state)
        return self.actor_head(features), self.critic_head(features)
```

In the constructor, we build the actor and critic networks. In this implementation, they share the same lower layers (called the *body*). This is common practice, as it reduces the total number of parameters and thereby increases data efficiency, but it also makes training a bit less stable since it couples the actor and critic more closely (another dog chasing its tail situation). The actor network takes a batch of environment states and outputs an action logit for each state (that's the logit for action 1, just like for REINFORCE). The critic network estimates the value of each given state. The `forward()` method takes a batch of states and runs them through both networks (with a shared body), and returns the action logits and state values.

Now let's write a function to choose an action. It's identical to the `choose_action()` function we wrote earlier for the REINFORCE policy network, except that it also returns the state value estimated by the critic network. This will be needed for training:

```
def choose_action_and_evaluate(model, obs):
    state = torch.as_tensor(obs)
    logit, state_value = model(state)
    dist = torch.distributions.Bernoulli(logits=logit)
    action = dist.sample()
    log_prob = dist.log_prob(action)
    return int(action.item()), log_prob, state_value
```

Great! Now let's see how to train our actor-critic. We'll start by defining a function that will perform one training step:

```
def ac_training_step(optimizer, criterion, state_value, target_value, log_prob,
                     critic_weight):
    td_error = target_value - state_value
    actor_loss = -log_prob * td_error.detach()
    critic_loss = criterion(state_value, target_value)
    loss = actor_loss + critic_weight * critic_loss
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

First, we compute the TD error, which is the difference between the target value $y = r + \gamma V(s')$ and the state value $V(s)$. The actor's loss is the same as in REINFORCE, except that we multiply the log probability by the TD error instead of the (standardized) return. In other words, we encourage actions that performed better than the value network expected. As for the critic's loss, it encourages the critic's value estimates $V(s)$ to match the target values y (e.g., using the MSE). Lastly, the overall loss is a weighted sum of the actor's loss and the critic's loss. To stabilize training, it's generally a good idea to give less weight to the critic's loss. Then we perform an optimizer step to minimize the loss. Oh, and note that we call `td_error.detach()` because we don't want gradient descent to affect the critic network via the actor's loss.

We'll also need a function to compute the target value:

```
def get_target_value(model, next_obs, reward, done, truncated, discount_factor):
    with torch.inference_mode():
        _, _, next_state_value = choose_action_and_evaluate(model, next_obs)

    running = 0.0 if (done or truncated) else 1.0
    target_value = reward + running * discount_factor * next_state_value
    return target_value
```

This code first evaluates $V(s')$ using the `choose_action_and_evaluate()` function (we ignore the chosen action and its log probability). We run this in inference mode because we are computing the target: we don't want gradient descent to affect it. Next, we simply evaluate the target $y = r + \gamma V(s')$. If the episode is over, then $y = r$.

With that, we have all we need to write a function that will run a whole episode and train the actor-critic at each step (we also compute the total rewards and return it when the episode is over):

```
def run_episode_and_train(model, optimizer, criterion, env, discount_factor,
                          critic_weight, seed=None):
    obs, _info = env.reset(seed=seed)
    total_rewards = 0
    while True:
        action, log_prob, state_value = choose_action_and_evaluate(model, obs)
        next_obs, reward, done, truncated, _info = env.step(action)
        target_value = get_target_value(model, next_obs, reward, done,
                                        truncated, discount_factor)
        ac_training_step(optimizer, criterion, state_value, target_value,
                         log_prob, critic_weight)
        total_rewards += reward
        if done or truncated:
            return total_rewards
        obs = next_obs
```

And lastly, we can write our main training function, which just calls the `run_episode_and_train()` function many times and returns the total rewards for each episode:

```
def train_actor_critic(model, optimizer, criterion, env, n_episodes=400,
                       discount_factor=0.95, critic_weight=0.3):
    totals = []
    model.train()
    for episode in range(n_episodes):
        seed = torch.randint(0, 2**32, size=()).item()
        total_rewards = run_episode_and_train(model, optimizer, criterion, env,
                                              discount_factor, critic_weight,
                                              seed=seed)
        totals.append(total_rewards)
        print(f"\rEpisode: {episode + 1}, Rewards: {total_rewards}", end=" ")

    return totals
```

Let's run it!

```
torch.manual_seed(42)
ac_model = ActorCritic()
optimizer = torch.optim.NAdam(ac_model.parameters(), lr=1.1e-3)
criterion = nn.MSELoss()
totals = train_actor_critic(ac_model, optimizer, criterion, env)
```

And it works! We get a very stable CartPole that collects the maximum rewards. That said, this implementation is still very sensitive to the choice of hyperparameters and random seeds, and training is still very unstable. Luckily, researchers have come up with various techniques that can stabilize the actor-critic. Here are some of the most popular:

Asynchronous advantage actor-critic (A3C)[20]
: This is an important actor-critic variant introduced by DeepMind researchers in 2016 where multiple agents learn in parallel, exploring different copies of the environment. At regular intervals, but asynchronously (hence the name), each agent pushes some weight updates to a master network, then it pulls the latest weights from that network. Each agent thus contributes to improving the master network and benefits from what the other agents have learned. Moreover, instead of estimating the state values, or even the Q-values, the critic estimates the *advantage* of each action (hence the second A in the name), just like in the Dueling DQN.

Advantage actor-critic (A2C)
: A2C is a variant of the A3C algorithm that removes the asynchronicity. All model updates are synchronous, so gradient updates are performed over larger batches, which allows the model to better utilize the power of the GPU.

20 Volodymyr Mnih et al., "Asynchronous Methods for Deep Reinforcement Learning", *Proceedings of the 33rd International Conference on Machine Learning* (2016): 1928–1937.

Soft actor-critic (SAC)[21]

SAC is an actor-critic variant proposed in 2018 by Tuomas Haarnoja and other UC Berkeley researchers. It learns not only rewards, but also how to maximize the entropy of its actions. In other words, it tries to be as unpredictable as possible while still getting as many rewards as possible. This encourages the agent to explore the environment, which speeds up training and makes it less likely to repeatedly execute the same action when the critic produces imperfect estimates. This algorithm has demonstrated an amazing sample efficiency (contrary to all the previous algorithms, which learn very slowly).

Proximal policy optimization (PPO)[22]

This algorithm by John Schulman and other OpenAI researchers is based on A2C, but it clips the loss function to avoid excessively large weight updates (which often lead to training instabilities). PPO is a simplification of the previous *trust region policy optimization* (TRPO) algorithm,[23] also by OpenAI. OpenAI made the news in April 2019 with its AI called OpenAI Five, based on the PPO algorithm, which defeated the world champions at the multiplayer game *Dota 2*.

The last two algorithms, SAC and PPO, are among the most widely used RL algorithms today, and several libraries provide easy to use and highly optimized implementations. For example, let's use the popular Stable-Baselines3 library to train a PPO agent on the *Breakout* Atari game.

Which RL algorithm should you use? PPO is a great general-purpose RL algorithm—a good bet if you're not sure. SAC is the most sample efficient for continuous action tasks, making it ideal for robotics. DQN remains strong for discrete tasks such as Atari games or board games.

21 Tuomas Haarnoja et al., "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor", *Proceedings of the 35th International Conference on Machine Learning* (2018): 1856–1865.

22 John Schulman et al., "Proximal Policy Optimization Algorithms", arXiv preprint arXiv:1707.06347 (2017).

23 John Schulman et al., "Trust Region Policy Optimization", *Proceedings of the 32nd International Conference on Machine Learning* (2015): 1889–1897.

Mastering Atari Breakout Using the Stable-Baselines3 PPO Implementation

Since Stable-Baselines3 (SB3) is not installed by default on Colab, we must first run `%pip install -q stable_baselines3`, which will take a couple of minutes. However, if you are running the code on your own machine and you followed the installation instructions, then it's already installed.

Next, we must create an ALE interface: it will run the Atari 2600 emulator and allow Gymnasium to interface with it (the Atari games will appear in the list of available environments):

```
import ale_py

ale = ale_py.ALEInterface()
```

Atari games were stored on read-only memory (ROM) cartridges. These ROMs can now be downloaded freely and used for research and educational purposes. On Colab, they are preinstalled, and if you followed the installation instructions to run the code locally, then they are also preinstalled.

Now that we have SB3, the ALE interface, and the ROMs, we are ready to create the *Breakout* environment. But instead of creating it using Gymnasium directly, we will use SB3's `make_atari_env()` function: it creates a wrapper environment containing multiple *Breakout* environments that will run in parallel. Each observation from the wrapper environment will contain one observation for each *Breakout* environment. Similarly, the wrapper environment's `step()` function will take an array containing one action for each *Breakout* environment. Lastly, the wrapper environment will take care of preprocessing the images, converting them from 210 × 160 RGB images to 84 × 84 grayscale images. Very convenient! So let's create an SB3 environment containing four *Breakout* environments, and reset it to get an observation:

```
from stable_baselines3.common.env_util import make_atari_env

envs = make_atari_env("BreakoutNoFrameskip-v4", n_envs=4)
obs = envs.reset()  # a 4 × 84 × 84 × 1 NumPy array (note: no info dict)
```

The `env.get_images()` method returns the original images, before preprocessing (see Figure 19-11).

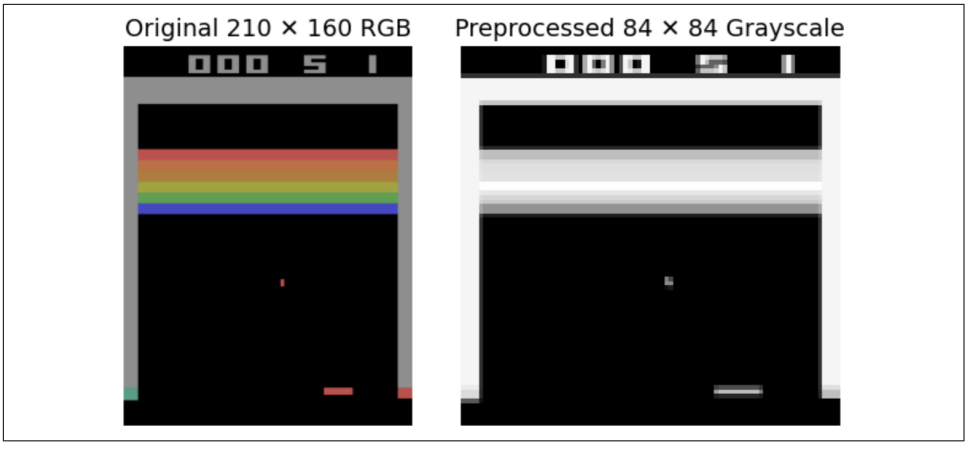

Figure 19-11. A Breakout frame before (left) and after (right) preprocessing

The ALE interface runs at 60 frames per second, which is quite fast, so consecutive frames look very similar, which wastes computation. To avoid this, the default *Breakout* environment repeats each action four times and returns only the final observation; this is called *frame skipping*. However, instead of skipping the frames, it's preferable to stack them into a single four-channel image and use that as the observation. For this, we must first avoid frame skipping: this is why we used the `BreakoutNoFrameskip-v4` environment rather than `Breakout-v4`.[24] Then, we must wrap the environment in a `VecFrameStack`; this wrapper environment will repeat each action several times (four in our case) and stack the resulting frames along the channel dimension (i.e., the last one):

```
from stable_baselines3.common.vec_env import VecFrameStack

envs_stacked = VecFrameStack(envs, n_stack=4)
obs = envs_stacked.reset()  # returns a 4 × 84 × 84 × 4 NumPy array
```

