CHAPTER 3
Transformer Anatomy

In Chapter 2, we saw what it takes to fine-tune and evaluate a transformer. Now let's take a look at how they work under the hood. In this chapter we'll explore the main building blocks of transformer models and how to implement them using PyTorch. We'll also provide guidance on how to do the same in TensorFlow. We'll first focus on building the attention mechanism, and then add the bits and pieces necessary to make a transformer encoder work. We'll also have a brief look at the architectural differences between the encoder and decoder modules. By the end of this chapter you will be able to implement a simple transformer model yourself!

While a deep technical understanding of the Transformer architecture is generally not necessary to use 🤗 Transformers and fine-tune models for your use case, it can be helpful for comprehending and navigating the limitations of transformers and using them in new domains.

This chapter also introduces a taxonomy of transformers to help you understand the zoo of models that have emerged in recent years. Before diving into the code, let's start with an overview of the original architecture that kick-started the transformer revolution.

The Transformer Architecture

As we saw in Chapter 1, the original Transformer is based on the *encoder-decoder* architecture that is widely used for tasks like machine translation, where a sequence of words is translated from one language to another. This architecture consists of two components:

Encoder

Converts an input sequence of tokens into a sequence of embedding vectors, often called the *hidden state* or *context*

Decoder
: Uses the encoder's hidden state to iteratively generate an output sequence of tokens, one token at a time

As illustrated in Figure 3-1, the encoder and decoder are themselves composed of several building blocks.

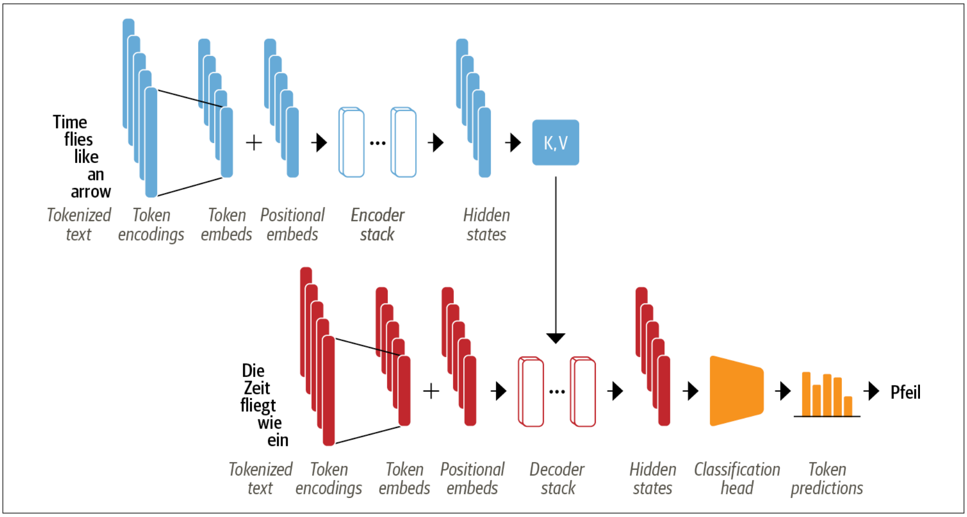


Figure 3-1. Encoder-decoder architecture of the transformer, with the encoder shown in the upper half of the figure and the decoder in the lower half

We'll look at each of the components in detail shortly, but we can already see a few things in Figure 3-1 that characterize the Transformer architecture:

- The input text is tokenized and converted to *token embeddings* using the techniques we encountered in Chapter 2. Since the attention mechanism is not aware of the relative positions of the tokens, we need a way to inject some information about token positions into the input to model the sequential nature of text. The token embeddings are thus combined with *positional embeddings* that contain positional information for each token.
- The encoder is composed of a stack of *encoder layers* or "blocks," which is analogous to stacking convolutional layers in computer vision. The same is true of the decoder, which has its own stack of *decoder layers*.
- The encoder's output is fed to each decoder layer, and the decoder then generates a prediction for the most probable next token in the sequence. The output of this step is then fed back into the decoder to generate the next token, and so on until a special end-of-sequence (EOS) token is reached. In the example from Figure 3-1, imagine the decoder has already predicted "Die" and "Zeit". Now it

gets these two as an input as well as all the encoder's outputs to predict the next token, "fliegt". In the next step the decoder gets "fliegt" as an additional input. We repeat the process until the decoder predicts the EOS token or we reached a maximum length.

The Transformer architecture was originally designed for sequence-to-sequence tasks like machine translation, but both the encoder and decoder blocks were soon adapted as standalone models. Although there are hundreds of different transformer models, most of them belong to one of three types:

Encoder-only
These models convert an input sequence of text into a rich numerical representation that is well suited for tasks like text classification or named entity recognition. BERT and its variants, like RoBERTa and DistilBERT, belong to this class of architectures. The representation computed for a given token in this architecture depends both on the left (before the token) and the right (after the token) contexts. This is often called *bidirectional attention*.

Decoder-only
Given a prompt of text like "Thanks for lunch, I had a…" these models will auto-complete the sequence by iteratively predicting the most probable next word. The family of GPT models belong to this class. The representation computed for a given token in this architecture depends only on the left context. This is often called *causal* or *autoregressive attention*.

Encoder-decoder
These are used for modeling complex mappings from one sequence of text to another; they're suitable for machine translation and summarization tasks. In addition to the Transformer architecture, which as we've seen combines an encoder and a decoder, the BART and T5 models belong to this class.

> In reality, the distinction between applications for decoder-only versus encoder-only architectures is a bit blurry. For example, decoder-only models like those in the GPT family can be primed for tasks like translation that are conventionally thought of as sequence-to-sequence tasks. Similarly, encoder-only models like BERT can be applied to summarization tasks that are usually associated with encoder-decoder or decoder-only models.[1]

Now that you have a high-level understanding of the Transformer architecture, let's take a closer look at the inner workings of the encoder.

[1] Y. Liu and M. Lapata, "Text Summarization with Pretrained Encoder", (2019).

The Encoder

As we saw earlier, the transformer's encoder consists of many encoder layers stacked next to each other. As illustrated in Figure 3-2, each encoder layer receives a sequence of embeddings and feeds them through the following sublayers:

- A multi-head self-attention layer
- A fully connected feed-forward layer that is applied to each input embedding

The output embeddings of each encoder layer have the same size as the inputs, and we'll soon see that the main role of the encoder stack is to "update" the input embeddings to produce representations that encode some contextual information in the sequence. For example, the word "apple" will be updated to be more "company-like" and less "fruit-like" if the words "keynote" or "phone" are close to it.

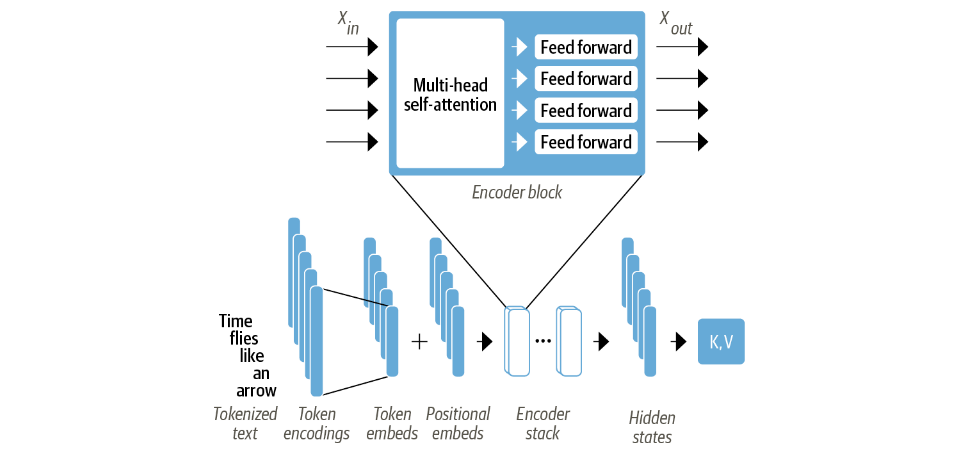


Figure 3-2. Zooming into the encoder layer

Each of these sublayers also uses skip connections and layer normalization, which are standard tricks to train deep neural networks effectively. But to truly understand what makes a transformer work, we have to go deeper. Let's start with the most important building block: the self-attention layer.

Self-Attention

As we discussed in Chapter 1, attention is a mechanism that allows neural networks to assign a different amount of weight or "attention" to each element in a sequence. For text sequences, the elements are *token embeddings* like the ones we encountered in Chapter 2, where each token is mapped to a vector of some fixed dimension. For example, in BERT each token is represented as a 768-dimensional vector. The "self" part of self-attention refers to the fact that these weights are computed for all hidden states in the same set—for example, all the hidden states of the encoder. By contrast, the attention mechanism associated with recurrent models involves computing the relevance of each encoder hidden state to the decoder hidden state at a given decoding timestep.

The main idea behind self-attention is that instead of using a fixed embedding for each token, we can use the whole sequence to compute a *weighted average* of each embedding. Another way to formulate this is to say that given a sequence of token embeddings $x_1, ..., x_n$, self-attention produces a sequence of new embeddings $x'_1, ..., x'_n$ where each x'_i is a linear combination of all the x_j:

$$x'_i = \sum_{j=1}^{n} w_{ji} x_j$$

The coefficients w_{ji} are called *attention weights* and are normalized so that $\sum_j w_{ji} = 1$. To see why averaging the token embeddings might be a good idea, consider what comes to mind when you see the word "flies". You might think of annoying insects, but if you were given more context, like "time flies like an arrow", then you would realize that "flies" refers to the verb instead. Similarly, we can create a representation for "flies" that incorporates this context by combining all the token embeddings in different proportions, perhaps by assigning a larger weight w_{ji} to the token embeddings for "time" and "arrow". Embeddings that are generated in this way are called *contextualized embeddings* and predate the invention of transformers in language models like ELMo.[2] A diagram of the process is shown in Figure 3-3, where we illustrate how, depending on the context, two different representations for "flies" can be generated via self-attention.

[2] M.E. Peters et al., "Deep Contextualized Word Representations", (2017).

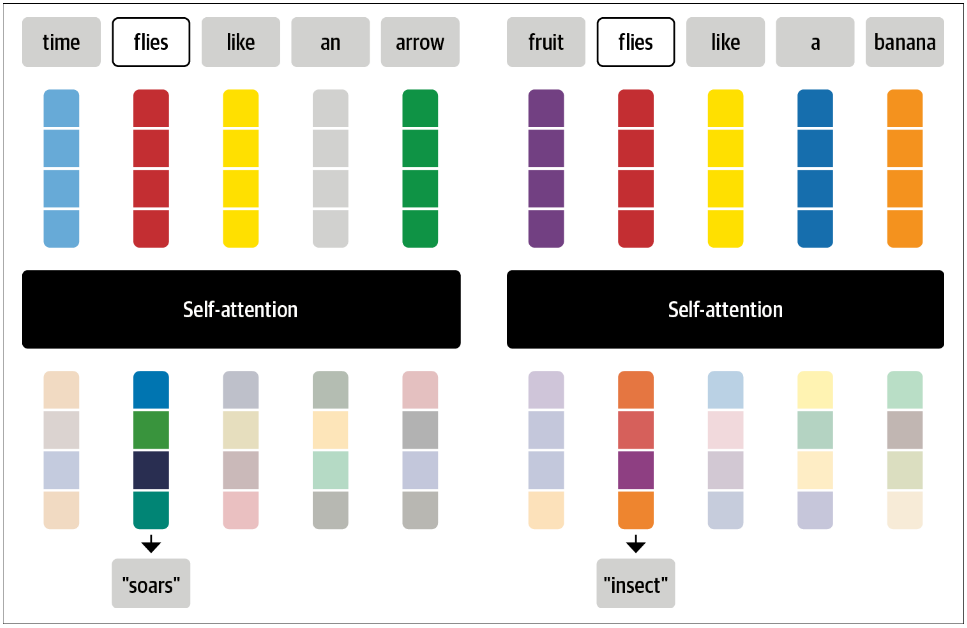


Figure 3-3. Diagram showing how self-attention updates raw token embeddings (upper) into contextualized embeddings (lower) to create representations that incorporate information from the whole sequence

Let's now take a look at how we can calculate the attention weights.

Scaled dot-product attention

There are several ways to implement a self-attention layer, but the most common one is *scaled dot-product attention*, from the paper introducing the Transformer architecture.[3] There are four main steps required to implement this mechanism:

1. Project each token embedding into three vectors called *query*, *key*, and *value*.
2. Compute attention scores. We determine how much the query and key vectors relate to each other using a *similarity function*. As the name suggests, the similarity function for scaled dot-product attention is the dot product, computed efficiently using matrix multiplication of the embeddings. Queries and keys that are similar will have a large dot product, while those that don't share much in common will have little to no overlap. The outputs from this step are called the *attention scores*, and for a sequence with n input tokens there is a corresponding $n \times n$ matrix of attention scores.

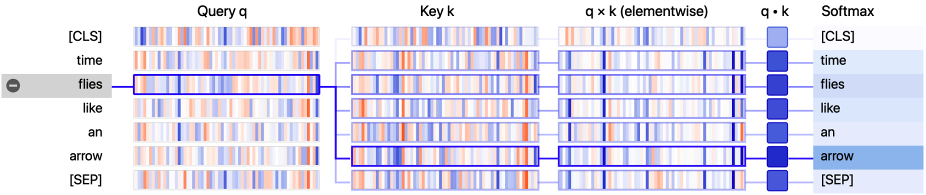
[3] A. Vaswani et al., "Attention Is All You Need", (2017).

3. Compute attention weights. Dot products can in general produce arbitrarily large numbers, which can destabilize the training process. To handle this, the attention scores are first multiplied by a scaling factor to normalize their variance and then normalized with a softmax to ensure all the column values sum to 1. The resulting $n \times n$ matrix now contains all the attention weights, w_{ji}.
4. Update the token embeddings. Once the attention weights are computed, we multiply them by the value vector $v_1, ..., v_n$ to obtain an updated representation for embedding $x_i' = \sum_j w_{ji} v_j$.

