"me gusta" (24%), and "me encanta" (12%). Right now, the sentence "me gustan" is winning, but "me gusta" has not been eliminated.

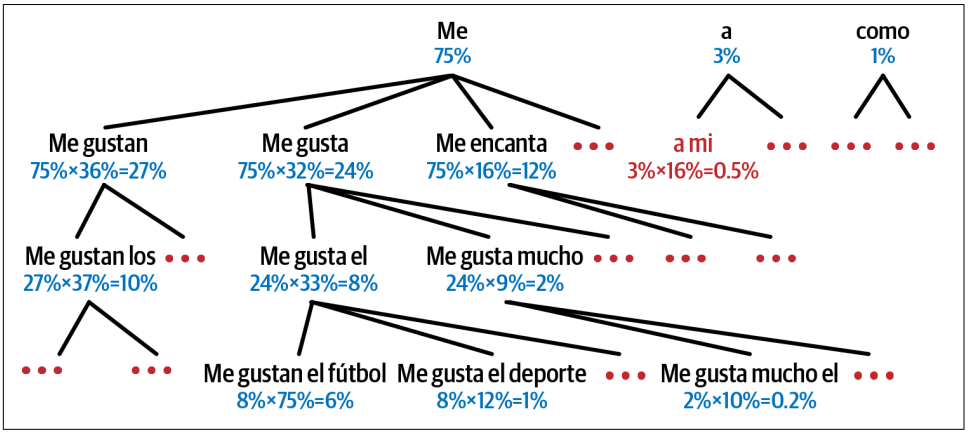

Figure 16-6. Beam search, with a beam width of 3

Then we repeat the same process: we use the model to predict the next word in each of these three sentences, and we compute the probabilities of all 3,000 three-word sentences we considered. Perhaps the top three are now "me gustan los" (10%), "me gusta el" (8%), and "me gusta mucho" (2%). At the next step we may get "me gusta el fútbol" (6%), "me gusta mucho el" (1%), and "me gusta el deporte" (0.2%). Notice that "me gustan" was eliminated, and the correct translation is now ahead. We boosted our encoder–decoder model's performance without any extra training, simply by using it more wisely.

The TensorFlow Addons library includes a full seq2seq API that lets you build encoder–decoder models with attention, including beam search, and more. However, its documentation is currently very limited. Implementing beam search is a good exercise, so give it a try! Check out this chapter's notebook for a possible solution.

With all this, you can get reasonably good translations for fairly short sentences. Unfortunately, this model will be really bad at translating long sentences. Once again, the problem comes from the limited short-term memory of RNNs. *Attention mechanisms* are the game-changing innovation that addressed this problem.

Attention Mechanisms

Consider the path from the word "soccer" to its translation "fútbol" back in Figure 16-3: it is quite long! This means that a representation of this word (along with all

the other words) needs to be carried over many steps before it is actually used. Can't we make this path shorter?

This was the core idea in a landmark 2014 paper[18] by Dzmitry Bahdanau et al., where the authors introduced a technique that allowed the decoder to focus on the appropriate words (as encoded by the encoder) at each time step. For example, at the time step where the decoder needs to output the word "fútbol", it will focus its attention on the word "soccer". This means that the path from an input word to its translation is now much shorter, so the short-term memory limitations of RNNs have much less impact. Attention mechanisms revolutionized neural machine translation (and deep learning in general), allowing a significant improvement in the state of the art, especially for long sentences (e.g., over 30 words).

The most common metric used in NMT is the *bilingual evaluation understudy* (BLEU) score, which compares each translation produced by the model with several good translations produced by humans: it counts the number of *n*-grams (sequences of *n* words) that appear in any of the target translations and adjusts the score to take into account the frequency of the produced *n*-grams in the target translations.

Figure 16-7 shows our encoder–decoder model with an added attention mechanism. On the left, you have the encoder and the decoder. Instead of just sending the encoder's final hidden state to the decoder, as well as the previous target word at each step (which is still done, although it is not shown in the figure), we now send all of the encoder's outputs to the decoder as well. Since the decoder cannot deal with all these encoder outputs at once, they need to be aggregated: at each time step, the decoder's memory cell computes a weighted sum of all the encoder outputs. This determines which words it will focus on at this step. The weight $\alpha_{(t,i)}$ is the weight of the i^{th} encoder output at the t^{th} decoder time step. For example, if the weight $\alpha_{(3,2)}$ is much larger than the weights $\alpha_{(3,0)}$ and $\alpha_{(3,1)}$, then the decoder will pay much more attention to the encoder's output for word #2 ("soccer") than to the other two outputs, at least at this time step. The rest of the decoder works just like earlier: at each time step the memory cell receives the inputs we just discussed, plus the hidden state from the previous time step, and finally (although it is not represented in the diagram) it receives the target word from the previous time step (or at inference time, the output from the previous time step).

18 Dzmitry Bahdanau et al., "Neural Machine Translation by Jointly Learning to Align and Translate", arXiv preprint arXiv:1409.0473 (2014).

