Language is the essence of human communication. Civilizations would never have been born without the word sequences that form language. We now mostly live in a world of digital representations of language. Our daily lives rely on NLP digitalized language functions: web search engines, emails, social networks, posts, tweets, smartphone texting, translations, web pages, speech-to-text on streaming sites for transcripts, text-to-speech on hotline services, and many more everyday functions.*Chapter 1*, *What are Transformers?*, explained the limits of RNNs and the birth of cloud AI transformers taking over a fair share of design and development. The role of the AI specialist is to understand the architecture of the original Transformer and the multiple transformer ecosystems that followed.In December 2017, Google Brain and Google Research published the seminal *Vaswani* et al. *Attention is All You Need* paper. The Transformer was born. The Transformer outperformed the existing state-of-the-art NLP models. The Transformer trained faster than previous architectures and obtained higher evaluation results. As a result, transformers have become a key component of NLP.Since 2017, transformer models such as OpenAI's ChatGPT and GPT-4, Google's PaLM and LaMBDA, and other **Large Language Models** (**LLMs**) have emerged. However, this is just the beginning! You need to understand how attention heads work to join this new era of LLM for AI experts.The idea of the attention head of the Transformer is to do away with recurrent neural network features. In this chapter, we will open the hood of the original Transformer model described by *Vaswani* et al. (2017) and examine the main components of its architecture. Then, we will explore the fascinating world of attention and illustrate the key components of the Transformer.This chapter covers the following topics:

- The architecture of the Transformer
- The Transformer's self-attention model
- The encoding and decoding stacks
- Input and output embedding
- Positional embedding
- Self-attention
- Multi-head attention
- Masked multi-attention
- Residual connections
- Normalization
- Feedforward network
- Output probabilities

Let's dive directly into the structure of the original Transformer's architecture.

The Rise of the Transformer: Attention is All You Need

As mentioned earlier, in December 2017, *Vaswani* et al. (2017) published their seminal paper, *Attention is All You Need.* They performed their work at Google Research and Google Brain. I will refer to the model described in *Attention is All You Need* as the "original Transformer model" throughout this chapter and book.Exploring the architecture of the Transformer is essential for the following reasons:

- The source code of any machine learning or deep learning model remains a product of classical mathematics.
- It's only when the system is put to work that it becomes artificial intelligence.
- Artificial intelligence is the function of a model built with classical mathematics.
- The Transformer is no exception. If you look inside the algorithm, you will discover classical mathematics.

> The magic of AI is not in math but in the **behavior** of the system when it actually runs and performs tasks.
>
> Sometimes, we must get our hands dirty to find ways to optimize the functions implemented in our AI (or any other software).

In this section, we will look at the structure of the Transformer model they built. In the following sections, we will explore what is inside each component of the model.The original Transformer model is a stack of 6 layers. Each layer contains sub-layers. The output of layer *l* is the input of layer *l*+1 until the final prediction is reached. There is a 6-layer encoder stack on the left and a 6-layer decoder stack on the right:

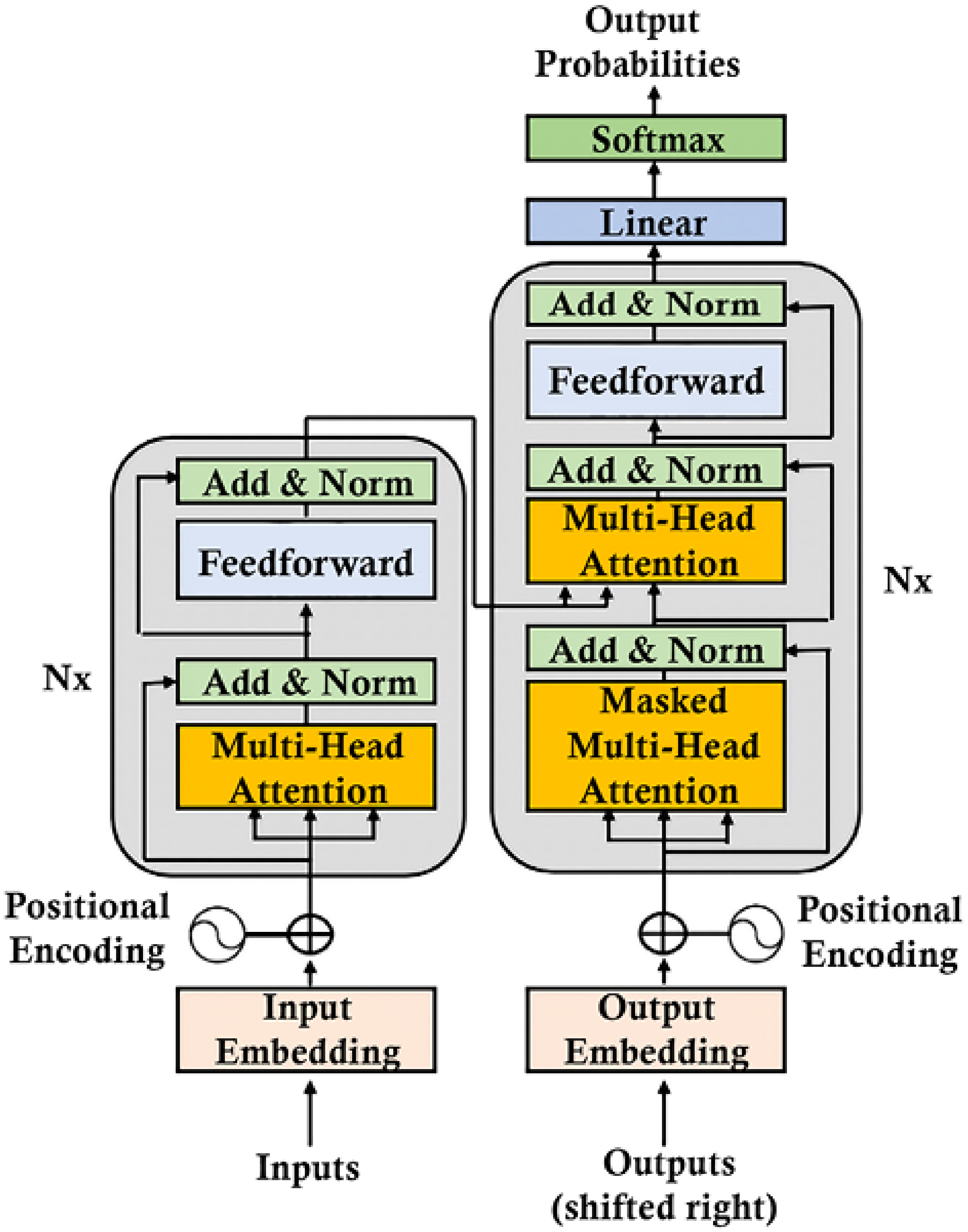


Figure 2.1: The architecture of the Transformer

On the left, the inputs enter the encoder side of the Transformer through an attention sublayer and a feedforward sublayer. On the right, the target outputs go into the decoder side of the Transformer through two attention sublayers and a feedforward network sublayer. We immediately notice that there is no RNN, LSTM, or CNN. This is because recurrence has been abandoned in this architecture.Attention has replaced recurrence functions requiring increasing parameters an increase in the number of parameters as the distance between two words increases. The attention mechanism is a "word-to-word" operation. It is actually a token-to-token operation, but we will keep it to the word level to keep the explanation simple. The attention mechanism will find how each word is related to all other words in a sequence, including the word being analyzed itself. For example, let's examine the following sequence:

```
The cat sat on the mat.
```

Attention will run dot products between word vectors and determine the strongest relationships of a word with all the other words, including itself ("cat" and "cat"):

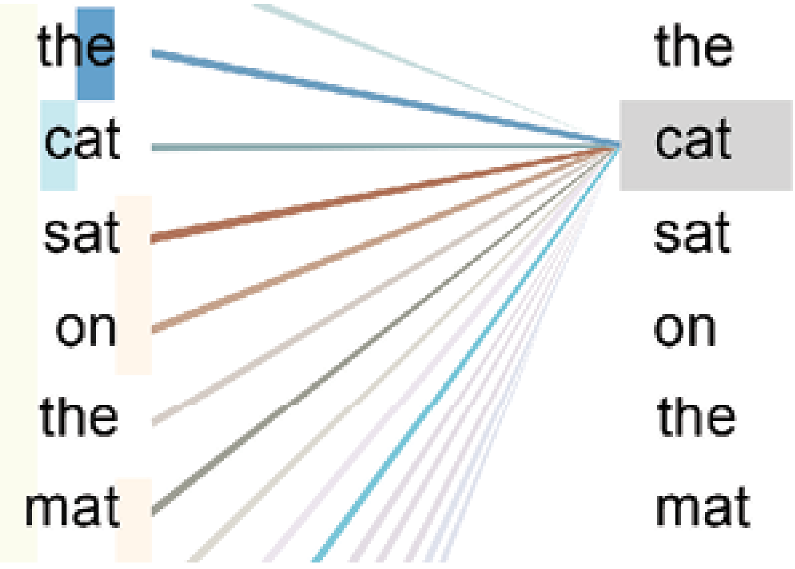


Figure 2.2: Attending to all the words

The attention mechanism will provide a deeper relationship between words and produce better results.For each attention sublayer, the original Transformer model runs not one but eight attention mechanisms in parallel to speed up the calculations. We will explore this architecture in the following section, *The encoder stack*. This process is named "multi-head attention," providing:

- A broader in-depth analysis of sequences
- The preclusion of recurrence reducing calculation operations
- Implementation of parallelization, which reduces training time
- Each attention mechanism learns different perspectives of the same input sequence

Attention replaced recurrence. However, several other creative aspects of the Transformer are as critical as the attention mechanism, as you will see when we look inside the architecture.We just looked at the Transformer structure from the outside. Let's now go into each component of the Transformer. We will start with the encoder.

The encoder stack

The layers of the encoder and decoder of the original Transformer model are *stacks of layers.* Each layer of the encoder stack has the following structure:

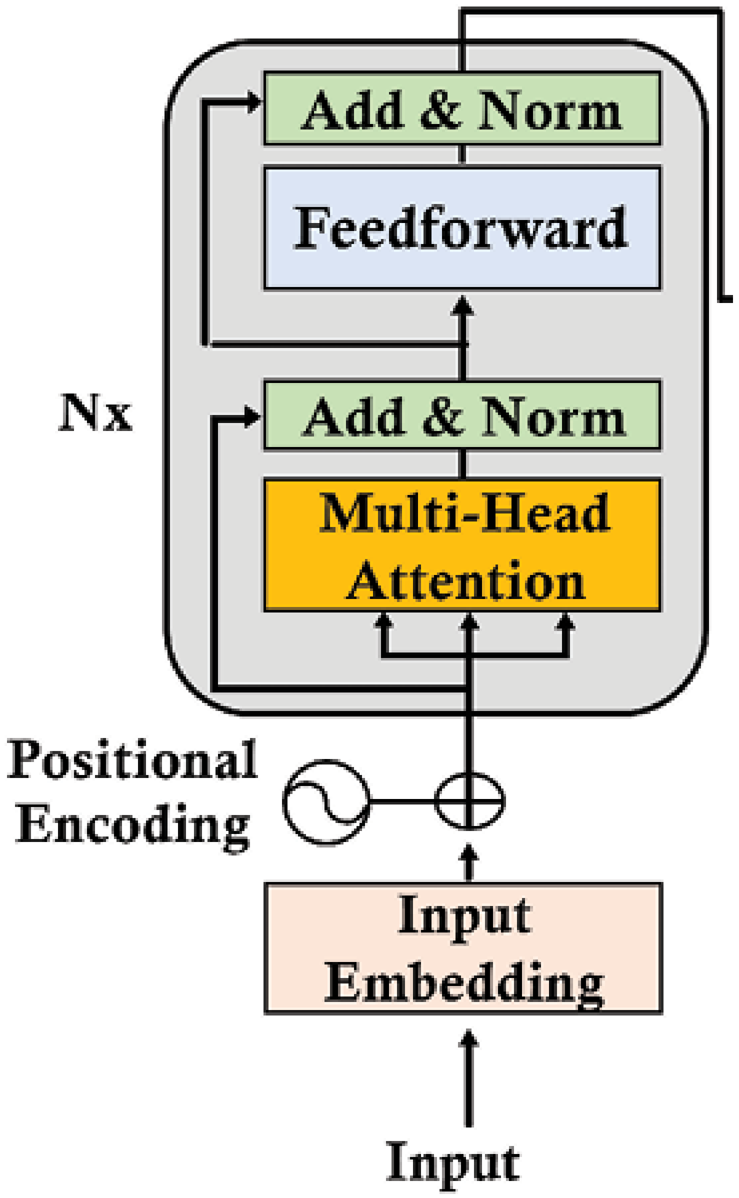


Figure 2.3: A layer of the encoder stack of the Transformer

The original encoder layer structure remains the same for all *N*=6 layers of the Transformer model. Each layer contains two main sublayers: a multi-headed attention mechanism and a fully connected position-wise feedforward network.Notice that a residual connection surrounds each main sublayer, *sublayer*(*x*), in the Transformer model. These connections transport the unprocessed input *x* of a sublayer to a layer normalization function. This way, we are certain that key information, such as positional encoding, is not lost on the way. The normalized output of each layer is thus:*LayerNormalization* (*x* + *Sublayer*(*x*))Though the structure of each of the *N*=6 layers of the encoder is identical, the content of each layer is not strictly identical to the previous layer through the different weights of each layer.For example, the embedding sublayer is only present at the bottom level of the stack. The other five layers do not contain an embedding layer, guaranteeing that the encoded input is stable through all the layers.Also, the multi-head attention mechanisms perform the same functions from layers 1 to 6. However, they do not perform the same tasks. Each layer learns from the previous layer and explores different ways of associating the tokens in the sequence. It looks for various associations of words, just like we look for different associations of letters and words when we solve a crossword puzzle.The designers of the Transformer introduced a very efficient constraint. The output of every sublayer of the model has a constant dimension, including the embedding layer and the residual connections. This dimension is *dmodel* and can be set to another value depending on your goals. In the original Transformer architecture, *dmodel* = 512.*dmodel* has a powerful consequence. Practically all the key operations are dot products. As a result, the dimensions remain stable, which reduces the number of operations to calculate, reduces machine consumption, and makes it easier to trace the information as it flows through the model.This global view of the encoder shows the highly optimized architecture of the Transformer. In the following sections, we will zoom into each of the sublayers and mechanisms.We will begin with the embedding sublayer.

