

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.

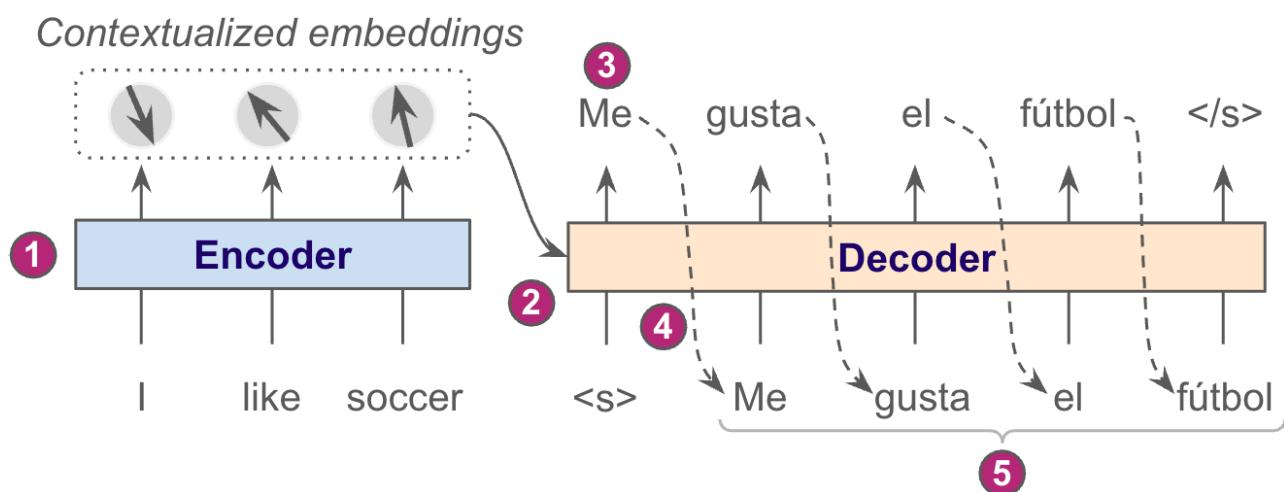


Figure 15-1. Using the Transformer model for English-to-Spanish translation

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. Although it was initially designed for NMT, 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 pre-trained on a large corpus of text, its ability to generate text was unprecedented, it could autocomplete 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 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 since 2018.³ Note that the vertical axis is in *billions* of parameters, and it uses a logarithmic scale.

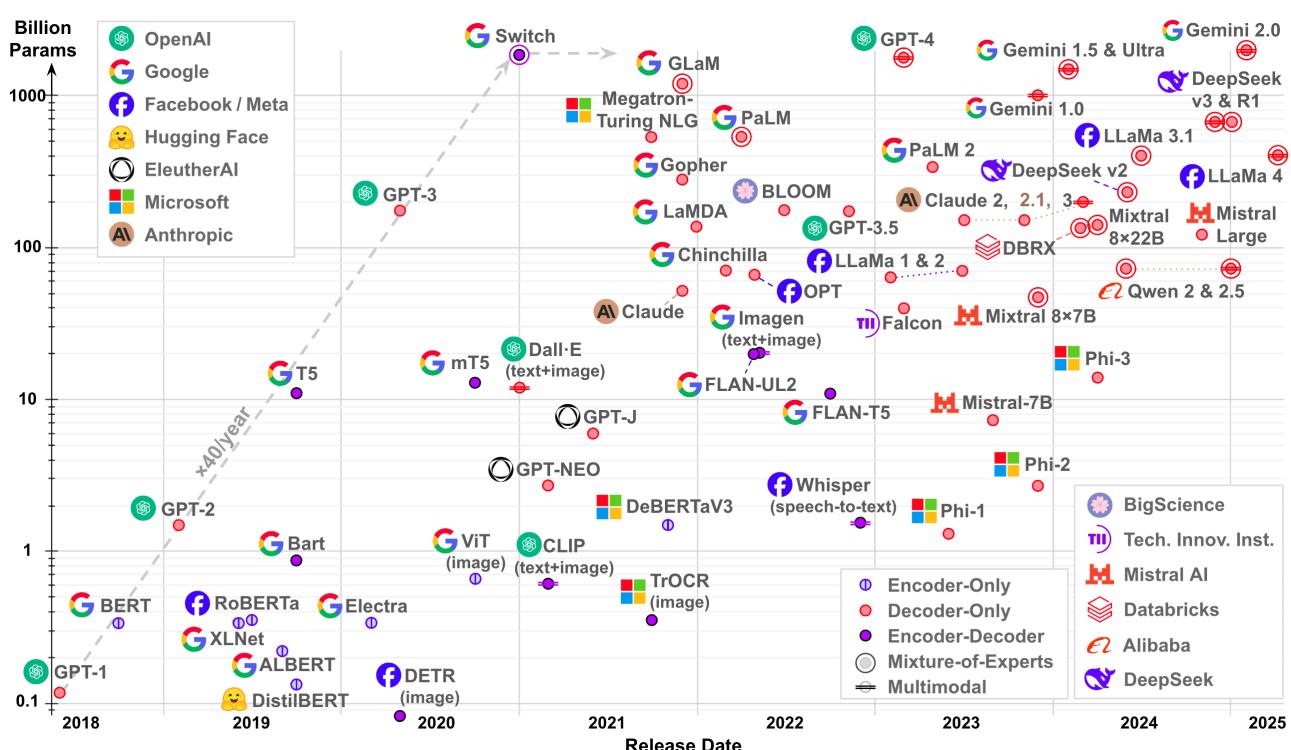


Figure 15-2 Some of the most influential transformers released since 2018

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.

NOTE

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:

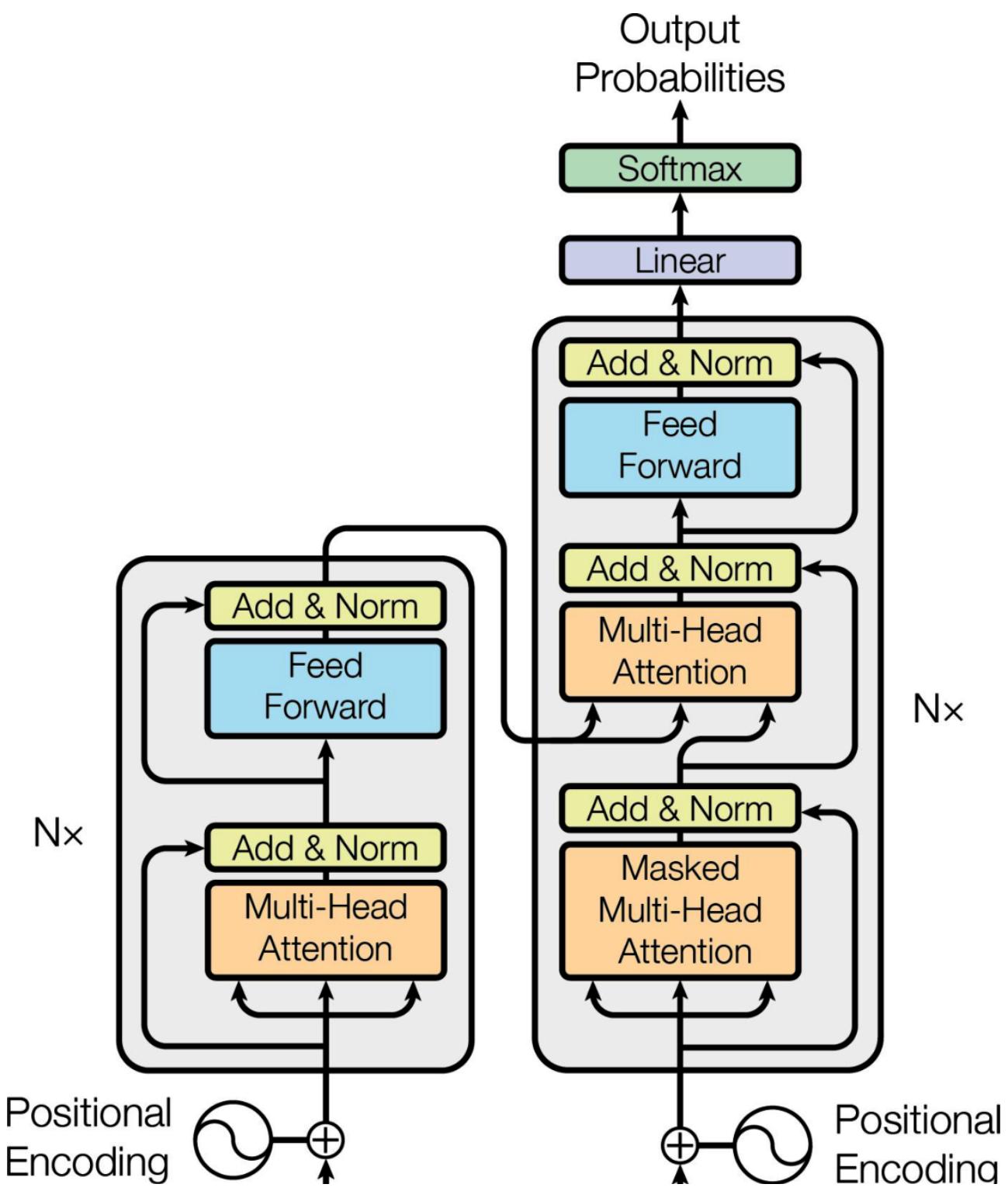
- We will start by opening up the original Transformer architecture and inspecting its components to fully understand how everything works.
- Then we will build and train a transformer from scratch for English-to-Spanish translation.
- After that, we will look into encoder-only models like BERT, learn 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 quite well.
- 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](#) also discusses 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.



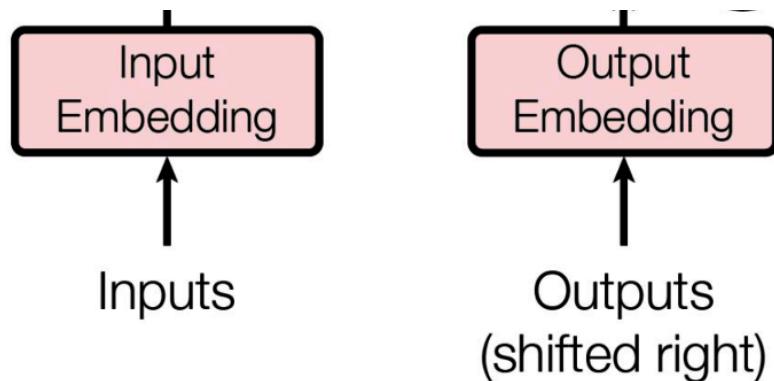


Figure 15-3. The original 2017 transformer architecture⁴

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.

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. Obviously, the order of words matters, which is why we must somehow give positional information to the Transformer: adding positional encodings to the token representations is a good way to achieve this.

