Since the introduction of the Transformer architecture [by Ashish Vaswani et al. in 2017](https://arxiv.org/abs/1706.03762), it has become the de facto standard for any large-scale natural language processing task. From the [pioneering GPT model](https://s3-us-west-2.amazonaws.com/openai-assets/research-covers/language-unsupervised/language_understanding_paper.pdf) in 2018 to the now impressive ChatGPT, even text-to-image synthesis models such as Stable Diffusion are based on, or inspired by, the Transformer.

Given its significance as a breakthrough in the field of NLP and machine learning as a whole, it is surprising that the original paper which introduced the Transformer offers only a limited explanation of its architecture and, most critically, how the metaphorical gears of a Transformer model turn and why they work so well. This article aims to fill that gap by offering a comprehensive and illuminating illustration of the Transformer through an intuitive, visual-first approach and “decode” all the fuss around its incredible comprehension of language.

## Why Transformers?

Along with the Transformer architecture that the authors of the paper proposed, they brought a revolutionary approach to sequence based modeling: the self-attention mechanism. Previously, language modeling tasks were dominated by recurrence-based techniques such as RNNs, GRUs, and LSTMs, which struggled to retain information from earlier time steps in longer sequences, resulting in poor performance on tasks involving long-range dependencies. The Transformer architecture sidesteps these limitations by allowing the entire input sequence to be processed simultaneously, enabling it to have perfect memory and significantly improved computational speed. On the other hand, the concept of self-attention is an extension from attention mechanisms introduced before the Transformer, which allows the model to focus on certain parts of the input, giving more “attention” to one part than another.

In the following section, we will delve into the fundamental methodology underlying the Transformer model and most sequence-to-sequence modeling approaches: the encoder and the decoder. This will serve as a springboard for dissecting the Transformer model architecture and gaining an in-depth understanding of its inner workings.

## The Encoder-Decoder Concept

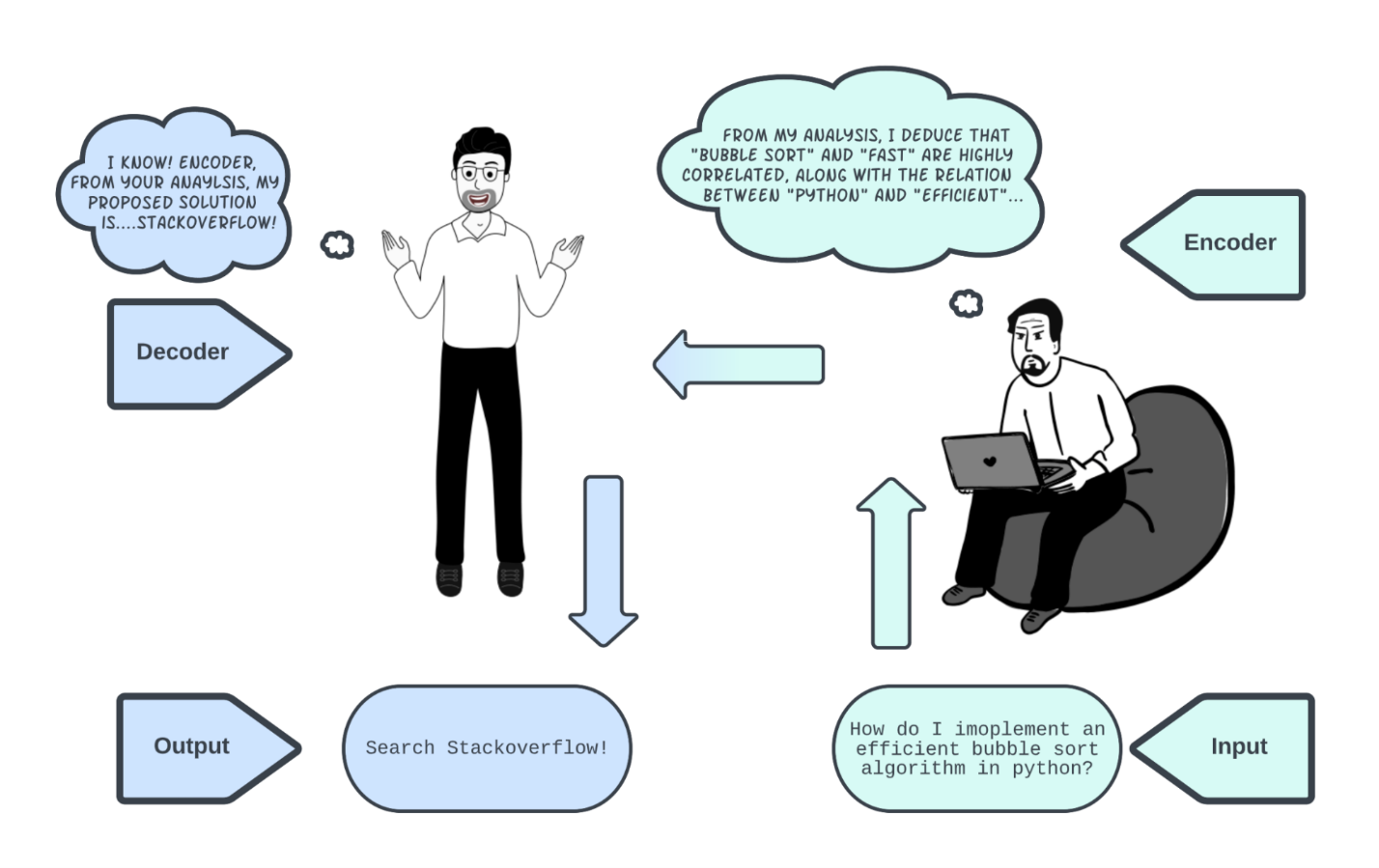
The Transformer model relies on the interactions between two separate, smaller models: the encoder and the decoder. The encoder receives the input, while the decoder outputs the prediction. Implementing an encoder and a decoder to process sequence-to-sequence data has been relatively standard practice since 2014, first applied to recurrence-based models, then later used by the Transformer.

Before the existence of the encoder-decoder architecture, predictions for sequence-based problems were solely based on accumulated knowledge over the entire input sequence “squeezed” into one sample's worth of representation. Although architectures such as LSTM and GRU attempted to improve the issue of long-range dependence, they did not completely resolve the underlying problem of RNNs, which was the inability to carry the information of long sequences through prediction fully.

In an encoder-decoder schema, the encoder takes in the entirety of the input sequence. It transforms it into a vectorized representation that contains accumulated knowledge of the input sequence at every time step. Then the entire vectorized representation of the input sequence is fed into the decoder, which “decodes” the information collected by the encoder and attempts to make a valid prediction.

In the context of the Transformer model, one can make an analogy between the encoder and the decoder as a researcher developing initial ideas and discoveries. At the same time, a programmer implements a solution based on the researcher’s findings. Think of the encoder as a researcher picking out the crucial aspects of the input, such as sentence structure, syntax, and semantics. The encoder then passes on the insights learned from the inputs to the decoder. Like a programmer, the decoder will “program” a practical solution based on the insights gained by the encoder, or, in our analogy, the researcher.

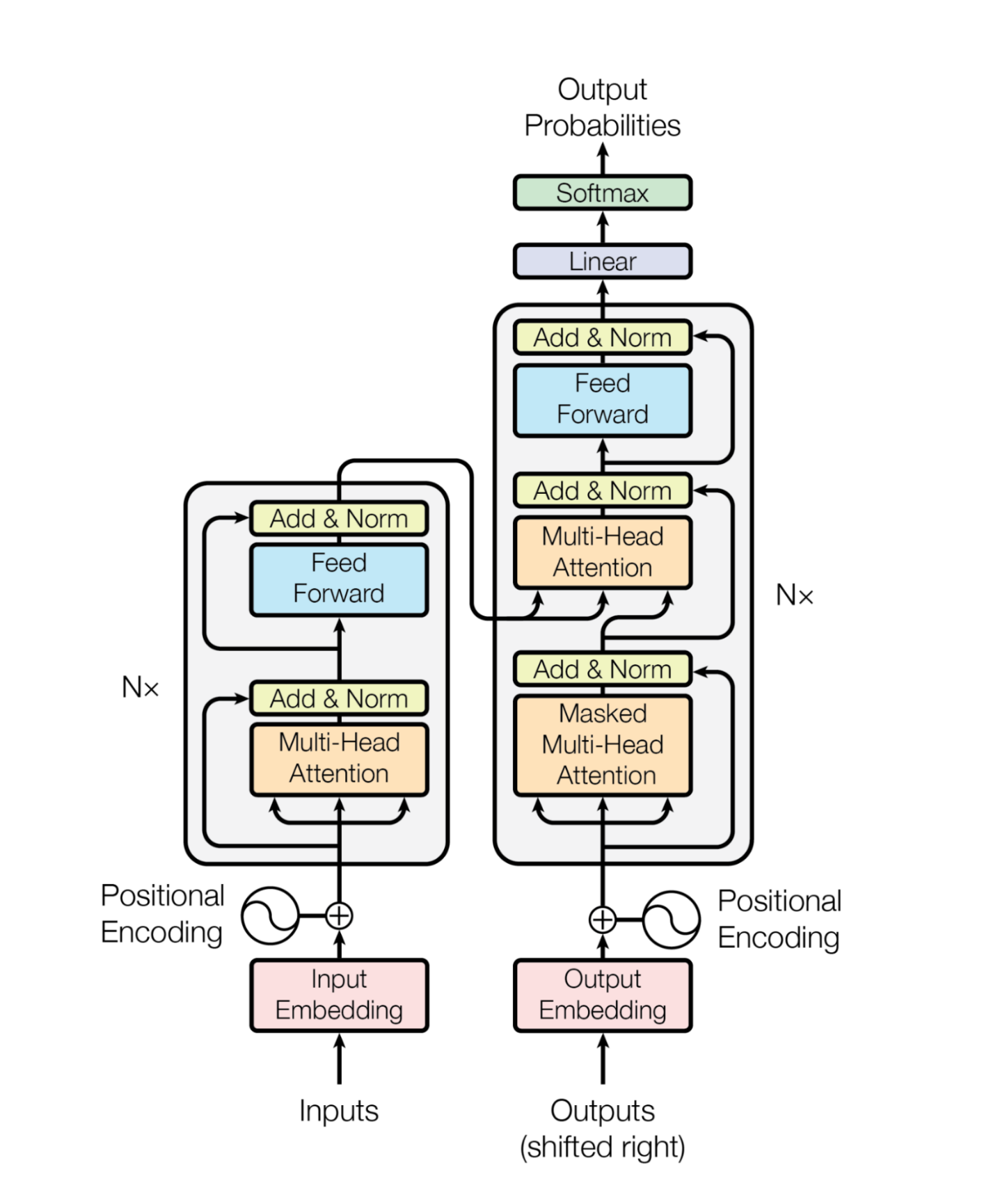
The decoder receives a separate sequence of inputs before decoding the information provided by the encoder. The decoder bases its predictions not only on the original input sequence but also on its previous outputs. It is auto-regressive in nature. Every time the decoder makes a prediction, it produces a single word and then concatenates it with words predicted by the model in previous time steps. This prediction loop continues until the maximum number of outputs is reached—usually a hyperparameter specified by the user—or the model deduces that it has reached the natural end of a sentence or phrase.



## Architecture Overview

Before diving into the nuts and bolts of the Transformer model, it's necessary to have a broad understanding of how information flows through the various components of the model and in what ways these components are essential to understanding the fundamental structure of language, all the while (hopefully) producing a more-or-less correct output. The image below is extracted from the original paper describing the Transformer model architecture.

Let's see how these components fit together.



### Encoders

The encoder of a Transformer is responsible for turning an input sequence into a machine-readable representation. This representation captures the similarity between words and their relative position in the sequence. The input sequence is first passed through an input embedding and a positional encoding layer. These operations transform the input sequence into a form suitable for processing by the encoder layer.

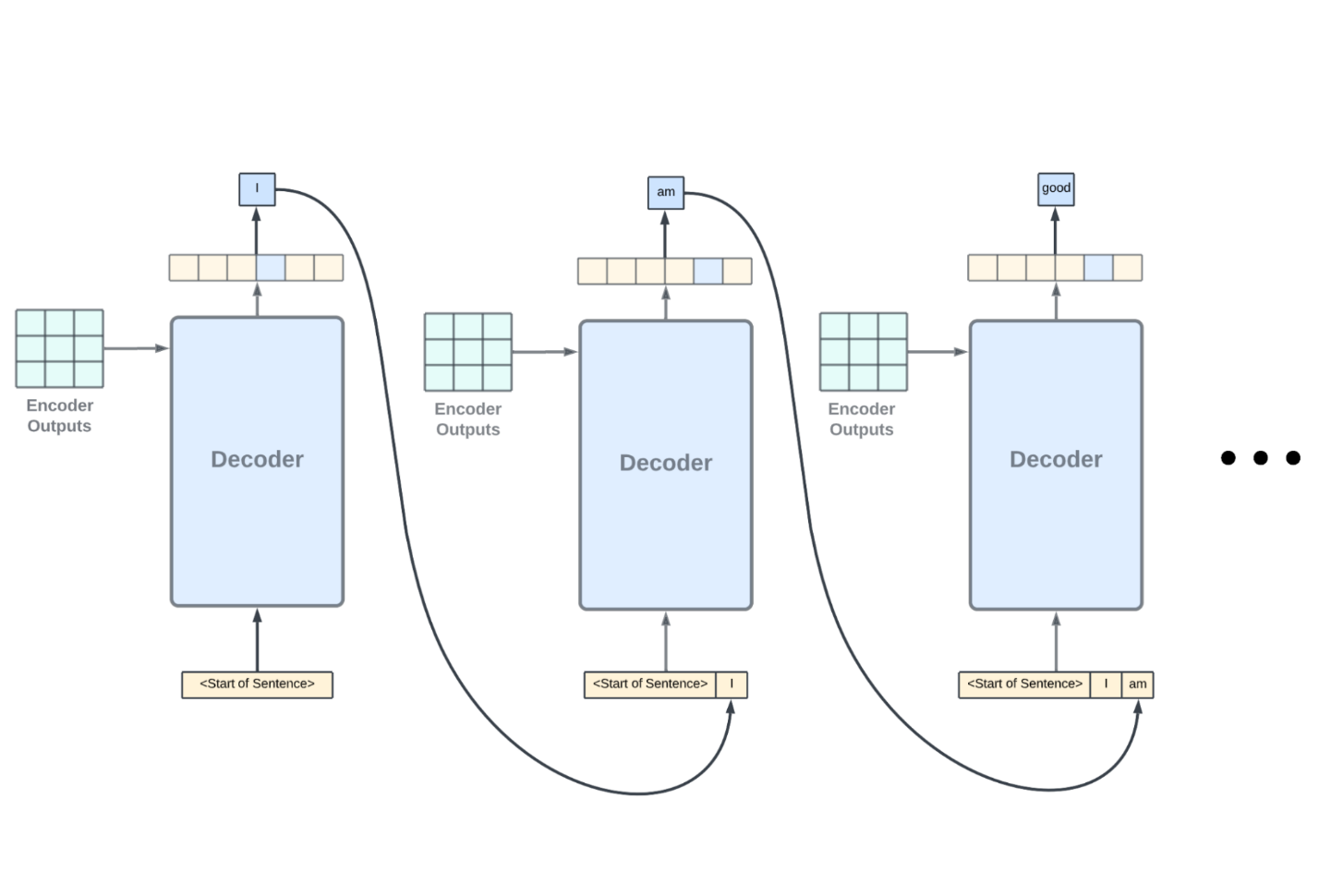
The encoder layer is the encoder's core, where the bulk of the "magic" happens. In the original paper, it's suggested to stack the encoder layer six times. However, this can be adjusted depending on the situation. The encoder consists of a single multi-head attention block followed by a feed-forward neural network with residual connections and layer normalizations after both outputs.