Now let's create a PPO model with some good hyperparameters:

```
from stable_baselines3 import PPO

ppo_model = PPO("CnnPolicy", envs_stacked, device=device, learning_rate=2.5e-4,
                batch_size=256, n_steps=256, n_epochs=4, clip_range=0.1,
                vf_coef=0.5, ent_coef=0.01, gamma=0.99, verbose=0)
```

24 The "v4" suffix is the version number; it's unrelated to frame skipping or the number of parallel environments.

That's a lot of arguments! Let's see what they do:

- The first argument is the policy network. Since we specified `CnnPolicy`, SB3 will build a good CNN for us, based on the chosen algorithm (PPO in this case) and the observation space. If you're curious, take a look at `ppo_model.policy` to see the CNN's architecture: it's a deep CNN with an actor head (for the action logits) and a critic head (for the state values). There are four possible actions: left, right, fire (to launch the ball), and no-op (do nothing). If you prefer to use a custom neural net, you must create a subclass of the `ActorCriticPolicy` class, located in the `stable_baselines3.common.policies` module. See SB3's documentation for more details.
- `env`, `device`, `learning_rate`, and `batch_size` are self-explanatory.
- `n_steps` is the number of environment steps to run (per environment) before each policy update.
- `n_epochs` is the number of training steps to run on each batch during optimization.
- `clip_range` limits the magnitude of the policy updates to avoid large changes that might cause catastrophic forgetting.
- `vf_coef` is the weight of the value function loss in the total loss (similar to our actor-critic's `critic_weight` hyperparameter).
- `ent_coef` is the weight of the entropy term that encourages exploration.
- `gamma` is the discount rate.
- `verbose` is the logging verbosity (0 = silent, 1 = info, 2 = debug).

For new tasks, the default `PPO` hyperparameters are a good place to start. If learning is too slow, try more parallel environments first; then consider using a higher learning rate or a larger clip range. You can also shrink `n_steps` or `batch_size`, but this risks noisier gradients. If learning is unstable, try lowering the learning rate or clip range, and use larger rollouts (i.e., `n_steps`) or batch sizes. Use `gamma` near 0.95 for short-horizon tasks, and 0.995 to 0.999 for long-horizon ones. Lastly, increase `ent_coef` if you want to encourage more exploration.

And now let's start training. The following code will train the model for 30 million steps. This will take many hours and will probably be too long for a Colab session (unless you get a paid subscription), so the notebook also includes code to download the trained model if you prefer. Whenever you train a model for a long time, it's important to save checkpoints at regular intervals (e.g., every 100,000 calls to the

`step()` method) to avoid having to start from scratch in case of a crash or a power outage. For this, we can create a checkpoint callback and pass it to the `learn()` method:

```
from stable_baselines3.common.callbacks import CheckpointCallback

cb = CheckpointCallback(save_freq=100_000, save_path="my_ppo_breakout.ckpt")
ppo_model.learn(total_timesteps=30_000_000, progress_bar=True, callback=cb)
ppo_model.save("my_ppo_breakout")  # save the final model
```

The `save_freq` argument counts calls to the `step()` method. Since there are 4 environments running in parallel, 50,000 calls correspond to 200,000 total time steps.

To see the progress during training, one option is to load the latest checkpoint in another notebook, and try it out. A simpler option is to use TensorBoard to visualize the learning curves, especially the mean reward per episode. For this, you must first activate the TensorBoard extension in Colab or Jupyter by running `%load_ext tensorboard` (this is done at the start of this chapter's notebook). Next, you must start the TensorBoard server, point it to a log directory, and choose the TCP port it will listen on. The following "magic" command (i.e., starting with a %) will do that and also open up the TensorBoard client interface directly inside Colab or Jupyter:

```
tensorboard_logdir = "my_ppo_breakout_tensorboard"  # path to the log directory
%tensorboard --logdir={tensorboard_logdir} --port 6006
```

Next, you must tell the PPO model where to save its TensorBoard logs. This is done when creating the model:

```
ppo_model = PPO("CnnPolicy", [...], tensorboard_log=tensorboard_logdir)
```

And that's it. Once you start training, you will see the learning curves change every 30 seconds or so in the TensorBoard interface (or click the refresh button). The most important metric to track is the `rollout/ep_rew_mean`, which is the mean reward per episode: it should slowly ramp up, even though it will sometimes go down a bit. After 1 million total steps it will typically reach around 20; that's not a very good agent. But if you let training run for 10 million steps, it should reach human level. And after 50 million steps, it will generally be superhuman.

Congratulations, you know how to train a superhuman AI! You can try it out like this:

```
ppo_model = PPO.load("my_ppo_agent_breakout")  # or load the best checkpoint
eval_env = make_atari_env("BreakoutNoFrameskip-v4", n_envs=1, seed=42)
eval_stacked = VecFrameStack(eval_env, n_stack=4)
frames = []
obs = eval_stacked.reset()
```

```
for _ in range(5000):  # some limit in case the agent never loses
    frames.append(eval_stacked.render())
    action, _ = ppo_model.predict(obs, deterministic=True) # for reproducibility
    obs, reward, done, info = eval_stacked.step(action)
    if done[0]:  # note: there's no `truncated`
        break

eval_stacked.close()
```

This will capture all the frames during one episode. You can render them as an animation using Matplotlib (see the notebook for an example). If you trained the agent for long enough (or used the pretrained model), you will see that the agent plays pretty well, and even found the strategy of digging tunnels on the sides and sending the ball through them: that's one of the best strategies in this game!

Overview of Some Popular RL Algorithms

Before we close this chapter, let's take a brief look at a few other popular algorithms:

AlphaGo[25]

AlphaGo uses a variant of *Monte Carlo tree search* (MCTS) based on deep neural networks to beat human champions at the game of Go. MCTS was invented in 1949 by Nicholas Metropolis and Stanislaw Ulam. It selects the best move after running many simulations, repeatedly exploring the search tree starting from the current position, and spending more time on the most promising branches. When it reaches a node that it hasn't visited before, it plays randomly until the game ends, and updates its estimates for each visited node (excluding the random moves), increasing or decreasing each estimate, depending on the final outcome.

AlphaGo is based on the same principle, but it uses a policy network to select moves, rather than playing randomly. This policy net is trained using policy gradients. The original algorithm involved three additional neural networks, and was more complicated, but it was simplified in the AlphaGo Zero paper,[26] which uses a single neural network to both select moves and evaluate game states. The AlphaZero paper[27] generalized this algorithm, making it capable of tackling not only the game of Go, but also chess and shogi (Japanese chess). Lastly, the MuZero paper[28] continued to improve upon this algorithm, outperforming the

25 David Silver et al., "Mastering the Game of Go with Deep Neural Networks and Tree Search", *Nature* 529 (2016): 484–489.

26 David Silver et al., "Mastering the Game of Go Without Human Knowledge", *Nature* 550 (2017): 354–359.

27 David Silver et al., "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm", arXiv preprint arXiv:1712.01815.

previous iterations even though the agent starts out without even knowing the rules of the game!

The rules of the game of Go were hardcoded into AlphaGo. In contrast, MuZero gradually learns a model of the environment: given a state *s* and an action *a*, it learns to predict the reward *r* and the probability of reaching state *s'*. Having a model of the environment (hardcoded or learned) allows these algorithms to plan ahead (in this case using MCTS). For this reason, both of these algorithms belong to the broad class of *model-based RL* algorithms. In contrast, policy gradients, value-based methods, and actor-critic methods are all *model-free RL* algorithms: they have a policy model and/or a value model, but not an *environment* model.

Curiosity-based exploration[29]

A recurring problem in RL is the sparsity of the rewards, which makes learning very slow and inefficient. Deepak Pathak and other UC Berkeley researchers have proposed an exciting way to tackle this issue: why not ignore the rewards and just make the agent extremely curious to explore the environment? The rewards thus become intrinsic to the agent, rather than coming from the environment. Similarly, stimulating curiosity in a child is more likely to give good results than purely rewarding the child for getting good grades.

How does this work? The agent continuously tries to predict the outcome of its actions, and it seeks situations where the outcome does not match its predictions. In other words, it wants to be surprised. If the outcome is predictable (boring), it goes elsewhere. However, if the outcome is unpredictable but the agent notices that it has no control over it, it also gets bored after a while. With only curiosity, the authors succeeded in training an agent at many video games: even though the agent gets no penalty for losing, it finds it boring to lose because the game starts over, so it learns to avoid it.

Open-ended learning (OEL)

The objective of OEL is to train agents capable of endlessly learning new and interesting tasks, typically generated procedurally. We're not there yet, but there has been some amazing progress over the last few years. For example, a 2019 paper[30] by a team of researchers from Uber AI introduced the *POET algorithm*, which generates multiple simulated 2D environments with bumps and holes, and

28 Julian Schrittwieser et al., "Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model", arXiv preprint arXiv:1911.08265 (2019).

29 Deepak Pathak et al., "Curiosity-Driven Exploration by Self-Supervised Prediction", *Proceedings of the 34th International Conference on Machine Learning* (2017): 2778–2787.

trains one agent per environment. The agent's goal is to walk as fast as possible while avoiding the obstacles.

The algorithm starts out with simple environments, but they gradually get harder over time: this is called *curriculum learning*. Moreover, although each agent is only trained within one environment, it must regularly compete against other agents, across all environments. In each environment, the winner is copied over and replaces the agent that was there before. This way, knowledge is regularly transferred across environments, and the most adaptable agents are selected.

In the end, the agents are much better walkers than agents trained on a single task, and they can tackle much harder environments. Of course, this principle can be applied to other environments and tasks as well. If you're interested in OEL, make sure to check out the Enhanced POET paper,[31] as well as DeepMind's 2021 paper[32] on this topic.

If you'd like to learn more about reinforcement learning, check out the book *Reinforcement Learning* by Phil Winder (O'Reilly).

We covered many topics in this chapter. We learned about policy gradient methods; we implemented the REINFORCE algorithm to solve the CartPole problem using Gymnasium; we explored Markov chains and Markov decision processes, which led us to value-based methods; and we implemented a deep Q-Learning model. Then we discussed actor-critic methods, and we used the Stable-Baselines3 library to implement a PPO model that beat the Atari game *Breakout*. Lastly, we took a peek at some of the other areas of RL, including model-based RL and more. Reinforcement learning is a huge and exciting field, with new ideas and algorithms popping out every day, so I hope this chapter sparked your curiosity. There is a whole world to explore!

30 Rui Wang et al., "Paired Open-Ended Trailblazer (POET): Endlessly Generating Increasingly Complex and Diverse Learning Environments and Their Solutions", arXiv preprint arXiv:1901.01753 (2019).

31 Rui Wang et al., "Enhanced POET: Open-Ended Reinforcement Learning Through Unbounded Invention of Learning Challenges and Their Solutions", arXiv preprint arXiv:2003.08536 (2020).

32 Open-Ended Learning Team et al., "Open-Ended Learning Leads to Generally Capable Agents", arXiv preprint arXiv:2107.12808 (2021).