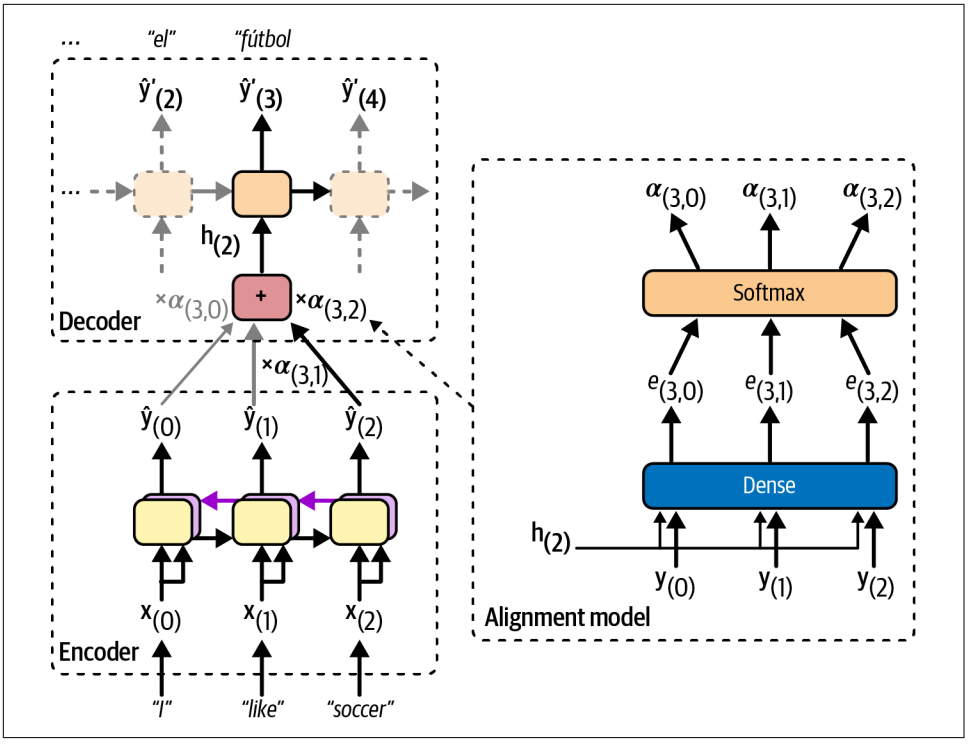

Figure 16-7. Neural machine translation using an encoder–decoder network with an attention model

But where do these $\alpha_{(t,i)}$ weights come from? Well, they are generated by a small neural network called an *alignment model* (or an *attention layer*), which is trained jointly with the rest of the encoder–decoder model. This alignment model is illustrated on the righthand side of Figure 16-7. It starts with a `Dense` layer composed of a single neuron that processes each of the encoder's outputs, along with the decoder's previous hidden state (e.g., $\mathbf{h}_{(2)}$). This layer outputs a score (or energy) for each encoder output (e.g., $e_{(3,\ 2)}$): this score measures how well each output is aligned with the decoder's previous hidden state. For example, in Figure 16-7, the model has already output "me gusta el" (meaning "I like"), so it's now expecting a noun: the word "soccer" is the one that best aligns with the current state, so it gets a high score. Finally, all the scores go through a softmax layer to get a final weight for each encoder output (e.g., $\alpha_{(3,2)}$). All the weights for a given decoder time step add up to 1. This particular attention mechanism is called *Bahdanau attention* (named after the 2014 paper's first author). Since it concatenates the encoder output with the decoder's previous hidden state, it is sometimes called *concatenative attention* (or *additive attention*).

If the input sentence is n words long, and assuming the output sentence is about as long, then this model will need to compute about n^2 weights. Fortunately, this quadratic computational complexity is still tractable because even long sentences don't have thousands of words.

Another common attention mechanism, known as *Luong attention* or *multiplicative attention*, was proposed shortly after, in 2015,[19] by Minh-Thang Luong et al. Because the goal of the alignment model is to measure the similarity between one of the encoder's outputs and the decoder's previous hidden state, the authors proposed to simply compute the dot product (see Chapter 4) of these two vectors, as this is often a fairly good similarity measure, and modern hardware can compute it very efficiently. For this to be possible, both vectors must have the same dimensionality. The dot product gives a score, and all the scores (at a given decoder time step) go through a softmax layer to give the final weights, just like in Bahdanau attention. Another simplification Luong et al. proposed was to use the decoder's hidden state at the current time step rather than at the previous time step (i.e., $\mathbf{h}_{(t)}$ rather than $\mathbf{h}_{(t-1)}$), then to use the output of the attention mechanism (noted $\widetilde{\mathbf{h}}_{(t)}$) directly to compute the decoder's predictions, rather than using it to compute the decoder's current hidden state. The researchers also proposed a variant of the dot product mechanism where the encoder outputs first go through a fully connected layer (without a bias term) before the dot products are computed. This is called the "general" dot product approach. The researchers compared both dot product approaches with the concatenative attention mechanism (adding a rescaling parameter vector $\mathbf{v}$), and they observed that the dot product variants performed better than concatenative attention. For this reason, concatenative attention is much less used now. The equations for these three attention mechanisms are summarized in Equation 16-1.

