Session Summary

Introduction to Transformers

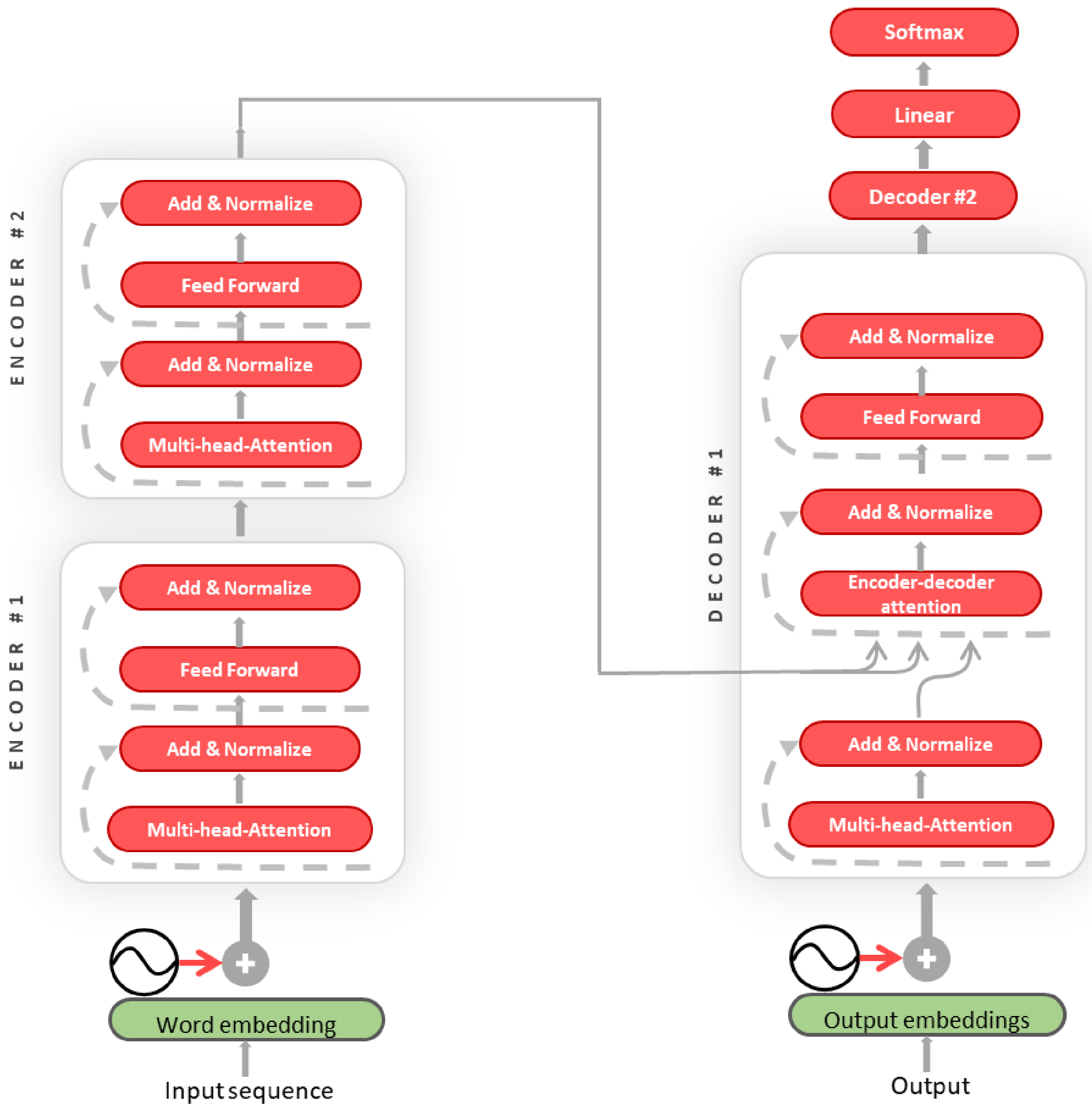
The Transformer architecture was designed to tackle the challenges associated with attention posed by traditional Seq2Seq models. Some of these challenges were increased computational and memory needs when working with long sequences.

The Transformer architecture takes advantage of the best features of two proven methods: attention mechanisms for capturing relative dependencies and CNNs for parallel processing. By combining these concepts, the Transformer can analyze input sequences in parallel and generate a context vector that reflects their relative dependencies.

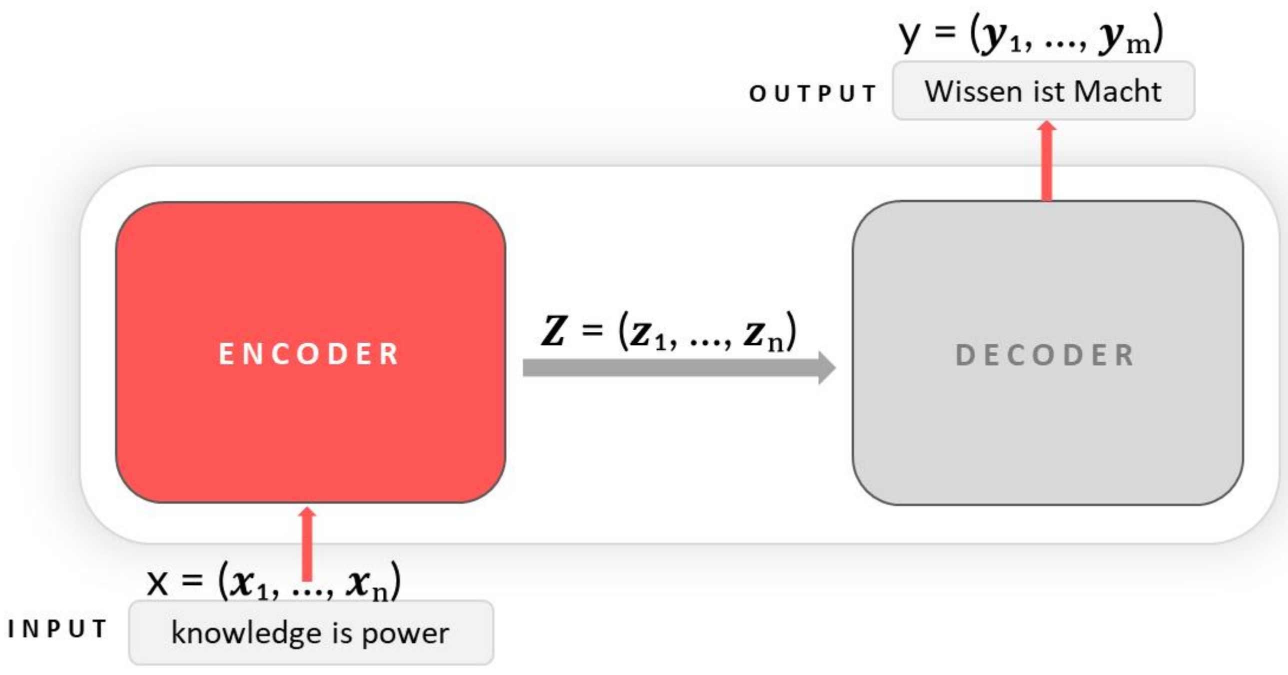
Convolution
Attention
Transformer

Understanding the Transformer Architecture

The Google Brain team introduced the transformer architecture in their paper **Attention Is All You Need** and sparked a shift in the advancement of natural language understanding (NLU).