Exercises

1. How would you define reinforcement learning? How is it different from regular supervised or unsupervised learning?
2. Can you think of three possible applications of RL that were not mentioned in this chapter? For each of them, what is the environment? What is the agent? What are some possible actions? What are the rewards?
3. What is the discount factor? Can the optimal policy change if you modify the discount factor?
4. How do you measure the performance of a reinforcement learning agent?
5. What is the credit assignment problem? When does it occur? How can you alleviate it?
6. What is the point of using a replay buffer?
7. What is an off-policy RL algorithm? What are the benefits?
8. What is a model-based RL algorithm? Can you give some examples?
9. Use policy gradients to solve Gymnasium's LunarLander-v2 environment.
10. Solve the BipedalWalker-v3 environment using the RL algorithm of your choice.
11. If you have about $100 to spare, you can purchase a Raspberry Pi 3 plus some cheap robotics components, install PyTorch on the Pi, and go wild! Start with simple goals, like making the robot turn around to find the brightest angle (if it has a light sensor) or the closest object (if it has a sonar sensor), and move in that direction. Then you can start using deep learning. For example, if the robot has a camera, you can try to implement an object detection algorithm so it detects people and moves toward them. You can also try to use RL to make the agent learn on its own how to use the motors to achieve that goal. Have fun!

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

Thank You!

Before we close the last chapter of this book, I would like to thank you for reading it up to the last paragraph. I truly hope that you had as much pleasure reading this book as I had writing it, and that it will be useful for your projects, big or small.

If you find errors, please send feedback. More generally, I would love to know what you think, so please don't hesitate to contact me via O'Reilly, or through the *ageron/handson-mlp* GitHub project.

Going forward, my best advice to you is to practice and practice: try going through all the exercises (if you have not done so already), play with the Jupyter notebooks, join Kaggle.com or some other ML community, watch ML courses, read papers, attend conferences, and meet experts. It also helps tremendously to have a concrete project to work on, whether it is for work or fun (ideally for both), so if there's anything you have always dreamt of building, give it a shot! Work incrementally; don't shoot for the moon right away, but stay focused on your project and build it piece by piece. It will require patience and perseverance, but when you have a walking robot, or a working chatbot, or whatever else you fancy to build, it will be immensely rewarding.

My greatest hope is that this book will inspire you to build a wonderful ML application that will benefit all of us! What will it be?

—Aurélien Geron
October 22, 2025

APPENDIX A
Autodiff

This appendix explains how PyTorch's automatic differentiation (autodiff) feature works, and how it compares to other solutions.

Suppose you define a function $f(x, y) = x^2y + y + 2$, and you need its partial derivatives $\partial f/\partial x$ and $\partial f/\partial y$, typically to perform gradient descent (or some other optimization algorithm). Your main options are manual differentiation, finite difference approximation, forward-mode autodiff, and reverse-mode autodiff. PyTorch implements reverse-mode autodiff, but to fully understand it, it's useful to look at the other options first. So let's go through each of them, starting with manual differentiation.

Manual Differentiation

The first approach to compute derivatives is to pick up a pencil and a piece of paper and use your calculus knowledge to derive the appropriate equation. For the function $f(x, y)$ just defined, it is not too hard; you just need to use five rules:

- The derivative of a constant is 0.
- The derivative of λx is λ (where λ is a constant).
- The derivative of x^λ is $\lambda x^{\lambda - 1}$, so the derivative of x^2 is $2x$.
- The derivative of a sum of functions is the sum of these functions' derivatives.
- The derivative of λ times a function is λ times its derivative.

From these rules, you can derive Equation A-1.

Equation A-1. Partial derivatives of f(x, y)

$$\frac{\partial f}{\partial x} = \frac{\partial\left(x^2 y\right)}{\partial x} + \frac{\partial y}{\partial x} + \frac{\partial 2}{\partial x} = y\frac{\partial\left(x^2\right)}{\partial x} + 0 + 0 = 2xy$$

$$\frac{\partial f}{\partial y} = \frac{\partial\left(x^2 y\right)}{\partial y} + \frac{\partial y}{\partial y} + \frac{\partial 2}{\partial y} = x^2 + 1 + 0 = x^2 + 1$$

This approach can become very tedious for more complex functions, and you run the risk of making mistakes. Fortunately, there are other options. Let's look at finite difference approximation now.

Finite Difference Approximation

Recall that the derivative $h'(x_0)$ of a function $h(x)$ at a point x_0 is the slope of the function at that point. More precisely, the derivative is defined as the limit of the slope of a straight line going through this point x_0 and another point x on the function, as x gets infinitely close to x_0 (see Equation A-2).

Equation A-2. Definition of the derivative of a function h(x) at point x_0

$$h'(x_0) = \lim_{x \to x_0} \frac{h(x) - h(x_0)}{x - x_0} = \lim_{\varepsilon \to 0} \frac{h(x_0 + \varepsilon) - h(x_0)}{\varepsilon}$$

So if we wanted to calculate the partial derivative of $f(x, y)$ with regard to x at $x = 3$ and $y = 4$, we could compute $f(3 + \varepsilon, 4) - f(3, 4)$ and divide the result by ε, using a very small value for ε. This type of numerical approximation of the derivative is called a *finite difference approximation*, and this specific equation is called *Newton's difference quotient*. That's exactly what the following code does:

```
def f(x, y):
    return x**2*y + y + 2

def derivative(f, x, y, x_eps, y_eps):
    return (f(x + x_eps, y + y_eps) - f(x, y)) / (x_eps + y_eps)

df_dx = derivative(f, 3, 4, 0.00001, 0)
df_dy = derivative(f, 3, 4, 0, 0.00001)
```

Unfortunately, the result is imprecise (and it gets worse for more complicated functions). The correct results are respectively 24 and 10, but instead we get:

```
>>> df_dx
24.000039999805264
```

```
>>> df_dy
10.000000000331966
```

Notice that to compute both partial derivatives, we have to call `f()` at least three times (we called it four times in the preceding code, but it could be optimized). If there were 1,000 parameters, we would need to call `f()` at least 1,001 times. When you are dealing with large neural networks, this makes finite difference approximation way too inefficient.

However, this method is so simple to implement that it is a great tool to check that the other methods are implemented correctly. For example, if it disagrees with your manually derived function, then your function probably contains a mistake.

So far, we have considered two ways to compute gradients: using manual differentiation and using finite difference approximation. Unfortunately, both are fatally flawed for training a large-scale neural network. So let's turn to autodiff, starting with forward mode.

Forward-Mode Autodiff

Figure A-1 shows how forward-mode autodiff works on an even simpler function, $g(x, y) = 5 + xy$. The graph for that function is represented on the left. After forward-mode autodiff, we get the graph on the right, which represents the partial derivative $\partial g/\partial x = 0 + (0 \times x + y \times 1) = y$ (we could similarly obtain the partial derivative with regard to y).

The algorithm will go through the computation graph from the inputs to the outputs (hence the name "forward mode"). It starts by getting the partial derivatives of the leaf nodes. The constant node (5) returns the constant 0, since the derivative of a constant is always 0. The variable x returns the constant 1 since $\partial x/\partial x = 1$, and the variable y returns the constant 0 since $\partial y/\partial x = 0$ (if we were looking for the partial derivative with regard to y, it would be the reverse).

Now we have all we need to move up the graph to the multiplication node in function g. Calculus tells us that the derivative of the product of two functions u and v is $\partial(u \times v)/\partial x = \partial v/\partial x \times u + v \times \partial u/\partial x$. We can therefore construct a large part of the graph on the right, representing $0 \times x + y \times 1$.

Finally, we can go up to the addition node in function g. As mentioned, the derivative of a sum of functions is the sum of these functions' derivatives, so we just need to create an addition node and connect it to the parts of the graph we have already computed. We get the correct partial derivative: $\partial g/\partial x = 0 + (0 \times x + y \times 1)$.

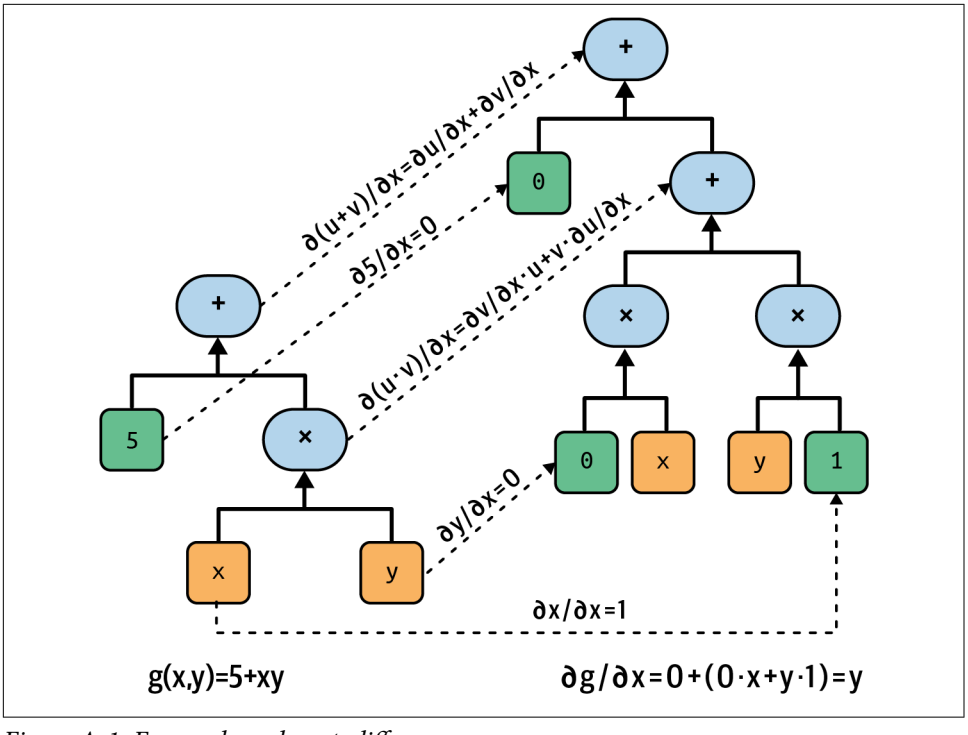

Figure A-1. Forward-mode autodiff

However, this equation can be simplified (a lot). By applying a few pruning steps to the computation graph to get rid of all the unnecessary operations, we get a much smaller graph with just one node: $\partial g/\partial x = y$. In this case simplification is fairly easy, but for a more complex function, forward-mode autodiff can produce a huge graph that may be tough to simplify and lead to suboptimal performance.

Note that we started with a computation graph, and forward-mode autodiff produced another computation graph. This is called *symbolic differentiation*, and it has two nice features. First, once the computation graph of the derivative has been produced, we can use it as many times as we want to compute the derivatives of the given function for any value of x and y. Second, we can run forward-mode autodiff again on the resulting graph to get second-order derivatives if we ever need to (i.e., derivatives of derivatives). We could even compute third-order derivatives, and so on.

But it is also possible to run forward-mode autodiff without constructing a graph (i.e., numerically, not symbolically) just by computing intermediate results on the fly. One way to do this is to use *dual numbers*, which are weird but fascinating numbers of the form $a + b\varepsilon$, where a and b are real numbers, and ε is an infinitesimal number such that $\varepsilon^2 = 0$ (but $\varepsilon \neq 0$). You can think of the dual number $42 + 24\varepsilon$ as something akin to $42.0000\cdots000024$ with an infinite number of 0s (but of course this

is simplified just to give you some idea of what dual numbers are). A dual number is represented in memory as a pair of floats. For example, 42 + 24ε is represented by the pair (42.0, 24.0).