NOTE

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 a `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__()
        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)])
```

NOTE

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). 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 [Link to Come].

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. Its vectorized equation is presented in [Equation 15-1](#).

Equation 15-1. Scaled dot-product attention

In this equation:

- \mathbf{Q} is a matrix representing a *query* (e.g., an English or Spanish sequence, depending on the attention layer). Its shape is $[L_q, D_q]$, 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).
- \mathbf{K} is a matrix representing a key. Its shape is $[L_k, D_k]$, 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 .
- \mathbf{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 $\mathbf{Q} \mathbf{K}^T$ 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 [Chapter 16](#) and [Chapter 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 \mathbf{Q} , \mathbf{K} , and \mathbf{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](#).



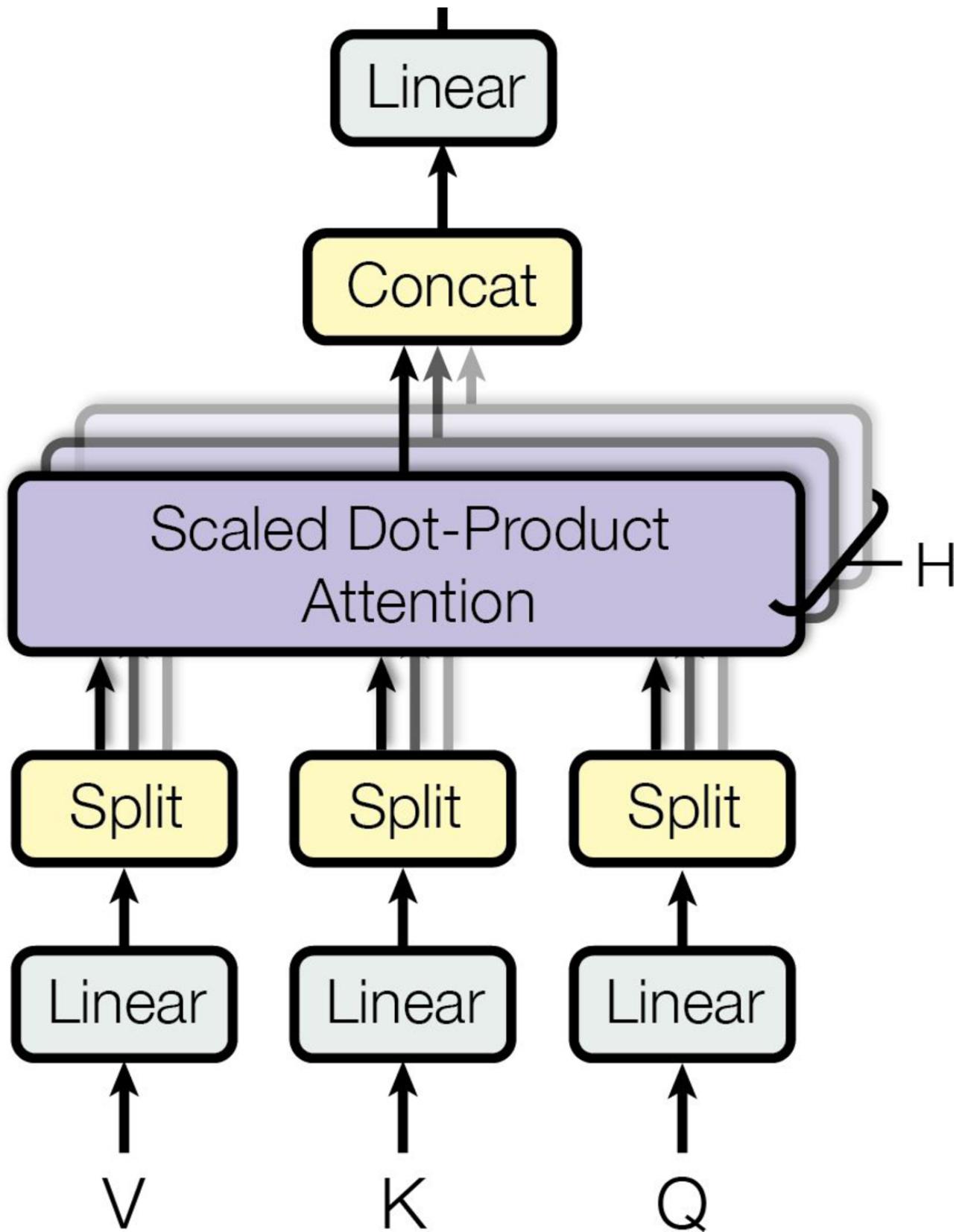


Figure 15-4. Multi-head attention layer architecture⁶

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 dif-

ferent 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.

This is why the multi-head attention 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.

WARNING

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](#).

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 = self.dropout(scores.softmax(dim=-1))  # (B, H, Lq, Lk)
        Z = 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 x D)
        return (self.out_proj(Z), weights)  # (B, Lq, H x 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).

TIP

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 [Chapter 16](#) and [Chapter 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 `nn.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 will use our `PositionalEncoding` 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 = PositionalEncoding(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.

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.

WARNING

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.

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 1024 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

parts 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

segment #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.

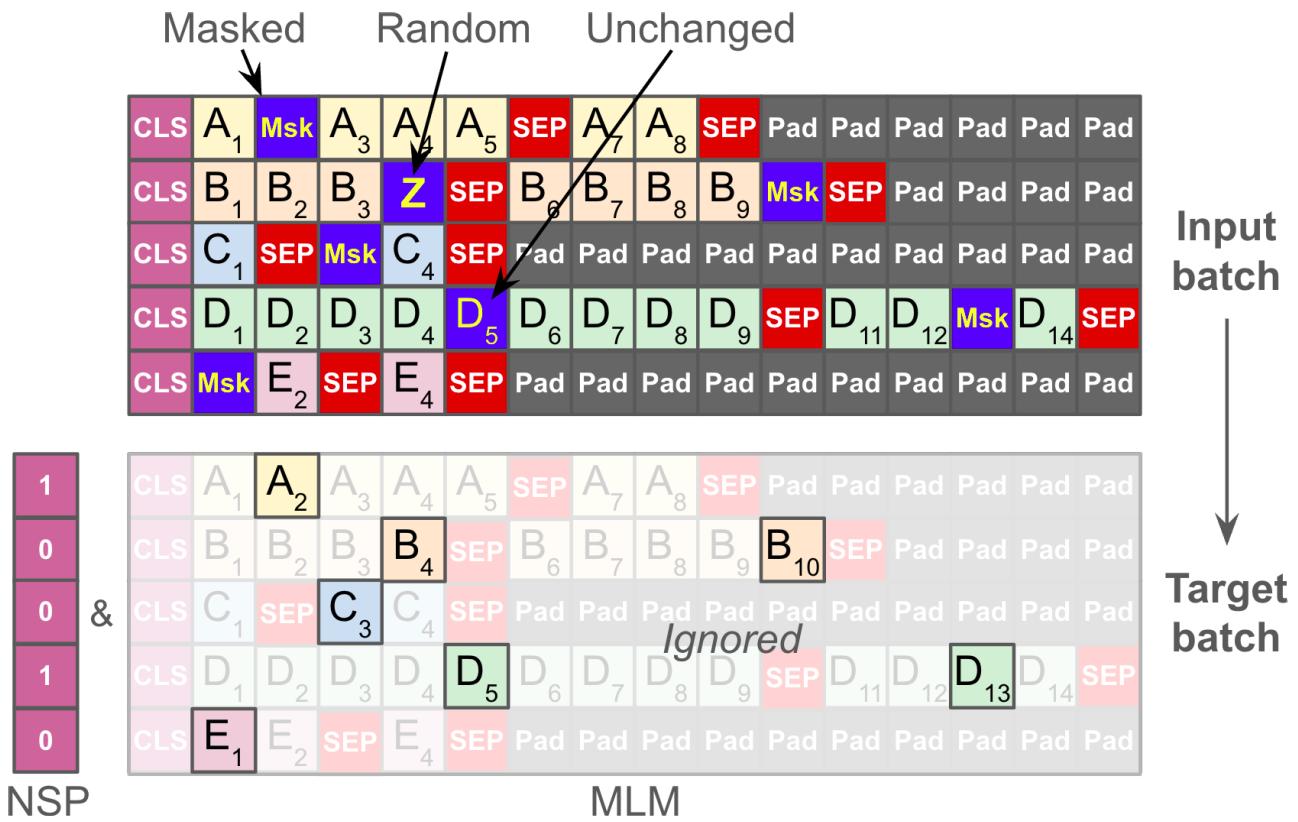


Figure 15-5. Input and target during BERT pretraining, using MLM and NSP

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 a

`nn.TransformerEncoderLayer` with `norm_first=True` to respect BERT’s architecture), then preprocess your dataset according to the MLM algorithm, and train your model. 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=2, 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': 0.044691428542137146,
 'token': 1996,
 'token_str': 'the',
```

```
'sequence': 'the capital of the is rome.'
```

What? Rome is not the capital of “the”! 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](#)).

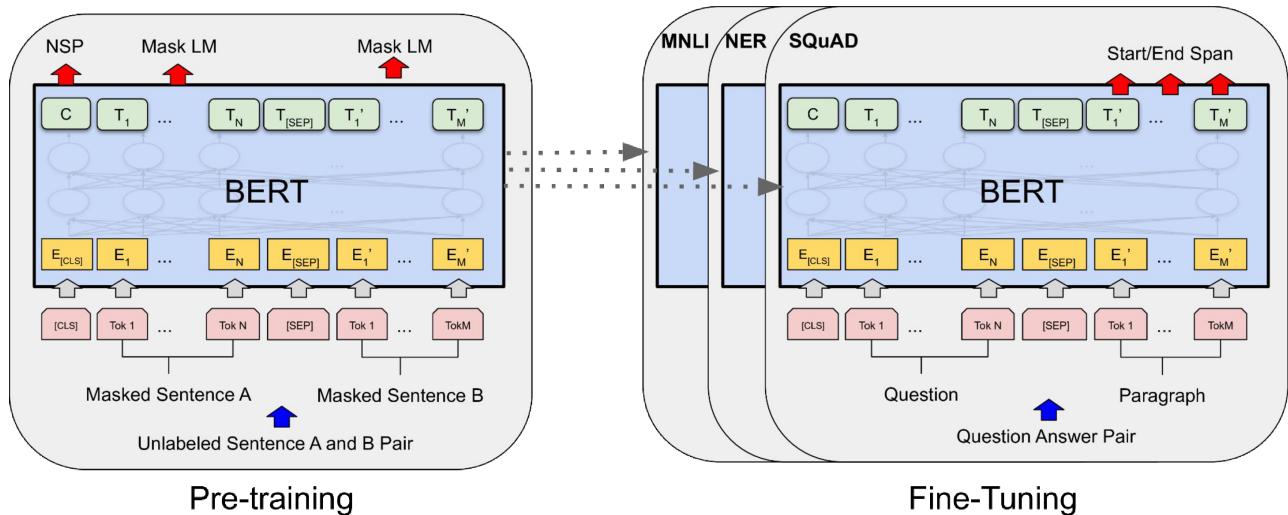


Figure 15-6. BERT pre-training (left) and fine-tuning process (right)⁸

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.