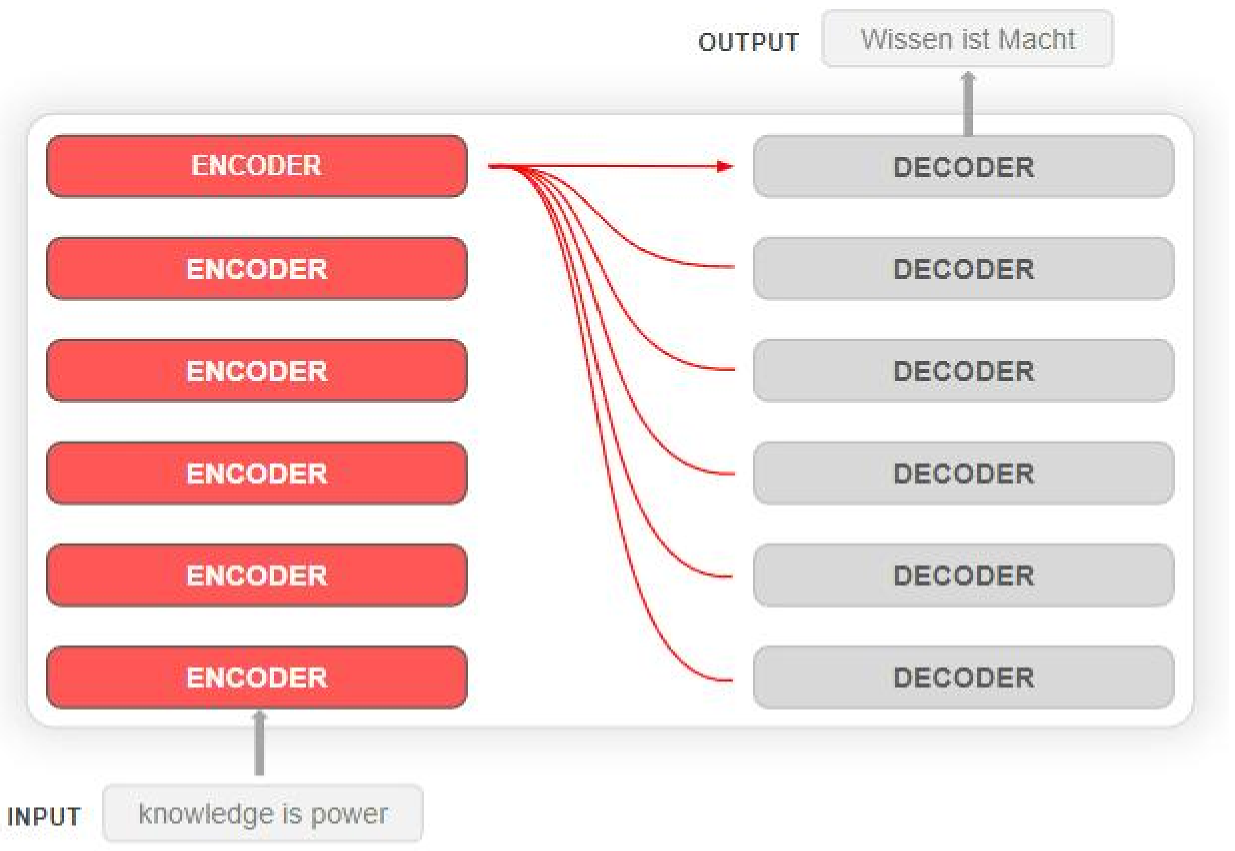


If you relate the entire transformer architecture to a black box, the base architecture corresponds to a sequence-to-sequence model consisting of two main building blocks: an encoder and a decoder.



Given an input representation sequence x=$(x_1, \ldots, x_n)$, the encoder block generates a sequence of context vectors $Z = (z_1, \ldots, z_n)$. And then, the decoder block generates the output sequence y = $(y_1, \ldots, y_m)$ using this contextual information (Z).

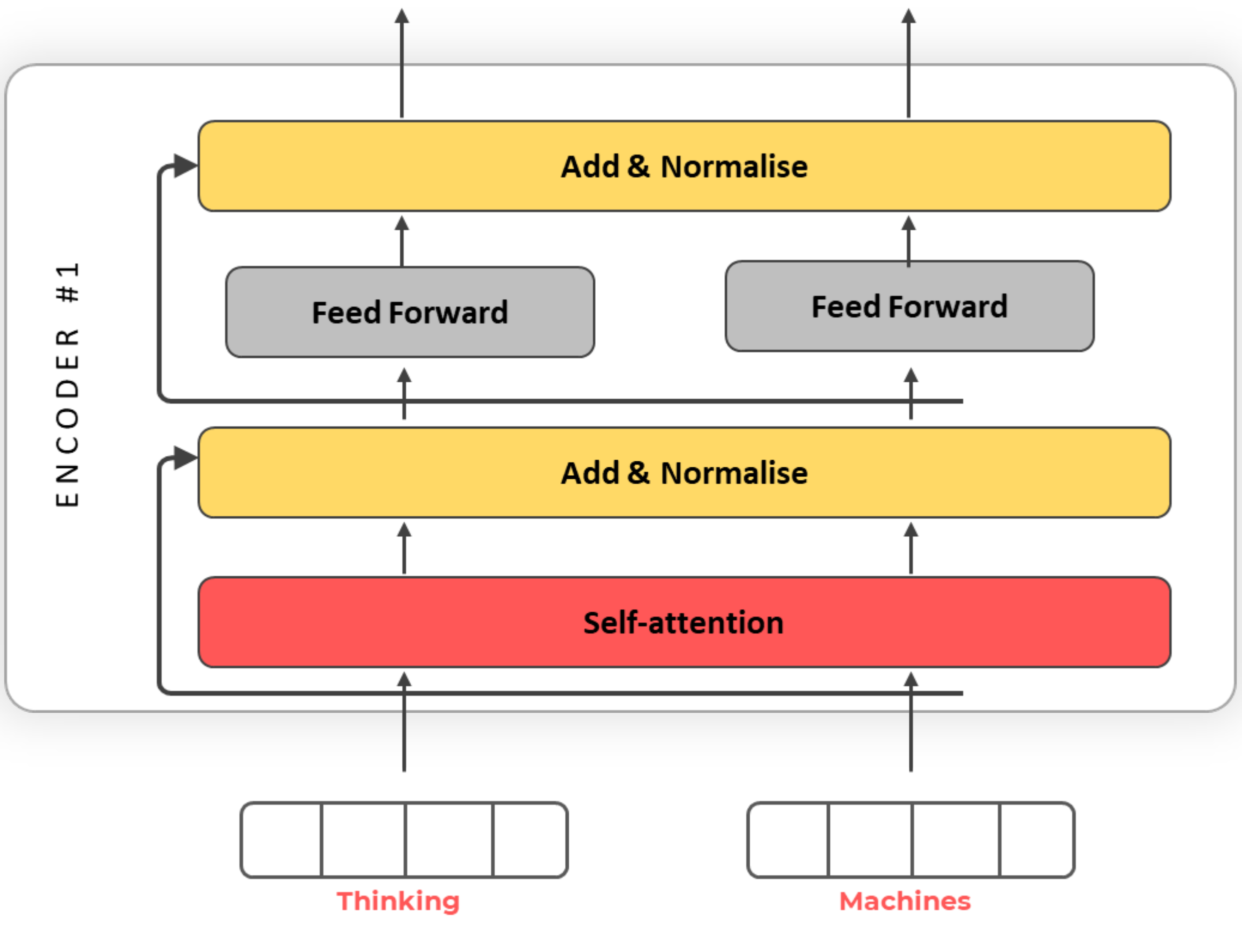
A transformer has multiple encoders and decoders stacked on top of each other (generally, a stack of 6 identical layers).