Equation 16-1. Attention mechanisms

$$\widetilde{\mathbf{h}}_{(t)} = \sum_i \alpha_{(t,i)} \mathbf{y}_{(i)}$$

$$\text{with } \alpha_{(t,i)} = \frac{\exp\left(e_{(t,i)}\right)}{\sum_{i'} \exp\left(e_{(t,i')}\right)}$$

$$\text{and } e_{(t,i)} = \begin{cases} \mathbf{h}_{(t)}^{\top} \mathbf{y}_{(i)} & dot \\ \mathbf{h}_{(t)}^{\top} \mathbf{W} \mathbf{y}_{(i)} & general \\ \mathbf{v}^{\top} \tanh\left(\mathbf{W}\left[\mathbf{h}_{(t)}; \mathbf{y}_{(i)}\right]\right) & concat \end{cases}$$

19 Minh-Thang Luong et al., "Effective Approaches to Attention-Based Neural Machine Translation", *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing* (2015): 1412–1421.

Keras provides a `tf.keras.layers.Attention` layer for Luong attention, and an `AdditiveAttention` layer for Bahdanau attention. Let's add Luong attention to our encoder–decoder model. Since we will need to pass all the encoder's outputs to the `Attention` layer, we first need to set `return_sequences=True` when creating the encoder:

```
encoder = tf.keras.layers.Bidirectional(
    tf.keras.layers.LSTM(256, return_sequences=True, return_state=True))
```

Next, we need to create the attention layer and pass it the decoder's states and the encoder's outputs. However, to access the decoder's states at each step we would need to write a custom memory cell. For simplicity, let's use the decoder's outputs instead of its states: in practice this works well too, and it's much easier to code. Then we just pass the attention layer's outputs directly to the output layer, as suggested in the Luong attention paper:

```
attention_layer = tf.keras.layers.Attention()
attention_outputs = attention_layer([decoder_outputs, encoder_outputs])
output_layer = tf.keras.layers.Dense(vocab_size, activation="softmax")
Y_proba = output_layer(attention_outputs)
```

And that's it! If you train this model, you will find that it now handles much longer sentences. For example:

```
>>> translate("I like soccer and also going to the beach")
'me gusta el fútbol y también ir a la playa'
```

In short, the attention layer provides a way to focus the attention of the model on part of the inputs. But there's another way to think of this layer: it acts as a differentiable memory retrieval mechanism.

For example, let's suppose the encoder analyzed the input sentence "I like soccer", and it managed to understand that the word "I" is the subject and the word "like" is the verb, so it encoded this information in its outputs for these words. Now suppose the decoder has already translated the subject, and it thinks that it should translate the verb next. For this, it needs to fetch the verb from the input sentence. This is analogous to a dictionary lookup: it's as if the encoder had created a dictionary {"subject": "They", "verb": "played", …} and the decoder wanted to look up the value that corresponds to the key "verb".

However, the model does not have discrete tokens to represent the keys (like "subject" or "verb"); instead, it has vectorized representations of these concepts that it learned during training, so the query it will use for the lookup will not perfectly match any key in the dictionary. The solution is to compute a similarity measure between the query and each key in the dictionary, and then use the softmax function to convert these similarity scores to weights that add up to 1. As we saw earlier, that's exactly what the attention layer does. If the key that represents the verb is by far the most similar to the query, then that key's weight will be close to 1.

Next, the attention layer computes a weighted sum of the corresponding values: if the weight of the "verb" key is close to 1, then the weighted sum will be very close to the representation of the word "played".

This is why the Keras `Attention` and `AdditiveAttention` layers both expect a list as input, containing two or three items: the *queries*, the *keys*, and optionally the *values*. If you do not pass any values, then they are automatically equal to the keys. So, looking at the previous code example again, the decoder outputs are the queries, and the encoder outputs are both the keys and the values. For each decoder output (i.e., each query), the attention layer returns a weighted sum of the encoder outputs (i.e., the keys/values) that are most similar to the decoder output.

The bottom line is that an attention mechanism is a trainable memory retrieval system. It is so powerful that you can actually build state-of-the-art models using only attention mechanisms. Enter the transformer architecture.

Attention Is All You Need: The Original Transformer Architecture

In a groundbreaking 2017 paper,[20] a team of Google researchers suggested that "Attention Is All You Need". They created an architecture called the *transformer*, which significantly improved the state-of-the-art in NMT without using any recurrent or convolutional layers,[21] just attention mechanisms (plus embedding layers, dense layers, normalization layers, and a few other bits and pieces). Because the model is not recurrent, it doesn't suffer as much from the vanishing or exploding gradients problems as RNNs, it can be trained in fewer steps, it's easier to parallelize across multiple GPUs, and it can better capture long-range patterns than RNNs. The original 2017 transformer architecture is represented in Figure 16-8.

In short, the left part of Figure 16-8 is the encoder, and the right part is the decoder. Each embedding layer outputs a 3D tensor of shape [*batch size*, *sequence length*, *embedding size*]. After that, the tensors are gradually transformed as they flow through the transformer, but their shape remains the same.

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

21 Since the transformer uses time-distributed dense layers, you could argue that it uses 1D convolutional layers with a kernel size of 1.

