State-of-the-Art in Deep Learning: Transformers

So far, we have explored several major deep learning architectures, including fully connected networks, convolutional neural networks, and recurrent neural networks. Each of these models was applied to a specific problem domain, such as image reconstruction, image classification, and time-series forecasting. In this chapter, the focus shifts to a newer and more powerful class of models known as **Transformers**, which represent the current state of the art in deep learning.

Transformers were popularized by Vaswani et al. in the landmark paper *“Attention Is All You Need”*. Since their introduction, they have rapidly transformed the field of natural language processing, with major technology companies such as Google, OpenAI, and Facebook developing increasingly large and capable Transformer-based models. These models have consistently outperformed earlier architectures, particularly in language-related tasks. Although Transformers can also be applied to other domains such as computer vision, this chapter concentrates on their use in **natural language processing (NLP)**, with a specific emphasis on **machine translation**.

Understanding how Transformers work internally is essential for anyone aiming to solve complex real-world problems using deep learning. Their rapid adoption is largely due to their strong performance and scalability. To improve clarity, some advanced details from the original paper are deferred to later chapters, while this chapter focuses on the core concepts required to understand and implement a Transformer model.

5.1 Representing Text as Numbers

Before a Transformer—or any NLP model—can process text, the text must be converted into a numerical representation. Unlike images, which naturally map to numerical arrays through pixel intensities, text consists of discrete symbols such as characters and words that have no inherent numerical meaning. The challenge is therefore to transform sentences into numerical structures that machine learning models can process effectively.

This idea is introduced through an analogy called the *Word Boxes* game, where sentences must be encoded using only 0s and 1s within a fixed-size grid. The analogy highlights a core NLP problem: encoding text in a way that preserves meaning while conforming to strict structural constraints.

The first step in this process is **vocabulary indexing**. Each unique word in the dataset is assigned a unique integer ID, starting from 1, while reserving the value 0 for a special token used later. After this mapping, sentences are transformed into sequences of integers, where each number corresponds to a word in the vocabulary.

However, a practical issue arises because sentences can vary in length. Deep learning models operate on batches of data, and for efficient computation, all sequences within a batch must have the same length. To address this, shorter sentences are **padded** with a special <PAD> token, and longer sentences are **truncated** to a predefined maximum length. After this step, all sentences can be represented as a fixed-size 2D matrix, where rows correspond to sentences and columns correspond to word positions.

Once sentence lengths are standardized, each word ID is converted into a vector representation using **one-hot encoding**. In this representation, each word is mapped to a binary vector whose length equals the vocabulary size, with a single 1 at the index corresponding to the word ID and 0s elsewhere. This allows words to be represented numerically without implying any false similarity between them.

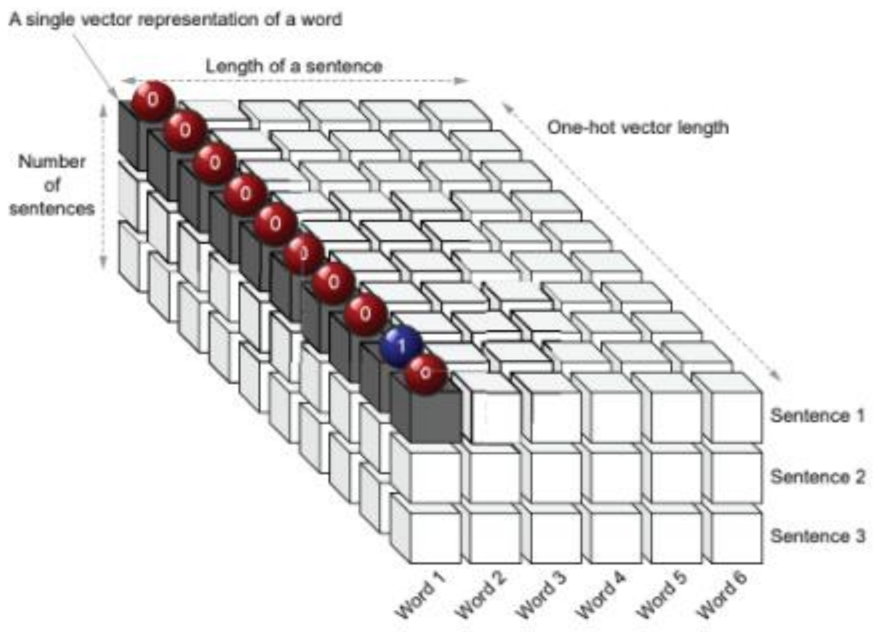


Figure 5.1 The boxes in the Word Boxes game

This figure visually demonstrates how a single word is represented using one-hot encoding, with exactly one active position corresponding to the word's ID. With access to the word-to-ID mapping, the original sentences can be reconstructed, except for words that were truncated.

Feeding raw word IDs directly into a neural network is avoided for two key reasons. First, the numerical range of word IDs in real-world vocabularies can be extremely large, which can destabilize training. Second, numerical proximity between IDs would incorrectly suggest semantic similarity between words, which is not necessarily true.

By converting word IDs into vectors, the data representation expands from a 2D matrix into a **3D tensor**, where the three dimensions correspond to **batch size**, **sequence length**, and **feature dimension**. Each word in a sentence is now represented by a vector rather than a single scalar.

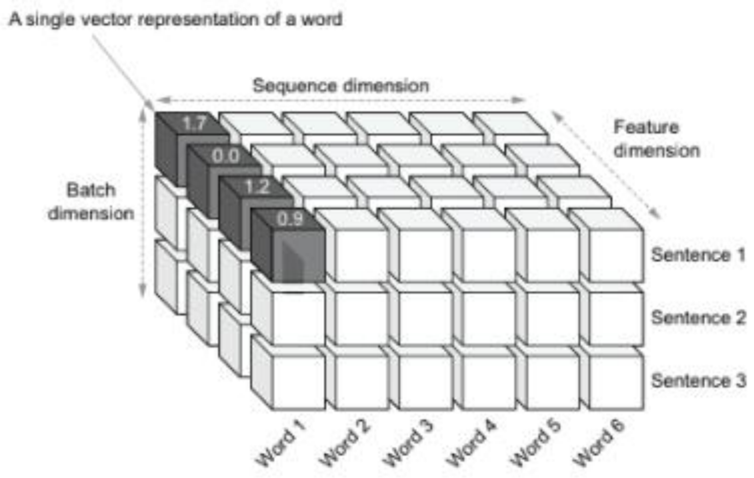


Figure 5.2 3D matrix representing a batch of word sequences

This figure illustrates the final data structure used by Transformer models, where each sentence in the batch consists of a sequence of word vectors. This structured numerical representation forms the foundation upon which Transformers operate.

With text now successfully converted into numerical form, the next step is to examine the internal components of the Transformer model and understand how they leverage these representations to perform complex NLP tasks.

5.2.1 The Encoder–Decoder View of the Transformer

The Transformer model is built around an **encoder–decoder architecture**, a pattern commonly used in deep learning tasks such as machine translation, question answering, and unsupervised reconstruction. In this paradigm, the **encoder** takes an input sequence and maps it into a latent (hidden) representation, while the **decoder** uses this latent representation to generate a meaningful output sequence. In the context of machine translation, the encoder processes a sentence in a source language and produces an internal representation from which the decoder generates the translated sentence in the target language.

The encoder and decoder can be viewed as two separate neural networks, where the decoder explicitly depends on the output of the encoder. At any given time, both components consume batches of word sequences, with each word represented as a numerical vector rather than raw text. These numerical representations are obtained using techniques such as one-hot encoding or embeddings, as discussed previously.