The multi-head attention block is capable of discovering complex relationships between words and determining how each word contributes to the meaning of the input sequence. This allows the encoder to gain a deep understanding of the input sequence in a way that is similar to how humans analyze language. After the encoder layer, a feed-forward network further transforms the input sequence in preparation for the next encoder layer. Once the encoding process is complete, the accumulated knowledge gained by the encoder (the output of the last encoder layer) is passed on to the decoder, which uses it to generate the final output sequence.

### Decoders

The decoder receives the accumulated knowledge produced by the encoder, as depicted in our figure below.

Typically for the very first "prediction cycle," the decoder is fed with a "start of sentence" token since there were no "previous'' outputs. The decoder layer is similar to the encoder layer because it utilizes the same concept to analyze the combined information from the encoder and previous predictions. The decoder first derives insights solely based on its acquired prediction knowledge. Then, the data is combined with an encoder output to be further processed and analyzed. Finally, the prediction for the next time step is outputted as a probability of how likely the word chosen is the next in the sequence of outputs.



## Word Embeddings

The objective of word embedding is to convert the given input sequence into machine-readable representations. One way to do this would be to use one-hot encoding, where each word is represented by a large, sparse vector with a non-zero value at the word's corresponding index in the vector space. However, this approach is neither effective or elegant, as it results in huge vectors, 99% of which are zero values, which can negatively affect model performance due to [the curse of dimensionality](https://www.sciencedirect.com/topics/mathematics/curse-of-dimensionality). Additionally, for such a huge vector, the information that it's able to convey—a unique identifier for a word—is astonishingly small.

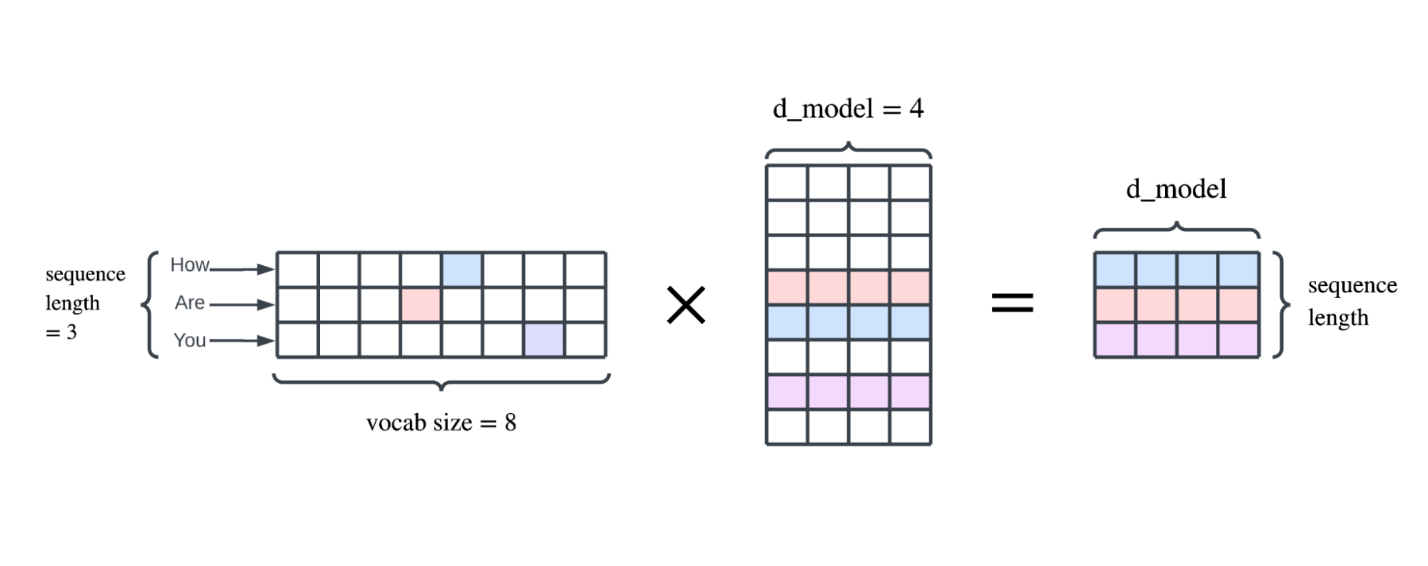
Word embeddings apply a further transformation learned from large corpora to the sparse, one-hot encoded vectors, producing dense, relatively low-dimensional representations of words while considering contextual information. For instance, in the sentence “the cat is brown and furry while the refrigerator is a lifeless, silver-colored machine,” the word embedding for “cat” would be far away from the word embedding of “refrigerator” since these two words convey completely different meanings. On the other hand, the word “brown” and “silver” would have similar embeddings as both words are used to describe colors. Typically, cosine similarity is used to compute the distance between two word embedding vectors.

One can view the task of word embedding as a pre-training technique for language models. Without using word embeddings, the model would have to learn contextual information about each word during training. Word embedding accomplishes this beforehand, allowing more information to be fed into the model.

Various methods and pre-trained algorithms are proposed to generate densely packed word embeddings effectively. However, the Transformer uses a word embedding attached to the model and is trained from scratch. Using a word embedding without pre-initialized parameters allows the model to learn the embedding representation with respect to the context of the input data and the overall model structure. The word embedding is the very first component in the transformer; the one-hot-encoded input is multiplied with a weight matrix trained through the backpropagation of the entire Transformer model.

The weight matrix has a shape (number of unique vocabularies, dimension of embedding). For the sake of consistency, we will use the terminology from the original paper on the Transformer and refer to the dimension of the word embedding as “d\_model”. The value of d\_model is set to 512 by the authors of the Transformer paper.

Word embeddings can be viewed as a lookup table that maps input, one-hot-encoded vectors to a lower dimensional space of shape (length of sequence, d\_model). This dimensionality reduction is achieved through the multiplication of the input vectors and a weight matrix. These embeddings are able to capture dependencies between words and provide contextual information. A common illustration of this is to subtract the embedding vector of "man" from "king" and obtain a vector similar to the embedding vector of "queen," implying the inherent relation between the three words.



Word embedding may be able to inject useful insights into the input sequence; it is unable to provide any positional clues to the model of where each word is relative to the input sequence. This is where positional encoding comes into play.

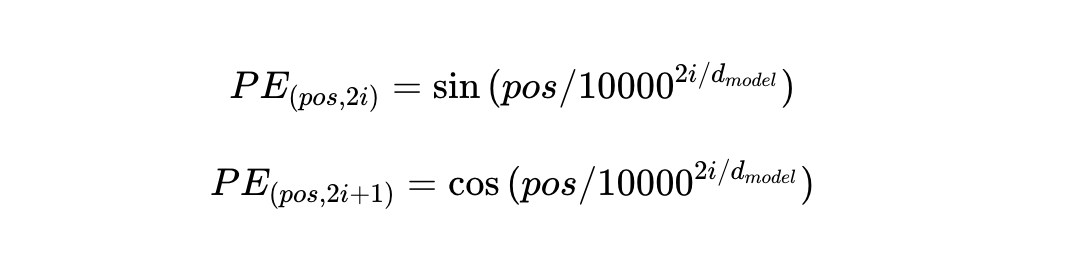
## Positional Encoding

As a refresher, the Transformer architecture discards the use of recurrence-based networks, instead relying on self-attention mechanisms to process input sequences. While this allows for faster training and improved handling of long-range dependencies, it does not inherently provide any information about the relative positions of words in the input.

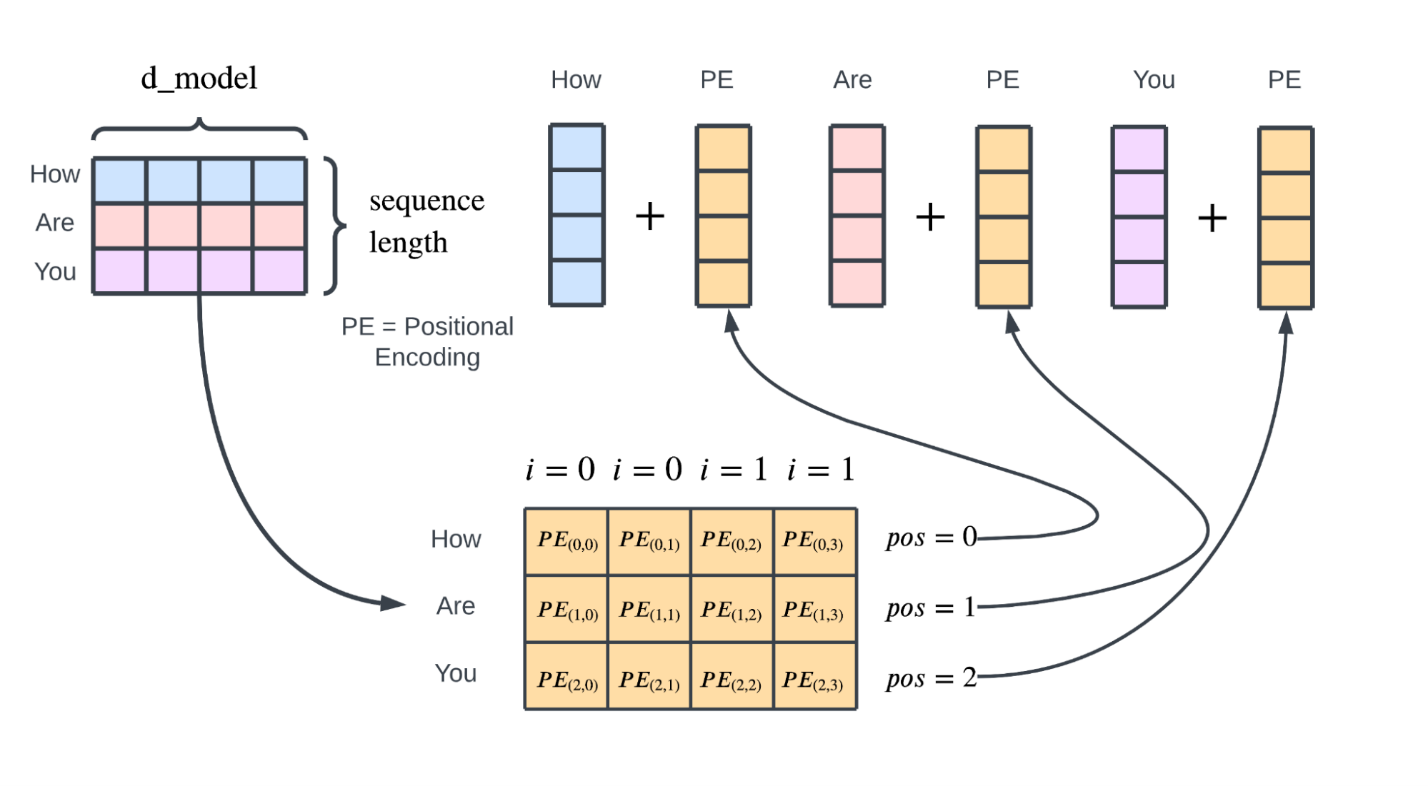
For instance, the sentences “I pet my dog” and “The dog pet me” convey distinct meanings depending on the position of “dog.” Despite this, the word embedding for “dog” in both cases is identical. Recurrence-based models process information in order, thus, the position of each word is already implied, but the Transformer requires additional information to differentiate the two “dogs” in these sentences.

To address this issue, positional encoding is employed, which adds a unique vector of length d\_model to each word embedding vector. This positional encoding vector is determined by the position of the word within the input sequence. This allows the model to extract the relative positions of words in the input and incorporate this information in its processing.

In the equation above, “pos” represents the position of the word in the input sequence while “i” is the position of each value within the word embedding. Both positional encoding functions are applied, producing two unique values for each “i” value. Thus, to output vectors of length d\_model, the value of “i” will range from 0 to half of d\_model. For example, in the sentence “How are you?" the word embedding for “you” would have a pos value of 3 while its “i” value ranges from 0 to 255  since d\_model, or the embedding dimension is 512.



Sinusoidal position encoding carries several advantages::



1. It has been stated in the original paper that the Transformer is able to “extrapolate to sequence lengths longer than the ones encountered during training” with the positional encoding function.
2. The relative position between words can be extrapolated since, for words in positions close to each other, their positional encoding vector will be similar as well.

After the positional encoding component, its output, with shape (length of sequence, d\_model), will be passed into the first encoder layer consisting of the self-attention block and the feed-forward neural network.

Note that the same preprocessing scheme (word embedding and positional encoding) is used for the input sequence to the decoder as well, which we will discuss later.

## Intuition Behind Self-Attention

Before getting into the technical details of self-attention, it’s critical to have a general intuition of the whys and the hows behind the algorithm.