Dual numbers can be added, multiplied, and so on, as shown in Equation A-3.

Equation A-3. A few operations with dual numbers

$$\lambda(a + b\varepsilon) = \lambda a + \lambda b\varepsilon$$
$$(a + b\varepsilon) + (c + d\varepsilon) = (a + c) + (b + d)\varepsilon$$
$$(a + b\varepsilon) \times (c + d\varepsilon) = ac + (ad + bc)\varepsilon + (bd)\varepsilon^2 = ac + (ad + bc)\varepsilon$$

Most importantly, it can be shown that $h(a + b\varepsilon) = h(a) + b \times h'(a)\varepsilon$, so computing $h(a + \varepsilon)$ gives you both $h(a)$ and the derivative $h'(a)$ in just one shot. Figure A-2 shows that the partial derivative of $f(x, y)$ with regard to x at $x = 3$ and $y = 4$ (which I will write as $\partial f/\partial x$ (3, 4)) can be computed using dual numbers. All we need to do is compute $f(3 + \varepsilon, 4)$; this will output a dual number whose first component is equal to $f(3, 4)$ and whose second component is equal to $\partial f/\partial x$ (3, 4).

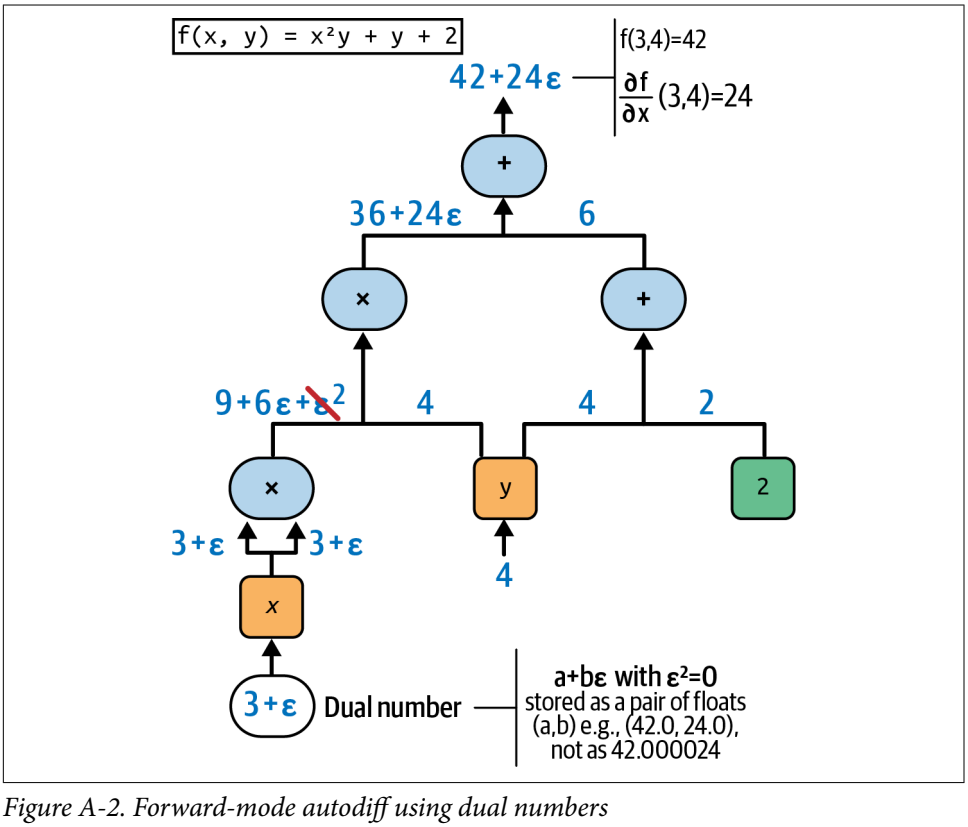

Figure A-2. Forward-mode autodiff using dual numbers

To compute $\partial f/\partial y$ (3, 4) we would have to go through the graph again, but this time with $x = 3$ and $y = 4 + \varepsilon$.

So, forward-mode autodiff is much more accurate than finite difference approximation, but it suffers from the same major flaw, at least when there are many inputs and few outputs (as is the case when dealing with neural networks): if there were 1,000 parameters, it would require 1,000 passes through the graph to compute all the partial derivatives. This is where reverse-mode autodiff shines: it can compute all of them in just two passes through the graph. Let's see how.

Reverse-Mode Autodiff

Reverse-mode autodiff is the solution implemented by PyTorch. It first goes through the graph in the forward direction (i.e., from the inputs to the output) to compute the value of each node. Then it does a second pass, this time in the reverse direction (i.e., from the output to the inputs) to compute all the partial derivatives. The name "reverse mode" comes from this second pass through the graph, where gradients flow in the reverse direction. Figure A-3 represents the second pass. During the first pass, all the node values were computed, starting from $x = 3$ and $y = 4$. You can see those values at the bottom right of each node (e.g., $x \times x = 9$). The nodes are labeled n_1 to n_7 for clarity. The output node is n_7: $f(3, 4) = n_7 = 42$.

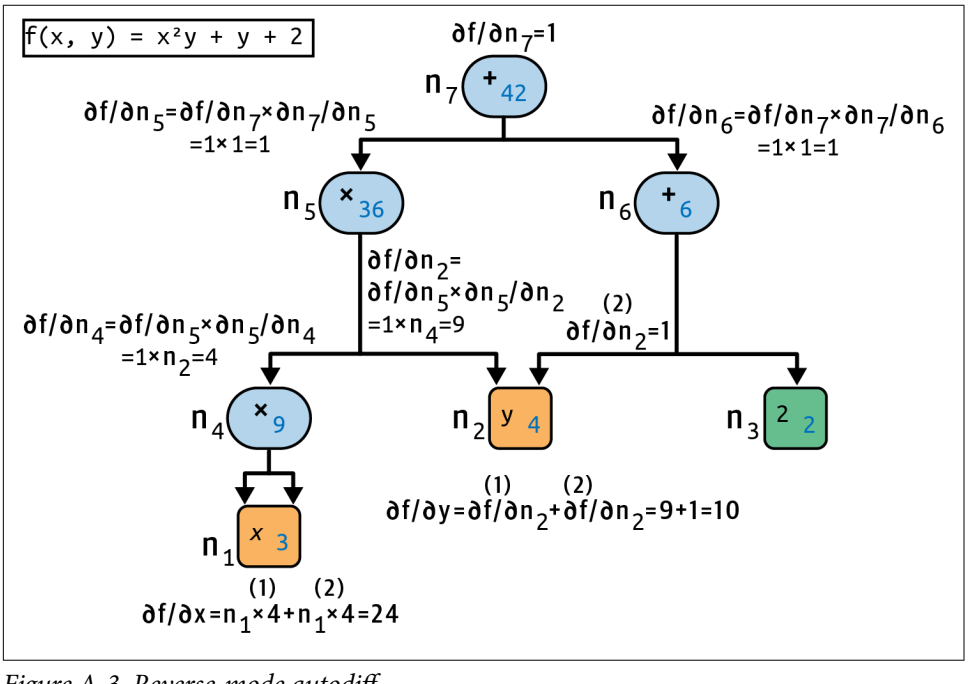

Figure A-3. Reverse-mode autodiff

The idea is to gradually go down the graph, computing the partial derivative of $f(x, y)$ with regard to each consecutive node, until we reach the variable nodes. For this, reverse-mode autodiff relies heavily on the *chain rule*, shown in Equation A-4.

Equation A-4. Chain rule

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial n_i} \times \frac{\partial n_i}{\partial x}$$

Since n_7 is the output node, $f = n_7$ so $\partial f / \partial n_7 = 1$.

Let's continue down the graph to n_5: how much does f vary when n_5 varies? The answer is $\partial f / \partial n_5 = \partial f / \partial n_7 \times \partial n_7 / \partial n_5$. We already know that $\partial f / \partial n_7 = 1$, so all we need is $\partial n_7 / \partial n_5$. Since n_7 simply performs the sum $n_5 + n_6$, we find that $\partial n_7 / \partial n_5 = 1$, so $\partial f / \partial n_5 = 1 \times 1 = 1$.

Now we can proceed to node n_4: how much does f vary when n_4 varies? The answer is $\partial f / \partial n_4 = \partial f / \partial n_5 \times \partial n_5 / \partial n_4$. Since $n_5 = n_4 \times n_2$, we find that $\partial n_5 / \partial n_4 = n_2$, so $\partial f / \partial n_4 = 1 \times n_2 = 4$.

The process continues until we reach the bottom of the graph. At that point we will have calculated all the partial derivatives of $f(x, y)$ at the point $x = 3$ and $y = 4$. In this example, we find $\partial f / \partial x = 24$ and $\partial f / \partial y = 10$. Sounds about right!

Reverse-mode autodiff is a very powerful and accurate technique, especially when there are many inputs and few outputs, since it requires only one forward pass plus one reverse pass per output to compute all the partial derivatives for all outputs with regard to all the inputs. When training neural networks, we generally want to minimize the loss, so there is a single output (the loss), and hence only two passes through the graph are needed to compute the gradients.

PyTorch builds a new graph on the fly during each forward pass. Whenever you run an operation on a tensor with `requires_grad=True`, PyTorch computes the resulting tensor and sets its `grad_fn` attribute to an operation-specific object that allows PyTorch to propagate the gradients backwards through this operation. Since the graph is built on the fly, your code can be highly dynamic, containing loops and conditionals, and everything will still work fine.

Reverse-mode autodiff can also handle functions that are not entirely differentiable, as long as you ask it to compute the partial derivatives at points that *are* differentiable.

Creating a tiny autodiff framework is a great exercise to truly master autodiff. Try creating one from scratch for a small set of operations. If you get stuck, check out this project's `extra_autodiff.ipynb` notebook, which you can run on Colab at *https://homl.info/colab-p*. You can also watch Andrej Karpathy's excellent YouTube video where he builds the micrograd library from scratch.

APPENDIX B
Mixed Precision and Quantization

By default, PyTorch uses 32-bit floats to represent model parameters: that's 4 bytes per parameter. If your model has 1 billion parameters, then you need at least 4 GB of RAM just to hold the model. At inference time you also need enough RAM to store the activations, and at training time you need enough RAM to store all the intermediate activations as well (for the backward pass), and to store the optimizer parameters (e.g., Adam needs two additional parameters for each model parameter—that's an extra 8 GB). This is a lot of RAM, and it's also plenty of time spent transferring data between the CPU and the GPU, not to mention storage space, download time, and energy consumption.

So how can we reduce the model's size? A simple option is to use a reduced precision float representation—typically 16-bit floats instead of 32-bit floats. If you train a 32-bit model then shrink it to 16-bits after training, its size will be halved, with little impact on its quality. Great!

However, if you try to train the model using 16-bit floats, you may run into convergence issues, as we will see. So a common strategy is *mixed-precision training* (MPT), where we keep the weights and weight updates at 32-bit precision during training, but the rest of the computations use 16-bit precision. After training, we shrink the weights down to 16-bits.

Finally, to shrink the model even further, you can use *quantization*: the parameters are discretized and represented as 8-bit integers, or even 4-bit integers or less. This is harder, and it degrades the model's quality a bit more, but it reduces the model size by a factor of 4 or more, and speeds it up significantly.