We can visualize how the attention weights are calculated with a nifty library called *BertViz* for Jupyter. This library provides several functions that can be used to visualize different aspects of attention in transformer models. To visualize the attention weights, we can use the `neuron_view` module, which traces the computation of the weights to show how the query and key vectors are combined to produce the final weight. Since BertViz needs to tap into the attention layers of the model, we'll instantiate our BERT checkpoint with the model class from BertViz and then use the `show()` function to generate the interactive visualization for a specific encoder layer and attention head. Note that you need to click the "+" on the left to activate the attention visualization:

```
from transformers import AutoTokenizer
from bertviz.transformers_neuron_view import BertModel
from bertviz.neuron_view import show

model_ckpt = "bert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(model_ckpt)
model = BertModel.from_pretrained(model_ckpt)
text = "time flies like an arrow"
show(model, "bert", tokenizer, text, display_mode="light", layer=0, head=8)
```



From the visualization, we can see the values of the query and key vectors are represented as vertical bands, where the intensity of each band corresponds to the magnitude. The connecting lines are weighted according to the attention between the tokens, and we can see that the query vector for "flies" has the strongest overlap with the key vector for "arrow".

Demystifying Queries, Keys, and Values

The notion of query, key, and value vectors may seem a bit cryptic the first time you encounter them. Their names were inspired by information retrieval systems, but we can motivate their meaning with a simple analogy. Imagine that you're at the supermarket buying all the ingredients you need for your dinner. You have the dish's recipe, and each of the required ingredients can be thought of as a query. As you scan the shelves, you look at the labels (keys) and check whether they match an ingredient on your list (similarity function). If you have a match, then you take the item (value) from the shelf.

In this analogy, you only get one grocery item for every label that matches the ingredient. Self-attention is a more abstract and "smooth" version of this: *every* label in the supermarket matches the ingredient to the extent to which each key matches the query. So if your list includes a dozen eggs, then you might end up grabbing 10 eggs, an omelette, and a chicken wing.

Let's take a look at this process in more detail by implementing the diagram of operations to compute scaled dot-product attention, as shown in Figure 3-4.

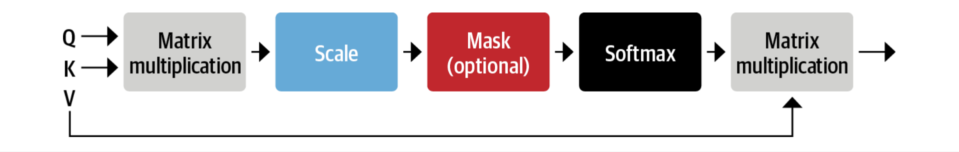


Figure 3-4. Operations in scaled dot-product attention

We will use PyTorch to implement the Transformer architecture in this chapter, but the steps in TensorFlow are analogous. We provide a mapping between the most important functions in the two frameworks in Table 3-1.

Table 3-1. PyTorch and TensorFlow (Keras) classes and methods used in this chapter

PyTorch	TensorFlow (Keras)	Creates/implements
`nn.Linear`	`keras.layers.Dense`	A dense neural network layer
`nn.Module`	`keras.layers.Layer`	The building blocks of models
`nn.Dropout`	`keras.layers.Dropout`	A dropout layer
`nn.LayerNorm`	`keras.layers.LayerNormalization`	Layer normalization
`nn.Embedding`	`keras.layers.Embedding`	An embedding layer
`nn.GELU`	`keras.activations.gelu`	The Gaussian Error Linear Unit activation function
`nn.bmm`	`tf.matmul`	Batched matrix multiplication
`model.forward`	`model.call`	The model's forward pass

The first thing we need to do is tokenize the text, so let's use our tokenizer to extract the input IDs:

```
inputs = tokenizer(text, return_tensors="pt", add_special_tokens=False)
inputs.input_ids
```

```
tensor([[ 2051, 10029,  2066,  2019,  8612]])
```

As we saw in Chapter 2, each token in the sentence has been mapped to a unique ID in the tokenizer's vocabulary. To keep things simple, we've also excluded the [CLS] and [SEP] tokens by setting add_special_tokens=False. Next, we need to create some dense embeddings. *Dense* in this context means that each entry in the embeddings contains a nonzero value. In contrast, the one-hot encodings we saw in Chapter 2 are *sparse*, since all entries except one are zero. In PyTorch, we can do this by using a torch.nn.Embedding layer that acts as a lookup table for each input ID:

```
from torch import nn
from transformers import AutoConfig

config = AutoConfig.from_pretrained(model_ckpt)
token_emb = nn.Embedding(config.vocab_size, config.hidden_size)
token_emb
```

```
Embedding(30522, 768)
```

Here we've used the AutoConfig class to load the *config.json* file associated with the bert-base-uncased checkpoint. In 🤗 Transformers, every checkpoint is assigned a configuration file that specifies various hyperparameters like vocab_size and hidden_size, which in our example shows us that each input ID will be mapped to one of the 30,522 embedding vectors stored in nn.Embedding, each with a size of 768. The AutoConfig class also stores additional metadata, such as the label names, which are used to format the model's predictions.

Note that the token embeddings at this point are independent of their context. This means that homonyms (words that have the same spelling but different meaning), like "flies" in the previous example, have the same representation. The role of the subsequent attention layers will be to mix these token embeddings to disambiguate and inform the representation of each token with the content of its context.

Now that we have our lookup table, we can generate the embeddings by feeding in the input IDs:

```
inputs_embeds = token_emb(inputs.input_ids)
inputs_embeds.size()
```

```
torch.Size([1, 5, 768])
```

This has given us a tensor of shape [batch_size, seq_len, hidden_dim], just like we saw in Chapter 2. We'll postpone the positional encodings, so the next step is to create the query, key, and value vectors and calculate the attention scores using the dot product as the similarity function:

```
import torch
from math import sqrt

query = key = value = inputs_embeds
dim_k = key.size(-1)
scores = torch.bmm(query, key.transpose(1,2)) / sqrt(dim_k)
scores.size()
```

```
torch.Size([1, 5, 5])
```

This has created a 5 × 5 matrix of attention scores per sample in the batch. We'll see later that the query, key, and value vectors are generated by applying independent weight matrices $W_{Q, K, V}$ to the embeddings, but for now we've kept them equal for simplicity. In scaled dot-product attention, the dot products are scaled by the size of the embedding vectors so that we don't get too many large numbers during training that can cause the softmax we will apply next to saturate.

> The `torch.bmm()` function performs a *batch matrix-matrix product* that simplifies the computation of the attention scores where the query and key vectors have the shape `[batch_size, seq_len, hidden_dim]`. If we ignored the batch dimension we could calculate the dot product between each query and key vector by simply transposing the key tensor to have the shape `[hidden_dim, seq_len]` and then using the matrix product to collect all the dot products in a `[seq_len, seq_len]` matrix. Since we want to do this for all sequences in the batch independently, we use `torch.bmm()`, which takes two batches of matrices and multiplies each matrix from the first batch with the corresponding matrix in the second batch.

Let's apply the softmax now:

```
import torch.nn.functional as F

weights = F.softmax(scores, dim=-1)
weights.sum(dim=-1)
```

```
tensor([[1., 1., 1., 1., 1.]], grad_fn=<SumBackward1>)
```

The final step is to multiply the attention weights by the values:

```
attn_outputs = torch.bmm(weights, value)
attn_outputs.shape
```

```
torch.Size([1, 5, 768])
```

And that's it—we've gone through all the steps to implement a simplified form of self-attention! Notice that the whole process is just two matrix multiplications and a softmax, so you can think of "self-attention" as just a fancy form of averaging.

Let's wrap these steps into a function that we can use later:

```
def scaled_dot_product_attention(query, key, value):
    dim_k = query.size(-1)
    scores = torch.bmm(query, key.transpose(1, 2)) / sqrt(dim_k)
    weights = F.softmax(scores, dim=-1)
    return torch.bmm(weights, value)
```

Our attention mechanism with equal query and key vectors will assign a very large score to identical words in the context, and in particular to the current word itself: the dot product of a query with itself is always 1. But in practice, the meaning of a word will be better informed by complementary words in the context than by identical words—for example, the meaning of "flies" is better defined by incorporating information from "time" and "arrow" than by another mention of "flies". How can we promote this behavior?

Let's allow the model to create a different set of vectors for the query, key, and value of a token by using three different linear projections to project our initial token vector into three different spaces.

Multi-headed attention

In our simple example, we only used the embeddings "as is" to compute the attention scores and weights, but that's far from the whole story. In practice, the self-attention layer applies three independent linear transformations to each embedding to generate the query, key, and value vectors. These transformations project the embeddings and each projection carries its own set of learnable parameters, which allows the self-attention layer to focus on different semantic aspects of the sequence.

It also turns out to be beneficial to have *multiple* sets of linear projections, each one representing a so-called *attention head*. The resulting *multi-head attention layer* is illustrated in Figure 3-5. But why do we need more than one attention head? The reason is that the softmax of one head tends to focus on mostly one aspect of similarity. Having several heads allows the model to focus on several aspects at once. For instance, one head can focus on subject-verb interaction, whereas another finds nearby adjectives. Obviously we don't handcraft these relations into the model, and they are fully learned from the data. If you are familiar with computer vision models you might see the resemblance to filters in convolutional neural networks, where one filter can be responsible for detecting faces and another one finds wheels of cars in images.

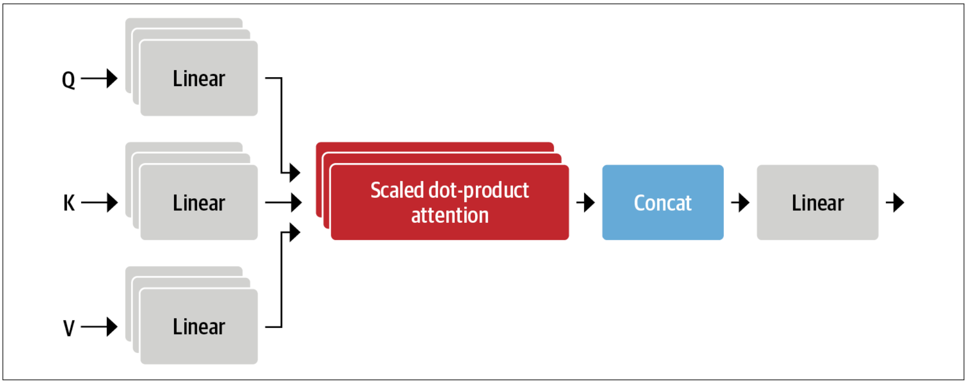


Figure 3-5. Multi-head attention

Let's implement this layer by first coding up a single attention head:

```
class AttentionHead(nn.Module):
    def __init__(self, embed_dim, head_dim):
        super().__init__()
        self.q = nn.Linear(embed_dim, head_dim)
        self.k = nn.Linear(embed_dim, head_dim)
        self.v = nn.Linear(embed_dim, head_dim)

    def forward(self, hidden_state):
        attn_outputs = scaled_dot_product_attention(
            self.q(hidden_state), self.k(hidden_state), self.v(hidden_state))
        return attn_outputs
```

Here we've initialized three independent linear layers that apply matrix multiplication to the embedding vectors to produce tensors of shape `[batch_size, seq_len, head_dim]`, where `head_dim` is the number of dimensions we are projecting into. Although `head_dim` does not have to be smaller than the number of embedding dimensions of the tokens (`embed_dim`), in practice it is chosen to be a multiple of `embed_dim` so that the computation across each head is constant. For example, BERT has 12 attention heads, so the dimension of each head is 768/12 = 64.

Now that we have a single attention head, we can concatenate the outputs of each one to implement the full multi-head attention layer:

```
class MultiHeadAttention(nn.Module):
    def __init__(self, config):
        super().__init__()
        embed_dim = config.hidden_size
        num_heads = config.num_attention_heads
        head_dim = embed_dim // num_heads
        self.heads = nn.ModuleList(
            [AttentionHead(embed_dim, head_dim) for _ in range(num_heads)]
        )
        self.output_linear = nn.Linear(embed_dim, embed_dim)
    def forward(self, hidden_state):
        x = torch.cat([h(hidden_state) for h in self.heads], dim=-1)
        x = self.output_linear(x)
        return x
```

Notice that the concatenated output from the attention heads is also fed through a final linear layer to produce an output tensor of shape `[batch_size, seq_len, hidden_dim]` that is suitable for the feed-forward network downstream. To confirm, let's see if the multi-head attention layer produces the expected shape of our inputs. We pass the configuration we loaded earlier from the pretrained BERT model when initializing the `MultiHeadAttention` module. This ensures that we use the same settings as BERT:

```
multihead_attn = MultiHeadAttention(config)
attn_output = multihead_attn(inputs_embeds)
attn_output.size()
```

```
torch.Size([1, 5, 768])
```

It works! To wrap up this section on attention, let's use BertViz again to visualize the attention for two different uses of the word "flies". Here we can use the `head_view()` function from BertViz by computing the attentions of a pretrained checkpoint and indicating where the sentence boundary lies:

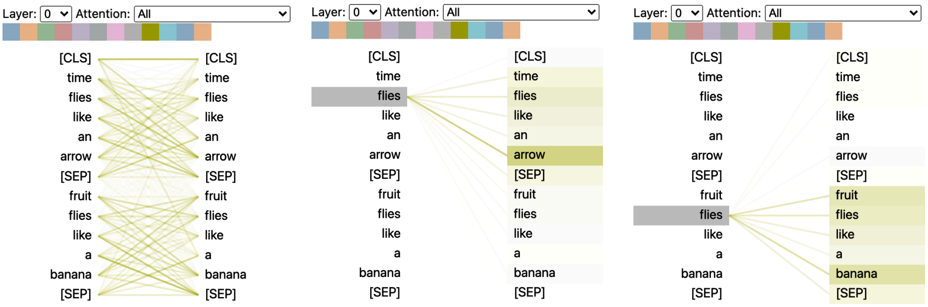
```
from bertviz import head_view
from transformers import AutoModel

model = AutoModel.from_pretrained(model_ckpt, output_attentions=True)

sentence_a = "time flies like an arrow"
sentence_b = "fruit flies like a banana"

viz_inputs = tokenizer(sentence_a, sentence_b, return_tensors='pt')
attention = model(**viz_inputs).attentions
sentence_b_start = (viz_inputs.token_type_ids == 0).sum(dim=1)
tokens = tokenizer.convert_ids_to_tokens(viz_inputs.input_ids[0])

head_view(attention, tokens, sentence_b_start, heads=[8])
```



Layer: 0 Attention: All

Layer: 0 Attention: All

Layer: 0 Attention: All

This visualization shows the attention weights as lines connecting the token whose embedding is getting updated (left) with every word that is being attended to (right). The intensity of the lines indicates the strength of the attention weights, with dark lines representing values close to 1, and faint lines representing values close to 0.