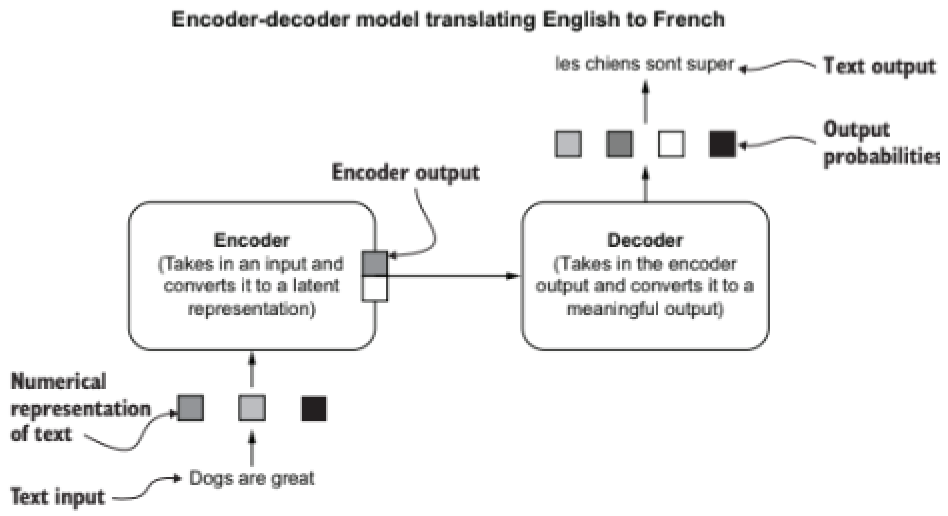

Encoder-decoder model translating English to French
les chiens sont super
Text output
Output probabilities
Encoder output
Encoder
(Takes in an input and converts it to a latent representation)
Decoder
(Takes in the encoder output and converts it to a meaningful output)
Numerical representation of text
Text input
Dogs are great

Figure 5.3 The encoder–decoder architecture for a machine translation task

A real-world analogy helps clarify this concept. Consider a tour guide translating a French menu for English-speaking tourists. The guide first builds a mental representation of the meaning of the French explanation and then expresses that meaning in English. Similarly, the encoder builds an abstract representation of the input sentence, and the decoder translates that representation into the output sentence.

5.2.2 Diving Deeper into Encoder and Decoder Layers

Although the terms "encoder" and "decoder" may recall autoencoders, the Transformer's encoder and decoder are fundamentally different. Each is a deep, multilayer neural network composed of **stacked layers**, where every layer consists of specialized **sublayers**. These layers operate on sequences, meaning that both inputs and outputs are ordered collections of elements, such as words in a sentence.

Each **encoder layer** consists of two sublayers: a **self-attention sublayer** followed by a **fully connected sublayer**. The self-attention sublayer differs significantly from a traditional fully connected layer. While a fully connected layer processes each element in the sequence independently, self-attention allows each element to selectively incorporate information from all other elements in the sequence. This enables the model to capture relationships between words regardless of their positions.

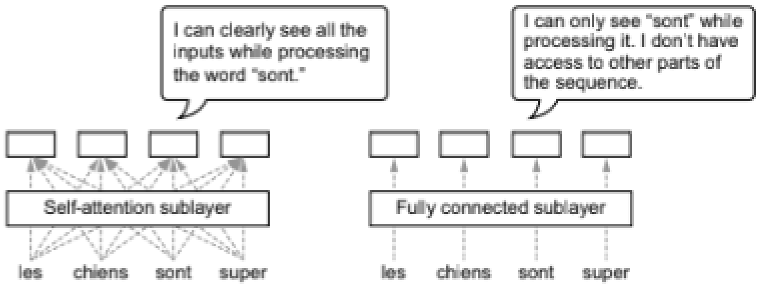

I can clearly see all the inputs while processing the word "sont."
I can only see "sont" while processing it. I don't have access to other parts of the sequence.
Self-attention sublayer
Fully connected sublayer
les
chiens
sont
super
les
chiens
sont
super

Figure 5.4 Difference between self-attention and fully connected sublayers

Self-attention is particularly powerful in NLP tasks because it allows the model to consider contextual relationships. For example, when processing the word "it" in the sentence "I kicked the ball and it disappeared," the model can attend to the word "ball" to resolve ambiguity. This ability to learn long-range dependencies is essential for accurate language understanding.

An analogy further illustrates this idea. Processing a question one word at a time without seeing the full sentence makes answering difficult, especially for complex questions. In contrast, seeing the entire question at once allows different parts to be considered simultaneously. Self-attention provides this global visibility to the model.

Following self-attention, the **fully connected sublayer** processes each position independently to produce a deeper representation. This sublayer consists of two linear transformations with a ReLU activation in between, increasing the expressive capacity of the model.

Encoder–Decoder Interaction During Translation

To illustrate data flow, consider translating the sentence "Dogs are great" into French as "Les chiens sont super." The encoder processes the entire English sentence simultaneously and produces a sequence of hidden representations, one for each word. The decoder then generates the translation **iteratively**, one word at a time.

The decoding process begins with a special <SOS> (start-of-sentence) token. Using this token and the encoder outputs, the decoder predicts the first word of the translation. The predicted word is then fed back into the decoder, along with the previous outputs, to generate the next word. This process continues until the model produces an <EOS> (end-of-sentence) token.

In the original Transformer architecture, the encoder consists of **six identical layers**, each containing a self-attention sublayer followed by a fully connected sublayer. Words are represented using embeddings (with additional encoding), where each embedding is a 512-dimensional vector. The self-attention mechanism computes an attended representation for each word by forming a weighted combination of all input words, with weights learned to reflect contextual importance.

Decoder Layers and Attention Mechanisms

The decoder also consists of six layers, but each decoder layer contains **three sublayers**: a **masked self-attention sublayer**, an **encoder–decoder attention sublayer**, and a **fully connected sublayer**.

The masked self-attention sublayer ensures that, while predicting a word, the decoder can only attend to previously generated words and not future ones. This constraint is crucial because, during inference, future words are not yet available.

The encoder–decoder attention sublayer allows the decoder to attend to the encoder's outputs. At each decoding step, it computes an attended representation by weighting encoder outputs according to their relevance to the current decoder position. This mechanism enables the decoder to align its predictions with relevant parts of the source sentence.

Finally, the fully connected sublayer transforms the attended representation into the output of the decoder layer, in the same manner as in the encoder.

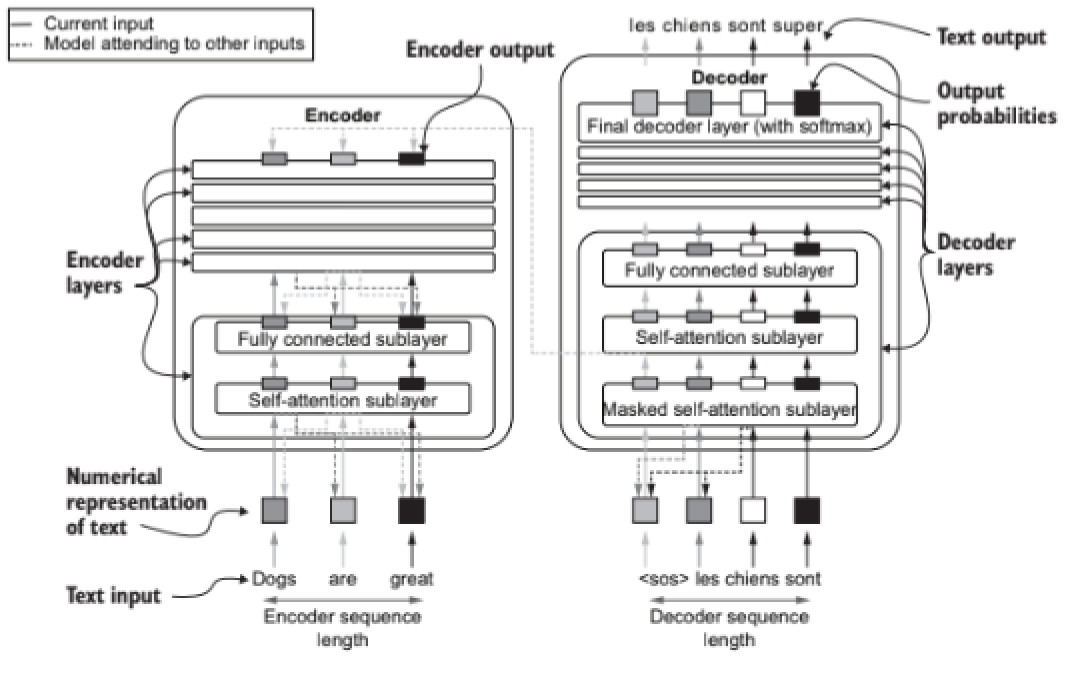


Figure 5.5 Encoder and decoder layers and their internal connections

This layered organization of attention-based sublayers allows the Transformer to model complex dependencies within and across sequences, forming the foundation of its superior performance in sequence-to-sequence tasks such as machine translation.