Components of Transformer Architecture: Encoder

Each encoder layer comprises of the following sublayers/components:

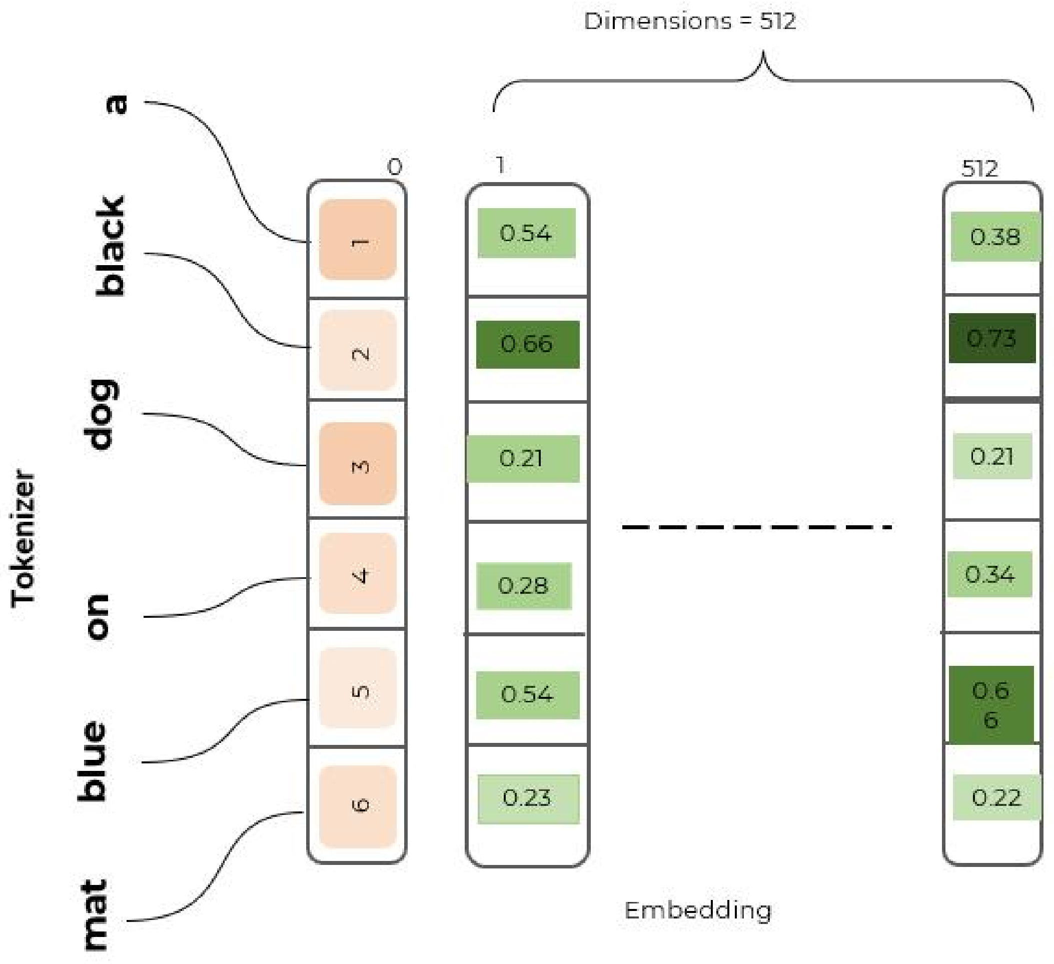
1. Positional encoding
2. Multi-head attention block
3. Normalization layer
4. Position-wise feedforward NNs



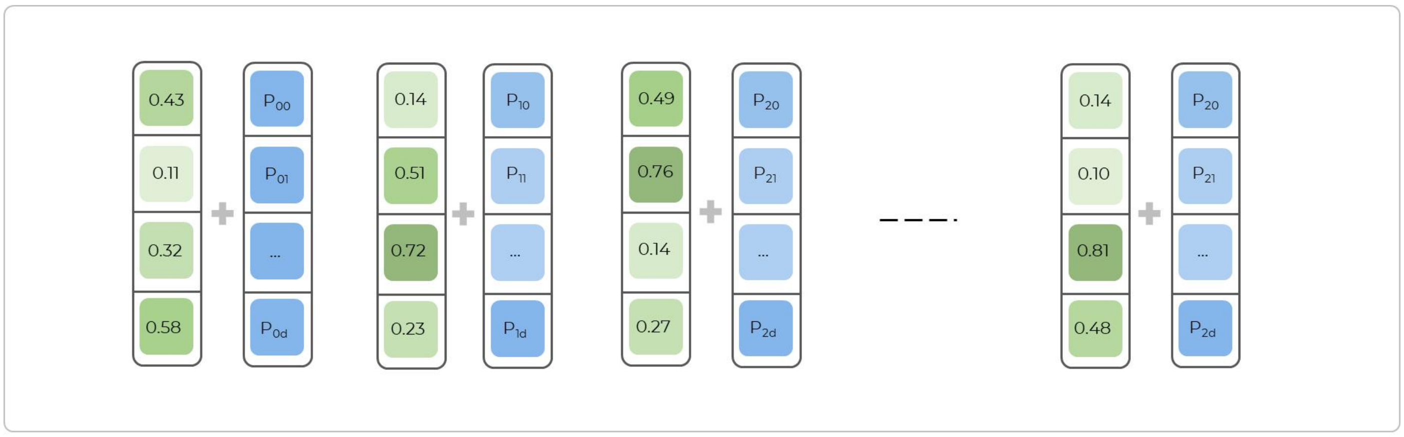
Moving on, you will quickly take a look at each of these components/sublayers.

Positional Encoding

The input sentence given by us is converted into **word embeddings** and then fed to an encoder as input. **Word embedding** is the process of converting a sentence into a numerical representation that a machine can understand. To store the relevant information, the embedding layer generates a matrix that contains the embeddings of all the tokens. This matrix is usually of the shape **(vocab_size, embedding_dim)**. Here, **vocab_size** is the number of unique tokens, and **embedding_dim** is the number of dimensions in the embedding space. Each dimension in the embedding space represents a specific feature of the token.



However, if these input tokens are provided in parallel, the information about their positions will be lost. To solve this problem, we pass positional embedding along with the input embedding to ensure that the information about the position of every token is also considered.



position vectors are added to the embedding vectors.

The author of the Transformer model suggested the use of the sine and cosine waves of different frequencies for storing positional encoding details.

$$PE_{(pos,2i)} = sin\left(\frac{pos}{10000^{\frac{2i}{d}}}\right)$$

$$PE_{(pos,2i+1)} = cos\left(\frac{pos}{10000^{\frac{2i}{d}}}\right)$$

Here, **pos** is the position of the word, **i** is the **ith** dimension of the word embedding, and **d/dmodel** is the **number of dimensions** in the embeddings

Self-Attention

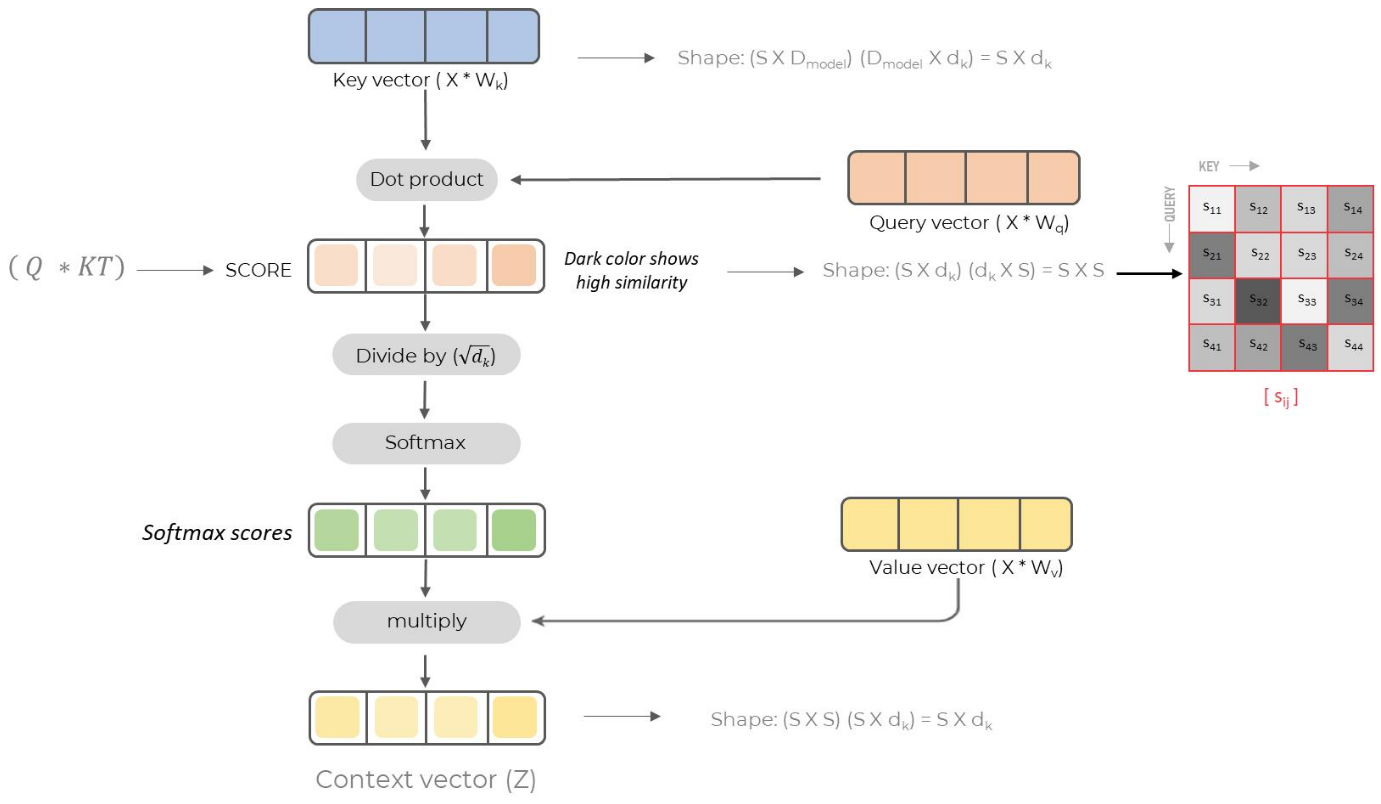
The self-attention mechanism used by the encoder allows it to selectively focus on important parts of the input while processing it. This allows the model to determine the relevance of different sections of the input and make informed decisions when translating languages.