In this example, the input consists of two sentences and the `[CLS]` and `[SEP]` tokens are the special tokens in BERT's tokenizer that we encountered in Chapter 2. One thing we can see from the visualization is that the attention weights are strongest between words that belong to the same sentence, which suggests BERT can tell that it should attend to words in the same sentence. However, for the word "flies" we can see that BERT has identified "arrow" as important in the first sentence and "fruit" and "banana" in the second. These attention weights allow the model to distinguish the use of "flies" as a verb or noun, depending on the context in which it occurs!

Now that we've covered attention, let's take a look at implementing the missing piece of the encoder layer: position-wise feed-forward networks.

The Feed-Forward Layer

The feed-forward sublayer in the encoder and decoder is just a simple two-layer fully connected neural network, but with a twist: instead of processing the whole sequence of embeddings as a single vector, it processes each embedding *independently*. For this reason, this layer is often referred to as a *position-wise feed-forward layer*. You may also see it referred to as a one-dimensional convolution with a kernel size of one, typically by people with a computer vision background (e.g., the OpenAI GPT codebase uses this nomenclature). A rule of thumb from the literature is for the hidden size of the first layer to be four times the size of the embeddings, and a GELU activation function is most commonly used. This is where most of the capacity and memorization is hypothesized to happen, and it's the part that is most often scaled when scaling up the models. We can implement this as a simple `nn.Module` as follows:

```
class FeedForward(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.linear_1 = nn.Linear(config.hidden_size, config.intermediate_size)
        self.linear_2 = nn.Linear(config.intermediate_size, config.hidden_size)
        self.gelu = nn.GELU()
        self.dropout = nn.Dropout(config.hidden_dropout_prob)

    def forward(self, x):
        x = self.linear_1(x)
        x = self.gelu(x)
        x = self.linear_2(x)
        x = self.dropout(x)
        return x
```

Note that a feed-forward layer such as `nn.Linear` is usually applied to a tensor of shape `(batch_size, input_dim)`, where it acts on each element of the batch dimension independently. This is actually true for any dimension except the last one, so when we pass a tensor of shape `(batch_size, seq_len, hidden_dim)` the layer is applied to all token embeddings of the batch and sequence independently, which is exactly what we want. Let's test this by passing the attention outputs:

```
feed_forward = FeedForward(config)
ff_outputs = feed_forward(attn_outputs)
ff_outputs.size()
```

```
torch.Size([1, 5, 768])
```

We now have all the ingredients to create a fully fledged transformer encoder layer! The only decision left to make is where to place the skip connections and layer normalization. Let's take a look at how this affects the model architecture.

Adding Layer Normalization

As mentioned earlier, the Transformer architecture makes use of *layer normalization* and *skip connections*. The former normalizes each input in the batch to have zero mean and unity variance. Skip connections pass a tensor to the next layer of the model without processing and add it to the processed tensor. When it comes to placing the layer normalization in the encoder or decoder layers of a transformer, there are two main choices adopted in the literature:

Post layer normalization

This is the arrangement used in the Transformer paper; it places layer normalization in between the skip connections. This arrangement is tricky to train from scratch as the gradients can diverge. For this reason, you will often see a concept known as *learning rate warm-up*, where the learning rate is gradually increased from a small value to some maximum value during training.

Pre layer normalization

This is the most common arrangement found in the literature; it places layer normalization within the span of the skip connections. This tends to be much more stable during training, and it does not usually require any learning rate warm-up.

The difference between the two arrangements is illustrated in Figure 3-6.

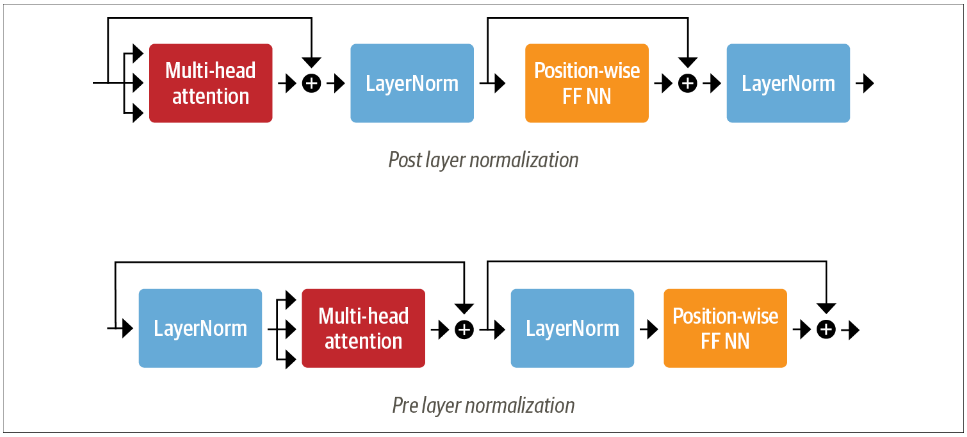


Figure 3-6. Different arrangements of layer normalization in a transformer encoder layer

We'll use the second arrangement, so we can simply stick together our building blocks as follows:

```
class TransformerEncoderLayer(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.layer_norm_1 = nn.LayerNorm(config.hidden_size)
        self.layer_norm_2 = nn.LayerNorm(config.hidden_size)
        self.attention = MultiHeadAttention(config)
        self.feed_forward = FeedForward(config)

    def forward(self, x):
        # Apply layer normalization and then copy input into query, key, value
        hidden_state = self.layer_norm_1(x)
        # Apply attention with a skip connection
        x = x + self.attention(hidden_state)
        # Apply feed-forward layer with a skip connection
        x = x + self.feed_forward(self.layer_norm_2(x))
        return x
```

Let's now test this with our input embeddings:

```
encoder_layer = TransformerEncoderLayer(config)
inputs_embeds.shape, encoder_layer(inputs_embeds).size()
```

```
(torch.Size([1, 5, 768]), torch.Size([1, 5, 768]))
```

We've now implemented our very first transformer encoder layer from scratch! However, there is a caveat with the way we set up the encoder layers: they are totally invariant to the position of the tokens. Since the multi-head attention layer is effectively a fancy weighted sum, the information on token position is lost.[4]

Luckily, there is an easy trick to incorporate positional information using positional embeddings. Let's take a look.

Positional Embeddings

Positional embeddings are based on a simple, yet very effective idea: augment the token embeddings with a position-dependent pattern of values arranged in a vector. If the pattern is characteristic for each position, the attention heads and feed-forward layers in each stack can learn to incorporate positional information into their transformations.

There are several ways to achieve this, and one of the most popular approaches is to use a learnable pattern, especially when the pretraining dataset is sufficiently large. This works exactly the same way as the token embeddings, but using the position index instead of the token ID as input. With that approach, an efficient way of encoding the positions of tokens is learned during pretraining.

Let's create a custom `Embeddings` module that combines a token embedding layer that projects the `input_ids` to a dense hidden state together with the positional embedding that does the same for `position_ids`. The resulting embedding is simply the sum of both embeddings:

```
class Embeddings(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.token_embeddings = nn.Embedding(config.vocab_size,
                                             config.hidden_size)
        self.position_embeddings = nn.Embedding(config.max_position_embeddings,
                                                config.hidden_size)
        self.layer_norm = nn.LayerNorm(config.hidden_size, eps=1e-12)
        self.dropout = nn.Dropout()

    def forward(self, input_ids):
        # Create position IDs for input sequence
        seq_length = input_ids.size(1)
        position_ids = torch.arange(seq_length, dtype=torch.long).unsqueeze(0)
        # Create token and position embeddings
        token_embeddings = self.token_embeddings(input_ids)
        position_embeddings = self.position_embeddings(position_ids)
        # Combine token and position embeddings
        embeddings = token_embeddings + position_embeddings
        embeddings = self.layer_norm(embeddings)
```

[4] In fancier terminology, the self-attention and feed-forward layers are said to be *permutation equivariant*—if the input is permuted then the corresponding output of the layer is permuted in exactly the same way.

```
        embeddings = self.dropout(embeddings)
        return embeddings

embedding_layer = Embeddings(config)
embedding_layer(inputs.input_ids).size()
```

```
torch.Size([1, 5, 768])
```

We see that the embedding layer now creates a single, dense embedding for each token.

While learnable position embeddings are easy to implement and widely used, there are some alternatives:

Absolute positional representations
: Transformer models can use static patterns consisting of modulated sine and cosine signals to encode the positions of the tokens. This works especially well when there are not large volumes of data available.

Relative positional representations
: Although absolute positions are important, one can argue that when computing an embedding, the surrounding tokens are most important. Relative positional representations follow that intuition and encode the relative positions between tokens. This cannot be set up by just introducing a new relative embedding layer at the beginning, since the relative embedding changes for each token depending on where from the sequence we are attending to it. Instead, the attention mechanism itself is modified with additional terms that take the relative position between tokens into account. Models such as DeBERTa use such representations.[5]

Let's put all of this together now by building the full transformer encoder combining the embeddings with the encoder layers:

```
class TransformerEncoder(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.embeddings = Embeddings(config)
        self.layers = nn.ModuleList([TransformerEncoderLayer(config)
                                     for _ in range(config.num_hidden_layers)])

    def forward(self, x):
        x = self.embeddings(x)
        for layer in self.layers:
            x = layer(x)
        return x
```

Let's check the output shapes of the encoder:

[5] By combining the idea of absolute and relative positional representations, rotary position embeddings achieve excellent results on many tasks. GPT-Neo is an example of a model with rotary position embeddings.

```
encoder = TransformerEncoder(config)
encoder(inputs.input_ids).size()
```

```
torch.Size([1, 5, 768])
```

We can see that we get a hidden state for each token in the batch. This output format makes the architecture very flexible, and we can easily adapt it for various applications such as predicting missing tokens in masked language modeling or predicting the start and end position of an answer in question answering. In the following section we'll see how we can build a classifier like the one we used in Chapter 2.

Adding a Classification Head

Transformer models are usually divided into a task-independent body and a task-specific head. We'll encounter this pattern again in Chapter 4 when we look at the design pattern of 🤗 Transformers. What we have built so far is the body, so if we wish to build a text classifier, we will need to attach a classification head to that body. We have a hidden state for each token, but we only need to make one prediction. There are several options to approach this. Traditionally, the first token in such models is used for the prediction and we can attach a dropout and a linear layer to make a classification prediction. The following class extends the existing encoder for sequence classification:

```
class TransformerForSequenceClassification(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.encoder = TransformerEncoder(config)
        self.dropout = nn.Dropout(config.hidden_dropout_prob)
        self.classifier = nn.Linear(config.hidden_size, config.num_labels)

    def forward(self, x):
        x = self.encoder(x)[:, 0, :] # select hidden state of [CLS] token
        x = self.dropout(x)
        x = self.classifier(x)
        return x
```

Before initializing the model we need to define how many classes we would like to predict:

```
config.num_labels = 3
encoder_classifier = TransformerForSequenceClassification(config)
encoder_classifier(inputs.input_ids).size()
```

```
torch.Size([1, 3])
```

That is exactly what we have been looking for. For each example in the batch we get the unnormalized logits for each class in the output. This corresponds to the BERT model that we used in Chapter 2 to detect emotions in tweets.

This concludes our analysis of the encoder and how we can combine it with a task-specific head. Let's now cast our attention (pun intended!) to the decoder.

The Decoder

As illustrated in Figure 3-7, the main difference between the decoder and encoder is that the decoder has *two* attention sublayers:

Masked multi-head self-attention layer

Ensures that the tokens we generate at each timestep are only based on the past outputs and the current token being predicted. Without this, the decoder could cheat during training by simply copying the target translations; masking the inputs ensures the task is not trivial.

Encoder-decoder attention layer

Performs multi-head attention over the output key and value vectors of the encoder stack, with the intermediate representations of the decoder acting as the queries.[6] This way the encoder-decoder attention layer learns how to relate tokens from two different sequences, such as two different languages. The decoder has access to the encoder keys and values in each block.