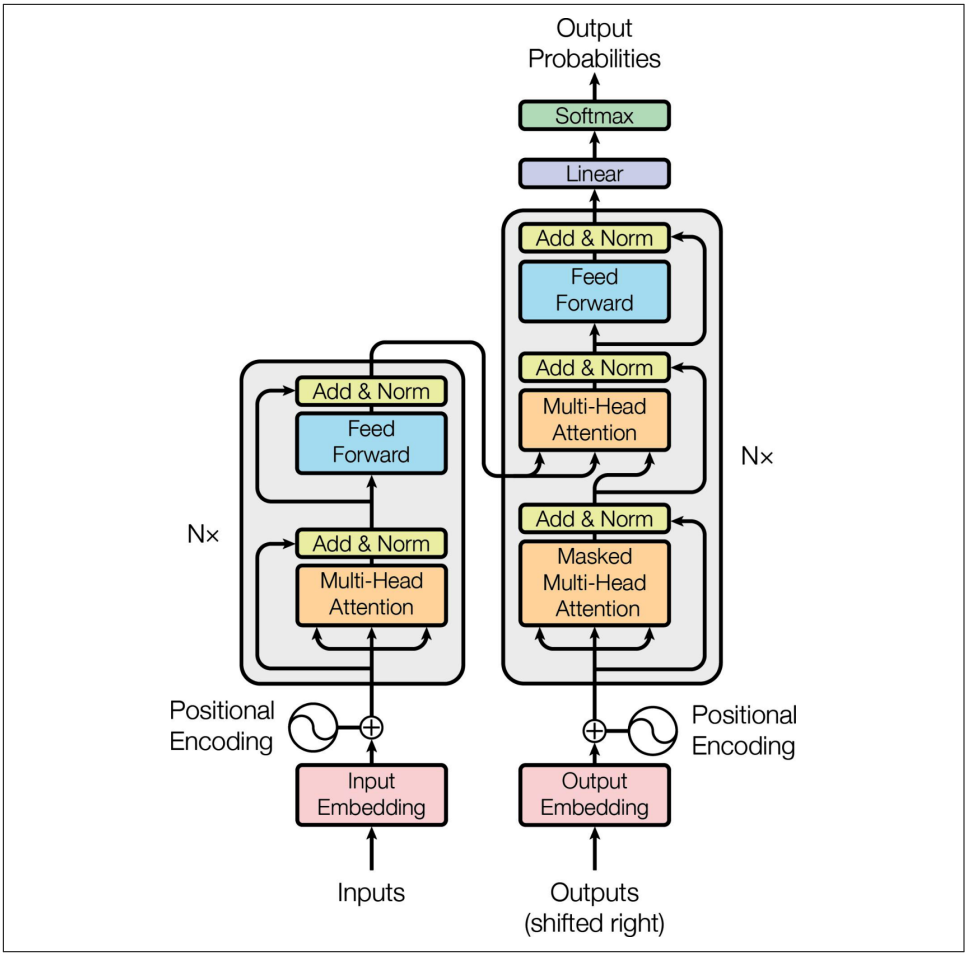

Figure 16-8. The original 2017 transformer architecture[22]

If you use the transformer for NMT, then during training you must feed the English sentences to the encoder and the corresponding Spanish translations to the decoder, with an extra SOS token inserted at the start of each sentence. At inference time, you must call the transformer multiple times, producing the translations one word at a time and feeding the partial translations to the decoder at each round, just like we did earlier in the `translate()` function.

22 This is figure 1 from the "Attention Is All You Need" paper, reproduced with the kind permission of the authors.

The encoder's role is to gradually transform the inputs—word representations of the English sentence—until each word's representation perfectly captures the meaning of the word, in the context of the sentence. For example, if you feed the encoder with the sentence "I like soccer", then the word "like" will start off with a rather vague representation, since this word could mean different things in different contexts: think of "I like soccer" versus "It's like that". But after going through the encoder, the word's representation should capture the correct meaning of "like" in the given sentence (i.e., to be fond of), as well as any other information that may be required for translation (e.g., it's a verb).

The decoder's role is to gradually transform each word representation in the translated sentence into a word representation of the next word in the translation. For example, if the sentence to translate is "I like soccer", and the decoder's input sentence is "<SOS> me gusta el fútbol", then after going through the decoder, the word representation of the word "el" will end up transformed into a representation of the word "fútbol". Similarly, the representation of the word "fútbol" will be transformed into a representation of the EOS token.

After going through the decoder, each word representation goes through a final `Dense` layer with a softmax activation function, which will hopefully output a high probability for the correct next word and a low probability for all other words. The predicted sentence should be "me gusta el fútbol <EOS>".

That was the big picture; now let's walk through Figure 16-8 in more detail:

- First, notice that both the encoder and the decoder contain modules that are stacked N times. In the paper, $N = 6$. The final outputs of the whole encoder stack are fed to the decoder at each of these N levels.
- Zooming in, you can see that you are already familiar with most components: there are two embedding layers; several skip connections, each of them followed by a layer normalization layer; several feedforward modules that are composed of two dense layers each (the first one using the ReLU activation function, the second with no activation function); and finally the output layer is a dense layer using the softmax activation function. You can also sprinkle a bit of dropout after the attention layers and the feedforward modules, if needed. Since all of these layers are time-distributed, each word is treated independently from all the others. But how can we translate a sentence by looking at the words completely separately? Well, we can't, so that's where the new components come in:
  - The encoder's *multi-head attention* layer updates each word representation by attending to (i.e., paying attention to) all other words in the same sentence. That's where the vague representation of the word "like" becomes a richer and more accurate representation, capturing its precise meaning in the given sentence. We will discuss exactly how this works shortly.