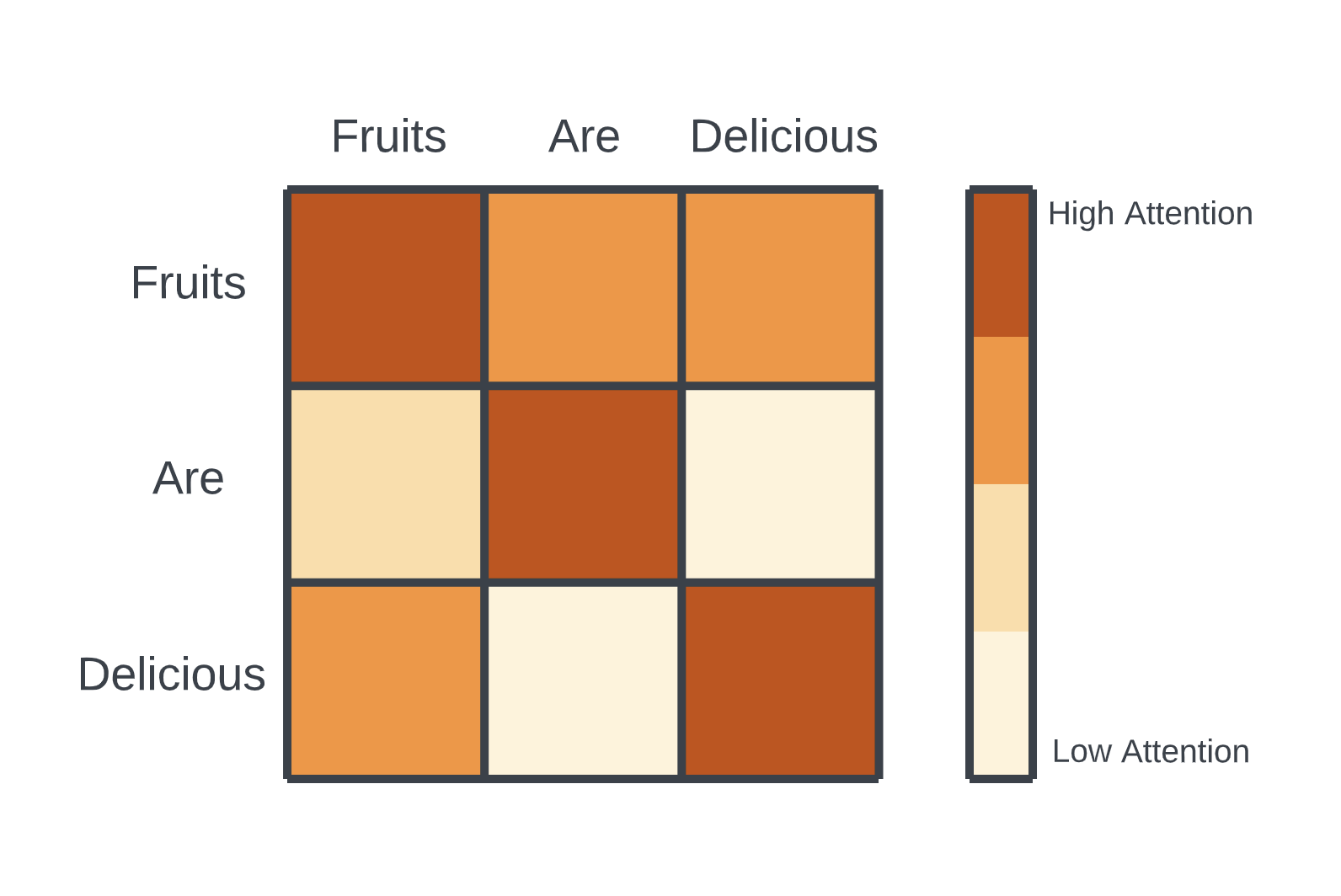
Attention, or global attention, in general, is nonetheless one of the most important contributing factors to successful natural language processing models. The basic idea behind attention is that the model can focus on certain input words more than others, depending on their relevance to the context. In other words, the model assigns varying degrees of "attention" to each input word, with more important words receiving more attention.

For example, consider the following sentences: "My dog has black, thick fur as well as an active personality. I also have a cat with brown fur. What is the breed of my dog?" Without attention, the model would assign equal importance to the information about the cat and the dog, which could lead to incorrect or misleading answers. However, with attention, a well-trained language model would assign less attention to the phrase "brown fur" because it is irrelevant to the question being asked. This ability to selectively focus on important words is a crucial component of language learning and helps improve the performance of natural language processing models.

The Transformer employs a self-attention mechanism called "scaled dot product attention." While global attention considers the importance of each word relative to the entire input sequence, self-attention deciphers dependencies between words within the sequence. For example, in the sentence "I went to the store and bought tons of fruits along with some furniture. They tasted amazing," a human reader would infer that "they" refers to the fruits, not the furniture. A model using global attention might assign higher attention values to "fruits," "furniture," and "amazing" without understanding the relationship between these words. In contrast, self-attention, which compares every word in the input sequence to every other word, would be able to discover the intended meaning of "they."

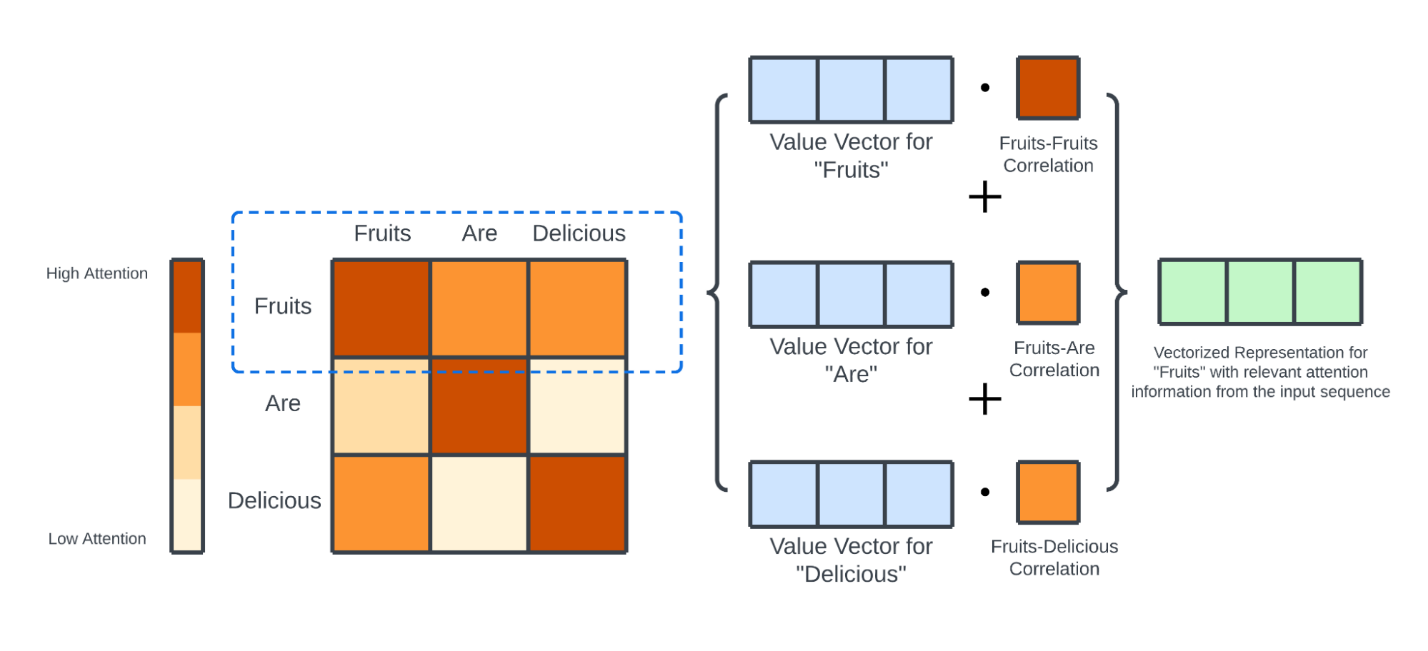
A helpful way to understand how self-attention assigns attention values or weights is to construct a correlation matrix by comparing each element of the input sequence with every other element in the sequence.

Self-attention is a mechanism that is analogous to information retrieval systems like Google search. In self-attention, there are three components: the query, the key, and the value. In an information retrieval system, the query is like the search query you enter into the search bar, the keys are like the titles of websites in the database, and the values are like the websites themselves. When you enter a search query, the system will compare it to the keys in the database and rank the values based on how similar the keys are to your query. While the actual Google search engine is much more complex than this, this simple example illustrates the roles of queries, keys, and values in self-attention.



The ideas of queries, keys, and values, as well as their interactions, are imitated by self-attention. Each word's vectorized representation is projected into three smaller-sized vectors that represent the word's key, value, and query.

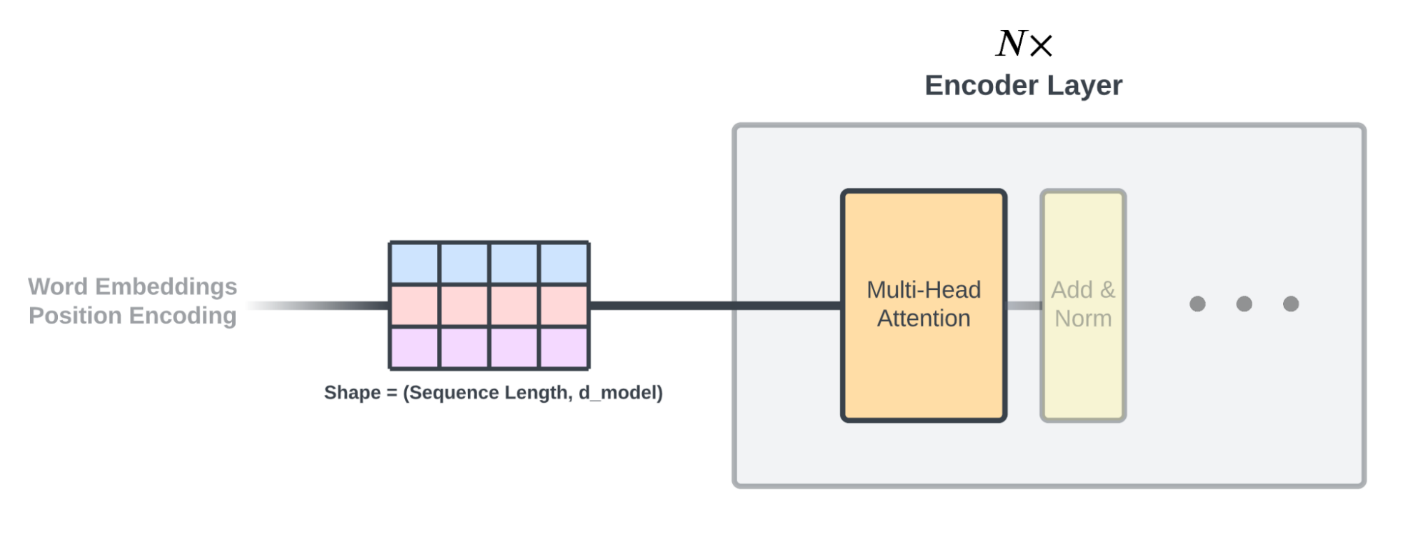
The key vector of every word in the input sequence, including itself, is compared with each word's query to determine the best match. As a result, the input sequence generates the attention matrix that is seen above, with each value in the matrix representing an "attention weight" for a particular combination of words. For each word, their attention weights against every word in the input sequence (that is, the attention values in the entire row of the attention matrix) are used as weights to calculate a weighted sum of their corresponding value vectors. This is done for every word. A new matrix is created with those elements as its entries. This essentially recreates the word embedding representation of the input sequence but with attention information.



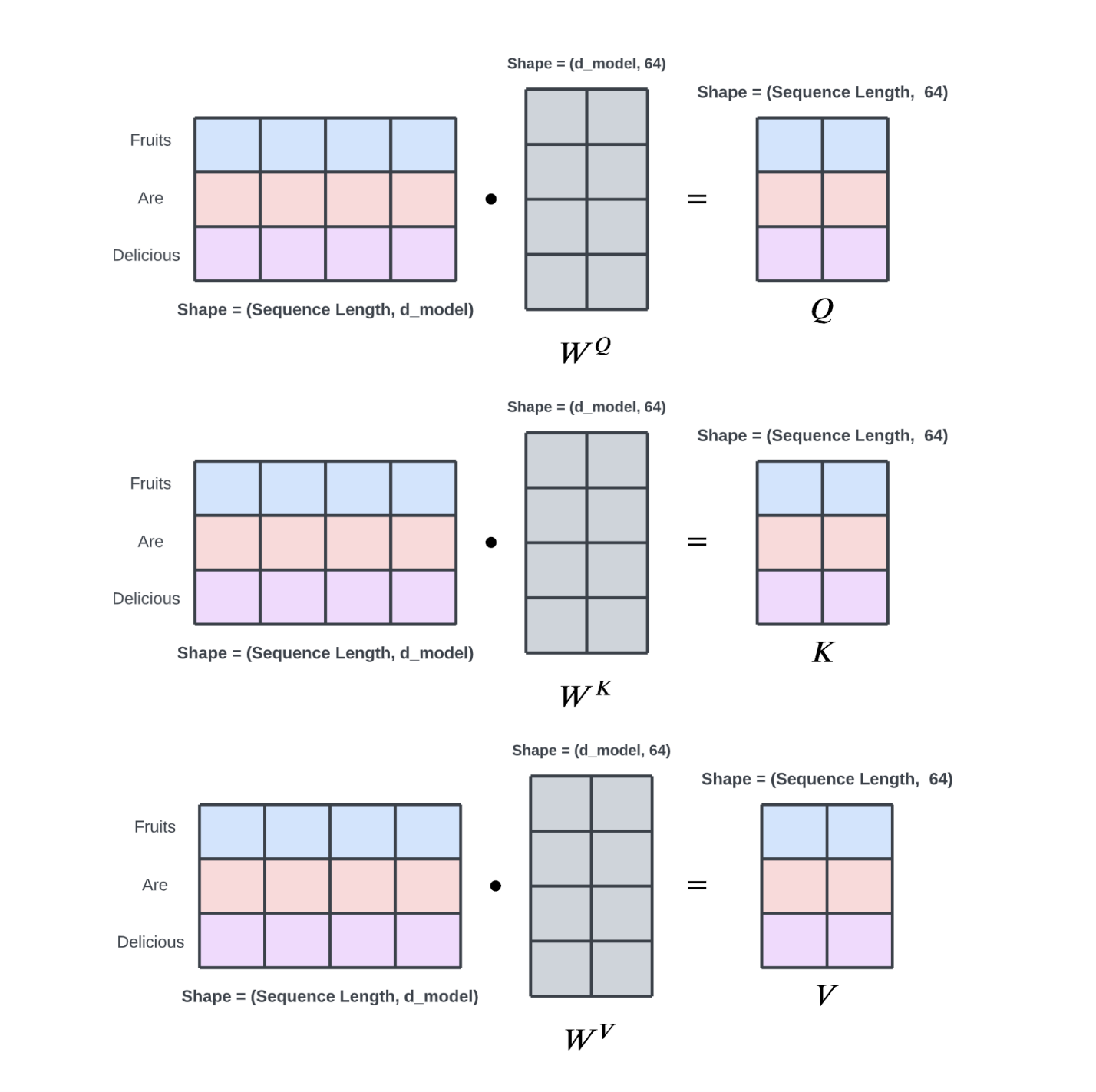
## Math Behind Self Attention

Recall that the output from positional encoding is in the shape of (length of sequence, d\_model) where d\_model can be interpreted as the embedding dimension. This matrix is the input to the encoder layer. For encoder layers that follow the first one, their input will be the output from the previous encoder layer.