Let's take a look at the modifications we need to make to include masking in our self-attention layer, and leave the implementation of the encoder-decoder attention layer as a homework problem. The trick with masked self-attention is to introduce a *mask matrix* with ones on the lower diagonal and zeros above:

```
seq_len = inputs.input_ids.size(-1)
mask = torch.tril(torch.ones(seq_len, seq_len)).unsqueeze(0)
mask[0]
```

```
tensor([[1., 0., 0., 0., 0.],
        [1., 1., 0., 0., 0.],
        [1., 1., 1., 0., 0.],
        [1., 1., 1., 1., 0.],
        [1., 1., 1., 1., 1.]])
```

Here we've used PyTorch's `tril()` function to create the lower triangular matrix. Once we have this mask matrix, we can prevent each attention head from peeking at future tokens by using `Tensor.masked_fill()` to replace all the zeros with negative infinity:

```
scores.masked_fill(mask == 0, -float("inf"))
```

[6] Note that unlike the self-attention layer, the key and query vectors in encoder-decoder attention can have different lengths. This is because the encoder and decoder inputs will generally involve sequences of differing length. As a result, the matrix of attention scores in this layer is rectangular, not square.

```
tensor([[[26.8082,    -inf,    -inf,    -inf,    -inf],
         [-0.6981, 26.9043,    -inf,    -inf,    -inf],
         [-2.3190,  1.2928, 27.8710,    -inf,    -inf],
         [-0.5897,  0.3497, -0.3807, 27.5488,    -inf],
         [ 0.5275,  2.0493, -0.4869,  1.6100, 29.0893]]],
       grad_fn=<MaskedFillBackward0>)
```

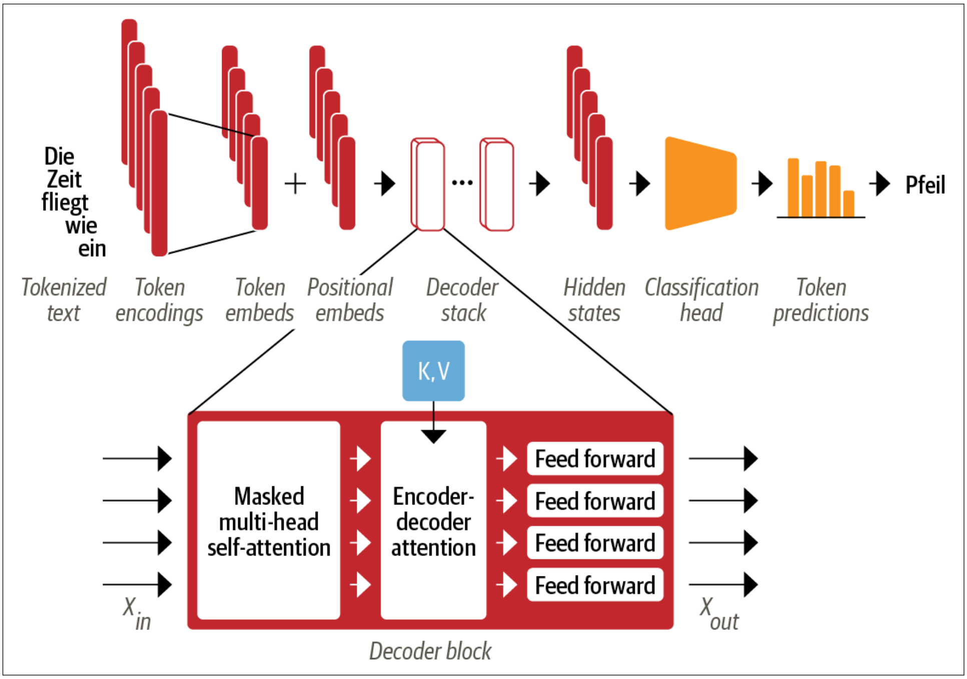


Figure 3-7. Zooming into the transformer decoder layer

By setting the upper values to negative infinity, we guarantee that the attention weights are all zero once we take the softmax over the scores because $e^{-\infty} = 0$ (recall that softmax calculates the normalized exponential). We can easily include this masking behavior with a small change to our scaled dot-product attention function that we implemented earlier in this chapter:

```
def scaled_dot_product_attention(query, key, value, mask=None):
    dim_k = query.size(-1)
    scores = torch.bmm(query, key.transpose(1, 2)) / sqrt(dim_k)
    if mask is not None:
        scores = scores.masked_fill(mask == 0, float("-inf"))
    weights = F.softmax(scores, dim=-1)
    return weights.bmm(value)
```

From here it is a simple matter to build up the decoder layer; we point the reader to the excellent implementation of minGPT by Andrej Karpathy for details.

We've given you a lot of technical information here, but now you should have a good understanding of how every piece of the Transformer architecture works. Before we move on to building models for tasks more advanced than text classification, let's round out the chapter by stepping back a bit and looking at the landscape of different transformer models and how they relate to each other.

Demystifying Encoder-Decoder Attention

Let's see if we can shed some light on the mysteries of encoder-decoder attention. Imagine you (the decoder) are in class taking an exam. Your task is to predict the next word based on the previous words (decoder inputs), which sounds simple but is incredibly hard (try it yourself and predict the next words in a passage of this book). Fortunately, your neighbor (the encoder) has the full text. Unfortunately, they're a foreign exchange student and the text is in their mother tongue. Cunning students that you are, you figure out a way to cheat anyway. You draw a little cartoon illustrating the text you already have (the query) and give it to your neighbor. They try to figure out which passage matches that description (the key), draw a cartoon describing the word following that passage (the value), and pass that back to you. With this system in place, you ace the exam.

Meet the Transformers

As you've seen in this chapter, there are three main architectures for transformer models: encoders, decoders, and encoder-decoders. The initial success of the early transformer models triggered a Cambrian explosion in model development as researchers built models on various datasets of different size and nature, used new pretraining objectives, and tweaked the architecture to further improve performance. Although the zoo of models is still growing fast, they can still be divided into these three categories.

In this section we'll provide a brief overview of the most important transformer models in each class. Let's start by taking a look at the transformer family tree.

The Transformer Tree of Life

Over time, each of the three main architectures has undergone an evolution of its own. This is illustrated in Figure 3-8, which shows a few of the most prominent models and their descendants.

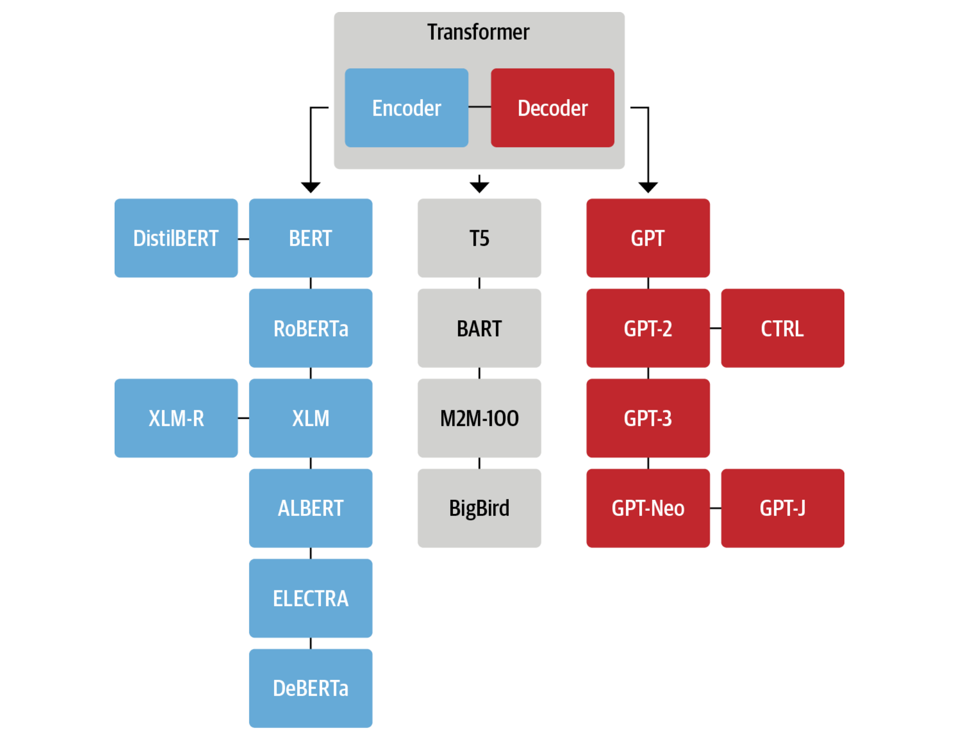


Figure 3-8. An overview of some of the most prominent transformer architectures

With over 50 different architectures included in 🤗 Transformers, this family tree by no means provides a complete overview of all the ones that exist: it simply highlights a few of the architectural milestones. We've covered the original Transformer architecture in depth in this chapter, so let's take a closer look at some of the key descendants, starting with the encoder branch.

The Encoder Branch

The first encoder-only model based on the Transformer architecture was BERT. At the time it was published, it outperformed all the state-of-the-art models on the popular GLUE benchmark,[7] which measures natural language understanding (NLU) across several tasks of varying difficulty. Subsequently, the pretraining objective and the architecture of BERT have been adapted to further improve performance. Encoder-only models still dominate research and industry on NLU tasks such as text

[7] A. Wang et al., "GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding", (2018).

classification, named entity recognition, and question answering. Let's have a brief look at the BERT model and its variants:

BERT

BERT is pretrained with the two objectives of predicting masked tokens in texts and determining if one text passage is likely to follow another.[8] The former task is called *masked language modeling* (MLM) and the latter *next sentence prediction* (NSP).

DistilBERT

Although BERT delivers great results, it's size can make it tricky to deploy in environments where low latencies are required. By using a technique known as knowledge distillation during pretraining, DistilBERT achieves 97% of BERT's performance while using 40% less memory and being 60% faster.[9] You can find more details on knowledge distillation in Chapter 8.

RoBERTa

A study following the release of BERT revealed that its performance can be further improved by modifying the pretraining scheme. RoBERTa is trained longer, on larger batches with more training data, and it drops the NSP task.[10] Together, these changes significantly improve its performance compared to the original BERT model.

XLM

Several pretraining objectives for building multilingual models were explored in the work on the cross-lingual language model (XLM),[11] including the autoregressive language modeling from GPT-like models and MLM from BERT. In addition, the authors of the paper on XLM pretraining introduced *translation language modeling* (TLM), which is an extension of MLM to multiple language inputs. Experimenting with these pretraining tasks, they achieved state-of-the-art results on several multilingual NLU benchmarks as well as on translation tasks.

XLM-RoBERTa

Following the work of XLM and RoBERTa, the XLM-RoBERTa or XLM-R model takes multilingual pretraining one step further by massively upscaling the training data.[12] Using the Common Crawl corpus, its developers created a dataset with 2.5 terabytes of text; they then trained an encoder with MLM on this

[8] J. Devlin et al., "BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding", (2018).

[9] V. Sanh et al., "DistilBERT, a Distilled Version of BERT: Smaller, Faster, Cheaper and Lighter", (2019).

[10] Y. Liu et al., "RoBERTa: A Robustly Optimized BERT Pretraining Approach", (2019).

[11] G. Lample, and A. Conneau, "Cross-Lingual Language Model Pretraining", (2019).

[12] A. Conneau et al., "Unsupervised Cross-Lingual Representation Learning at Scale", (2019).

dataset. Since the dataset only contains data without parallel texts (i.e., translations), the TLM objective of XLM was dropped. This approach beats XLM and multilingual BERT variants by a large margin, especially on low-resource languages.

ALBERT

The ALBERT model introduced three changes to make the encoder architecture more efficient.[13] First, it decouples the token embedding dimension from the hidden dimension, thus allowing the embedding dimension to be small and thereby saving parameters, especially when the vocabulary gets large. Second, all layers share the same parameters, which decreases the number of effective parameters even further. Finally, the NSP objective is replaced with a sentence-ordering prediction: the model needs to predict whether or not the order of two consecutive sentences was swapped rather than predicting if they belong together at all. These changes make it possible to train even larger models with fewer parameters and reach superior performance on NLU tasks.

ELECTRA

One limitation of the standard MLM pretraining objective is that at each training step only the representations of the masked tokens are updated, while the other input tokens are not. To address this issue, ELECTRA uses a two-model approach:[14] the first model (which is typically small) works like a standard masked language model and predicts masked tokens. The second model, called the *discriminator*, is then tasked to predict which of the tokens in the first model's output were originally masked. Therefore, the discriminator needs to make a binary classification for every token, which makes training 30 times more efficient. For downstream tasks the discriminator is fine-tuned like a standard BERT model.

DeBERTa

The DeBERTa model introduces two architectural changes.[15] First, each token is represented as two vectors: one for the content, the other for relative position. By disentangling the tokens' content from their relative positions, the self-attention layers can better model the dependency of nearby token pairs. On the other hand, the absolute position of a word is also important, especially for decoding. For this reason, an absolute position embedding is added just before the softmax layer of the token decoding head. DeBERTa is the first model (as an ensemble) to

[13] Z. Lan et al., "ALBERT: A Lite BERT for Self-Supervised Learning of Language Representations", (2019).

[14] K. Clark et al., "ELECTRA: Pre-Training Text Encoders as Discriminators Rather Than Generators", (2020).

[15] P. He et al., "DeBERTa: Decoding-Enhanced BERT with Disentangled Attention", (2020).

beat the human baseline on the SuperGLUE benchmark,[16] a more difficult version of GLUE consisting of several subtasks used to measure NLU performance.

Now that we've highlighted some of the major encoder-only architectures, let's take a look at the decoder-only models.

The Decoder Branch

The progress on transformer decoder models has been spearheaded to a large extent by OpenAI. These models are exceptionally good at predicting the next word in a sequence and are thus mostly used for text generation tasks (see Chapter 5 for more details). Their progress has been fueled by using larger datasets and scaling the language models to larger and larger sizes. Let's have a look at the evolution of these fascinating generation models:

GPT

The introduction of GPT combined two key ideas in NLP:[17] the novel and efficient transformer decoder architecture, and transfer learning. In that setup, the model was pretrained by predicting the next word based on the previous ones. The model was trained on the BookCorpus and achieved great results on downstream tasks such as classification.

GPT-2

Inspired by the success of the simple and scalable pretraining approach, the original model and training set were upscaled to produce GPT-2.[18] This model is able to produce long sequences of coherent text. Due to concerns about possible misuse, the model was released in a staged fashion, with smaller models being published first and the full model later.

CTRL

Models like GPT-2 can continue an input sequence (also called a *prompt*). However, the user has little control over the style of the generated sequence. The Conditional Transformer Language (CTRL) model addresses this issue by adding "control tokens" at the beginning of the sequence.[19] These allow the style of the generated text to be controlled, which allows for diverse generation.

[16] A. Wang et al., "SuperGLUE: A Stickier Benchmark for General-Purpose Language Understanding Systems", (2019).

[17] A. Radford et al., "Improving Language Understanding by Generative Pre-Training", OpenAI (2018).

[18] A. Radford et al., "Language Models Are Unsupervised Multitask Learners", OpenAI (2019).

[19] N.S. Keskar et al., "CTRL: A Conditional Transformer Language Model for Controllable Generation", (2019).