In this appendix, we will cover reduced precision, mixed-precision training, and quantization. But to fully understand these, we must first discuss common number representations in machine learning.

Common Number Representations

By default, PyTorch represents weights and activations using 32-bit floats based on the *IEEE Standard for Floating-Point Arithmetic* (IEEE 754), which specifies how floating-point numbers are represented in memory. It's a flexible and efficient format which can represent tiny values and huge values, as well as special values such as ±0,[1] ±infinity, and NaN (i.e., Not a Number).

The float32 data type (fp32 for short) can hold numbers as small as $\pm 1.4e^{-45}$ and as large as $\pm 3.4e^{38}$. It is represented at the top of Figure B-1. The first bit determines the *sign* S: 0 means positive, 1 means negative. The next 8 bits hold the *exponent* E, ranging from 0 to 255. And the last 23 bits represent the *fraction* F, ranging from 0 to $2^{23} - 1$. Here is how to compute the value:

- If E is between 1 and 254, then the number is called *normalized*: this is the most common scenario. In this case, the value v can be computed using $v = (-1)^S \cdot 2^{E-127} \cdot (1 + F \cdot 2^{-23})$. The last term $(1 + F \cdot 2^{-23})$ corresponds to the most significant digits, so it's called the *significand*.
- If $E = 0$ and $F > 0$, then the number is called *subnormal*: it is useful to represent the tiniest values.[2] In this case, $v = (-1)^S \cdot 2^{E+1-127} \cdot (0 + F \cdot 2^{-23}) = (-1)^S \cdot F \cdot 2^{-149}$.
- If $E = 0$ and $F = 0$, then $v = \pm 0$.
- If $E = 255$ and $F > 0$, then $v =$ NaN.
- If $E = 255$ and $F = 0$, then $v = \pm$infinity.

The other floating-point formats represented in Figure B-1 differ only by the number of bits used for the exponent and the fraction. For example float16 uses 5 bits for the exponent (i.e., it ranges from 0 to 31) and 10 bits for the fraction (ranging from 0 to 1,023), while float8 uses 4 bits for the exponent (from 0 to 15) and 3 bits for the fraction, so it's often denoted fp8 E4M3.[3] The equations to compute the value are adjusted accordingly, for example normalized float16 values are computed using $v = (-1)^S \cdot 2^{E-15} \cdot (1 + F \cdot 2^{-10})$.

1 In general, –0 and +0 are considered equal, but some operations give different results, for example 1 / –0 = –infinity, while 1 / +0 = +infinity.

2 Some high-performance computing applications deactivate subnormal numbers because they slow down computations, and normalized numbers are generally sufficient (e.g., normalized fp32 can represent numbers as small as $\pm 1.2e^{-38}$).

3 The *M* stands for *mantissa*, which is a term often used as a synonym for fraction. Unfortunately, it's also used as a synonym for significand, leading to some confusion. This is why IEEE 754 no longer uses the term mantissa.

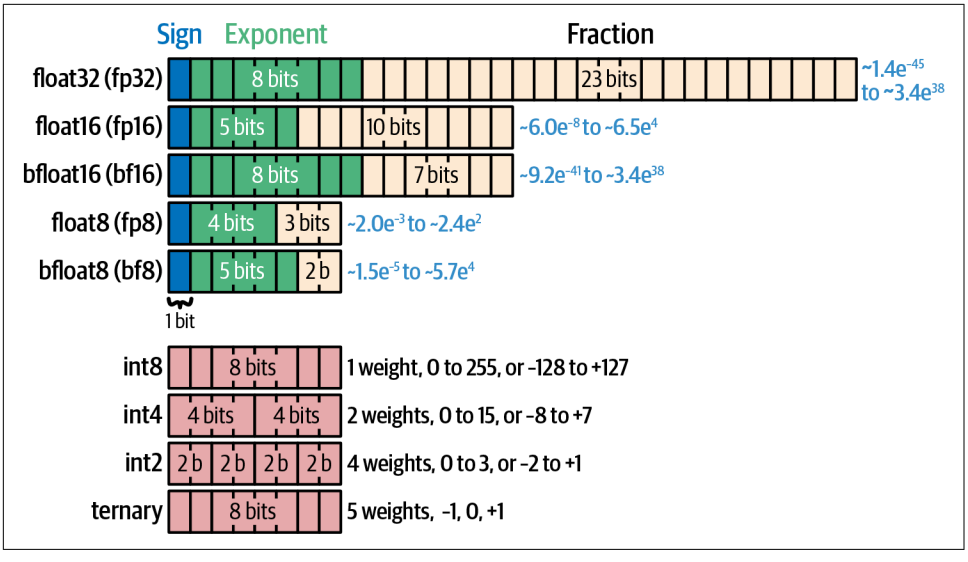

Figure B-1. Common number representations in machine learning

The bfloat16 and bfloat8 formats were proposed by Google Brain (hence the *b*), and they offer a wider range for the values, at the cost of a significantly reduced precision. We will come back to that.

Integers are often represented using 64 bits, with values ranging from 0 to 2^{64} – 1 (about $1.8e^{19}$) for unsigned integers, or –2^32 to 2^32 – 1 (about $\pm 4.3e^{9}$) for signed integers. Integers are also frequently represented using 32 bits, 16 bits, or 8 bits depending on the use case. In Figure B-1, I only represented the integer types frequently used for quantization, such as 8-bit integers (which can be unsigned or signed).

When quantizing down to 4 bits, we usually pack 2 weights per byte, and when quantizing down to 2 bits, we pack 4 weights per byte. It's even possible to quantize down to ternary values, where each weight can only be equal to –1, 0, or +1. In this case, it's common to store five weights per byte. For example, the byte 178 can be written as 20121 in base 3 (since $178 = 2 \cdot 3^4 + 0 \cdot 3^3 + 1 \cdot 3^2 + 2 \cdot 3^1 + 1 \cdot 3^0$), and if we subtract 1 from each digit, we get 1, –1, 0, 1, 0: these are the 5 ternary weights stored in this single byte. Since $3^5 = 243$, which is less than 256, we can fit five ternary values into one byte. This format only uses 1.6 bits per weight on average, which is 20 times less than using 32-bit floats!

It's technically possible to quantize weights down to a single bit each, storing 8 weights per byte: each bit represents a weight equal to either –1 or +1 (or sometimes 0 or 1). However, it's very difficult to get reasonable accuracy using such severe quantization.

As you can see, PyTorch's default weight representation (32-bit floats) takes up a *lot* of space compared to other representations: there is room for us to shrink our models quite a bit! Let's start by reducing the precision from 32 bits down to 16 bits.

Reduced Precision Models

If you have a 32-bit PyTorch model, you can convert all of its parameters to 16-bit floats—which is called *half-precision*—by calling the model's `half()` method:

```
import torch
import torch.nn as nn

model = nn.Sequential(nn.Linear(10, 100), nn.ReLU(), nn.Linear(100, 1))
# [...] pretend the 32-bit model is trained here
model.half()  # convert the model parameters to half precision (16 bits)
```

This is a quick and easy way to halve the size of a trained model, usually without much impact on its quality. Moreover, since many GPUs have 16-bit float optimizations, and since there will be less data to transfer between the CPU and the GPU, the model will typically run almost twice as fast.

When downloading a pretrained model using the Transformers library's `from_pretrained()` method, you can set `dtype="auto"` to let the library choose the optimal float representation for your hardware.

To use the model, you now need to feed it 16-bit inputs, and it will output 16-bit outputs as well:

```
X = torch.rand(3, 10, dtype=torch.float16)  # some 16-bit input
y_pred = model(X)  # 16-bit output
```

But what if you want to build and train a 16-bit model right from the start? In this case, you can set `dtype=torch.float16` whenever you create a tensor or a module with parameters, for example:

```
model = nn.Sequential(nn.Linear(10, 100, dtype=torch.float16), nn.ReLU(),
                      nn.Linear(100, 1, dtype=torch.float16))
```

If you prefer to avoid repeating `dtype=torch.float16` everywhere, then you can instead set the default data type to `torch.float16` using `torch.set_default_dtype(torch.float16)`. Be careful: this will apply to *all* tensors and modules created after that.

However, the reduced precision can cause some issues during training. Indeed, 16-bit floats have a limited *dynamic range* (i.e., the ratio between the largest and smallest

positive representable values): the smallest positive representable value is about 0.00000006 (i.e., $6.0e^{-8}$), while the largest is 65,504 (i.e., $\sim 6.5e^{4}$). This implies that any gradient update smaller than $\sim 6.0e^{-8}$ will *underflow*, meaning it will be rounded down to zero, and thus ignored. And conversely, any value larger than $\sim 6.5e^{4}$ will *overflow*, meaning it will be rounded up to infinity, causing training to fail (once some weights are infinite, the loss will be infinite or NaN).

To avoid underflows, one solution is to scale up the loss by a large factor (e.g., multiply it by 256): this will automatically scale up the gradients by the same factor during the backward pass, which will prevent them from being smaller than the smallest 16-bit representable value. However, you must scale the gradients back down before performing an optimizer step, and at this point you may get an underflow. Also, if you scale up the loss too much, you will run into overflows.

If you can't find a good scaling factor that avoids both underflows and overflows, you can try to use `torch.bfloat16` rather than `torch.float16`, since bloat16 has more bits for the exponent: the smallest value is $\sim 9.2e^{-41}$, while the largest is $\sim 3.4e^{38}$, so there's less risk of any significant gradient updates being ignored, or reasonable values being rounded up to infinity.

However, bfloat16 has historically had less hardware support (although this is improving), and it offers fewer bits for the fraction, which can cause some gradient updates to be ignored when the corresponding parameter values are much larger, causing training to stall. For example, if the gradient update is $4.5e^{-2}$ (i.e., 0.045) and the corresponding parameter value is equal to $1.23e^{2}$ (i.e., 123), then the sum should be $1.23045e^{2}$ (i.e., 123.045) but bfloat16 does not have enough fraction bits to store all these digits, so it must round the result to $1.23e^{2}$ (i.e., 123): as you can see, the gradient update is completely ignored. With regular 16-bit floats, the result would be 123.0625, which is not exactly right due to floating-point precision errors, but at least the parameter makes a step in the right direction. That said, if the gradient update was a bit smaller (e.g., 0.03), it would be ignored even in regular 16-bit float precision.

So if you try float16 and bfloat16 but you still encounter convergence issues during training, then you can try *mixed-precision training* instead.

Mixed-Precision Training

Mixed-precision training (MPT) was proposed by Baidu and Nvidia researchers in 2017,[4] to address the issues often observed with 16-bit training. Here's how it works:

4 P. Micikevicius et al., "Mixed Precision Training", arXiv preprint 2017, ICLR (2018).

- MPT stores a primary copy of the model parameters as 32-bit floats, and at each training iteration, it creates a 16-bit copy of these model parameters (see step 1 in Figure B-2), and uses them for the forward pass (step 2).
- The loss is then scaled up by a large factor (step 3) to avoid underflows, as we discussed earlier.
- Lastly, we switch back to 32-bit precision to scale the gradients back down: this greater precision avoids the risk of underflow. Next we use the gradients to perform one optimizer step, improving the primary parameters (step 5). Performing the actual optimizer step in 32-bit precision ensures that small weight updates are not ignored when applied to much larger parameter values, since 32-bit floats have a very large fraction (23 bits).

