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.

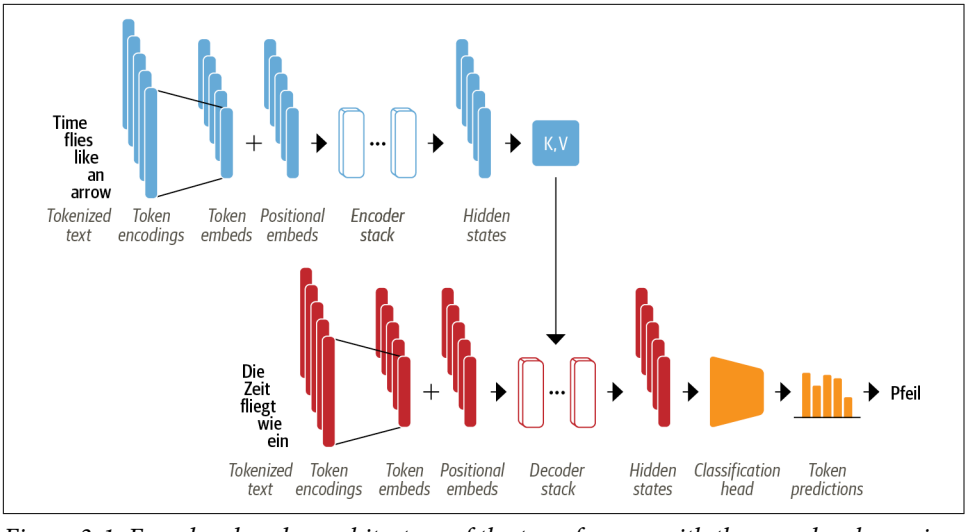


Figure 3-1. Encoder-decoder architecture of the transformer, with the encoder shown in the upper half of the figure and the decoder in the lower half

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 autocomplete 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.

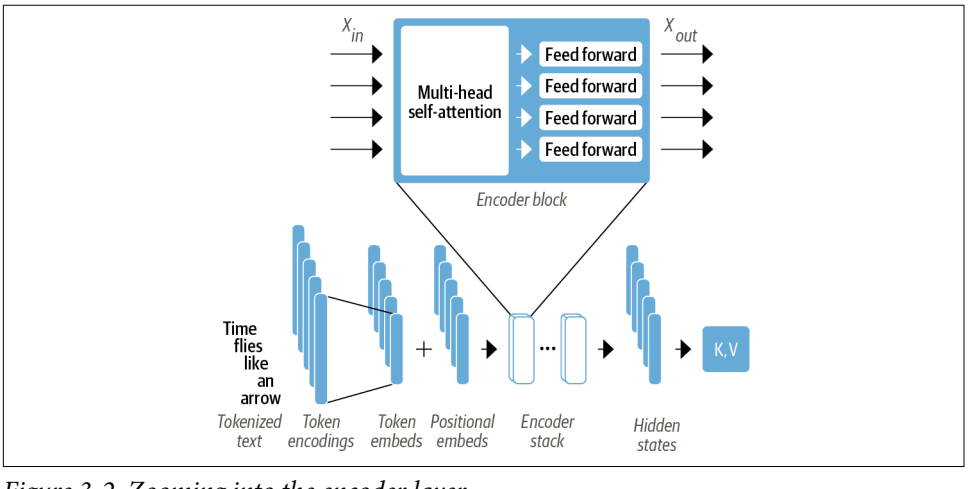


Figure 3-2. Zooming into the encoder layer

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).

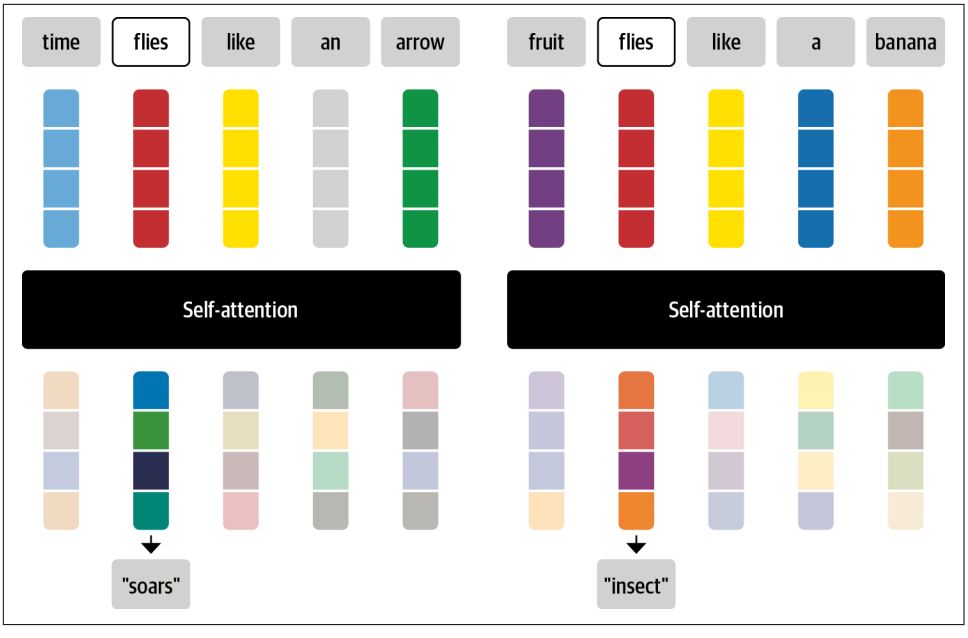


Figure 3-3. Diagram showing how self-attention updates raw token embeddings (upper) into contextualized embeddings (lower) to create representations that incorporate information from the whole sequence

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:

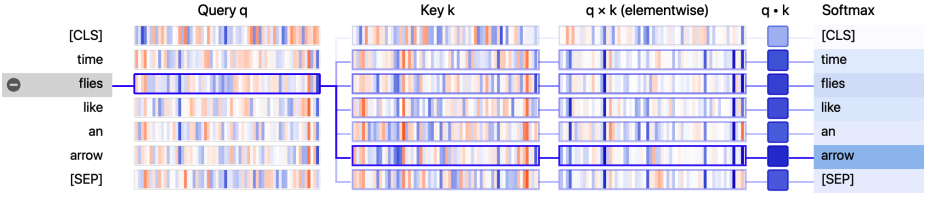
1. Project each token embedding into three vectors called *query*, *key*, and *value*.
2. Compute attention scores. We determine how much the query and key vectors relate to each other using a *similarity function*. As the name suggests, the similarity function for scaled dot-product attention is the 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.

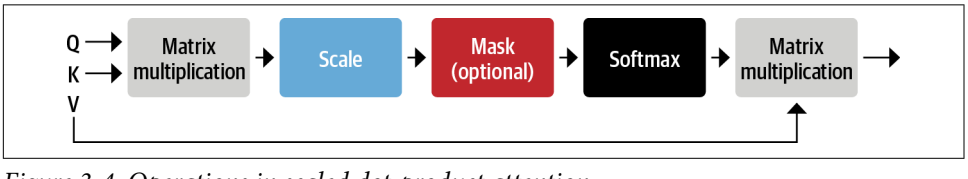


Figure 3-4. Operations in scaled dot-product attention

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.

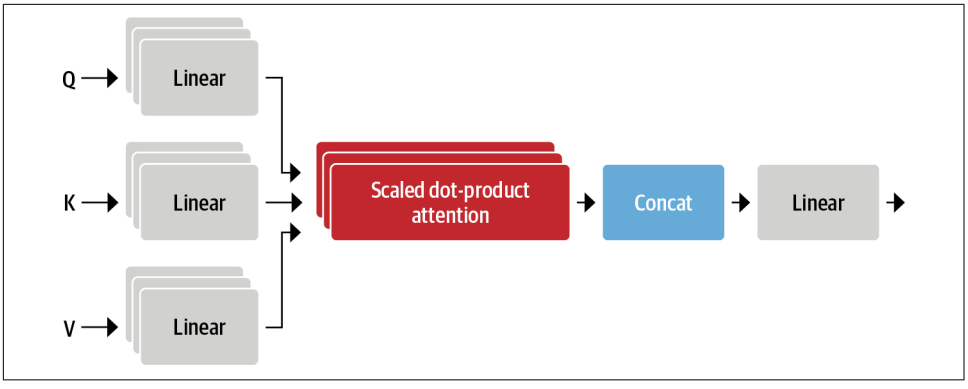


Figure 3-5. Multi-head attention

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
```

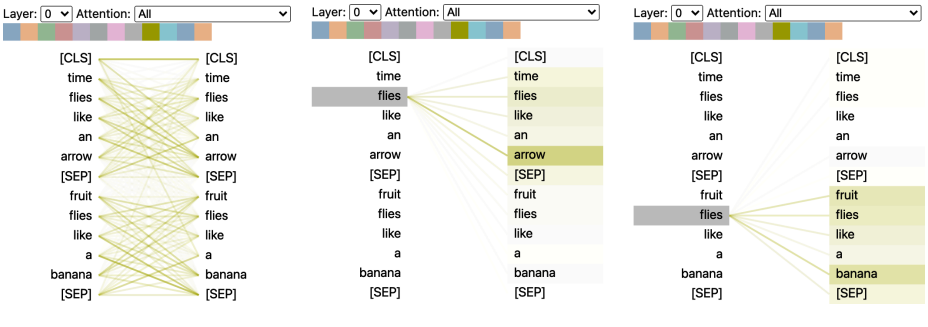
Notice that the concatenated output from the attention heads is also fed through a final linear layer to produce an output tensor of shape `[batch_size, seq_len, hidden_dim]` that is suitable for the feed-forward network downstream. To confirm, let's see if the multi-head attention layer produces the 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])
```



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.

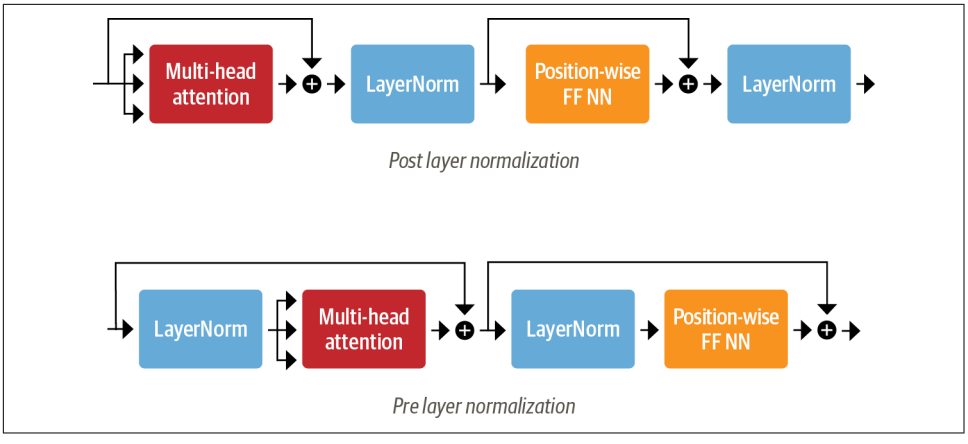


Figure 3-6. Different arrangements of layer normalization in a transformer encoder layer

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>)
```

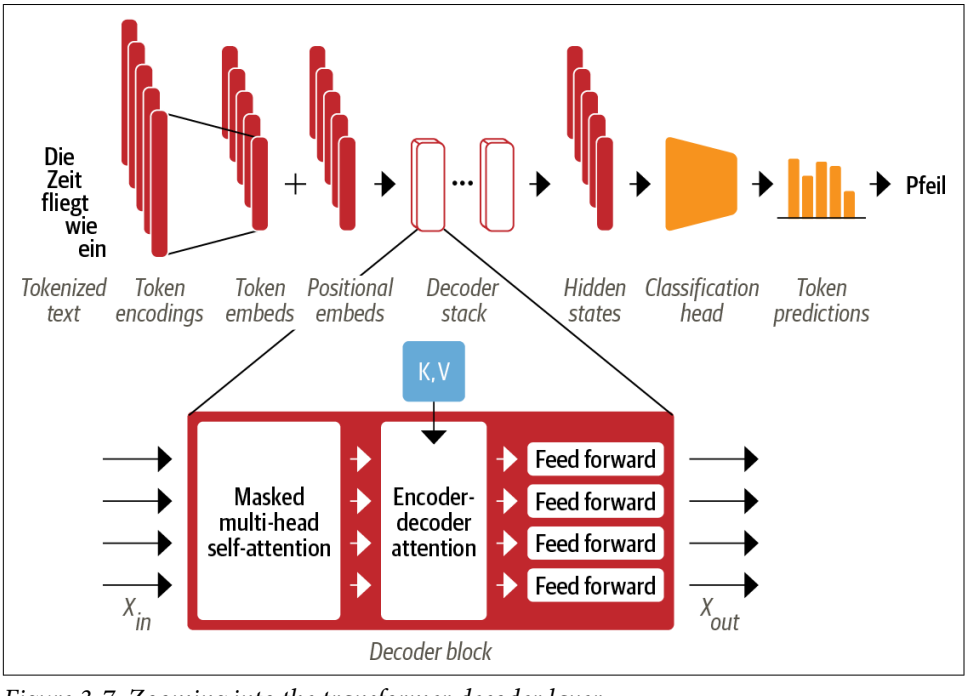


Figure 3-7. Zooming into the transformer decoder layer

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.

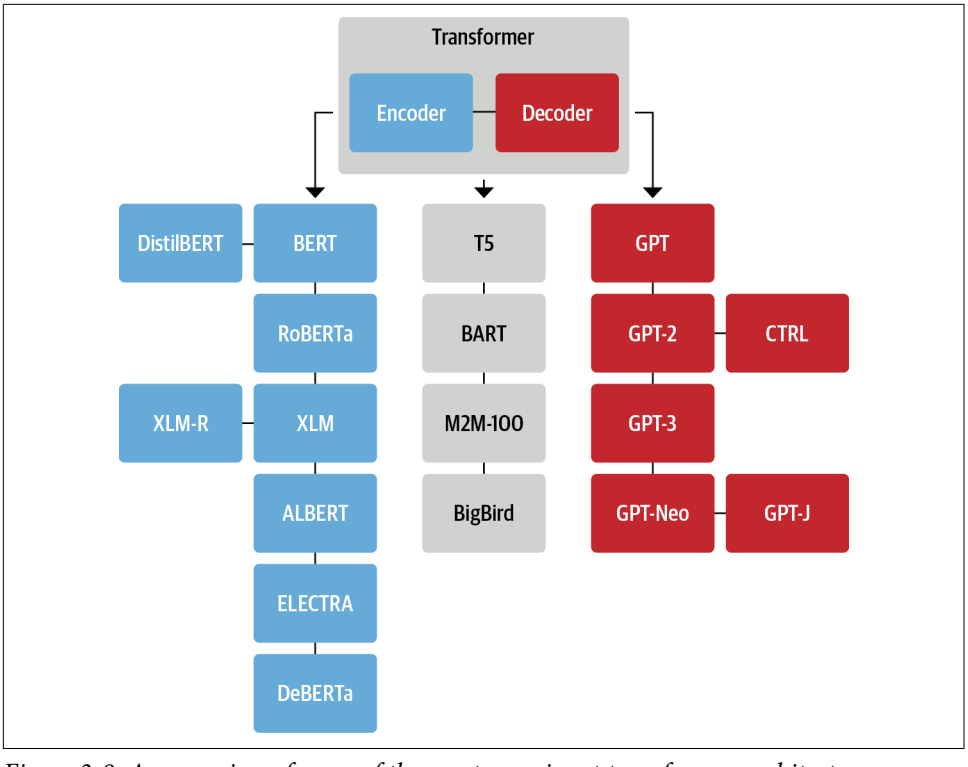


Figure 3-8. An overview of some of the most prominent transformer architectures

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).

transformer encoder model. We added embedding layers for tokens and positional information, we built in a feed-forward layer to complement the attention heads, and finally we added a classification head to the model body to make predictions. We also had a look at the decoder side of the Transformer architecture, and concluded the chapter with an overview of the most important model architectures.

Now that you have a better understanding of the underlying principles, let's go beyond simple classification and build a multilingual named entity recognition model.

CHAPTER 7

Question Answering

Whether you're a researcher, analyst, or data scientist, chances are that at some point you've needed to wade through oceans of documents to find the information you're looking for. To make matters worse, you're constantly reminded by Google and Bing that there exist better ways to search! For instance, if we search for "When did Marie Curie win her first Nobel Prize?" on Google, we immediately get the correct answer of "1903," as illustrated in Figure 7-1.

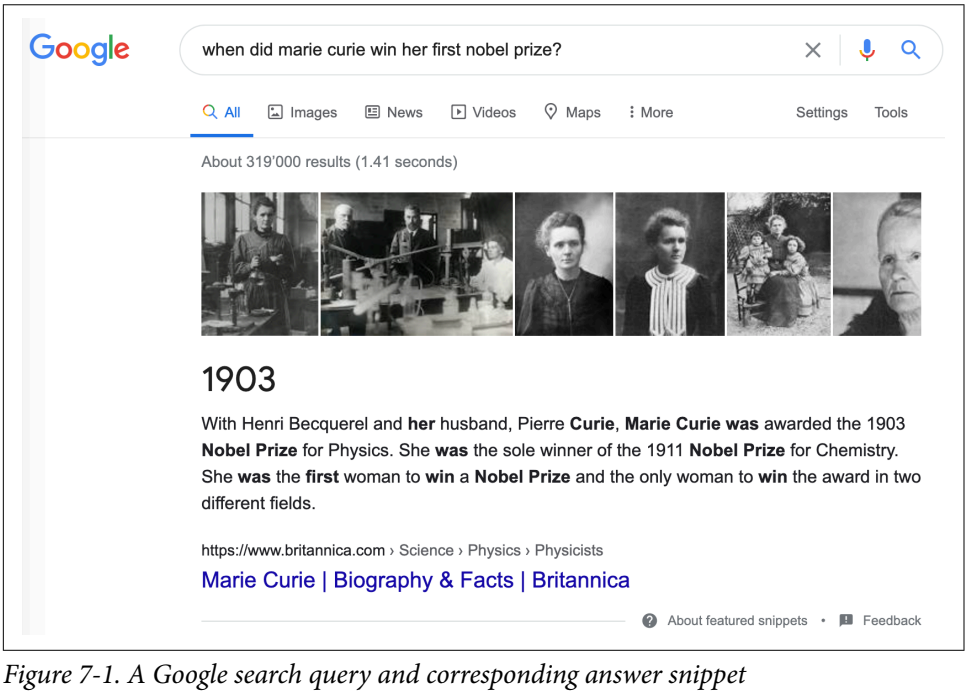


Figure 7-1. A Google search query and corresponding answer snippet

In this example, Google first retrieved around 319,000 documents that were relevant to the query, and then performed an additional processing step to extract the answer snippet with the corresponding passage and web page. It's not hard to see why these answer snippets are useful. For example, if we search for a trickier question like "Which guitar tuning is the best?" Google doesn't provide an answer, and instead we have to click on one of the web pages returned by the search engine to find it ourselves.[1]

The general approach behind this technology is called *question answering* (QA). There are many flavors of QA, but the most common is *extractive QA*, which involves questions whose answer can be identified as a span of text in a document, where the document might be a web page, legal contract, or news article. The two-stage process of first retrieving relevant documents and then extracting answers from them is also the basis for many modern QA systems, including semantic search engines, intelligent assistants, and automated information extractors. In this chapter, we'll apply this process to tackle a common problem facing ecommerce websites: helping consumers answer specific queries to evaluate a product. We'll see that customer reviews can be used as a rich and challenging source of information for QA, and along the way we'll learn how transformers act as powerful *reading comprehension* models that can extract meaning from text. Let's begin by fleshing out the use case.

This chapter focuses on extractive QA, but other forms of QA may be more suitable for your use case. For example, *community QA* involves gathering question-answer pairs that are generated by users on forums like Stack Overflow, and then using semantic similarity search to find the closest matching answer to a new question. There is also *long-form QA*, which aims to generate complex paragraph-length answers to open-ended questions like "Why is the sky blue?" Remarkably, it is also possible to do QA over tables, and transformer models like TAPAS can even perform aggregations to produce the final answer!

Building a Review-Based QA System

If you've ever purchased a product online, you probably relied on customer reviews to help inform your decision. These reviews can often help answer specific questions like "Does this guitar come with a strap?" or "Can I use this camera at night?" that may be hard to answer from the product description alone. However, popular products can have hundreds to thousands of reviews, so it can be a major drag to find one that is relevant. One alternative is to post your question on the community QA

1 Although, in this particular case, everyone agrees that Drop C is the best guitar tuning.

platforms provided by websites like Amazon, but it usually takes days to get an answer (if you get one at all). Wouldn't it be nice if we could get an immediate answer, like in the Google example from Figure 7-1? Let's see if we can do this using transformers!

The Dataset

To build our QA system we'll use the SubjQA dataset,[2] which consists of more than 10,000 customer reviews in English about products and services in six domains: TripAdvisor, Restaurants, Movies, Books, Electronics, and Grocery. As illustrated in Figure 7-2, each review is associated with a question that can be answered using one or more sentences from the review.[3]

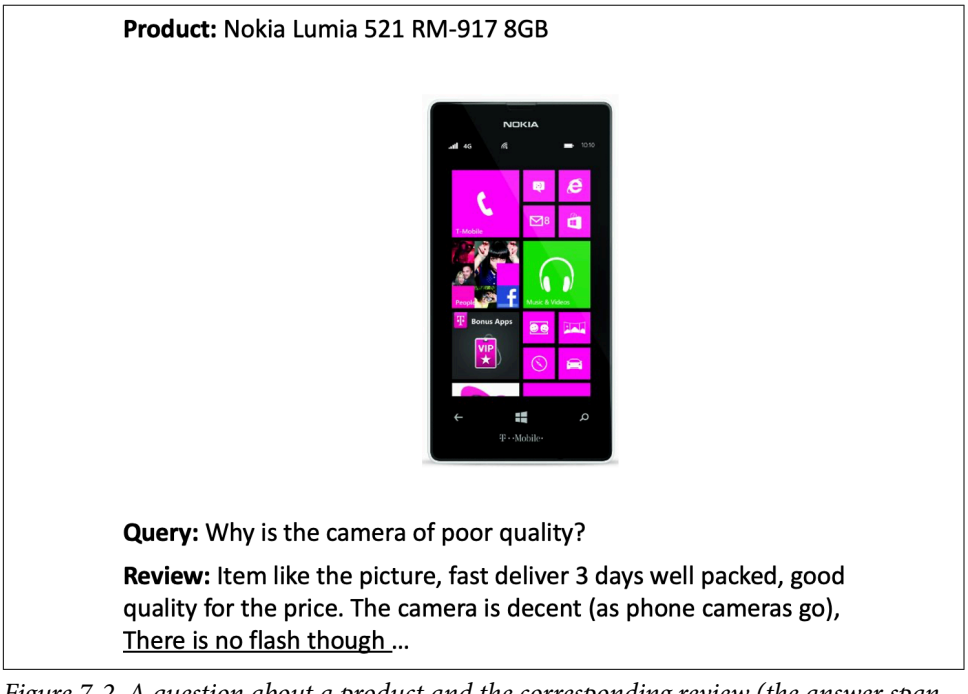


Figure 7-2. A question about a product and the corresponding review (the answer span is underlined)

The interesting aspect of this dataset is that most of the questions and answers are *subjective*; that is, they depend on the personal experience of the users. The example in Figure 7-2 shows why this feature makes the task potentially more difficult than

2 J. Bjerva et al., "SubjQA: A Dataset for Subjectivity and Review Comprehension", (2020).

3 As we'll soon see, there are also *unanswerable* questions that are designed to produce more robust models.

finding answers to factual questions like "What is the currency of the United Kingdom?" First, the query is about "poor quality," which is subjective and depends on the user's definition of quality. Second, important parts of the query do not appear in the review at all, which means it cannot be answered with shortcuts like keyword search or paraphrasing the input question. These features make SubjQA a realistic dataset to benchmark our review-based QA models on, since user-generated content like that shown in Figure 7-2 resembles what we might encounter in the wild.

QA systems are usually categorized by the *domain* of data that they have access to when responding to a query. *Closed-domain* QA deals with questions about a narrow topic (e.g., a single product category), while *open-domain* QA deals with questions about almost anything (e.g., Amazon's whole product catalog). In general, closed-domain QA involves searching through fewer documents than the open-domain case.

To get started, let's download the dataset from the Hugging Face Hub. As we did in Chapter 4, we can use the `get_dataset_config_names()` function to find out which subsets are available:

```
from datasets import get_dataset_config_names

domains = get_dataset_config_names("subjqa")
domains
```

```
['books', 'electronics', 'grocery', 'movies', 'restaurants', 'tripadvisor']
```

For our use case, we'll focus on building a QA system for the Electronics domain. To download the `electronics` subset, we just need to pass this value to the `name` argument of the `load_dataset()` function:

```
from datasets import load_dataset

subjqa = load_dataset("subjqa", name="electronics")
```

Like other question answering datasets on the Hub, SubjQA stores the answers to each question as a nested dictionary. For example, if we inspect one of the rows in the `answers` column:

```
print(subjqa["train"]["answers"][1])
```

```
{'text': ['Bass is weak as expected', 'Bass is weak as expected, even with EQ
adjusted up'], 'answer_start': [1302, 1302], 'answer_subj_level': [1, 1],
'ans_subj_score': [0.5083333253860474, 0.5083333253860474], 'is_ans_subjective':
[True, True]}
```

we can see that the answers are stored in a `text` field, while the starting character indices are provided in `answer_start`. To explore the dataset more easily, we'll flatten

these nested columns with the `flatten()` method and convert each split to a Pandas `DataFrame` as follows:

```
import pandas as pd

dfs = {split: dset.to_pandas() for split, dset in subjqa.flatten().items()}

for split, df in dfs.items():
    print(f"Number of questions in {split}: {df['id'].nunique()}")
```

```
Number of questions in train: 1295
Number of questions in test: 358
Number of questions in validation: 255
```

Notice that the dataset is relatively small, with only 1,908 examples in total. This simulates a real-world scenario, since getting domain experts to label extractive QA datasets is labor-intensive and expensive. For example, the CUAD dataset for extractive QA on legal contracts is estimated to have a value of $2 million to account for the legal expertise needed to annotate its 13,000 examples![4]

There are quite a few columns in the SubjQA dataset, but the most interesting ones for building our QA system are shown in Table 7-1.

Table 7-1. Column names and their descriptions from the SubjQA dataset

Column name	Description
`title`	The Amazon Standard Identification Number (ASIN) associated with each product
`question`	The question
`answers.answer_text`	The span of text in the review labeled by the annotator
`answers.answer_start`	The start character index of the answer span
`context`	The customer review

Let's focus on these columns and take a look at a few of the training examples. We can use the `sample()` method to select a random sample:

```
qa_cols = ["title", "question", "answers.text",
           "answers.answer_start", "context"]
sample_df = dfs["train"][qa_cols].sample(2, random_state=7)
sample_df
```

4 D. Hendrycks et al., "CUAD: An Expert-Annotated NLP Dataset for Legal Contract Review", (2021).

title	question	answers.text	answers.answer_start	context
B005DKZTMG	Does the keyboard lightweight?	[this keyboard is compact]	[215]	I really like this keyboard. I give it 4 stars because it doesn't have a CAPS LOCK key so I never know if my caps are on. But for the price, it really suffices as a wireless keyboard. I have very large hands and this keyboard is compact, but I have no complaints.
B00AAIPT76	How is the battery?	[]	[]	I bought this after the first spare gopro battery I bought wouldn't hold a charge. I have very realistic expectations of this sort of product, I am skeptical of amazing stories of charge time and battery life but I do expect the batteries to hold a charge for a couple of weeks at least and for the charger to work like a charger. In this I was not disappointed. I am a river rafter and found that the gopro burns through power in a hurry so this purchase solved that issue. the batteries held a charge, on shorter trips the extra two batteries were enough and on longer trips I could use my friends JOOS Orange to recharge them.I just bought a newtrent xtreme powerpak and expect to be able to charge these with that so I will not run out of power again.

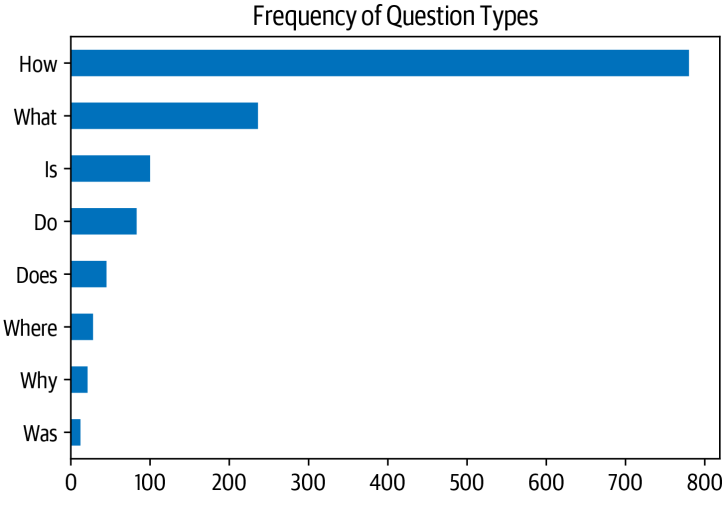
From these examples we can make a few observations. First, the questions are not grammatically correct, which is quite common in the FAQ sections of ecommerce websites. Second, an empty `answers.text` entry denotes "unanswerable" questions whose answer cannot be found in the review. Finally, we can use the start index and length of the answer span to slice out the span of text in the review that corresponds to the answer:

```
start_idx = sample_df["answers.answer_start"].iloc[0][0]
end_idx = start_idx + len(sample_df["answers.text"].iloc[0][0])
sample_df["context"].iloc[0][start_idx:end_idx]
```

```
'this keyboard is compact'
```

Next, let's get a feel for what types of questions are in the training set by counting the questions that begin with a few common starting words:

```
counts = {}
question_types = ["What", "How", "Is", "Does", "Do", "Was", "Where", "Why"]

for q in question_types:
    counts[q] = dfs["train"]["question"].str.startswith(q).value_counts()[True]

pd.Series(counts).sort_values().plot.barh()
plt.title("Frequency of Question Types")
plt.show()
```



We can see that questions beginning with "How", "What", and "Is" are the most common ones, so let's have a look at some examples:

```
for question_type in ["How", "What", "Is"]:
    for question in (
        dfs["train"][dfs["train"].question.str.startswith(question_type)]
        .sample(n=3, random_state=42)['question']):
        print(question)
```

```
How is the camera?
How do you like the control?
How fast is the charger?
What is direction?
What is the quality of the construction of the bag?
What is your impression of the product?
Is this how zoom works?
Is sound clear?
Is it a wireless keyboard?
```

The Stanford Question Answering Dataset

The *(question, review, [answer sentences])* format of SubjQA is commonly used in extractive QA datasets, and was pioneered in the Stanford Question Answering Dataset (SQuAD).[5] This is a famous dataset that is often used to test the ability of machines to read a passage of text and answer questions about it. The dataset was created by sampling several hundred English articles from Wikipedia, partitioning each article into paragraphs, and then asking crowdworkers to generate a set of questions

5 P. Rajpurkar et al., "SQuAD: 100,000+ Questions for Machine Comprehension of Text", (2016).

and answers for each paragraph. In the first version of SQuAD, each answer to a question was guaranteed to exist in the corresponding passage. But it wasn't long before sequence models started performing better than humans at extracting the correct span of text with the answer. To make the task more difficult, SQuAD 2.0 was created by augmenting SQuAD 1.1 with a set of adversarial questions that are relevant to a given passage but cannot be answered from the text alone.[6] The state of the art as of this book's writing is shown in Figure 7-3, with most models since 2019 surpassing human performance.

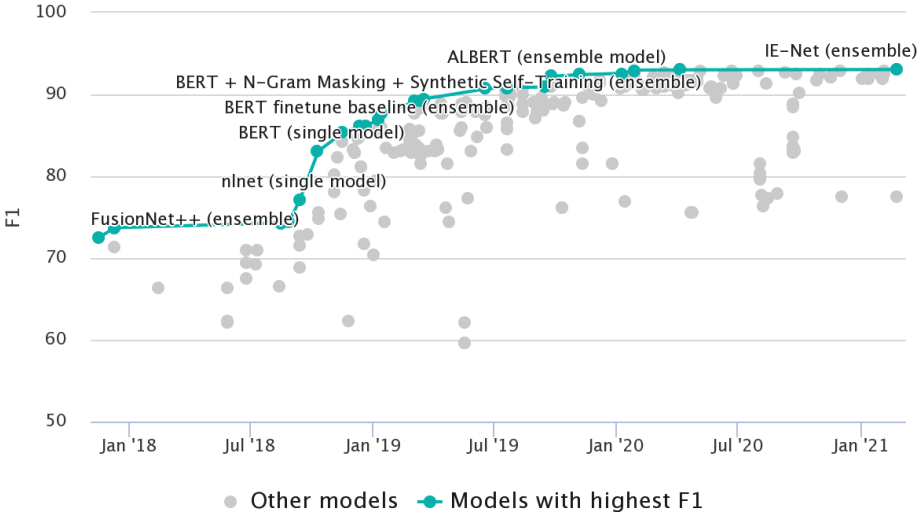


Figure 7-3. Progress on the SQuAD 2.0 benchmark (image from Papers with Code)

However, this superhuman performance does not appear to reflect genuine reading comprehension, since answers to the "unanswerable" questions can usually be identified through patterns in the passages like antonyms. To address these problems Google released the Natural Questions (NQ) dataset,[7] which involves fact-seeking questions obtained from Google Search users. The answers in NQ are much longer than in SQuAD and present a more challenging benchmark.

Now that we've explored our dataset a bit, let's dive into understanding how transformers can extract answers from text.

6 P. Rajpurkar, R. Jia, and P. Liang, "Know What You Don't Know: Unanswerable Questions for SQuAD", (2018).

7 T. Kwiatkowski et al., "Natural Questions: A Benchmark for Question Answering Research," *Transactions of the Association for Computational Linguistics* 7 (March 2019): 452–466, *http://dx.doi.org/10.1162/tacl_a_00276*.

Extracting Answers from Text

The first thing we'll need for our QA system is to find a way to identify a potential answer as a span of text in a customer review. For example, if a we have a question like "Is it waterproof?" and the review passage is "This watch is waterproof at 30m depth", then the model should output "waterproof at 30m". To do this we'll need to understand how to:

- Frame the supervised learning problem.
- Tokenize and encode text for QA tasks.
- Deal with long passages that exceed a model's maximum context size.

Let's start by taking a look at how to frame the problem.

Span classification

The most common way to extract answers from text is by framing the problem as a *span classification* task, where the start and end tokens of an answer span act as the labels that a model needs to predict. This process is illustrated in Figure 7-4.

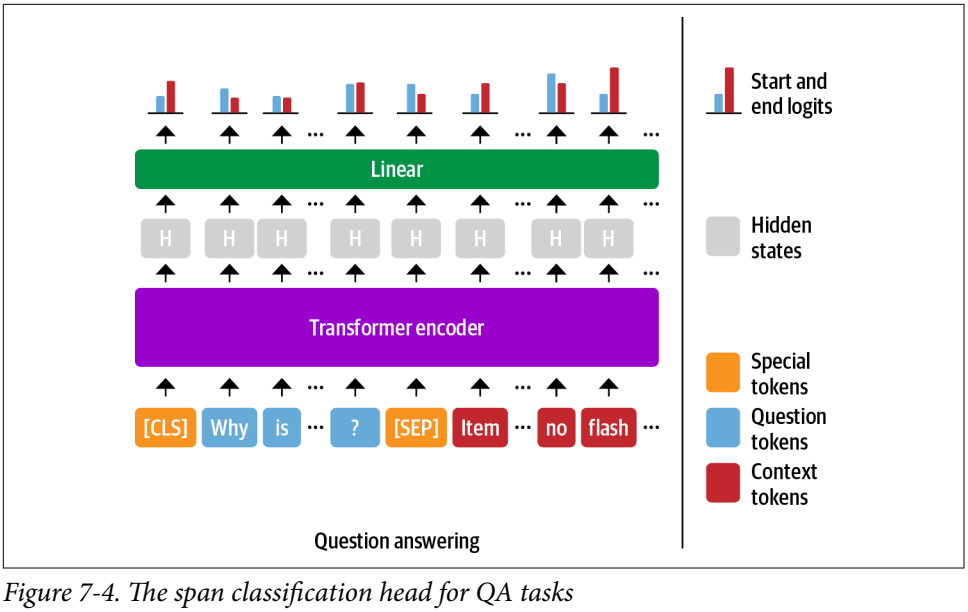


Figure 7-4. The span classification head for QA tasks

Since our training set is relatively small, with only 1,295 examples, a good strategy is to start with a language model that has already been fine-tuned on a large-scale QA dataset like SQuAD. In general, these models have strong reading comprehension capabilities and serve as a good baseline upon which to build a more accurate system. This is a somewhat different approach to that taken in previous chapters, where we

typically started with a pretrained model and fine-tuned the task-specific head ourselves. For example, in Chapter 2, we had to fine-tune the classification head because the number of classes was tied to the dataset at hand. For extractive QA, we can actually start with a fine-tuned model since the structure of the labels remains the same across datasets.

You can find a list of extractive QA models by navigating to the Hugging Face Hub and searching for "squad" on the Models tab (Figure 7-5).

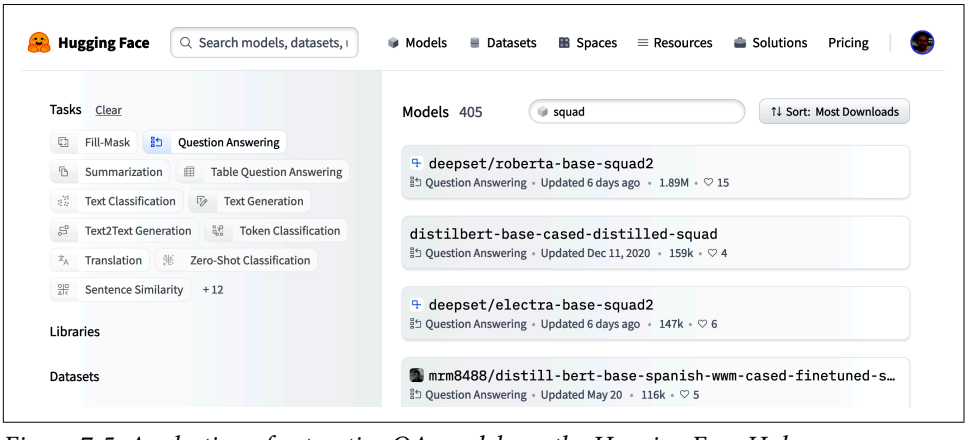


Figure 7-5. A selection of extractive QA models on the Hugging Face Hub

As you can see, at the time of writing, there are more than 350 QA models to choose from—so which one should you pick? In general, the answer depends on various factors like whether your corpus is mono- or multilingual and the constraints of running the model in a production environment. Table 7-2 lists a few models that provide a good foundation to build on.

Table 7-2. Baseline transformer models that are fine-tuned on SQuAD 2.0

Transformer	Description	Number of parameters	F_1-score on SQuAD 2.0
MiniLM	A distilled version of BERT-base that preserves 99% of the performance while being twice as fast	66M	79.5
RoBERTa-base	RoBERTa models have better performance than their BERT counterparts and can be fine-tuned on most QA datasets using a single GPU	125M	83.0
ALBERT-XXL	State-of-the-art performance on SQuAD 2.0, but computationally intensive and difficult to deploy	235M	88.1
XLM-RoBERTa-large	Multilingual model for 100 languages with strong zero-shot performance	570M	83.8

For the purposes of this chapter, we'll use a fine-tuned MiniLM model since it is fast to train and will allow us to quickly iterate on the techniques that we'll be exploring.[8] As usual, the first thing we need is a tokenizer to encode our texts, so let's take a look at how this works for QA tasks.

Tokenizing text for QA

To encode our texts, we'll load the MiniLM model checkpoint from the Hugging Face Hub as usual:

```
from transformers import AutoTokenizer

model_ckpt = "deepset/minilm-uncased-squad2"
tokenizer = AutoTokenizer.from_pretrained(model_ckpt)
```

To see the model in action, let's first try to extract an answer from a short passage of text. In extractive QA tasks, the inputs are provided as (question, context) pairs, so we pass them both to the tokenizer as follows:

```
question = "How much music can this hold?"
context = """An MP3 is about 1 MB/minute, so about 6000 hours depending on \
file size."""
inputs = tokenizer(question, context, return_tensors="pt")
```

Here we've returned PyTorch `Tensor` objects, since we'll need them to run the forward pass through the model. If we view the tokenized inputs as a table:

input_ids	101	2129	2172	2189	2064	2023	...	5834	2006	5371	2946	1012	102
token_type_ids	0	0	0	0	0	0	...	1	1	1	1	1	1
attention_mask	1	1	1	1	1	1	...	1	1	1	1	1	1

we can see the familiar `input_ids` and `attention_mask` tensors, while the `token_type_ids` tensor indicates which part of the inputs corresponds to the question and context (a 0 indicates a question token, a 1 indicates a context token).[9]

To understand how the tokenizer formats the inputs for QA tasks, let's decode the `input_ids` tensor:

```
print(tokenizer.decode(inputs["input_ids"][0]))
```

8 W. Wang et al., "MINILM: Deep Self-Attention Distillation for Task-Agnostic Compression of Pre-Trained Transformers", (2020).

9 Note that the `token_type_ids` are not present in all transformer models. In the case of BERT-like models such as MiniLM, the `token_type_ids` are also used during pretraining to incorporate the next sentence prediction task.

```
[CLS] how much music can this hold? [SEP] an mp3 is about 1 mb / minute, so
about 6000 hours depending on file size. [SEP]
```

We see that for each QA example, the inputs take the format:

```
[CLS] question tokens [SEP] context tokens [SEP]
```

where the location of the first `[SEP]` token is determined by the `token_type_ids`. Now that our text is tokenized, we just need to instantiate the model with a QA head and run the inputs through the forward pass:

```
import torch
from transformers import AutoModelForQuestionAnswering

model = AutoModelForQuestionAnswering.from_pretrained(model_ckpt)

with torch.no_grad():
    outputs = model(**inputs)
print(outputs)
```

```
QuestionAnsweringModelOutput(loss=None, start_logits=tensor([[-0.9862, -4.7750,
         -5.4025, -5.2378, -5.2863, -5.5117, -4.9819, -6.1880,
         -0.9862,  0.2596, -0.2144, -1.7136,  3.7806,  4.8561, -1.0546, -3.9097,
         -1.7374, -4.5944, -1.4278,  3.9949,  5.0390, -0.2018, -3.0193, -4.8549,
         -2.3107, -3.5110, -3.5713, -0.9862]]), end_logits=tensor([[-0.9623,
         -5.4733, -5.0326, -5.1639, -5.4278, -5.5151, -5.1749, -4.6233,
         -0.9623, -3.7855, -0.8715, -3.7745, -3.0161, -1.1780,  0.1758, -2.7365,
          4.8934,  0.3046, -3.1761, -3.2762,  0.8937,  5.6606, -0.3623, -4.9554,
         -3.2531, -0.0914,  1.6211, -0.9623]]), hidden_states=None,
attentions=None)
```

Here we can see that we get a `QuestionAnsweringModelOutput` object as the output of the QA head. As illustrated in Figure 7-4, the QA head corresponds to a linear layer that takes the hidden states from the encoder and computes the logits for the start and end spans.[10] This means that we treat QA as a form of token classification, similar to what we encountered for named entity recognition in Chapter 4. To convert the outputs into an answer span, we first need to get the logits for the start and end tokens:

```
start_logits = outputs.start_logits
end_logits = outputs.end_logits
```

If we compare the shapes of these logits to the input IDs:

```
print(f"Input IDs shape: {inputs.input_ids.size()}")
print(f"Start logits shape: {start_logits.size()}")
print(f"End logits shape: {end_logits.size()}")
```

10 See Chapter 2 for details on how these hidden states can be extracted.