5.2.3 Self-Attention Layer

The self-attention layer is designed to determine, while processing a word $w_t$at time step t, how important every other word $w_i$in the input sequence is for understanding that word. In practical terms, the layer computes pairwise relationships between all words in the sequence, enabling each word to selectively incorporate contextual information from all other words. This mechanism forms the foundation of the Transformer's ability to model long-range dependencies.

Three core entities participate in the self-attention computation: **queries**, **keys**, and **values**. A query represents the word currently being processed, keys represent candidate words that may be attended to, and values represent the information that will be aggregated to form the output. For each position in the input sequence, a query, key, and value vector is computed using learned weight matrices. Although the exact mathematical relationships are complex, this abstraction explains why three distinct representations are necessary.

The process begins by converting the input word sequence into numerical form using word embeddings. If the input sequence contains $n$words and each embedding has dimensionality d_{model}, the resulting matrix has shape $n \times d_{model}$. In the original Transformer architecture, $d_{model} = 512$. Three learned weight matrices—W_q, W_k, and W_v—are then used to project the input into query, key, and value spaces. These matrices map the embeddings into vectors of dimensions d_q, d_k, and d_v, respectively, which are typically set equal to d_{model}.

The queries, keys, and values are computed using matrix multiplication, producing tensors that represent all positions in the sequence simultaneously. Ignoring the batch dimension for clarity, the final output of the self-attention layer is computed by first forming a **probability matrix** using the scaled dot product of queries and keys, followed by a softmax operation. This probability matrix captures how strongly each word attends to every other word. The output is then obtained by multiplying this probability matrix with the value vectors, producing a new sequence of vectors with the same length as the input.

The self-attention mechanism is straightforward to implement in TensorFlow and can be encapsulated as a reusable Keras layer. The layer defines its parameters during initialization and build phases and performs the query–key–value computations in its call function. In practice, the same input sequence is often used to compute queries, keys, and values, although different inputs may be used in certain decoder computations. Returning both the final output and the attention probability matrix allows for easier visualization and interpretation of attention behavior.

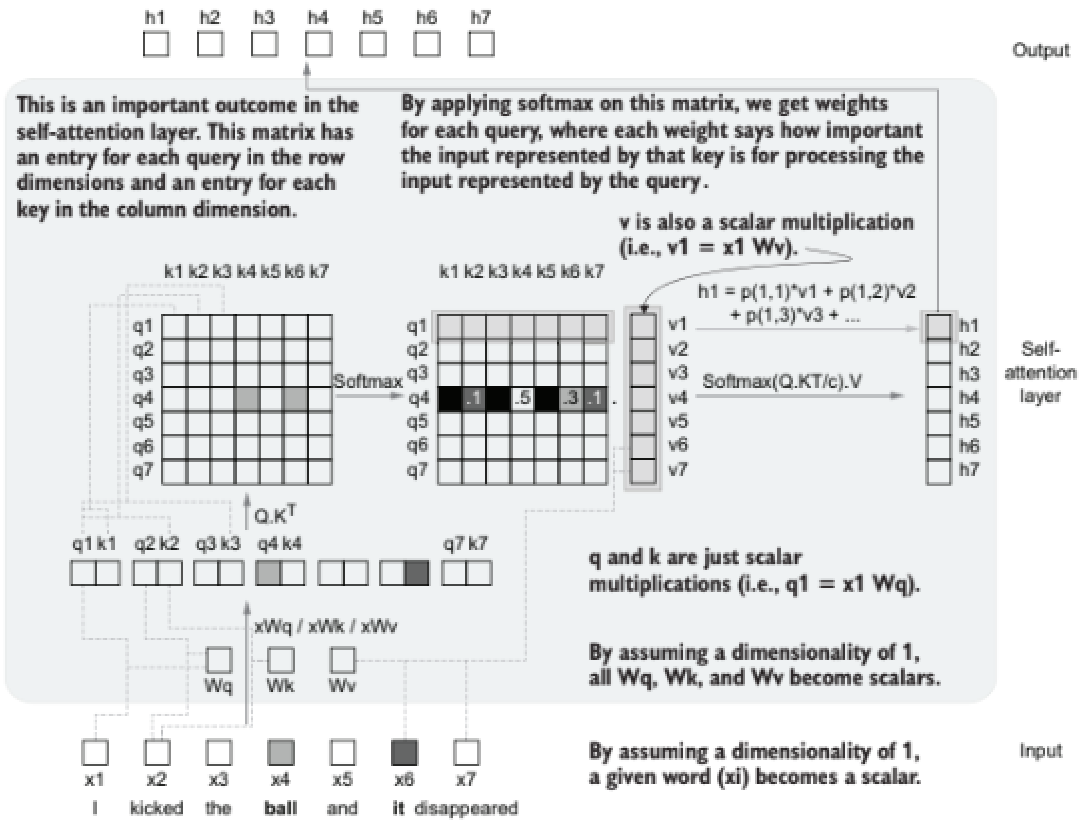
5.2.4 Understanding Self-Attention Using Scalars

To build intuition, the self-attention computation can be simplified by assuming a feature dimensionality of one, where each word is represented by a single scalar. Under this assumption, the input sequence becomes an $n \times 1$matrix, and the query, key, and value weight matrices reduce to scalars. Consequently, computing queries, keys, and values becomes simple scalar multiplication for each position in the sequence.

Next, the attention probability matrix is computed by taking pairwise products between queries and keys, resulting in an $n \times n$matrix. Each element represents the interaction between a query at position $i$and a key at position j. Applying the softmax function row-wise converts this matrix into a set of probability distributions, where each row sums to one. The normalization term $\sqrt{d_k}$is included to stabilize gradients, though it simplifies to one in this scalar example.

The final output is obtained by computing a weighted sum of the value scalars for each position, using the probabilities from the corresponding row of the attention matrix. This makes the role of queries, keys, and values explicit. Queries determine which row of the attention matrix is selected, keys determine how values are weighted within that row, and values provide the information that is aggregated to produce the output.

This scalar example clarifies the conceptual roles of the three entities. Queries and keys together form a soft indexing mechanism that determines how information is retrieved from the values. Each output is therefore a context-aware representation of the input, influenced by all other elements in the sequence.



h1 h2 h3 h4 h5 h6 h7
Output
This is an important outcome in the self-attention layer. This matrix has an entry for each query in the row dimensions and an entry for each key in the column dimension.
By applying softmax on this matrix, we get weights for each query, where each weight says how important the input represented by that key is for processing the input represented by the query.
v is also a scalar multiplication (i.e., v1 = x1 Wv).
k1 k2 k3 k4 k5 k6 k7
q1 q2 q3 q4 q5 q6 q7
Softmax
.1 .5 .3 .1
v1 v2 v3 v4 v5 v6 v7
h1 = p(1,1)*v1 + p(1,2)*v2 + p(1,3)*v3 + ...
Softmax(Q.KT/c).V
h1 h2 h3 h4 h5 h6 h7
Self-attention layer
Q.K^T
q1 k1 q2 k2 q3 k3 q4 k4 q7 k7
q and k are just scalar multiplications (i.e., q1 = x1 Wq).
xWq / xWk / xWv
Wq Wk Wv
By assuming a dimensionality of 1, all Wq, Wk, and Wv become scalars.
x1 x2 x3 x4 x5 x6 x7
I kicked the ball and it disappeared
By assuming a dimensionality of 1, a given word (xi) becomes a scalar.
Input

Figure 5.6 The computations in the self-attention layer

The same logic extends naturally to higher-dimensional embeddings. With a sequence of $n$words represented as vectors of size d_{model}, queries, keys, and values are computed via matrix multiplications using learned weight matrices. The probability matrix is formed using the scaled dot-product attention formula, and multiplying this matrix with the value vectors yields the final output sequence.