TIP

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

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.

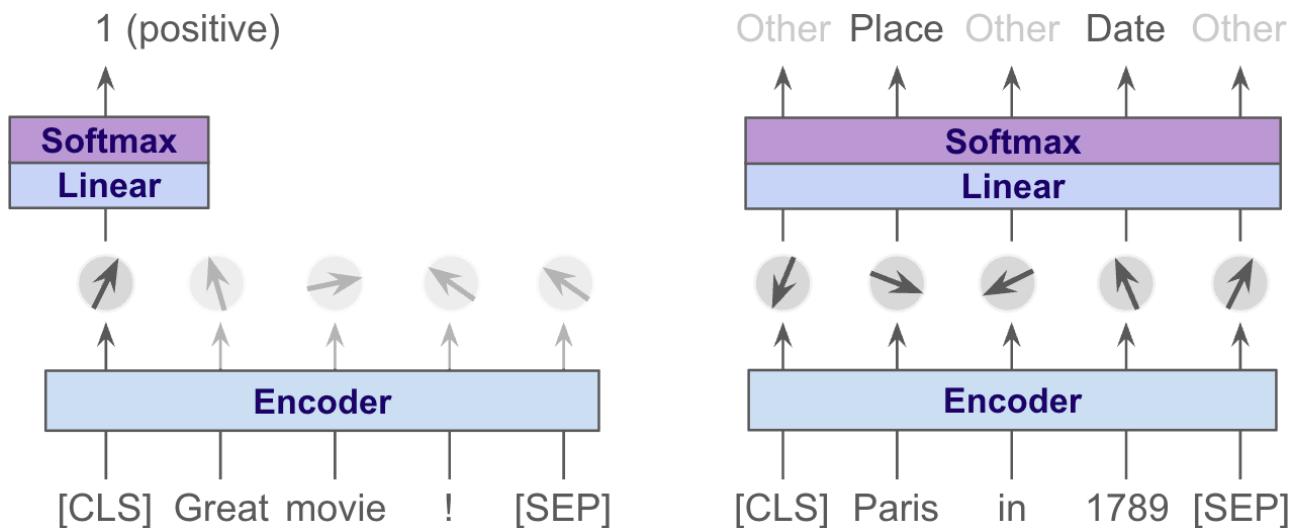
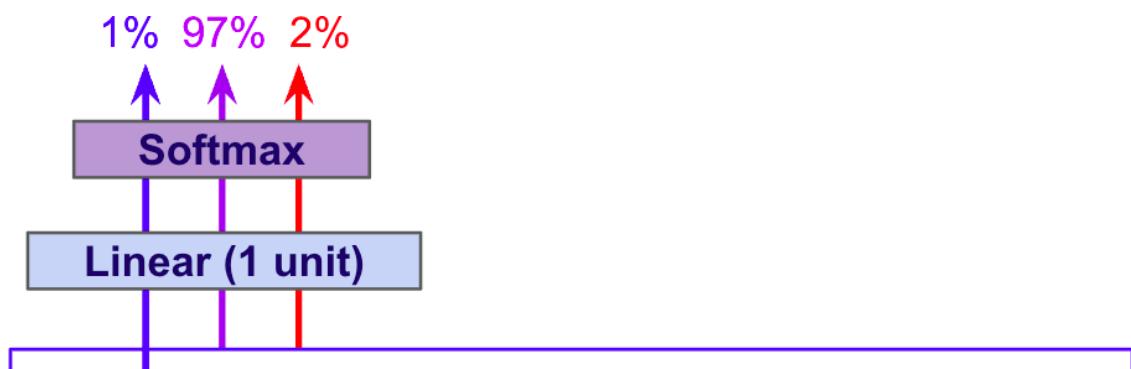


Figure 15-7. Fine-tuning BERT for sentence classification such as sentiment analysis (left) or for token classification such as NER (right)

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).



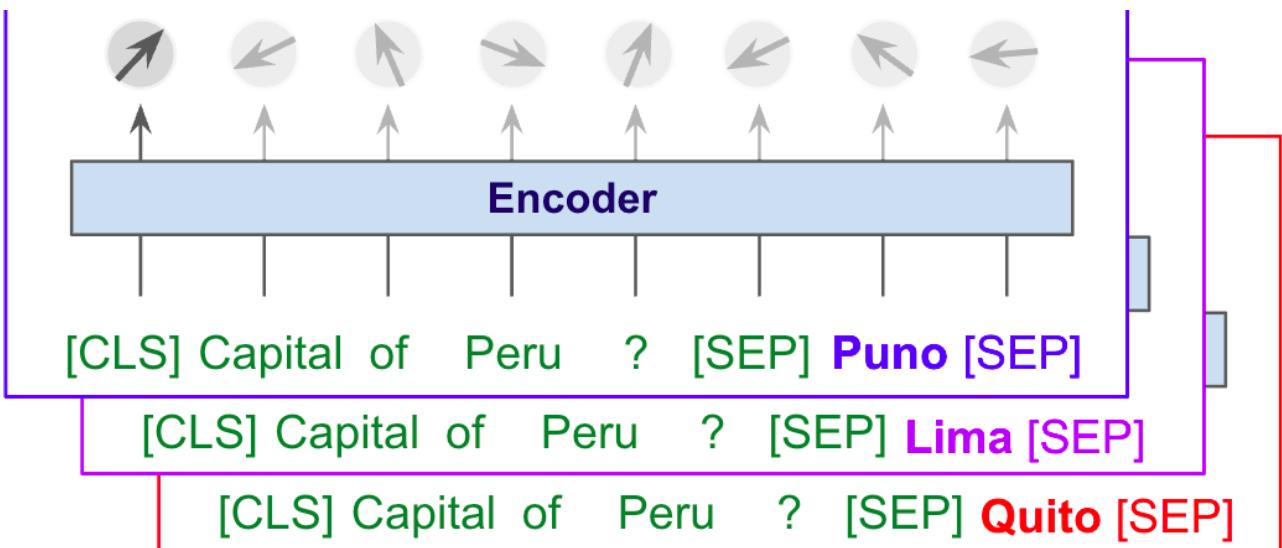


Figure 15-8. Using an encoder-only model to answer multiple-choice questions

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.

TIP

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, for example:

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.

TIP

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.

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.