Input embedding

The input embedding sublayer converts the input tokens to vectors of dimension *dmodel* = 512 using learned embeddings in the original Transformer model. The structure of the input embedding is classical:

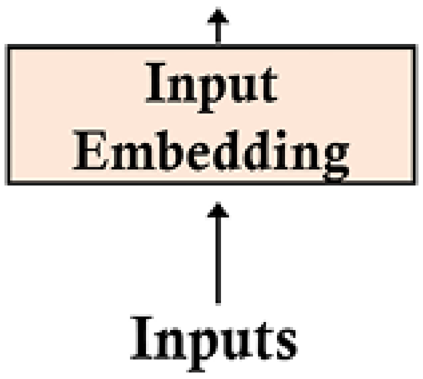


Figure 2.4: The input embedding sublayer of the Transformer

The embedding sublayer works like other standard transduction models. A tokenizer will transform a sentence into tokens. Each tokenizer has its methods, such as **Byte Pair Encoding** (**BPE**), word piece, and sentence piece methods. The Transformer initially used BPE, but other models use other methods.The goals are similar, and the choice depends on the strategy chosen. For example, a tokenizer applied to the sequence `the Transformer is an innovative NLP model!` will produce the following tokens in one type of model:

```
['the', 'transform', 'er', 'is', 'an', 'innovative', 'n', 'l', 'p', 'model', '!']
```

You will notice that this tokenizer normalized the string to lowercase and truncated it into subparts. A tokenizer will generally provide an integer representation that will be used for the embedding process. For example:

```
text = "The cat slept on the couch.It was too tired to get up."
tokenized text= [1996, 4937, 7771, 2006, 1996, 6411, 1012, 2009, 2001, 2205, 5458,
2000, 2131, 2039, 1012]
```

There is not enough information in the tokenized text at this point to go further. The tokenized text must be embedded.The Transformer contains a learned embedding sublayer. As a result, many embedding methods can be applied to the tokenized input.I chose the skip-gram architecture of the `word2vec` embedding approach Google made available in 2013 to illustrate the embedding sublayer of the Transformer. A skip-gram will focus on a center word in a window of words and predict *context* words. For example, if word(i) is the center word in a two-step window, a skip-gram model will analyze word(i-2), word(i-1), word(i+1), and word(i+2). Then, the window will *slide* and repeat the process. A skip-gram model generally contains an input layer, weights, a hidden layer, and an output containing the word embeddings of the tokenized input words.Suppose we need to perform embedding for the following sentence:

```
The black cat sat on the couch and the brown dog slept on the rug.
```

We will focus on two words, `black` and `brown`. The word embedding vectors of these two words should be similar.Since we must produce a vector of size *dmodel* = 512 for each word, we will obtain a size `512` vector embedding for each word:

```
black=[[-0.01206071  0.11632373  0.06206119  0.01403395  0.09541149  0.10695464
0.02560172  0.00185677 -0.04284821  0.06146432  0.09466285  0.04642421 0.08680347
0.05684567 -0.00717266 -0.03163519  0.03292002 -0.11397766 0.01304929  0.01964396
0.01902409  0.02831945  0.05870414  0.03390711 -0.06204525  0.06173197 -0.08613958
-0.04654748  0.02728105 -0.07830904
    …
0.04340003 -0.13192849 -0.00945092 -0.00835463 -0.06487109  0.05862355 -0.03407936
-0.00059001 -0.01640179  0.04123065
-0.04756588  0.08812257 0.00200338 -0.0931043  -0.03507337  0.02153351 -0.02621627
-0.02492662 -0.05771535 -0.01164199
-0.03879078 -0.05506947  0.01693138 -0.04124579 -0.03779858
-0.01950983 -0.05398201  0.07582296  0.00038318 -0.04639162
-0.06819214  0.01366171  0.01411388  0.00853774  0.02183574
-0.03016279 -0.03184025 -0.04273562]]
```

The word `black` is now represented by `512` dimensions. Other embedding methods could be used, and *dmodel* could have a higher number of dimensions.The word embedding of `brown` is also represented by `512` dimensions:

Copy Explain

```
brown=[[ 1.35794589e-02 -2.18823571e-02  1.34526128e-02  6.74355254e-02
   1.04376070e-01  1.09921647e-02 -5.46298288e-02 -1.18385479e-02
   4.41223830e-02 -1.84863899e-02 -6.84073642e-02  3.21860164e-02
   4.09143828e-02 -2.74433400e-02 -2.47369967e-02  7.74542615e-02
   9.80964210e-03  2.94299088e-02  2.93895267e-02 -3.29437815e-02
…
  7.20389187e-02  1.57317147e-02 -3.10291946e-02 -5.51304631e-02
  -7.03861639e-02  7.40829483e-02  1.04319192e-02 -2.01565702e-03
   2.43322570e-02  1.92969330e-02  2.57341694e-02 -1.13280728e-01
   8.45847875e-02  4.90090018e-03  5.33546880e-02 -2.31553353e-02
   3.87288055e-05  3.31782512e-02 -4.00604047e-02 -1.02028981e-01
   3.49597558e-02 -1.71501152e-02  3.55573371e-02 -1.77437533e-02
  -5.94457164e-02  2.21221056e-02  9.73121971e-02 -4.90022525e-02]]
```

To verify the word embedding produced for these two words, we can use cosine similarity to see if the word embeddings of the words `black` and `brown` are similar.Cosine similarity uses Euclidean (L2) norm to create vectors in a unit sphere. The dot product of the vectors we are comparing is the cosine between the points of those two vectors. For more on the theory of cosine similarity, you can consult scikit-learn's documentation, among many other sources: https://scikit-learn.org/stable/modules/metrics.html#cosine-similarity.The cosine similarity between the black vector of size *dmodel* = 512 and the brown vector of size *dmodel* = 512 in the embedding of the example is:

Copy Explain

```
cosine_similarity(black, brown)= [[0.9998901]]
```

The skip-gram produced two vectors that are close to each other. It detected that black and brown form a color subset of the dictionary of words.The Transformer's subsequent layers do not start empty-handed. Instead, they have learned word embeddings that already provide information on how the words can be associated.However, a big chunk of information is missing because no additional vector or information indicates a word's position in a sequence.The designers of the Transformer came up with yet another innovative feature: positional encoding.Let's see how positional encoding works.

Positional encoding

We enter this positional encoding function of the Transformer with no idea of the position of a word in a sequence:

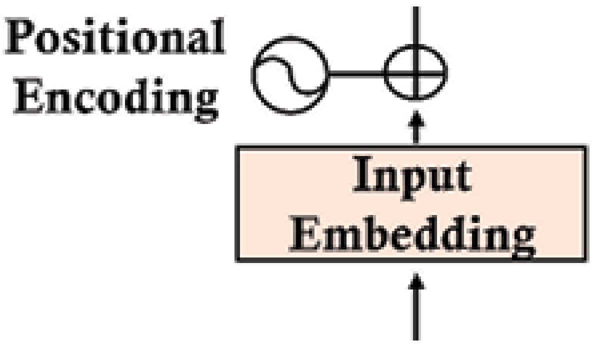


Figure 2.5: Positional encoding

We cannot create independent positional vectors that would have a high cost on the training speed of the Transformer and make attention sublayers overly complex to work with. The idea is to add a positional encoding value to the input embedding instead of having additional vectors to describe the position of a token in a sequence.The original Transformer model has only one vector that contains word embedding and position encoding. The Transformer expects a fixed size *dmodel* = 512 (or other constant value for the model) for each vector of the output of the positional encoding function.If we go back to the sentence we used in the word embedding sublayer, we can see that black and brown may be semantically similar, but they are far apart in the sentence:

Copy Explain

```
The black cat sat on the couch and the brown dog slept on the rug.
```

The word `black` is in position 2, `pos=2`, and the word `brown` is in position 10, `pos=10`.Our problem is to find a way to add a value to the word embedding of each word so that it has that information. However, we need to add a value to the *dmodel* = 512 dimensions! For each word embedding vector, we need to find a way to provide information to `i` in the `range(0,512)` dimensions of the word embedding vector of `black` and `brown`.*There are many ways to achieve positional encoding*. This section will focus on the designers' clever way to use a unit sphere to represent positional encoding with sine and cosine values that will thus remain small but useful.*Vaswani* et al. (2017) provide sine and cosine functions so that we can generate different frequencies for the **positional encoding** (PE) for each position and each dimension *i* of the *dmodel* = 512 of the word embedding vector:

$$PE_{(pos\ 2i)} = sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right) PE_{(pos\ 2i+1)} = cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

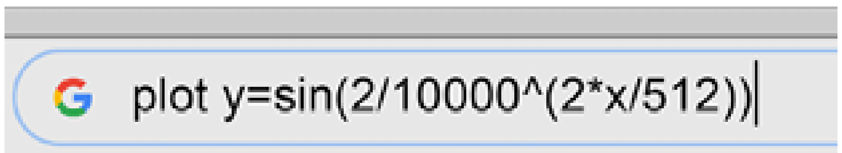
If we start at the beginning of the word embedding vector, we will begin with a constant (512), i=0, and end with i=511. This means that the sine function will be applied to the even numbers and the cosine function to the odd numbers. Some implementations do it differently. In that case, the domain of the sine function can be $i \in [0,255]$, and the domain of the cosine function can be $i \in [256,512]$. This will produce similar results.In this section, we will use the functions the way they were described by *Vaswani* et al. (2017). A literal translation into Python pseudo-code produces the following code for a positional vector pe[0][i] for a position pos:

```
def positional_encoding(pos,pe):
for i in range(0, 512,2):
         pe[0][i] = math.sin(pos / (10000 ** ((2 * i)/d_model)))
         pe[0][i+1] = math.cos(pos / (10000 ** ((2 * i)/d_model)))
return pe
```

Google Brain Trax and Hugging Face, among others, provide ready-to-use libraries for the word embedding section and the present positional encoding section. Thus, you don't need to run the code I shared in this section. However, if you wish to explore the code, you will find it in the positional_encoding.ipynb notebook in the directory of this chapter in the GitHub repository.Before going further, you might want to see the plot of the sine function, for example, for pos=2.You can Google the following plot, for example:

```
plot y=sin(2/10000^(2*x/512))
```

Just enter the plot request:



plot y=sin(2/10000^(2*x/512))

Figure 2.6: Plotting with Google

You will obtain the following graph:

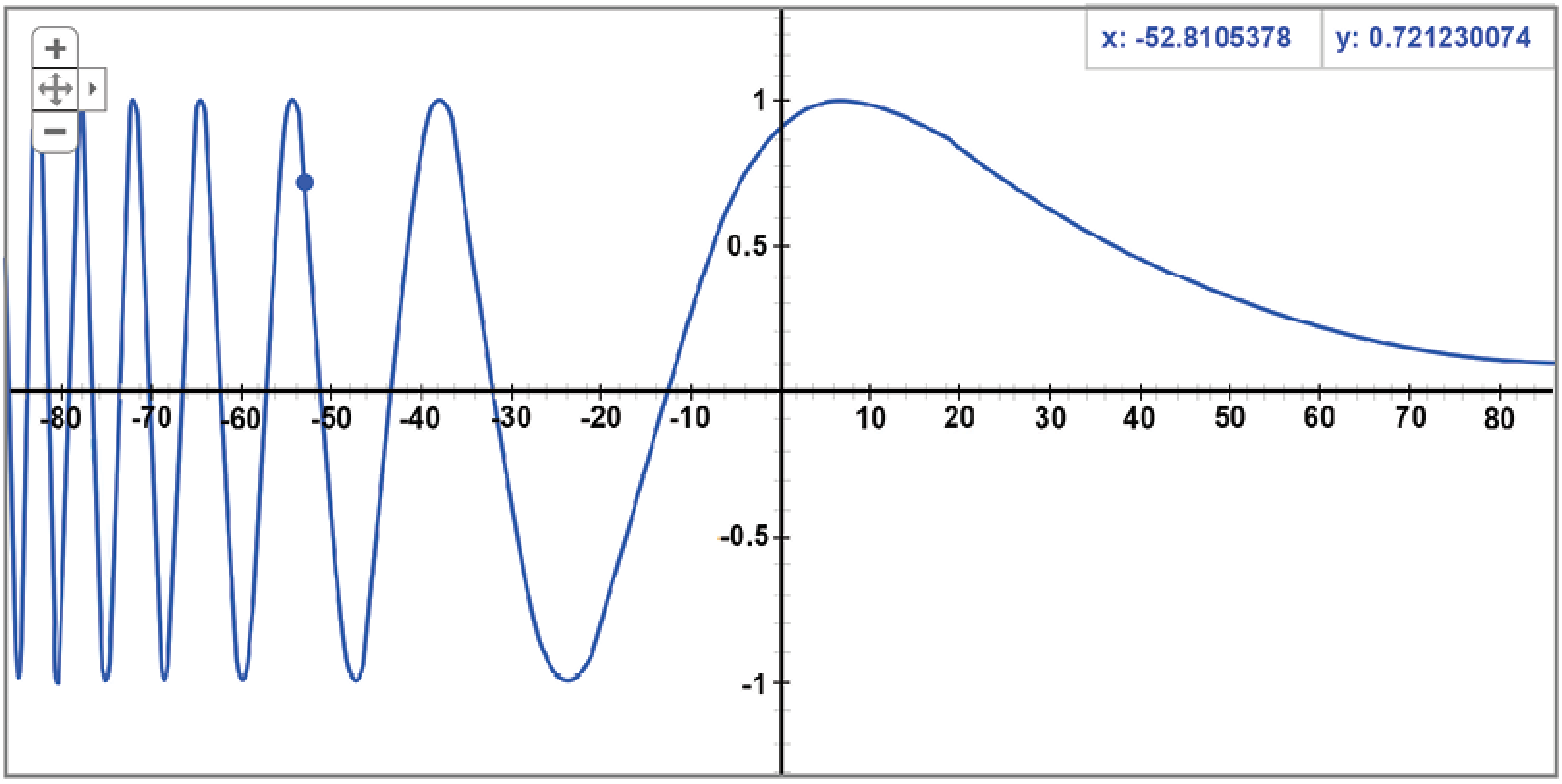


Figure 2.7: The graph

If we go back to the sentence we are parsing in this section, we can see that `black` is in position `pos=2` and `brown` is in position `pos=10`:

Copy Explain

```
The black cat sat on the couch and the brown dog slept on the rug.
```

If we apply the sine and cosine functions literally for `pos=2`, we obtain a size=`512` positional encoding vector:

Copy Explain

```
PE(2)=
[[ 9.09297407e-01 -4.16146845e-01  9.58144367e-01 -2.86285430e-01
   9.87046242e-01 -1.60435960e-01  9.99164224e-01 -4.08766568e-02
   9.97479975e-01  7.09482506e-02  9.84703004e-01  1.74241230e-01
   9.63226616e-01  2.68690288e-01  9.35118318e-01  3.54335666e-01
   9.02130723e-01  4.31462824e-01  8.65725577e-01  5.00518918e-01
   8.27103794e-01  5.62049210e-01  7.87237823e-01  6.16649508e-01
   7.46903539e-01  6.64932430e-01  7.06710517e-01  7.07502782e-01
…
   5.47683925e-08  1.00000000e+00  5.09659337e-08  1.00000000e+00
   4.74274735e-08  1.00000000e+00  4.41346799e-08  1.00000000e+00
   4.10704999e-08  1.00000000e+00  3.82190599e-08  1.00000000e+00
   3.55655878e-08  1.00000000e+00  3.30963417e-08  1.00000000e+00
   3.07985317e-08  1.00000000e+00  2.86602511e-08  1.00000000e+00
   2.66704294e-08  1.00000000e+00  2.48187551e-08  1.00000000e+00
   2.30956392e-08  1.00000000e+00  2.14921574e-08  1.00000000e+00]]
```

We also obtain a size=512 positional encoding vector for position 10, pos=10:

Copy Explain

```
PE(10)=
[[-5.44021130e-01 -8.39071512e-01  1.18776485e-01 -9.92920995e-01
   6.92634165e-01 -7.21289039e-01  9.79174793e-01 -2.03019097e-01
   9.37632740e-01  3.47627431e-01  6.40478015e-01  7.67976522e-01
   2.09077001e-01  9.77899194e-01 -2.37917677e-01  9.71285343e-01
  -6.12936735e-01  7.90131986e-01 -8.67519796e-01  4.97402608e-01
  -9.87655997e-01  1.56638563e-01 -9.83699203e-01 -1.79821849e-01
…
  2.73841977e-07  1.00000000e+00  2.54829672e-07  1.00000000e+00
   2.37137371e-07  1.00000000e+00  2.20673414e-07  1.00000000e+00
   2.05352507e-07  1.00000000e+00  1.91095296e-07  1.00000000e+00
   1.77827943e-07  1.00000000e+00  1.65481708e-07  1.00000000e+00
   1.53992659e-07  1.00000000e+00  1.43301250e-07  1.00000000e+00
   1.33352145e-07  1.00000000e+00  1.24093773e-07  1.00000000e+00
   1.15478201e-07  1.00000000e+00  1.07460785e-07  1.00000000e+00]]
```

When we look at the results obtained with an intuitive literal translation of the Vaswani et al. (2017) functions into Python, we want to check whether the results are meaningful.The cosine similarity function used for word embedding comes in handy for having a better visualization of the proximity of the positions:

Copy Explain

```
cosine_similarity(pos(2), pos(10))= [[0.8600013]]
```

The similarity between the position of the words black and brown and the lexical field (groups of words that go together) similarity is different:

Copy Explain

```
cosine_similarity(black, brown)= [[0.9998901]]
```

The encoding of the position shows a lower similarity value than the word embedding similarity.The positional encoding has taken these words apart. Bear in mind that word embeddings will vary with the corpus used to train them. The problem is now how to add the positional encoding to the word embedding vectors.

Adding positional encoding to the embedding vector

The authors of the Transformer found a simple way by merely adding the positional encoding vector to the word embedding vector:

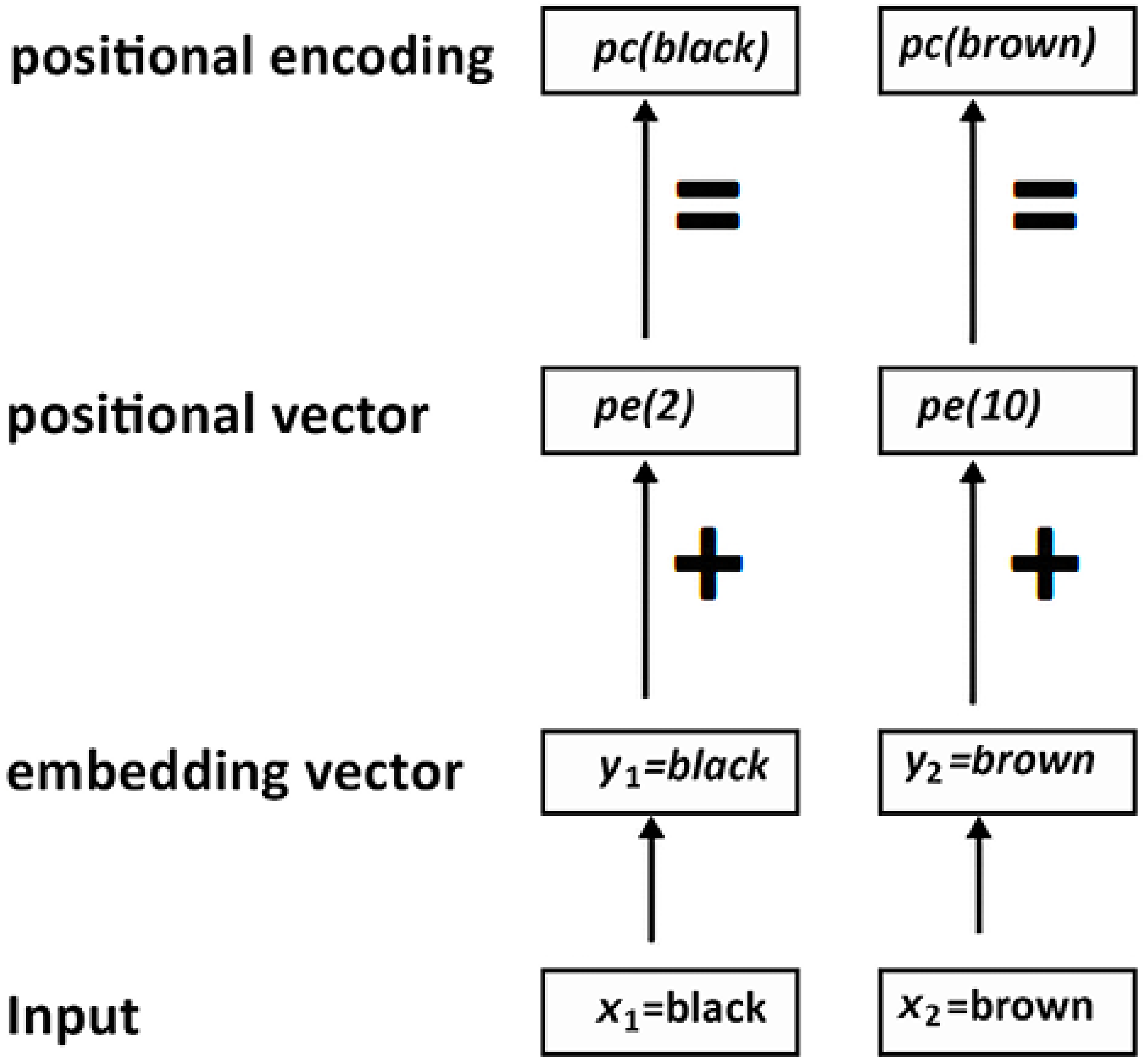


Figure 2.8: Positional encoding

If we go back and take the word embedding of `black`, for example, and name it, we are ready to add it to the positional vector we obtained with positional encoding functions. We will obtain the positional encoding of the input word `black`:In the following example, we add the positional vector to the embedding vector of the word `black`, which are both of the same size (`512`):

```
for i in range(0, 512,2):
          pe[0][i] = math.sin(pos / (10000 ** ((2 * i)/d_model)))
          pc[0][i] = (y[0][i]*math.sqrt(d_model))+ pe[0][i]

          pe[0][i+1] = math.cos(pos / (10000 ** ((2 * i)/d_model)))
          pc[0][i+1] = (y[0][i+1]*math.sqrt(d_model))+ pe[0][i+1]
```

The result obtained is the final positional encoding vector of dimension *dmodel* = 512:

```
pc(black)=
[[ 9.09297407e-01 -4.16146845e-01  9.58144367e-01 -2.86285430e-01
   9.87046242e-01 -1.60435960e-01  9.99164224e-01 -4.08766568e-02
   …
  4.74274735e-08  1.00000000e+00  4.41346799e-08  1.00000000e+00
   4.10704999e-08  1.00000000e+00  3.82190599e-08  1.00000000e+00
   2.66704294e-08  1.00000000e+00  2.48187551e-08  1.00000000e+00
   2.30956392e-08  1.00000000e+00  2.14921574e-08  1.00000000e+00]]
```

The same operation is applied to the word `brown` and all of the other words in a sequence.We can apply the cosine similarity function to the positional encoding vectors of `black` and `brown`:

Copy Explain

```
cosine_similarity(pc(black), pc(brown))= [[0.9627094]]
```

We now have a clear view of the positional encoding process through the three cosine similarity functions we applied to the three states representing the words `black` and `brown`:

Copy Explain

```
[[0.99987495]] word similarity
[[0.8600013]] positional encoding vector similarity
[[0.9627094]] final positional encoding similarity
```

We saw that the initial word similarity of their embeddings was high, with a value of `0.99`. Then, we saw that the positional encoding vector of positions 2 and 10 drew these two words apart with a lower similarity value of `0.86`.Finally, we added the word embedding vector of each word to its respective positional encoding vector. We saw that this brought the cosine similarity of the two words to `0.96`.The positional encoding of each word now contains the initial word embedding information and the positional encoding values.The output of positional encoding leads to the multi-head attention sublayer.

Sublayer 1: Multi-head attention

The multi-head attention sublayer contains eight heads and is followed by post-layer normalization, which will add residual connections to the output of the sublayer and normalize it:

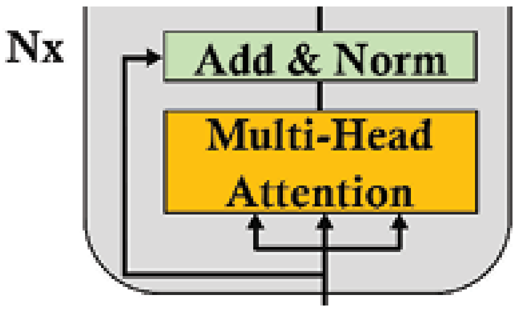


Figure 2.9: Multi-head attention sublayer

This section begins with the architecture of an attention layer. Then, an example of multi-attention is implemented in a small module in Python. Finally, post-layer normalization is described.Let's start with the architecture of multi-head attention.