```
Input IDs shape: torch.Size([1, 28])
Start logits shape: torch.Size([1, 28])
End logits shape: torch.Size([1, 28])
```

we see that there are two logits (a start and end) associated with each input token. As illustrated in Figure 7-6, larger, positive logits correspond to more likely candidates for the start and end tokens. In this example we can see that the model assigns the highest start token logits to the numbers "1" and "6000", which makes sense since our question is asking about a quantity. Similarly, we see that the end tokens with the highest logits are "minute" and "hours".

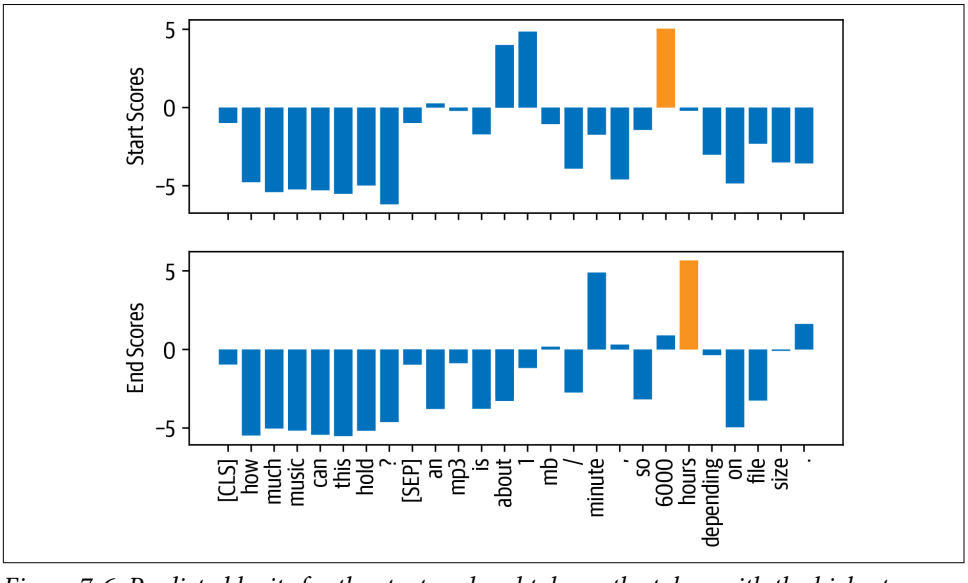


Figure 7-6. Predicted logits for the start and end tokens; the token with the highest score is colored in orange

To get the final answer, we can compute the argmax over the start and end token logits and then slice the span from the inputs. The following code performs these steps and decodes the result so we can print the resulting text:

```
import torch

start_idx = torch.argmax(start_logits)
end_idx = torch.argmax(end_logits) + 1
answer_span = inputs["input_ids"][0][start_idx:end_idx]
answer = tokenizer.decode(answer_span)
print(f"Question: {question}")
print(f"Answer: {answer}")
```

```
Question: How much music can this hold?
Answer: 6000 hours
```

Great, it worked! In 🤗 Transformers, all of these preprocessing and postprocessing steps are conveniently wrapped in a dedicated pipeline. We can instantiate the pipeline by passing our tokenizer and fine-tuned model as follows:

```
from transformers import pipeline

pipe = pipeline("question-answering", model=model, tokenizer=tokenizer)
pipe(question=question, context=context, topk=3)
```

```
[{'score': 0.26516005396842957,
  'start': 38,
  'end': 48,
  'answer': '6000 hours'},
 {'score': 0.2208300083875656,
  'start': 16,
  'end': 48,
  'answer': '1 MB/minute, so about 6000 hours'},
 {'score': 0.10253632068634033,
  'start': 16,
  'end': 27,
  'answer': '1 MB/minute'}]
```

In addition to the answer, the pipeline also returns the model's probability estimate in the `score` field (obtained by taking a softmax over the logits). This is handy when we want to compare multiple answers within a single context. We've also shown that we can have the model predict multiple answers by specifying the `topk` parameter. Sometimes, it is possible to have questions for which no answer is possible, like the empty `answers.answer_start` examples in SubjQA. In these cases the model will assign a high start and end score to the `[CLS]` token, and the pipeline maps this output to an empty string:

```
pipe(question="Why is there no data?", context=context,
     handle_impossible_answer=True)
```

```
{'score': 0.9068416357040405, 'start': 0, 'end': 0, 'answer': ''}
```

In our simple example, we obtained the start and end indices by taking the argmax of the corresponding logits. However, this heuristic can produce out-of-scope answers by selecting tokens that belong to the question instead of the context. In practice, the pipeline computes the best combination of start and end indices subject to various constraints such as being in-scope, requiring the start indices to precede the end indices, and so on.

Dealing with long passages

One subtlety faced by reading comprehension models is that the context often contains more tokens than the maximum sequence length of the model (which is usually a few hundred tokens at most). As illustrated in Figure 7-7, a decent portion of the SubjQA training set contains question-context pairs that won't fit within MiniLM's context size of 512 tokens.

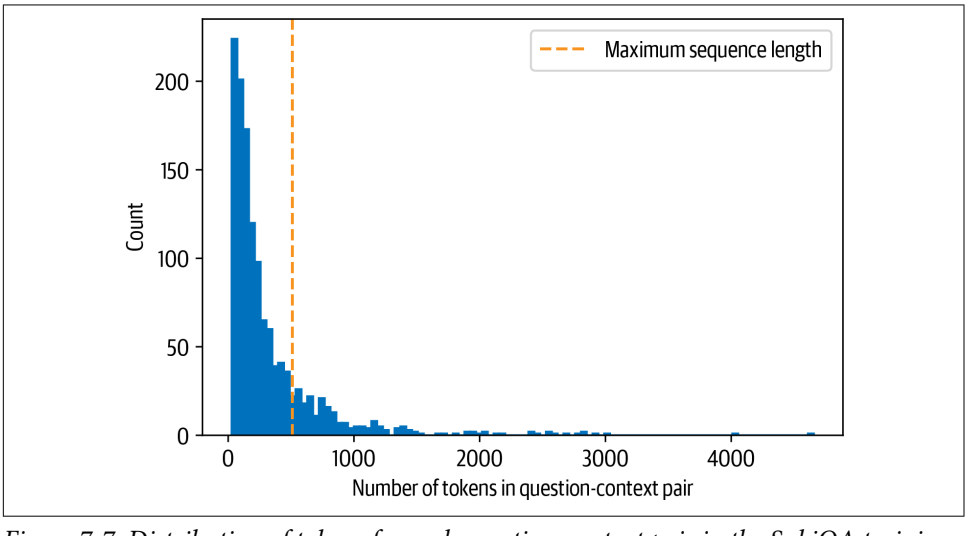


Figure 7-7. Distribution of tokens for each question-context pair in the SubjQA training set

For other tasks, like text classification, we simply truncated long texts under the assumption that enough information was contained in the embedding of the `[CLS]` token to generate accurate predictions. For QA, however, this strategy is problematic because the answer to a question could lie near the end of the context and thus would be removed by truncation. As illustrated in Figure 7-8, the standard way to deal with this is to apply a *sliding window* across the inputs, where each window contains a passage of tokens that fit in the model's context.

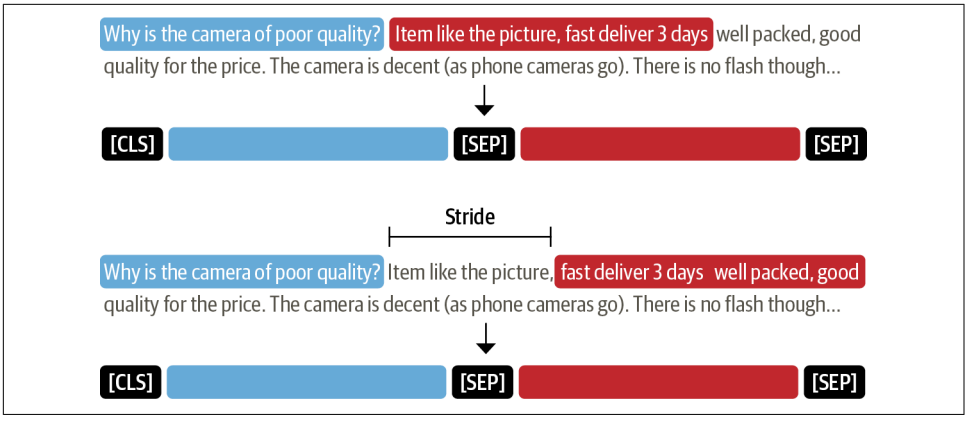


Figure 7-8. How the sliding window creates multiple question-context pairs for long documents—the first bar corresponds to the question, while the second bar is the context captured in each window

In 🤗 Transformers, we can set `return_overflowing_tokens=True` in the tokenizer to enable the sliding window. The size of the sliding window is controlled by the `max_seq_length` argument, and the size of the stride is controlled by `doc_stride`. Let's grab the first example from our training set and define a small window to illustrate how this works:

```
example = dfs["train"].iloc[0][["question", "context"]]
tokenized_example = tokenizer(example["question"], example["context"],
                              return_overflowing_tokens=True, max_length=100,
                              stride=25)
```

In this case we now get a list of `input_ids`, one for each window. Let's check the number of tokens we have in each window:

```
for idx, window in enumerate(tokenized_example["input_ids"]):
    print(f"Window #{idx} has {len(window)} tokens")
```

```
Window #0 has 100 tokens
Window #1 has 88 tokens
```

Finally, we can see where two windows overlap by decoding the inputs:

```
for window in tokenized_example["input_ids"]:
    print(f"{tokenizer.decode(window)} \n")
```

```
[CLS] how is the bass? [SEP] i have had koss headphones in the past, pro 4aa and
qz - 99. the koss portapro is portable and has great bass response. the work
great with my android phone and can be " rolled up " to be carried in my
motorcycle jacket or computer bag without getting crunched. they are very light
and don't feel heavy or bear down on your ears even after listening to music
with them on all day. the sound is [SEP]

[CLS] how is the bass? [SEP] and don't feel heavy or bear down on your ears even
```

```
after listening to music with them on all day. the sound is night and day better
than any ear - bud could be and are almost as good as the pro 4aa. they are "
open air " headphones so you cannot match the bass to the sealed types, but it
comes close. for $ 32, you cannot go wrong. [SEP]
```

Now that we have some intuition about how QA models can extract answers from text, let's look at the other components we need to build an end-to-end QA pipeline.

Using Haystack to Build a QA Pipeline

In our simple answer extraction example, we provided both the question and the context to the model. However, in reality our system's users will only provide a question about a product, so we need some way of selecting relevant passages from among all the reviews in our corpus. One way to do this would be to concatenate all the reviews of a given product together and feed them to the model as a single, long context. Although simple, the drawback of this approach is that the context can become extremely long and thereby introduce an unacceptable latency for our users' queries. For example, let's suppose that on average, each product has 30 reviews and each review takes 100 milliseconds to process. If we need to process all the reviews to get an answer, this would result in an average latency of 3 seconds per user query—much too long for ecommerce websites!

To handle this, modern QA systems are typically based on the *retriever-reader* architecture, which has two main components:

Retriever
: Responsible for retrieving relevant documents for a given query. Retrievers are usually categorized as *sparse* or *dense*. Sparse retrievers use word frequencies to represent each document and query as a sparse vector.[11] The relevance of a query and a document is then determined by computing an inner product of the vectors. On the other hand, dense retrievers use encoders like transformers to represent the query and document as contextualized embeddings (which are dense vectors). These embeddings encode semantic meaning, and allow dense retrievers to improve search accuracy by understanding the content of the query.

Reader
: Responsible for extracting an answer from the documents provided by the retriever. The reader is usually a reading comprehension model, although at the end of the chapter we'll see examples of models that can generate free-form answers.

11 A vector is sparse if most of its elements are zero.

As illustrated in Figure 7-9, there can also be other components that apply post-processing to the documents fetched by the retriever or to the answers extracted by the reader. For example, the retrieved documents may need reranking to eliminate noisy or irrelevant ones that can confuse the reader. Similarly, postprocessing of the reader's answers is often needed when the correct answer comes from various passages in a long document.

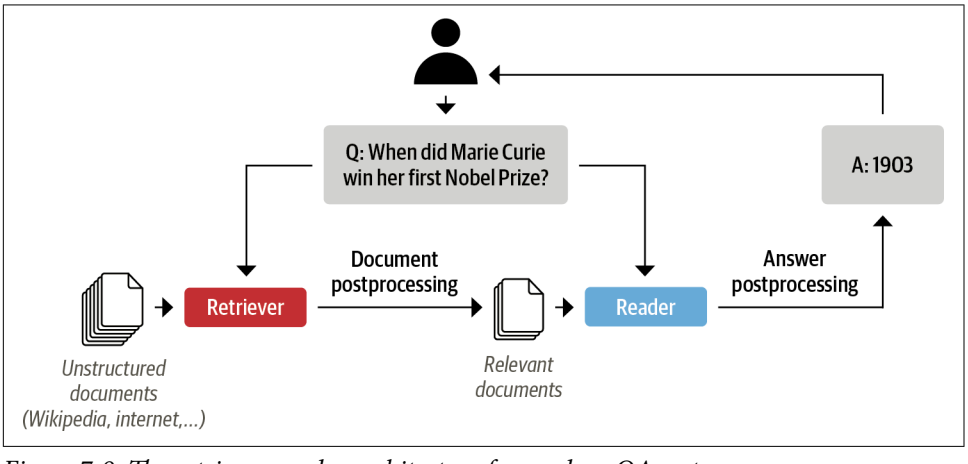


Figure 7-9. The retriever-reader architecture for modern QA systems

To build our QA system, we'll use the *Haystack* library developed by deepset, a German company focused on NLP. Haystack is based on the retriever-reader architecture, abstracts much of the complexity involved in building these systems, and integrates tightly with 🤗 Transformers.

In addition to the retriever and reader, there are two more components involved when building a QA pipeline with Haystack:

Document store
: A document-oriented database that stores documents and metadata which are provided to the retriever at query time

Pipeline
: Combines all the components of a QA system to enable custom query flows, merging documents from multiple retrievers, and more

In this section we'll look at how we can use these components to quickly build a prototype QA pipeline. Later, we'll examine how we can improve its performance.

This chapter was written using version 0.9.0 of the Haystack library. In version 0.10.0, the pipeline and evaluation APIs were redesigned to make it easier to inspect whether the retriever or reader are impacting performance. To see what this chapter's code looks like with the new API, check out the GitHub repository.

Initializing a document store

In Haystack, there are various document stores to choose from and each one can be paired with a dedicated set of retrievers. This is illustrated in Table 7-3, where the compatibility of sparse (TF-IDF, BM25) and dense (Embedding, DPR) retrievers is shown for each of the available document stores. We'll explain what all these acronyms mean later in this chapter.

Table 7-3. Compatibility of Haystack retrievers and document stores

	In memory	Elasticsearch	FAISS	Milvus
TF-IDF	Yes	Yes	No	No
BM25	No	Yes	No	No
Embedding	Yes	Yes	Yes	Yes
DPR	Yes	Yes	Yes	Yes

Since we'll be exploring both sparse and dense retrievers in this chapter, we'll use the `ElasticsearchDocumentStore`, which is compatible with both retriever types. Elasticsearch is a search engine that is capable of handling a diverse range of data types, including textual, numerical, geospatial, structured, and unstructured. Its ability to store huge volumes of data and quickly filter it with full-text search features makes it especially well suited for developing QA systems. It also has the advantage of being the industry standard for infrastructure analytics, so there's a good chance your company already has a cluster that you can work with.

To initialize the document store, we first need to download and install Elasticsearch. By following Elasticsearch's guide,[12] we can grab the latest release for Linux with `wget` and unpack it with the `tar` shell command:

```
url = """https://artifacts.elastic.co/downloads/elasticsearch/\
elasticsearch-7.9.2-linux-x86_64.tar.gz"""
!wget -nc -q {url}
!tar -xzf elasticsearch-7.9.2-linux-x86_64.tar.gz
```

Next we need to start the Elasticsearch server. Since we're running all the code in this book within Jupyter notebooks, we'll need to use Python's `Popen()` function to spawn

12 The guide also provides installation instructions for macOS and Windows.

a new process. While we're at it, let's also run the subprocess in the background using the `chown` shell command:

```
import os
from subprocess import Popen, PIPE, STDOUT

# Run Elasticsearch as a background process
!chown -R daemon:daemon elasticsearch-7.9.2
es_server = Popen(args=['elasticsearch-7.9.2/bin/elasticsearch'],
                  stdout=PIPE, stderr=STDOUT, preexec_fn=lambda: os.setuid(1))
# Wait until Elasticsearch has started
!sleep 30
```

In the `Popen()` function, the `args` specify the program we wish to execute, while `stdout=PIPE` creates a new pipe for the standard output and `stderr=STDOUT` collects the errors in the same pipe. The `preexec_fn` argument specifies the ID of the subprocess we wish to use. By default, Elasticsearch runs locally on port 9200, so we can test the connection by sending an HTTP request to `localhost`:

```
!curl -X GET "localhost:9200/?pretty"
```

```
{
  "name" : "96938eee37cd",
  "cluster_name" : "docker-cluster",
  "cluster_uuid" : "ABGDdvbbRWmMb9Umz79HbA",
  "version" : {
    "number" : "7.9.2",
    "build_flavor" : "default",
    "build_type" : "docker",
    "build_hash" : "d34da0ea4a966c4e49417f2da2f244e3e97b4e6e",
    "build_date" : "2020-09-23T00:45:33.626720Z",
    "build_snapshot" : false,
    "lucene_version" : "8.6.2",
    "minimum_wire_compatibility_version" : "6.8.0",
    "minimum_index_compatibility_version" : "6.0.0-beta1"
  },
  "tagline" : "You Know, for Search"
}
```

Now that our Elasticsearch server is up and running, the next thing to do is instantiate the document store:

```
from haystack.document_store.elasticsearch import ElasticsearchDocumentStore

# Return the document embedding for later use with dense retriever
document_store = ElasticsearchDocumentStore(return_embedding=True)
```

By default, `ElasticsearchDocumentStore` creates two indices on Elasticsearch: one called `document` for (you guessed it) storing documents, and another called `label` for storing the annotated answer spans. For now, we'll just populate the `document` index

with the SubjQA reviews, and Haystack's document stores expect a list of dictionaries with `text` and `meta` keys as follows:

```
{
    "text": "<the-context>",
    "meta": {
        "field_01": "<additional-metadata>",
        "field_02": "<additional-metadata>",
        ...
    }
}
```

The fields in `meta` can be used for applying filters during retrieval. For our purposes we'll include the `item_id` and `q_review_id` columns of SubjQA so we can filter by product and question ID, along with the corresponding training split. We can then loop through the examples in each `DataFrame` and add them to the index with the `write_documents()` method as follows:

```
for split, df in dfs.items():
    # Exclude duplicate reviews
    docs = [{"text": row["context"],
             "meta":{"item_id": row["title"], "question_id": row["id"],
                     "split": split}}
        for _,row in df.drop_duplicates(subset="context").iterrows()]
    document_store.write_documents(docs, index="document")

print(f"Loaded {document_store.get_document_count()} documents")
```

```
Loaded 1615 documents
```

Great, we've loaded all our reviews into an index! To search the index we'll need a retriever, so let's look at how we can initialize one for Elasticsearch.

Initializing a retriever

The Elasticsearch document store can be paired with any of the Haystack retrievers, so let's start by using a sparse retriever based on BM25 (short for "Best Match 25"). BM25 is an improved version of the classic Term Frequency-Inverse Document Frequency (TF-IDF) algorithm and represents the question and context as sparse vectors that can be searched efficiently on Elasticsearch. The BM25 score measures how much matched text is about a search query and improves on TF-IDF by saturating TF values quickly and normalizing the document length so that short documents are favored over long ones.[13]

13 For an in-depth explanation of document scoring with TF-IDF and BM25 see Chapter 23 of *Speech and Language Processing*, 3rd edition, by D. Jurafsky and J.H. Martin (Prentice Hall).

In Haystack, the BM25 retriever is used by default in `ElasticsearchRetriever`, so let's initialize this class by specifying the document store we wish to search over:

```
from haystack.retriever.sparse import ElasticsearchRetriever

es_retriever = ElasticsearchRetriever(document_store=document_store)
```

Next, let's look at a simple query for a single electronics product in the training set. For review-based QA systems like ours, it's important to restrict the queries to a single item because otherwise the retriever would source reviews about products that are not related to a user's query. For example, asking "Is the camera quality any good?" without a product filter could return reviews about phones, when the user might be asking about a specific laptop camera instead. By themselves, the ASIN values in our dataset are a bit cryptic, but we can decipher them with online tools like *amazon ASIN* or by simply appending the value of `item_id` to the *www.amazon.com/dp/* URL. The following item ID corresponds to one of Amazon's Fire tablets, so let's use the retriever's `retrieve()` method to ask if it's any good for reading with:

```
item_id = "B0074BW614"
query = "Is it good for reading?"
retrieved_docs = es_retriever.retrieve(
    query=query, top_k=3, filters={"item_id":[item_id], "split":["train"]})
```

Here we've specified how many documents to return with the `top_k` argument and applied a filter on both the `item_id` and `split` keys that were included in the `meta` field of our documents. Each element of `retrieved_docs` is a Haystack `Document` object that is used to represent documents and includes the retriever's query score along with other metadata. Let's have a look at one of the retrieved documents:

```
print(retrieved_docs[0])
```

```
{'text': 'This is a gift to myself.  I have been a kindle user for 4 years and
this is my third one.  I never thought I would want a fire for I mainly use it
for book reading.  I decided to try the fire for when I travel I take my laptop,
my phone and my iPod classic.  I love my iPod but watching movies on the plane
with it can be challenging because it is so small. Laptops battery life is not
as good as the Kindle.  So the Fire combines for me what I needed all three to
do. So far so good.', 'score': 6.243799, 'probability': 0.6857824513476455,
'question': None, 'meta': {'item_id': 'B0074BW614', 'question_id':
'868e311275e26dbafe5af70774a300f3', 'split': 'train'}, 'embedding': None, 'id':
'252e83e25d52df7311d597dc89eef9f6'}
```

In addition to the document's text, we can see the `score` that Elasticsearch computed for its relevance to the query (larger scores imply a better match). Under the hood, Elasticsearch relies on Lucene for indexing and search, so by default it uses Lucene's *practical scoring function*. You can find the nitty-gritty details behind the scoring function in the Elasticsearch documentation, but in brief terms it first filters the candidate documents by applying a Boolean test (does the document match the query?),

and then applies a similarity metric that's based on representing both the document and the query as vectors.

Now that we have a way to retrieve relevant documents, the next thing we need is a way to extract answers from them. This is where the reader comes in, so let's take a look at how we can load our MiniLM model in Haystack.

Initializing a reader

In Haystack, there are two types of readers one can use to extract answers from a given context:

`FARMReader`
: Based on deepset's *FARM* framework for fine-tuning and deploying transformers. Compatible with models trained using 🤗 Transformers and can load models directly from the Hugging Face Hub.

`TransformersReader`
: Based on the QA pipeline from 🤗 Transformers. Suitable for running inference only.

Although both readers handle a model's weights in the same way, there are some differences in the way the predictions are converted to produce answers:

- In 🤗 Transformers, the QA pipeline normalizes the start and end logits with a softmax in each passage. This means that it is only meaningful to compare answer scores between answers extracted from the same passage, where the probabilities sum to 1. For example, an answer score of 0.9 from one passage is not necessarily better than a score of 0.8 in another. In FARM, the logits are not normalized, so inter-passage answers can be compared more easily.
- The `TransformersReader` sometimes predicts the same answer twice, but with different scores. This can happen in long contexts if the answer lies across two overlapping windows. In FARM, these duplicates are removed.

Since we will be fine-tuning the reader later in the chapter, we'll use the `FARMReader`. As with 🤗 Transformers, to load the model we just need to specify the MiniLM checkpoint on the Hugging Face Hub along with some QA-specific arguments:

```
from haystack.reader.farm import FARMReader

model_ckpt = "deepset/minilm-uncased-squad2"
max_seq_length, doc_stride = 384, 128
reader = FARMReader(model_name_or_path=model_ckpt, progress_bar=False,
                    max_seq_len=max_seq_length, doc_stride=doc_stride,
                    return_no_answer=True)
```

It is also possible to fine-tune a reading comprehension model directly in 🤗 Transformers and then load it in `TransformersReader` to run inference. For details on how to do the fine-tuning step, see the question answering tutorial in the library's documentation.

In `FARMReader`, the behavior of the sliding window is controlled by the same `max_seq_length` and `doc_stride` arguments that we saw for the tokenizer. Here we've used the values from the MiniLM paper. To confirm, let's now test the reader on our simple example from earlier:

```
print(reader.predict_on_texts(question=question, texts=[context], top_k=1))
```

```
{'query': 'How much music can this hold?', 'no_ans_gap': 12.648084878921509,
'answers': [{'answer': '6000 hours', 'score': 10.69961929321289, 'probability':
0.3988136053085327, 'context': 'An MP3 is about 1 MB/minute, so about 6000 hours
depending on file size.', 'offset_start': 38, 'offset_end': 48,
'offset_start_in_doc': 38, 'offset_end_in_doc': 48, 'document_id':
'e344757014e804eff50faa3ecf1c9c75'}]}
```

Great, the reader appears to be working as expected—so next, let's tie together all our components using one of Haystack's pipelines.

Putting it all together

Haystack provides a `Pipeline` abstraction that allows us to combine retrievers, readers, and other components together as a graph that can be easily customized for each use case. There are also predefined pipelines analogous to those in 🤗 Transformers, but specialized for QA systems. In our case, we're interested in extracting answers, so we'll use the `ExtractiveQAPipeline`, which takes a single retriever-reader pair as its arguments:

```
from haystack.pipeline import ExtractiveQAPipeline

pipe = ExtractiveQAPipeline(reader, es_retriever)
```

Each `Pipeline` has a `run()` method that specifies how the query flow should be executed. For the `ExtractiveQAPipeline` we just need to pass the `query`, the number of documents to retrieve with `top_k_retriever`, and the number of answers to extract from these documents with `top_k_reader`. In our case, we also need to specify a filter over the item ID, which can be done using the `filters` argument as we did with the retriever earlier. Let's run a simple example using our question about the Amazon Fire tablet again, but this time returning the extracted answers:

```
n_answers = 3
preds = pipe.run(query=query, top_k_retriever=3, top_k_reader=n_answers,
                 filters={"item_id": [item_id], "split":["train"]})

print(f"Question: {preds['query']} \n")
```

```
for idx in range(n_answers):
    print(f"Answer {idx+1}: {preds['answers'][idx]['answer']}")
    print(f"Review snippet: ...{preds['answers'][idx]['context']}...")
    print("\n\n")
```

```
Question: Is it good for reading?

Answer 1: I mainly use it for book reading
Review snippet: ... is my third one.  I never thought I would want a fire for I
mainly use it for book reading.  I decided to try the fire for when I travel I
take my la...


Answer 2: the larger screen compared to the Kindle makes for easier reading
Review snippet: ...ght enough that I can hold it to read, but the larger screen
compared to the Kindle makes for easier reading. I love the color, something I
never thou...


Answer 3: it is great for reading books when no light is available
Review snippet: ...ecoming addicted to hers! Our son LOVES it and it is great
for reading books when no light is available. Amazing sound but I suggest good
headphones t...
```

Great, we now have an end-to-end QA system for Amazon product reviews! This is a good start, but notice that the second and third answers are closer to what the question is actually asking. To do better, we'll need some metrics to quantify the performance of the retriever and reader. We'll take a look at that next.

Improving Our QA Pipeline

Although much of the recent research on QA has focused on improving reading comprehension models, in practice it doesn't matter how good your reader is if the retriever can't find the relevant documents in the first place! In particular, the retriever sets an upper bound on the performance of the whole QA system, so it's important to make sure it's doing a good job. With this in mind, let's start by introducing some common metrics to evaluate the retriever so that we can compare the performance of sparse and dense representations.

Evaluating the Retriever

A common metric for evaluating retrievers is *recall*, which measures the fraction of all relevant documents that are retrieved. In this context, "relevant" simply means whether the answer is present in a passage of text or not, so given a set of questions, we can compute recall by counting the number of times an answer appears in the top k documents returned by the retriever.

In Haystack, there are two ways to evaluate retrievers:

- Use the retriever's in-built `eval()` method. This can be used for both open- and closed-domain QA, but not for datasets like SubjQA where each document is paired with a single product and we need to filter by product ID for every query.
- Build a custom `Pipeline` that combines a retriever with the `EvalRetriever` class. This enables the implementation of custom metrics and query flows.

A complementary metric to recall is *mean average precision* (mAP), which rewards retrievers that can place the correct answers higher up in the document ranking.

Since we need to evaluate the recall per product and then aggregate across all products, we'll opt for the second approach. Each node in the `Pipeline` graph represents a class that takes some inputs and produces some outputs via a `run()` method:

```
class PipelineNode:
    def __init__(self):
        self.outgoing_edges = 1

    def run(self, **kwargs):
        ...
        return (outputs, "outgoing_edge_name")
```

Here `kwargs` corresponds to the outputs from the previous node in the graph, which is manipulated within the `run()` method to return a tuple of the outputs for the next node, along with a name for the outgoing edge. The only other requirement is to include an `outgoing_edges` attribute that indicates the number of outputs from the node (in most cases `outgoing_edges=1`, unless you have branches in the pipeline that route the inputs according to some criterion).

In our case, we need a node to evaluate the retriever, so we'll use the `EvalRetriever` class whose `run()` method keeps track of which documents have answers that match the ground truth. With this class we can then build up a `Pipeline` graph by adding the evaluation node after a node that represents the retriever itself:

```
from haystack.pipeline import Pipeline
from haystack.eval import EvalDocuments

class EvalRetrieverPipeline:
    def __init__(self, retriever):
        self.retriever = retriever
        self.eval_retriever = EvalDocuments()
        pipe = Pipeline()
        pipe.add_node(component=self.retriever, name="ESRetriever",
```

```
                    inputs=["Query"])
        pipe.add_node(component=self.eval_retriever, name="EvalRetriever",
                    inputs=["ESRetriever"])
        self.pipeline = pipe


pipe = EvalRetrieverPipeline(es_retriever)
```

Notice that each node is given a `name` and a list of `inputs`. In most cases, each node has a single outgoing edge, so we just need to include the name of the previous node in `inputs`.

Now that we have our evaluation pipeline, we need to pass some queries and their corresponding answers. To do this, we'll add the answers to a dedicated `label` index on our document store. Haystack provides a `Label` object that represents the answer spans and their metadata in a standardized fashion. To populate the `label` index, we'll first create a list of `Label` objects by looping over each question in the test set and extracting the matching answers and additional metadata:

```
from haystack import Label

labels = []
for i, row in dfs["test"].iterrows():
    # Metadata used for filtering in the Retriever
    meta = {"item_id": row["title"], "question_id": row["id"]}
    # Populate labels for questions with answers
    if len(row["answers.text"]):
        for answer in row["answers.text"]:
            label = Label(
                question=row["question"], answer=answer, id=i, origin=row["id"],
                meta=meta, is_correct_answer=True, is_correct_document=True,
                no_answer=False)
            labels.append(label)
    # Populate labels for questions without answers
    else:
        label = Label(
            question=row["question"], answer="", id=i, origin=row["id"],
            meta=meta, is_correct_answer=True, is_correct_document=True,
            no_answer=True)
        labels.append(label)
```

If we peek at one of these labels:

```
print(labels[0])
```

```
{'id': 'e28f5e62-85e8-41b2-8a34-fbff63b7a466', 'created_at': None, 'updated_at':
None, 'question': 'What is the tonal balance of these headphones?', 'answer': 'I
have been a headphone fanatic for thirty years', 'is_correct_answer': True,
'is_correct_document': True, 'origin': 'd0781d13200014aa25860e44da9d5ea7',
'document_id': None, 'offset_start_in_doc': None, 'no_answer': False,
'model_id': None, 'meta': {'item_id': 'B00001WRSJ', 'question_id':
'd0781d13200014aa25860e44da9d5ea7'}}
```

we can see the question-answer pair, along with an `origin` field that contains the unique question ID so we can filter the document store per question. We've also added the product ID to the `meta` field so we can filter the labels by product. Now that we have our labels, we can write them to the `label` index on Elasticsearch as follows:

```
document_store.write_labels(labels, index="label")
print(f"""Loaded {document_store.get_label_count(index="label")} \
question-answer pairs""")
```

```
Loaded 358 question-answer pairs
```

Next, we need to build up a mapping between our question IDs and corresponding answers that we can pass to the pipeline. To get all the labels, we can use the `get_all_labels_aggregated()` method from the document store that will aggregate all question-answer pairs associated with a unique ID. This method returns a list of `MultiLabel` objects, but in our case we only get one element since we're filtering by question ID. We can build up a list of aggregated labels as follows:

```
labels_agg = document_store.get_all_labels_aggregated(
    index="label",
    open_domain=True,
    aggregate_by_meta=["item_id"]
)
print(len(labels_agg))
```

```
330
```

By peeking at one of these labels we can see that all the answers associated with a given question are aggregated together in a `multiple_answers` field:

```
print(labels_agg[109])
```

```
{'question': 'How does the fan work?', 'multiple_answers': ['the fan is really
really good', "the fan itself isn't super loud. There is an adjustable dial to
change fan speed"], 'is_correct_answer': True, 'is_correct_document': True,
'origin': '5a9b7616541f700f103d21f8ad41bc4b', 'multiple_document_ids': [None,
None], 'multiple_offset_start_in_docs': [None, None], 'no_answer': False,
'model_id': None, 'meta': {'item_id': 'B002MU1ZRS'}}
```

We now have all the ingredients for evaluating the retriever, so let's define a function that feeds each question-answer pair associated with each product to the evaluation pipeline and tracks the correct retrievals in our `pipe` object:

```
def run_pipeline(pipeline, top_k_retriever=10, top_k_reader=4):
    for l in labels_agg:
        _ = pipeline.pipeline.run(
            query=l.question,
            top_k_retriever=top_k_retriever,
            top_k_reader=top_k_reader,
            top_k_eval_documents=top_k_retriever,
            labels=l,
            filters={"item_id": [l.meta["item_id"]], "split": ["test"]})
```

```
run_pipeline(pipe, top_k_retriever=3)
print(f"Recall@3: {pipe.eval_retriever.recall:.2f}")
```

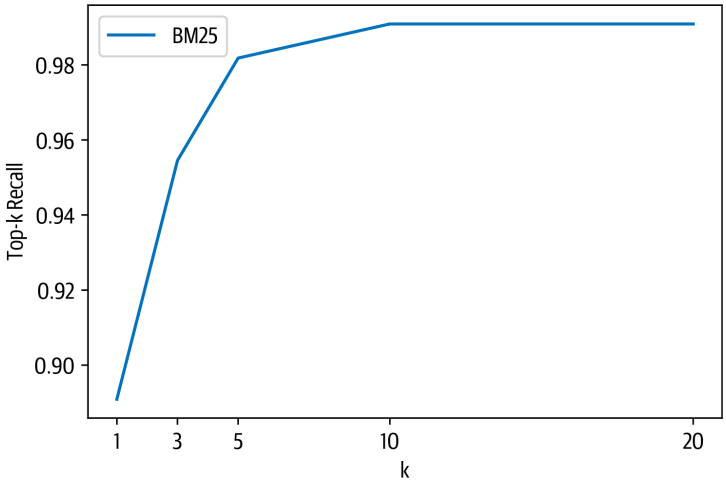
```
Recall@3: 0.95
```

Great, it works! Notice that we picked a specific value for `top_k_retriever` to specify the number of documents to retrieve. In general, increasing this parameter will improve the recall, but at the expense of providing more documents to the reader and slowing down the end-to-end pipeline. To guide our decision on which value to pick, we'll create a function that loops over several k values and compute the recall across the whole test set for each k:

```
def evaluate_retriever(retriever, topk_values = [1,3,5,10,20]):
    topk_results = {}

    for topk in topk_values:
        # Create Pipeline
        p = EvalRetrieverPipeline(retriever)
        # Loop over each question-answers pair in test set
        run_pipeline(p, top_k_retriever=topk)
        # Get metrics
        topk_results[topk] = {"recall": p.eval_retriever.recall}

    return pd.DataFrame.from_dict(topk_results, orient="index")


es_topk_df = evaluate_retriever(es_retriever)
```

If we plot the results, we can see how the recall improves as we increase k:

```
def plot_retriever_eval(dfs, retriever_names):
    fig, ax = plt.subplots()
    for df, retriever_name in zip(dfs, retriever_names):
        df.plot(y="recall", ax=ax, label=retriever_name)
    plt.xticks(df.index)
    plt.ylabel("Top-k Recall")
    plt.xlabel("k")
    plt.show()

plot_retriever_eval([es_topk_df], ["BM25"])
```



From the plot, we can see that there's an inflection point around $k = 5$ and we get almost perfect recall from $k = 10$ onwards. Let's now take a look at retrieving documents with dense vector techniques.

Dense Passage Retrieval

We've seen that we get almost perfect recall when our sparse retriever returns $k = 10$ documents, but can we do better at smaller values of k? The advantage of doing so is that we can pass fewer documents to the reader and thereby reduce the overall latency of our QA pipeline. A well-known limitation of sparse retrievers like BM25 is that they can fail to capture the relevant documents if the user query contains terms that don't match exactly those of the review. One promising alternative is to use dense embeddings to represent the question and document, and the current state of the art is an architecture known as *Dense Passage Retrieval* (DPR).[14] The main idea behind DPR is to use two BERT models as encoders for the question and the passage. As illustrated in Figure 7-10, these encoders map the input text into a d-dimensional vector representation of the `[CLS]` token.

14 V. Karpukhin et al., "Dense Passage Retrieval for Open-Domain Question Answering", (2020).

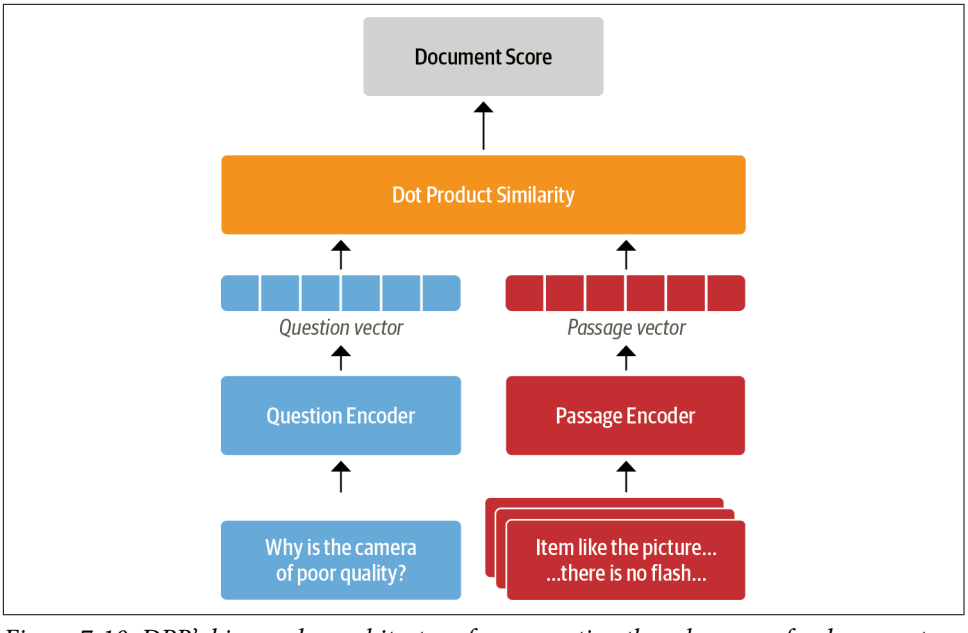


Figure 7-10. DPR's bi-encoder architecture for computing the relevance of a document and query

In Haystack, we can initialize a retriever for DPR in a similar way to what we did for BM25. In addition to specifying the document store, we also need to pick the BERT encoders for the question and passage. These encoders are trained by giving them questions with relevant (positive) passages and irrelevant (negative) passages, where the goal is to learn that relevant question-passage pairs have a higher similarity. For our use case, we'll use encoders that have been fine-tuned on the NQ corpus in this way:

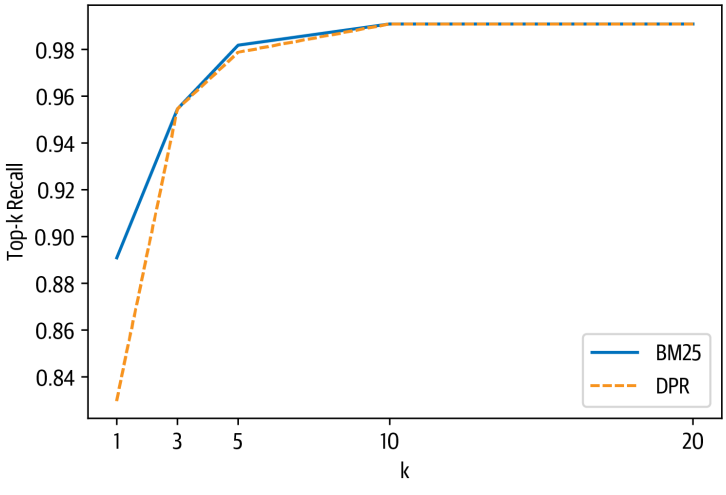
```
from haystack.retriever.dense import DensePassageRetriever

dpr_retriever = DensePassageRetriever(document_store=document_store,
    query_embedding_model="facebook/dpr-question_encoder-single-nq-base",
    passage_embedding_model="facebook/dpr-ctx_encoder-single-nq-base",
    embed_title=False)
```

Here we've also set `embed_title=False` since concatenating the document's title (i.e., `item_id`) doesn't provide any additional information because we filter per product. Once we've initialized the dense retriever, the next step is to iterate over all the indexed documents in our Elasticsearch index and apply the encoders to update the embedding representation. This can be done as follows:

```
document_store.update_embeddings(retriever=dpr_retriever)
```

We're now set to go! We can evaluate the dense retriever in the same way we did for BM25 and compare the top-k recall:

```
dpr_topk_df = evaluate_retriever(dpr_retriever)
plot_retriever_eval([es_topk_df, dpr_topk_df], ["BM25", "DPR"])
```



Here we can see that DPR does not provide a boost in recall over BM25 and saturates around $k = 3$.

Performing similarity search of the embeddings can be sped up by using Facebook's FAISS library as the document store. Similarly, the performance of the DPR retriever can be improved by fine-tuning on the target domain. If you'd like to learn how to fine-tune DPR, check out the Haystack tutorial.

Now that we've explored the evaluation of the retriever, let's turn to evaluating the reader.

Evaluating the Reader

In extractive QA, there are two main metrics that are used for evaluating readers:

Exact Match (EM)
: A binary metric that gives EM = 1 if the characters in the predicted and ground truth answers match exactly, and EM = 0 otherwise. If no answer is expected, the model gets EM = 0 if it predicts any text at all.

F_1-score
: Measures the harmonic mean of the precision and recall.

Let's see how these metrics work by importing some helper functions from FARM and applying them to a simple example:

```
from farm.evaluation.squad_evaluation import compute_f1, compute_exact

pred = "about 6000 hours"
label = "6000 hours"
print(f"EM: {compute_exact(label, pred)}")
print(f"F1: {compute_f1(label, pred)}")
```

```
EM: 0
F1: 0.8
```

Under the hood, these functions first normalize the prediction and label by removing punctuation, fixing whitespace, and converting to lowercase. The normalized strings are then tokenized as a bag-of-words, before finally computing the metric at the token level. From this simple example we can see that EM is a much stricter metric than the F_1-score: adding a single token to the prediction gives an EM of zero. On the other hand, the F_1-score can fail to catch truly incorrect answers. For example, if our predicted answer span is "about 6000 dollars", then we get:

```
pred = "about 6000 dollars"
print(f"EM: {compute_exact(label, pred)}")
print(f"F1: {compute_f1(label, pred)}")
```

```
EM: 0
F1: 0.4
```

Relying on just the F_1-score is thus misleading, and tracking both metrics is a good strategy to balance the trade-off between underestimating (EM) and overestimating (F_1-score) model performance.

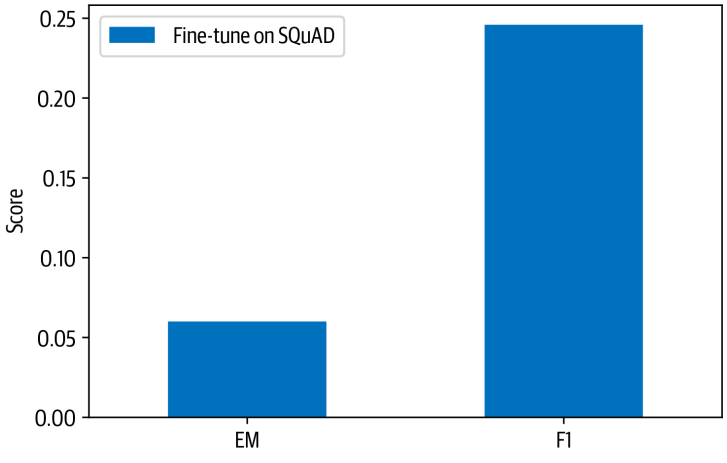
Now in general, there are multiple valid answers per question, so these metrics are calculated for each question-answer pair in the evaluation set, and the best score is selected over all possible answers. The overall EM and F_1 scores for the model are then obtained by averaging over the individual scores of each question-answer pair.

To evaluate the reader we'll create a new pipeline with two nodes: a reader node and a node to evaluate the reader. We'll use the `EvalReader` class that takes the predictions from the reader and computes the corresponding EM and F_1 scores. To compare with the SQuAD evaluation, we'll take the best answers for each query with the `top_1_em` and `top_1_f1` metrics that are stored in `EvalAnswers`:

```
from haystack.eval import EvalAnswers

def evaluate_reader(reader):
    score_keys = ['top_1_em', 'top_1_f1']
    eval_reader = EvalAnswers(skip_incorrect_retrieval=False)
    pipe = Pipeline()
    pipe.add_node(component=reader, name="QAReader", inputs=["Query"])
    pipe.add_node(component=eval_reader, name="EvalReader", inputs=["QAReader"])

    for l in labels_agg:
        doc = document_store.query(l.question,
                                   filters={"question_id":[l.origin]})
        _ = pipe.run(query=l.question, documents=doc, labels=l)

    return {k:v for k,v in eval_reader.__dict__.items() if k in score_keys}

reader_eval = {}
reader_eval["Fine-tune on SQuAD"] = evaluate_reader(reader)
```

Notice that we specified `skip_incorrect_retrieval=False`. This is to ensure that the retriever always passes the context to the reader (as in the SQuAD evaluation). Now that we've run every question through the reader, let's print the scores:

```
def plot_reader_eval(reader_eval):
    fig, ax = plt.subplots()
    df = pd.DataFrame.from_dict(reader_eval)
    df.plot(kind="bar", ylabel="Score", rot=0, ax=ax)
    ax.set_xticklabels(["EM", "F1"])
    plt.legend(loc='upper left')
    plt.show()

plot_reader_eval(reader_eval)
```



OK, it seems that the fine-tuned model performs significantly worse on SubjQA than on SQuAD 2.0, where MiniLM achieves EM and F_1 scores of 76.1 and 79.5, respectively. One reason for the performance drop is that customer reviews are quite different from the Wikipedia articles the SQuAD 2.0 dataset is generated from, and the language they use is often informal. Another factor is likely the inherent subjectivity of our dataset, where both questions and answers differ from the factual information contained in Wikipedia. Let's look at how to fine-tune a model on a dataset to get better results with domain adaptation.

Domain Adaptation

Although models that are fine-tuned on SQuAD will often generalize well to other domains, we've seen that for SubjQA the EM and F_1 scores of our model were much worse than for SQuAD. This failure to generalize has also been observed in other extractive QA datasets and is understood as evidence that transformer models are particularly adept at overfitting to SQuAD.[15] The most straightforward way to improve the reader is by fine-tuning our MiniLM model further on the SubjQA training set. The `FARMReader` has a `train()` method that is designed for this purpose and expects the data to be in SQuAD JSON format, where all the question-answer pairs are grouped together for each item as illustrated in Figure 7-11.

15 D. Yogatama et al., "Learning and Evaluating General Linguistic Intelligence", (2019).

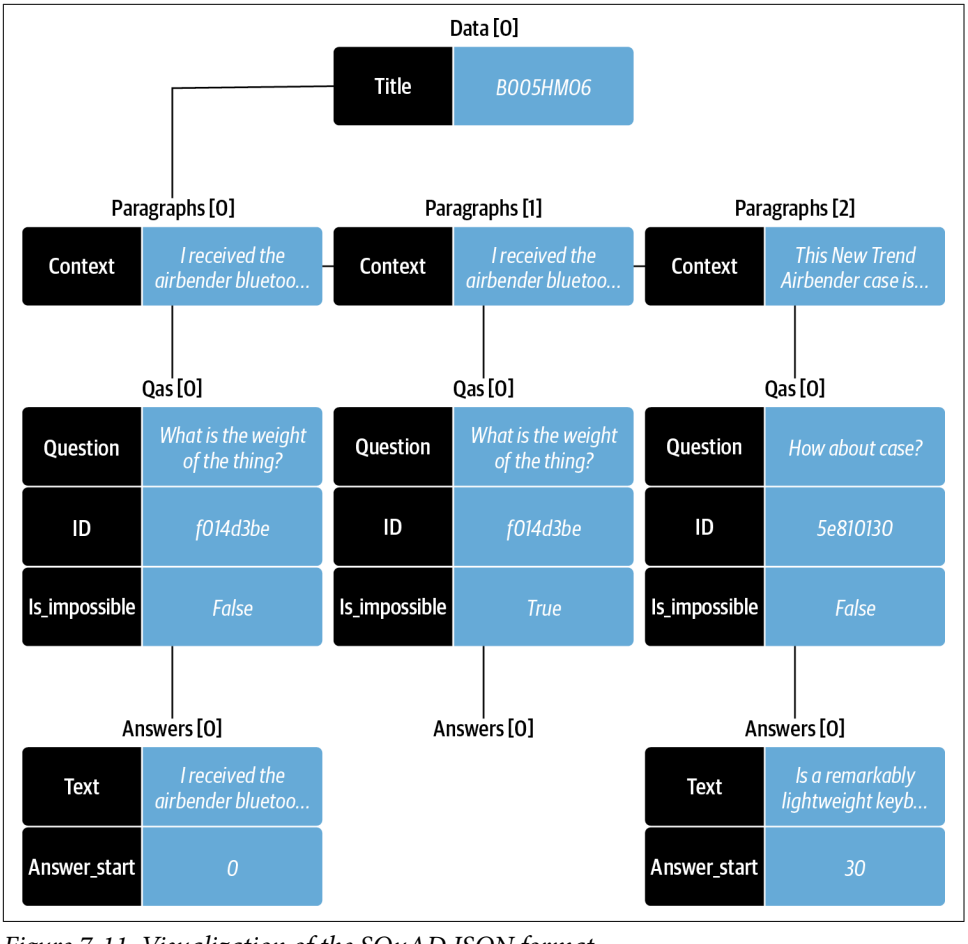


Figure 7-11. Visualization of the SQuAD JSON format

This is quite a complex data format, so we'll need a few functions and some Pandas magic to help us do the conversion. The first thing we need to do is implement a function that can create the `paragraphs` array associated with each product ID. Each element in this array contains a single context (i.e., review) and a `qas` array of question-answer pairs. Here's a function that builds up the `paragraphs` array:

```
def create_paragraphs(df):
    paragraphs = []
    id2context = dict(zip(df["review_id"], df["context"]))
    for review_id, review in id2context.items():
        qas = []
        # Filter for all question-answer pairs about a specific context
        review_df = df.query(f"review_id == '{review_id}'")
        id2question = dict(zip(review_df["id"], review_df["question"]))
        # Build up the qas array
```

```
        for qid, question in id2question.items():
            # Filter for a single question ID
            question_df = df.query(f"id == '{qid}'").to_dict(orient="list")
            ans_start_idxs = question_df["answers.answer_start"][0].tolist()
            ans_text = question_df["answers.text"][0].tolist()
            # Fill answerable questions
            if len(ans_start_idxs):
                answers = [
                    {"text": text, "answer_start": answer_start}
                    for text, answer_start in zip(ans_text, ans_start_idxs)]
                is_impossible = False
            else:
                answers = []
                is_impossible = True
            # Add question-answer pairs to qas
            qas.append({"question": question, "id": qid,
                        "is_impossible": is_impossible, "answers": answers})
        # Add context and question-answer pairs to paragraphs
        paragraphs.append({"qas": qas, "context": review})
    return paragraphs
```

Now, when we apply to the rows of a `DataFrame` associated with a single product ID, we get the SQuAD format:

```
product = dfs["train"].query("title == 'B00001P4ZH'")
create_paragraphs(product)
```

```
[{'qas': [{'question': 'How is the bass?',
    'id': '2543d296da9766d8d17d040ecc781699',
    'is_impossible': True,
    'answers': []}],
  'context': 'I have had Koss headphones ...',
    'id': 'd476830bf9282e2b9033e2bb44bbb995',
    'is_impossible': False,
    'answers': [{'text': 'Bass is weak as expected', 'answer_start': 1302},
     {'text': 'Bass is weak as expected, even with EQ adjusted up',
      'answer_start': 1302}]}],
  'context': 'To anyone who hasn\'t tried all ...'},
 {'qas': [{'question': 'How is the bass?',
    'id': '455575557886d6dfeea5aa19577e5de4',
    'is_impossible': False,
    'answers': [{'text': 'The only fault in the sound is the bass',
      'answer_start': 650}]}],
  'context': "I have had many sub-$100 headphones ..."}]
```

The final step is to then apply this function to each product ID in the `DataFrame` of each split. The following `convert_to_squad()` function does this trick and stores the result in an *electronics-{split}.json* file:

```
import json

def convert_to_squad(dfs):
    for split, df in dfs.items():
```

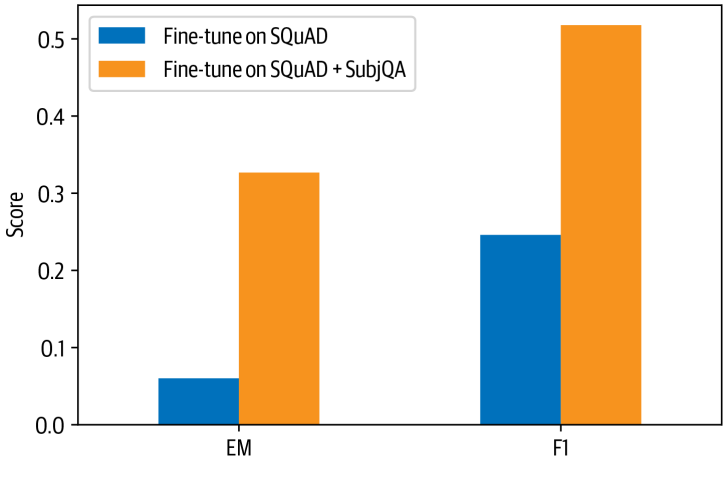
```
        subjqa_data = {}
        # Create `paragraphs` for each product ID
        groups = (df.groupby("title").apply(create_paragraphs)
            .to_frame(name="paragraphs").reset_index())
        subjqa_data["data"] = groups.to_dict(orient="records")
        # Save the result to disk
        with open(f"electronics-{split}.json", "w+", encoding="utf-8") as f:
            json.dump(subjqa_data, f)

convert_to_squad(dfs)
```

Now that we have the splits in the right format, let's fine-tune our reader by specifying the locations of the train and dev splits, along with where to save the fine-tuned model:

```
train_filename = "electronics-train.json"
dev_filename = "electronics-validation.json"

reader.train(data_dir=".", use_gpu=True, n_epochs=1, batch_size=16,
             train_filename=train_filename, dev_filename=dev_filename)
```

With the reader fine-tuned, let's now compare its performance on the test set against our baseline model:

```
reader_eval["Fine-tune on SQuAD + SubjQA"] = evaluate_reader(reader)
plot_reader_eval(reader_eval)
```



Wow, domain adaptation has increased our EM score by a factor of six and more than doubled the F_1-score! At this point, you might be wondering why we didn't just fine-tune a pretrained language model directly on the SubjQA training set. One reason is that we only have 1,295 training examples in SubjQA while SQuAD has over 100,000, so we might run into challenges with overfitting. Nevertheless, let's take a look at what naive fine-tuning produces. For a fair comparison, we'll use the same language model

that was used for fine-tuning our baseline on SQuAD. As before, we'll load up the model with the `FARMReader`:

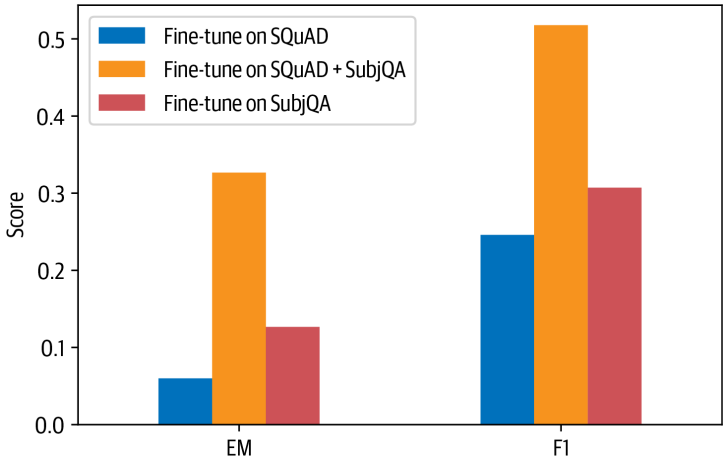
```
minilm_ckpt = "microsoft/MiniLM-L12-H384-uncased"
minilm_reader = FARMReader(model_name_or_path=minilm_ckpt, progress_bar=False,
                           max_seq_len=max_seq_length, doc_stride=doc_stride,
                           return_no_answer=True)
```

Next, we fine-tune for one epoch:

```
minilm_reader.train(data_dir=".", use_gpu=True, n_epochs=1, batch_size=16,
            train_filename=train_filename, dev_filename=dev_filename)
```

and include the evaluation on the test set:

```
reader_eval["Fine-tune on SubjQA"] = evaluate_reader(minilm_reader)
plot_reader_eval(reader_eval)
```



We can see that fine-tuning the language model directly on SubjQA results in considerably worse performance than fine-tuning on SQuAD and SubjQA.

When dealing with small datasets, it is best practice to use cross-validation when evaluating transformers as they can be prone to overfitting. You can find an example of how to perform cross-validation with SQuAD-formatted datasets in the FARM repository.

Evaluating the Whole QA Pipeline

Now that we've seen how to evaluate the reader and retriever components individually, let's tie them together to measure the overall performance of our pipeline. To do so, we'll need to augment our retriever pipeline with nodes for the reader and its

evaluation. We've seen that we get almost perfect recall at $k = 10$, so we can fix this value and assess the impact this has on the reader's performance (since it will now receive multiple contexts per query compared to the SQuAD-style evaluation):

```
# Initialize retriever pipeline
pipe = EvalRetrieverPipeline(es_retriever)
# Add nodes for reader
eval_reader = EvalAnswers()
pipe.pipeline.add_node(component=reader, name="QAReader",
                       inputs=["EvalRetriever"])
pipe.pipeline.add_node(component=eval_reader, name="EvalReader",
                       inputs=["QAReader"])
# Evaluate!
run_pipeline(pipe)
# Extract metrics from reader
reader_eval["QA Pipeline (top-1)"] = {
    k:v for k,v in eval_reader.__dict__.items()
    if k in ["top_1_em", "top_1_f1"]}
```

We can then compare the top 1 EM and F_1 scores for the model to predict an answer in the documents returned by the retriever in Figure 7-12.

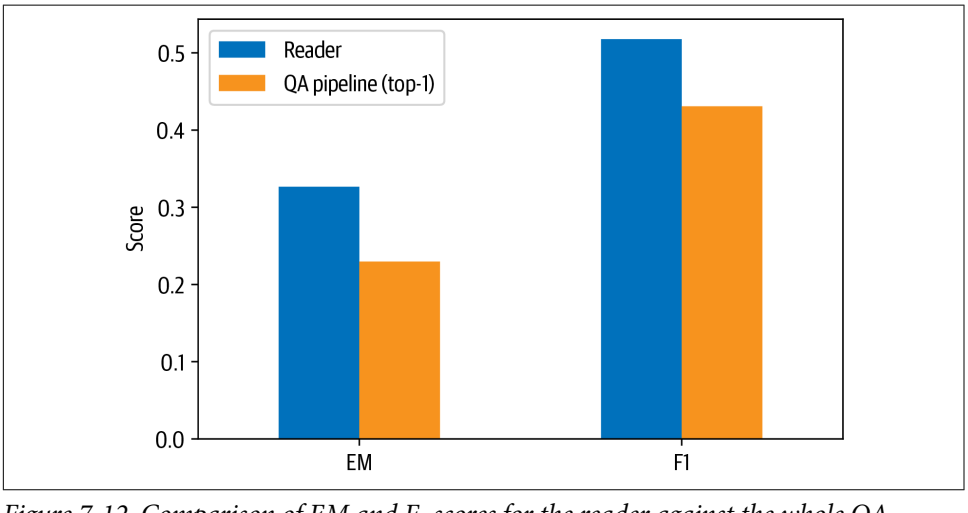


Figure 7-12. Comparison of EM and F_1 scores for the reader against the whole QA pipeline

From this plot we can see the effect that the retriever has on the overall performance. In particular, there is an overall degradation compared to matching the question-context pairs, as is done in the SQuAD-style evaluation. This can be circumvented by increasing the number of possible answers that the reader is allowed to predict.

Until now we have only extracted answer spans from the context, but in general it could be that bits and pieces of the answer are scattered throughout the document

and we would like our model to synthesize these fragments into a single coherent answer. Let's have a look at how we can use generative QA to succeed at this task.

Going Beyond Extractive QA

One interesting alternative to extracting answers as spans of text in a document is to generate them with a pretrained language model. This approach is often referred to as *abstractive* or *generative QA* and has the potential to produce better-phrased answers that synthesize evidence across multiple passages. Although less mature than extractive QA, this is a fast-moving field of research, so chances are that these approaches will be widely adopted in industry by the time you are reading this! In this section we'll briefly touch on the current state of the art: *retrieval-augmented generation* (RAG).[16]

RAG extends the classic retriever-reader architecture that we've seen in this chapter by swapping the reader for a *generator* and using DPR as the retriever. The generator is a pretrained sequence-to-sequence transformer like T5 or BART that receives latent vectors of documents from DPR and then iteratively generates an answer based on the query and these documents. Since DPR and the generator are differentiable, the whole process can be fine-tuned end-to-end as illustrated in Figure 7-13.

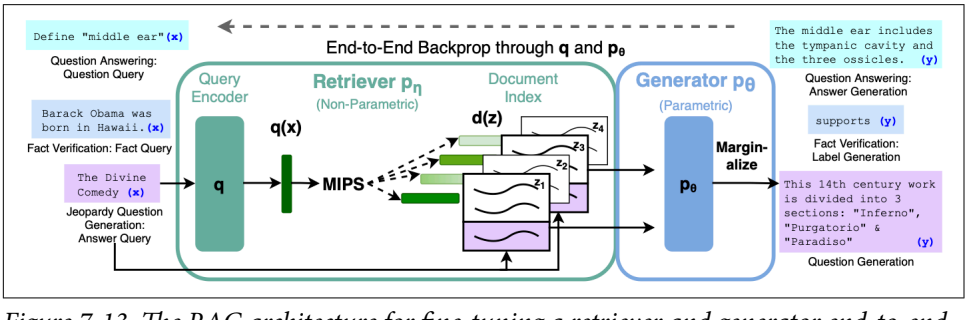


Figure 7-13. The RAG architecture for fine-tuning a retriever and generator end-to-end (courtesy of Ethan Perez)

To show RAG in action we'll use the `DPRetriever` from earlier, so we just need to instantiate a generator. There are two types of RAG models to choose from:

RAG-Sequence
: Uses the same retrieved document to generate the complete answer. In particular, the top k documents from the retriever are fed to the generator, which produces an output sequence for each document, and the result is marginalized to obtain the best answer.

16 P. Lewis et al., "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks", (2020).

RAG-Token
: Can use a different document to generate each token in the answer. This allows the generator to synthesize evidence from multiple documents.

Since RAG-Token models tend to perform better than RAG-Sequence ones, we'll use the token model that was fine-tuned on NQ as our generator. Instantiating a generator in Haystack is similar to instantiating the reader, but instead of specifying the `max_seq_length` and `doc_stride` parameters for a sliding window over the contexts, we specify hyperparameters that control the text generation:

```
from haystack.generator.transformers import RAGenerator

generator = RAGenerator(model_name_or_path="facebook/rag-token-nq",
                        embed_title=False, num_beams=5)
```

Here `num_beams` specifies the number of beams to use in beam search (text generation is covered at length in Chapter 5). As we did with the DPR retriever, we don't embed the document titles since our corpus is always filtered per product ID.

The next thing to do is tie together the retriever and generator using Haystack's `GenerativeQAPipeline`:

```
from haystack.pipeline import GenerativeQAPipeline

pipe = GenerativeQAPipeline(generator=generator, retriever=dpr_retriever)
```

In RAG, both the query encoder and the generator are trained end-to-end, while the context encoder is frozen. In Haystack, the `GenerativeQAPipeline` uses the query encoder from `RAGenerator` and the context encoder from `DensePassageRetriever`.

Let's now give RAG a spin by feeding in some queries about the Amazon Fire tablet from before. To simplify the querying, we'll write a simple function that takes the query and prints out the top answers:

```
def generate_answers(query, top_k_generator=3):
    preds = pipe.run(query=query, top_k_generator=top_k_generator,
                     top_k_retriever=5, filters={"item_id":["B0074BW614"]})
    print(f"Question: {preds['query']} \n")
    for idx in range(top_k_generator):
        print(f"Answer {idx+1}: {preds['answers'][idx]['answer']}")
```

OK, now we're ready to give it a test:

```
generate_answers(query)
```

```
Question: Is it good for reading?