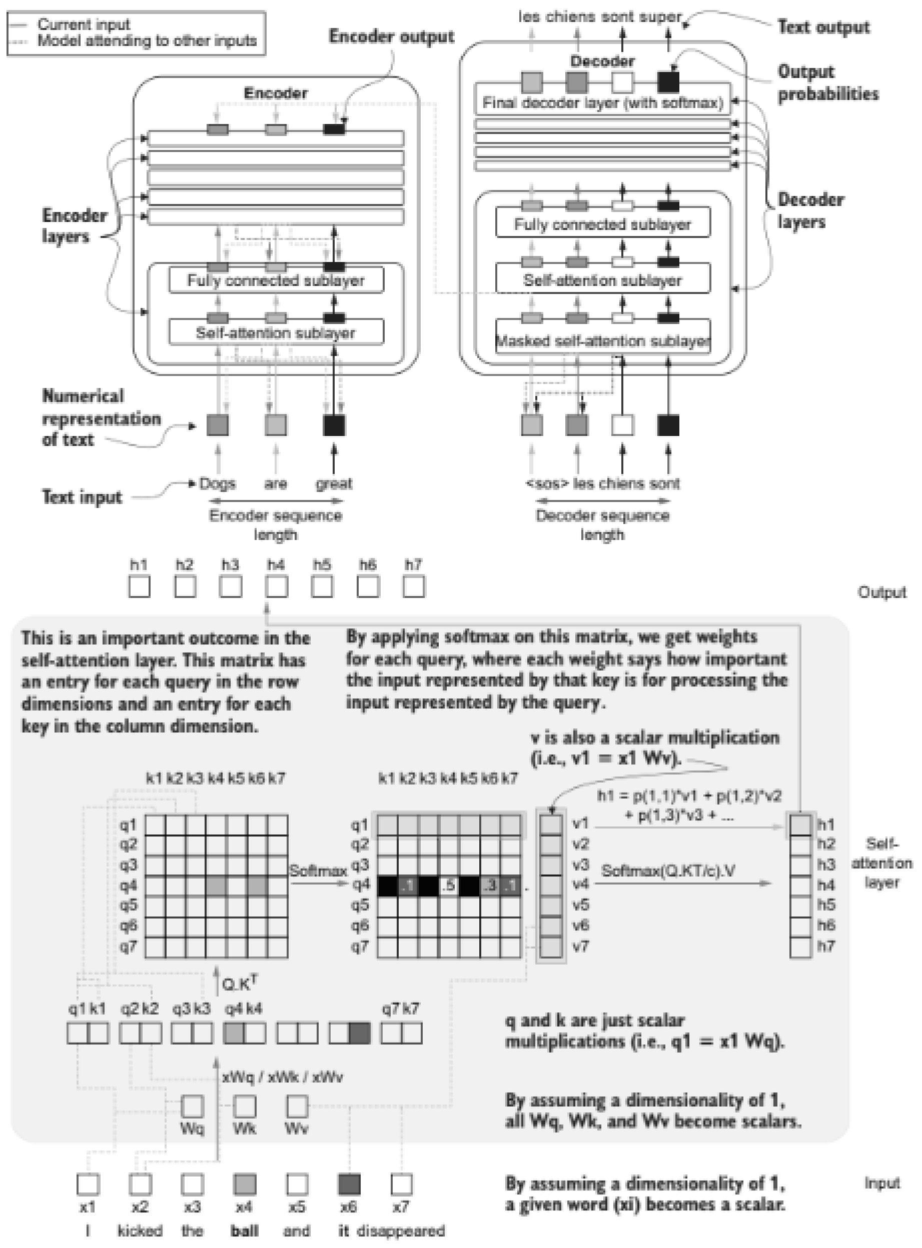

Current input
Model attending to other inputs
Encoder output
Encoder
Encoder layers
Fully connected sublayer
Self-attention sublayer
Numerical representation of text
Text input
Dogs are great
Encoder sequence length
les chiens sont super
Text output
Decoder
Output probabilities
Final decoder layer (with softmax)
Decoder layers
Fully connected sublayer
Self-attention sublayer
Masked self-attention sublayer
<sos> les chiens sont
Decoder sequence length
h1 h2 h3 h4 h5 h6 h7
Output
This is an important outcome in the self-attention layer. This matrix has an entry for each query in the row dimensions and an entry for each key in the column dimension.
By applying softmax on this matrix, we get weights for each query, where each weight says how important the input represented by that key is for processing the input represented by the query.
v is also a scalar multiplication (i.e., v1 = x1 Wv).
k1 k2 k3 k4 k5 k6 k7
q1 q2 q3 q4 q5 q6 q7
Softmax
.1 .5 .3 .1
v1 v2 v3 v4 v5 v6 v7
h1 = p(1,1)*v1 + p(1,2)*v2 + p(1,3)*v3 + ...
Softmax(Q.KT/c).V
Self-attention layer
Q.K^T
q1 k1 q2 k2 q3 k3 q4 k4 q7 k7
q and k are just scalar multiplications (i.e., q1 = x1 Wq).
xWq / xWk / xWv
Wq Wk Wv
By assuming a dimensionality of 1, all Wq, Wk, and Wv become scalars.
By assuming a dimensionality of 1, a given word (xi) becomes a scalar.
Input
x1 x2 x3 x4 x5 x6 x7
I kicked the ball and it disappeared

Figure 5.7 Relationship between abstract Transformer architecture and scalar self-attention computation

The self-attention layer takes a batch of fixed-length word sequences and produces a batch of equally long sequences of hidden vectors. Each output vector encodes information from the entire input sequence, weighted by learned attention scores.

Self-Attention versus Recurrent Neural Networks

Before Transformers, recurrent neural networks dominated NLP tasks because they naturally process sequential data. RNNs consume one word at a time while maintaining an internal state that summarizes past inputs. However, as sequences grow longer, RNNs struggle to preserve early information, leading to degraded performance.

Self-attention addresses this limitation by allowing the model to access the entire sequence simultaneously at every processing step. This global view eliminates the need to rely on long-term memory propagation and enables Transformers to outperform RNN-based models, particularly on long and complex sequences.

5.2.5 Self-Attention as a Cooking Competition

An analogy using a cooking competition is introduced to convey the intuition behind self-attention. Contestants correspond to queries, grocery items correspond to keys, and the prepared beverages correspond to values. Each contestant selects ingredients in different proportions depending on their assigned task, analogous to attention weight distribution.

This analogy reinforces the idea that self-attention dynamically mixes information across the entire sequence based on relevance.

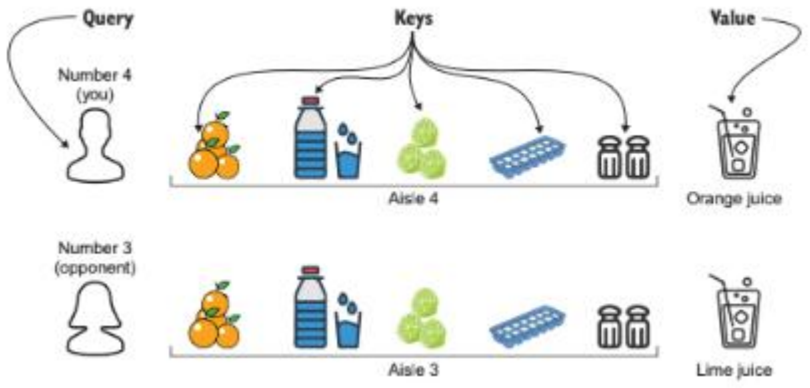


Figure 5.8 Self-attention depicted with the help of a cooking competition. The contestants are the queries, the keys are the grocery items you have to choose from, and the values are the final beverage you're making.