The architecture of multi-head attention

The input of the multi-attention sublayer of the first layer of the encoder stack is a vector that contains the embedding and the positional encoding of each word. The next layers of the stack do not start these operations over.The dimension of the vector of each word xn of an input sequence is $dmodel = 512$:$pe(xn)$=[$d1$=9.09297407e-01, $d2$=-4.16146845e-01, .., $d512$=1.00000000e+00]The representation of each word xn has become a vector of $dmodel = 512$ dimensions.Each word is mapped to all the other words to determine how it fits in a sequence.In the following sentence, we can see that it could be related to `cat` and `rug` in the sequence:

Copy Explain

```
Sequence =The cat sat on the rug and it was dry-cleaned.
```

The model will train to determine if `it` is related to `cat` or `rug`. We could run a huge calculation by training the model using the $dmodel = 512$ dimensions as they are now.However, we would only get one point of view at a time by analyzing the sequence with one $dmodel$ block. Furthermore, it would take quite some calculation time to find other perspectives.A better way is for the 8 heads of the model to project the $dmodel = 512$ dimensions of each word xn in sequence x into dk=64 dimensions.We then can run the 8 "heads" in parallel to speed up the training and obtain 8 different representation subspaces of how each word relates to another:

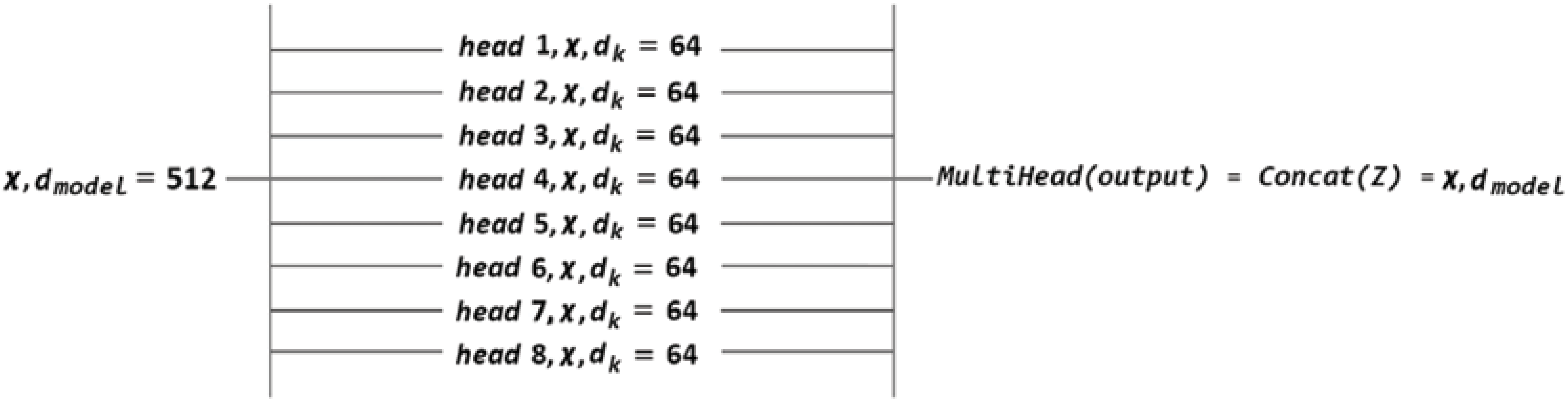


Figure 2.10: Multi-head representations

You can see that there are now `8` heads running in parallel. For example, one head might decide that `it` fits well with `cat` , another that it fits well with rug , and another that `rug` fits well with `dry-cleaned`.The output of each head is a matrix Zi with a shape of $x * dk$. The output of a multi-attention head is Z defined as:$Z = (Z0, Z1, Z2, Z3, Z4, Z5, Z6, Z7)$However, Z must be concatenated so that the output of the multi-head sublayer is not a sequence of dimensions but one line of an $xm * dmodel$ matrix.Before exiting the multi-head attention sublayer, the elements of Z are concatenated:$MultiHead(output) = Concat(Z0, Z1, Z2, Z3, Z4, Z5, Z6, Z7) = x,$

*dmodel*Notice that each head is concatenated into *z*, which has a dimension of *dmodel* = 512. The output of the multi-headed layer respects the constraint of the original Transformer model.Inside each head *hn* of the attention mechanism, the "word" matrices have three representations:A query matrix (*Q*) that has a dimension of *dq* =64, which seeks all the key-value pairs of the other "word" matrices. A key matrix (*K*) with a dimension of *dk* =64. A value matrix (*V*) with a dimension of *dv* =64. Attention is defined as "Scaled Dot-Product Attention," which is represented in the following equation in which we plug *Q*, *K*, and *V*:

$$Attention(\boldsymbol{Q},\boldsymbol{K},\boldsymbol{V}) = softmax\left(\frac{\boldsymbol{QK}^T}{\sqrt{\boldsymbol{d}_k}}\right)V$$

The matrices all have the same dimension, making it relatively simple to use a scaled dot product to obtain the attention values for each head and then concatenate the output *Z* of the 8 heads.To obtain *Q*, *K*, and *V*, we must train the model with their respective weight matrices *Qw*, *Kw*, and *Vw*, which have *dk* =64 columns and *dmodel* = 512 rows. For example, *Q* is obtained by a dot-product between *x* and *Qw*. *Q* will have a dimension of *dk* =64.You can modify all the parameters, such as the number of layers, heads, *dmodel*, *dk*, and other variables of the Transformer to fit your model. This chapter describes the original Transformer parameters by *Vaswani* et al. (2017). It is essential to understand the original architecture before modifying it or exploring variants of the original model designed by others.Google Brain Trax, OpenAI, Google Cloud AI, and Hugging Face, among others, provide ready-to-use libraries that we will be using throughout this book.However, let's open the hood of the Transformer model and get our hands dirty in Python to illustrate the architecture we just explored to visualize the model in code and show it with intermediate images.We will use basic Python code with only `numpy` and a `softmax` function in 10 steps to run the main aspects of the attention mechanism.Remember that an AI Specialist will face the challenge of multiple architectures for the same algorithm.Let's now start building *Step 1* of our model to represent the input.

Step 1: Represent the input

Save `Multi_Head_Attention_Sub_Layer.ipynb` to your Google Drive (make sure you have a Gmail account) and then open it in Google Colaboratory. The notebook is in the GitHub repository for this chapter.We will start by only using minimal Python functions to understand the Transformer at a low level with the inner workings of an attention head. Then, we will explore the inner workings of the multi-head attention sublayer using basic code:

Copy Explain

```
import numpy as np
from scipy.special import softmax
```

The input of the attention mechanism we are building is scaled down to *dmodel* ==4 instead of *dmodel* = 512. This brings the dimensions of the vector of an input *x* down to *dmodel* =4, which is easier to visualize.*x* contains 3 inputs with 4 dimensions each instead of 512:

Copy Explain

```
print("Step 1: Input : 3 inputs, d_model=4")
x =np.array([[1.0, 0.0, 1.0, 0.0],    # Input 1
             [0.0, 2.0, 0.0, 2.0],    # Input 2
             [1.0, 1.0, 1.0, 1.0]])  # Input 3
print(x)
```

The output shows that we have 3 vectors of *dmodel* =4:

Copy Explain

```
Step 1: Input : 3 inputs, d_model=4
[[1. 0. 1. 0.]
 [0. 2. 0. 2.]
 [1. 1. 1. 1.]]
```

The first step of our model is ready:

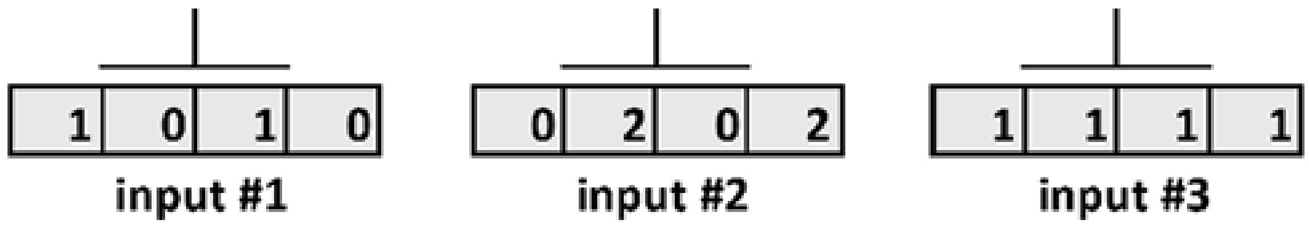


Figure 2.11: Input of a multi-head attention sublayer

We will now add the weight matrices to our model.

Step 2: Initializing the weight matrices

Each input has 3 weight matrices:*Qw* to train the queries*Kw* to train the keys*Vw* to train the valuesThese 3 weight matrices will be applied to all the inputs in this model.The weight matrices described by *Vaswani* et al. (2017) are *dK* ==64 dimensions. However, let's scale the matrices down to *dK* ==3. The dimensions are scaled down to 3*4 weight matrices to visualize the intermediate results more easily and perform dot products with the input *x*.The size and shape of the matrices in this educational notebook are arbitrary. The goal is to go through the overall process of an attention mechanism.The three weight matrices are initialized starting with the query weight matrix:

```
print("Step 2: weights 3 dimensions x d_model=4")
print("w_query")
w_query =np.array([[1, 0, 1],
                   [1, 0, 0],
                   [0, 0, 1],
                   [0, 1, 1]])
print(w_query)
```

The output is the `w_query` weight matrix:

```
w_query
[[1 0 1]
 [1 0 0]
 [0 0 1]
 [0 1 1]]
```

We will now initialize the key weight matrix:

```
print("w_key")
w_key =np.array([[0, 0, 1],
                 [1, 1, 0],
                 [0, 1, 0],
                 [1, 1, 0]])
print(w_key)
```

The output is the key weight matrix:

```
w_key
[[0 0 1]
 [1 1 0]
 [0 1 0]
 [1 1 0]]
```

Finally, we initialize the value weight matrix:

```
print("w_value")
w_value = np.array([[0, 2, 0],
                    [0, 3, 0],
                    [1, 0, 3],
                    [1, 1, 0]])
print(w_value)
```

The output is the value weight matrix:

```
w_value
[[0 2 0]
 [0 3 0]
 [1 0 3]
 [1 1 0]]
```

The second step of our model is ready:

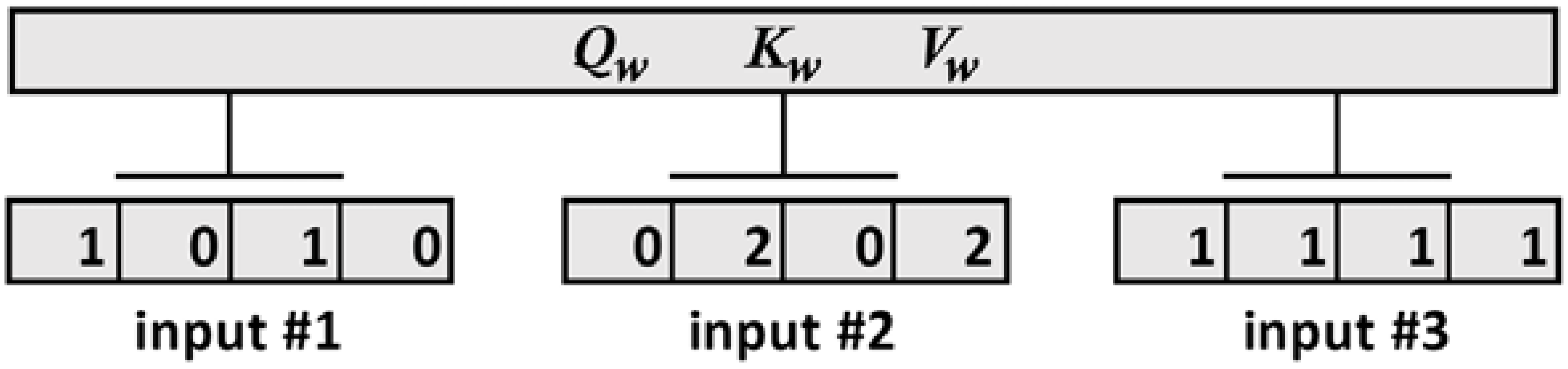


Figure 2.12: Weight matrices added to the model

We will now multiply the weights by the input vectors to obtain *Q*, *K*, and *V*.