The self-attention process starts by generating three distinct vectors from the input vector. So, for each word, we create the following vectors:

- Query vector (Q)
- Key vector (K)
- Value vector (V)

These abstractions of the input vectors are produced by multiplying the input embedding with the matrices Wq, Wv, and Wk. Here are the operations we perform to arrive at the context vector:

- A dot product for **each query (Q)** with **all the keys (K)** is calculated. This gives a square matrix, which captures the **similarity scores** of one token in relation to the other tokens in the input sequence.
- A **scaling factor (1/d_k)** is to shrink the result of the dot product. This ensures that the softmax operation following it will produce small gradients, which can ultimately lead to a vanishing gradient problem.
- The distribution of scores generated from the previous step should add up to 1. So, we perform a **softmax operation** to normalize the result in a limited range, giving you each vector's attention weights.
- The weights obtained are multiplied with the value vector (V) to produce the **context vector**.



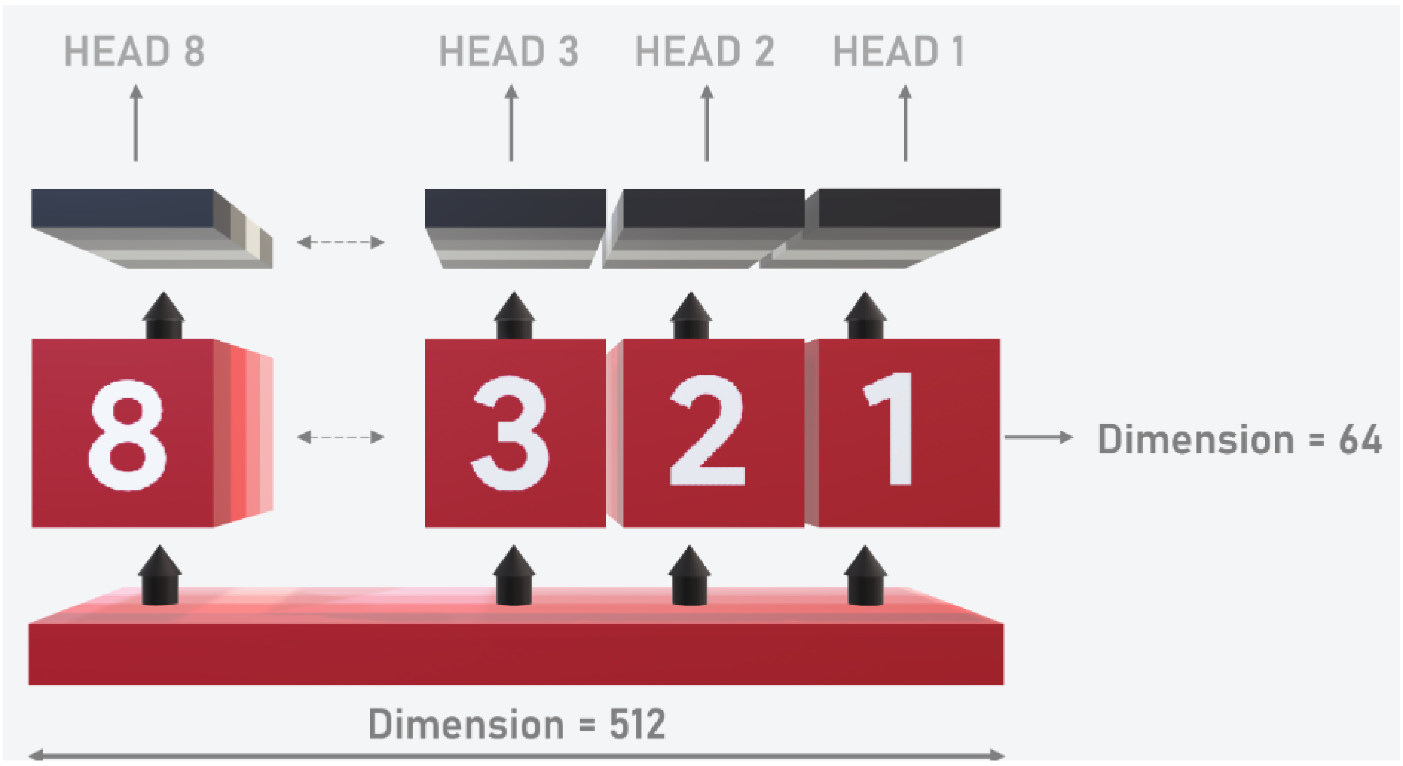
Key vector (X * W_k)
Shape: (S X D_{model}) (D_{model} X d_k) = S X d_k
Dot product
Query vector (X * W_q)
KEY
QUERY
s_{11} s_{12} s_{13} s_{14}
s_{21} s_{22} s_{23} s_{24}
s_{31} s_{32} s_{33} s_{34}
s_{41} s_{42} s_{43} s_{44}
$(Q * KT)$
SCORE
Dark color shows high similarity
Shape: (S X d_k) (d_k X S) = S X S
[s_{ij}]
Divide by ($\sqrt{d_k}$)
Softmax
Softmax scores
Value vector (X * W_v)
multiply
Shape: (S X S) (S X d_k) = S X d_k
Context vector (Z)

The steps we discussed above can be formulated as shown below.

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

Multi-Head Attention

According to the original paper, the use of eight attention heads, referred to as multi-head attention, is recommended. Multi-head attention enhances the speed of context vector calculation by providing multiple parallel attention layers, each of which can capture different aspects of the input and improve the model's performance.