— The decoder's *masked multi-head attention* layer does the same thing, but when it processes a word, it doesn't attend to words located after it: it's a causal layer. For example, when it processes the word "gusta", it only attends to the words "<SOS> me gusta", and it ignores the words "el fútbol" (or else that would be cheating).

— The decoder's upper *multi-head attention* layer is where the decoder pays attention to the words in the English sentence. This is called *cross*-attention, not *self*-attention in this case. For example, the decoder will probably pay close attention to the word "soccer" when it processes the word "el" and transforms its representation into a representation of the word "fútbol".

— The *positional encodings* are dense vectors (much like word embeddings) that represent the position of each word in the sentence. The n^{th} positional encoding is added to the word embedding of the n^{th} word in each sentence. This is needed because all layers in the transformer architecture ignore word positions: without positional encodings, you could shuffle the input sequences, and it would just shuffle the output sequences in the same way. Obviously, the order of words matters, which is why we need to give positional information to the transformer somehow: adding positional encodings to the word representations is a good way to achieve this.

The first two arrows going into each multi-head attention layer in Figure 16-8 represent the keys and values, and the third arrow represents the queries. In the self-attention layers, all three are equal to the word representations output by the previous layer, while in the decoder's upper attention layer, the keys and values are equal to the encoder's final word representations, and the queries are equal to the word representations output by the previous layer.

Let's go through the novel components of the transformer architecture in more detail, starting with the positional encodings.

Positional encodings

A positional encoding is a dense vector that encodes the position of a word within a sentence: the i^{th} positional encoding is added to the word embedding of the i^{th} word in the sentence. The easiest way to implement this is to use an `Embedding` layer and make it encode all the positions from 0 to the maximum sequence length in the batch, then add the result to the word embeddings. The rules of broadcasting will ensure that the positional encodings get applied to every input sequence. For example, here is how to add positional encodings to the encoder and decoder inputs:

```
max_length = 50  # max length in the whole training set
embed_size = 128
pos_embed_layer = tf.keras.layers.Embedding(max_length, embed_size)
batch_max_len_enc = tf.shape(encoder_embeddings)[1]
encoder_in = encoder_embeddings + pos_embed_layer(tf.range(batch_max_len_enc))
batch_max_len_dec = tf.shape(decoder_embeddings)[1]
decoder_in = decoder_embeddings + pos_embed_layer(tf.range(batch_max_len_dec))
```

Note that this implementation assumes that the embeddings are represented as regular tensors, not ragged tensors.[23] The encoder and the decoder share the same `Embedding` layer for the positional encodings, since they have the same embedding size (this is often the case).

Instead of using trainable positional encodings, the authors of the transformer paper chose to use fixed positional encodings, based on the sine and cosine functions at different frequencies. The positional encoding matrix **P** is defined in Equation 16-2 and represented at the top of Figure 16-9 (transposed), where $P_{p,i}$ is the i^{th} component of the encoding for the word located at the p^{th} position in the sentence.

Equation 16-2. Sine/cosine positional encodings

$$P_{p,i} = \begin{cases} \sin\left(p/10000^{i/d}\right) & \text{if } i \text{ is even} \\ \cos\left(p/10000^{(i-1)/d}\right) & \text{if } i \text{ is odd} \end{cases}$$

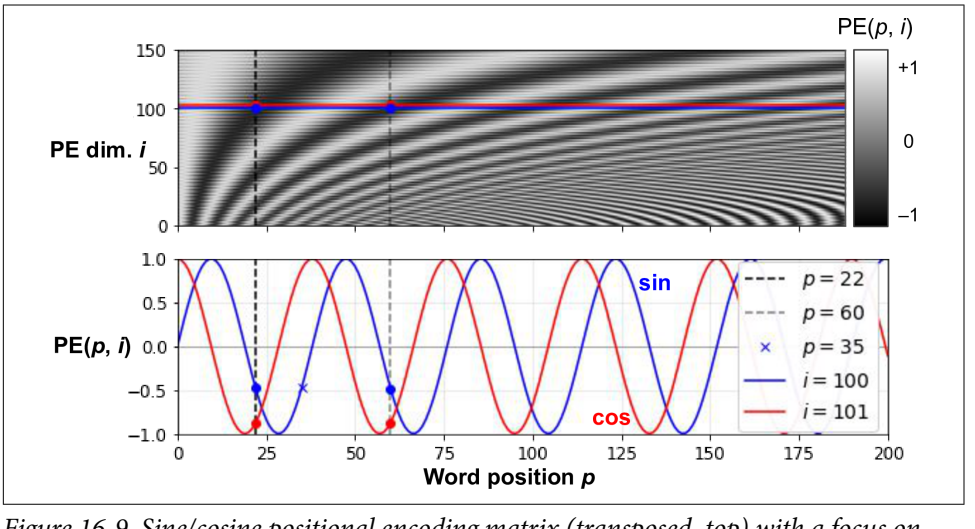

Figure 16-9. Sine/cosine positional encoding matrix (transposed, top) with a focus on two values of i (bottom)

23 It's possible to use ragged tensors instead, if you are using the latest version of TensorFlow.

This solution can give the same performance as trainable positional encodings, and it can extend to arbitrarily long sentences without adding any parameters to the model (however, when there is a large amount of pretraining data, trainable positional encodings are usually favored). After these positional encodings are added to the word embeddings, the rest of the model has access to the absolute position of each word in the sentence because there is a unique positional encoding for each position (e.g., the positional encoding for the word located at the 22nd position in a sentence is represented by the vertical dashed line at the top left of Figure 16-9, and you can see that it is unique to that position). Moreover, the choice of oscillating functions (sine and cosine) makes it possible for the model to learn relative positions as well. For example, words located 38 words apart (e.g., at positions $p = 22$ and $p = 60$) always have the same positional encoding values in the encoding dimensions $i = 100$ and $i = 101$, as you can see in Figure 16-9. This explains why we need both the sine and the cosine for each frequency: if we only used the sine (the blue wave at $i = 100$), the model would not be able to distinguish positions $p = 22$ and $p = 35$ (marked by a cross).