Answer 1:  the screen is absolutely beautiful
```

```
Answer 2:  the Screen is absolutely beautiful
Answer 3:  Kindle fire
```

This result isn't too bad for an answer, but it does suggest that the subjective nature of the question is confusing the generator. Let's try with something a bit more factual:

```
generate_answers("What is the main drawback?")
```

```
Question: What is the main drawback?

Answer 1:  the price
Answer 2:  no flash support
Answer 3:  the cost
```

This is more sensible! To get better results we could fine-tune RAG end-to-end on SubjQA; we'll leave this as an exercise, but if you're interested in exploring it there are scripts in the 🤗 Transformers repository to help you get started.

Conclusion

Well, that was a whirlwind tour of QA, and you probably have many more questions that you'd like answered (pun intended!). In this chapter, we discussed two approaches to QA (extractive and generative) and examined two different retrieval algorithms (BM25 and DPR). Along the way, we saw that domain adaptation can be a simple technique to boost the performance of our QA system by a significant margin, and we looked at a few of the most common metrics that are used for evaluating such systems. Although we focused on closed-domain QA (i.e., a single domain of electronic products), the techniques in this chapter can easily be generalized to the open-domain case; we recommend reading Cloudera's excellent Fast Forward QA series to see what's involved.

Deploying QA systems in the wild can be a tricky business to get right, and our experience is that a significant part of the value comes from first providing end users with useful search capabilities, followed by an extractive component. In this respect, the reader can be used in novel ways beyond answering on-demand user queries. For example, researchers at Grid Dynamics were able to use their reader to automatically extract a set of pros and cons for each product in a client's catalog. They also showed that a reader can be used to extract named entities in a zero-shot fashion by creating queries like "What kind of camera?" Given its infancy and subtle failure modes, we recommend exploring generative QA only once the other two approaches have been exhausted. This "hierarchy of needs" for tackling QA problems is illustrated in Figure 7-14.

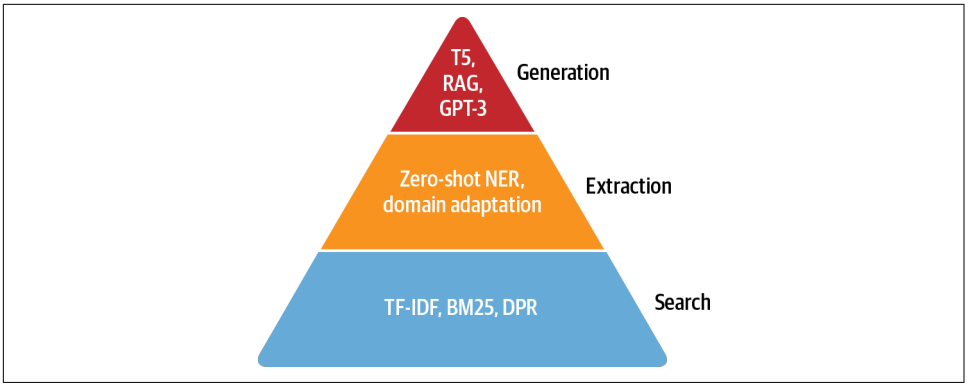


Figure 7-14. The QA hierarchy of needs

Looking ahead, one exciting research area is *multimodal QA*, which involves QA over multiple modalities like text, tables, and images. As described in the MultiModalQA benchmark,[17] such systems could enable users to answer complex questions that integrate information across different modalities, like "When was the famous painting with two touching fingers completed?" Another area with practical business applications is QA over a *knowledge graph*, where the nodes of the graph correspond to real-world entities and their relations are defined by the edges. By encoding factoids as (*subject*, *predicate*, *object*) triples, one can use the graph to answer questions about a missing element. For an example that combines transformers with knowledge graphs, see the Haystack tutorials. One more promising direction is *automatic question generation* as a way to do some form of unsupervised/weakly supervised training using unlabeled data or data augmentation. Two recent examples include the papers on the Probably Answered Questions (PAQ) benchmark and synthetic data augmentation for cross-lingual settings.[18]

In this chapter we've seen that in order to successfully use QA models for real-world use cases we need to apply a few tricks, such as implementing a fast retrieval pipeline to make predictions in near real time. Still, applying a QA model to a handful of preselected documents can take a couple of seconds on production hardware. Although this may not sound like much, imagine how different your experience would be if you had to wait a few seconds to get the results of a Google search—a few seconds of wait time can decide the fate of your transformer-powered application. In the next chapter we'll have a look at a few methods to accelerate model predictions further.

17 A. Talmor et al., "MultiModalQA: Complex Question Answering over Text, Tables and Images", (2021).

18 P. Lewis et al., "PAQ: 65 Million Probably-Asked Questions and What You Can Do with Them", (2021); A. Riabi et al., "Synthetic Data Augmentation for Zero-Shot Cross-Lingual Question Answering", (2020).

CHAPTER 11

Future Directions

Throughout this book we've explored the powerful capabilities of transformers across a wide range of NLP tasks. In this final chapter, we'll shift our perspective and look at some of the current challenges with these models and the research trends that are trying to overcome them. In the first part we explore the topic of scaling up transformers, both in terms of model and corpus size. Then we turn our attention toward various techniques that have been proposed to make the self-attention mechanism more efficient. Finally, we explore the emerging and exciting field of *multimodal transformers*, which can model inputs across multiple domains like text, images, and audio.

Scaling Transformers

In 2019, the researcher Richard Sutton wrote a provocative essay entitled "The Bitter Lesson" in which he argued that:

> The biggest lesson that can be read from 70 years of AI research is that general methods that leverage computation are ultimately the most effective, and by a large margin… . Seeking an improvement that makes a difference in the shorter term, researchers seek to leverage their human knowledge of the domain, but the only thing that matters in the long run is the leveraging of computation. These two need not run counter to each other, but in practice they tend to… . And the human-knowledge approach tends to complicate methods in ways that make them less suited to taking advantage of general methods leveraging computation.

The essay provides several historical examples, such as playing chess or Go, where the approach of encoding human knowledge within AI systems was ultimately outdone by increased computation. Sutton calls this the "bitter lesson" for the AI research field:

> We have to learn the bitter lesson that building in how we think we think does not work in the long run.… One thing that should be learned from the bitter lesson is the great power of general purpose methods, of methods that continue to scale with increased computation even as the available computation becomes very great. The two methods that seem to scale arbitrarily in this way are *search* and *learning*.

There are now signs that a similar lesson is at play with transformers; while many of the early BERT and GPT descendants focused on tweaking the architecture or pretraining objectives, the best-performing models in mid-2021, like GPT-3, are essentially basic scaled-up versions of the original models without many architectural modifications. In Figure 11-1 you can see a timeline of the development of the largest models since the release of the original Transformer architecture in 2017, which shows that model size has increased by over four orders of magnitude in just a few years!

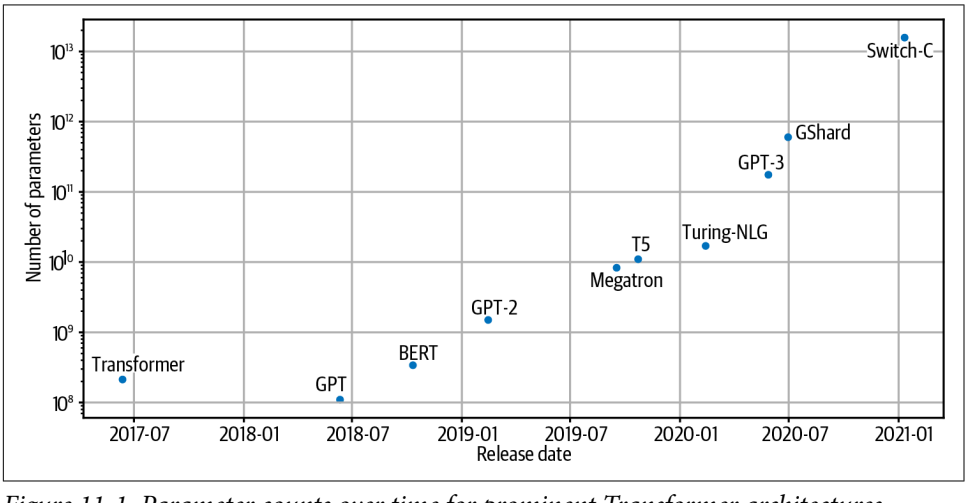


Figure 11-1. Parameter counts over time for prominent Transformer architectures

This dramatic growth is motivated by empirical evidence that large language models perform better on downstream tasks and that interesting capabilities such as zero-shot and few-shot learning emerge in the 10- to 100-billion parameter range. However, the number of parameters is not the only factor that affects model performance; the amount of compute and training data must also be scaled in tandem to train these monsters. Given that large language models like GPT-3 are estimated to cost $4.6 million to train, it is clearly desirable to be able to estimate the model's performance in advance. Somewhat surprisingly, the performance of language models appears to

obey a *power law relationship* with model size and other factors that is codified in a set of scaling laws.[1] Let's take a look at this exciting area of research.

Scaling Laws

Scaling laws allow one to empirically quantify the "bigger is better" paradigm for language models by studying their behavior with varying compute budget C, dataset size D, and model size N.[2] The basic idea is to chart the dependence of the cross-entropy loss L on these three factors and determine if a relationship emerges. For autoregressive models like those in the GPT family, the resulting loss curves are shown in Figure 11-2, where each blue curve represents the training run of a single model.

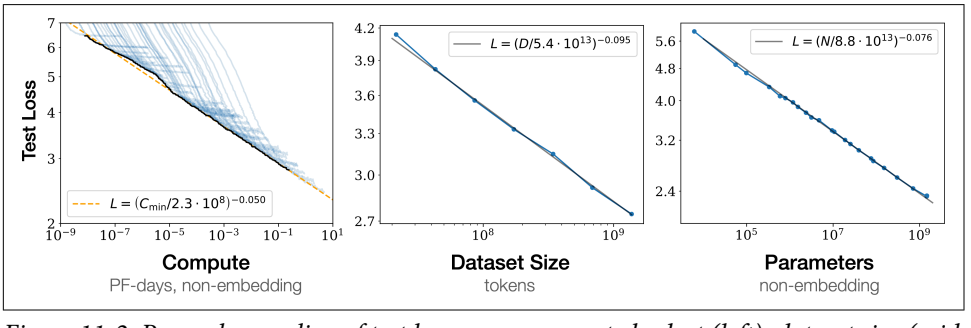


Figure 11-2. Power-law scaling of test loss versus compute budget (left), dataset size (middle), and model size (right) (courtesy of Jared Kaplan)

From these loss curves we can draw a few conclusions about:

The relationship of performance and scale
: Although many NLP researchers focus on architectural tweaks or hyperparameter optimization (like tuning the number of layers or attention heads) to improve performance on a fixed set of datasets, the implication of scaling laws is that a more productive path toward better models is to focus on increasing N, C, and D in tandem.

Smooth power laws
: The test loss L has a power law relationship with each of N, C, and D across several orders of magnitude (power law relationships are linear on a log-log scale). For $X = N, C, D$ we can express these power law relationships as $L(X) \sim 1/X^{\alpha}$, where α is a scaling exponent that is determined by a fit to the loss curves shown

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

2 The dataset size is measured in the number of tokens, while the model size excludes parameters from the embedding layers.

in Figure 11-2.[3] Typical values for α_X lie in the 0.05–0.095 range, and one attractive feature of these power laws is that the early part of a loss curve can be extrapolated to predict what the approximate loss would be if training was conducted for much longer.

Sample efficiency
: Large models are able to reach the same performance as smaller models with a smaller number of training steps. This can be seen by comparing the regions where a loss curve plateaus over some number of training steps, which indicates one gets diminishing returns in performance compared to simply scaling up the model.

Somewhat surprisingly, scaling laws have also been observed for other modalities, like images, videos, and mathematical problem solving, as illustrated in Figure 11-3.

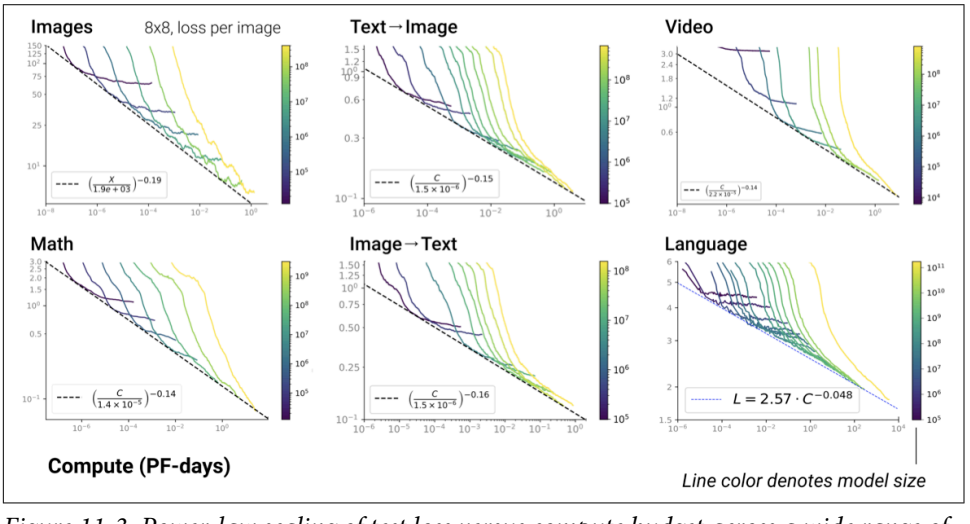


Figure 11-3. Power-law scaling of test loss versus compute budget across a wide range of modalities (courtesy of Tom Henighan)

Whether power-law scaling is a universal property of transformer language models is currently unknown. For now, we can use scaling laws as a tool to extrapolate large, expensive models without having to explicitly train them. However, scaling isn't quite as easy as it sounds. Let's now look at a few challenges that crop up when charting this frontier.

3 T. Henighan et al., "Scaling Laws for Autoregressive Generative Modeling", (2020).

Challenges with Scaling

While scaling up sounds simple in theory ("just add more layers!"), in practice there are many difficulties. Here are a few of the biggest challenges you're likely to encounter when scaling language models:

Infrastructure
: Provisioning and managing infrastructure that potentially spans hundreds or thousands of nodes with as many GPUs is not for the faint-hearted. Are the required number of nodes available? Is communication between nodes a bottleneck? Tackling these issues requires a very different skill set than that found in most data science teams, and typically involves specialized engineers familiar with running large-scale, distributed experiments.

Cost
: Most ML practitioners have experienced the feeling of waking up in the middle of the night in a cold sweat, remembering they forgot to shut down that fancy GPU on the cloud. This feeling intensifies when running large-scale experiments, and most companies cannot afford the teams and resources necessary to train models at the largest scales. Training a single GPT-3-sized model can cost several million dollars, which is not the kind of pocket change that many companies have lying around.[4]

Dataset curation
: A model is only as good as the data it is trained on. Training large models requires large, high-quality datasets. When using terabytes of text data it becomes harder to make sure the dataset contains high-quality text, and even preprocessing becomes challenging. Furthermore, one needs to ensure that there is a way to control biases like sexism and racism that these language models can acquire when trained on large-scale webtext corpora. Another type of consideration revolves around licensing issues with the training data and personal information that can be embedded in large text datasets.

Model evaluation
: Once the model is trained, the challenges don't stop. Evaluating the model on downstream tasks again requires time and resources. In addition, you'll want to probe the model for biased and toxic generations, even if you are confident that you created a clean dataset. These steps take time and need to be carried out thoroughly to minimize the risks of adverse effects later on.

4 However, recently a distributed deep learning framework has been proposed that enables smaller groups to pool their computational resources and pretrain models in a collaborative fashion. See M. Diskin et al., "Distributed Deep Learning in Open Collaborations", (2021).

Deployment
: Finally, serving large language models also poses a significant challenge. In Chapter 8 we looked at a few approaches, such as distillation, pruning, and quantization, to help with these issues. However, this may not be enough if you are starting with a model that is hundreds of gigabytes in size. Hosted services such as the OpenAI API or Hugging Face's Accelerated Inference API are designed to help companies that cannot or do not want to deal with these deployment challenges.

This is by no means an exhaustive list, but it should give you an idea of the kinds of considerations and challenges that go hand in hand with scaling language models to ever larger sizes. While most of these efforts are centralized around a few institutions that have the resources and know-how to push the boundaries, there are currently two community-led projects that aim to produce and probe large language models in the open:

BigScience
: This is a one-year-long research workshop that runs from 2021 to 2022 and is focused on large language models. The workshop aims to foster discussions and reflections around the research questions surrounding these models (capabilities, limitations, potential improvements, bias, ethics, environmental impact, role in the general AI/cognitive research landscape) as well as the challenges around creating and sharing such models and datasets for research purposes and among the research community. The collaborative tasks involve creating, sharing, and evaluating a large multilingual dataset and a large language model. An unusually large compute budget was allocated for these collaborative tasks (several million GPU hours on several thousands GPUs). If successful, this workshop will run again in the future, focusing on involving an updated or different set of collaborative tasks. If you want to join the effort, you can find more information at the project's website.

EleutherAI
: This is a decentralized collective of volunteer researchers, engineers, and developers focused on AI alignment, scaling, and open source AI research. One of its aims is to train and open-source a GPT-3-sized model, and the group has already released some impressive models like GPT-Neo and GPT-J, which is a 6-billion-parameter model and currently the best-performing publicly available transformer in terms of zero-shot performance. You can find more information at EleutherAI's website.

Now that we've explored how to scale transformers across compute, model size, and dataset size, let's examine another active area of research: making self-attention more efficient.

Attention Please!

We've seen throughout this book that the self-attention mechanism plays a central role in the architecture of transformers; after all, the original Transformer paper is called "Attention Is All You Need"! However, there is a key challenge associated with self-attention: since the weights are generated from pairwise comparisons of all the tokens in a sequence, this layer becomes a computational bottleneck when trying to process long documents or apply transformers to domains like speech processing or computer vision. In terms of time and memory complexity, the self-attention layer of the Transformer architecture naively scales like $\mathcal{O}\left(n^2\right)$, where n is the length of the sequence.[5]

As a result, much of the recent research on transformers has focused on making self-attention more efficient. The research directions are broadly clustered in Figure 11-4.

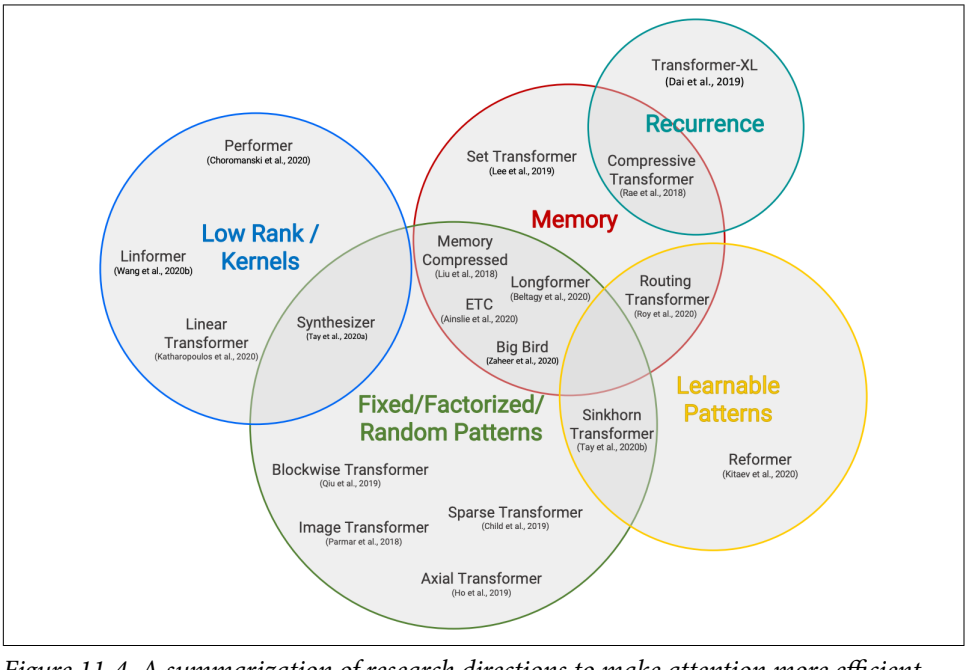


Figure 11-4. A summarization of research directions to make attention more efficient (courtesy of Yi Tay et al.)[6]

5 Although standard implementations of self-attention have $O\left(n^2\right)$ time and memory complexity, a recent paper by Google researchers shows that the memory complexity can be reduced to $O(\log n)$ via a simple reordering of the operations.

6 Yi Tay et al., "Efficient Transformers: A Survey", (2020).

A common pattern is to make attention more efficient by introducing sparsity into the attention mechanism or by applying kernels to the attention matrix. Let's take a quick look at some of the most popular approaches to make self-attention more efficient, starting with sparsity.

Sparse Attention

One way to reduce the number of computations that are performed in the self-attention layer is to simply limit the number of query-key pairs that are generated according to some predefined pattern. There have been many sparsity patterns explored in the literature, but most of them can be decomposed into a handful of "atomic" patterns illustrated in Figure 11-5.

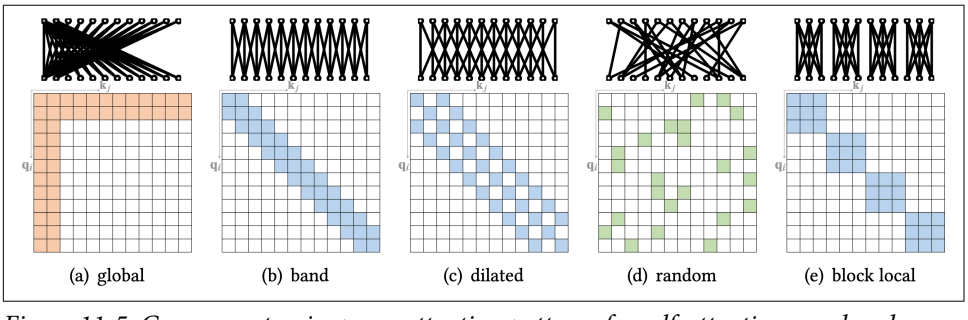


Figure 11-5. Common atomic sparse attention patterns for self-attention: a colored square means the attention score is calculated, while a blank square means the score is discarded (courtesy of Tianyang Lin)

We can describe these patterns as follows:[7]

Global attention
: Defines a few special tokens in the sequence that are allowed to attend to all other tokens

Band attention
: Computes attention over a diagonal band

Dilated attention
: Skips some query-key pairs by using a dilated window with gaps

Random attention
: Randomly samples a few keys for each query to compute attention scores

7 T. Lin et al., "A Survey of Transformers", (2021).

Block local attention
: Divides the sequence into blocks and restricts attention within these blocks

In practice, most transformer models with sparse attention use a mix of the atomic sparsity patterns shown in Figure 11-5 to generate the final attention matrix. As illustrated in Figure 11-6, models like Longformer use a mix of global and band attention, while BigBird adds random attention to the mix. Introducing sparsity into the attention matrix enables these models to process much longer sequences; in the case of Longformer and BigBird the maximum sequence length is 4,096 tokens, which is 8 times larger than BERT!

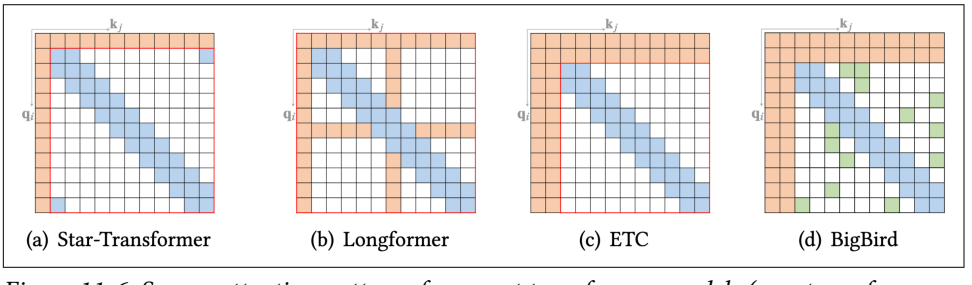


Figure 11-6. Sparse attention patterns for recent transformer models (courtesy of Tianyang Lin)

It is also possible to *learn* the sparsity pattern in a data-driven manner. The basic idea behind such approaches is to cluster the tokens into chunks. For example, Reformer uses a hash function to cluster similar tokens together.

Now that we've seen how sparsity can reduce the complexity of self-attention, let's take a look at another popular approach based on changing the operations directly.

Linearized Attention

An alternative way to make self-attention more efficient is to change the order of operations that are involved in computing the attention scores. Recall that to compute the self-attention scores of the queries and keys we need a similarity function, which for the transformer is just a simple dot product. However, for a general similarity function $\text{sim}(q_i, k_j)$ we can express the attention outputs as the following equation:

$$y_i = \sum_j \frac{\text{sim}(Q_i, K_j)}{\Sigma_k \text{sim}(Q_i, K_k)} V_j$$

The trick behind linearized attention mechanisms is to express the similarity function as a *kernel function* that decomposes the operation into two pieces:

$$\text{sim}\left(Q_j, K_j\right) = \varphi\left(Q_i\right)^T \varphi\left(K_j\right)$$

where φ is typically a high-dimensional feature map. Since $\varphi\left(Q_i\right)$ is independent of j and k, we can pull it under the sums to write the attention outputs as follows:

$$y_i = \frac{\varphi\left(Q_i\right)^T \Sigma_j \varphi\left(K_j\right) V_j^T}{\varphi\left(Q_i\right)^T \Sigma_k \varphi\left(K_k\right)}$$

By first computing $\Sigma_j \varphi\left(K_j\right) V_j^T$ and $\Sigma_k \varphi\left(K_k\right)$, we can effectively linearize the space and time complexity of self-attention! The comparison between the two approaches is illustrated in Figure 11-7. Popular models that implement linearized self-attention include Linear Transformer and Performer.[8]

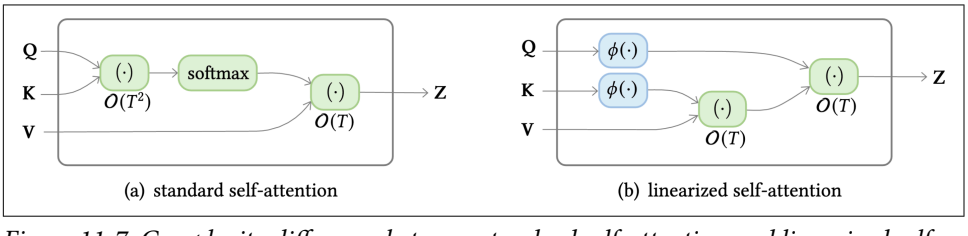


Figure 11-7. Complexity difference between standard self-attention and linearized self-attention (courtesy of Tianyang Lin)

In this section we've seen how Transformer architectures in general and attention in particular can be scaled up to achieve even better performance on a wide range of tasks. In the next section we'll have a look at how transformers are branching out of NLP into other domains such as audio and computer vision.

Going Beyond Text

Using text to train language models has been the driving force behind the success of transformer language models, in combination with transfer learning. On the one hand, text is abundant and enables self-supervised training of large models. On the other hand, textual tasks such as classification and question answering are common,

8 A. Katharopoulos et al., "Transformers Are RNNs: Fast Autoregressive Transformers with Linear Attention", (2020); K. Choromanski et al., "Rethinking Attention with Performers", (2020).

and developing effective strategies for them allows us to address a wide range of real-world problems.

However, there are limits to this approach, including:

Human reporting bias
: The frequencies of events in text may not represent their true frequencies.[9] A model solely trained on text from the internet might have a very distorted image of the world.

Common sense
: Common sense is a fundamental quality of human reasoning, but is rarely written down. As such, language models trained on text might know many facts about the world, but lack basic common-sense reasoning.

Facts
: A probabilistic language model cannot store facts in a reliable way and can produce text that is factually wrong. Similarly, such models can detect named entities, but have no direct way to access information about them.

Modality
: Language models have no way to connect to other modalities that could address the previous points, such as audio or visual signals or tabular data.

So, if we could solve the modality limitations we could potentially address some of the others as well. Recently there has been a lot of progress in pushing transformers to new modalities, and even building multimodal models. In this section we'll highlight a few of these advances.

Vision

Vision has been the stronghold of convolutional neural networks (CNNs) since they kickstarted the deep learning revolution. More recently, transformers have begun to be applied to this domain and to achieve efficiency similar to or better than CNNs. Let's have a look at a few examples.

iGPT

Inspired by the success of the GPT family of models with text, iGPT (short for image GPT) applies the same methods to images.[10] By viewing images as sequences of pixels, iGPT uses the GPT architecture and autoregressive pretraining objective to predict

9 J. Gordon and B. Van Durme, "Reporting Bias and Knowledge Extraction", (2013).

10 M. Chen et al., "Generative Pretraining from Pixels," *Proceedings of the 37th International Conference on Machine Learning* 119 (2020):1691–1703, *https://proceedings.mlr.press/v119/chen20s.html*.

the next pixel values. Pretraining on large image datasets enables iGPT to "autocomplete" partial images, as displayed in Figure 11-8. It also achieves performant results on classification tasks when a classification head is added to the model.

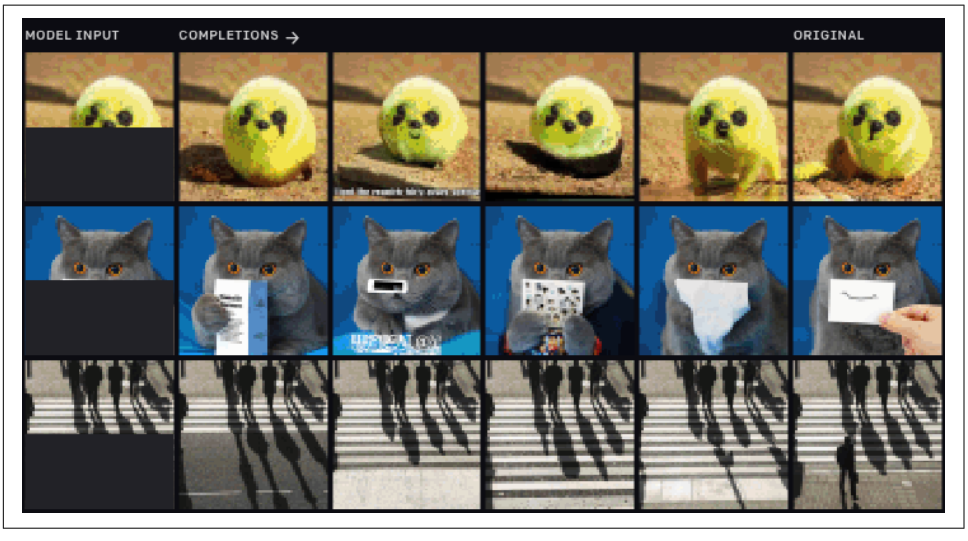


Figure 11-8. Examples of image completions with iGPT (courtesy of Mark Chen)

ViT

We saw that iGPT follows closely the GPT-style architecture and pretraining procedure. Vision Transformer (ViT)[11] is a BERT-style take on transformers for vision, as illustrated in Figure 11-9. First the image is split into smaller patches, and each of these patches is embedded with a linear projection. The results strongly resemble the token embeddings in BERT, and what follows is virtually identical. The patch embeddings are combined with position embeddings and then fed through an ordinary transformer encoder. During pretraining some of the patches are masked or distorted, and the objective is to predict the average color of the masked patch.

11 A. Dosovitskiy et al., "An Image Is Worth 16x16 Words: Transformers for Image Recognition at Scale", (2020).

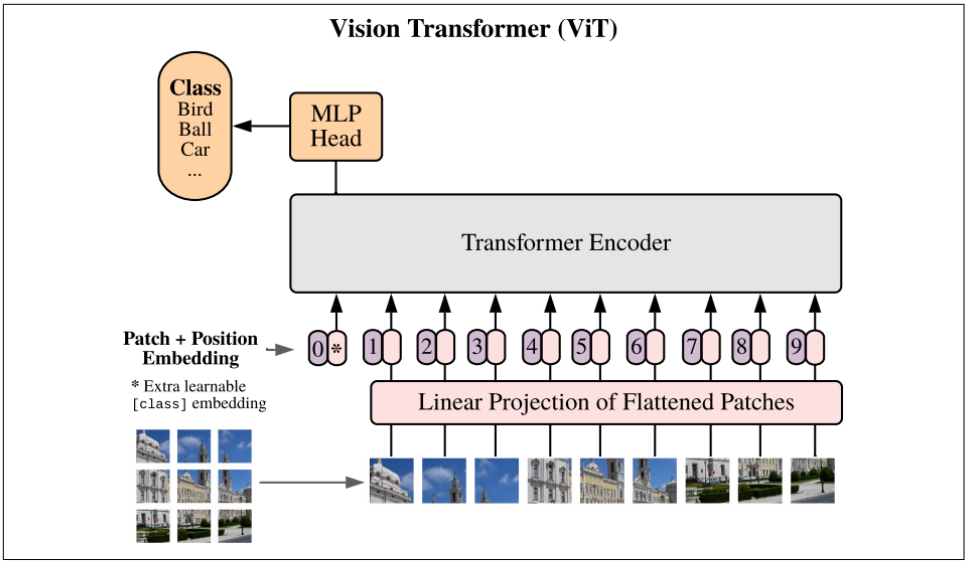
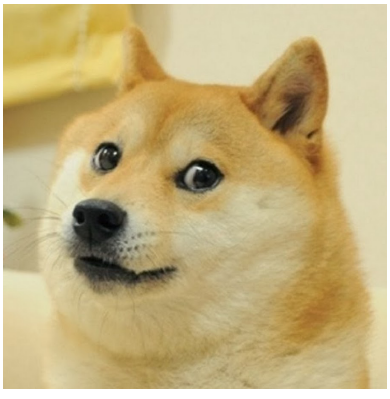


Figure 11-9. The ViT architecture (courtesy of Alexey Dosovitskiy et al.)

Although this approach did not produce better results when pretrained on the standard ImageNet dataset, it scaled significantly better than CNNs on larger datasets.

ViT is integrated in 🤗 Transformers, and using it is very similar to the NLP pipelines that we've used throughout this book. Let's start by loading the image of a rather famous dog:

```
from PIL import Image
import matplotlib.pyplot as plt

image = Image.open("images/doge.jpg")
plt.imshow(image)
plt.axis("off")
plt.show()
```



To load a ViT model, we just need to specify the `image-classification` pipeline, and then we feed in the image to extract the predicted classes:

```
import pandas as pd
from transformers import pipeline

image_classifier = pipeline("image-classification")
preds = image_classifier(image)
preds_df = pd.DataFrame(preds)
preds_df
```

	score	label
0	0.643599	Eskimo dog, husky
1	0.207407	Siberian husky
2	0.060160	dingo, warrigal, warragal, Canis dingo
3	0.035359	Norwegian elkhound, elkhound
4	0.012927	malamute, malemute, Alaskan malamute

Great, the predicted class seems to match the image!

A natural extension of image models is video models. In addition to the spatial dimensions, videos come with a temporal dimension. This makes the task more challenging as the volume of data gets much bigger and one needs to deal with the extra dimension. Models such as TimeSformer introduce a spatial and temporal attention mechanism to account for both.[12] In the future, such models can help build tools for a wide range of tasks such as classification or annotation of video sequences.

12 G. Bertasius, H. Wang, and L. Torresani, "Is Space-Time Attention All You Need for Video Understanding?", (2021).

Tables

A lot of data, such as customer data within a company, is stored in structured databases instead of as raw text. We saw in Chapter 7 that with question answering models we can query text with a question in natural text. Wouldn't it be nice if we could do the same with tables, as shown in Figure 11-10?

Table

Rank	Name	No. of reigns	Combined days
1	Lou Thesz	3	3,749
2	Ric Flair	8	3,103
3	Harley Race	7	1,799
4	Dory Funk Jr.	1	1,563
5	Dan Severn	2	1,559
6	Gene Kiniski	1	1,131

Example questions

#	Question	Answer	Example Type
1	*Which wrestler had the most number of reigns?*	Ric Flair	Cell selection
2	*Average time as champion for top 2 wrestlers?*	AVG(3749,3103)=3426	Scalar answer
3	*How many world champions are there with only one reign?*	COUNT(Dory Funk Jr., Gene Kiniski)=2	Ambiguous answer
4	*What is the number of reigns for* Harley Race?	7	Ambiguous answer
5	*Which of the following wrestlers were ranked in the bottom 3?*	{Dory Funk Jr., Dan Severn, Gene Kiniski}	Cell selection
	Out of these, who had more than one reign?	Dan Severn	Cell selection

Figure 11-10. Question answering over a table (courtesy of Jonathan Herzig)

TAPAS (short for Table Parser)[13] to the rescue! This model applies the Transformer architecture to tables by combining the tabular information with the query, as illustrated in Figure 11-11.

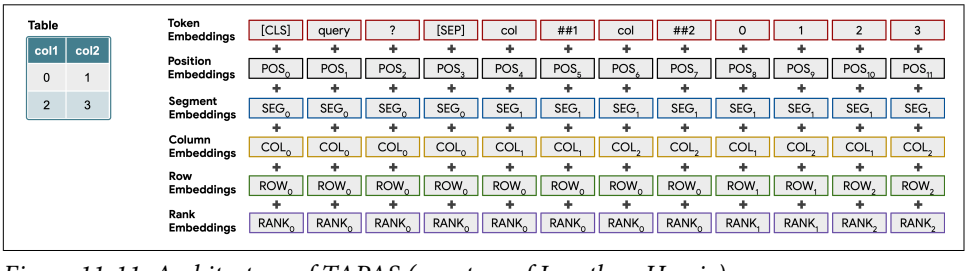


Figure 11-11. Architecture of TAPAS (courtesy of Jonathan Herzig)

Let's look at an example of how TAPAS works in practice. We have created a fictitious version of this book's table of contents. It contains the chapter number, the name of the chapter, as well as the starting and ending pages of the chapters:

```
book_data = [
    {"chapter": 0, "name": "Introduction", "start_page": 1, "end_page": 11},
    {"chapter": 1, "name": "Text classification", "start_page": 12,
     "end_page": 48},
    {"chapter": 2, "name": "Named Entity Recognition", "start_page": 49,
     "end_page": 73},
    {"chapter": 3, "name": "Question Answering", "start_page": 74,
```

13 J. Herzig et al., "TAPAS: Weakly Supervised Table Parsing via Pre-Training", (2020).