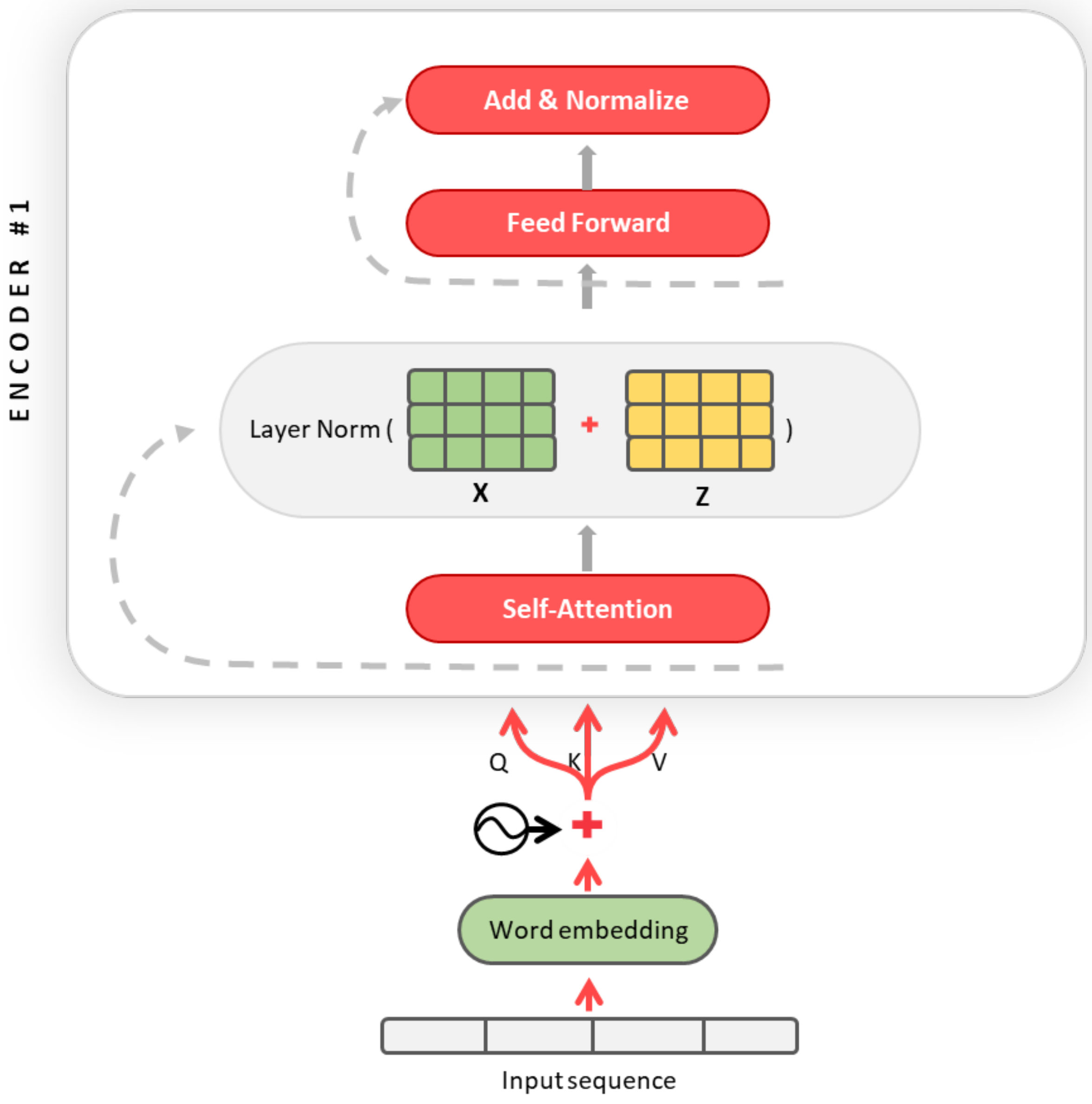


Here is the entire process for multi-head self-attention:

- Query(Q), Key(K), and Value(V) are calculated by multiplying the input embeddings with their respective matrices Wq, Wk, and Wv.
- Then, context vector(Z) is calculated from each attention.
- Finally, all the context vectors (Z1, Z2……Z8) are concatenated and multiplied with a weights matrix such that the output(Z)=(SxDmodel).

Residual Connections and “add &normalize” Layer

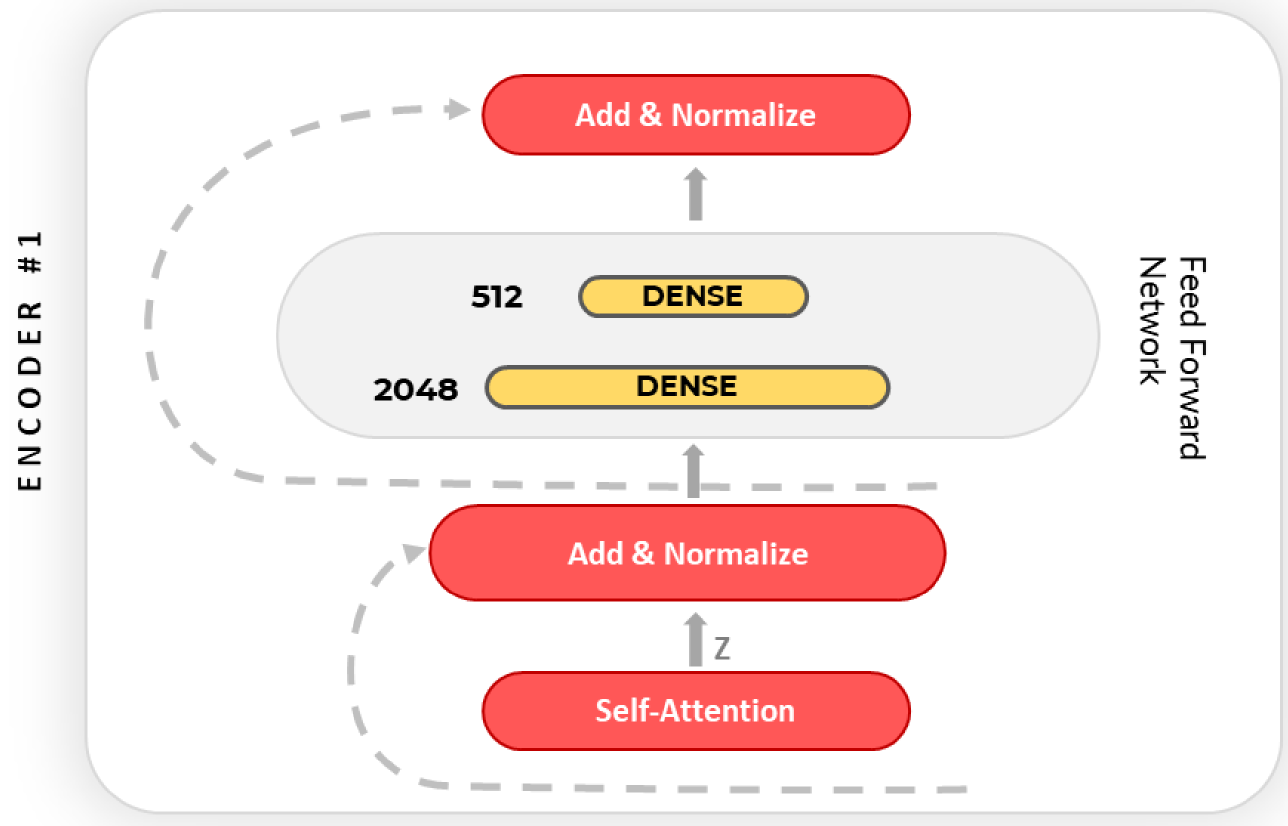
As the input embedding passes through multiple layers, there is a risk of losing important information, which can impair the network's ability to understand the context of the input and in turn negatively impact performance. To address this issue, **residual connections** are utilized, followed by an **add and normalize layer**. The **add** layer sums the output of a layer with its input (F(x)+x), while the **normalize** layer adjusts the values it receives, usually through a process called **Layer Normalization**.



The goal of the add and normalize layers is to ensure that the outputs from the different layers have similar scales, which can make it easier to combine the outputs.