There is no `PositionalEncoding` layer in TensorFlow, but it is not too hard to create one. For efficiency reasons, we precompute the positional encoding matrix in the constructor. The `call()` method just truncates this encoding matrix to the max length of the input sequences, and it adds them to the inputs. We also set `supports_masking=True` to propagate the input's automatic mask to the next layer:

```
class PositionalEncoding(tf.keras.layers.Layer):
    def __init__(self, max_length, embed_size, dtype=tf.float32, **kwargs):
        super().__init__(dtype=dtype, **kwargs)
        assert embed_size % 2 == 0, "embed_size must be even"
        p, i = np.meshgrid(np.arange(max_length),
                           2 * np.arange(embed_size // 2))
        pos_emb = np.empty((1, max_length, embed_size))
        pos_emb[0, :, ::2] = np.sin(p / 10_000 ** (i / embed_size)).T
        pos_emb[0, :, 1::2] = np.cos(p / 10_000 ** (i / embed_size)).T
        self.pos_encodings = tf.constant(pos_emb.astype(self.dtype))
        self.supports_masking = True

    def call(self, inputs):
        batch_max_length = tf.shape(inputs)[1]
        return inputs + self.pos_encodings[:, :batch_max_length]
```

Let's use this layer to add the positional encoding to the encoder's inputs:

```
pos_embed_layer = PositionalEncoding(max_length, embed_size)
encoder_in = pos_embed_layer(encoder_embeddings)
decoder_in = pos_embed_layer(decoder_embeddings)
```

Now let's look deeper into the heart of the transformer model, at the multi-head attention layer.

Multi-head attention

To understand how a multi-head attention layer works, we must first understand the *scaled dot-product attention* layer, which it is based on. Its equation is shown in Equation 16-3, in a vectorized form. It's the same as Luong attention, except for a scaling factor.

Equation 16-3. Scaled dot-product attention

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

In this equation:

- $\mathbf{Q}$ is a matrix containing one row per *query*. Its shape is $[n_{\text{queries}}, d_{\text{keys}}]$, where n_{queries} is the number of queries and d_{keys} is the number of dimensions of each query and each key.
- $\mathbf{K}$ is a matrix containing one row per *key*. Its shape is $[n_{\text{keys}}, d_{\text{keys}}]$, where n_{keys} is the number of keys and values.
- $\mathbf{V}$ is a matrix containing one row per *value*. Its shape is $[n_{\text{keys}}, d_{\text{values}}]$, where d_{values} is the number of dimensions of each value.
- The shape of $\mathbf{Q}\ \mathbf{K}^\intercal$ is $[n_{\text{queries}}, n_{\text{keys}}]$: it contains one similarity score for each query/key pair. To prevent this matrix from being huge, the input sequences must not be too long (we will discuss how to overcome this limitation later in this chapter). The output of the softmax function has the same shape, but all rows sum up to 1. The final output has a shape of $[n_{\text{queries}}, d_{\text{values}}]$: there is one row per query, where each row represents the query result (a weighted sum of the values).
- The scaling factor $1 / (\sqrt{d_{\text{keys}}})$ scales down the similarity scores to avoid saturating the softmax function, which would lead to tiny gradients.
- It is possible to mask out some key/value pairs by adding a very large negative value to the corresponding similarity scores, just before computing the softmax. This is useful in the masked multi-head attention layer.

If you set `use_scale=True` when creating a `tf.keras.layers.Attention` layer, then it will create an additional parameter that lets the layer learn how to properly downscale the similarity scores. The scaled dot-product attention used in the transformer model is almost the same, except it always scales the similarity scores by the same factor, $1 / (\sqrt{d_{\text{keys}}})$.

Note that the `Attention` layer's inputs are just like $\mathbf{Q}$, $\mathbf{K}$, and $\mathbf{V}$, except with an extra batch dimension (the first dimension). Internally, the layer computes all the attention scores for all sentences in the batch with just one call to `tf.matmul(queries, keys)`:

this makes it extremely efficient. Indeed, in TensorFlow, if `A` and `B` are tensors with more than two dimensions—say, of shape [2, 3, 4, 5] and [2, 3, 5, 6], respectively—then `tf.matmul(A, B)` will treat these tensors as 2 × 3 arrays where each cell contains a matrix, and it will multiply the corresponding matrices: the matrix at the i^{th} row and j^{th} column in `A` will be multiplied by the matrix at the i^{th} row and j^{th} column in `B`. Since the product of a 4 × 5 matrix with a 5 × 6 matrix is a 4 × 6 matrix, `tf.matmul(A, B)` will return an array of shape [2, 3, 4, 6].