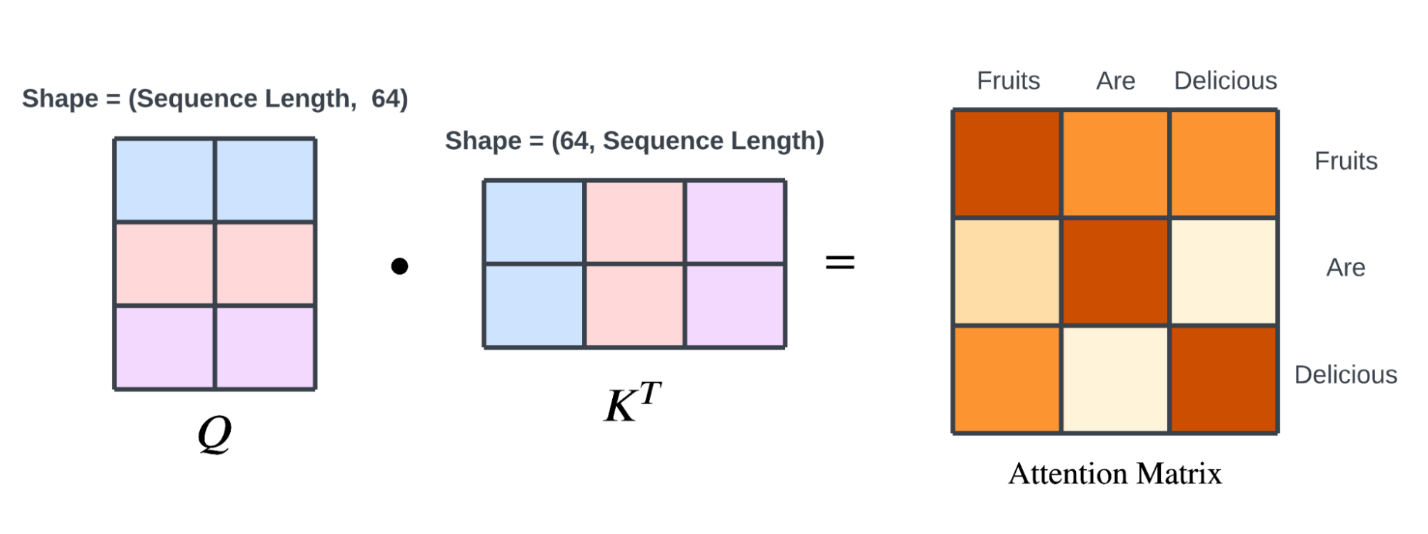
The input matrix is linearly projected into three smaller matrices representing the queries, keys, and values through three separate weight matrices. These matrices have the shape (length of sequence, 64), where dimension 64 is an arbitrary value chosen by the authors of the paper. The weight matrices for the query, key, and value are referred to as WQ, WK, and WV, respectively. These weight matrices are trained through backpropagation along with the entire model. The value of 64 chosen for the dimensions of the matrices does not affect the computation of self-attention.



Each entry in the query, key, value matrices corresponds to the word’s query, key, and value. The query and key matrices are dotted together to produce the “correlation matrix” shown above.



To understand why taking the dot product between the query and key matrices would produce similarity scores between words, remember that the entry in the query matrix corresponds to the vectorized representation of the word in the query, while the entry in the key matrix corresponds to the vectorized representation of the word in the key.

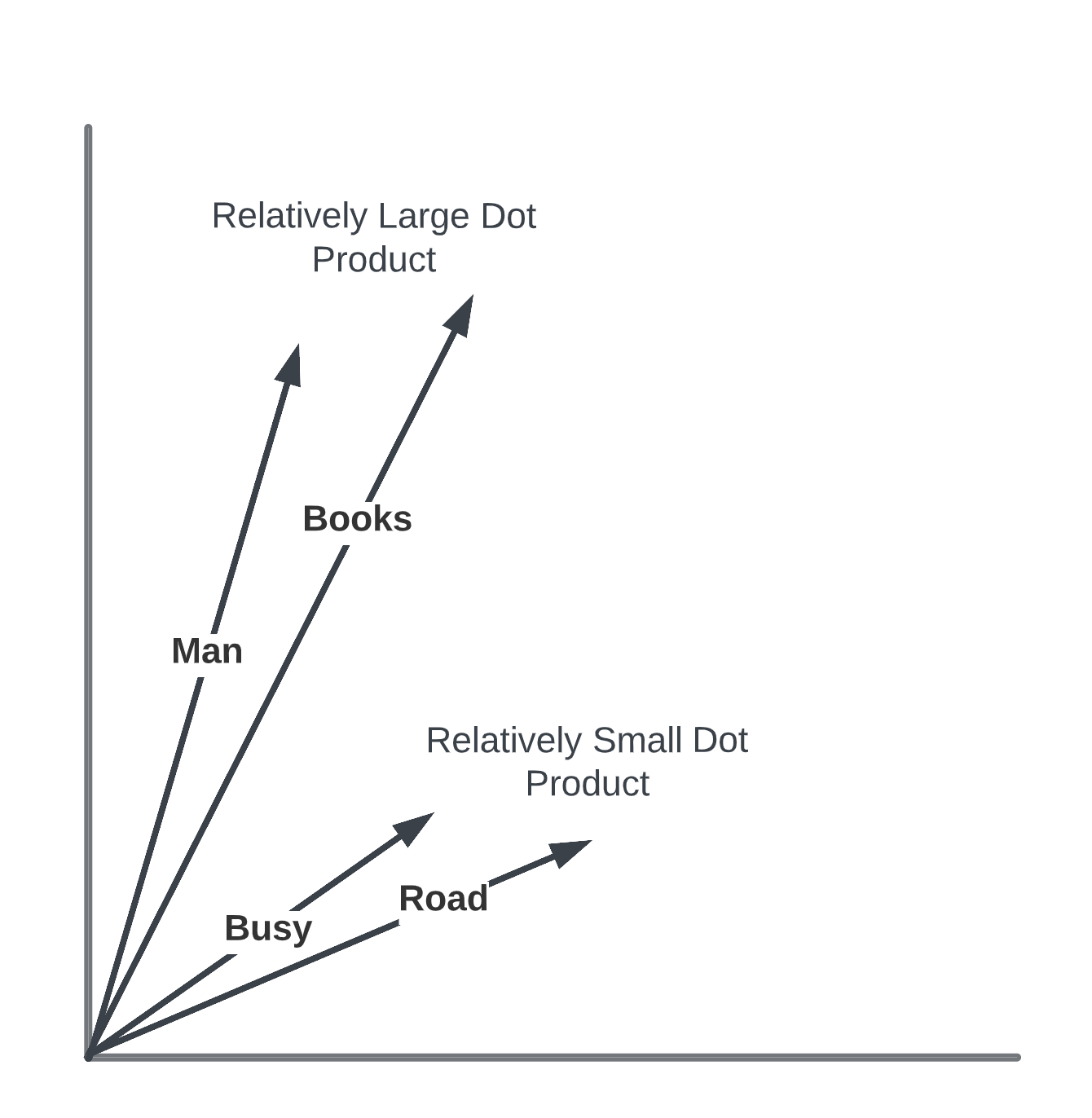


The dot product between two vectors in a two-dimensional space can be seen as a measure of the cosine similarity between the vectors, scaled by the product of their magnitudes. As an example, consider the sentence, "The man walks down the busy road carrying some books he just bought; where could he be coming from?" For the sake of visualization, let's assume that the dimensionality of the model (d\_model) is equal to 2 so that the query and key vectors can be projected into a 2-dimensional space.

In this example, let's take the query vectors for the words "man" and "busy," along with the key vectors for the words "books" and "road." We would expect the query and key vectors for "man" and "books" to have a relatively large cosine similarity since the word "books" describes the "man." Similarly, we would expect the query and key vectors for "busy" and "road" to have a relatively large cosine similarity as well.

However, to a human analyzer, the relationship between "busy" and "road" may not be as relevant to the question being asked in the sentence as the relationship between "man" and "books." In other words, the phrase "busy road" might not be as useful in inferring where the man is coming from as the relationship between "man" and "books," which suggests that he might be coming from a library.

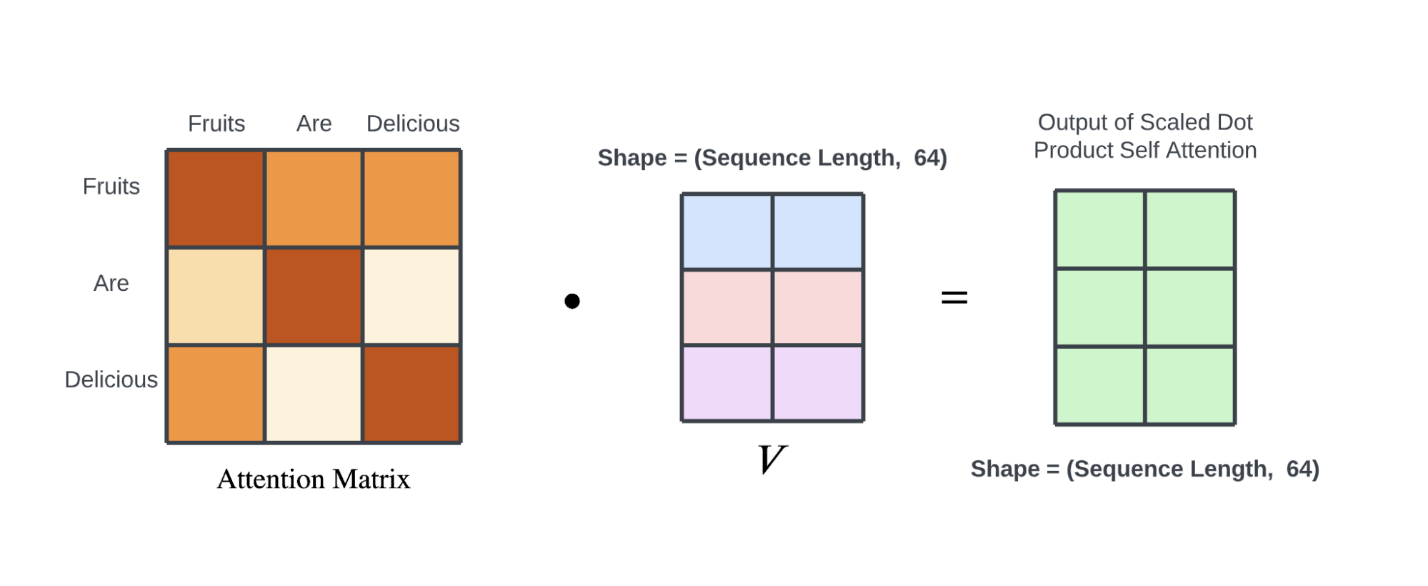
To illustrate this difference, the relative magnitude of the query and key vectors produced by the query and key weight matrices for "busy" and "road" would be smaller than for "man" and "books." Since the dot product measures not only the cosine similarity but also the product of the magnitudes of the vectors, the final dot product would give more weight to the relationship between "man" and "books" than to the relationship between "busy" and "road."



The attention matrix produced by the dot product of the query and key matrices has a shape of (length of sequence, length of sequence). Each value in the attention matrix is divided by the square root of the size of the key, query, and value matrices (8, in this case). This step is used to stabilize the gradients during training. The attention matrix is then passed through a softmax function, which normalizes its values to be between 0 and 1 and ensures that the values of each row in the matrix sum up to 1.

As mentioned earlier, a weighted sum is performed using the attention values and the value vectors. The normalization of the attention scores to sum up to 1 makes this weighted sum operation possible. Finally, the normalized attention matrix is dotted with the value matrix, producing a matrix of size (length of sequence, 64), which can be seen as a smaller, vectorized representation of the input sequence with attention information.

The first row of the output matrix is a weighted sum of the row vectors in the value matrix, with the weights being the attention values of the first word against all other words in the input sequence.



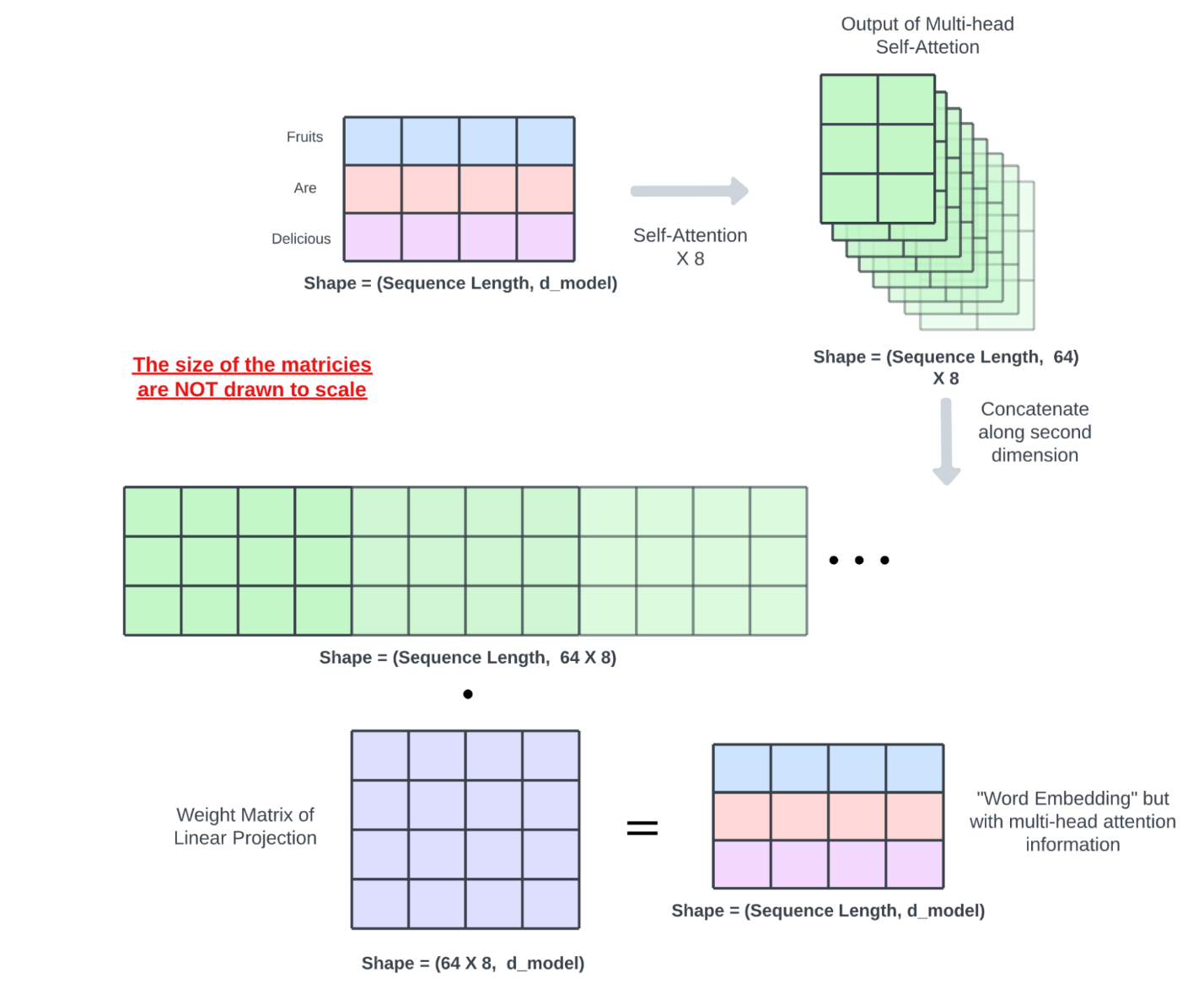
Note that the output matrix has a size of (length of sequence, 64) and not (length of sequence, 512). The reason for this will become clear in the next section, where we discuss multi-head self-attention and the process of upscaling the "attention word embedding." It is important to remember that the output matrix should have the same size as the original word embedding since it will be used as the input to the next encoder layer, which, in the case of the first encoding layer, expects the word embedding as input.