NOTE

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

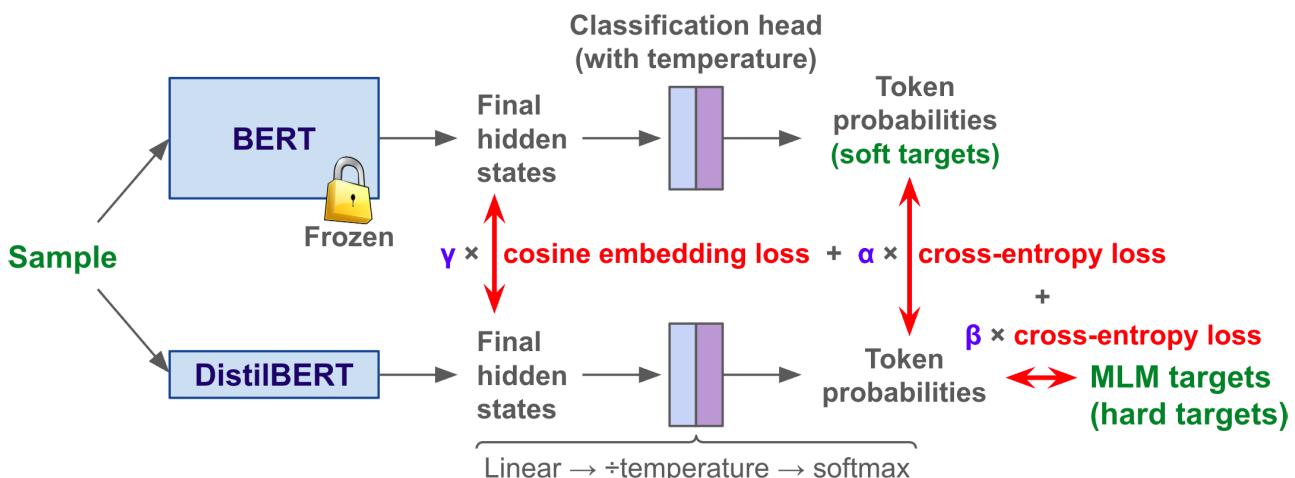


Figure 15-9. DistilBERT pretraining using a weighted sum of two distillation losses and the MLM loss

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 ($\sim 30,000 \times 128$), and the linear layer has about 0.13M parameters ($128 \times 1,024$), so the total is less than 4M parameters, down from over 30M: nice!

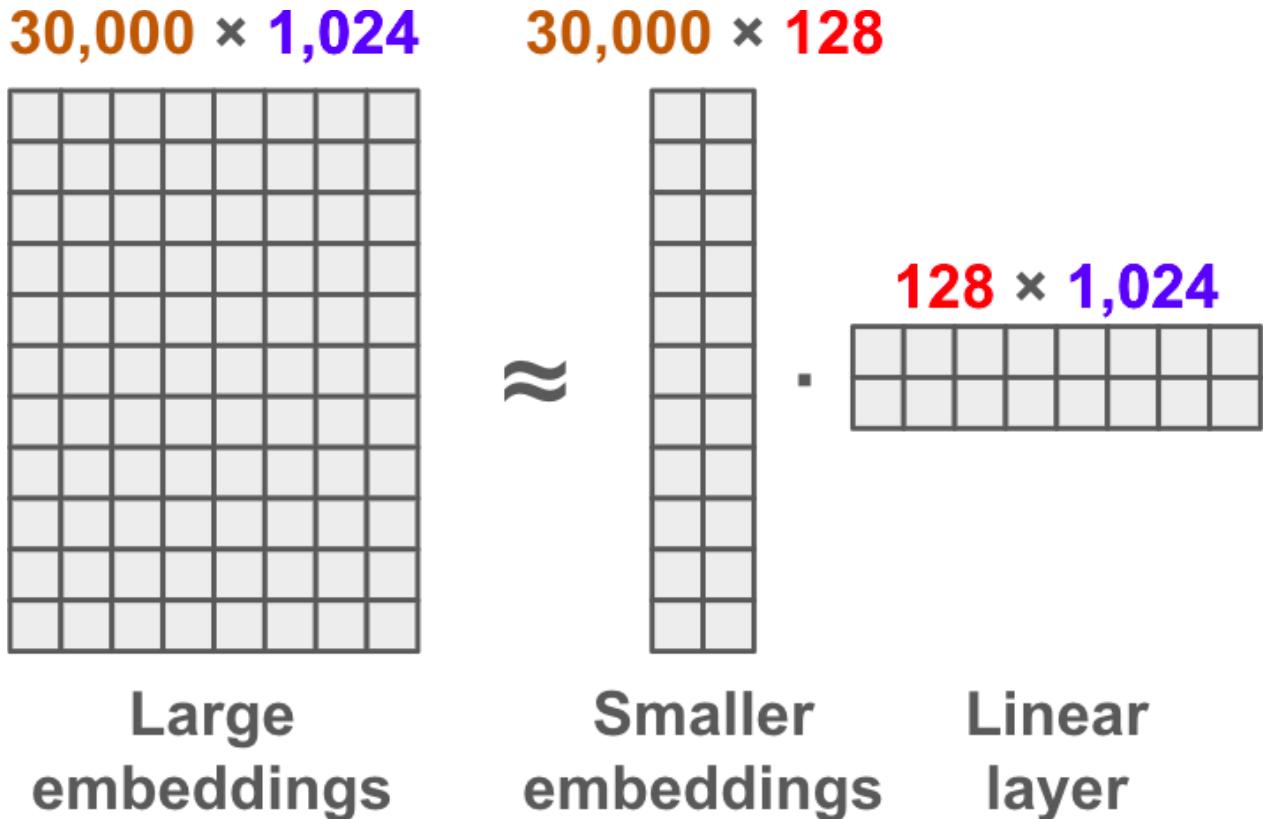


Figure 15-10. An excessively large embedding matrix can be replaced with the product of two smaller matrices. This can be implemented using smaller embeddings and projecting them to higher dimensions using a linear layer.

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.