```
     "end_page": 120},
    {"chapter": 4, "name": "Summarization", "start_page": 121,
     "end_page": 140},
    {"chapter": 5, "name": "Conclusion", "start_page": 141,
     "end_page": 144}
]
```

We can also easily add the number of pages each chapter has with the existing fields. In order to play nicely with the TAPAS model, we need to make sure that all columns are of type `str`:

```
table = pd.DataFrame(book_data)
table['number_of_pages'] = table['end_page']-table['start_page']
table = table.astype(str)
table
```

	chapter	name	start_page	end_page	number_of_pages
0	0	Introduction	1	11	10
1	1	Text classification	12	48	36
2	2	Named Entity Recognition	49	73	24
3	3	Question Answering	74	120	46
4	4	Summarization	121	140	19
5	5	Conclusion	141	144	3

By now you should know the drill. We first load the `table-question-answering` pipeline:

```
table_qa = pipeline("table-question-answering")
```

and then pass some queries to extract the answers:

```
table_qa = pipeline("table-question-answering")
queries = ["What's the topic in chapter 4?",
           "What is the total number of pages?",
           "On which page does the chapter about question-answering start?",
           "How many chapters have more than 20 pages?"]
preds = table_qa(table, queries)
```

These predictions store the type of table operation in an `aggregator` field, along with the answer. Let's see how well TAPAS fared on our questions:

```
for query, pred in zip(queries, preds):
    print(query)
    if pred["aggregator"] == "NONE":
        print("Predicted answer: " + pred["answer"])
    else:
        print("Predicted answer: " + pred["answer"])
    print('='*50)
```

```
What's the topic in chapter 4?
Predicted answer: Summarization
==================================================
What is the total number of pages?
Predicted answer: SUM > 10, 36, 24, 46, 19, 3
==================================================
On which page does the chapter about question-answering start?
Predicted answer: AVERAGE > 74
==================================================
How many chapters have more than 20 pages?
Predicted answer: COUNT > 1, 2, 3
==================================================
```

For the first chapter, the model predicted exactly one cell with no aggregation. If we look at the table, we see that the answer is in fact correct. In the next example the model predicted all the cells containing the number of pages in combination with the sum aggregator, which again is the correct way of calculating the total number of pages. The answer to question three is also correct; the average aggregation is not necessary in that case, but it doesn't make a difference. Finally, we have a question that is a little bit more complex. To determine how many chapters have more than 20 pages we first need to find out which chapters satisfy that criterion and then count them. It seem that TAPAS again got it right and correctly determined that chapters 1, 2, and 3 have more than 20 pages, and added a count aggregator to the cells.

The kinds of questions we asked can also be solved with a few simple Pandas commands; however, the ability to ask questions in natural language instead of Python code allows a much wider audience to query the data to answer specific questions. Imagine such tools in the hands of business analysts or managers who are able verify their own hypotheses about the data!

Multimodal Transformers

So far we've looked at extending transformers to a single new modality. TAPAS is arguably multimodal since it combines text and tables, but the table is also treated as text. In this section we examine transformers that combine two modalities at once: audio plus text and vision plus text.

Speech-to-Text

Although being able to use text to interface with a computer is a huge step forward, using spoken language is an even more natural way for us to communicate. You can see this trend in industry, where applications such as Siri and Alexa are on the rise and becoming progressively more useful. Also, for a large fraction of the population, writing and reading are more challenging than speaking. So, being able to process and understand audio is not only convenient, but can help many people access more information. A common task in this domain is *automatic speech recognition* (ASR),

which converts spoken words to text and enables voice technologies like Siri to answer questions like "What is the weather like today?"

The wav2vec 2.0 family of models are one of the most recent developments in ASR: they use a transformer layer in combination with a CNN, as illustrated in Figure 11-12.[14] By leveraging unlabeled data during pretraining, these models achieve competitive results with only a few minutes of labeled data.

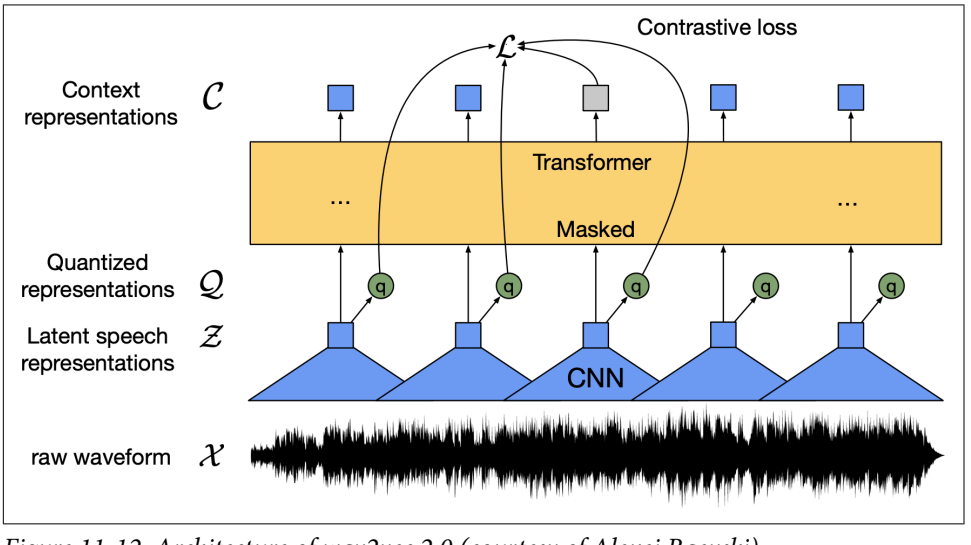


Figure 11-12. Architecture of wav2vec 2.0 (courtesy of Alexei Baevski)

The wav2vec 2.0 models are integrated in 🤗 Transformers, and you won't be surprised to learn that loading and using them follows the familiar steps that we have seen throughout this book. Let's load a pretrained model that was trained on 960 hours of speech audio:

```
asr = pipeline("automatic-speech-recognition")
```

To apply this model to some audio files we'll use the ASR subset of the SUPERB dataset, which is the same dataset the model was pretrained on. Since the dataset is quite large, we'll just load one example for our demo purposes:

```
from datasets import load_dataset

ds = load_dataset("superb", "asr", split="validation[:1]")
print(ds[0])
```

```
{'chapter_id': 128104, 'speaker_id': 1272, 'file': '~/.cache/huggingf
ace/datasets/downloads/extracted/e4e70a454363bec1c1a8ce336139866a39442114d86a433
```

14 A. Baevski et al., "wav2vec 2.0: A Framework for Self-Supervised Learning of Speech Representations", (2020).

```
6014acd4b1ed55e55/LibriSpeech/dev-clean/1272/128104/1272-128104-0000.flac',
'id': '1272-128104-0000', 'text': 'MISTER QUILTER IS THE APOSTLE OF THE MIDDLE
CLASSES AND WE ARE GLAD TO WELCOME HIS GOSPEL'}
```

Here we can see that the audio in the `file` column is stored in the FLAC coding format, while the expected transcription is given by the `text` column. To convert the audio to an array of floats, we can use the *SoundFile* library to read each file in our dataset with `map()`:

```
import soundfile as sf

def map_to_array(batch):
    speech, _ = sf.read(batch["file"])
    batch["speech"] = speech
    return batch

ds = ds.map(map_to_array)
```

If you are using a Jupyter notebook you can easily play the sound files with the following `IPython` widgets:

```
from IPython.display import Audio

display(Audio(ds[0]['speech'], rate=16000))
```

Finally, we can pass the inputs to the pipeline and inspect the prediction:

```
pred = asr(ds[0]["speech"])
print(pred)
```

```
{'text': 'MISTER QUILTER IS THE APOSTLE OF THE MIDDLE CLASSES AND WE ARE GLAD TO
WELCOME HIS GOSPEL'}
```

This transcription seems to be correct. We can see that some punctuation is missing, but this is hard to get from audio alone and could be added in a postprocessing step. With only a handful of lines of code we can build ourselves a state-of-the-art speech-to-text application!

Building a model for a new language still requires a minimum amount of labeled data, which can be challenging to obtain, especially for low-resource languages. Soon after the release of wav2vec 2.0, a paper describing a method named wav2vec-U was published.[15] In this work, a combination of clever clustering and GAN training is used to build a speech-to-text model using only independent unlabeled speech and unlabeled text data. This process is visualized in detail in Figure 11-13. No aligned speech and text data is required at all, which enables the training of highly performant speech-to-text models for a much larger spectrum of languages.

15 A. Baevski et al., "Unsupervised Speech Recognition", (2021).

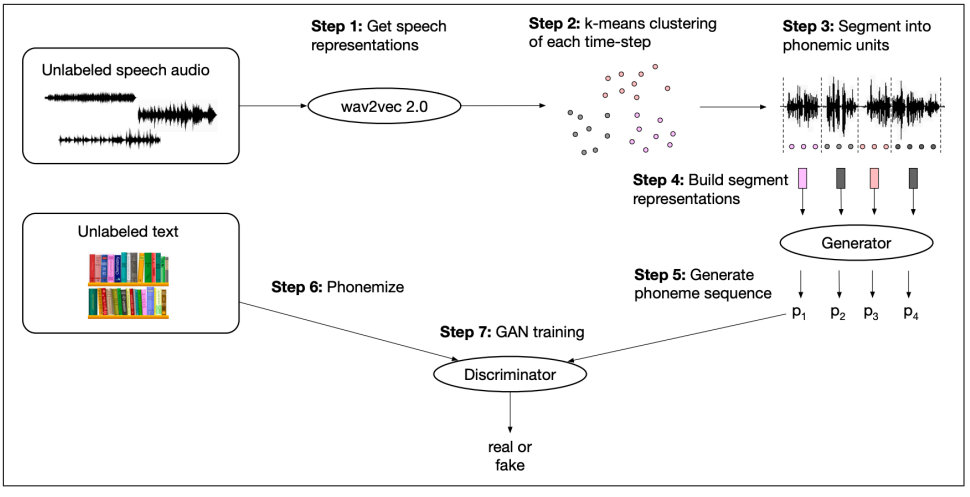


Figure 11-13. Training scheme for wav2vec-U (courtesy of Alexsei Baevski)

Great, so transformers can now "read" text and "hear" audio—can they also "see"? The answer is yes, and this is one of the current hot research frontiers in the field.

Vision and Text

Vision and text are another natural pair of modalities to combine since we frequently use language to communicate and reason about the contents of images and videos. In addition to the vision transformers, there have been several developments in the direction of combining visual and textual information. In this section we will look at four examples of models combining vision and text: VisualQA, LayoutLM, DALL·E, and CLIP.

VQA

In Chapter 7 we explored how we can use transformer models to extract answers to text-based questions. This can be done ad hoc to extract information from texts or offline, where the question answering model is used to extract structured information from a set of documents. There have been several efforts to expand this approach to vision with datasets such as VQA,[16] shown in Figure 11-14.

16 Y. Goyal et al., "Making the V in VQA Matter: Elevating the Role of Image Understanding in Visual Question Answering", (2016).

Figure 11-14. Example of a visual question answering task from the VQA dataset (courtesy of Yash Goyal)

Models such as LXMERT and VisualBERT use vision models like ResNets to extract features from the pictures and then use transformer encoders to combine them with the natural questions and predict an answer.[17]

LayoutLM

Analyzing scanned business documents like receipts, invoices, or reports is another area where extracting visual and layout information can be a useful way to recognize text fields of interest. Here the LayoutLM family of models are the current state of the art. They use an enhanced Transformer architecture that receives three modalities as input: text, image, and layout. Accordingly, as shown in Figure 11-15, there are embedding layers associated with each modality, a spatially aware self-attention mechanism, and a mix of image and text/image pretraining objectives to align the different modalities. By pretraining on millions of scanned documents, LayoutLM models are able to transfer to various downstream tasks in a manner similar to BERT for NLP.

17 H. Tan and M. Bansal, "LXMERT: Learning Cross-Modality Encoder Representations from Transformers", (2019); L.H. Li et al., "VisualBERT: A Simple and Performant Baseline for Vision and Language", (2019).

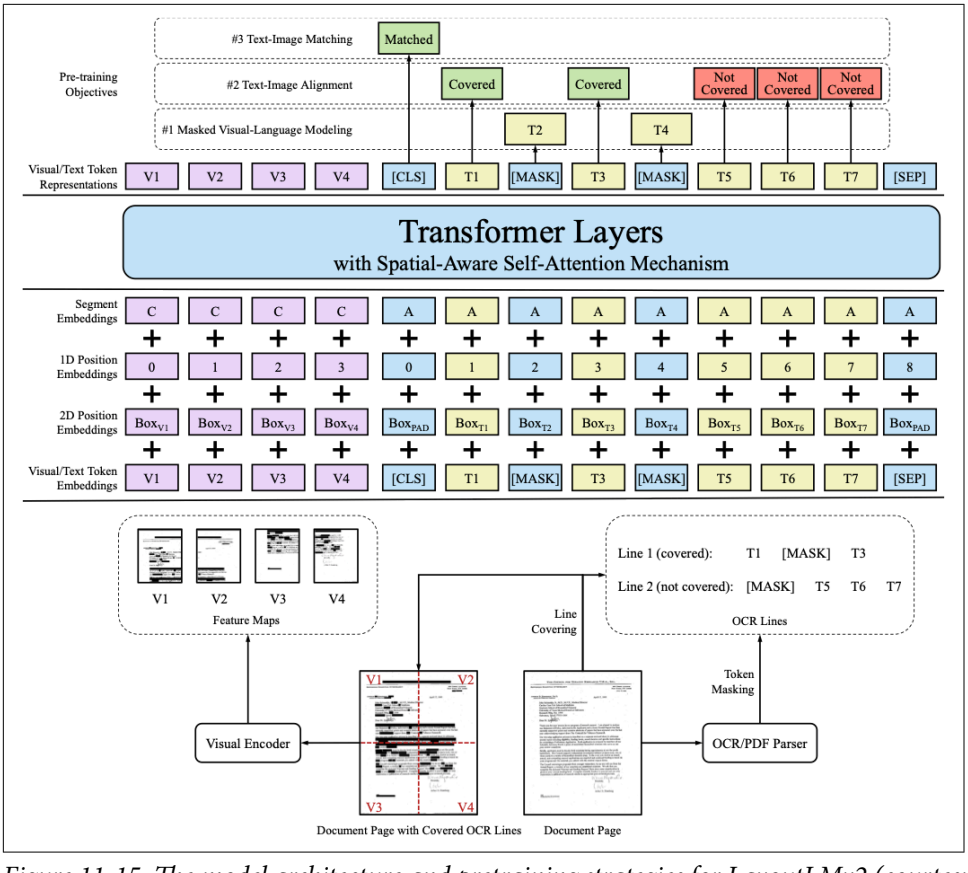


Figure 11-15. The model architecture and pretraining strategies for LayoutLMv2 (courtesy of Yang Xu)

DALL·E

A model that combines vision and text for *generative* tasks is DALL·E.[18] It uses the GPT architecture and autoregressive modeling to generate images from text. Inspired by iGPT, it regards the words and pixels as one sequence of tokens and is thus able to continue generating an image from a text prompt, as shown in Figure 11-16.

18 A. Ramesh et al., "Zero-Shot Text-to-Image Generation", (2021).

Figure 11-16. Generation examples with DALL·E (courtesy of Aditya Ramesh)

CLIP

Finally, let's have a look at CLIP,[19] which also combines text and vision but is designed for supervised tasks. Its creators constructed a dataset with 400 million image/caption pairs and used contrastive learning to pretrain the model. The CLIP architecture consists of a text and an image encoder (both transformers) that create embeddings of the captions and images. A batch of images with captions is sampled, and the contrastive objective is to maximize the similarity of the embeddings (as measured by the dot product) of the corresponding pair while minimizing the similarity of the rest, as illustrated in Figure 11-17.

In order to use the pretrained model for classification the possible classes are embedded with the text encoder, similar to how we used the zero-shot pipeline. Then the embeddings of all the classes are compared to the image embedding that we want to classify, and the class with the highest similarity is chosen.

19 A. Radford et al., "Learning Transferable Visual Models from Natural Language Supervision", (2021).

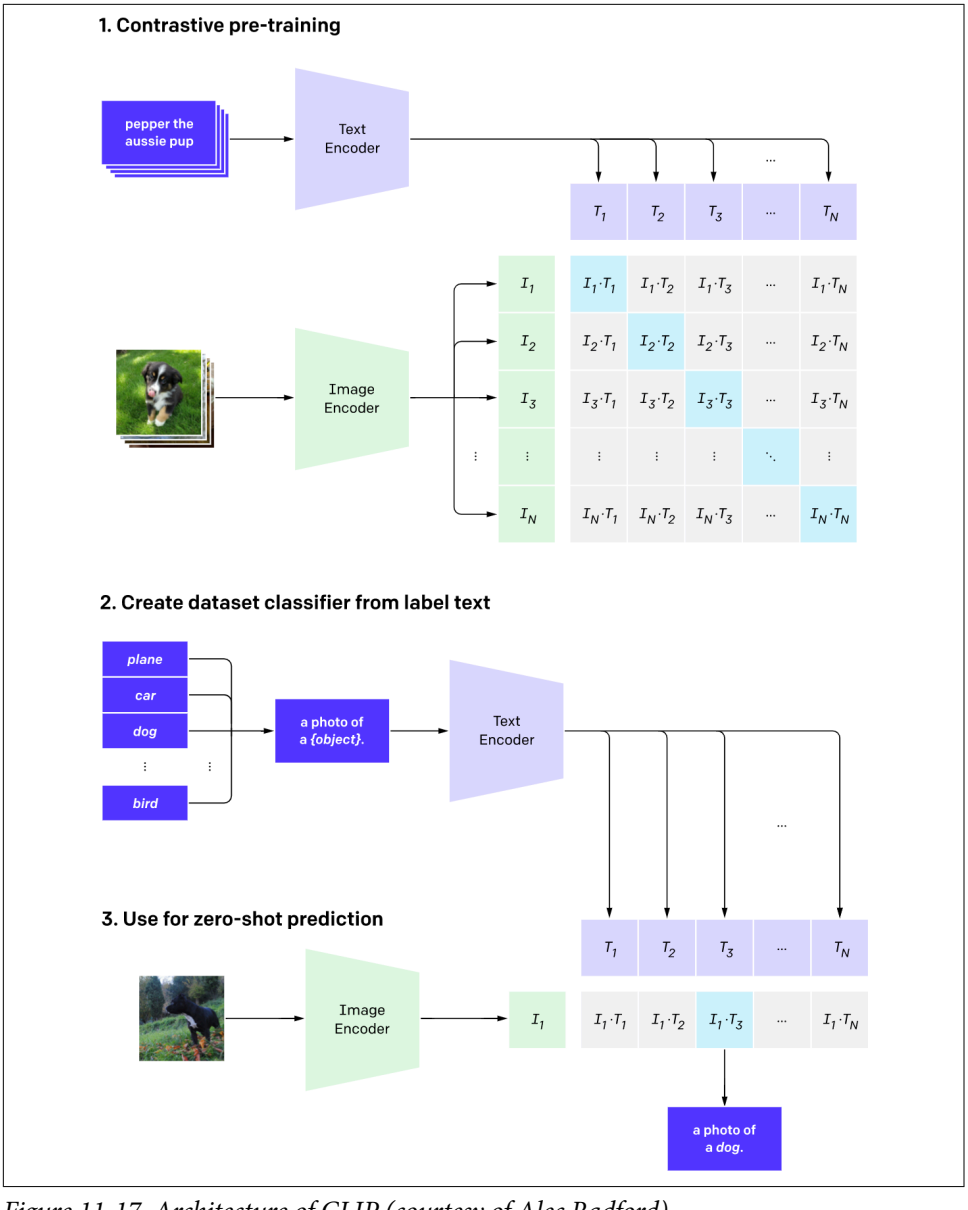


Figure 11-17. Architecture of CLIP (courtesy of Alec Radford)

The zero-shot image classification performance of CLIP is remarkable and competitive with fully supervised trained vision models, while being more flexible with regard to new classes. CLIP is also fully integrated in 🤗 Transformers, so we can try it out. For image-to-text tasks, we instantiate a *processor* that consists of a *feature extractor* and a tokenizer. The role of the feature extractor is to convert the image into a

form suitable for the model, while the tokenizer is responsible for decoding the model's predictions into text:

```
from transformers import CLIPProcessor, CLIPModel

clip_ckpt = "openai/clip-vit-base-patch32"
model = CLIPModel.from_pretrained(clip_ckpt)
processor = CLIPProcessor.from_pretrained(clip_ckpt)
```

Then we need a fitting image to try it out. What would be better suited than a picture of Optimus Prime?

```
image = Image.open("images/optimusprime.jpg")
plt.imshow(image)
plt.axis("off")
plt.show()
```

Next, we set up the texts to compare the image against and pass it through the model:

```
import torch

texts = ["a photo of a transformer", "a photo of a robot", "a photo of agi"]
inputs = processor(text=texts, images=image, return_tensors="pt", padding=True)
with torch.no_grad():
    outputs = model(**inputs)
logits_per_image = outputs.logits_per_image
probs = logits_per_image.softmax(dim=1)
probs
```

```
tensor([[0.9557, 0.0413, 0.0031]])
```

Well, it almost got the right answer (a photo of AGI of course). Jokes aside, CLIP makes image classification very flexible by allowing us to define classes through text instead of having the classes hardcoded in the model architecture. This concludes our tour of multimodal transformer models, but we hope we've whetted your appetite.

Where to from Here?

Well that's the end of the ride; thanks for joining us on this journey through the transformers landscape! Throughout this book we've explored how transformers can address a wide range of tasks and achieve state-of-the-art results. In this chapter we've seen how the current generation of models are being pushed to their limits with scaling and how they are also branching out into new domains and modalities.

If you want to reinforce the concepts and skills that you've learned in this book, here are a few ideas for where to go from here:

Join a Hugging Face community event
: Hugging Face hosts short sprints focused on improving the libraries in the ecosystem, and these events are a great way to meet the community and get a taste for open source software development. So far there have been sprints on adding 600+ datasets to 🤗 Datasets, fine-tuning 300+ ASR models in various languages, and implementing hundreds of projects in JAX/Flax.

Build your own project
: One very effective way to test your knowledge in machine learning is to build a project to solve a problem that interests you. You could reimplement a transformer paper, or apply transformers to a novel domain.

Contribute a model to 🤗 Transformers
: If you're looking for something more advanced, then contributing a newly published architecture to 🤗 Transformers is a great way to dive into the nuts and bolts of the library. There is a detailed guide to help you get started in the 🤗 Transformers documentation.

Blog about what you've learned
: Teaching others what you've learned is a powerful test of your own knowledge, and in a sense this was one of the driving motivations behind us writing this book! There are great tools to help you get started with technical blogging; we recommend *fastpages* as you can easily use Jupyter notebooks for everything.

About the Authors

Lewis Tunstall is a machine learning engineer at Hugging Face. He has built machine learning applications for startups and enterprises in the domains of NLP, topological data analysis, and time series. Lewis has a PhD in theoretical physics and has held research positions in Australia, the USA, and Switzerland. His current work focuses on developing tools for the NLP community and teaching people to use them effectively.

Leandro von Werra is a machine learning engineer in the open source team at Hugging Face. He has several years of industry experience bringing NLP projects to production by working across the whole machine learning stack, and is the creator of a popular Python library called TRL, which combines transformers with reinforcement learning.

Thomas Wolf is chief science officer at and cofounder of Hugging Face. His team is on a mission to catalyze and democratize NLP research. Prior to cofounding Hugging Face, Thomas earned a PhD in physics and later a law degree. He has worked as a physics researcher and a European patent attorney.

Colophon

The bird on the cover of *Natural Language Processing with Transformers* is a coconut lorikeet (*Trichoglossus haematodus*), a relative of parakeets and parrots. It is also known as the green-naped lorikeet and is native to Oceania.

The plumage of coconut lorikeets blends into their colorful tropical and subtropical surroundings; their green nape meets a yellow collar beneath a deep dark blue head, which ends in an orange-red bill. Their eyes are orange and the breast feathers are red. Coconut lorikeets have one of the longest, pointed tails of the seven species of lorikeet, which is green from above and yellow underneath. These birds measure 10 to 12 inches long and weigh 3.8 to 4.8 ounces.

Coconut lorikeets have one monogamous partner and lay two matte white eggs at a time. They build nests over 80 feet high in eucalyptus trees and live 15 to 20 years in the wild. This species suffers from habitat loss and capture for the pet trade. Many of the animals on O'Reilly's covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on a black and white engraving from *English Cyclopedia*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.