## Multi Head Self Attention

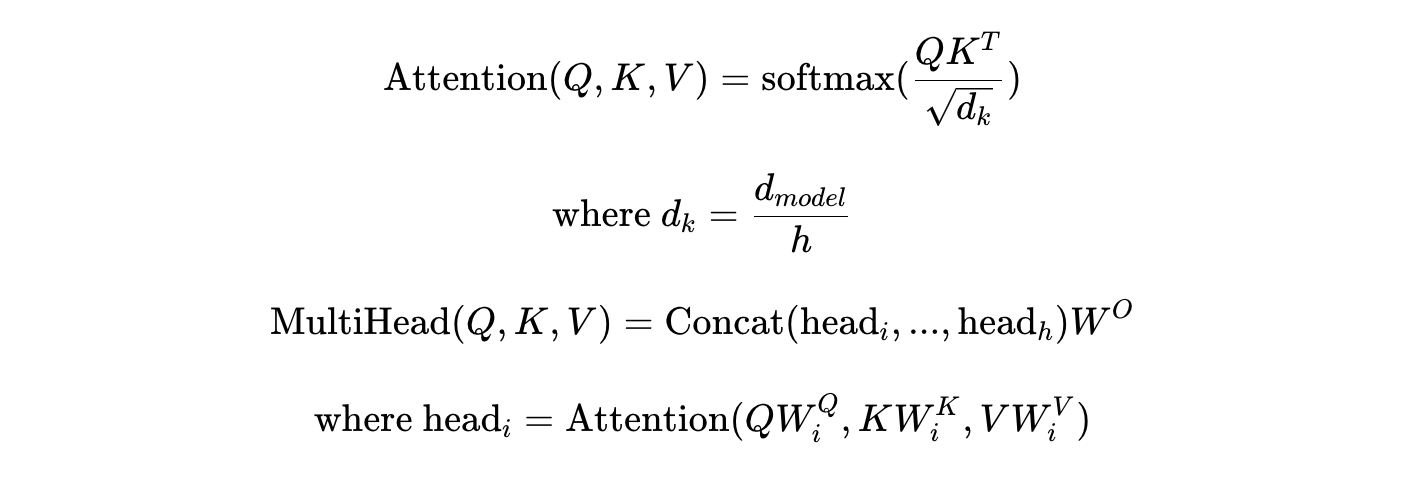
Multi-head self-attention is precisely what it sounds like: applying multiple “attention heads” to the same sequence. The exact self-attention mechanism is reapplied eight times to the same input sequence in parallel. For each attention head, its query, key, and value weight matrices are initialized randomly in hopes that each head will capture different types of information from the input sequence.

Each attention head produces a matrix of shape (length of sequence, 64); they are then concatenated along their second dimension, creating a matrix of shape (length of sequence, 8\*64). A linear projection is performed on this matrix to “combine” the knowledge of all the heads. The weight matrix used for the linear projection is trained through backpropagation along with the rest of the model.

To recap, and for those who like things presented numerically, the multi-head attention mechanism can be written as:



In the original Transformer paper, the authors used eight attention heads. However, later research showed that this might not be necessary. In the paper “[Analyzing Multi-Head Self-Attention: Specialized Heads Do the Heavy Lifting, the Rest Can Be Pruned](https://arxiv.org/abs/1905.09418),” Elena Voita et al. proposed that within the eight attention heads, there are three “specialized” attention heads which do most of the work. Specifically, these specialized attention heads’ roles have been hypothesized as such:



1. Positional head: this attention head is responsible for discovering the relationship between the relative positioning of words. Typically the highest attention score for each word points to an adjacent token.
2. Syntactic head: this attention head is responsible for analyzing the syntactic relationship between words.
3. Rare words: this attention head filters out words that infrequently appear in an input sequence, indicating that they could be more important than more commonly occurring words, such as “the” and “a”.

## Add, Norm, and Feed Forward Networks

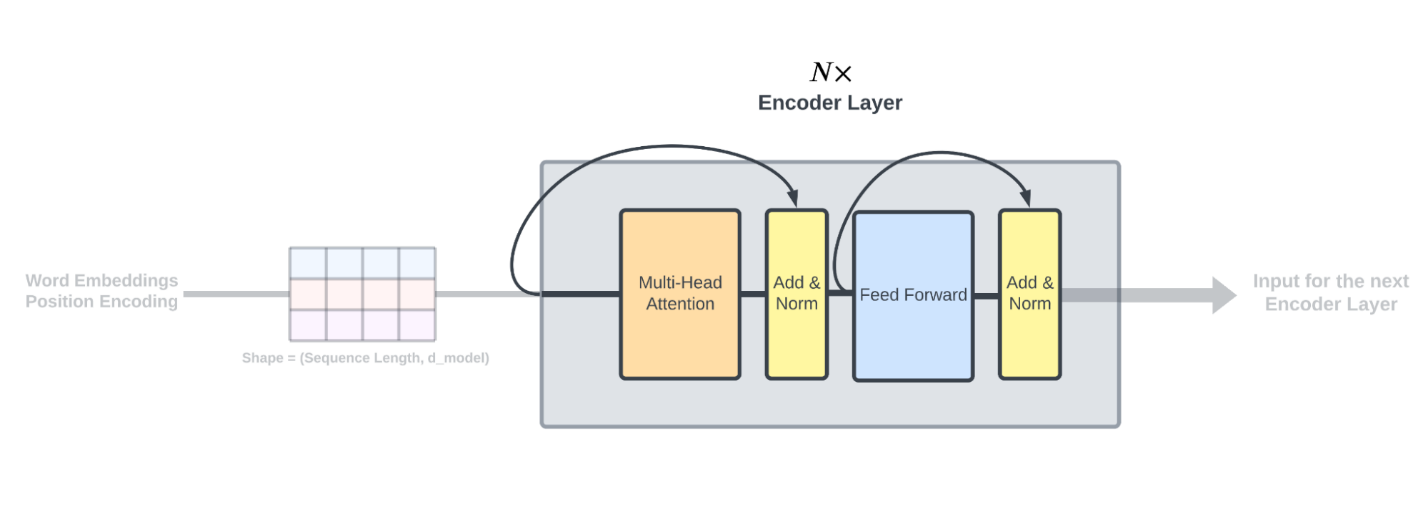
The output from the multi-head attention block is passed through a layer normalization component, which applies normalization to the inputs of the encoder layer. This normalization helps to improve the stability and speed of training for the transformer model by ensuring that the inputs have a consistent distribution. It also helps to reduce the effect of vanishing gradients, which can slow down the training.

The key difference between batch and layer normalization is the method that they use to normalize the inputs. In batch normalization, which is more common, each feature is normalized independently across the entire batch, while in layer normalization, each sample is normalized independently across the whole batch.

The layer normalization component also has a residual connection, which allows the input to be directly added to the output of the normalization layer. This residual connection helps to improve the flow of gradients through the model during training, which can further improve the stability and speed of training.

After the output has been normalized, it is passed through a shallow three-layer feed-forward network, which processes the output and generates the final encoded representation of the input sequence. The feed-forward network consists of two linear transformations with a ReLU activation function in between, which helps to capture complex relationships in the data.

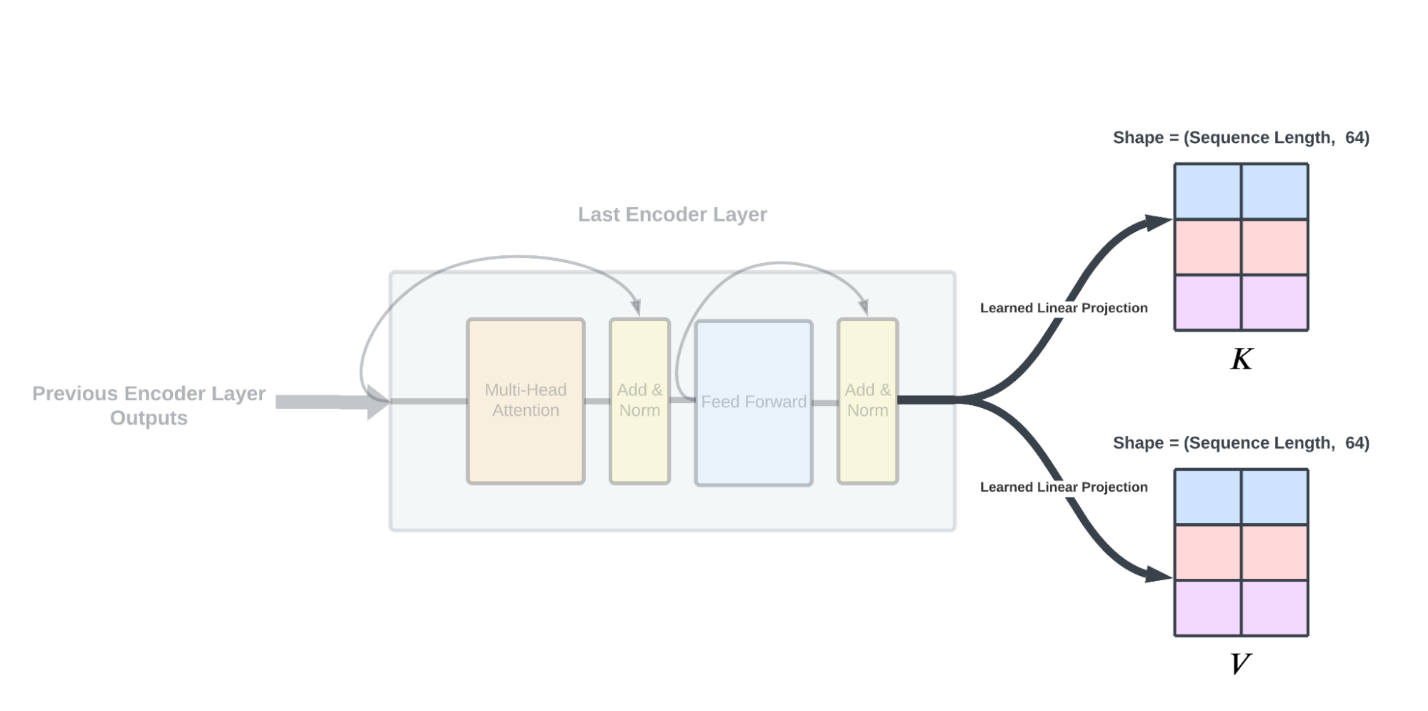
This final step completes the encoding layer of the transformer model. In particular, the input and output both have d\_model neurons, or 512, in the original paper, and the middle hidden layer has 2048 neurons. Another layer normalization and residual connection (Add & Norm) are employed after the shallow three-layer feed-forward network.



## Encoder Outputs

The output from the last encoder layer goes through another set of linear projections learned by backpropagation similar to those performed in the self-attention block, producing a key and a value matrix to be fed into the decoder.

After that exhaustive explanation of the encoder side of things, let's dig deeper into decoders.



## Decoder Training and Inference Scheme (a.k.a. Teacher Forcing)

The decoder processes its inputs in the same manner as the encoder - first through word embeddings and then through positional encoding. To briefly review, information flows through the decoder by initially feeding it a single "start of sentence" token. The decoder then generates the next output in the sequence, which is subsequently used as the input concatenated with the input from the previous time step.

This iterative process continues until the decoder produces an "end of sentence" token or the user-specified limit is reached. This approach allows the decoder to generate outputs autoregressively, one element at a time. Note that the decoder also receives the same key and value matrices produced by the encoder at every decoder layer at every prediction time step.