Position-Wise Fully Connected Feed-Forward Network

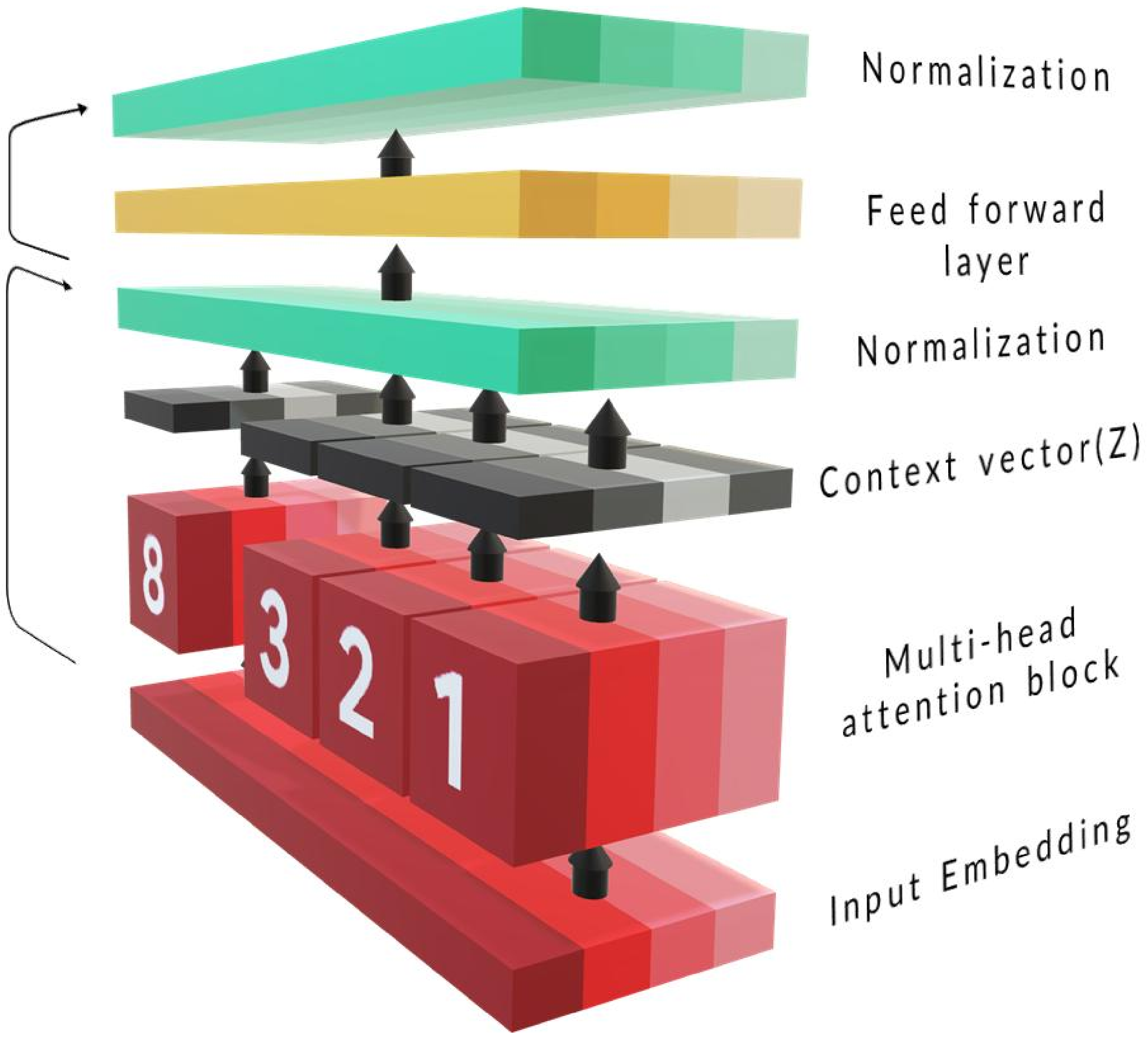
This layer helps the model learn more complex and nuanced relationships between the input and output data. The point-wise feed-forward network consists of two fully-connected layers with a ReLU activation in between.



The workings of a feed-forward network can be summarized as follows:

1. The input information is first **transformed linearly**, enabling the model to identify various features within the input embeddings.
2. The output is then passed through a **non-linear activation function**, such as ReLU, introducing non-linearity to the model and allowing it to learn intricate and nuanced relationships within the input embeddings.
3. A **second linear transformation** is applied to the output from the activation function to complete the process.

With this, you have gone through all the important components of the encoder block.


Normalization
Feed forward layer
Normalization
Context vector(Z)
Multi-head attention block
Input Embedding
8
3
2
1

Components of Transformer Architecture: Decoder

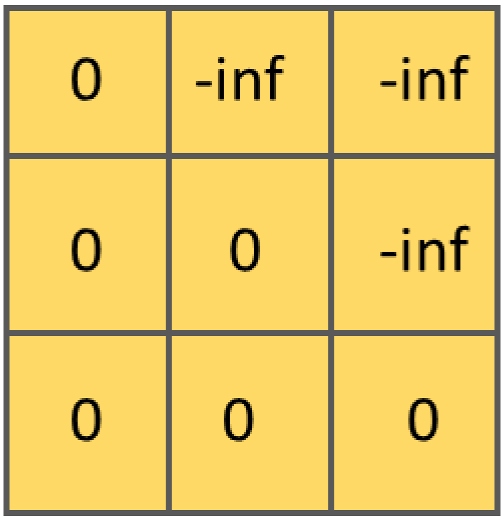
The primary role of the decoder block is to make predictions about the next token in a sequence, taking into account the previous tokens and the attention received from the encoder block.

During training, the decoder is provided with the **target output** as input. However, this is shifted to the right by including a <**start**> token to account for the lack of previous tokens. The decoder then uses the <**start**> token and the **context vector** to predict the **first token**. With each subsequent prediction, the new word is added as input to the decoder in the next step, and this process continues until the <**end**> token is predicted.

The components of the decoder block remain similar to those of the encoder, except for two new additions:

1. Look-ahead mask
2. Cross-attention (encoder-decoder attention)

Look Ahead Mask



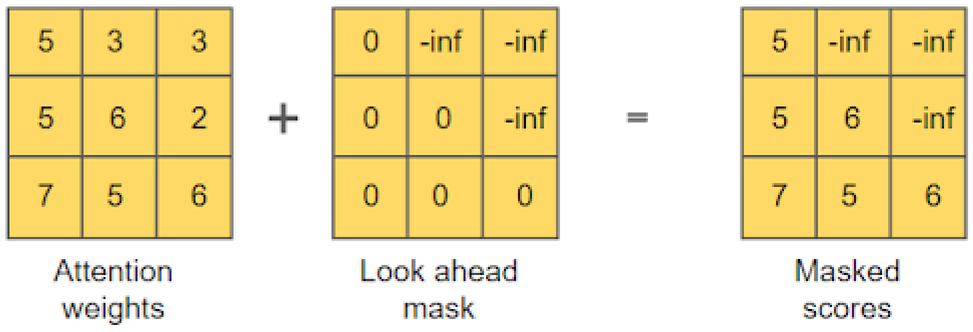
0	-inf	-inf
0	0	-inf
0	0	0

Look ahead mask

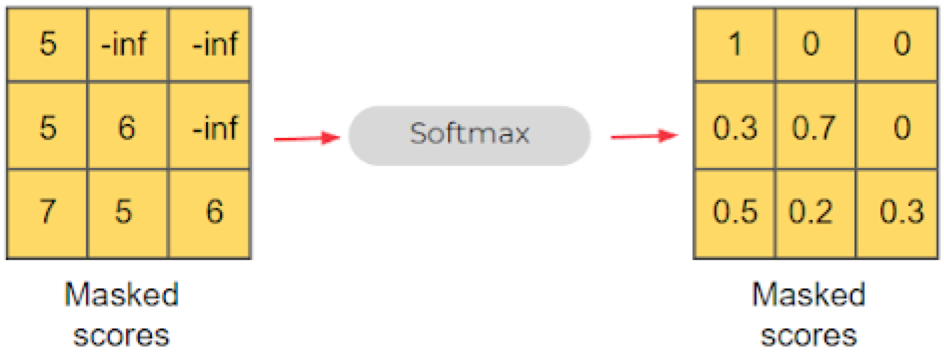
In the transformer decoder, the model must generate a new word based on the context of all the previous words in the input sequence, but it should not have access to information about the future words that have not been generated yet. To ensure that the model only has access to the relevant information, a look-ahead mask is employed. This mask restricts the model's access to future words, allowing it to focus solely on the previous words and make decisions based on that context.

The steps involved in implementing look-ahead masking are as follows:

- Attention weights are added (element-wise) to another matrix (made of 0 and -∞).



- Now, a softmax is applied to the masked score matrix.