5.2.6 Masked Self-Attention Layers

Masked self-attention is introduced to prevent the decoder from attending to future tokens during training. This ensures causal consistency and prevents information leakage that would otherwise lead to unrealistic model performance.

Masking is implemented by modifying the attention score matrix so that positions corresponding to future tokens receive negligible probability mass after softmax.

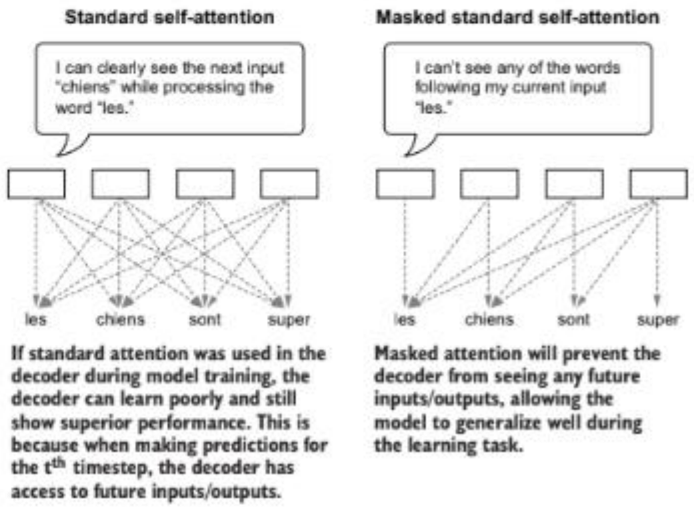
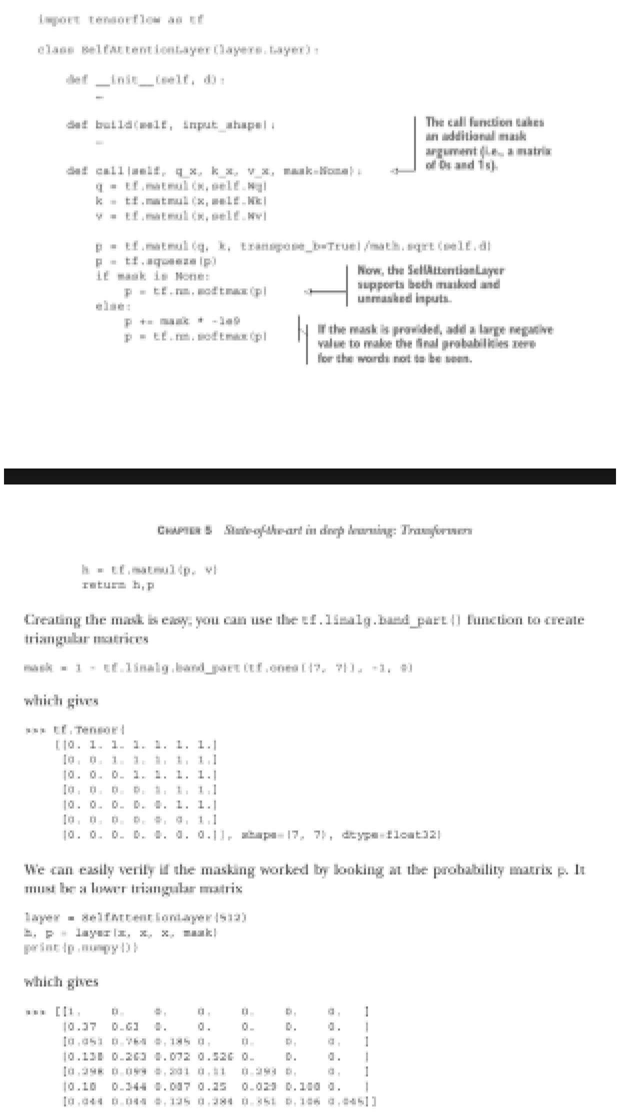

Standard self-attention
I can clearly see the next input "chiens" while processing the word "les."
les
chiens
sont
super
If standard attention was used in the decoder during model training, the decoder can learn poorly and still show superior performance. This is because when making predictions for the t^th timestep, the decoder has access to future inputs/outputs.
Masked standard self-attention
I can't see any of the words following my current input "les."
les
chiens
sont
super
Masked attention will prevent the decoder from seeing any future inputs/outputs, allowing the model to generalize well during the learning task.

Figure 5.9 Standard self-attention versus masked self-attention methods. In the standard attention method, a given step can see an input from any other timestep, regardless of whether those inputs appear before or after the current time step. However, in the masked self-attention method, the current timestep can only see the current input and what came before that time step.



```
import tensorflow as tf

class SelfAttentionLayer(layers.Layer):

    def __init__(self, d):
        ...

    def build(self, input_shape):
        ...

    def call(self, q_x, k_x, v_x, mask=None):
        q = tf.matmul(x,self.Wq)
        k = tf.matmul(x,self.Wk)
        v = tf.matmul(x,self.Wv)

        p = tf.matmul(q, k, transpose_b=True)/math.sqrt(self.d)
        p = tf.squeeze(p)
        if mask is None:
            p = tf.nn.softmax(p)
        else:
            p += mask * -1e9
            p = tf.nn.softmax(p)
```

The call function takes an additional mask argument (i.e., a matrix of 0s and 1s).

Now, the SelfAttentionLayer supports both masked and unmasked inputs.

If the mask is provided, add a large negative value to make the final probabilities zero for the words not to be seen.

CHAPTER 5 *State-of-the-art in deep learning: Transformers*

```
        h = tf.matmul(p, v)
        return h,p
```

Creating the mask is easy; you can use the `tf.linalg.band_part()` function to create triangular matrices

```
mask = 1 - tf.linalg.band_part(tf.ones((7, 7)), -1, 0)
```

which gives

```
>>> tf.Tensor(
    [[0. 1. 1. 1. 1. 1. 1.]
     [0. 0. 1. 1. 1. 1. 1.]
     [0. 0. 0. 1. 1. 1. 1.]
     [0. 0. 0. 0. 1. 1. 1.]
     [0. 0. 0. 0. 0. 1. 1.]
     [0. 0. 0. 0. 0. 0. 1.]
     [0. 0. 0. 0. 0. 0. 0.]], shape=(7, 7), dtype=float32)
```

We can easily verify if the masking worked by looking at the probability matrix p. It must be a lower triangular matrix

```
layer = SelfAttentionLayer(512)
h, p = layer(x, x, x, mask)
print(p.numpy())
```

which gives

```
>>> [[1.    0.    0.    0.    0.    0.    0.   ]
     [0.37  0.63  0.    0.    0.    0.    0.   ]
     [0.051 0.764 0.185 0.    0.    0.    0.   ]
     [0.138 0.263 0.072 0.526 0.    0.    0.   ]
     [0.298 0.099 0.201 0.11  0.293 0.    0.   ]
     [0.18  0.344 0.087 0.25  0.029 0.108 0.   ]
     [0.044 0.044 0.125 0.284 0.351 0.106 0.045]]
```

Listing 5.2 Masked self-attention sublayer

5.2.7 Multi-Head Attention

Multi-head attention extends self-attention by executing multiple attention operations in parallel. Each attention head operates on a lower-dimensional subspace, and their outputs are concatenated to form the final representation.