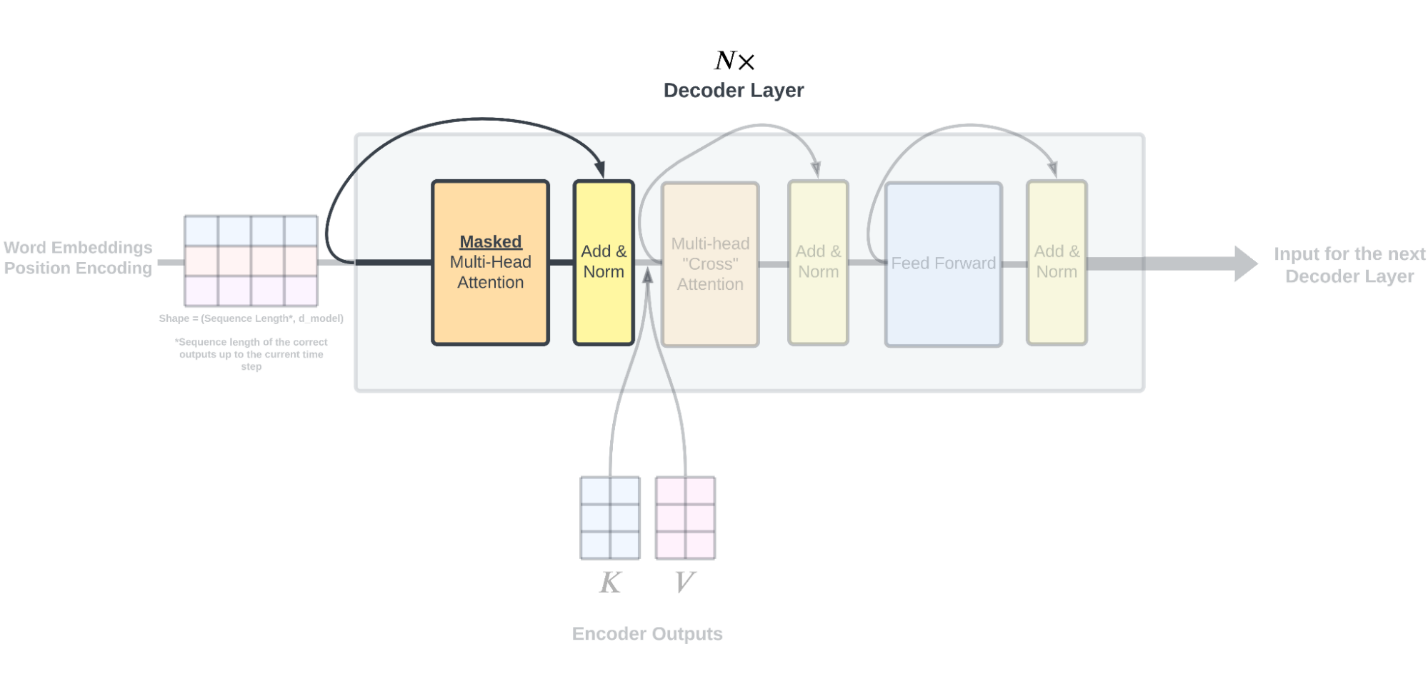
During training, we can take advantage of the fact that we have access to the full target sequence. Instead of training the decoder autoagressively, the input to a decoder at each time step is the expected output from the beginning of the sequence to the previous time step.

This allows the decoder to make predictions based on the known sequence rather than its own predictions, which can improve the overall performance of the model. This training technique is known as teacher forcing. Teacher forcing can also allow for parallelization of training as the output for each time step can be computed independently.

## Decoder Masked Attention

The decoder layer is similar in structure to the encoder layer, but it contains two attention blocks instead of one, and each of the attention blocks works slightly differently than the encoder’s.

The first attention blocked is the masked multi-head self-attention. This attention mechanism works the same way as the encoder's self-attention except when computing the attention matrix. Since this attention component calculates self-attention, it does not receive the key and value matrices from the encoder; those are for the next attention block.

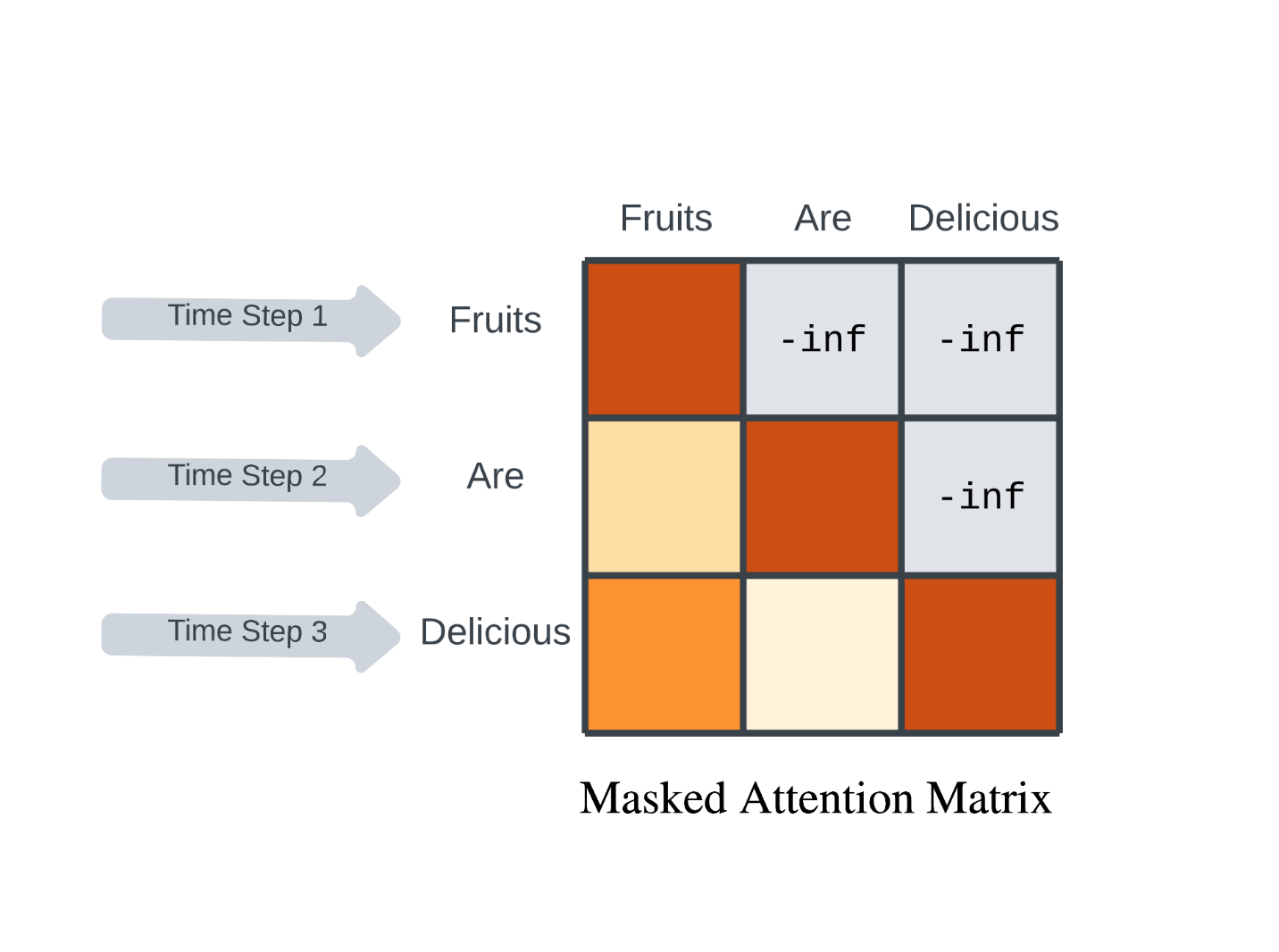


The first decoder self-attention block uses a mask to prevent the model from attending to future words in the sequence during inference. To understand why this is necessary, consider the self-attention matrix. In the first row of the matrix, each cell (other than the first one) represents how the first word attends to future words in the sequence. Since the decoder is trained using teacher forcing, the words that the decoder attends to in the first row of the matrix are the correct words. However, during inference, the model only has access to its own outputs, so these words will not necessarily be the correct ones.

In order to prevent the model from attending to future tokens in the output sequence during inference, a mask is applied to the attention matrix to mask out the attention values that are ahead of the current time step for each row. This helps the model to focus on the words it has generated so far and improve its predictions.

Consider the sentence "fruits are delicious," as shown in the figure above. The attention matrix is used to calculate the relationship between each word in the sentence and the rest of the words in the sequence. In a realistic scenario, when the model is calculating the relationship between the first word "fruits" and the rest of the sentence, it should not have access to the correct output words "are" and "delicious".

It is only when the model reaches the word "delicious" that it makes sense to compute attention against the other words in the sequence, as they appear earlier in the sentence. We can then assume that those earlier words are “predicted” by the model (they’re not since we’re using teacher forcing) just like during inference.



To formulate masking in terms of matrices, masking essentially applies a triangular mask to the top right portion of the attention matrix, setting those values to -inf thus preventing the model from using that information. Note that the query, key, and value matrices are still being computed the same way as those were in the encoder self-attention, the masking only comes into play before the softmax operation where the top-right triangular portion of the attention matrix is set to -inf. Once passed through the softmax operation, those values will be “squeezed” to 0, nullifying their influence.

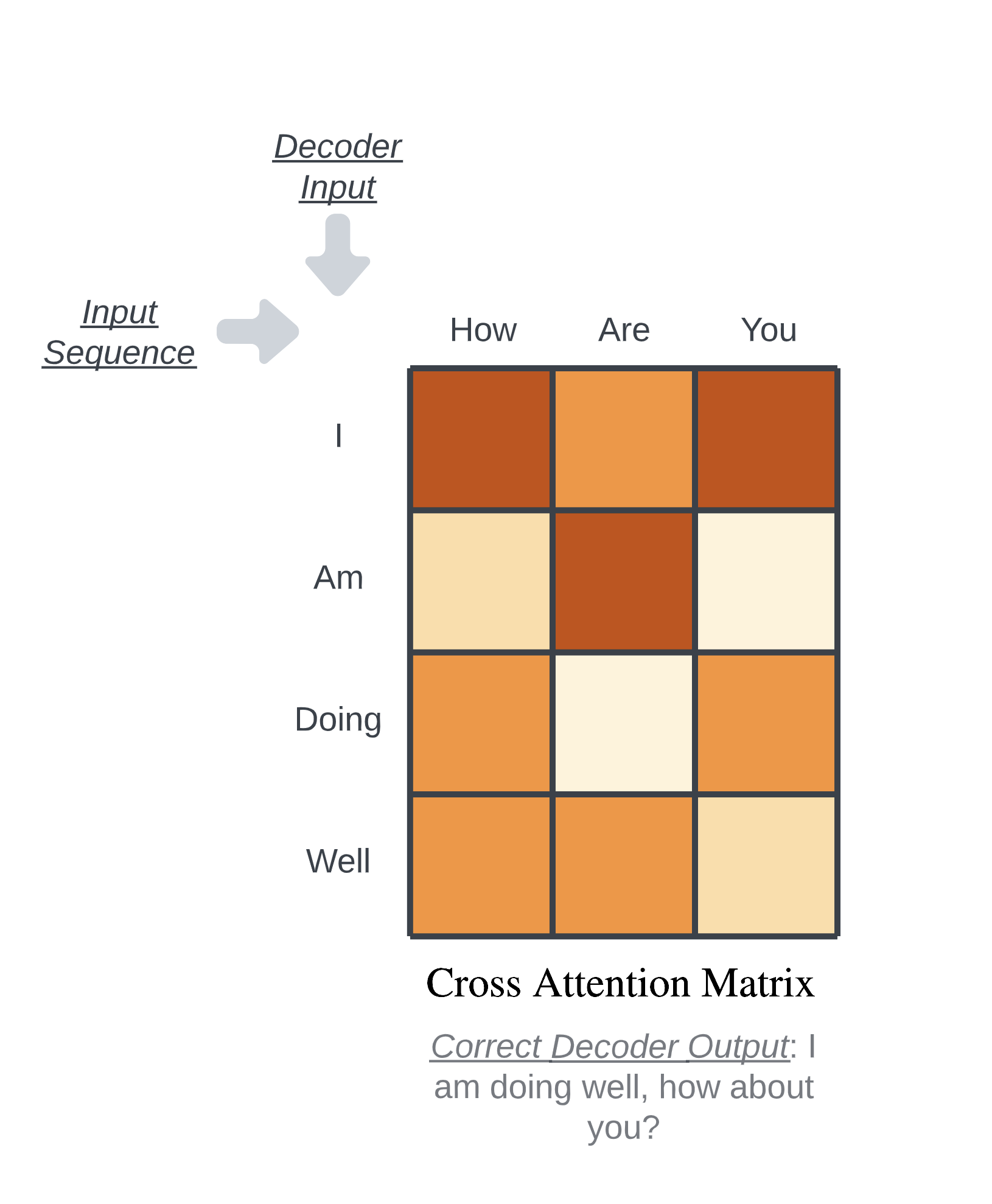
During the inference/prediction stage, the masking is removed during inference as the model is basing its predictions solely on its own outputs from previous time steps. Everything else about multi-head attention, add and layer normalization remains the same mechanism compared to the encoder self-attention.

## Decoder Cross Attention

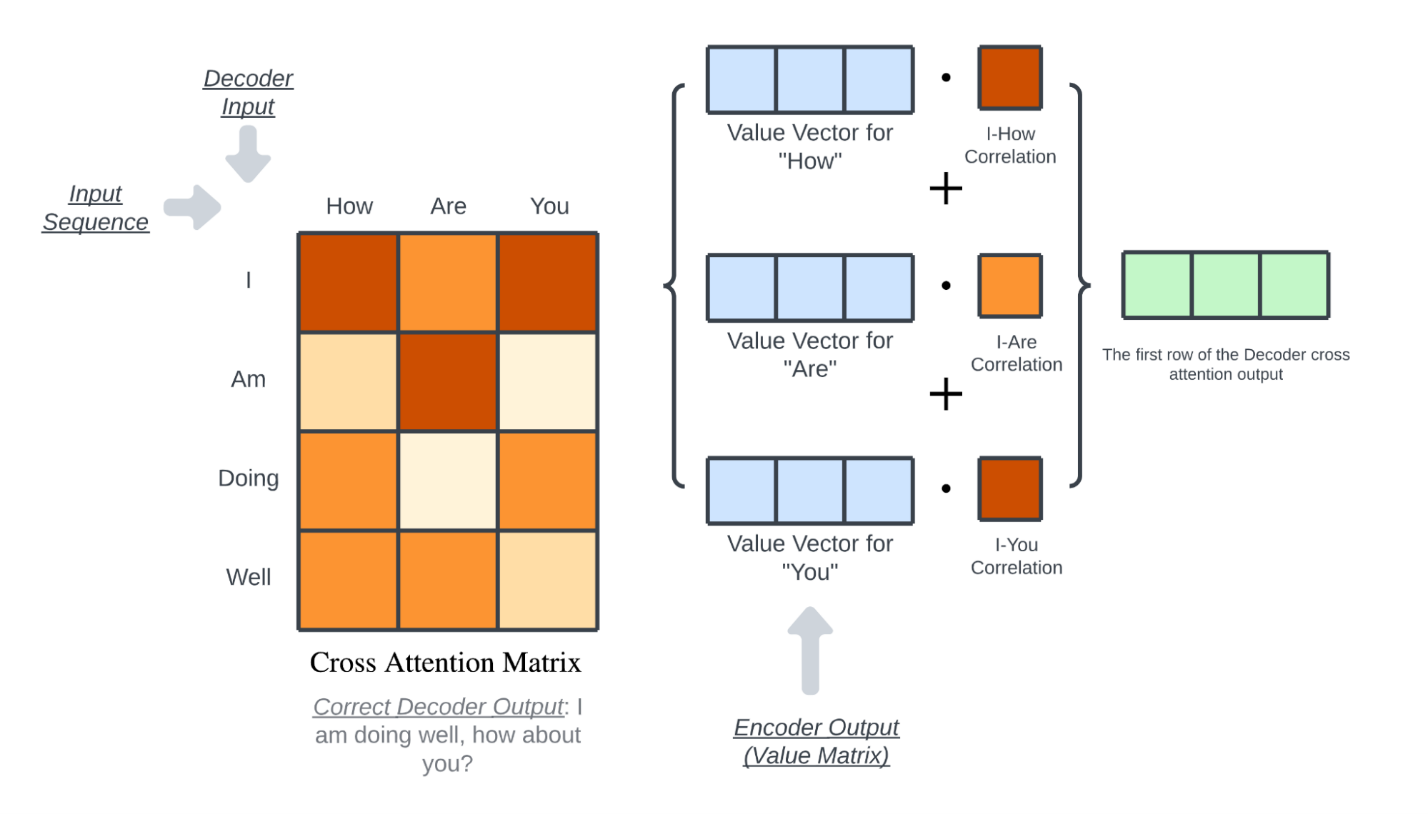
The decoder cross attention block is a crucial part of the transformer model. After the masked multi-head self-attention block and the add and layer normalization, the decoder uses another attention block to combine the output from the encoder with its own input. This is the first time that the encoder and decoder are dealing with sequences from each other's input. The decoder computes the query matrix from its own input, while the key and value matrices come from the encoder output.