GPT-3

Following the success of scaling GPT up to GPT-2, a thorough analysis on the behavior of language models at different scales revealed that there are simple power laws that govern the relation between compute, dataset size, model size, and the performance of a language model.[20] Inspired by these insights, GPT-2 was upscaled by a factor of 100 to yield GPT-3,[21] with 175 billion parameters. Besides being able to generate impressively realistic text passages, the model also exhibits few-shot learning capabilities: with a few examples of a novel task such as translating text to code, the model is able to accomplish the task on new examples. OpenAI has not open-sourced this model, but provides an interface through the OpenAI API.

GPT-Neo/GPT-J-6B

GPT-Neo and GPT-J-6B are GPT-like models that were trained by EleutherAI, a collective of researchers who aim to re-create and release GPT-3 scale models.[22] The current models are smaller variants of the full 175-billion-parameter model, with 1.3, 2.7, and 6 billion parameters, and are competitive with the smaller GPT-3 models OpenAI offers.

The final branch in the transformers tree of life is the encoder-decoder models. Let's take a look.

The Encoder-Decoder Branch

Although it has become common to build models using a single encoder or decoder stack, there are several encoder-decoder variants of the Transformer architecture that have novel applications across both NLU and NLG domains:

T5

The T5 model unifies all NLU and NLG tasks by converting them into text-to-text tasks.[23] All tasks are framed as sequence-to-sequence tasks, where adopting an encoder-decoder architecture is natural. For text classification problems, for example, this means that the text is used as the encoder input and the decoder has to generate the label as normal text instead of a class. We will look at this in more detail in Chapter 6. The T5 architecture uses the original Transformer architecture. Using the large crawled C4 dataset, the model is pretrained with masked language modeling as well as the SuperGLUE tasks by translating all of

[20] J. Kaplan et al., "Scaling Laws for Neural Language Models", (2020).

[21] T. Brown et al., "Language Models Are Few-Shot Learners", (2020).

[22] S. Black et al., "GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-TensorFlow", (2021); B. Wang and A. Komatsuzaki, "GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model", (2021).

[23] C. Raffel et al., "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer", (2019).

them to text-to-text tasks. The largest model with 11 billion parameters yielded state-of-the-art results on several benchmarks.

BART

BART combines the pretraining procedures of BERT and GPT within the encoder-decoder architecture.[24] The input sequences undergo one of several possible transformations, from simple masking to sentence permutation, token deletion, and document rotation. These modified inputs are passed through the encoder, and the decoder has to reconstruct the original texts. This makes the model more flexible as it is possible to use it for NLU as well as NLG tasks, and it achieves state-of-the-art-performance on both.

M2M-100

Conventionally a translation model is built for one language pair and translation direction. Naturally, this does not scale to many languages, and in addition there might be shared knowledge between language pairs that could be leveraged for translation between rare languages. M2M-100 is the first translation model that can translate between any of 100 languages.[25] This allows for high-quality translations between rare and underrepresented languages. The model uses prefix tokens (similar to the special `[CLS]` token) to indicate the source and target language.

BigBird

One main limitation of transformer models is the maximum context size, due to the quadratic memory requirements of the attention mechanism. BigBird addresses this issue by using a sparse form of attention that scales linearly.[26] This allows for the drastic scaling of contexts from 512 tokens in most BERT models to 4,096 in BigBird. This is especially useful in cases where long dependencies need to be conserved, such as in text summarization.

Pretrained checkpoints of all models that we have seen in this section are available on the Hugging Face Hub and can be fine-tuned to your use case with 🤗 Transformers, as described in the previous chapter.

Conclusion

In this chapter we started at the heart of the Transformer architecture with a deep dive into self-attention, and we subsequently added all the necessary parts to build a

[24] M. Lewis et al., "BART: Denoising Sequence-to-Sequence Pre-Training for Natural Language Generation, Translation, and Comprehension", (2019).

[25] A. Fan et al., "Beyond English-Centric Multilingual Machine Translation", (2020).

[26] M. Zaheer et al., "Big Bird: Transformers for Longer Sequences", (2020).

CHAPTER 5

Text Generation

One of the most uncanny features of transformer-based language models is their ability to generate text that is almost indistinguishable from text written by humans. A famous example is OpenAI's GPT-2, which when given the prompt:[1]

> In a shocking finding, scientist discovered a herd of unicorns living in a remote, previously unexplored valley, in the Andes Mountains. Even more surprising to the researchers was the fact that the unicorns spoke perfect English.

was able to generate a compelling news article about talking unicorns:

> The scientist named the population, after their distinctive horn, Ovid's Unicorn. These four-horned, silver-white unicorns were previously unknown to science. Now, after almost two centuries, the mystery of what sparked this odd phenomenon is finally solved. Dr. Jorge Pérez, an evolutionary biologist from the University of La Paz, and several companions, were exploring the Andes Mountains when they found a small valley, with no other animals or humans. Pérez noticed that the valley had what appeared to be a natural fountain, surrounded by two peaks of rock and silver snow. Pérez and the others then ventured further into the valley. "By the time we reached the top of one peak, the water looked blue, with some crystals on top," said Pérez. Pérez and his friends were astonished to see the unicorn herd. These creatures could be seen from the air without having to move too much to see them—they were so close they could touch their horns. While examining these bizarre creatures the scientists discovered that the creatures also spoke some fairly regular English …

[1] This example comes from OpenAI's blog post on GPT-2.

What makes this example so remarkable is that it was generated without any explicit supervision! By simply learning to predict the next word in the text of millions of web pages, GPT-2 and its more powerful descendants like GPT-3 are able to acquire a broad set of skills and pattern recognition abilities that can be activated with different kinds of input prompts. Figure 5-1 shows how language models are sometimes exposed during pretraining to sequences of tasks where they need to predict the following tokens based on the context alone, like addition, unscrambling words, and translation. This allows them to transfer this knowledge effectively during fine-tuning or (if the model is large enough) at inference time. These tasks are not chosen ahead of time, but occur naturally in the huge corpora used to train billion-parameter language models.

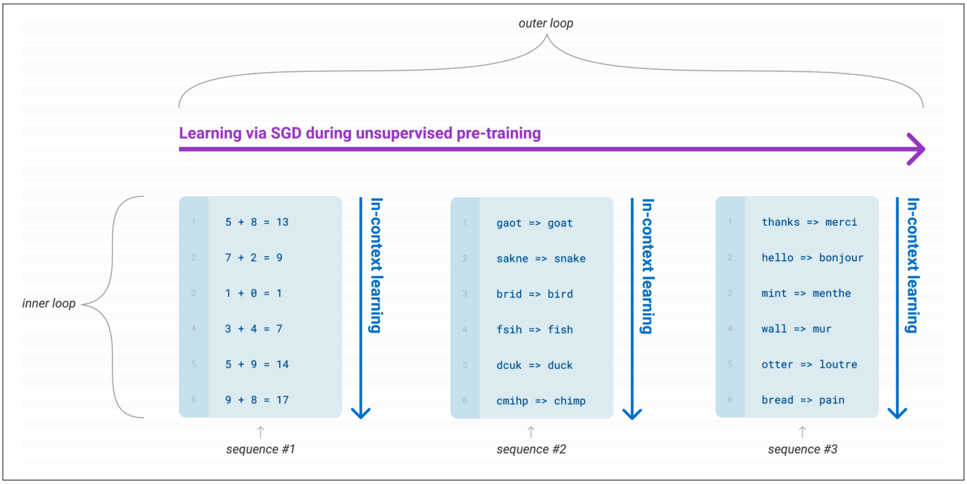
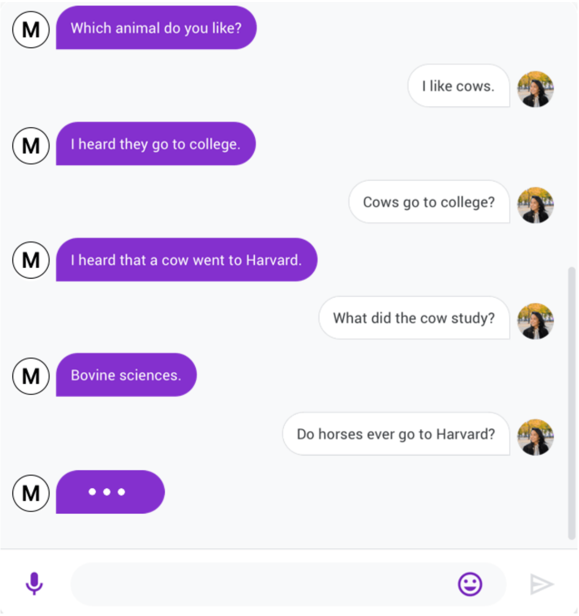


Figure 5-1. During pretraining, language models are exposed to sequences of tasks that can be adapted during inference (courtesy of Tom B. Brown)

The ability of transformers to generate realistic text has led to a diverse range of applications, like InferKit, Write With Transformer, AI Dungeon, and conversational agents like Google's Meena that can even tell corny jokes, as shown in Figure 5-2![2]

[2] However, as Delip Rao points out, whether Meena *intends* to tell corny jokes is a subtle question.



Which animal do you like?

I like cows.

I heard they go to college.

Cows go to college?

I heard that a cow went to Harvard.

What did the cow study?

Bovine sciences.

Do horses ever go to Harvard?

Figure 5-2. Meena on the left telling a corny joke to a human on the right (courtesy of Daniel Adiwardana and Thang Luong)

In this chapter we'll use GPT-2 to illustrate how text generation works for language models and explore how different decoding strategies impact the generated texts.

The Challenge with Generating Coherent Text

So far in this book, we have focused on tackling NLP tasks via a combination of pretraining and supervised fine-tuning. As we've seen, for task-specific heads like sequence or token classification, generating predictions is fairly straightforward; the model produces some logits and we either take the maximum value to get the predicted class, or apply a softmax function to obtain the predicted probabilities per class. By contrast, converting the model's probabilistic output to text requires a *decoding method*, which introduces a few challenges that are unique to text generation:

- The decoding is done *iteratively* and thus involves significantly more compute than simply passing inputs once through the forward pass of a model.
- The *quality* and *diversity* of the generated text depend on the choice of decoding method and associated hyperparameters.

To understand how this decoding process works, let's start by examining how GPT-2 is pretrained and subsequently applied to generate text.

Like other *autoregressive* or *causal language models*, GPT-2 is pretrained to estimate the probability $P(\mathbf{y}|\mathbf{x})$ of a sequence of tokens $\mathbf{y} = y_1, y_2, ...y_t$ occurring in the text, given some initial prompt or context sequence $\mathbf{x} = x_1, x_2, ...x_k$. Since it is impractical to acquire enough training data to estimate $P(\mathbf{y}|\mathbf{x})$ directly, it is common to use the chain rule of probability to factorize it as a product of *conditional* probabilities:

$$P\left(y_1, ..., y_t|\mathbf{x}\right) = \prod_{t=1}^{N} P\left(y_t|y_{<t}, \mathbf{x}\right)$$

where $y_{<t}$ is a shorthand notation for the sequence $y_1, ..., y_{t-1}$. It is from these conditional probabilities that we pick up the intuition that autoregressive language modeling amounts to predicting each word given the preceding words in a sentence; this is exactly what the probability on the righthand side of the preceding equation describes. Notice that this pretraining objective is quite different from BERT's, which utilizes both *past* and *future* contexts to predict a *masked* token.

By now you may have guessed how we can adapt this next token prediction task to generate text sequences of arbitrary length. As shown in Figure 5-3, we start with a prompt like "Transformers are the" and use the model to predict the next token. Once we have determined the next token, we append it to the prompt and then use the new input sequence to generate another token. We do this until we have reached a special end-of-sequence token or a predefined maximum length.

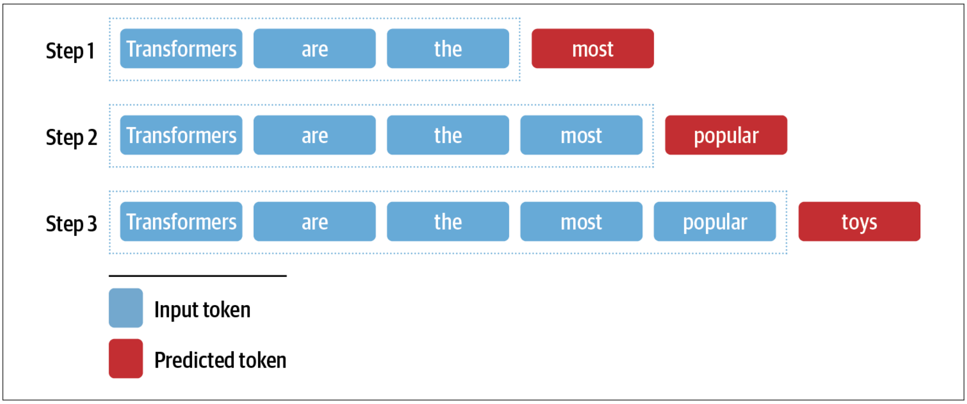


Figure 5-3. Generating text from an input sequence by adding a new word to the input at each step

Since the output sequence is *conditioned* on the choice of input prompt, this type of text generation is often called *conditional text generation*.

At the heart of this process lies a decoding method that determines which token is selected at each timestep. Since the language model head produces a logit $z_{t,i}$ per token in the vocabulary at each step, we can get the probability distribution over the next possible token w_i by taking the softmax:

$$P\left(y_t = w_i \middle| y_{<t}, \mathbf{x}\right) = \text{softmax}\left(z_{t,i}\right)$$

The goal of most decoding methods is to search for the most likely overall sequence by picking a $\hat{\mathbf{y}}$ such that:

$$\hat{\mathbf{y}} = \underset{\mathbf{y}}{\text{argmax}}\, P(\mathbf{y} | \mathbf{x})$$

Finding $\hat{\mathbf{y}}$ directly would involve evaluating every possible sequence with the language model. Since there does not exist an algorithm that can do this in a reasonable amount of time, we rely on approximations instead. In this chapter we'll explore a few of these approximations and gradually build up toward smarter and more complex algorithms that can be used to generate high-quality texts.