Now we're ready to look at the multi-head attention layer. Its architecture is shown in Figure 16-10.

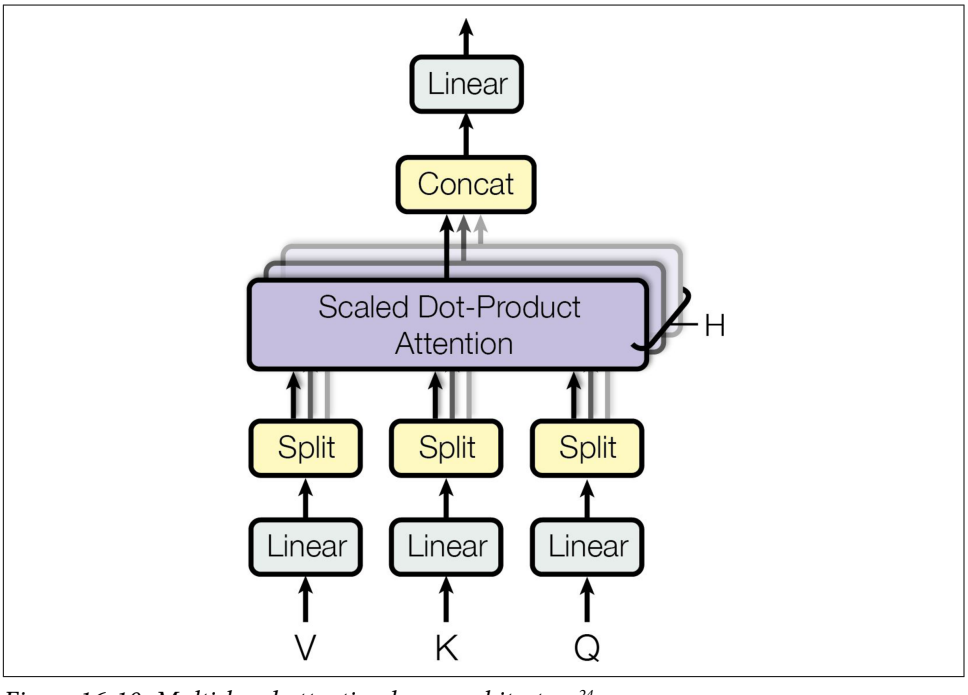

Figure 16-10. Multi-head attention layer architecture[24]

As you can see, it is just a bunch of scaled dot-product attention layers, each preceded by a linear transformation of the values, keys, and queries (i.e., a time-distributed dense layer with no activation function). All the outputs are simply concatenated, and they go through a final linear transformation (again, time-distributed).

But why? What is the intuition behind this architecture? Well, consider once again the word "like" in the sentence "I like soccer". The encoder was smart enough to

24 This is the righthand part of figure 2 from "Attention Is All You Need", reproduced with the kind authorization of the authors.

encode the fact that it is a verb. But the word representation also includes its position in the text, thanks to the positional encodings, and it probably includes many other features that are useful for its translation, such as the fact that it is in the present tense. In short, the word representation encodes many different characteristics of the word. If we just used a single scaled dot-product attention layer, we would only be able to query all of these characteristics in one shot.

This is why the multi-head attention layer applies *multiple* different linear transformations of the values, keys, and queries: this allows the model to apply many different projections of the word representation into different subspaces, each focusing on a subset of the word's characteristics. Perhaps one of the linear layers will project the word representation into a subspace where all that remains is the information that the word is a verb, another linear layer will extract just the fact that it is present tense, and so on. Then the scaled dot-product attention layers implement the lookup phase, and finally we concatenate all the results and project them back to the original space.

Keras includes a `tf.keras.layers.MultiHeadAttention` layer, so we now have everything we need to build the rest of the transformer. Let's start with the full encoder, which is exactly like in Figure 16-8, except we use a stack of two blocks (`N = 2`) instead of six, since we don't have a huge training set, and we add a bit of dropout as well:

```
N = 2  # instead of 6
num_heads = 8
dropout_rate = 0.1
n_units = 128  # for the first dense layer in each feedforward block
encoder_pad_mask = tf.math.not_equal(encoder_input_ids, 0)[:, tf.newaxis]
Z = encoder_in
for _ in range(N):
    skip = Z
    attn_layer = tf.keras.layers.MultiHeadAttention(
        num_heads=num_heads, key_dim=embed_size, dropout=dropout_rate)
    Z = attn_layer(Z, value=Z, attention_mask=encoder_pad_mask)
    Z = tf.keras.layers.LayerNormalization()(tf.keras.layers.Add()([Z, skip]))
    skip = Z
    Z = tf.keras.layers.Dense(n_units, activation="relu")(Z)
    Z = tf.keras.layers.Dense(embed_size)(Z)
    Z = tf.keras.layers.Dropout(dropout_rate)(Z)
    Z = tf.keras.layers.LayerNormalization()(tf.keras.layers.Add()([Z, skip]))
```

This code should be mostly straightforward, except for one thing: masking. As of the time of writing, the `MultiHeadAttention` layer does not support automatic masking,[25] so we must handle it manually. How can we do that?

25 This will most likely change by the time you read this; check out Keras issue #16248 for more details. When this happens, there will be no need to set the `attention_mask` argument, and therefore no need to create `encoder_pad_mask`.