Then, attention is performed as usual, but this time the focus is on finding relationships between the input sequence and the output generated so far by the decoder. This step allows the decoder to consider how the entire input sequence relates to what has already been outputted, and use this information to make more accurate predictions about the next word in the sequence.

The decoder cross attention mechanism can be thought of as a way of reconstructing the "word embedding" representation of the decoder input, but instead of using self-attention information, it uses information from the input sequence. Essentially, each row in the newly produced decoder "word embedding" is a weighted sum of the value vectors from the encoder, where the weights are determined by the relationship of each word in the input sequence to the given word from the decoder inputs.



The feed forward network functions the same way as the ones in the encoder layers. In the original paper, the decoder layer is stacked six times as well.



## Decoder Linear and Softmax

The last layer of the transformer decoder produces a matrix of size (length of sequence, d\_model). As mentioned previously, this output can be interpreted as an enhanced version of the original word embedding, incorporating insights from the 6 encoder and 6 decoder layers. This output is then fed through a learned linear transformation, which maps the matrix to a vector with length equal to the number of vocabularies. This vector is then passed through a softmax function to convert the values into probabilities. Each index of the vector represents a unique word, and the index with the highest probability is the model's prediction for the next word in the sequence.

## Training Scheme

Congratulations! That was the entire Transformer architecture. It may seem daunting at first, but when broken down into individual components, it's simply a combination of weights and matrices working together to analyze language in a similar way to humans. Here are some additional bits and pieces of information about the general training scheme of the Transformer that weren't mentioned in the explanation of the architecture:

* There are various methods for training a Transformer, and the teacher forcing approach is just the most commonly used one.
* The typical loss function used is the cross-entropy loss, which compares the probability distribution output by the model to the ideal probability distribution. The ideal distribution has a value of 1.0 for the element representing the correct output, and all other elements are zeroed out.
* The learned linear projection that transforms tokenized text into word embeddings is shared between the encoder and decoder.

## Conclusion

While this article has provided a detailed overview of the Transformer’s key components and how it works, there is still much more to explore and learn about this fascinating model. Its continued development and application in various domains hold the promise of exciting advancements in the future.

## What’s a Transformer Model?

A transformer model is a neural network that learns context and thus meaning by tracking relationships in sequential data like the words in this sentence.

Transformer models apply an evolving set of mathematical techniques, called attention or self-attention, to detect subtle ways even distant data elements in a series influence and depend on each other.

First described in [a 2017 paper](https://arxiv.org/abs/1706.03762) from Google, transformers are among the newest and one of the most powerful classes of models invented to date. They’re driving a wave of advances in machine learning some have dubbed transformer AI.

Stanford researchers called transformers “foundation models” in an [August 2021 paper](https://arxiv.org/pdf/2108.07258.pdf) because they see them driving a paradigm shift in AI. The “sheer scale and scope of foundation models over the last few years have stretched our imagination of what is possible,” they wrote.

## What Can Transformer Models Do?

Transformers are translating text and speech in near real-time, opening meetings and classrooms to diverse and hearing-impaired attendees.

They’re helping researchers understand the chains of genes in DNA and amino acids in proteins in ways that can speed drug design.

**What are Transformers?**

Transformers are essentially neural networks. Neural networks that specialize in learning context from the data.

But what makes them special is the presence of mechanisms that eliminate the need for **labeled datasets**and**convolution or recurrence** in the network.

**What are these special mechanisms?**

There are many. But the two mechanisms that are truly the force behind the transformers are attention weighting and feed-forward networks (FFN).

**What is attention-weighting?**

Attention-weighting is a technique by which the model learns which part of the incoming sequence needs to be focused on. Think of it as the ‘Eye of Sauron’ scanning everything at all times and throwing light on the parts that are relevant.

*Fun-fact: Apparently, the researchers had almost named the Transformer model ‘Attention-Net’, given Attention is such a crucial part of it.*

**What is FFN?**

In the context of transformers, FFN is essentially a regular multilayer perceptron acting on a batch of independent data vectors. Combined with attention, it produces the correct ‘position-dimension’ combination.

**How do Attention and FFN work?**

So, without further ado, let’s dive into how **attention-weighting** and **FFN** make transformers so powerful.

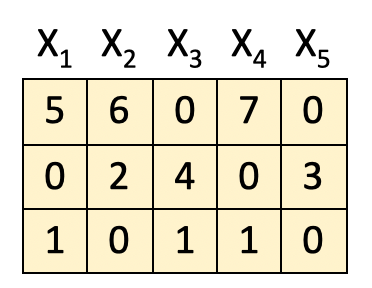
This discussion is based on Prof. Tom Yeh’s wonderful AI by Hand Series on [Transformers](https://lnkd.in/g39jcD7j) . (All the images below, unless otherwise noted, are by Prof. Tom Yeh from the above-mentioned LinkedIn posts, which I have edited with his permission.)

So here we go:

The key ideas here : **attention weighting and feed-forward network (FFN)**.

Keeping those in mind, suppose we are given:

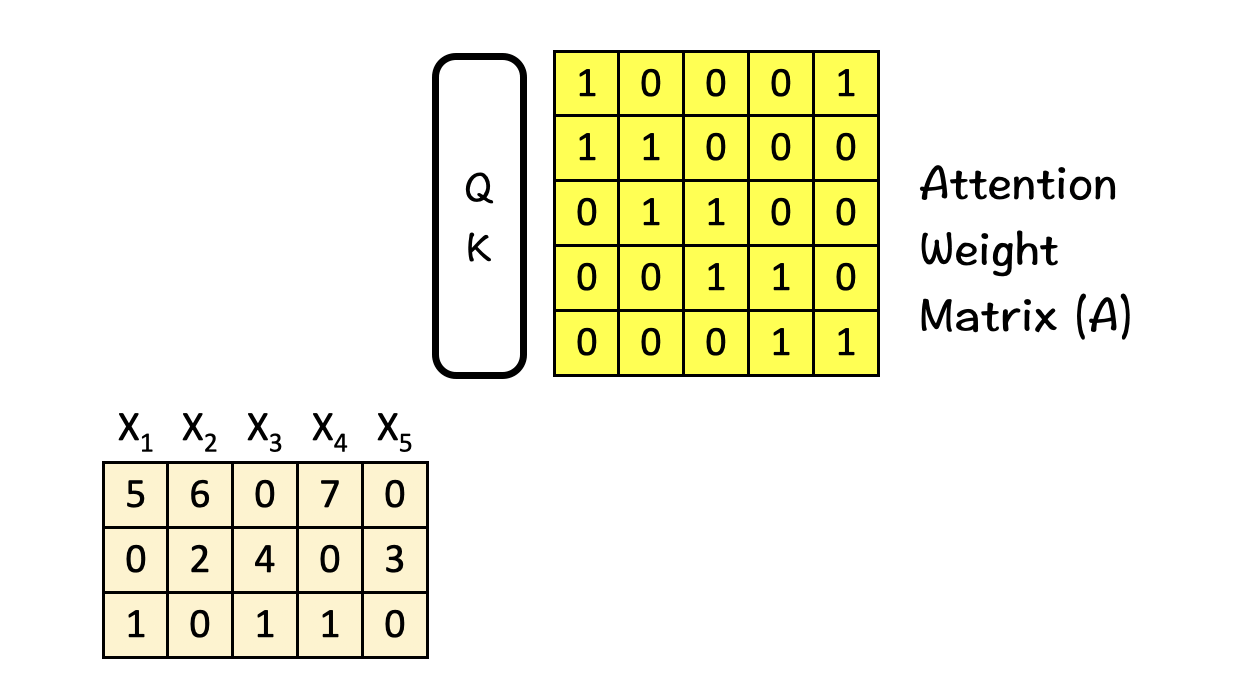
* 5 input features from a previous block (A 3x5 matrix here, where X1, X2, X3, X4 and X5 are the features and each of the three rows denote their characteristics respectively.)



[1] **Obtain attention weight matrix A**

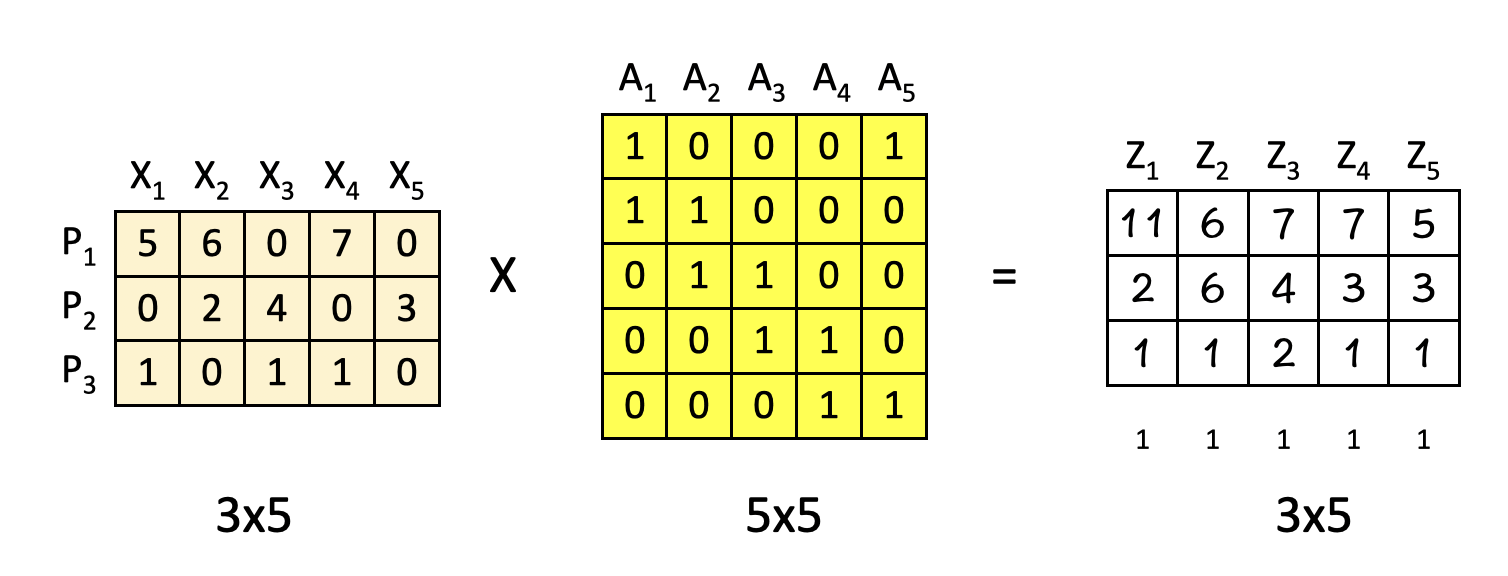
The first step in the process is to obtain the **attention weight matrix A**. This is the part where the self-attention mechanism comes to play. What it is trying to do is find the most relevant parts in this input sequence.

We do it by feeding the input features into the query-key (QK) module. For simplicity, the details of the QK module are not included here.



[2] **Attention Weighting**

Once we have the **attention weight matrix A (5x5)**, we multiply the input features (3x5) with it to obtain the **attention-weighted features Z**.



The important part here is that the features here are combined **based on their positions** P1, P2 and P3 i.e. **horizontally**.

To break it down further, consider this calculation performed row-wise:

P1 X A1 = Z1 → Position [1,1] = 11

P1 X A2 = Z2 → Position [1,2] = 6

P1 X A3 = Z3 → Position [1,3] = 7

P1 X A4 = Z4 → Position [1,4] = 7

P1 X A5 = Z5 → Positon [1,5] = 5

.

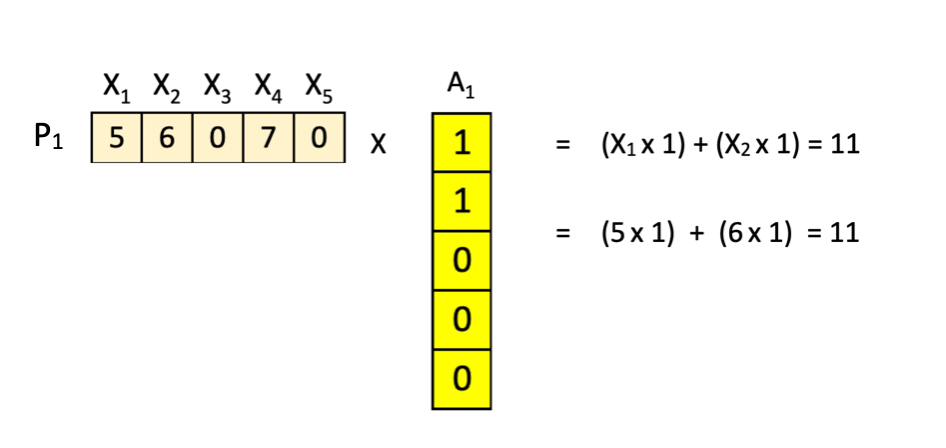
.

.

P2 X A4 = Z4 → Position [2,4] = 3

P3 X A5 = Z5 →Position [3,5] = 1

As an example:



It seems a little tedious in the beginning but follow the multiplication row-wise and the result should be pretty straight-forward.