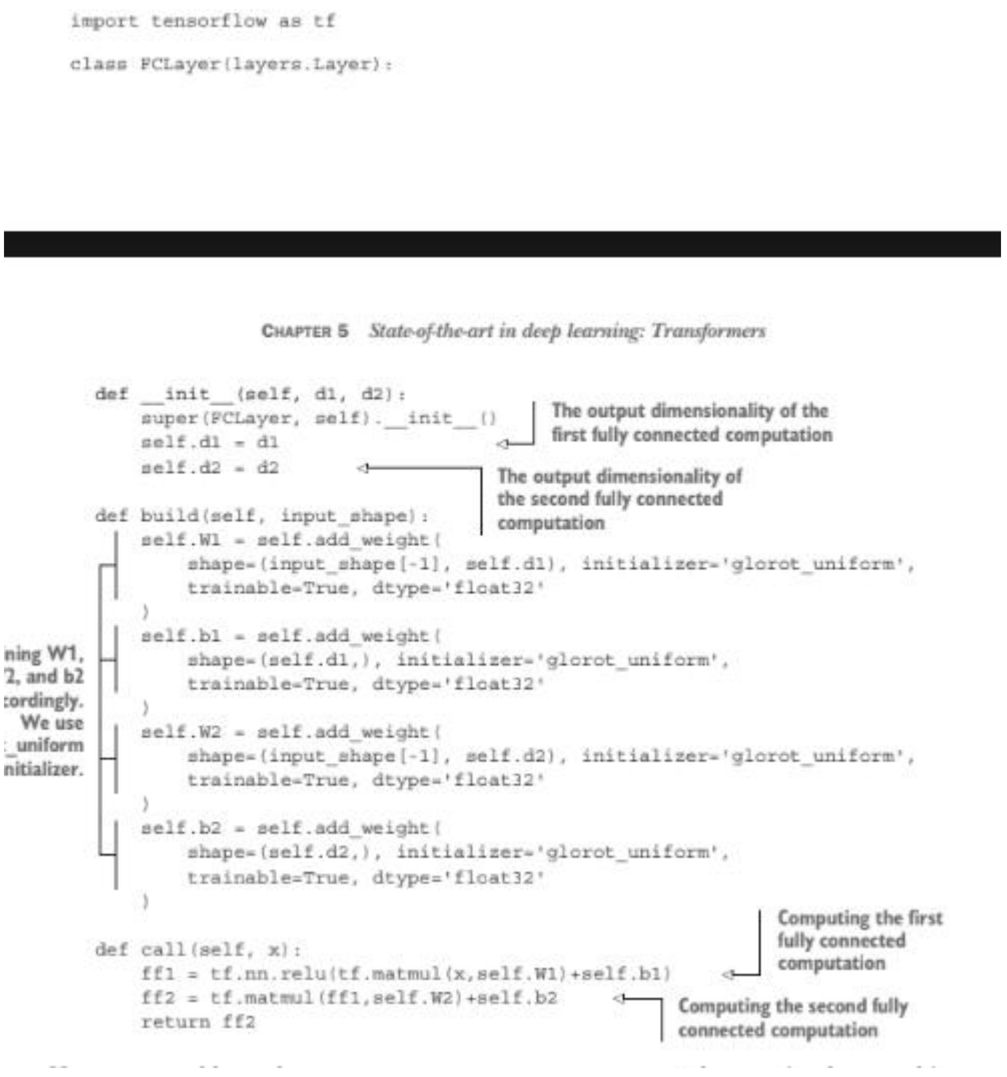
This design allows the model to capture diverse attention patterns simultaneously while maintaining the original output dimensionality.

(No new figures introduced in this subsection.)

5.2.8 Fully Connected Layer

The fully connected sublayer applies two linear transformations with a nonlinear activation function in between to each position independently. This sublayer increases the representational capacity of the model following the attention mechanism.

The implementation can be done using raw TensorFlow operations or Keras Dense layers.



```
import tensorflow as tf

class FCLayer(layers.Layer):
```

```
    def __init__(self, d1, d2):
        super(FCLayer, self).__init__()
        self.d1 = d1
        self.d2 = d2

    def build(self, input_shape):
        self.W1 = self.add_weight(
            shape=(input_shape[-1], self.d1), initializer='glorot_uniform',
            trainable=True, dtype='float32'
        )
        self.b1 = self.add_weight(
            shape=(self.d1,), initializer='glorot_uniform',
            trainable=True, dtype='float32'
        )
        self.W2 = self.add_weight(
            shape=(input_shape[-1], self.d2), initializer='glorot_uniform',
            trainable=True, dtype='float32'
        )
        self.b2 = self.add_weight(
            shape=(self.d2,), initializer='glorot_uniform',
            trainable=True, dtype='float32'
        )

    def call(self, x):
        ff1 = tf.nn.relu(tf.matmul(x,self.W1)+self.b1)
        ff2 = tf.matmul(ff1,self.W2)+self.b2
        return ff2
```

The output dimensionality of the first fully connected computation

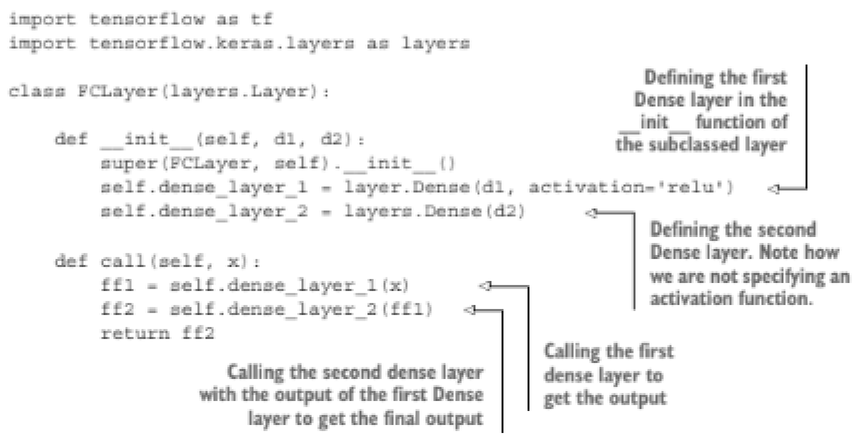
The output dimensionality of the second fully connected computation

ning W1, 2, and b2 cordingly. We use _uniform nitializer.

Computing the first fully connected computation

Computing the second fully connected computation

Listing 5.3 The fully connected sublayer



```
import tensorflow as tf
import tensorflow.keras.layers as layers

class FCLayer(layers.Layer):

    def __init__(self, d1, d2):
        super(FCLayer, self).__init__()
        self.dense_layer_1 = layer.Dense(d1, activation='relu')
        self.dense_layer_2 = layers.Dense(d2)

    def call(self, x):
        ff1 = self.dense_layer_1(x)
        ff2 = self.dense_layer_2(ff1)
        return ff2
```

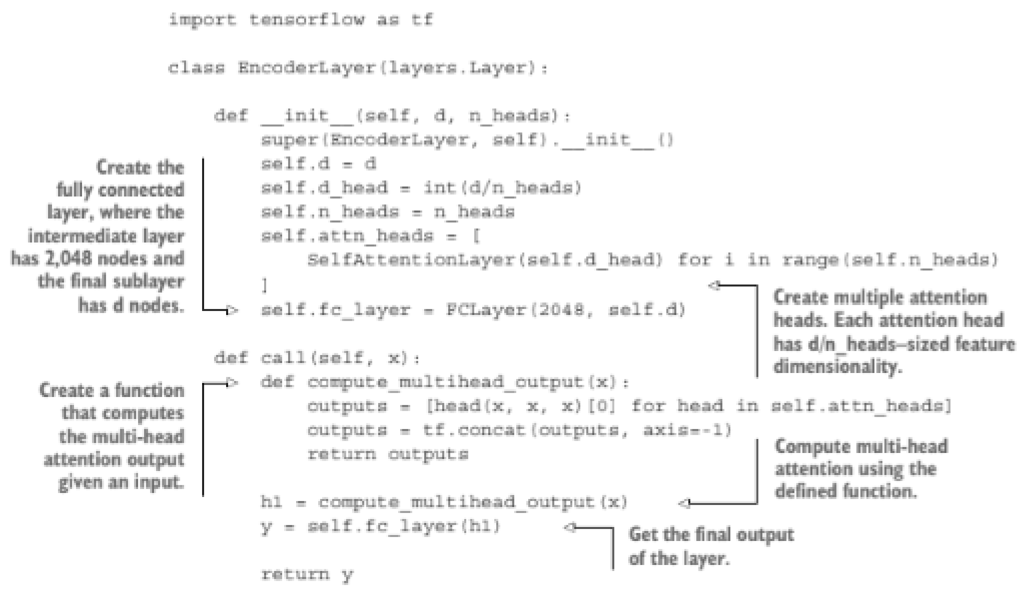
Defining the first Dense layer in the __init__ function of the subclassed layer
Defining the second Dense layer. Note how we are not specifying an activation function.
Calling the first dense layer to get the output
Calling the second dense layer with the output of the first Dense layer to get the final output