Greedy Search Decoding

The simplest decoding method to get discrete tokens from a model's continuous output is to greedily select the token with the highest probability at each timestep:

$$\hat{y}_t = \underset{y_t}{\text{argmax}}\, P\left(y_t \middle| y_{<t}, \mathbf{x}\right)$$

To see how greedy search works, let's start by loading the 1.5-billion-parameter version of GPT-2 with a language modeling head:[3]

```
import torch
from transformers import AutoTokenizer, AutoModelForCausalLM

device = "cuda" if torch.cuda.is_available() else "cpu"
model_name = "gpt2-xl"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(model_name).to(device)
```

Now let's generate some text! Although 🤗 Transformers provides a `generate()` function for autoregressive models like GPT-2, we'll implement this decoding method

[3] If you run out of memory on your machine, you can load a smaller GPT-2 version by replacing `model_name = "gpt-xl"` with `model_name = "gpt"`.

ourselves to see what goes on under the hood. To warm up, we'll take the same iterative approach shown in Figure 5-3: we'll use "Transformers are the" as the input prompt and run the decoding for eight timesteps. At each timestep, we pick out the model's logits for the last token in the prompt and wrap them with a softmax to get a probability distribution. We then pick the next token with the highest probability, add it to the input sequence, and run the process again. The following code does the job, and also stores the five most probable tokens at each timestep so we can visualize the alternatives:

```python
import pandas as pd

input_txt = "Transformers are the"
input_ids = tokenizer(input_txt, return_tensors="pt")["input_ids"].to(device)
iterations = []
n_steps = 8
choices_per_step = 5

with torch.no_grad():
    for _ in range(n_steps):
        iteration = dict()
        iteration["Input"] = tokenizer.decode(input_ids[0])
        output = model(input_ids=input_ids)
        # Select logits of the first batch and the last token and apply softmax
        next_token_logits = output.logits[0, -1, :]
        next_token_probs = torch.softmax(next_token_logits, dim=-1)
        sorted_ids = torch.argsort(next_token_probs, dim=-1, descending=True)
        # Store tokens with highest probabilities
        for choice_idx in range(choices_per_step):
            token_id = sorted_ids[choice_idx]
            token_prob = next_token_probs[token_id].cpu().numpy()
            token_choice = (
                f"{tokenizer.decode(token_id)} ({100 * token_prob:.2f}%)"
            )
            iteration[f"Choice {choice_idx+1}"] = token_choice
        # Append predicted next token to input
        input_ids = torch.cat([input_ids, sorted_ids[None, 0, None]], dim=-1)
        iterations.append(iteration)

pd.DataFrame(iterations)
```

	Input	Choice 1	Choice 2	Choice 3	Choice 4	Choice 5
0	Transformers are the	most (8.53%)	only (4.96%)	best (4.65%)	Transformers (4.37%)	ultimate (2.16%)
1	Transformers are the most	popular (16.78%)	powerful (5.37%)	common (4.96%)	famous (3.72%)	successful (3.20%)
2	Transformers are the most popular	toy (10.63%)	toys (7.23%)	Transformers (6.60%)	of (5.46%)	and (3.76%)
3	Transformers are the most popular toy	line (34.38%)	in (18.20%)	of (11.71%)	brand (6.10%)	line (2.69%)

	Input	Choice 1	Choice 2	Choice 3	Choice 4	Choice 5
4	Transformers are the most popular toy line	in (46.28%)	of (15.09%)	, (4.94%)	on (4.40%)	ever (2.72%)
5	Transformers are the most popular toy line in	the (65.99%)	history (12.42%)	America (6.91%)	Japan (2.44%)	North (1.40%)
6	Transformers are the most popular toy line in the	world (69.26%)	United (4.55%)	history (4.29%)	US (4.23%)	U (2.30%)
7	Transformers are the most popular toy line in the world	, (39.73%)	. (30.64%)	and (9.87%)	with (2.32%)	today (1.74%)

With this simple method we were able to generate the sentence "Transformers are the most popular toy line in the world". Interestingly, this indicates that GPT-2 has internalized some knowledge about the Transformers media franchise, which was created by two toy companies (Hasbro and Takara Tomy). We can also see the other possible continuations at each step, which shows the iterative nature of text generation. Unlike other tasks such as sequence classification where a single forward pass suffices to generate the predictions, with text generation we need to decode the output tokens one at a time.

Implementing greedy search wasn't too hard, but we'll want to use the built-in `generate()` function from 🤗 Transformers to explore more sophisticated decoding methods. To reproduce our simple example, let's make sure sampling is switched off (it's off by default, unless the specific configuration of the model you are loading the checkpoint from states otherwise) and specify the `max_new_tokens` for the number of newly generated tokens:

```
input_ids = tokenizer(input_txt, return_tensors="pt")["input_ids"].to(device)
output = model.generate(input_ids, max_new_tokens=n_steps, do_sample=False)
print(tokenizer.decode(output[0]))
```

```
Transformers are the most popular toy line in the world,
```

Now let's try something a bit more interesting: can we reproduce the unicorn story from OpenAI? As we did previously, we'll encode the prompt with the tokenizer, and we'll specify a larger value for `max_length` to generate a longer sequence of text:

```
max_length = 128
input_txt = """In a shocking finding, scientist discovered \
a herd of unicorns living in a remote, previously unexplored \
valley, in the Andes Mountains. Even more surprising to the \
researchers was the fact that the unicorns spoke perfect English.\n\n
"""
input_ids = tokenizer(input_txt, return_tensors="pt")["input_ids"].to(device)
output_greedy = model.generate(input_ids, max_length=max_length,
                               do_sample=False)
print(tokenizer.decode(output_greedy[0]))
```

```
In a shocking finding, scientist discovered a herd of unicorns living in a
remote, previously unexplored valley, in the Andes Mountains. Even more
surprising to the researchers was the fact that the unicorns spoke perfect
English.


The researchers, from the University of California, Davis, and the University of
Colorado, Boulder, were conducting a study on the Andean cloud forest, which is
home to the rare species of cloud forest trees.


The researchers were surprised to find that the unicorns were able to
communicate with each other, and even with humans.


The researchers were surprised to find that the unicorns were able
```

Well, the first few sentences are quite different from the OpenAI example and amusingly involve different universities being credited with the discovery! We can also see one of the main drawbacks with greedy search decoding: it tends to produce repetitive output sequences, which is certainly undesirable in a news article. This is a common problem with greedy search algorithms, which can fail to give you the optimal solution; in the context of decoding, they can miss word sequences whose overall probability is higher just because high-probability words happen to be preceded by low-probability ones.

Fortunately, we can do better—let's examine a popular method known as *beam search decoding*.

Although greedy search decoding is rarely used for text generation tasks that require diversity, it can be useful for producing short sequences like arithmetic where a deterministic and factually correct output is preferred.[4] For these tasks, you can condition GPT-2 by providing a few line-separated examples in the format `"5 + 8 => 13 \n 7 + 2 => 9 \n 1 + 0 =>"` as the input prompt.

Beam Search Decoding

Instead of decoding the token with the highest probability at each step, beam search keeps track of the top-*b* most probable next tokens, where *b* is referred to as the number of *beams* or *partial hypotheses*. The next set of beams are chosen by considering all possible next-token extensions of the existing set and selecting the *b* most likely extensions. The process is repeated until we reach the maximum length or an EOS

[4] N.S. Keskar et al., "CTRL: A Conditional Transformer Language Model for Controllable Generation", (2019).

token, and the most likely sequence is selected by ranking the b beams according to their log probabilities. An example of beam search is shown in Figure 5-4.

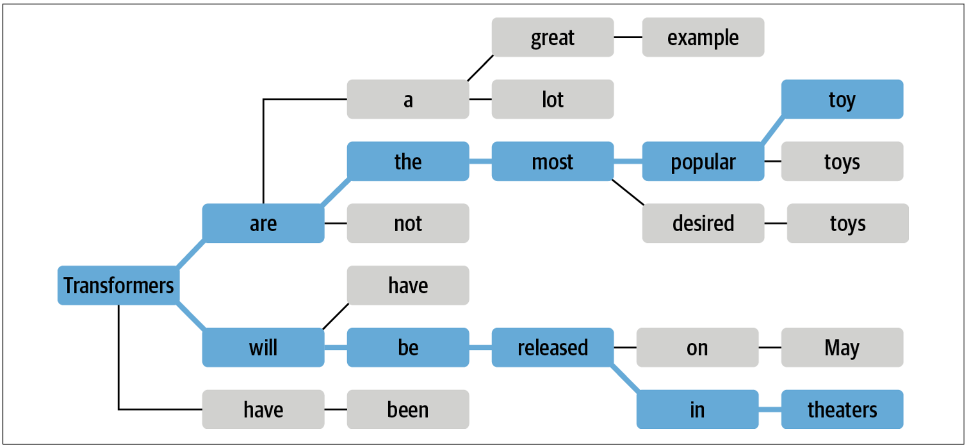


Figure 5-4. Beam search with two beams

Why do we score the sequences using log probabilities instead of the probabilities themselves? That calculating the overall probability of a sequence $P\left(y_1, y_2, ..., y_t \middle| \mathbf{x}\right)$ involves calculating a *product* of conditional probabilities $P\left(y_t \middle| y_{<t}, \mathbf{x}\right)$ is one reason. Since each conditional probability is typically a small number in the range $[0, 1]$, taking their product can lead to an overall probability that can easily underflow. This means that the computer can no longer precisely represent the result of the calculation. For example, suppose we have a sequence of $t = 1024$ tokens and generously assume that the probability for each token is 0.5. The overall probability for this sequence is an extremely small number:

```
0.5 ** 1024
```

```
5.562684646268003e-309
```

which leads to numerical instability as we run into underflow. We can avoid this by calculating a related term, the log probability. If we apply the logarithm to the joint and conditional probabilities, then with the help of the product rule for logarithms we get:

$$\log P\left(y_1, ...y_t \middle| \mathbf{x}\right) = \sum_{t=1}^{N} \log P\left(y_t \middle| y_{<t}, \mathbf{x}\right)$$

In other words, the product of probabilities we saw earlier becomes a sum of log probabilities, which is much less likely to run into numerical instabilities. For example, calculating the log probability of the same example as before gives:

```python
import numpy as np

sum([np.log(0.5)] * 1024)
```

```
-709.7827128933695
```

This is a number we can easily deal with, and this approach still works for much smaller numbers. Since we only want to compare relative probabilities, we can do this directly with log probabilities.

Let's calculate and compare the log probabilities of the texts generated by greedy and beam search to see if beam search can improve the overall probability. Since 🤗 Transformers models return the unnormalized logits for the next token given the input tokens, we first need to normalize the logits to create a probability distribution over the whole vocabulary for each token in the sequence. We then need to select only the token probabilities that were present in the sequence. The following function implements these steps:

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

def log_probs_from_logits(logits, labels):
    logp = F.log_softmax(logits, dim=-1)
    logp_label = torch.gather(logp, 2, labels.unsqueeze(2)).squeeze(-1)
    return logp_label
```

This gives us the log probability for a single token, so to get the total log probability of a sequence we just need to sum the log probabilities for each token:

```python
def sequence_logprob(model, labels, input_len=0):
    with torch.no_grad():
        output = model(labels)
        log_probs = log_probs_from_logits(
            output.logits[:, :-1, :], labels[:, 1:])
        seq_log_prob = torch.sum(log_probs[:, input_len:])
    return seq_log_prob.cpu().numpy()
```

Note that we ignore the log probabilities of the input sequence because they are not generated by the model. We can also see that it is important to align the logits and the labels; since the model predicts the next token, we do not get a logit for the first label, and we don't need the last logit because we don't have a ground truth token for it.

Let's use these functions to first calculate the sequence log probability of the greedy decoder on the OpenAI prompt:

```python
logp = sequence_logprob(model, output_greedy, input_len=len(input_ids[0]))
print(tokenizer.decode(output_greedy[0]))
print(f"\nlog-prob: {logp:.2f}")
```

```
In a shocking finding, scientist discovered a herd of unicorns living in a
remote, previously unexplored valley, in the Andes Mountains. Even more
surprising to the researchers was the fact that the unicorns spoke perfect
English.
```

```
The researchers, from the University of California, Davis, and the University of
Colorado, Boulder, were conducting a study on the Andean cloud forest, which is
home to the rare species of cloud forest trees.


The researchers were surprised to find that the unicorns were able to
communicate with each other, and even with humans.


The researchers were surprised to find that the unicorns were able

log-prob: -87.43
```

Now let's compare this to a sequence that is generated with beam search. To activate beam search with the `generate()` function we just need to specify the number of beams with the `num_beams` parameter. The more beams we choose, the better the result potentially gets; however, the generation process becomes much slower since we generate parallel sequences for each beam:

```
output_beam = model.generate(input_ids, max_length=max_length, num_beams=5,
                             do_sample=False)
logp = sequence_logprob(model, output_beam, input_len=len(input_ids[0]))
print(tokenizer.decode(output_beam[0]))
print(f"\nlog-prob: {logp:.2f}")
```

```
In a shocking finding, scientist discovered a herd of unicorns living in a
remote, previously unexplored valley, in the Andes Mountains. Even more
surprising to the researchers was the fact that the unicorns spoke perfect
English.


The discovery of the unicorns was made by a team of scientists from the
University of California, Santa Cruz, and the National Geographic Society.


The scientists were conducting a study of the Andes Mountains when they
discovered a herd of unicorns living in a remote, previously unexplored valley,
in the Andes Mountains. Even more surprising to the researchers was the fact
that the unicorns spoke perfect English

log-prob: -55.23
```

We can see that we get a better log probability (higher is better) with beam search than we did with simple greedy decoding. However, we can see that beam search also suffers from repetitive text. One way to address this is to impose an *n*-gram penalty with the `no_repeat_ngram_size` parameter that tracks which *n*-grams have been seen and sets the next token probability to zero if it would produce a previously seen *n*-gram:

```
output_beam = model.generate(input_ids, max_length=max_length, num_beams=5,
                             do_sample=False, no_repeat_ngram_size=2)
logp = sequence_logprob(model, output_beam, input_len=len(input_ids[0]))
print(tokenizer.decode(output_beam[0]))
print(f"\nlog-prob: {logp:.2f}")
```

```
In a shocking finding, scientist discovered a herd of unicorns living in a
remote, previously unexplored valley, in the Andes Mountains. Even more
surprising to the researchers was the fact that the unicorns spoke perfect
English.