Cool thing is the way our attention-weight matrix**A** is arranged, the new features **Z** turn out to be the combinations of **X**as below:

Z1 = X1 + X2

Z2 = X2 + X3

Z3 = X3 + X4

Z4 = X4 + X5

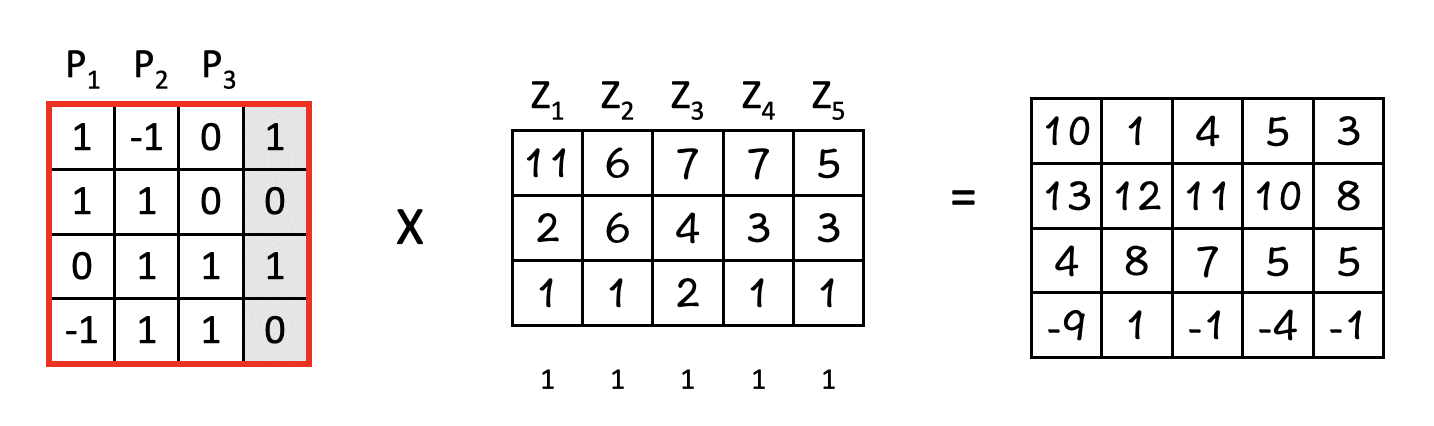
Z5 = X5 + X1

(Hint : Look at the positions of 0s and 1s in matrix **A**).

[3] **FFN : First Layer**

The next step is to feed the attention-weighted features into the feed-forward neural network.

However, the difference here lies in **combining the values across dimensions** as opposed to positions in the previous step. It is done as below:



What this does is that it looks at the data from the other direction.

**- In the attention step, we combined our input on the basis of the original features to obtain new features.**

**- In this FFN step, we consider their characteristics i.e. combine features vertically to obtain our new matrix.**

*Eg: P1(1,1) \* Z1(1,1)*

*+ P2(1,2) \* Z1 (2,1)*

*+ P3 (1,3) \* Z1(3,1) + b(1) = 11, where b is bias.*

Once again element-wise row operations to the rescue. Notice that here the number of dimensions of the new matrix is increased to 4 here.

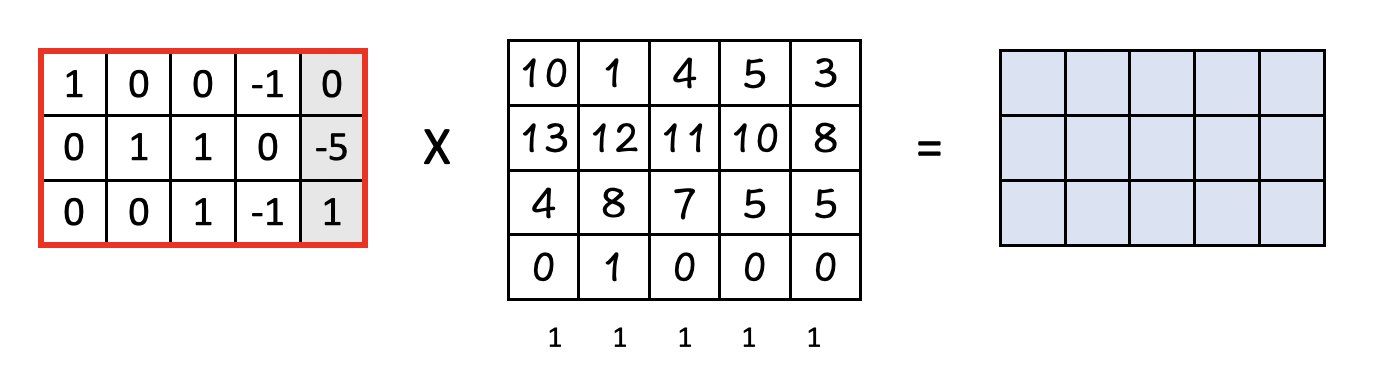
[4] **ReLU**

Our favorite step : ReLU, where the negative values obtained in the previous matrix are returned as zero and the positive value remain unchanged.



[5] **FFN : Second Layer**

Finally we pass it through the second layer where the dimensionality of the resultant matrix is reduced from 4 back to 3.



The output here is ready to be fed to the next block (see its similarity to the original matrix) and the entire process is repeated from the beginning.

**The two key things to remember here are:**

1. **The attention layer combines across positions (horizontally).**
2. **The feed-forward layer combines across dimensions (vertically).**

And this is the secret sauce behind the power of the transformers — the ability to analyze data from different directions.

To summarize the ideas above, here are the key points:

1. The transformer architecture can be perceived as a combination of the attention layer and the feed-forward layer.
2. The **attention layer combines the features** to produce a new feature. E.g. think of combining two robots Robo-Truck and Optimus Prime to get a new robot Robtimus Prime.
3. The **feed-forward (FFN) layer combines the parts or the characteristics** of the a feature to produce new parts/characteristics. E.g. wheels of Robo-Truck and Ion-laser of Optimus Prime could produce a wheeled-laser.

**The ever powerful Transformers**

Neural networks have existed for quite some time now. Convolutional Neural Networks (CNN) and Recurrent Neural Networks (RNN) had been reigning supreme but things took quite an eventful turn once Transformers were introduced in the year 2017. And since then, the field of AI has grown at an exponential rate — with new models, new benchmarks, new learnings coming in every single day. And only time will tell if this phenomenal idea will one day lead the way for something even bigger — a real ‘Transformer’.

What is a Transformer 2

* Let's kick things off with a very high-level preview of how data flows through a transformer. We will spend much more time motivating and interpreting and expanding on the details of each step, but in broad strokes. When one of these chatbots generates a given word, here's what's going on under the hood.
* First, the input is broken up into a bunch of little pieces. These pieces are called tokens. And in the case of text, these tend to be words or little pieces of words or other common character combinations. If images or sound are involved, then tokens could be little patches of that image or little chunks of that sound.
* Each one of these tokens is then associated with a vector, meaning some list of numbers, which is meant to somehow encode the meaning of that piece. If you think of these vectors as giving coordinates in some very highdimensional space, words with similar meanings tend to land on vectors that are close to each other in that space.
* This sequence of vectors then passes through an operation that's known as an attention block. And this allows the vectors to talk to each other and pass information back and forth to update their values. For example, the meaning of the word model in the phrase a machine learning model is different from its meaning in the phrase a fashion model.
* The attention block is what's responsible for figuring out which words in the context are relevant to updating the meanings of which other words and how exactly those meanings should be updated. And again, whenever I use the word meaning, this is somehow entirely encoded in the entries of those vectors. After that, these vectors pass through a different kind of operation.
* And depending on the source that you're reading, this will be referred to as a multi-layer perceptron or maybe a feed forward layer. And here, the vectors don't talk to each other. They all go through the same operation in parallel. And while this block is a little bit harder to interpret, later on, we'll talk about how the step is a little bit like asking a long list of questions about each vector and then updating them based on the answers to those questions.
* All of the operations in both of these blocks look like a giant pile of matrix multiplications. And our primary job is going to be to understand how to read the underlying matrices. I'm glossing over some details about some normalization steps that happen in between. But this is after all a highle preview.
* After that, the process essentially repeats. You go back and forth between attention blocks and multi-layer perceptron blocks until at the very end the hope is that all of the essential meaning of the passage has somehow been baked into the very last vector in the sequence. We then perform a certain operation on that last vector that produces a probability distribution over all possible tokens, all possible little chunks of text that might come next.
* And like I said, once you have a tool that predicts what comes next given a snippet of text, you can feed it a little bit of seed text and have it repeatedly play this game of predicting what comes next, sampling from the distribution, appending it, and then repeating over and over. Some of you in the know may remember how long before chat GPT came into the scene, this is what early demos of GPT3 looked like.
* You would have it autocomplete stories and essays based on an initial snippet

What is a Transformer 5

* For example, those 175 billion weights in GPT3 are organized into just under 28,000 distinct matrices. Those matrices in turn fall into eight different categories. And what you and I are going to do is step through each one of those categories to understand what that type does.
* As we go through, I think it's kind of fun to reference the specific numbers from GPT3 to count up exactly where those 175 billion come from. Even if nowadays there are bigger and better models, this one has a certain charm as the first large language model to really capture the world's attention outside of ML communities.
* Also, practically speaking, companies tend to keep much tighter lips around the specific numbers for more modern networks. I just want to set the scene going in that as you peak under the hood to see what happens inside a tool like Chat GPT, almost all of the actual computation looks like matrix vector multiplication.
* There's a little bit of a risk getting lost in the sea of billions of numbers, but you should draw a very sharp distinction in your mind between the weights of the model, which I'll always color in blue or red, and the data being processed, which I'll always color in gray. The weights are the actual brains. They are the things learned during training, and they determine how it behaves.
* The data being processed simply encodes whatever specific input is fed into the model for a given run, like an example snippet of text. With all of that as foundation, let's dig into the first step of this text processing example, which is to break up the input into little chunks and turn those chunks into vectors.
* I mentioned how those chunks are called tokens, which might be pieces of words or punctuation, but every now and then in this chapter, and especially in the next one, I'd like to just pretend that it's broken more cleanly into words. Because we humans think in words, this will just make it much easier to reference little examples and clarify each step.
* The model has a predefined vocabulary, some list of all possible words, say 50,000 of them. And the first matrix that we'll encounter, known as the embedding matrix, has a single column for each one of these words. These columns are what determines what vector each word turns into. In that first step, we label it we like all the matrices we see, its values begin random, but they're going to be learned based on data.
* Turning words into vectors was common practice in machine learning long before transformers, but it's a little weird if you've never seen it before, and it sets the foundation for everything that follows. So, let's take a moment to get familiar with it. We often call this embedding a word, which invites you to think of these vectors very geometrically, as points in some high dimensional space.
* Visualizing a list of three numbers as coordinates for points in 3D space would be no problem, but word embeddings tend to be much, much higher dimensional. In GPT3, they have 12,288 dimensions. And as you'll see, it matters to work in a space that has a lot of distinct directions. In the same way that you could take a two-dimensional slice through a 3D space and project all the points onto that slice, for the sake of animating word embeddings that a simple model is giving me, I'm going to do an analogous thing by choosing a three-dimensional slice
* through this very highdimensional space and projecting the word vectors down onto that and displaying the results. The big idea here is that as a model tweaks and tunes its weights to determine how exactly words get embedded as vectors during training, it tends to settle on a set of embeddings where directions in the space have a kind of semantic meaning.
* For the simple wordtovector model I'm running here, if I run a search for all the words whose embeddings are closest to that of tower, you'll notice how they all seem to give very similar towerish vibes. And if you want to pull up some Python and play along at home, this is the specific model that I'm using to make the animations.
* It's not a transformer, but it's enough to illustrate the idea that directions in the space can carry semantic meaning

What is a Transformer 6

* Classic example of this is how if you take the difference between the vectors for woman and man, something you would visualize as a little vector in the space connecting the tip of one to the tip of the other. It's very similar to the difference between king and queen. So let's say you didn't know the word for a female monarch.
* You could find it by taking king, adding this woman minus man direction, and searching for the embeddings closest to that point. at least kind of. Despite this being a classic example for the model I'm playing with, the true embedding of queen is actually a little farther off than this would suggest. Presumably because the way that queen is used in training data is not merely a feminine version of king.
* When I played around, family relations seem to illustrate the idea much better. The point is, it looks like during training, the model found it advantageous to choose embeddings such that one direction in this space encodes gender information. Another example is that if you take the embedding of Italy and you subtract the embedding of Germany and then you add that to the embedding of Hitler, you get something very close to the embedding of Mussolini.
* It's as if the model learned to associate some directions with Italianness and others with World War II axis leaders. Maybe my favorite example in this vein is how in some models if you take the difference between Germany and Japan and you add it to sushi, you end up very close to broadwurst. Also, in playing this game of finding nearest neighbors, I was very pleased to see how close cat was to both beast and monster.
* One bit of mathematical intuition that's helpful to have in mind, especially for the next chapter, is how the dotproduct of two vectors can be thought of as a way to measure how well they align. Computationally, dotproducts involve multiplying all the corresponding components and then adding the results, which is good since so much of our computation has to look like weighted sums.
* Geometrically, the dotproduct is positive when vectors point in similar directions. It's zero if they're perpendicular, and it's negative whenever they point in opposite directions. For example, let's say you were playing with this model and you hypothesize that the embedding of cats minus cat might represent a sort of plurality direction in this space.
* To test this, I'm going to take this vector and compute its dotproduct against the embeddings of certain singular nouns and compare it to the dotproducts with the corresponding plural nouns. If you play around with this, you'll notice that the plural ones do indeed seem to consistently give higher values than the singular ones, indicating that they align more with this direction.
* It's also fun how if you take this dotproduct with the embeddings of the words 1 2 3 and so on, they give increasing values. So, it's as if we can quantitatively measure how plural the model finds a given word. Again, the specifics for how words get embedded is learned using data. This embedding matrix whose columns tell us what happens to each word is the first pile of weights in our model.
* And using the GPT3 numbers, the vocabulary size specifically is 50,257. And again, technically this consists not of words per se, but of tokens and the embedding dimension is 12,288. Multiplying those tells us this consists of about 617 million weights. Let's go ahead and add this to a running tally, remembering that by the end we should count up to 175 billion.
* In the case of transformers, you really want to think of the vectors in this embedding space as not merely representing individual words. For one thing, they also encode information about the position of that word, which we'll talk about later. But more importantly, you should think of them as having the capacity to soak in context. A vector that started its life as the embedding of the word king, for example, might progressively get tugged and pulled by various blocks in this network, so that by the end it points in a much more specific and nuanced
* direction that somehow encodes that it was a king who lived in Scotland and who had achieved his post after murdering the previous king and who's being described in Shakespearean language.