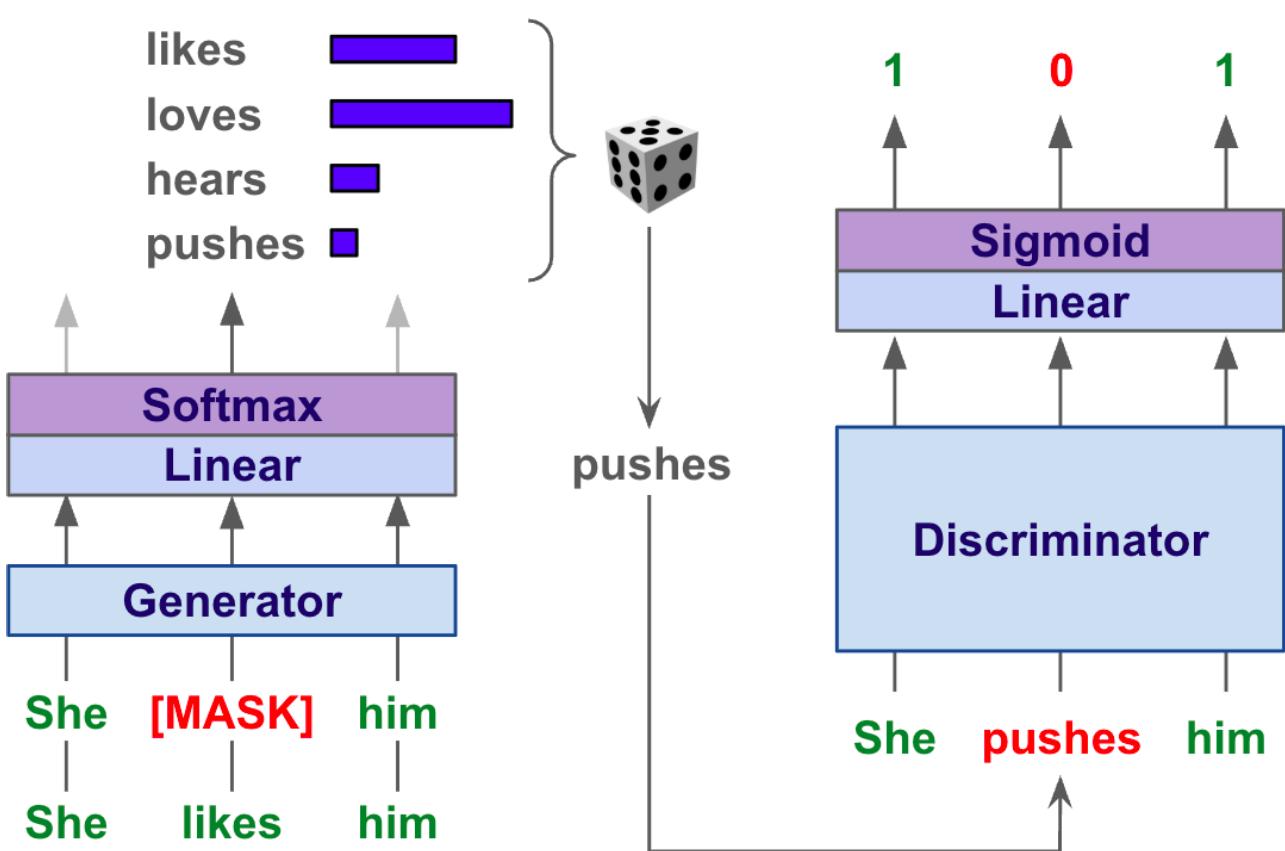


Figure 15-11. Replaced token detection (RTD)

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.

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 vs 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 earlier 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.

TIP

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](#)).

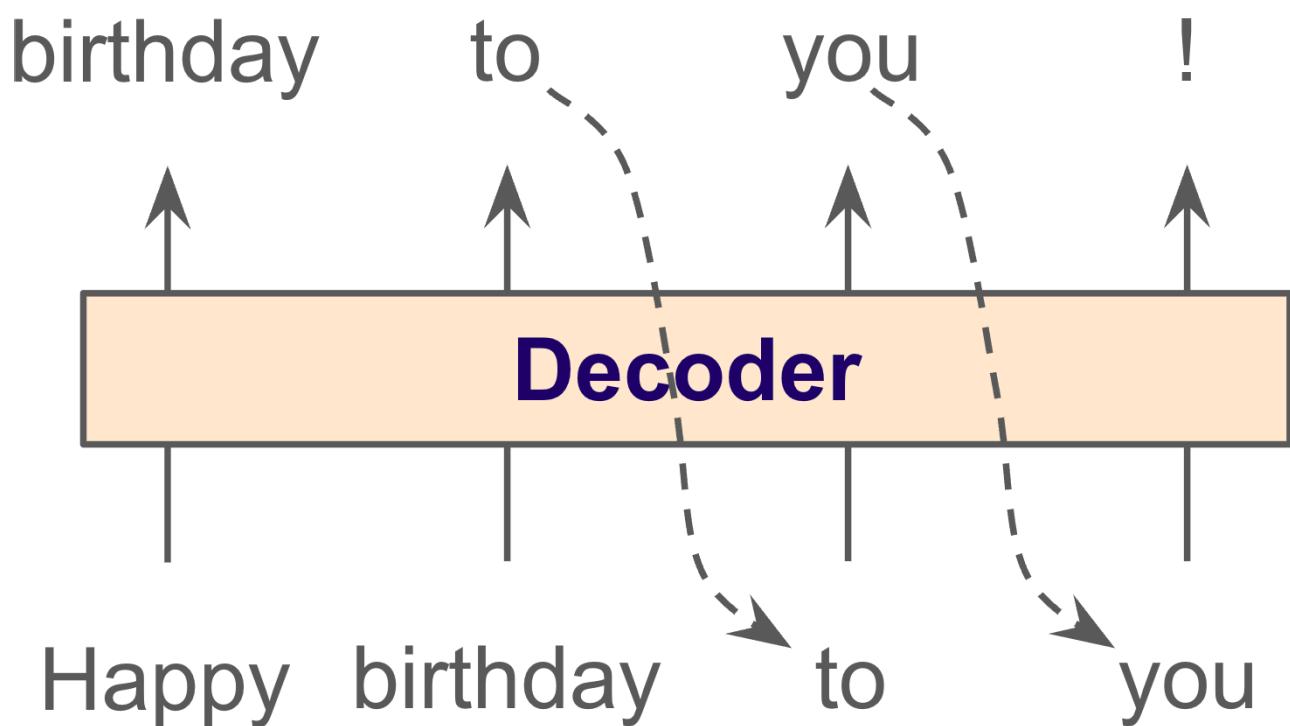


Figure 15-12. Generating text one token at a time using a decoder-only model like GPT

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

WARNING

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:

1. 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).
2. 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.

WARNING

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.

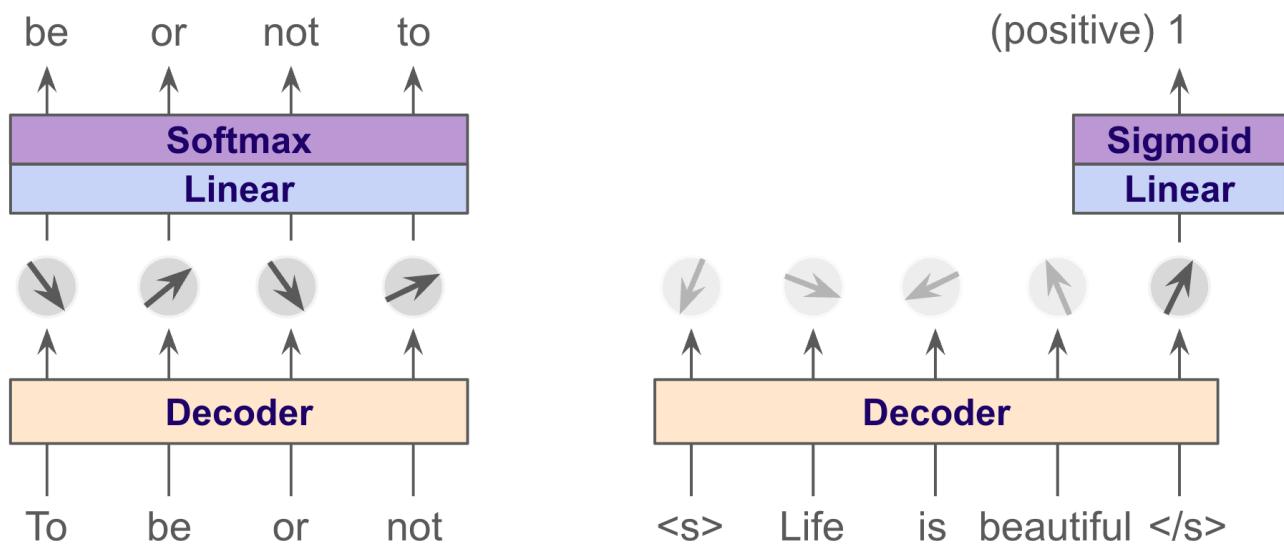


Figure 15-13. Pretraining GPT-1 using next token prediction (NTP, left) and fine-tuning it for classification (right)

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.
- 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?