Step 3: Matrix multiplication to obtain Q, K, and V

We will now multiply the input vectors by the weight matrices to obtain a query, key, and value vector for each input.In this model, we will assume that there is one `w_query`, `w_key`, and `w_value` weight matrix for all inputs. Other approaches are possible.Let's first multiply the input vectors by the `w_query` weight matrix:

```
print("Step 3: Matrix multiplication to obtain Q,K,V")
print("Query: x * w_query")
Q=np.matmul(x,w_query)
print(Q)
```

The output is a vector for *Q1* ==64= [1, 0, 2], *Q2*= [2,2, 2], and *Q3*= [2,1, 3]:

```
Step 3: Matrix multiplication to obtain Q,K,V
Query: x * w_query
[[1. 0. 2.]
 [2. 2. 2.]
 [2. 1. 3.]]
```

We now multiply the input vectors by the `w_key` weight matrix:

```
print("Key: x * w_key")
K=np.matmul(x,w_key)
print(K)
```

We obtain a vector for *K1*= [0, 1, 1], *K2*= [4, 4, 0], and *K3*= [2 ,3, 1]:

```
Key: x * w_key
[[0. 1. 1.]
 [4. 4. 0.]
 [2. 3. 1.]]
```

Finally, we multiply the input vectors by the `w_value` weight matrix:

```
print("Value: x * w_value")
V=np.matmul(x,w_value)
print(V)
```

We obtain a vector for *V1*= [1, 2, 3], *V2*= [2, 8, 0], and *V3*= [2 ,6, 3]:

```
Value: x * w_value
[[1. 2. 3.]
 [2. 8. 0.]
 [2. 6. 3.]]
```

The third step of our model is ready:

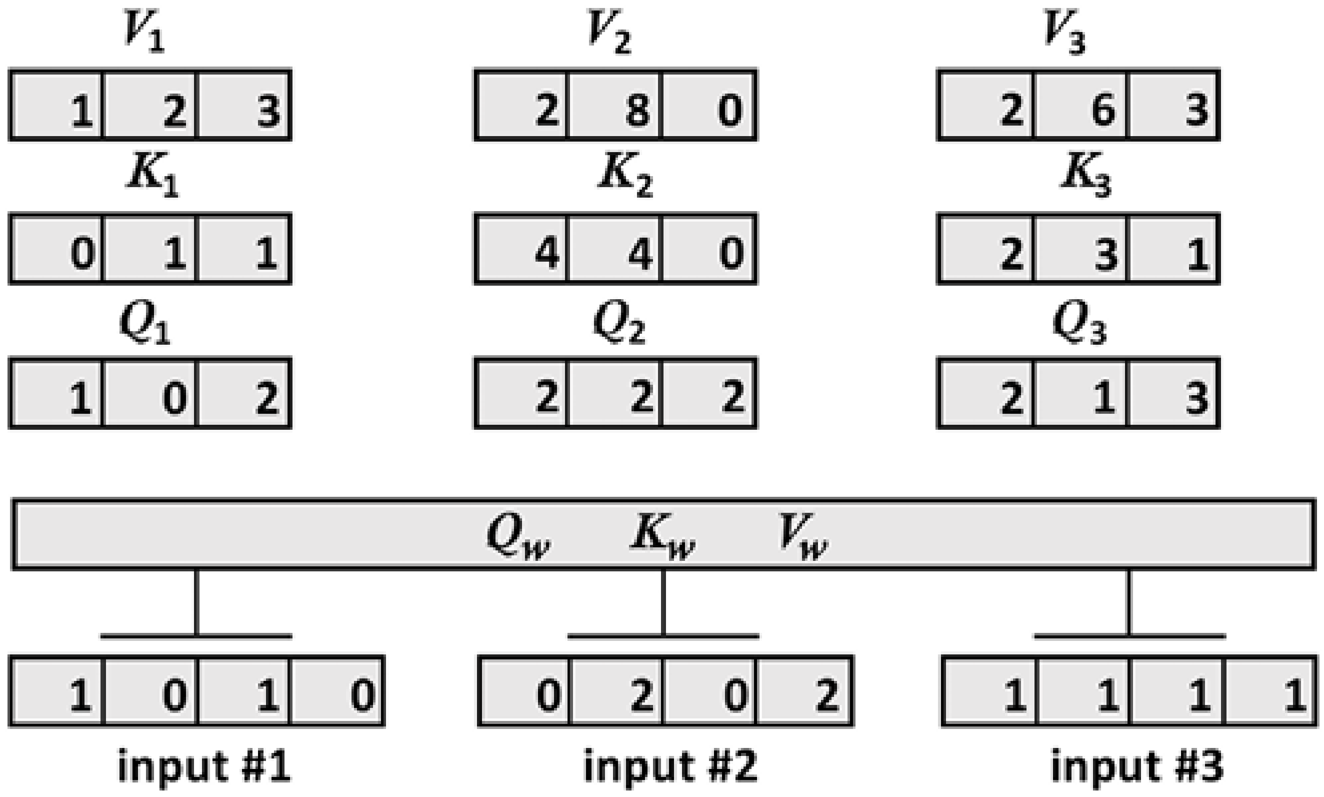


Figure 2.13: Q, K, and V are generated

We have the *Q*, *K*, and *V* values we need to calculate the attention scores.

Step 4: Scaled attention scores

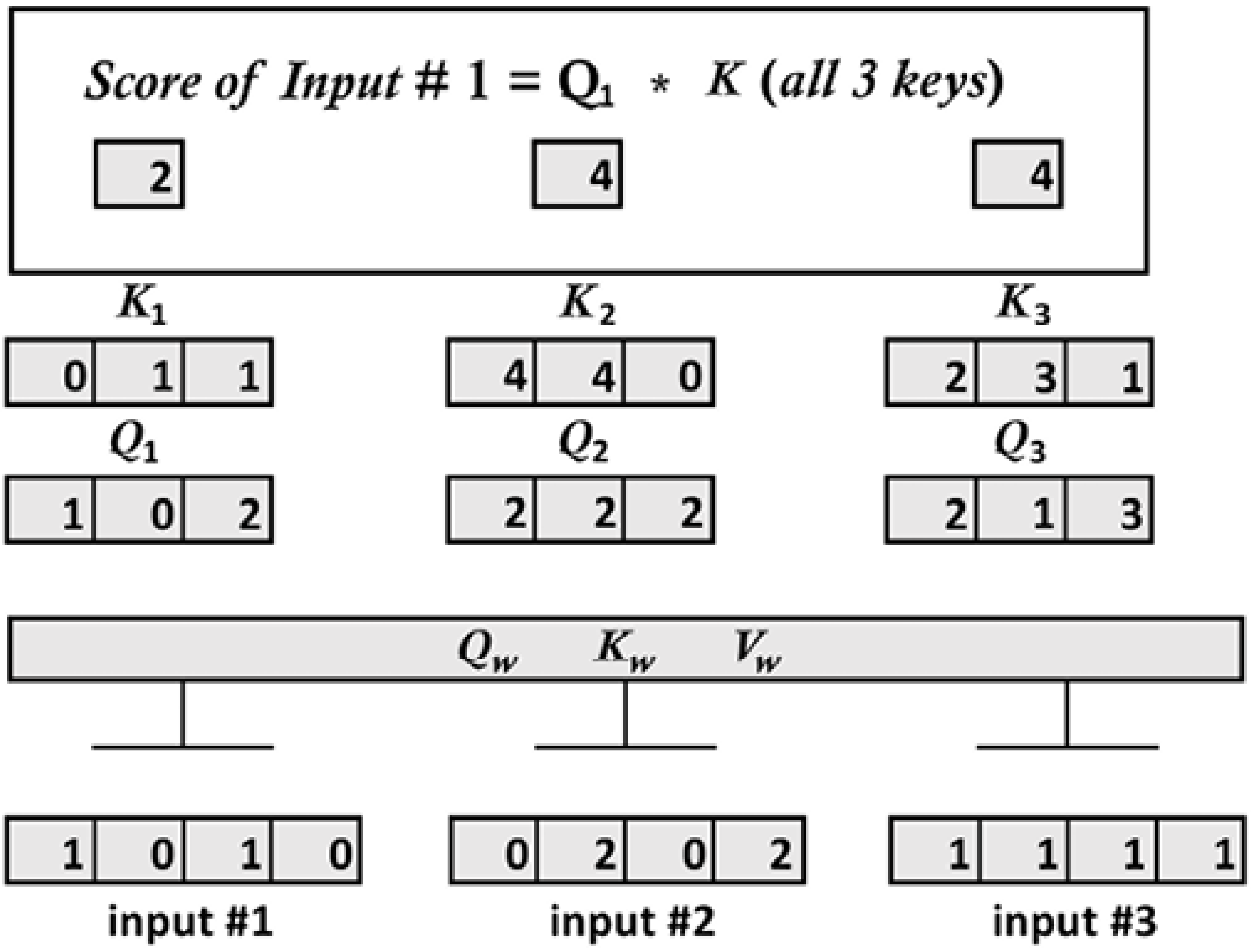
The attention head now implements the original Transformer equation:

$$\left(\frac{QK^T}{\sqrt{d_k}}\right)$$

Step 4 focuses on *Q* and *K*:

$$\sqrt{d_k}$$

For this model, we will round $\sqrt{3}$ = = 1.75 to 1 and plug the values into the *Q* and *K* part of the equation:



Copy Explain

```
print("Step 4: Scaled Attention Scores")
k_d=1   #square root of k_d=3 rounded down to 1 for this example
attention_scores = (Q @ K.transpose())/k_d
print(attention_scores)
```

The intermediate result is displayed:

Copy Explain

```
Step 4: Scaled Attention Scores
[[ 2.   4.   4.]
 [ 4. 16. 12.]
 [ 4. 12. 10.]]
```

Step 4 is now complete. For example, the score for *x*1 is [2,4,4] across the *K* vectors across the head as displayed:

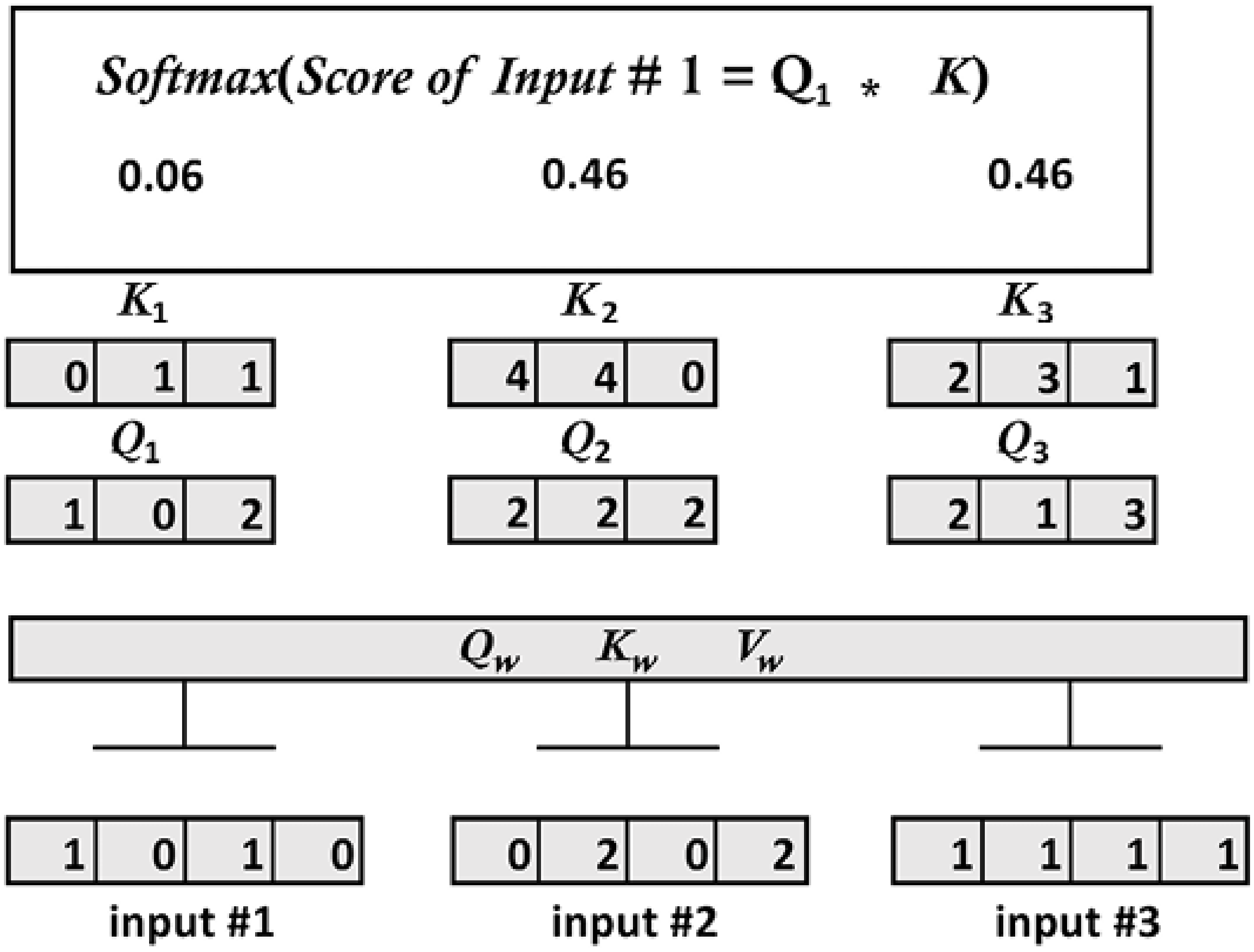


Figure 2.14: Scaled attention scores for input #1

The attention equation will now apply softmax to the intermediate scores for each vector.