The discovery was made by a team of scientists from the University of
California, Santa Cruz, and the National Geographic Society.

According to a press release, the scientists were conducting a survey of the
area when they came across the herd. They were surprised to find that they were
able to converse with the animals in English, even though they had never seen a
unicorn in person before. The researchers were

log-prob: -93.12
```

This isn't too bad! We've managed to stop the repetitions, and we can see that despite producing a lower score, the text remains coherent. Beam search with *n*-gram penalty is a good way to find a trade-off between focusing on high-probability tokens (with beam search) while reducing repetitions (with *n*-gram penalty), and it's commonly used in applications such as summarization or machine translation where factual correctness is important. When factual correctness is less important than the diversity of generated output, for instance in open-domain chitchat or story generation, another alternative to reduce repetitions while improving diversity is to use sampling. Let's round out our exploration of text generation by examining a few of the most common sampling methods.

Sampling Methods

The simplest sampling method is to randomly sample from the probability distribution of the model's outputs over the full vocabulary at each timestep:

$$P\left(y_t = w_i \middle| y_{<t}, \mathbf{x}\right) = \mathrm{softmax}\left(z_{t,i}\right) = \frac{\exp\left(z_{t,i}\right)}{\sum_{j=1}^{|V|} \exp\left(z_{t,j}\right)}$$

where $|V|$ denotes the cardinality of the vocabulary. We can easily control the diversity of the output by adding a temperature parameter T that rescales the logits before taking the softmax:

$$P\left(y_t = w_i \middle| y_{<t}, \mathbf{x}\right) = \frac{\exp\left(z_{t,i}/T\right)}{\Sigma_{j=1}^{|V|} \exp\left(z_{t,j}/T\right)}$$

By tuning T we can control the shape of the probability distribution.[5] When $T \ll 1$, the distribution becomes peaked around the origin and the rare tokens are suppressed. On the other hand, when $T \gg 1$, the distribution flattens out and each token becomes equally likely. The effect of temperature on token probabilities is shown in Figure 5-5.

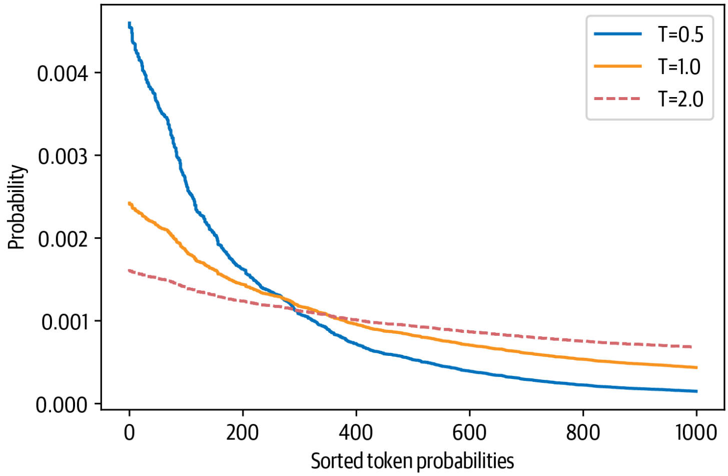


Figure 5-5. Distribution of randomly generated token probabilities for three selected temperatures

To see how we can use temperature to influence the generated text, let's sample with $T = 2$ by setting the `temperature` parameter in the `generate()` function (we'll explain the meaning of the `top_k` parameter in the next section):

```
output_temp = model.generate(input_ids, max_length=max_length, do_sample=True,
                             temperature=2.0, top_k=0)
print(tokenizer.decode(output_temp[0]))
```

```
In a shocking finding, scientist discovered a herd of unicorns living in a
remote, previously unexplored valley, in the Andes Mountains. Even more
surprising to the researchers was the fact that the unicorns spoke perfect
English.


While the station aren protagonist receive Pengala nostalgiates tidbitRegarding
```

[5] If you know some physics, you may recognize a striking resemblance to the Boltzmann distribution.

```
Jenny loclonju AgreementCON irrational �rite Continent seaf A jer Turner
Dorbecue WILL Pumpkin mere Thatvernuildagain YoAniamond disse *
Runewitingkusstemprop});b zo coachinginventorymodules deflation press
Vaticanpres Wrestling chargesThingsctureddong Ty physician PET KimBi66 graz Oz
at aff da temporou MD6 radi iter
```

We can clearly see that a high temperature has produced mostly gibberish; by accentuating the rare tokens, we've caused the model to create strange grammar and quite a few made-up words! Let's see what happens if we cool down the temperature:

```
output_temp = model.generate(input_ids, max_length=max_length, do_sample=True,
                             temperature=0.5, top_k=0)
print(tokenizer.decode(output_temp[0]))
```

```
In a shocking finding, scientist discovered a herd of unicorns living in a
remote, previously unexplored valley, in the Andes Mountains. Even more
surprising to the researchers was the fact that the unicorns spoke perfect
English.


The scientists were searching for the source of the mysterious sound, which was
making the animals laugh and cry.


The unicorns were living in a remote valley in the Andes mountains

'When we first heard the noise of the animals, we thought it was a lion or a
tiger,' said Luis Guzman, a researcher from the University of Buenos Aires,
Argentina.


'But when
```

This is significantly more coherent, and even includes a quote from yet another university being credited with the discovery! The main lesson we can draw from temperature is that it allows us to control the quality of the samples, but there's always a trade-off between coherence (low temperature) and diversity (high temperature) that one has to tune to the use case at hand.

Another way to adjust the trade-off between coherence and diversity is to truncate the distribution of the vocabulary. This allows us to adjust the diversity freely with the temperature, but in a more limited range that excludes words that would be too strange in the context (i.e., low-probability words). There are two main ways to do this: top-*k* and nucleus (or top-*p*) sampling. Let's take a look.

Top-k and Nucleus Sampling

Top-*k* and nucleus (top-*p*) sampling are two popular alternatives or extensions to using temperature. In both cases, the basic idea is to restrict the number of possible tokens we can sample from at each timestep. To see how this works, let's first visualize the cumulative probability distribution of the model's outputs at $T = 1$ as seen in Figure 5-6.

Let's tease apart these plots, since they contain a lot of information. In the upper plot we can see a histogram of the token probabilities. It has a peak around 10^{-8} and a second, smaller peak around 10^{-4}, followed by a sharp drop with just a handful of tokens occurring with probability between 10^{-2} and 10^{-1}. Looking at this diagram, we can see that the probability of picking the token with the highest probability (the isolated bar at 10^{-1}) is 1 in 10.

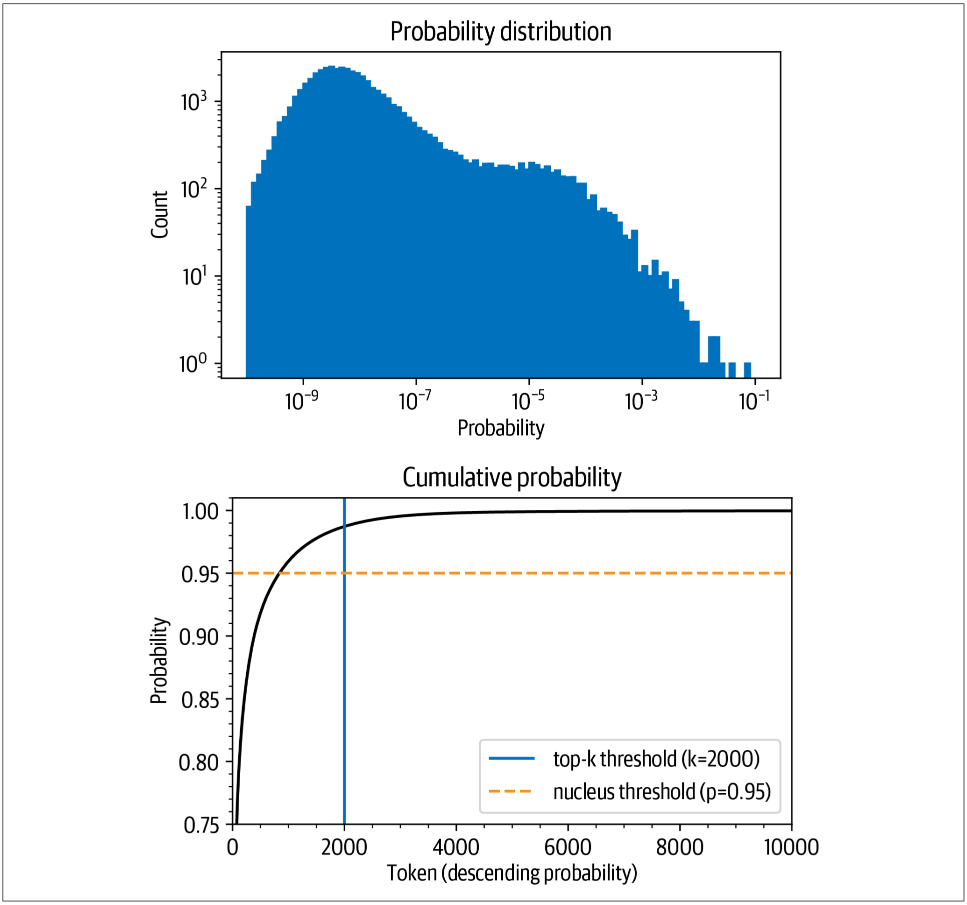


Figure 5-6. Probability distribution of next token prediction (upper) and cumulative distribution of descending token probabilities (lower)

In the lower plot, we've ordered the tokens by descending probability and calculated the cumulative sum of the first 10,000 tokens (in total, there are 50,257 tokens in GPT-2's vocabulary). The curved line represents the probability of picking any of the preceding tokens. For example, there is roughly a 96% chance of picking any of the 1,000 tokens with the highest probability. We see that the probability rises quickly above 90% but saturates to close to 100% only after several thousand tokens. The plot shows that there is a 1 in 100 chance of not picking any of the tokens that are not even in the top 2,000.

Although these numbers might appear small at first sight, they become important because we sample once per token when generating text. So even if there is only a 1 in 100 or 1,000 chance, if we sample hundreds of times there is a significant chance of picking an unlikely token at some point—and picking such tokens when sampling can badly influence the quality of the generated text. For this reason, we generally want to avoid these very unlikely tokens. This is where top-*k* and top-*p* sampling come into play.

The idea behind top-*k* sampling is to avoid the low-probability choices by only sampling from the *k* tokens with the highest probability. This puts a fixed cut on the long tail of the distribution and ensures that we only sample from likely choices. Going back to Figure 5-6, top-*k* sampling is equivalent to defining a vertical line and sampling from the tokens on the left. Again, the `generate()` function provides an easy method to achieve this with the `top_k` argument:

```python
output_topk = model.generate(input_ids, max_length=max_length, do_sample=True,
                             top_k=50)
print(tokenizer.decode(output_topk[0]))
```

```
In a shocking finding, scientist discovered a herd of unicorns living in a
remote, previously unexplored valley, in the Andes Mountains. Even more
surprising to the researchers was the fact that the unicorns spoke perfect
English.


The wild unicorns roam the Andes Mountains in the region of Cajamarca, on the
border with Argentina (Picture: Alamy/Ecole Nationale Supérieure d'Histoire
Naturelle)

The researchers came across about 50 of the animals in the valley. They had
lived in such a remote and isolated area at that location for nearly a thousand
years that
```

This is arguably the most human-looking text we've generated so far. But how do we choose *k*? The value of *k* is chosen manually and is the same for each choice in the sequence, independent of the actual output distribution. We can find a good value for *k* by looking at some text quality metrics, which we will explore in the next chapter—but that fixed cutoff might not be very satisfactory.

An alternative is to use a *dynamic* cutoff. With nucleus or top-*p* sampling, instead of choosing a fixed cutoff value, we set a condition of when to cut off. This condition is when a certain probability mass in the selection is reached. Let's say we set that value to 95%. We then order all tokens in descending order by probability and add one token after another from the top of the list until the sum of the probabilities of the selected tokens is 95%. Returning to Figure 5-6, the value for *p* defines a horizontal line on the cumulative sum of probabilities plot, and we sample only from tokens below the line. Depending on the output distribution, this could be just one (very likely) token or a hundred (more equally likely) tokens. At this point, you are probably not surprised that the `generate()` function also provides an argument to activate top-*p* sampling. Let's try it out:

```
output_topp = model.generate(input_ids, max_length=max_length, do_sample=True,
                             top_p=0.90)
print(tokenizer.decode(output_topp[0]))
```

```
In a shocking finding, scientist discovered a herd of unicorns living in a
remote, previously unexplored valley, in the Andes Mountains. Even more
surprising to the researchers was the fact that the unicorns spoke perfect
English.


The scientists studied the DNA of the animals and came to the conclusion that
the herd are descendants of a prehistoric herd that lived in Argentina about
50,000 years ago.


According to the scientific analysis, the first humans who migrated to South
America migrated into the Andes Mountains from South Africa and Australia, after
the last ice age had ended.