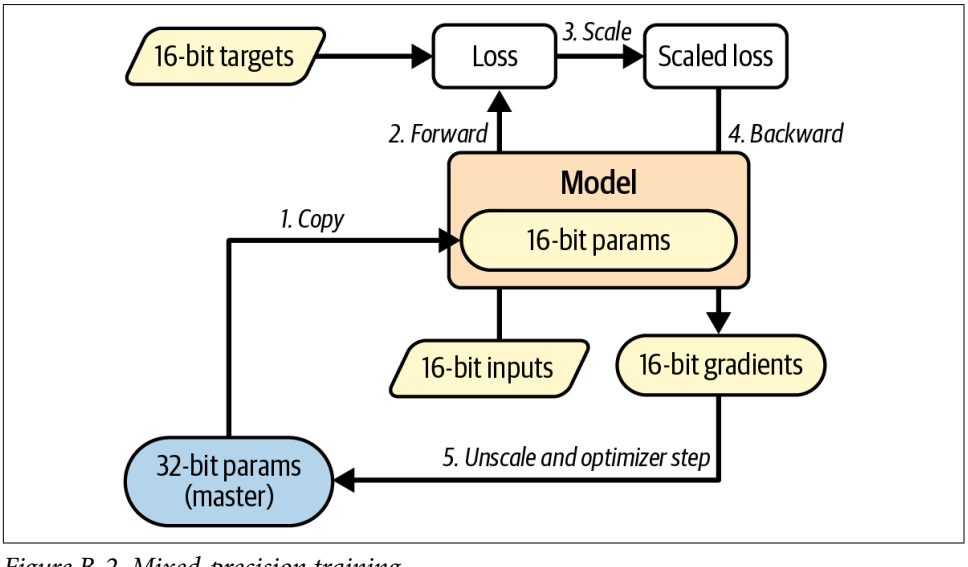

Figure B-2. Mixed-precision training

MPT offers almost all of the benefits of 16-bit training, without the instabilities. However, the model parameters take 50% more space than in 32-bit training because of the 16-bit copy at each training iteration, so how is this any better? Well, during training, most of the RAM is used to store the activations, not the model parameters, so in practice MPT requires just a bit more than half the RAM used by regular 32-bit training. And it typically runs twice as fast. Moreover, once training is finished, we no longer need 32-bit parameters, we can convert them to 16 bits, and we get a pure 16-bit model.

MPT does not always accelerate training: it depends on the model, the batch size, and the hardware. That said, most large transformers are trained using MPT.

Rather than finding the best scaling factor by trial and error, you can run training in 32-bit precision for a little while (assuming you have enough RAM) and measure the gradient statistics to find the optimal scaling factor for your task: it should be large enough to avoid underflows, and small enough to avoid overflows.

Alternatively, your training script can adapt the factor dynamically during training: if some gradients are infinite or NaN, this means that an overflow occurred so the factor must be reduced (e.g., halved) and the training step must be skipped, but if no overflow is detected then the scaling factor can be gradually increased (e.g., doubled every 2,000 training steps). PyTorch provides a `torch.amp.GradScaler` class that implements this approach, and also scales down the learning rate appropriately.

PyTorch also provides a `torch.autocast()` function that returns a context within which many operations will automatically run in 16-bit precision. This includes operations that typically benefit the most from 16-bit precision, such as matrix multiplication and convolutions, but it does not include operations like reductions (e.g., `torch.sum()`) since running these in half precision offers no significant benefit and can damage precision.

Let's update our training function to run the forward pass within an autocast context and use a `GradScaler` to dynamically scale the loss:

```
from torch.amp import GradScaler

def train_mpt(model, optimizer, criterion, train_loader, n_epochs,
              dtype=torch.float16, init_scale=2.0**16):
    grad_scaler = GradScaler(device=device, init_scale=init_scale)
    model.train()
    for epoch in range(n_epochs):
        for X_batch, y_batch in train_loader:
            X_batch, y_batch = X_batch.to(device), y_batch.to(device)
            with torch.autocast(device_type=device, dtype=dtype):
                y_pred = model(X_batch)
                loss = criterion(y_pred, y_batch)
            grad_scaler.scale(loss).backward()
            grad_scaler.step(optimizer)
            grad_scaler.update()
            optimizer.zero_grad()
```

When fine-tuning a transformer using the Hugging Face Transformers library, you can set `fp16=True` or `bf16=True` in the `TrainingArguments` to activate mixed-precision training.

Reducing precision down to 16-bits often works great, but can we shrink our models even further? Yes, we can, using quantization.

Quantization

Quantization means mapping continuous values to discrete ones. In deep learning, this typically involves converting parameters, and often activations as well, from floats to integers—usually 32-bit floats to 8-bit integers. More generally, the goal is to shrink and speed up our model by reducing the number of bits used in parameters, and often in activations as well. Moreover, some embedded devices (e.g., ARM Cortex-M0) do not support floating-point operations at all (in part to reduce their cost and energy consumption), so models have to be quantized entirely (both weights and activations) before you can use them on the device. Modern smartphones do support floating point operations but still benefit significantly from quantization: int8 operations are 2 to 4 times faster and use 5 to 10 times less energy than FP32.

The simplest approach is *linear quantization*, so we'll discuss it now. We will discuss a few nonlinear quantization methods later in this appendix.

Linear Quantization

Linear quantization dates back to digital signal processing in the 1950s, but it has become particularly important in machine learning over the past decade since models have become gigantic, and yet we wish to run them on mobile phones and other limited devices. It has two variants: asymmetric and symmetric. In *asymmetric linear quantization*, float values are simply mapped linearly to unsigned bytes with values ranging from 0 to 255 (or more generally from 0 to $2^n - 1$ when quantizing to n-bit integers). For example, if the weights range between $a = -0.1$ and $b = 0.6$, then the float –0.1 will be mapped to the byte 0, the float 0.0 to integer 36, 0.1 to 72, …, 0.6 to 255, and more generally, the float tensor $\mathbf{w}$ will be mapped to the integer tensor $\mathbf{q}$ using Equation B-1.

Equation B-1. Asymmetric linear quantization

$$q_i = \text{round}\left(\frac{w_i}{s}\right) + z$$

$$\text{with } s = \frac{b-a}{2^n - 1} \text{ and } z = -\text{round}\left(\frac{a}{s}\right)$$

$$\text{where } a = \min_i w_i \text{ and } b = \max_i w_i$$

In this equation:

- w_i is the i^{th} float in the original tensor **w**.
- q_i is the i^{th} integer in the quantized tensor **q**. It is clamped between 0 and $2^n - 1$ (e.g., 255 for 8-bit quantization).
- s is the *quantization scale*. Note that some authors define it as 1 / s and adapt the equation accordingly (i.e., they multiply rather than divide).
- z is the *quantization bias* or *zero point*.
- a is the minimum value of **w**, and b is the maximum value of **w**.

The range $[a, b]$ is known for weights, since their values do not change after training. However, the range of activation values depends on the inputs we feed to the model. As a result, for each activation that we want to quantize (e.g., the inputs of each layer), we will either have to compute a and b on the fly for each new input batch (this is called *dynamic quantization*) or run a calibration dataset once through the model to determine the typical range of activation values, then use this range to quantize the activations of all subsequent batches (this is called *static quantization*). Static quantization is a faster but less precise.

To approximately recover the original value w_i from a quantized value q_i, we can compute $w_i \approx s \times (q_i - z)$. This is called *dequantization*. For example, if $q_i = 72$, then we get $w_i \approx 0.0988$, which is indeed close to 0.1. The difference between the dequantized value (0.0988) and the original value (0.1) is called the *quantization noise*: with 8-bit quantization, the quantization noise usually leads to a slightly degraded accuracy. With 6-bit, 4-bit, or less, the quantization noise can hurt even more, especially since it has a cumulative effect: the deeper the network, the stronger the impact.

Equation B-1 guarantees that any float equal to 0.0 can be quantized and dequantized back to 0.0 exactly: indeed, if $w_i = 0.0$ then $q_i = z$, and dequantizing q_i gives back $w_i = s \times (z - z) = 0.0$. This is particularly useful for sparse models where many weights are equal to zero. It is also important when using activations like ReLU which produce many zero activations.

In PyTorch, the `torch.quantize_per_tensor()` function lets you create a quantized tensor: this is a special kind of tensor that contains the quantized values (i.e., integers), as well as the *quantization parameters* (i.e., the scale and zero point). Let's use this function to quantize a tensor, then dequantize it. In this example we will use the data type `torch.quint8`, which uses 8-bit unsigned integers:

```
>>> w = torch.tensor([0.1, -0.1, 0.6, 0.0])  # 32-bit floats
>>> s = (w.max() - w.min()) / 255.  # compute the scale
>>> z = -(w.min() / s).round()  # compute the zero point
>>> qw = torch.quantize_per_tensor(w, scale=s, zero_point=z, dtype=torch.quint8)
>>> qw  # this is a quantized tensor internally represented using integers
tensor([ 0.0988, -0.0988,  0.6012,  0.0000], size=(4,), dtype=torch.quint8,
       quantization_scheme=torch.per_tensor_affine, scale=0.002745098201557994,
       zero_point=36)
>>> qw.dequantize()  # back to 32-bit floats (close to the original tensor)
tensor([ 0.0988, -0.0988,  0.6012,  0.0000])
```

Quantizing a model to 8-bits divides its size by almost 4. For example, suppose we have a convolutional layer with 64 kernels, 3 × 3 each, and it has 32 input channels. This layer requires 64 × 32 × 3 × 3 = 18,432 parameters (ignoring the bias terms). That's 18,432 × 4 = 73,728 bytes before quantization, and just 18,432 bytes after quantization, plus 2 × 4 = 8 bytes to store s and z (indeed, they are both stored as 32-bit floats, so 4 bytes each).

PyTorch also has a `torch.quantize_per_channel()` function which quantizes each channel separately: this offers better precision but requires a bit more space for the additional quantization parameters.

When the float values are approximately symmetric around zero, we can use *symmetric linear quantization*, where the values are mapped between –127 and +127, or more generally between $-r$ and $+r$ with $r = 2^{n-1} - 1$, using Equation B-2.

Equation B-2. Symmetric linear quantization

$$q_i = \text{round}\left(\frac{w_i}{s}\right) \text{ with } s = \frac{\max_i |w_i|}{2^{n-1} - 1}$$

To implement symmetric linear quantization in PyTorch, we can use the `torch.quantize_per_tensor()` function again, but using a zero point equal to 0, and data type `qint8` (quantized signed 8-bit integer):

```
>>> w = torch.tensor([0.0, -0.94, 0.92, 0.93])  # 32-bit floats
>>> s = w.abs().max() / 127.
>>> qw = torch.quantize_per_tensor(w, scale=s, zero_point=0, dtype=torch.qint8)
>>> qw
```

```
tensor([ 0.0000, -0.9400,  0.9178,  0.9326], size=(4,), dtype=torch.qint8,
       quantization_scheme=torch.per_tensor_affine, scale=0.007401574868708849,
       zero_point=0)
```

Figure B-3 shows some floats ranging between –0.94 and +0.93, quantized to signed bytes (i.e., 8-bits) ranging between –127 and +127,[5] using symmetric linear quantization. Notice that float 0.0 is always mapped to integer 0.