Listing 5.4 The fully connected layer implemented using Keras Dense layers

5.2.9 Putting Everything Together

This section integrates all previously discussed components to construct a complete Transformer network. The architecture follows an **encoder–decoder structure**, where both encoder and decoder are composed of self-attention mechanisms and fully connected layers.

The construction begins with the **encoder layer**, which consists of multiple self-attention heads followed by a fully connected layer. Each attention head independently attends to the input sequence, and their outputs are combined to produce the encoder representation.



```
import tensorflow as tf

class EncoderLayer(layers.Layer):

    def __init__(self, d, n_heads):
        super(EncoderLayer, self).__init__()
        self.d = d
        self.d_head = int(d/n_heads)
        self.n_heads = n_heads
        self.attn_heads = [
            SelfAttentionLayer(self.d_head) for i in range(self.n_heads)
        ]
        self.fc_layer = FCLayer(2048, self.d)

    def call(self, x):
        def compute_multihead_output(x):
            outputs = [head(x, x, x)[0] for head in self.attn_heads]
            outputs = tf.concat(outputs, axis=-1)
            return outputs

        h1 = compute_multihead_output(x)
        y = self.fc_layer(h1)

        return y
```

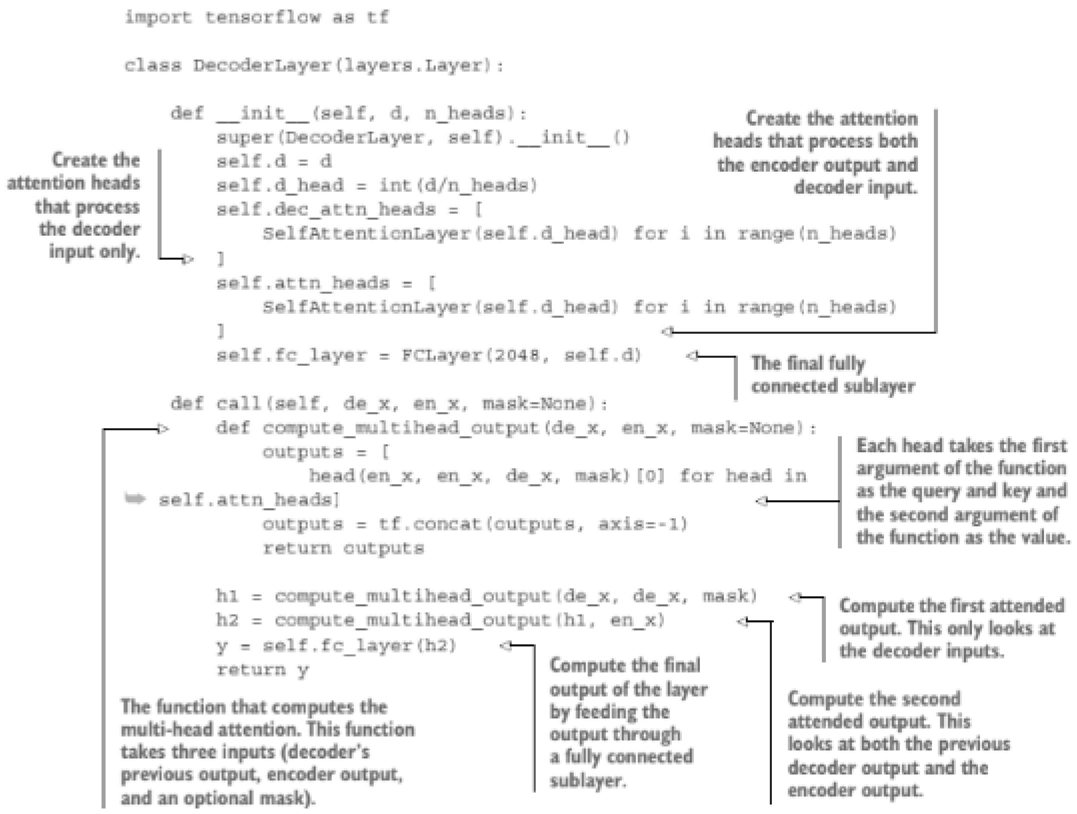
Create the fully connected layer, where the intermediate layer has 2,048 nodes and the final sublayer has d nodes.
Create multiple attention heads. Each attention head has d/n_heads–sized feature dimensionality.
Create a function that computes the multi-head attention output given an input.
Compute multi-head attention using the defined function.
Get the final output of the layer.

Listing 5.5 The encoder layer

Decoder Layer Construction

The decoder layer differs from the encoder in two fundamental ways. First, it contains **two multi-head attention sublayers**: a masked self-attention layer and an unmasked encoder–decoder attention layer. Second, it includes a final fully connected layer that produces output predictions.

The masked self-attention layer ensures that, during training, the decoder cannot attend to future tokens. The second attention layer combines information from the encoder outputs (as query and key) with the decoder's own intermediate outputs (as values), effectively mixing source and target representations.



```
import tensorflow as tf

class DecoderLayer(layers.Layer):

    def __init__(self, d, n_heads):
        super(DecoderLayer, self).__init__()
        self.d = d
        self.d_head = int(d/n_heads)
        self.dec_attn_heads = [
            SelfAttentionLayer(self.d_head) for i in range(n_heads)
        ]
        self.attn_heads = [
            SelfAttentionLayer(self.d_head) for i in range(n_heads)
        ]
        self.fc_layer = FCLayer(2048, self.d)

    def call(self, de_x, en_x, mask=None):
        def compute_multihead_output(de_x, en_x, mask=None):
            outputs = [
                head(en_x, en_x, de_x, mask)[0] for head in
➥ self.attn_heads]
            outputs = tf.concat(outputs, axis=-1)
            return outputs

        h1 = compute_multihead_output(de_x, de_x, mask)
        h2 = compute_multihead_output(h1, en_x)
        y = self.fc_layer(h2)
        return y
```

Create the attention heads that process both the encoder output and decoder input.

Create the attention heads that process the decoder input only.

The final fully connected sublayer

Each head takes the first argument of the function as the query and key and the second argument of the function as the value.

Compute the first attended output. This only looks at the decoder inputs.

Compute the final output of the layer by feeding the output through a fully connected sublayer.

Compute the second attended output. This looks at both the previous decoder output and the encoder output.