Since their migration, the animals have been adapting to
```

Top-*p* sampling has also produced a coherent story, and this time with a new twist about migrations from Australia to South America. You can even combine the two sampling approaches to get the best of both worlds. Setting `top_k=50` and `top_p=0.9` corresponds to the rule of choosing tokens with a probability mass of 90%, from a pool of at most 50 tokens.

We can also apply beam search when we use sampling. Instead of selecting the next batch of candidate tokens greedily, we can sample them and build up the beams in the same way.

Which Decoding Method Is Best?

Unfortunately, there is no universally "best" decoding method. Which approach is best will depend on the nature of the task you are generating text for. If you want your model to perform a precise task like arithmetic or providing an answer to a specific question, then you should lower the temperature or use deterministic methods like greedy search in combination with beam search to guarantee getting the most likely answer. If you want the model to generate longer texts and even be a bit creative, then you should switch to sampling methods and increase the temperature or use a mix of top-*k* and nucleus sampling.

Conclusion

In this chapter we looked at text generation, which is a very different task from the NLU tasks we encountered previously. Generating text requires at least one forward pass per generated token, and even more if we use beam search. This makes text generation computationally demanding, and one needs the right infrastructure to run a text generation model at scale. In addition, a good decoding strategy that transforms the model's output probabilities into discrete tokens can improve the text quality. Finding the best decoding strategy requires some experimentation and a subjective evaluation of the generated texts.

In practice, however, we don't want to make these decisions based on gut feeling alone! Like with other NLP tasks, we should choose a model performance metric that reflects the problem we want to solve. Unsurprisingly, there are a wide range of choices, and we will encounter the most common ones in the next chapter, where we have a look at how to train and evaluate a model for text summarization. Or, if you can't wait to learn how to train a GPT-type model from scratch, you can skip right to Chapter 10, where we collect a large dataset of code and then train an autoregressive language model on it.

CHAPTER 6

Summarization

At one point or another, you've probably needed to summarize a document, be it a research article, a financial earnings report, or a thread of emails. If you think about it, this requires a range of abilities, such as understanding long passages, reasoning about the contents, and producing fluent text that incorporates the main topics from the original document. Moreover, accurately summarizing a news article is very different from summarizing a legal contract, so being able to do so requires a sophisticated degree of domain generalization. For these reasons, text summarization is a difficult task for neural language models, including transformers. Despite these challenges, text summarization offers the prospect for domain experts to significantly speed up their workflows and is used by enterprises to condense internal knowledge, summarize contracts, automatically generate content for social media releases, and more.

To help you understand the challenges involved, this chapter will explore how we can leverage pretrained transformers to summarize documents. Summarization is a classic sequence-to-sequence (seq2seq) task with an input text and a target text. As we saw in Chapter 1, this is where encoder-decoder transformers excel.

In this chapter we will build our own encoder-decoder model to condense dialogues between several people into a crisp summary. But before we get to that, let's begin by taking a look at one of the canonical datasets for summarization: the CNN/DailyMail corpus.

The CNN/DailyMail Dataset

The CNN/DailyMail dataset consists of around 300,000 pairs of news articles and their corresponding summaries, composed from the bullet points that CNN and the DailyMail attach to their articles. An important aspect of the dataset is that the summaries are *abstractive* and not *extractive*, which means that they consist of new sentences instead of simple excerpts. The dataset is available on the Hub; we'll use version 3.0.0, which is a nonanonymized version set up for summarization. We can select versions in a similar manner as splits, we saw in Chapter 4, with a `version` keyword. So let's dive in and have a look at it:

```
from datasets import load_dataset

dataset = load_dataset("cnn_dailymail", version="3.0.0")
print(f"Features: {dataset['train'].column_names}")
```

```
Features: ['article', 'highlights', 'id']
```

The dataset has three columns: `article`, which contains the news articles, `highlights` with the summaries, and `id` to uniquely identify each article. Let's look at an excerpt from an article:

```
sample = dataset["train"][1]
print(f"""
Article (excerpt of 500 characters, total length: {len(sample["article"])}):
""")
print(sample["article"][:500])
print(f'\nSummary (length: {len(sample["highlights"])}):')
print(sample["highlights"])
```

```
Article (excerpt of 500 characters, total length: 3192):

(CNN) -- Usain Bolt rounded off the world championships Sunday by claiming his
third gold in Moscow as he anchored Jamaica to victory in the men's 4x100m
relay. The fastest man in the world charged clear of United States rival Justin
Gatlin as the Jamaican quartet of Nesta Carter, Kemar Bailey-Cole, Nickel
Ashmeade and Bolt won in 37.36 seconds. The U.S finished second in 37.56 seconds
with Canada taking the bronze after Britain were disqualified for a faulty
handover. The 26-year-old Bolt has n

Summary (length: 180):
Usain Bolt wins third gold of world championship .
Anchors Jamaica to 4x100m relay victory .
Eighth gold at the championships for Bolt .
Jamaica double up in women's 4x100m relay .
```

We see that the articles can be very long compared to the target summary; in this particular case the difference is 17-fold. Long articles pose a challenge to most transformer models since the context size is usually limited to 1,000 tokens or so, which is equivalent to a few paragraphs of text. The standard, yet crude way to deal with this for summarization is to simply truncate the texts beyond the model's context size. Obviously there could be important information for the summary toward the end of the text, but for now we need to live with this limitation of the model architectures.

Text Summarization Pipelines

Let's see how a few of the most popular transformer models for summarization perform by first looking qualitatively at the outputs for the preceding example. Although the model architectures we will be exploring have varying maximum input sizes, let's restrict the input text to 2,000 characters to have the same input for all models and thus make the outputs more comparable:

```
sample_text = dataset["train"][1]["article"][:2000]
# We'll collect the generated summaries of each model in a dictionary
summaries = {}
```

A convention in summarization is to separate the summary sentences by a newline. We could add a newline token after each full stop, but this simple heuristic would fail for strings like "U.S." or "U.N." The Natural Language Toolkit (NLTK) package includes a more sophisticated algorithm that can differentiate the end of a sentence from punctuation that occurs in abbreviations:

```
import nltk
from nltk.tokenize import sent_tokenize

nltk.download("punkt")
```

```
string = "The U.S. are a country. The U.N. is an organization."
sent_tokenize(string)
```

```
['The U.S. are a country.', 'The U.N. is an organization.']
```

In the following sections we will load several large models. If you run out of memory, you can either replace the large models with smaller checkpoints (e.g., "gpt", "t5-small") or skip this section and jump to "Evaluating PEGASUS on the CNN/DailyMail Dataset" on page 154.

Summarization Baseline

A common baseline for summarizing news articles is to simply take the first three sentences of the article. With NLTK's sentence tokenizer, we can easily implement such a baseline:

```
def three_sentence_summary(text):
    return "\n".join(sent_tokenize(text)[:3])
```

```
summaries["baseline"] = three_sentence_summary(sample_text)
```

GPT-2

We've already seen in Chapter 5 how GPT-2 can generate text given some prompt. One of the model's surprising features is that we can also use it to generate summaries by simply appending "TL;DR" at the end of the input text. The expression "TL;DR" (too long; didn't read) is often used on platforms like Reddit to indicate a short version of a long post. We will start our summarization experiment by re-creating the procedure of the original paper with the `pipeline()` function from 🤗 Transformers.[1] We create a text generation pipeline and load the large GPT-2 model:

```python
from transformers import pipeline, set_seed

set_seed(42)
pipe = pipeline("text-generation", model="gpt2-xl")
gpt2_query = sample_text + "\nTL;DR:\n"
pipe_out = pipe(gpt2_query, max_length=512, clean_up_tokenization_spaces=True)
summaries["gpt2"] = "\n".join(
    sent_tokenize(pipe_out[0]["generated_text"][len(gpt2_query) :]))
```

Here we just store the summaries of the generated text by slicing off the input query and keep the result in a Python dictionary for later comparison.

T5

Next let's try the T5 transformer. As we saw in Chapter 3, the developers of this model performed a comprehensive study of transfer learning in NLP and found they could create a universal transformer architecture by formulating all tasks as text-to-text tasks. The T5 checkpoints are trained on a mixture of unsupervised data (to reconstruct masked words) and supervised data for several tasks, including summarization. These checkpoints can thus be directly used to perform summarization without fine-tuning by using the same prompts used during pretraining. In this framework, the input format for the model to summarize a document is `"summarize: <ARTICLE>"`, and for translation it looks like `"translate English to German: <TEXT>"`. As shown in Figure 6-1, this makes T5 extremely versatile and allows you to solve many tasks with a single model.

We can directly load T5 for summarization with the `pipeline()` function, which also takes care of formatting the inputs in the text-to-text format so we don't need to prepend them with `"summarize"`:

```python
pipe = pipeline("summarization", model="t5-large")
pipe_out = pipe(sample_text)
summaries["t5"] = "\n".join(sent_tokenize(pipe_out[0]["summary_text"]))
```

[1] A. Radford et al., "Language Models Are Unsupervised Multitask Learners", OpenAI (2019).

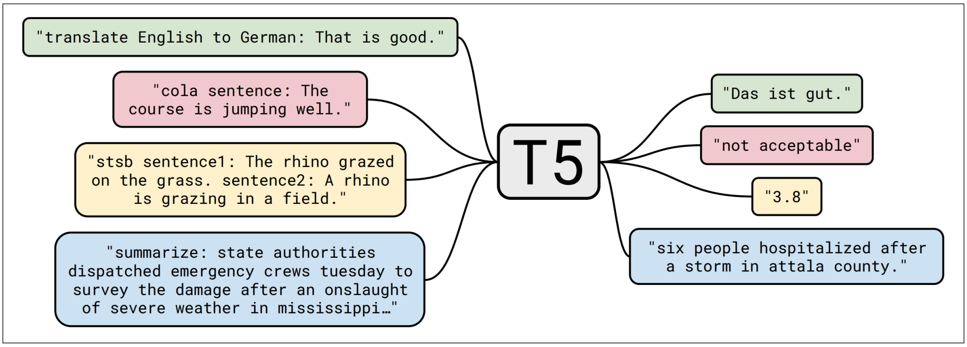

"translate English to German: That is good."
"cola sentence: The course is jumping well."
"stsb sentence1: The rhino grazed on the grass. sentence2: A rhino is grazing in a field."
"summarize: state authorities dispatched emergency crews tuesday to survey the damage after an onslaught of severe weather in mississippi…"
T5
"Das ist gut."
"not acceptable"
"3.8"
"six people hospitalized after a storm in attala county."

Figure 6-1. Diagram of T5's text-to-text framework (courtesy of Colin Raffel); besides translation and summarization, the CoLA (linguistic acceptability) and STSB (semantic similarity) tasks are shown

BART

BART also uses an encoder-decoder architecture and is trained to reconstruct corrupted inputs. It combines the pretraining schemes of BERT and GPT-2.[2] We'll use the `facebook/bart-large-ccn` checkpoint, which has been specifically fine-tuned on the CNN/DailyMail dataset:

```
pipe = pipeline("summarization", model="facebook/bart-large-cnn")
pipe_out = pipe(sample_text)
summaries["bart"] = "\n".join(sent_tokenize(pipe_out[0]["summary_text"]))
```

PEGASUS

Like BART, PEGASUS is an encoder-decoder transformer.[3] As shown in Figure 6-2, its pretraining objective is to predict masked sentences in multisentence texts. The authors argue that the closer the pretraining objective is to the downstream task, the more effective it is. With the aim of finding a pretraining objective that is closer to summarization than general language modeling, they automatically identified, in a very large corpus, sentences containing most of the content of their surrounding paragraphs (using summarization evaluation metrics as a heuristic for content overlap) and pretrained the PEGASUS model to reconstruct these sentences, thereby obtaining a state-of-the-art model for text summarization.

[2] M. Lewis et al., "BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension", (2019).

[3] J. Zhang et al., "PEGASUS: Pre-Training with Extracted Gap-Sentences for Abstractive Summarization", (2019).

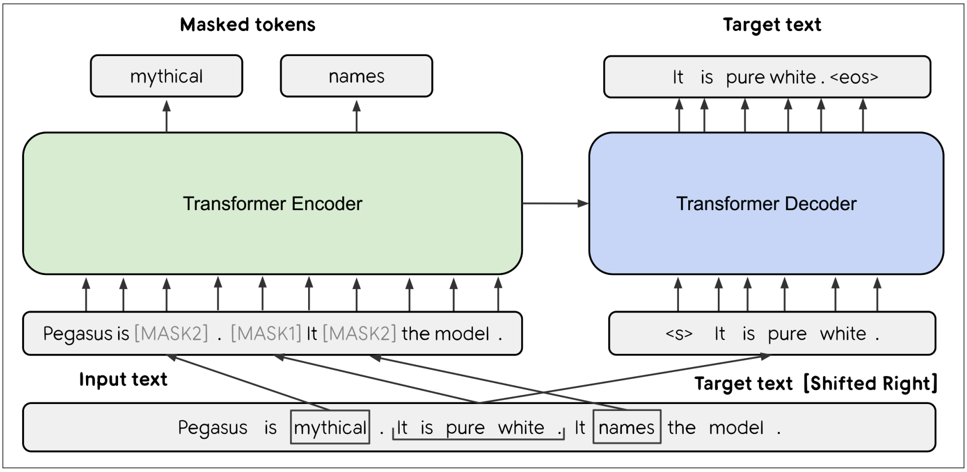

Masked tokens
mythical
names
Target text
It is pure white . <eos>
Transformer Encoder
Transformer Decoder
Pegasus is [MASK2] . [MASK1] It [MASK2] the model .
<s> It is pure white .
Input text
Target text [Shifted Right]
Pegasus is mythical . It is pure white . It names the model .

Figure 6-2. Diagram of PEGASUS architecture (courtesy of Jingqing Zhang et al.)

This model has a special token for newlines, which is why we don't need the `sent_tokenize()` function:

```
pipe = pipeline("summarization", model="google/pegasus-cnn_dailymail")
pipe_out = pipe(sample_text)
summaries["pegasus"] = pipe_out[0]["summary_text"].replace(" .<n>", ".\n")
```

Comparing Different Summaries

Now that we have generated summaries with four different models, let's compare the results. Keep in mind that one model has not been trained on the dataset at all (GPT-2), one model has been fine-tuned on this task among others (T5), and two models have exclusively been fine-tuned on this task (BART and PEGASUS). Let's have a look at the summaries these models have generated:

```
print("GROUND TRUTH")
print(dataset["train"][1]["highlights"])
print("")

for model_name in summaries:
    print(model_name.upper())
    print(summaries[model_name])
    print("")
```

```
GROUND TRUTH
Usain Bolt wins third gold of world championship .
Anchors Jamaica to 4x100m relay victory .
Eighth gold at the championships for Bolt .
Jamaica double up in women's 4x100m relay .

BASELINE
```