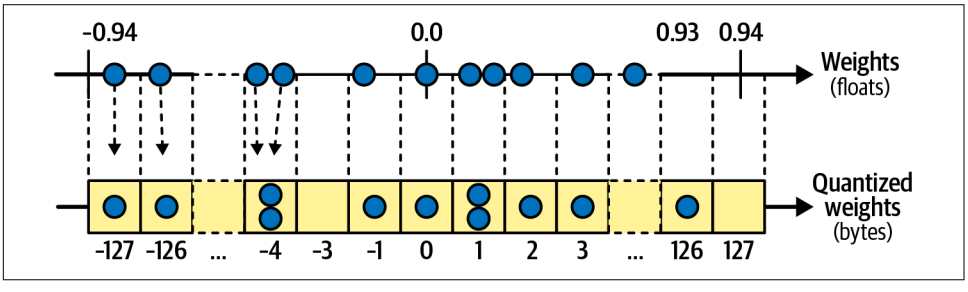

Figure B-3. Symmetric linear quantization

Symmetric mode is often a bit faster than asymmetric mode, because there's no zero point z to worry about. However, if the values are not symmetric, part of the integer range will be wasted. For example, if all the weights are positive, then symmetric mode will only use bytes 0 to 127 (rather than –127 to 127). As a result, symmetric mode can be a bit less precise than asymmetric mode. In practice, symmetric mode is generally preferred for weights (which are often fairly symmetric), and asymmetric mode for activations (especially when using ReLU, since it outputs only nonnegative values).

Let's now see how to quantize your models in practice using PyTorch's `torch.ao.quantization` package. The first approach is to quantize a trained model, which is called *post-training quantization* (PTQ). The second is to train (or fine-tune) your model with some fake quantization to get it used to the noise: this is called *quantization-aware training* (QAT). Let's start with PTQ.

Post-Training Quantization Using torch.ao.quantization

The `torch.ao` package contains tools for architecture optimization (hence the name), including pruning, sparsity, and quantization. The `torch.ao.quantization` package offers two solutions to quantize trained models: dynamic quantization and static quantization. Let's see how to implement both.

5 PyTorch implements *restricted symmetric quantization*, meaning that it excludes the lowest possible signed integer (e.g., –128 for 8-bit integers) to ensure that the range is symmetric (e.g., –127 to +127). Some other implementations allow the full signed byte range (from –128 to +127): this is called *unrestricted symmetric quantization*. These implementations also subtract 0.5 instead of 1 in the denominator of Equation B-2.

Dynamic quantization

Dynamic quantization is best for MLPs, RNNs, and transformers. To implement it using PyTorch's `torch.ao.quantization` package, you must first choose a quantization engine: PyTorch currently supports the *Facebook General Matrix Multiplication* (FBGEMM) engine for x86 CPUs, plus a newer x86 engine that supports recent x86 CPUs but is less battle-tested, and finally the *Quantized Neural Networks Package* (QNNPACK) engine for ARM/mobile. This code will pick the appropriate engine depending on the platform:

```
import platform

machine = platform.machine().lower()
engine = "qnnpack" if ("arm" in machine or "aarch64" in machine) else "x86"
```

PyTorch does not offer an engine for CUDA or other hardware accelerators, but other libraries do, such as the bitsandbytes library (as we will see shortly).

Once you have selected an engine, you can use the `quantize_dynamic()` function from the `torch.ao.quantization` package; just pass it your trained model, tell it the types of layers to quantize (typically just the `Linear` and RNN layers), specify the quantized data type, and boom, you have a ready-to-use quantized model:

```
from torch.ao.quantization import quantize_dynamic

model = nn.Sequential(nn.Linear(10, 100), nn.ReLU(), nn.Linear(100, 1))
# [...] pretend the 32-bit model is trained here
torch.backends.quantized.engine = engine
qmodel = quantize_dynamic(model, {nn.Linear}, dtype=torch.qint8)
X = torch.randn(3, 10)
y_pred = qmodel(X)  # float inputs and outputs, but quantized internally
```

The `quantize_dynamic()` function replaces each `Linear` layer with a `DynamicQuantizedLinear` layer, with int8 weights. This layer behaves just like a regular linear layer, with float inputs and outputs, but it quantizes its inputs on the fly (recomputing the zero points and scales for each batch), performs matrix multiplication using integers only (with 32-bit integer accumulators), and dequantizes the result so the next layer gets float inputs. Now let's look at static quantization.

Static quantization

This option is best for CNNs, and max inference speed. It's also compulsory for edge devices without a *floating-point unit* (FPU), as they don't support floats at all. Both the weights and activations are prepared for quantization ahead of time, for all layers. As we discussed earlier, weights are constant so they can be quantized once, while

activations require a calibration step to determine their typical range. The model is then converted to a fully quantized model. Here is how to implement it:

```
from torch.ao.quantization import get_default_qconfig, QuantStub, DeQuantStub

model = nn.Sequential(QuantStub(),
                      nn.Linear(10, 100), nn.ReLU(), nn.Linear(100, 1),
                      DeQuantStub())
# [...] pretend the 32-bit model is trained here
model.qconfig = get_default_qconfig(engine)
torch.ao.quantization.prepare(model, inplace=True)
for X_batch, _ in calibration_loader:
    model(X_batch)
torch.ao.quantization.convert(model, inplace=True)
```

Let's go through this code step by step:

- After the imports, we create our 32-bit model, but this time we add a `QuantStub` layer as the first layer, and a `DeQuantStub` layer as the last. Both layers are just passthrough for now.
- Next, the model can be trained normally (another option would be to take a pretrained model and place it between a `QuantStub` layer and a `DeQuantStub` layer).
- Next, we set the model's `qconfig` to the output of the `get_default_qconfig()` function: this function takes the name of the desired quantization engine and returns a `QConfig` object containing a default quantization configuration for this engine. It specifies the quantization data type (e.g., `torch.qint8`), the quantization scheme (e.g., symmetric linear quantization per tensor), and two functions that will observe the weights and activations to determine their ranges.
- Next we call the `torch.ao.quantization.prepare()` function: it uses the weight observer specified in the configuration to determine the weights range, which it immediately uses to compute the zero points and scales for the weights. Since we don't know what the input data looks like at this point, the function cannot compute the quantization parameters for the activations yet, so it inserts activation observers in the model itself: these are attached to the outputs of the `QuantStub` and `Linear` layers. The observer appended to the `QuantStub` layer is responsible for tracking the input range.
- Next, we take a representative sample of input batches (i.e., the kind the model will get in production), and we pass these batches through the model: this allows the activation observers to track the activations.
- Once we have given the model enough data, we finally call the `torch.ao.quantization.convert()` function, which removes the observers from the model and replaces the layers with quantized versions. The `QuantStub` layer is

replaced with a `Quantize` layer which will quantize the inputs. The `Linear` layers are replaced with `QuantizedLinear` layers. And the `DeQuantStub` layer is replaced with a `DeQuantize` layer which will dequantize the outputs.

There are a few observers to choose from: they can just keep track of the minimum and maximum values for each tensor (`MinMaxObserver`), or for each channel (`PerChannelMinMaxObserver`), or they can compute an exponential moving average of the min/max values, which reduces the impact of a few outliers. Finally, they can even record a histogram of the observed values (`HistogramObserver`), making it possible to find an optimal quantization range that minimizes the quantization error. That said, the default observers are usually fine.

We now have a model that we can use normally, with float inputs and outputs, but which works entirely with integers internally, making it lightweight and fast. To deploy it to mobile or embedded devices, there are many options to choose from (which are beyond the scope of this book), including:

- Use ExecuTorch, which is PyTorch's lightweight edge runtime
- Export the model to ONNX and run it with ONNX Runtime (cross-platform)
- Convert it to TFLite or TFLite Micro
- Compile it for the target device using TVM or microTVM

Moreover, the PyTorch team has released a separate library named *PyTorch-native Architecture Optimization* (TorchAO), designed to be a robust and extensible model optimization framework. Over time, many features in PyTorch's `torch.ao` package are expected to be migrated to—or superseded by—TorchAO. The library already includes advanced features such as 4-bit weight support and *per-block quantization*, in which each tensor is split into small blocks and each block is quantized independently, trading space for improved precision.

Post-training quantization (either dynamic or static) can shrink and speed up your models significantly, but it will also degrade their accuracy. This is particularly the case when quantizing down to 4 bits or less, and it's worse for static quantization than for dynamic quantization (which can at least adapt to each input batch independently). When the accuracy drop is unacceptable, you can try quantization-aware training, as we will discuss now.

Quantization-Aware Training (QAT)

QAT was introduced in a 2017 paper by Google researchers.[6] It rests upon a simple idea: why not introduce some fake quantization noise during training so the model can learn to cope with it? After training, we can then quantize the model for real, and it should remain fairly accurate. QAT also makes it possible to quantize more aggressively without losing too much accuracy, down to 4 bits, or even less. Sound promising? Let's see how it can be done.

To add fake quantization noise to weights, we can simply quantize them and immediately dequantize them. For example, a weight equal to 0.42 might be quantized to the 4-bit integer 7, and immediately dequantized back to 0.39: we've successfully introduced quantization noise, and it's precisely the quantization noise that we would get if the model were really quantized. This fake quantization operation can be executed at each training step, and it can also be applied to some of the activations (e.g., to each layer output).

However, there is one little problem: quantization involves rounding to the nearest integer, and the rounding operation has gradients equal to zero (or undefined at integer boundaries), so gradient descent cannot make any progress. Luckily, we can sidestep this issue by using the *straight-through estimator* (STE) trick: during the backward phase, we pretend that the fake quantization operation was just the identity function, so the gradients flow straight through it untouched. This works because the loss landscape is generally fairly smooth locally, so gradients are likely to be similar within a small region around the quantized value, including at the original value.

Implementing QAT in PyTorch is fairly straightforward:

```
from torch.ao.quantization import get_default_qat_qconfig

model = nn.Sequential(nn.Linear(10, 100), nn.ReLU(), nn.Linear(100, 1))
model.qconfig = get_default_qat_qconfig(engine)
torch.ao.quantization.prepare_qat(model, inplace=True)
train(model, optimizer, [...])  # train the model normally
torch.ao.quantization.convert(model.eval(), inplace=True)
```

After the import, we create our model, set its `qconfig` attribute to the default QAT configuration object for the chosen quantization engine, then we call the `prepare_qat()` function to add fake quantization operations to the model. This step also adds observers to determine the usual range of activation values. Next, we can train the model normally. Lastly, we switch the model to eval mode, and we call the `convert()` function to truly quantize it.

6 Benoit Jacob et al., "Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference", arXiv preprint arXiv:1712.05877 (2017)".

QAT doesn't have to be used during all of training: you can take a pretrained model and just fine-tune it for a few epochs using QAT, typically using a lower learning rate to avoid damaging the pretrained weights.

We've seen how to implement PTQ and QAT using PyTorch's `torch.ao` package. However, it's primarily designed for CPUs. What if you want to run an LLM on a GPU that doesn't quite have enough RAM? One option is to use the TorchAO library, which has growing GPU support. Another is to use the bitsandbytes library: let's discuss it now.