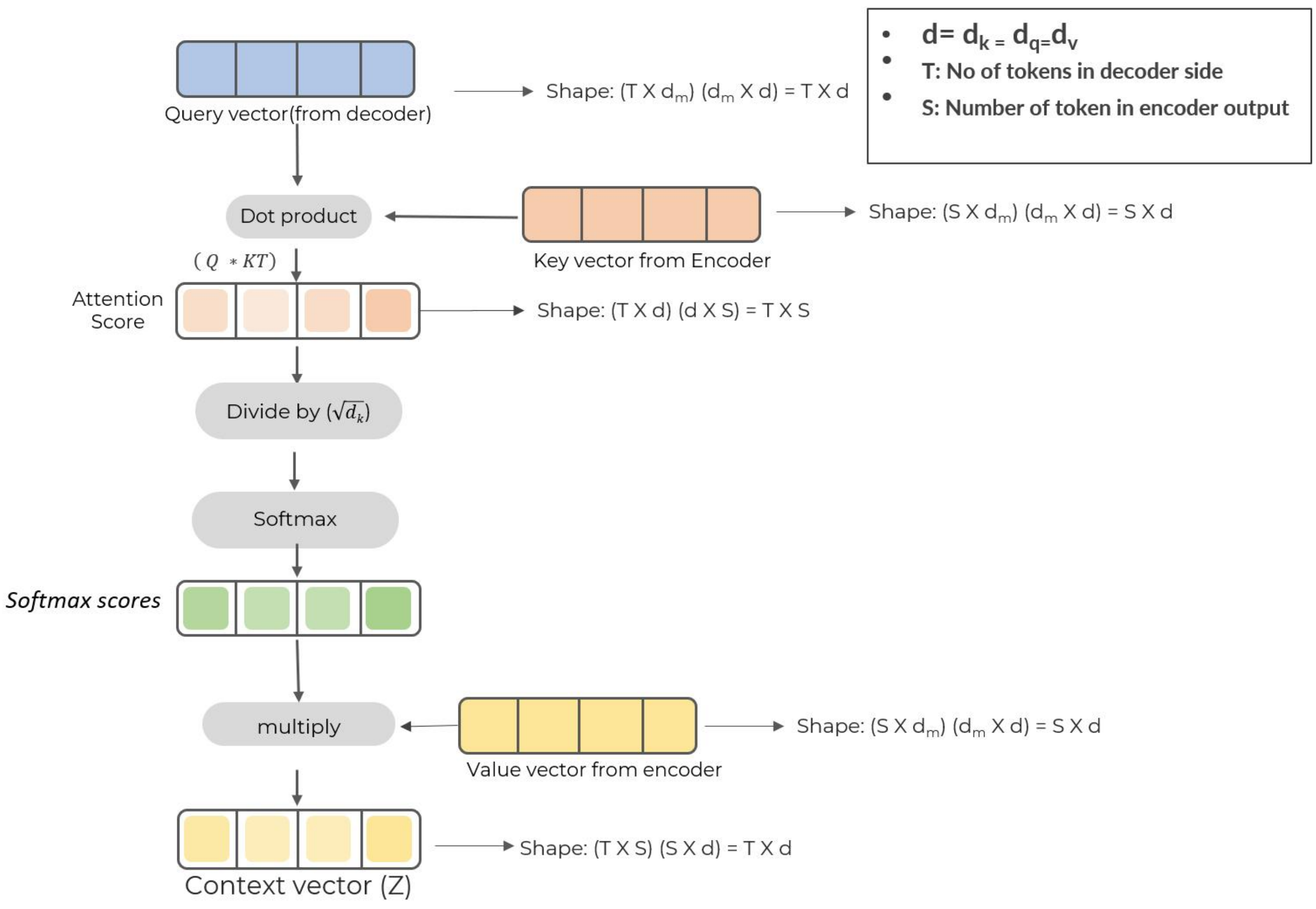
- Once the attention layers are applied to the current positions, a normalization layer is applied, just like it is done in the encoder.

Cross-Attention

After applying the normalization layer, cross-attention (encoder-decoder attention) is applied to the output received from the previous layer of the decoder and the output of the encoder (context vector).

Cross-attention obtains its queries (Q) from the previous decoder layer and the keys (K) and values (V) from the encoder output. This allows every position in the decoder to look over all the positions in the input sequence (similar to the typical encoder-decoder architecture).

The below image depicts how cross-attention is calculated.



$d = d_k = d_q = d_v$
T: No of tokens in decoder side
S: Number of token in encoder output
Query vector(from decoder)
Shape: $(T \times d_m)(d_m \times d) = T \times d$
Dot product
Key vector from Encoder
Shape: $(S \times d_m)(d_m \times d) = S \times d$
$(Q * KT)$
Attention Score
Shape: $(T \times d)(d \times S) = T \times S$
Divide by $(\sqrt{d_k})$
Softmax
Softmax scores
multiply
Value vector from encoder
Shape: $(S \times d_m)(d_m \times d) = S \times d$
Context vector (Z)
Shape: $(T \times S)(S \times d) = T \times d$

Post cross-attention, the following layers are added:

- Add and normalization
- Feed-forward network
- Add and normalization

These steps are repeated for six iterations, building newer representations on top of the previous ones:

- Once the repeated iteration is performed, a linear layer followed by a softmax function is applied to generate the probability scores for the next token.
- Once the most probable token is predicted, it is passed as the tail to the input sequence of the decoder.
- This is repeated until we receive the <end> token, which marks the completion of the output sequence.

Variants of Transformer

Broadly, the variants of transformers are categorized into the following three categories:

Autoregressive models: They correspond to the decoder of the original transformer model. They are trained on traditional language modeling tasks, such as next-word prediction, and their most common use case is text generation, as seen in the GPT family of models.

Autoencoding models: They correspond to the encoder of the original transformer model. Autoregressive models are trained by obscuring certain tokens in the input and having the model attempt to reconstruct the original sentence. Their primary application is in sentence classification or token classification, as seen in the BERT family of models.

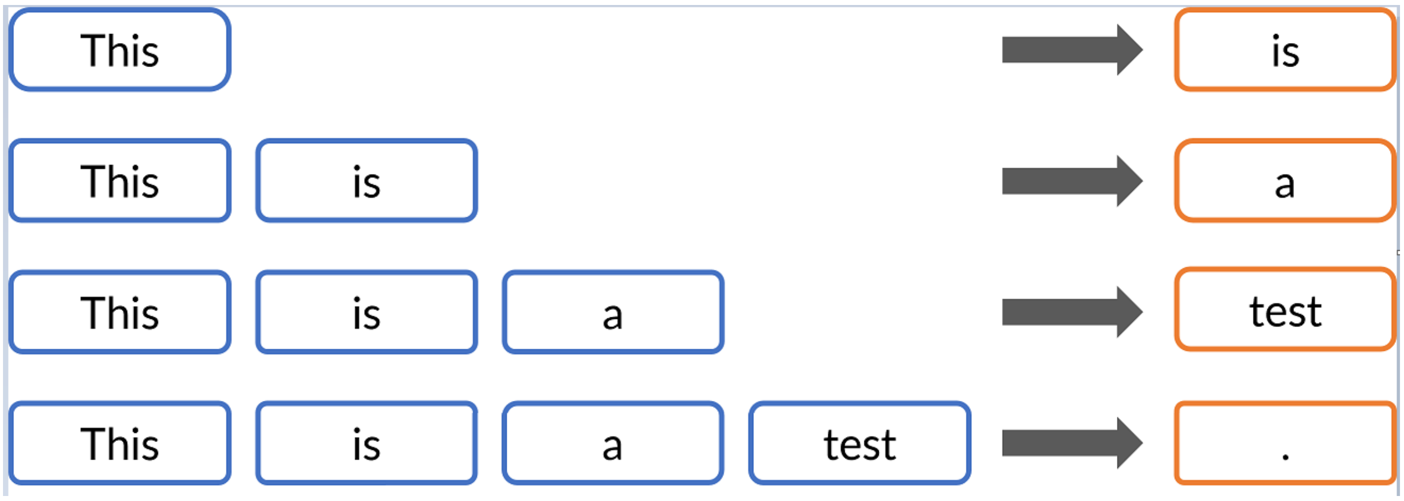
Sequence-to-sequence models: They use the encoder and decoder of the original transformer. The primary use cases for the original transformer model and its variants include translation, summarization, and question-answering. The original transformer model, designed specifically for language translation, is one example. Other examples include T5 and BART.