NOTE

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.

NOTE

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", torch_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 `torch_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 pre-trained 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.

NOTE

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:
https://www.facebook.com/videos/#...\n\nYou can listen to the audio of an
interview with Paul Hines as he talks about the development of his podcast
about animals. It's a full length version of his show on PBS where"
```

Well, that's certainly less repetitive, but it's not exactly what we were hoping for: it looks like a random web page—which is exactly what GPT-2 was pretrained on, so it actually makes sense! 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 at 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 Villemainne, 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.

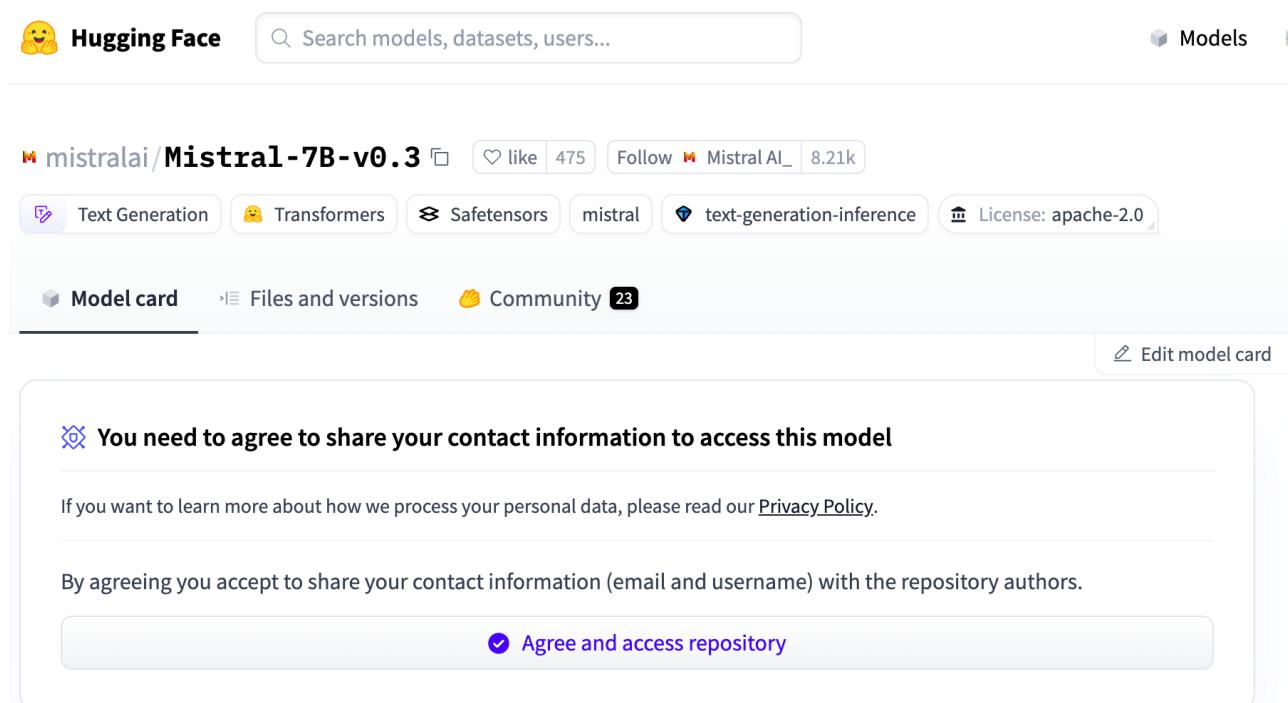


Figure 15-14. The Mistral-7B-v0.3 model on the Hugging Face Hub requires agreeing to share your identity with the model authors

Next, you will 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.

WARNING

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.

The screenshot shows the 'Create new Access Token' page. It includes fields for 'Token type' (set to 'Fine-grained'), 'Token name' ('hf-read-mistral'), and sections for 'User permissions' and 'Repositories permissions'. The 'User permissions' section lists three options: 'Read access to contents of all repos under your personal namespace' (unchecked), 'Read access to contents of all public gated repos you can access' (checked), and 'Write access to contents/settings of all repos under your personal namespace' (unchecked). The 'Repositories permissions' section shows a search bar ('Search for repos') with 'mistralai/Mistral-7B-v0.3' selected. Under 'Repositories permissions', there are three checkboxes: 'Read access to contents of selected repos' (checked), 'Interact with discussions / Open pull requests on selected repos' (unchecked), and 'Write access to contents/settings of selected repos' (unchecked). A 'Create token' button is at the bottom right.

< **Create new Access Token** [...]

Token type

Fine-grained Read Write

! This cannot be changed after token creation.

Token name

hf-read-mistral

User permissions (ageron)

Repositories

Read access to contents of all repos under your personal namespace

Read access to contents of all public gated repos you can access

Write access to contents/settings of all repos under your personal namespace

Search for repos

mistralai/Mistral-7B-v0.3

Read access to contents of selected repos

Interact with discussions / Open pull requests on selected repos

Write access to contents/settings of selected repos

[...]

Create token

Figure 15-15. Creating a Hugging Face access token

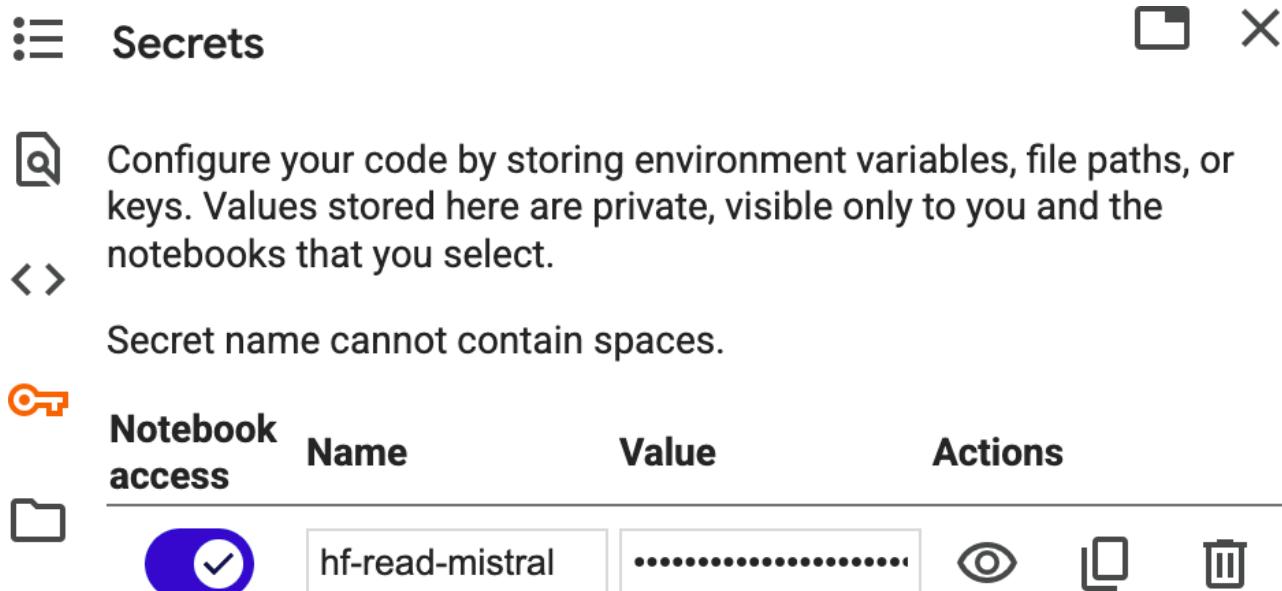
OK, let’s go back to Colab now. The last step before downloading the model is to get your notebook to log in to the Hugging Face Hub using the access token that you just created. However, hardcoding access tokens directly in your code is highly insecure: if anyone can read your notebook, they will know 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.

WARNING

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.



Notebook access	Name	Value	Actions
<input checked="" type="checkbox"/>	hf-read-mistral	

+ Add new secret

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", torch_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.  
"""
```