Step 5: Scaled softmax attention scores for each vector

We now apply a softmax function to each intermediate attention score. Instead of doing a matrix multiplication, let's zoom down to each individual vector:

```
print("Step 5: Scaled softmax attention_scores for each vector")
attention_scores[0]=softmax(attention_scores[0])
attention_scores[1]=softmax(attention_scores[1])
attention_scores[2]=softmax(attention_scores[2])
print(attention_scores[0])
print(attention_scores[1])
print(attention_scores[2])
```

We obtain scaled softmax attention scores for each vector:

```
Step 5: Scaled softmax attention_scores for each vector
[0.06337894 0.46831053 0.46831053]
[6.03366485e-06 9.82007865e-01 1.79861014e-02]
[2.95387223e-04 8.80536902e-01 1.19167711e-01]
```

Step 5 is now complete. For example, the softmax of the score of x_1 for all the keys is:

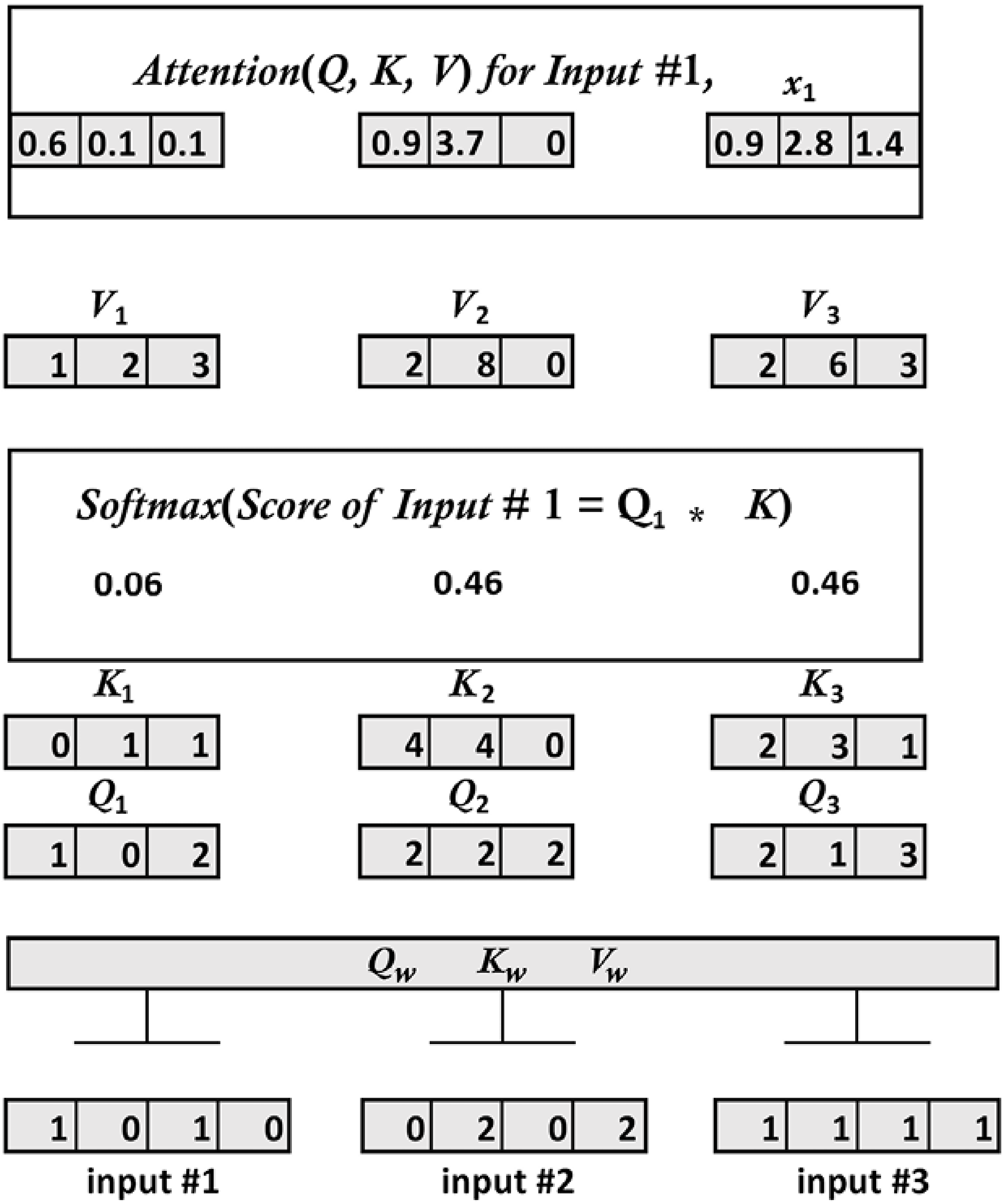
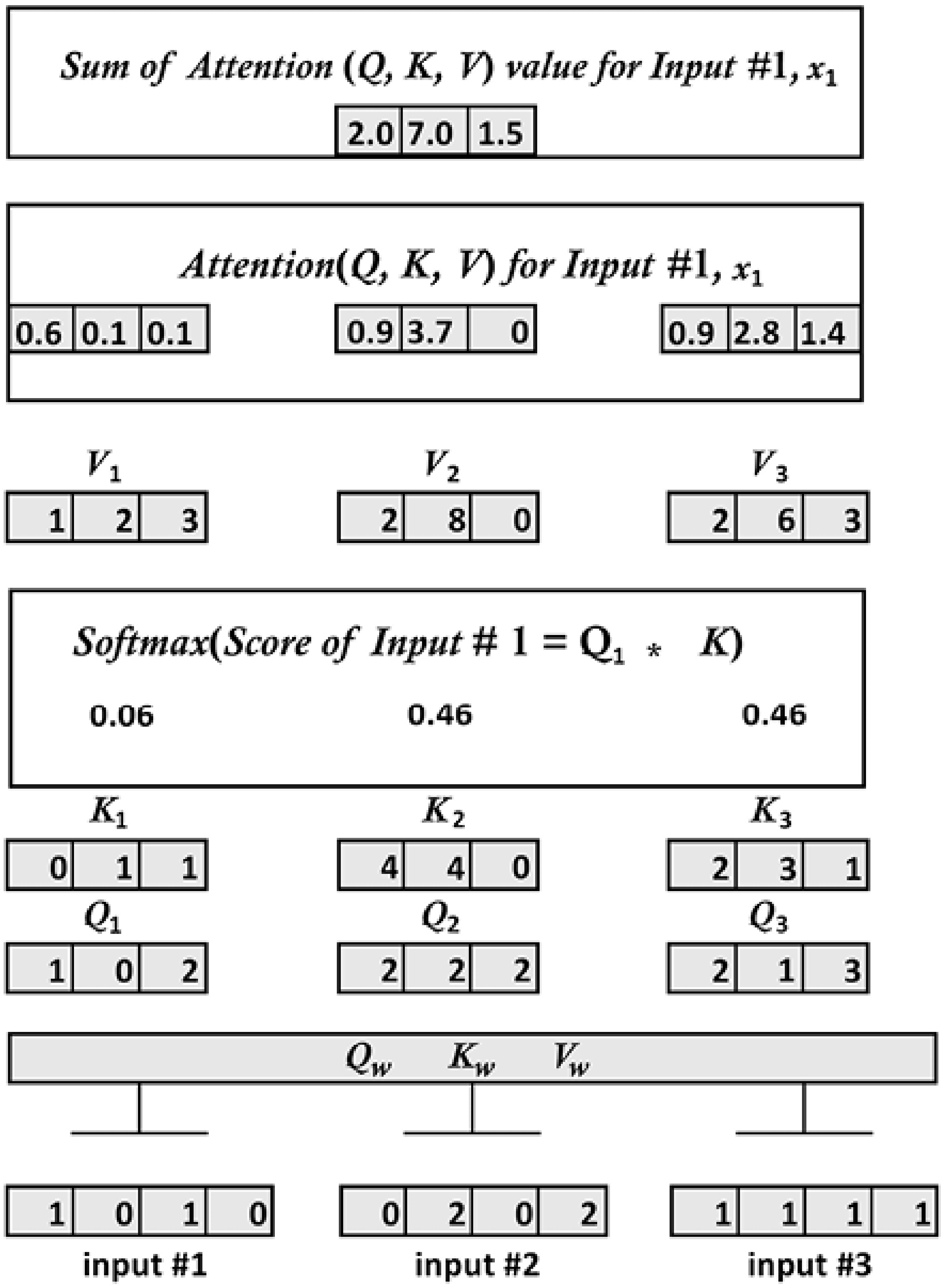


Figure 2.15: The softmax score of input #1 for all of the keys

We can now calculate the final attention values with the complete equation.

Step 6: The final attention representations

We can now finalize the attention equation by plugging *V* in:



We will first calculate the attention score of input *x*1 for *Steps 6* and *7*. We calculate one attention value for one word vector. When we reach *Step 8*, we will generalize the attention calculation to the other two input vectors.To obtain *Attention*(*Q,K,V*) for x_1 we multiply the intermediate attention score by the 3 value vectors one by one to zoom down into the inner workings of the equation:

Copy Explain

```
print("Step 6: attention value obtained by score1/k_d * V")
print(V[0])
print(V[1])
print(V[2])
print("Attention 1")
attention1=attention_scores[0].reshape(-1,1)
attention1=attention_scores[0][0]*V[0]
print(attention1)
print("Attention 2")
attention2=attention_scores[0][1]*V[1]
print(attention2)
print("Attention 3")
attention3=attention_scores[0][2]*V[2]
print(attention3)
Step 6: attention value obtained by score1/k_d * V
[1. 2. 3.]
[2. 8. 0.]
[2. 6. 3.]
Attention 1
[0.06337894 0.12675788 0.19013681]
Attention 2
[0.93662106 3.74648425 0.        ]
Attention 3
[0.93662106 2.80986319 1.40493159]
```

Step 6 is complete. For example, the 3 attention values for *x*1 for each input have been calculated:

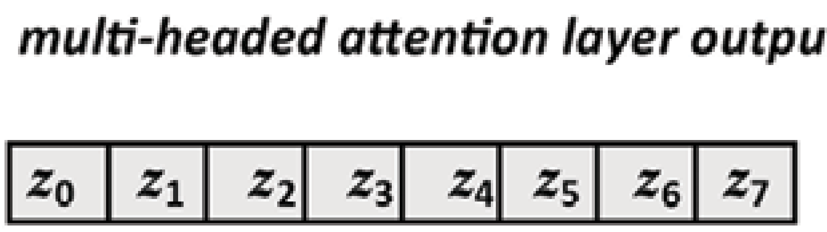


Figure 2.16: Attention representations

The attention values now need to be summed up.

Step 7: Summing up the results

The 3 attention values of input #1 obtained will now be summed to obtain the first line of the output matrix:

Copy Explain

```
print("Step 7: summed the results to create the first line of the output matrix")
attention_input1=attention1+attention2+attention3
print(attention_input1)
```

The output is the first line of the output matrix for input #1:

```
Step 7: summed the results to create the first line of the output matrix
[1.93662106 6.68310531 1.59506841]]
```

The second line will be for the output of the next input, input #2, for example.We can see the summed attention value for *x*1 in *Figure 2.17*:

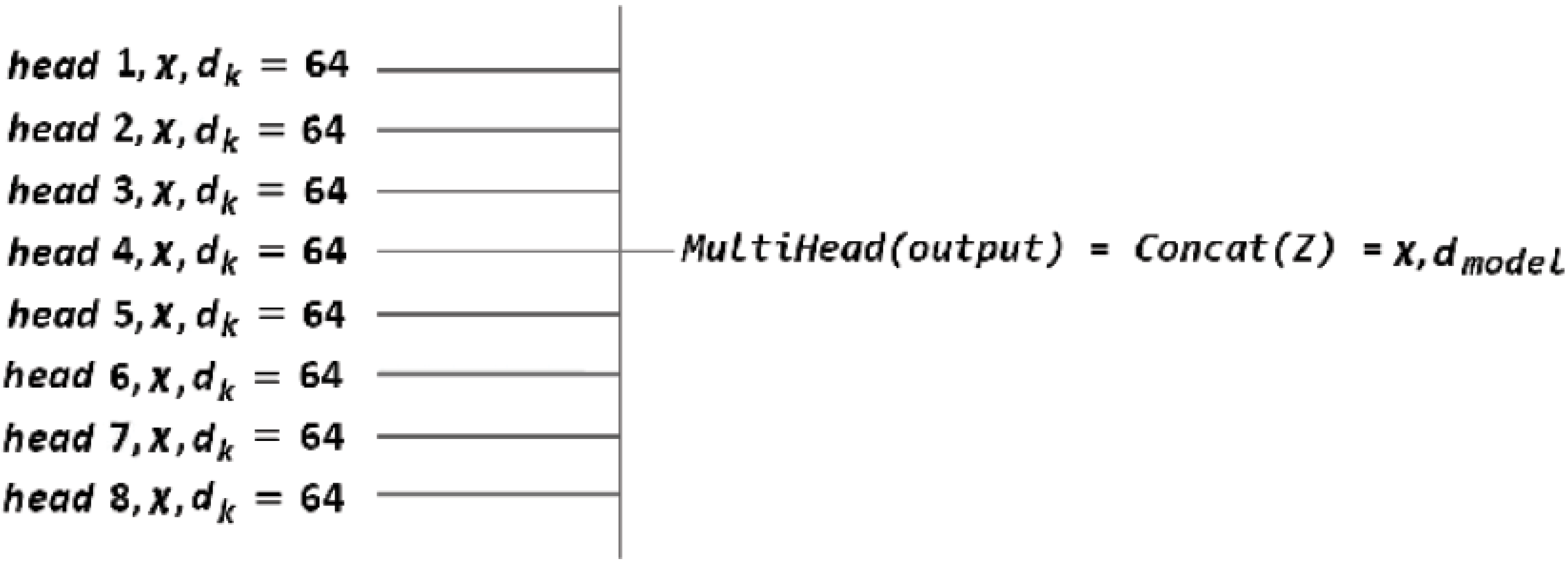


Figure 2.17: Summed results for one input

We have completed the steps for input #1. We now need to add the results of all the inputs to the model.