The `MultiHeadAttention` layer accepts an `attention_mask` argument, which is a Boolean tensor of shape [*batch size*, *max query length*, *max value length*]: for every token in every query sequence, this mask indicates which tokens in the corresponding value sequence should be attended to. We want to tell the `MultiHeadAttention` layer to ignore all the padding tokens in the values. So, we first compute the padding mask using `tf.math.not_equal(encoder_input_ids, 0)`. This returns a Boolean tensor of shape [*batch size*, *max sequence length*]. We then insert a second axis using `[:, tf.newaxis]`, to get a mask of shape [*batch size*, 1, *max sequence length*]. This allows us to use this mask as the `attention_mask` when calling the `MultiHeadAttention` layer: thanks to broadcasting, the same mask will be used for all tokens in each query. This way, the padding tokens in the values will be ignored correctly.

However, the layer will compute outputs for every single query token, including the padding tokens. We need to mask the outputs that correspond to these padding tokens. Recall that we used `mask_zero` in the `Embedding` layers, and we set `supports_masking` to `True` in the `PositionalEncoding` layer, so the automatic mask was propagated all the way to the `MultiHeadAttention` layer's inputs (`encoder_in`). We can use this to our advantage in the skip connection: indeed, the `Add` layer supports automatic masking, so when we add `Z` and `skip` (which is initially equal to `encoder_in`), the outputs get automatically masked correctly.[26] Yikes! Masking required much more explanation than code.

Now on to the decoder! Once again, masking is going to be the only tricky part, so let's start with that. The first multi-head attention layer is a self-attention layer, like in the encoder, but it is a *masked* multi-head attention layer, meaning it is causal: it should ignore all tokens in the future. So, we need two masks: a padding mask and a causal mask. Let's create them:

```
decoder_pad_mask = tf.math.not_equal(decoder_input_ids, 0)[:, tf.newaxis]
causal_mask = tf.linalg.band_part(  # creates a lower triangular matrix
    tf.ones((batch_max_len_dec, batch_max_len_dec), tf.bool), -1, 0)
```

The padding mask is exactly like the one we created for the encoder, except it's based on the decoder's inputs rather than the encoder's. The causal mask is created using the `tf.linalg.band_part()` function, which takes a tensor and returns a copy with all the values outside a diagonal band set to zero. With these arguments, we get a square matrix of size `batch_max_len_dec` (the max length of the input sequences in the batch), with 1s in the lower-left triangle and 0s in the upper right. If we use this mask as the attention mask, we will get exactly what we want: the first query token will only attend to the first value token, the second will only attend to the first two,

26 Currently `Z + skip` does not support automatic masking, which is why we had to write `tf.keras.layers.Add()([Z, skip])` instead. Again, this may change by the time you read this.

the third will only attend to the first three, and so on. In other words, query tokens cannot attend to any value token in the future.

Let's now build the decoder:

```
encoder_outputs = Z  # let's save the encoder's final outputs
Z = decoder_in  # the decoder starts with its own inputs
for _ in range(N):
    skip = Z
    attn_layer = tf.keras.layers.MultiHeadAttention(
        num_heads=num_heads, key_dim=embed_size, dropout=dropout_rate)
    Z = attn_layer(Z, value=Z, attention_mask=causal_mask & decoder_pad_mask)
    Z = tf.keras.layers.LayerNormalization()(tf.keras.layers.Add()([Z, skip]))
    skip = Z
    attn_layer = tf.keras.layers.MultiHeadAttention(
        num_heads=num_heads, key_dim=embed_size, dropout=dropout_rate)
    Z = attn_layer(Z, value=encoder_outputs, attention_mask=encoder_pad_mask)
    Z = tf.keras.layers.LayerNormalization()(tf.keras.layers.Add()([Z, skip]))
    skip = Z
    Z = tf.keras.layers.Dense(n_units, activation="relu")(Z)
    Z = tf.keras.layers.Dense(embed_size)(Z)
    Z = tf.keras.layers.LayerNormalization()(tf.keras.layers.Add()([Z, skip]))
```

For the first attention layer, we use `causal_mask & decoder_pad_mask` to mask both the padding tokens and future tokens. The causal mask only has two dimensions: it's missing the batch dimension, but that's okay since broadcasting ensures that it gets copied across all the instances in the batch.

For the second attention layer, there's nothing special. The only thing to note is that we are using `encoder_pad_mask`, not `decoder_pad_mask`, because this attention layer uses the encoder's final outputs as its values.

We're almost done. We just need to add the final output layer, create the model, compile it, and train it:

```
Y_proba = tf.keras.layers.Dense(vocab_size, activation="softmax")(Z)
model = tf.keras.Model(inputs=[encoder_inputs, decoder_inputs],
                       outputs=[Y_proba])
model.compile(loss="sparse_categorical_crossentropy", optimizer="nadam",
              metrics=["accuracy"])
model.fit((X_train, X_train_dec), Y_train, epochs=10,
          validation_data=((X_valid, X_valid_dec), Y_valid))
```

Congratulations! You've built a full transformer from scratch, and trained it for automatic translation. This is getting quite advanced!

The Keras team has created a new Keras NLP project, including an API to build a transformer more easily. You may also be interested in the new Keras CV project for computer vision.