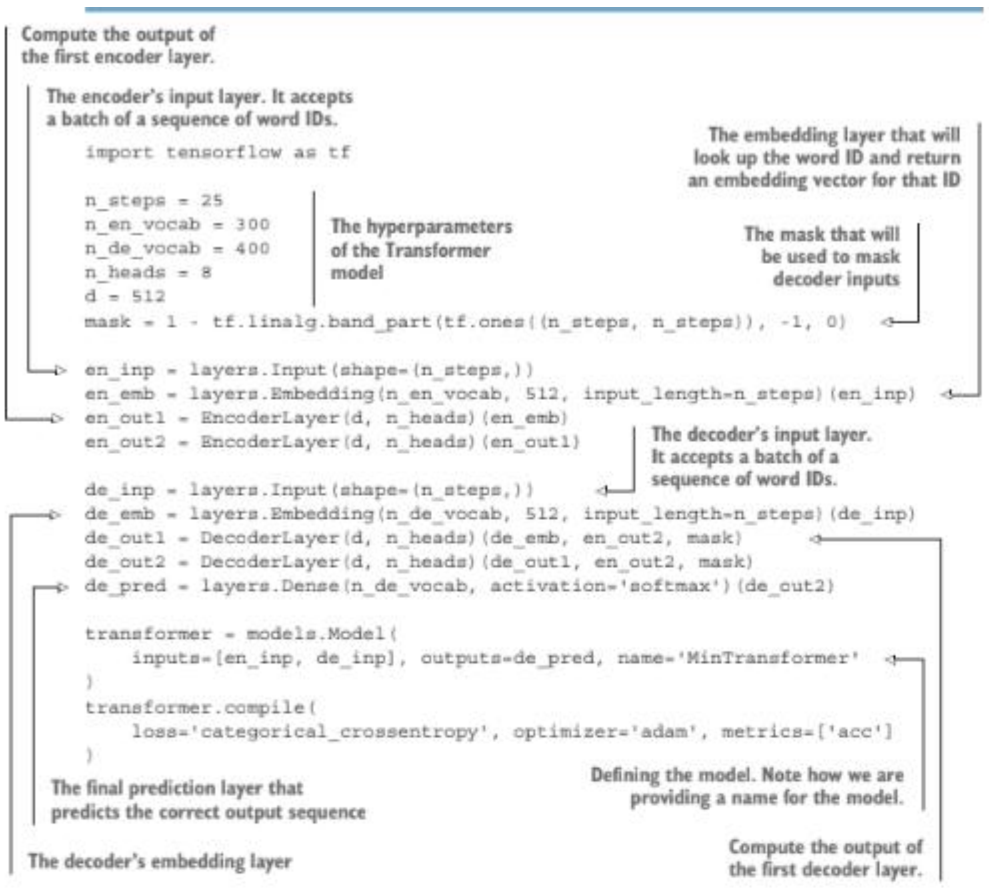
The function that computes the multi-head attention. This function takes three inputs (decoder's previous output, encoder output, and an optional mask).

Listing 5.6 The DecoderLayer

Full Transformer Model Assembly

With encoder and decoder layers defined, a full Transformer model is constructed using the **Keras Functional API**. The model contains two encoder layers and two decoder layers stacked sequentially. Hyperparameters such as sequence length (n_steps), vocabulary sizes for encoder and decoder, number of attention heads, and embedding dimensionality are defined prior to model construction.

Before examining the implementation, it is useful to revisit the overall Transformer architecture.



Compute the output of the first encoder layer.

The encoder's input layer. It accepts a batch of a sequence of word IDs.

The embedding layer that will look up the word ID and return an embedding vector for that ID

The hyperparameters of the Transformer model

The mask that will be used to mask decoder inputs

The decoder's input layer. It accepts a batch of a sequence of word IDs.

```
import tensorflow as tf

n_steps = 25
n_en_vocab = 300
n_de_vocab = 400
n_heads = 8
d = 512
mask = 1 - tf.linalg.band_part(tf.ones((n_steps, n_steps)), -1, 0)

en_inp = layers.Input(shape=(n_steps,))
en_emb = layers.Embedding(n_en_vocab, 512, input_length=n_steps)(en_inp)
en_out1 = EncoderLayer(d, n_heads)(en_emb)
en_out2 = EncoderLayer(d, n_heads)(en_out1)

de_inp = layers.Input(shape=(n_steps,))
de_emb = layers.Embedding(n_de_vocab, 512, input_length=n_steps)(de_inp)
de_out1 = DecoderLayer(d, n_heads)(de_emb, en_out2, mask)
de_out2 = DecoderLayer(d, n_heads)(de_out1, en_out2, mask)
de_pred = layers.Dense(n_de_vocab, activation='softmax')(de_out2)

transformer = models.Model(
    inputs=[en_inp, de_inp], outputs=de_pred, name='MinTransformer'
)
transformer.compile(
    loss='categorical_crossentropy', optimizer='adam', metrics=['acc']
)
```

The final prediction layer that predicts the correct output sequence

Defining the model. Note how we are providing a name for the model.

The decoder's embedding layer

Compute the output of the first decoder layer.

Listing 5.7 The full Transformer model

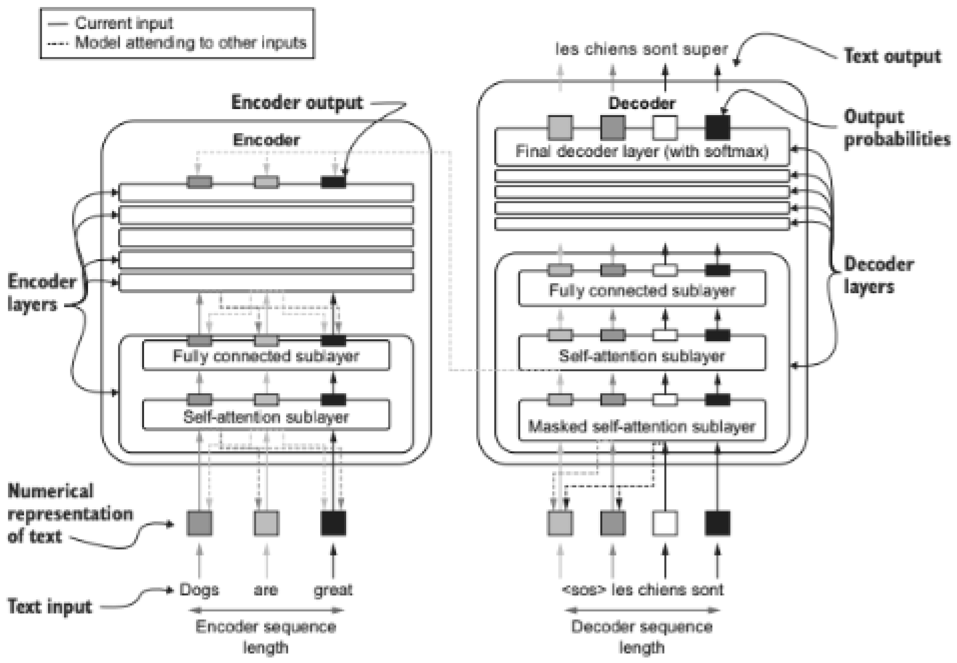


Figure 5.10 The Transformer model architecture

Input Representation and Embedding

The encoder input consists of fixed-length sequences of word IDs. Each ID is mapped to a dense vector representation using an **Embedding layer**, resulting in tensors of shape *(batch size, n_steps, d)*. These embeddings serve as inputs to the encoder's self-attention layers.

Padding and truncation are applied to ensure uniform sequence length. Larger sequence lengths preserve more contextual information but increase memory consumption.

Decoder Inputs and Outputs

The decoder receives both the encoder's final output and its own input embeddings. Unlike the encoder, the decoder includes a final Dense layer with softmax activation that outputs probability distributions over the target vocabulary for each time step.

The full Transformer model is then compiled using categorical cross-entropy loss and the Adam optimizer.

Model Summary and Parameterization

The model summary confirms the layered structure of the Transformer and provides insight into parameter counts and tensor dimensions across layers. The architecture includes embedding layers, stacked encoder and decoder layers, and a final Dense output layer. All parameters in the model are trainable.

Section Conclusion

This section demonstrates how individual Transformer components—self-attention layers, masked attention, multi-head attention, and fully connected layers—are composed into a complete, functional model. By assembling reusable custom layers, a compact yet fully operational Transformer architecture is achieved.

The implementation serves as a foundation for training Transformer models on NLP tasks such as machine translation. More advanced topics, including large-scale training and pretrained Transformer models, are deferred to later chapters.