Step 8: Steps 1 to 7 for all the inputs

The Transformer can now produce the attention values of input #2 and input #3 using the same method described from *Steps 1* to *Step 7* for one attention head.From this step onwards, we will assume we have 3 attention values with learned weights with *dmodel* = 64. We now want to see what the original dimensions look like when they reach the sublayer's output.We have seen the attention representation process in detail with a small model. Let's go directly to the result and assume we have generated the 3 attention representations with a dimension of *dmodel* = 64:

```
print("Step 8: Step 1 to 7 for inputs 1 to 3")
#We assume we have 3 results with learned weights (they were not trained in this
example)
#We assume we are implementing the original Transformer paper.We will have 3 results
of 64 dimensions each
attention_head1=np.random.random((3, 64))
print(attention_head1)
```

The following output displays the simulation of *z*0, which represents the 3 output vectors of *dmodel* = 64 dimensions for head 1:

Copy Explain

```
Step 8: Step 1 to 7 for inputs 1 to 3
[[0.31982626 0.99175996…(61 squeezed values)…0.16233212]
 [0.99584327 0.55528662…(61 squeezed values)…0.70160307]
 [0.14811583 0.50875291…(61 squeezed values)…0.83141355]]
```

The results will vary when you run the notebook because of the stochastic nature of the generation of the vectors.The Transformer now has the output vectors for the inputs of one head. The next step is to generate the output of the 8 heads to create the final output of the attention sublayer.

Step 9: The output of the heads of the attention sublayer

We assume that we have trained the 8 heads of the attention sublayer. The Transformer now has 3 output vectors (of the 3 input vectors that are words or word pieces) of *dmodel* = 64 dimensions each:

Copy Explain

```
print("Step 9: We assume we have trained the 8 heads of the attention sublayer")
z0h1=np.random.random((3, 64))
z1h2=np.random.random((3, 64))
z2h3=np.random.random((3, 64))
z3h4=np.random.random((3, 64))
z4h5=np.random.random((3, 64))
z5h6=np.random.random((3, 64))
z6h7=np.random.random((3, 64))
z7h8=np.random.random((3, 64))
print("shape of one head",z0h1.shape,"dimension of 8 heads",64*8)
```

The output shows the shape of one of the heads:

Copy Explain

```
Step 9: We assume we have trained the 8 heads of the attention sublayer
shape of one head (3, 64) dimension of 8 heads 512
```

The 8 heads have now produced *Z*:*Z* = (*Z0*, *Z1*, *Z2*, *Z3*, *Z4*, *Z5*, *Z6*, *Z7*)The Transformer will now concatenate the 8 elements of *Z* for the final output of the multi-head attention sublayer.

Step 10: Concatenation of the output of the heads

The Transformer concatenates the 8 elements of *Z*:*MultiHead*(*Output*) = *Concat* (*Z0*, *Z1*, *Z2*, *Z3*, *Z4*, *Z5*, *Z6*, *Z7*) *W0* = *x*, *dmodel*Note that *Z* is multiplied by *W0*, a weight matrix that is also trained. In this model, we will assume *W0* is trained and integrated into the concatenation function. *Z0* to *Z7* are concantenated:

Copy Explain

```
print("Step 10: Concantenation of heads 1 to 8 to obtain the original 8x64=512 ouput
dimension of the model")
output_attention=np.hstack((z0h1,z1h2,z2h3,z3h4,z4h5,z5h6,z6h7,z7h8))
print(output_attention)
```

The output is the concatenation of *Z*:

Copy Explain

```
Step 10: Concatenation of heads 1 to 8 to obtain the original 8x64=512 output
dimension of the model
[[0.65218495 0.11961095 0.9555153  ... 0.48399266 0.80186221 0.16486792]
 [0.95510952 0.29918492 0.7010377  ... 0.20682832 0.4123836  0.90879359]
 [0.20211378 0.86541746 0.01557758 ... 0.69449636 0.02458972 0.889699  ]]
```

The concatenation can be visualized as stacking the elements of *Z* side by side:

Add & Norm
Multi-Head Attention

Figure 2.18: Attention sublayer output

The concatenation produced a standard *dmodel* = 512-dimensional output:

$$LayerNormalization(v) = \gamma \frac{v - \mu}{\sigma} + \beta$$

Figure 2.19: Concatenation of the output of the 8 heads

Layer normalization will now process the attention sublayer.

Post-layer normalization

Each attention sublayer and each Feedforward sublayer of the Transformer is followed by **Post-Layer Normalization (Post-LN)**:

μ

Figure 2.20: Post-layer normalization

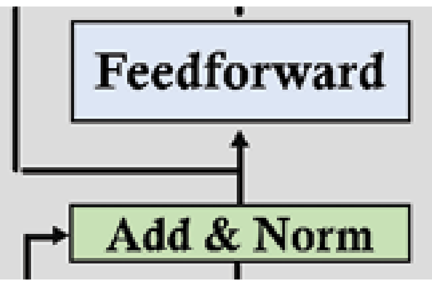
The Post-LN contains an add function and a layer normalization process. The *add* function processes the residual connections that come from the input of the sublayer and are normalized. The goal of the residual connections is to make sure critical information is not lost. The Post-LN or layer normalization can thus be described as follows:*LayerNormalization* (*x* + *Sublayer*(*x*))*Sublayer*(*x*) is the sublayer itself. *x* is the information available at the input step of *Sublayer*(*x*).The input of the *LayerNormalization* is a vector *v* resulting from *x* + *Sublayer*(*x*). *dmodel* = 512 for every input and output of the Transformer, which standardizes all the processes.Many layer normalization methods exist, and variations exist from one model to another. The basic concept for *v* = *x* + *Sublayer*(*x*) can be defined by *LayerNormalization* (*v*):

$$\mu = \frac{1}{d}\sum_{k=1}^{d} v_k$$

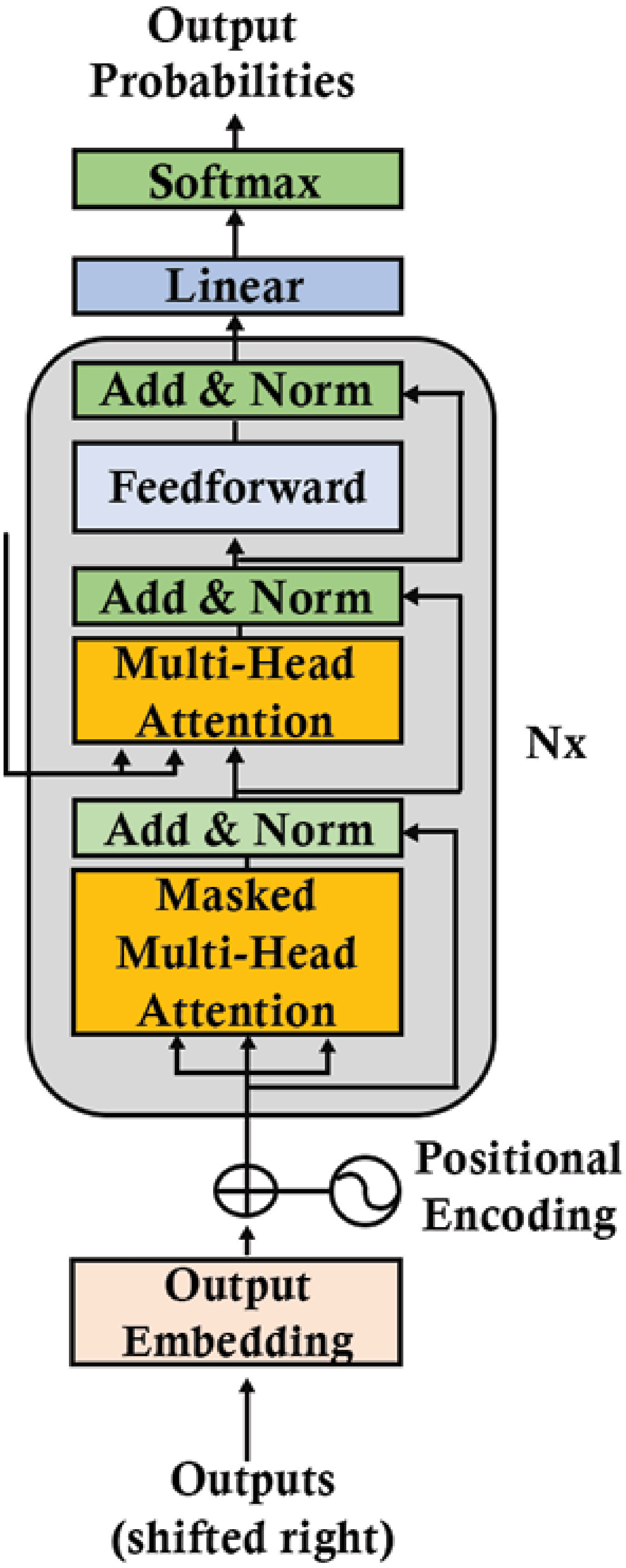
The variables are:σ is the mean of *v* of dimension *d*. As such:$\sigma^2 = \frac{1}{d}\sum_{k=1}^{d}(v_{k-\mu})\gamma$ is the

standard deviation *v* of dimension *d*. As such:

β



is a scaling parameter.



is a bias vector.

This version of *LayerNormalization* (*v*) shows the general idea of the many possible post-LN methods. The next sublayer can now process the output of the post-LN or *LayerNormalization* (*v*). In this case, the sublayer is a feedforward network.

Sublayer 2: Feedforward network

The input of the **Feedforward Network** (**FFN**) is the *dmodel* = 512 output of the post-LN of the previous sublayer:

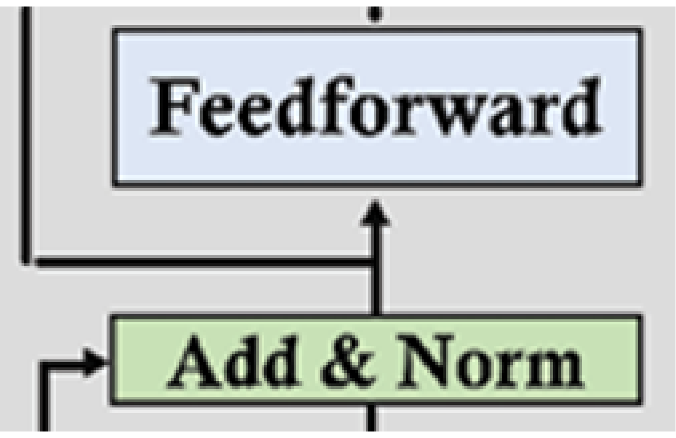


Figure 2.21: The feedforward sublayer

The FFN sublayer can be described as follows:

The FFNs in the encoder and decoder are fully connected.

The FFN is a position-wise network. Each position is processed separately and in an identical way.

The FFN contains two layers and applies a ReLU activation function.

The input and output of the FFN layers is d_{model} = 512, but the inner layer is larger with d_{ff} =2048.

The FFN can be viewed as performing two convolutions with size 1 kernels.

Taking this description into account, we can describe the optimized and standardized FFN as follows:$FFN(x) = \max (0, xW1 + b1) W2+b2$The output of the FFN goes to post-LN, as described in the previous section. Then the output is sent to the next layer of the encoder stack and the multi-head attention layer of the decoder stack.Let's now explore the decoder stack.