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)
>>> 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 an iconic landmark in Paris and one of the most
recognizable structures in the world. It was built in 1889 and stands at a
height of 324 meters. Visitors can take an elevator to the top of the tower
for panoramic views of the city.'
>>> bob.chat("And Rome?")
'Rome is a city steeped in history and culture. It is home to many famous
landmarks, including the Colosseum, the Pantheon, and the Trevi Fountain.
Visitors can explore the city's ancient ruins, visit museums, and sample
delicious Italian cuisine.'
```

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. Then, you can bake them in the oven for about 10 minutes”. 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. Below 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 multi-step 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 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 *failbreak*

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.

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.

NOTE

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.

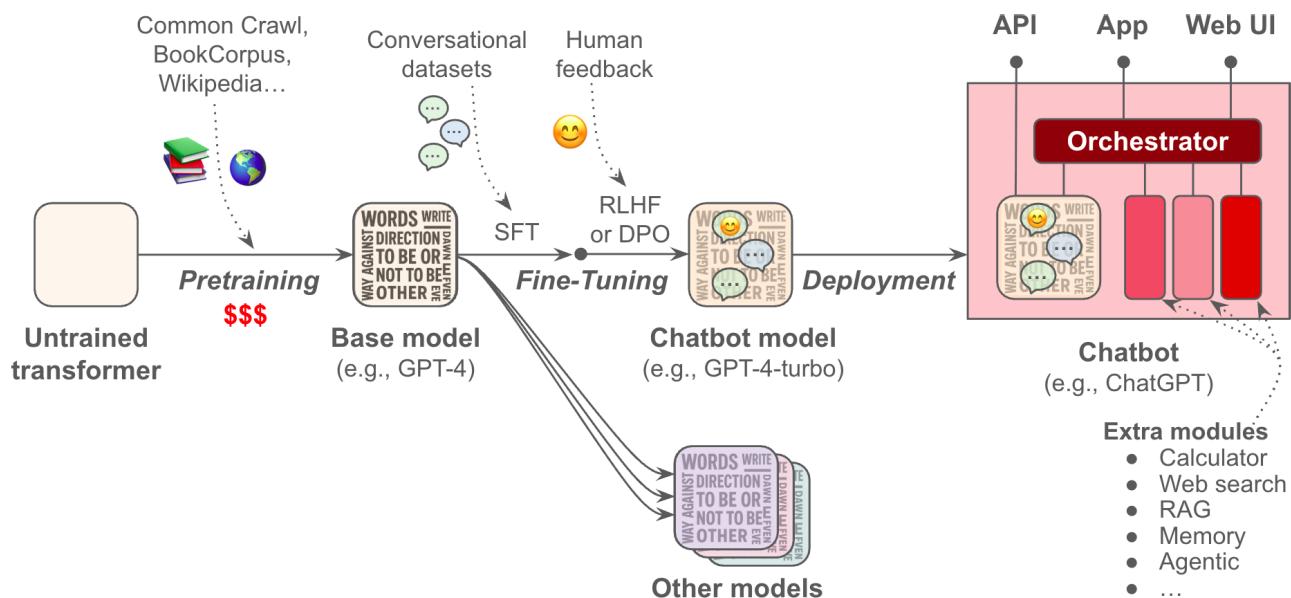


Figure 15-17. How to build a chatbot: pretraining, two-step fine-tuning, and deployment

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.

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

In this equation:

- $J(\theta)$ is the DPO loss for an instance (x, y_c, y_r) , given the current model parameters θ and a frozen reference model.
- x is the prompt, y_c is the chosen answer, and y_r is the rejected answer.
- $\sigma(\cdot)$ is the usual sigmoid function:
- $\log p_\theta(y | x)$ is our model's estimated log probability for answer y (either y_c or y_r), given the prompt x .
- $\log p_{\text{ref}}(y | x)$ is the reference model's estimated log probability for answer y given 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.

TIP

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(\cdot))`.

To compute $\log p(y | x)$, where p is either p_θ or p_{ref} , and y is either y_c or y_r , we start by concatenating x and 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])]
['3.32%', '0.02%', '51.95%', '0.38%', '32.03%', '11.23%', '99.61%']
>>> [f"{p.item():.2%}" for p in torch.exp(next_token_log_probas[1])]
['0.00%', '3.32%', '0.02%', '51.95%', '0.38%', '31.64%', '0.00%']
```

The first estimated probability is for the token “The” (3.32%), then “capital” (0.02%), and so on. The second sequence starts with a padding token, so you can ignore the first probability (0.00%). The estimated probabilities are the same in both sequences for the prompt tokens, but they differ for the answer tokens: 11.23% 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.²⁶

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 11.23%:

```
>>> 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
0.1123046875
```

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 $p(\mathbf{y} | \mathbf{x})$ with log $p(\mathbf{xy})$ in [Equation 15-2](#) (for both p_θ and p_{ref} , and for both \mathbf{y}_c and \mathbf{y}_r). This is because $\log p(\mathbf{xy}) = \log p(\mathbf{x}) + \log p(\mathbf{y} | \mathbf{x})$, 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([-21.3750, -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)
    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 multturn 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)
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.”

WARNING

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 $525.6 * 315 / 3942$?
System: 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 fine-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] $525.6 * 315 / 3942$ [/cal-

culator_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.

TIP

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 1982, a Soviet lieutenant colonel named Stanislaw

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 file system 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.”).

TIP

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, 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.

Finally, 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).

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

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.

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.

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

- 5** Queries, keys, and values were introduced in [Chapter 14](#) when we discussed dot-product attention.
- 6** This is the righthand part of figure 2 from “Attention Is All You Need”, reproduced with the kind authorization of the authors.
- 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).
- 8** This is Figure 1 from the BERT paper, reproduced with the kind authorization of the authors.
- 9** Nils Reimers, Iryna Gurevych, “Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks”, arXiv preprint arXiv:1908.10084 (2019).
- 10** Geoffrey Hinton et al., “Distilling the Knowledge in a Neural Network”, arXiv preprint arXiv:1503.02531 (2015).
- 11** Alec Radford et al., [“Improving Language Understanding by Generative Pre-Training”](#) (2018).
- 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.
- 14** Tom B. Brown et al., “Language Models are Few-Shot Learners”, arXiv preprint arXiv:2005.14165 (2020).
- 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.
- 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).
- 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).
- 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).
- ²⁶ There’s a slight difference for the token “is”, which I assume is due to the accumulation of floating point errors in this large model.
- ²⁷ Colin Raffel et al., “Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer”, arXiv preprint arXiv:1910.10683 (2019).