Quantizing LLMs Using the bitsandbytes Library

The bitsandbytes library (bnb), created by Tim Dettmers, is designed to make it easier to train and run large models on GPUs with limited VRAM. For this, it offers:

- Quantization tools, including 4-bit quantization, block-wise quantization, and more
- Memory-efficient versions of popular optimizers such as Adam or AdamW, that operate on 8-bit tensors
- Custom CUDA kernels written specifically for 8-bit or 4-bit quantized models, for maximum speed

The bitsandbytes library is designed for Nvidia GPUs. It also has some limited support for CPUs and AMD GPUs.

For example, let's see how to implement post-training static quantization down to 4 bits. If you are using Colab, you must first install the bitsandbytes library using `%pip install bitsandbytes`, then run this code:

```
from transformers import AutoModelForCausalLM, BitsAndBytesConfig

model_id = "TinyLlama/TinyLlama-1.1B-Chat-v1.0"
bnb_config = BitsAndBytesConfig(load_in_4bit=True, bnb_4bit_quant_type="nf4",
                                bnb_4bit_compute_dtype=torch.bfloat16)
model = AutoModelForCausalLM.from_pretrained(model_id, device_map="auto",
                                             quantization_config=bnb_config)
```

This code starts by importing the necessary classes from the Transformers library (introduced in Chapter 14), then it creates a `BitsAndBytesConfig` object, which I will explain shortly. Lastly, it downloads a pretrained model (in this case a 1.1 billion parameter version of Llama named TinyLlama, fine-tuned for chat), specifying the desired quantization configuration.

Under the hood, the Transformers library uses the bitsandbytes library to quantize the model weights down to 4 bits just as they are loaded into the GPU: no extra step is required. You can now use this model normally to generate text (see Chapter 15). During inference, whenever some weights are needed, they are dequantized on the fly to the type specified by the `bnb_4bit_compute_dtype` argument (`bfloat16` in this case), and the computations are performed in this higher precision. As soon as the dequantized weights are no longer needed, they are dropped, so memory usage remains low.

In this example, the `BitsAndBytesConfig` object specifies *4-bit Normal Float* (NF4) quantization using `bfloat16` for computations. NF4 is a nonlinear 4-bit scheme where each of the 16 possible integer values represents a specific float value between –1 and +1. Instead of being equally spaced (as in linear quantization), these values correspond to the quantiles of the normal distribution centered on zero: this means that they are closer together near zero. This improves accuracy because model weights tend to follow a normal distribution centered on zero, so having more precision near zero is helpful.

NF4 was introduced as part of QLoRA,[7] a technique that quantizes a frozen pretrained model with NF4, then uses LoRA adapters (see Chapter 17) for fine-tuning, along with activation checkpointing (see Chapter 12). This approach drastically reduces VRAM usage and compute: the authors managed to fine-tune a 65-billion parameter model using a single GPU with 48 GB of RAM, with only a small accuracy drop. Although activation checkpointing reduces VRAM usage overall, it can lead to memory spikes when processing batches with long sequences. To deal with such spikes, the QLoRA authors also introduced *paged optimizers* which take advantage of Nvidia unified memory: the CUDA driver automatically moves pages of data from GPU VRAM to CPU RAM whenever needed. Lastly, the authors also used *double quantization*, meaning that the quantization parameters themselves were quantized to save a bit more VRAM.

For more details on 4-bit quantization in the Hugging Face ecosystem, check out this great post by the QLoRA authors and other contributors.

7 Tim Dettmers et al., "QLORA: Efficient Finetuning of Quantized LLMs", arXiv preprint arXiv:2305.14314 (2023).

Using Pre-Quantized Models

Many popular pretrained models have already been quantized and published online, in particular on the Hugging Face Hub. For example, Tom Jobbins, better known by his Hugging Face username TheBloke, has published thousands of quantized models available at *https://huggingface.co/TheBloke*. Many of these models were quantized using one of the following modern methods:

Generative pre-training quantization (GPTQ)
: GPTQ[8] is a post-training quantization method, usually down to 4 bits, that treats quantization as an optimization problem. GPTQ goes through each layer, one by one, and optimizes the 4-bit weights to minimize the MSE between the layer's original outputs (i.e., using the full precision weights) and the approximate outputs (i.e., using the 4-bit weights). Once the optimal 4-bit weights are found, the approximate outputs are passed to the next layer, and the process is repeated all the way to the output layer. During inference, the weights are dequantized whenever they are needed. GPTQ only quantizes the weights, not the activations: this is called *weight-only quantization*, which is great for inference, not for training. You can use the Hugging Face Optimum library or the GPTQModel library to quantize your models with GPTQ.

Activation-aware Weight Quantization (AWQ)
: AWQ[9] aims to improve the accuracy of block-wise weight-only quantization (typically 4-bit quantization). The idea is to preserve the precision of the most important weights. To identify these so-called *salient weights*, the algorithm runs a calibration dataset through the model and finds the largest activations for each quantization group (e.g., the largest 0.1% to 1% activations), and the corresponding weights are considered salient. The authors observed that storing the salient weights using float16 greatly reduces the model's *perplexity* (a common metric equal to the exponential of the cross-entropy). However, mixing 4-bit and 16-bit weights is not hardware-friendly, so AWQ uses another method to preserve the salient weight's precision: they simply scale them up by some factor and add an operation in the model to scale down the corresponding activations (but this operation can generally be fused into the previous operation). Rather than using a fixed scaling factor, AWQ performs a search for the optimal factor, leading to the lowest quantization error. To implement AWQ, you can use the Hugging Face Optimum library.

8 Elias Frantar et al., "GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers", arXiv preprint arXiv:2210.17323 (2022).

9 Ji Lin et al., "AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration", arXiv preprint arXiv:2306.00978 (2023).

Llama.cpp quantization using the GPT-Generated Unified Format (GGUF)

GGUF is a binary file format designed to store LLMs efficiently. It was introduced by Georgi Gerganov, the creator of llama.cpp, and it supersedes previous file formats such as GGML, GGMF, and GGJT. A GGUF file includes the weights, the tokenizer, special tokens, the model architecture, the vocabulary size, and other metadata. Llama.cpp offers quantizers (e.g., using the `quantize` tool) to convert the model weights to one of GGUF's supported quantized formats, such as Q4_K_M. Q4 stands for 4-bit quantization, K stands for per-block quantization (typically 32 or 64 weights per block depending on the chosen format), and M means medium size and precision for this quantization level (other options are S = Small and L = Large). There are also more recent and efficient quantization options such as Importance-aware Quantization (IQ), which uses various techniques to improve accuracy (e.g., nonlinear quantization), and Ternary Quantization (TQ).

On the Hugging Face Hub, every repository is backed by Git, so it has branches and commits. When you call `from_pretrained()`, the model is fetched from the default branch, which is almost always `main`. But quantized models are often placed in a different branch. When calling `from_pretrained()`, you can choose a branch, a tag, or even a commit hash, by using the `revision` argument. Check the model card for the list of available files and versions. For GGUF models, you must specify the filename using the `gguf_file` argument.

In conclusion, reduced precision, mixed-precision training, and quantization are arguably the most important tools to allow large models to run on limited hardware. But there are many more, including the following:

- You could tweak the model's architecture before training, by reducing the number of layers, or the number of neurons per layer, or by sharing weights across layers (e.g., as in the ALBERT model, introduced in Chapter 15).
- If you have a large trained model, you can shrink it by removing some of its weights, for example the ones with the smallest magnitude, or the ones with the smallest effect on the loss. You can also remove whole channels, layers, or attention heads. This is called *model pruning*, and you can implement it using the `torch.nn.utils.prune` module, or the Hugging Face Optimum library.
- As we saw in Chapter 15, you can also use a large trained model as a teacher to train a smaller model: this is called distillation.
- A trained model can also be shrunk by fusing some of its layers, removing redundancy. For example, a batch-norm layer (introduced in Chapter 11) performs a linear operation, so if it comes immediately after a linear layer, you can fuse

both layers into a single linear layer. Similarly, you can fuse a convolutional layer followed by a batch-norm layer into a single convolutional layer. This only works after training, since the batch-norm layer must compute running averages during training. You can implement layer fusion with the `torch.quantization.fuse_modules()` function, or with the Hugging Face Optimum library. In any case, make sure to fuse layers *before* quantizing your model: less layers means less quantization noise.

- You can use low-rank approximations, where a large matrix is replaced by the product of two smaller ones. For example, replace a large linear layer such as `Linear(10_000, 20_000)` with two linear layers `Linear(10_000, 100)` and `Linear(100, 20_000)`. This reduces the number of parameters from about 200 million down to just three million, and also drastically reduces computations. The intermediate dimensionality (100 in this example) is a hyperparameter you can tune to balance accuracy and model size. This technique can be performed after training by factorizing the weight matrix using SVD (see the notebook for an example).

Give these techniques a try: shrink the models!

Chapter 17 and Appendices C, D, and E are available online at *https://homl.info*.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

About the Author

Aurélien Geron is a machine learning consultant and lecturer. A former Googler, he led YouTube's video classification team. He's been a founder of and CTO at a few different companies: Wifirst, a leading wireless ISP in France; Polyconseil, a consulting firm focused on telecoms, media, and strategy; and Geron AI, a consulting firm focused on machine learning.

Before all that Aurélien worked as an engineer in a variety of domains: finance (JP Morgan and Société Générale), defense (Canada's DOD), and healthcare (blood transfusion). He also published a few technical books (on C++, WiFi, and internet architectures) and lectured in several universities.

A few fun facts: he taught his three children to count in binary with their fingers (up to 1,023), he studied microbiology and evolutionary genetics before going into software engineering, and his parachute didn't open on the second jump.

Colophon

The animal on the cover of *Hands-On Machine Learning with Scikit-Learn and PyTorch* is a Bornean orangutan (*Pongo pygmaeus*). Orangutans belong to the family Hominidae, or the great apes, along with gorillas, chimpanzees, and humans.

Native to the rainforests of Borneo, these orangutans are arboreal but will travel large distances on the ground in search of food. Their diet includes fruit, seeds, flowers, vines, and bird eggs. They build elaborate nests in trees to sleep in at night. Juveniles start to practice nest-building at 6 months old, and are proficient by the age of 3.

Bornean orangutans are highly sexually dimorphic. Males are larger than females and they have larger cheek pads (known as flanges), larger canines, and more facial hair. Males average 4 feet, 9 inches tall, and weigh 165 pounds, while females average 3 feet, 7 inches tall and weigh 85 pounds. Orangutans have long arms for climbing in trees. They have gray skin and a shaggy, reddish-brown coat.

Bornean orangutans are more solitary than other orangutan species. They aren't considered territorial, but males will exhibit threatening behavior when meeting other males. They only socialize with females to mate. Females reach maturity at around 15 years old. They give birth to a single infant every 8 or 9 years. Orangutans have the longest infant development period of all the great apes. Juveniles are weaned at about 4 years old and become completely independent around 7 years old. Their lifespan is 35–45 years.

There are about 105,000 Bornean orangutans living in the wild. Deforestation and hunting have made them increasingly endangered. Although hunting the orangutan is illegal, perpetrators are rarely prosecuted. Their distribution throughout Borneo is

now patchy due to habitat destruction. Additionally, as rainfall has decreased on the island, food has become less available, which has impacted the orangutan's birthrates. They have an IUCN status of Critically Endangered. Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by José Marzan Jr., based on an antique line engraving by Friedrich Specht (1888). The series design is by Edie Freedman, Ellie Volckhausen, and Karen Montgomery. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.