The decoder stack

The layers of the decoder of the Transformer model are *stacks of layers* like the encoder layers. Each layer of the decoder stack has the following structure:

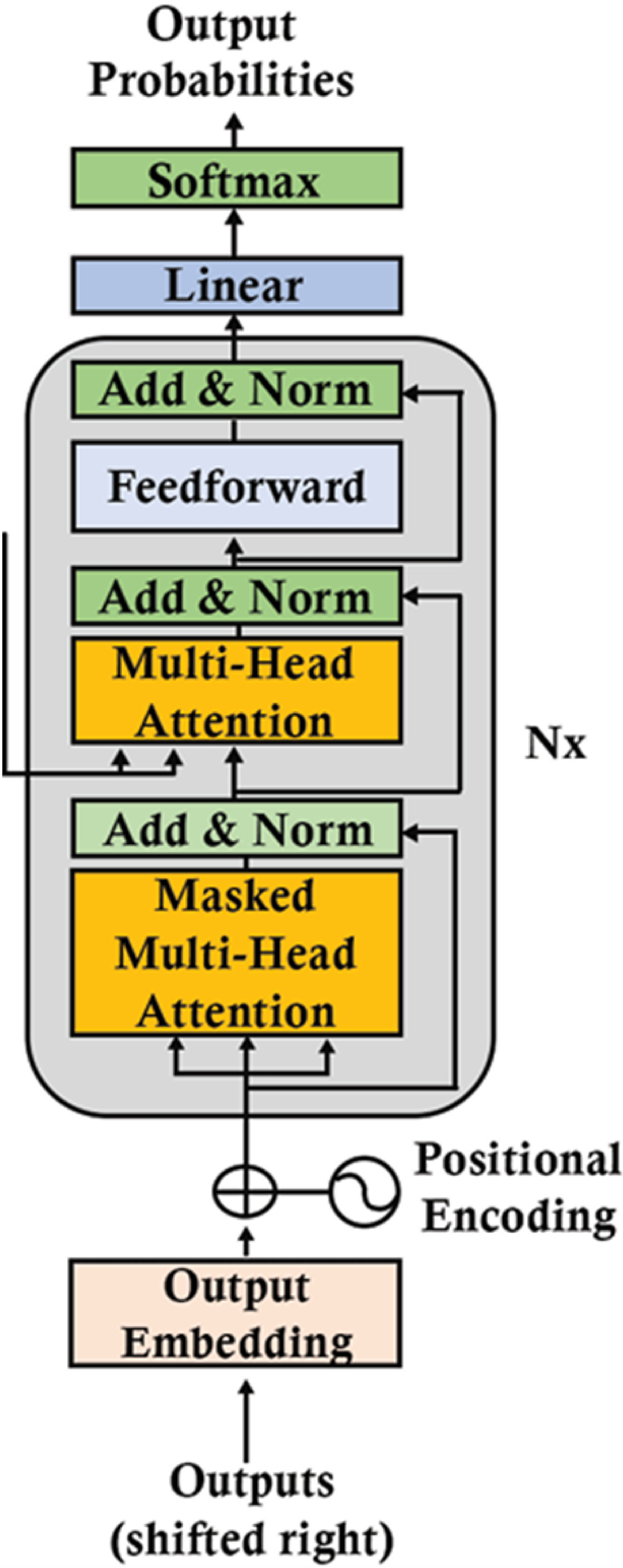

Output
Probabilities
Softmax
Linear
Add & Norm
Feedforward
Add & Norm
Multi-Head
Attention
Nx
Add & Norm
Masked
Multi-Head
Attention
Positional
Encoding
Output
Embedding
Outputs
(shifted right)

Figure 2.22: A layer of the decoder stack of the Transformer

The structure of the decoder layer remains the same as the encoder for all the *N*=6 layers of the Transformer model. Each layer contains three sublayers: a multi-headed masked attention mechanism, a multi-headed attention mechanism, and a fully connected position-wise feedforward network.The decoder has a third main sublayer, which is the masked multi-head attention mechanism. In this sublayer output, at a given position, the following words are masked so that the Transformer bases its assumptions on its inferences without seeing the rest of the sequence. That way, in this model, it cannot see future parts of the sequence.A residual connection, *Sublayer*(*x*), surrounds each of the three main sublayers in the Transformer model like in the encoder stack:*LayerNormalization* (*x* + *Sublayer*(*x*))The embedding layer sublayer is only present at the bottom level of the stack, like for the encoder stack. The output of every sublayer of the decoder stack has a constant dimension, *dmodel* like in the encoder stack, including the embedding layer and the output of the residual connections.We can see that the designers worked hard to create symmetrical encoder and decoder stacks.The structure of each sublayer and function of the decoder is similar to the encoder. In this section, we can refer to the encoder for the same functionality when we need to. We will only focus on the differences between the decoder and the encoder.

Output embedding and position encoding

The structure of the sublayers of the decoder is mostly the same as the sublayers of the encoder. The output embedding layer and position encoding function are the same as in the encoder stack.In the Transformer usage we are exploring through the model presented by *Vaswani* et al. (2017), the output is a translation we need to learn. I chose sto use a French translation:

Copy Explain

```
Output=Le chat noir était assis sur le canapé et le chien marron dormait sur le tapis
```

This output is the French translation of the English input sentence:

Copy Explain

```
Input=The black cat sat on the couch and the brown dog slept on the rug.
```

The output words go through the word embedding layer and then the positional encoding function, like in the first layer of the encoder stack.Let's see the specific properties of the multi-head attention layers of the decoder stack.

The attention layers

The Transformer is an auto-regressive model. It uses the previous output sequences as an additional input. The multi-head attention layers of the decoder use the same process as the encoder.However, the masked multi-head attention sublayer 1 only lets attention apply to the positions up to and including the current position. The future words are hidden from the Transformer, and this forces it to learn how to predict.A post-layer normalization process follows the masked multi-head attention sublayer 1 as in the encoder.The multi-head attention sublayer 2 also only attends to the positions up to the current position the Transformer is predicting to avoid seeing the sequence it must predict.The multi-head attention sublayer 2 draws information from the encoder by taking encoder (K, V) into account during the dot-product attention operations. This sublayer also draws information from the masked multi-head attention sublayer 1 (masked attention) by also taking sublayer 1(Q) into account during the dot-product attention operations. The decoder thus uses the trained information of the encoder. We can define the input of the self-attention multi-head sublayer of a decoder as:*Input_Attention* = (*Output_decoder_sub_layer* – 1(Q), *Output_encoder_layer*(K, V))A post-layer normalization process follows the masked multi-head attention sublayer 1 as in the encoder.The Transformer then goes to the FFN sublayer, followed by a post-LN and the linear layer.

The FFN sublayer, the post-LN, and the linear layer

The FFN sublayer has the same structure as the FFN of the encoder stack. The post-layer normalization of the FFN works as the layer normalization of the encoder stack.The Transformer produces an output sequence of only one element at a time:*Output sequence* = (y1, y2, ... yn)The linear layer produces an output sequence with a linear function that varies per model but relies on the standard method:$y = w * x + b$$w$ and b are learned parameters.The linear layer will thus produce the next probable elements of a sequence that a softmax function will convert into a probable element.The decoder layer, like the encoder layer, will then go from layer l to layer $l+1$, up to the top layer of the N=6-layer transformer stack.At the top layer of the decoder, the transformer will reach the output layer, which will map the outputs of the model to the size of the vocabulary to produce the raw logits of the prediction. The raw logits of the output can go through a softmax function, apply the values obtained to the tokens in the vocabulary and choose the best probable token for the task requested. Or, as shown in the *From One Token to an AI Revolution* section of *Chapter 1, What are Transformers?,* the pipeline can apply sampling functions that will vary from one API to another.Let's now see how the Transformer was trained and the persformance it obtained.

Training and performance

The original Transformer was trained on a 4.5 million sentence pair English-German dataset and a 36 million sentence pair English-French dataset.The datasets come from **Workshops on Machine Translation (WMT)**, which can be found at the following link if you wish to explore the WMT datasets: http://www.statmt.org/wmt14/The training of the original Transformer base models took 12 hours to train for 100,000 steps on a machine with 8 NVIDIA P100 GPUs. The big models took 3.5 days for 300,000 steps.The original Transformer outperformed all the previous machine translation models with a BLEU score of 41.8. The result was obtained on the WMT English-to-French dataset.BLEU stands for Bilingual Evaluation Understudy. It is an algorithm that evaluates the quality of the results of machine translations.The Google Research and Google Brain team applied optimization strategies to improve the performance of the Transformer. For example, the Adam optimizer was used, but the learning rate varied by first going through warmup states with a linear rate and decreasing the rate afterward.Different types of regularization techniques, such as residual dropout and dropouts, were applied to the sums of embeddings. Also, the Transformer applies label smoothing that avoids overfitting with overconfident one-hot outputs. It introduces less accurate evaluations and forces the model to train more and better.Several other Transformer model variations have led to other models and usages that we will explore in the subsequent chapters.Before the end of the chapter, let's get a feel of the simplicity of ready-to-use transformer models in Hugging Face, for example.

Hugging Face Transformer models

Everything you saw in this chapter can be condensed in to a ready-to-use Hugging Face transformer model.With Hugging Face, you can implement machine translation in three lines of code!Open `Multi_Head_Attention_Sub_Layer.ipynb` in Google Colaboratory. Save the notebook in your Google Drive (make sure you have a Gmail account). Then, go to the two last cells.We first ensure that Hugging Face transformers are installed:

Copy Explain

```
!pip -q install transformers
```

The first cell imports the Hugging Face pipeline that contains several transformer usages:

Copy Explain

```
#@title Retrieve pipeline of modules and choose English to French translation
from transformers import pipeline
```

We then implement the Hugging Face pipeline, which contains ready-to-use functions. In our case, to illustrate the Transformer model of this chapter, we activate the translator model and enter a sentence to translate from English to French:

Copy Explain

```
translator = pipeline("translation_en_to_fr")
#One line of code!
print(translator("It is easy to translate languages with transformers", max_length=40))
```

And *voilã*! The translation is displayed:

Copy Explain

```
[{'translation_text': 'Il est facile de traduire des langues à l'aide de
transformateurs.'}]
```

Hugging Face shows how transformer architectures can be used in ready-to-use models.

Summary

In this chapter, we first started by examining the mind-blowing long-distance dependencies transformer architectures can uncover. Transformers can perform transductions from written and oral sequences to meaningful representations as never before in the history of **Natural Language Understanding (NLU)**.These two dimensions, the expansion of transduction and the simplification of implementation, are taking artificial intelligence to a level never seen before.We explored the bold approach of removing RNNs, LSTMs, and CNNs from transduction problems and sequence modeling to build the Transformer architecture. The symmetrical design of the standardized dimensions of the encoder and decoder makes the flow from one sublayer to another nearly seamless.We saw that beyond removing recurrent network models, transformers introduce parallelized layers that reduce training time. In addition, we discovered other innovations, such as positional encoding and masked multi-headed attention.The flexible, original Transformer architecture provides the basis for many other innovative variations that open the way for yet more powerful transduction problems and language modeling.We will zoom more in-depth into some aspects of the Transformer's architecture in the following chapters when describing the many variants of the original model.The arrival of the Transformer marks the beginning of a new generation of ready-to-use artificial intelligence models. For example, Hugging Face and Google Brain make artificial intelligence easy to implement with a few lines of code.Before continuing to the next chapter, make sure you capture the details of the paradigm shift of architecture of the original Transformer. You will then be able to face any present and future transformer model.In this chapter, we have dived into the architecture of transformers. Now, we will see what they can do. In *Chapter 3, Emergent vs Downstream Tasks: the Unseen Depths of Transformers*, we will explore the wide range of tasks transformer models can perform.

Questions

1. Transformers can encode and decode text representations. (True/False)
2. **Natural Language Understanding** (**NLU**) is a subset of **Natural Language Processing** (**NLP**). (True/False)
3. Language modeling algorithms generate probable sequences of words based on input sequences. (True/False)
4. A transformer is a customized LSTM with a CNN layer. (True/False)
5. A transformer does not contain LSTM or CNN layers. (True/False)
6. Attention examines all the tokens in a sequence, not just the last one. (True/False)
7. A transformer uses a positional vector, not positional encoding. (True/False)
8. A transformer contains a feedforward network. (True/False)

9. The masked multi-headed attention component of the decoder of a transformer prevents the algorithm parsing a given position from seeing the rest of a sequence that is being processed. (True/False)
10. Transformers can analyze long-distance dependencies bestter than LSTMs. (True/False)

References

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin, 2017, Attention Is All You Need, https://arxiv.org/abs/1706.03762Hugging Face Transformer Usage: https://huggingface.co/transformers/usage.htmlTensor2Tensor (T2T) Introduction: https://colab.research.google.com/github/tensorflow/tensor2tensor/blob/master/tensor2tensor/notebooks/hello_t2t.ipynb?hl=enManuel Romero Notebook with link to explanations by *Raimi Karim*: https://colab.research.google.com/drive/1rPk3ohrmVclqhH7uQ7qys4oznDdAhpzFGoogle language research: https://research.google/teams/language/Hugging Face research: https://huggingface.co/transformers/index.html*The Annotated Transformer*: http://nlp.seas.harvard.edu/2018/04/03/attention.html*Jay Alammar, The Illustrated Transformer*: http://jalammar.github.io/illustrated-transformer/

Further Reading

Transformers could not have increased their potential without hardware innovation. Nvidia, for example, offers interesting insights on transformers and the related hardware:https://blogs.nvidia.com/blog/2022/03/25/what-is-a-transformer-model/https://docs.nvidia.com/deeplearning/transformer-engine/user-guide/index.html

Previous Chapter

Next Chapter