Transfer Learning With Transformer

Here are the two popular ways to pre-train methods:

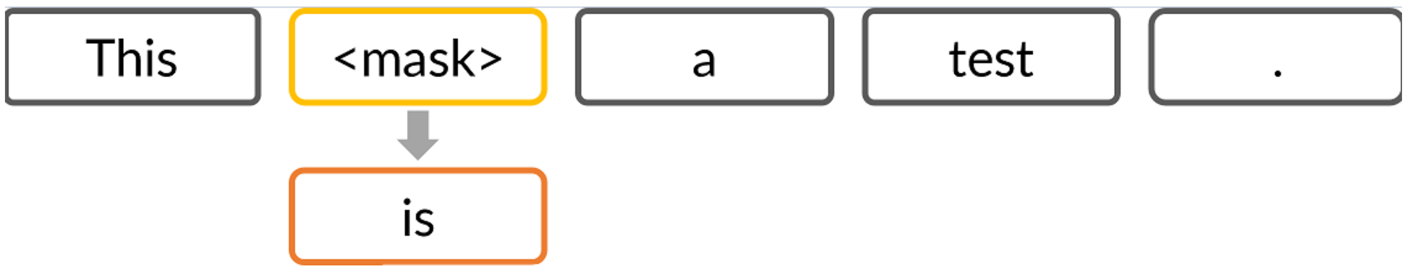
Causal Language Model

This type of modeling helps predict the next token in a sequence of tokens, and the model can only attend to the tokens on the left. Such a training scheme makes this model unidirectional.



Masked Language Model

This type of modeling masks a word or a certain percentage of words in a given sentence, and the model is trained to predict those masked words based on the remaining words in the sentence. Such a training scheme makes this model bidirectional because the representation of the masked word is learned based on the words preceding and following it.



Hugging Face Library

Hugging Face library is known for its NLP platform, which provides a collection of pre-trained machine learning models for various NLP tasks, such as text classification, text generation, question-answering, and more. This platform is accessible through various programming languages, including Python, and it offers easy access to the latest and most advanced NLP models.

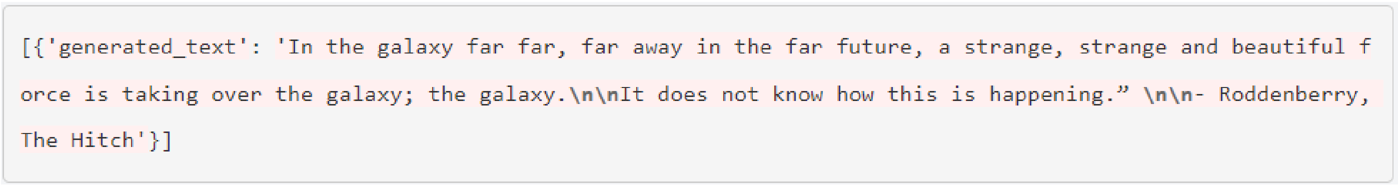
The models available on the Hugging Face platform have been trained on extensive text data and can be further improved by fine-tuning them on specific tasks.

Pipelines provide an effortless and convenient solution for utilizing a variety of models for inference. These pipelines are designed as objects that encapsulate the complicated code from the Hugging Face library, making it simple to perform various NLP tasks with a dedicated API. These tasks include recognizing named entities, performing masked language modeling, analyzing sentiment, extracting features, and answering questions.

- For text generation, you can use pipeline() in the following manner.

```
generator = pipeline("text-generation")
generator("In the galaxy far far")
```

Here is the output.



```
[{'generated_text': 'In the galaxy far far, far away in the far future, a strange, strange and beautiful f
orce is taking over the galaxy; the galaxy.\n\nIt does not know how this is happening.” \n\n- Roddenberry,
The Hitch'}]
```

This code will by default download the GPT-2 model. You can also customize the model type as per your requirement.

```
generator = pipeline("text-generation", model="distilgpt2")
generator(
    "In the galaxy far far",
    max_length=30,
    num_return_sequences=2,
)
```

Output

```
[{'generated_text': 'In the galaxy far far away, which mean a few million years ago. Now it’s looking at h
ow much it can melt - maybe'},
 {'generated_text': "In the galaxy far far, far away, it’s very familiar to the ancient history"}]
```

- The code to perform the task of masked language modeling (where the downloaded model predicts masked words) would look like this.

```
unmasker = pipeline("fill-mask")
unmasker("You are going to <mask> about a wonderful library today.", top_k=2)
```

In the above code, we are asking the model to make the top two predictions for the <mask> token.

Output

```
[{'score': 0.5679107308387756,
  'token': 1798,
  'token_str': ' hear',
  'sequence': 'You are going to hear about a wonderful library today.'},
 {'score': 0.22818315029144287,
  'token': 1532,
  'token_str': ' learn',
  'sequence': 'You are going to learn about a wonderful library today.'}]
```

- You can do sentiment classification like this.

```
input_sentences = [
        "I don't like this movie",
        "Upgrad is helping me learn new and wonderful things.",

    ]
classifier = pipeline("sentiment-analysis")
classifier(
    input_sentences
```

)

Output

```
[{'label': 'NEGATIVE', 'score': 0.9839025139808655},
 {'label': 'POSITIVE', 'score': 0.9998325109481812}]
```

A pipeline is composed of a series of components, each of which performs a specific NLP task. The following are some of the common components that can be used inside a pipeline: a tokenizer, model, named entity recognizer, sentiment analyzer, question answering model, text generation Model, and text summarizer.

The first component inside the pipeline() function is pre-processing. Pre-processing is done by a tokenizer inside the transformer API. This is how an input text is pre-processed:

- Tokenization

  The transformer tokenizers convert the input text into the numerical representation of each token.

  ```
  from transformers import BertTokenizer
  tokenizer = BertTokenizer.from_pretrained("bert-base-cased")
  tokenizer("Learning NLP is so much rewarding")
  ```

  Output

  ```
  {'input_ids': [101, 9681, 21239, 2101, 1110, 1177, 1277, 10703, 1158, 102], 'token_type_ids': [0, 0, 0, 0, 0, 0, 0, 0, 0, 0], 'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]}
  ```

  Here, the input text is transformed into a dictionary consisting of the following three keys: “input_ids,” “token_type_ids,” and “attention_mask.”

- In case the tokenized outputs are not of the same length, the tokenizer function allows us to control the output using **padding** and **truncation**.

  ```
  sequences = ["Learning NLP is so much rewarding","Another test
  sentence"]
  model_inputs = tokenizer(sequences, padding="longest")
  pp.pprint(model_inputs)
  ```

  This is reflected in attention_mask and input_ids.

  ```
  'attention_mask': [[1, 1, 1, 1, 1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 0, 0, 0, 0, 0]]
  'input_ids': [[101, 9681, 21239, 2101, 1110, 1177, 1277, 10703, 1158, 102],
                [101, 2543, 2774, 5650, 102, 0, 0, 0, 0, 0]]
  ```

Fine-Tuning BERT Model

BERT stands for Bidirectional Encoder Representations from Transformers. BERT is a transformer-based architecture that uses an attention mechanism to learn the contextual relationships between words in a sentence.
During pre-training, BERT is exposed to a large corpus of text, such as books and articles, and is optimized to predict missing words or words that have been masked in the input. This pre-training allows BERT to build a representation of language and its context, which can then be fine-tuned for specific NLP tasks. The main innovation of BERT is its bidirectional approach, which considers the context to the left and right of each word, giving the model a more nuanced understanding of the context.

In the module, you tried to solve the following problem statement using the BERT model.

Predict whether any given two sentences (questions) are semantically similar to each other. You used the Quora Question Pair (QQP) data set, which is part of the GLUE benchmark.

The goal of solving this exercise was:

- To get you to be comfortable working with Hugging Face data sets
- Learn to load, train, and save BERT-based models (BERT and ALBERT among others)
- Perform end-to-end implementation (training, validation, prediction, and evaluation)

You performed the following key steps to solve the problem statement:

- You downloaded the data set. There were 363,846 entries of data with four columns: question1, question2, label, and idx.

```
train                                                                              =
pd.read_csv('/content/drive/MyDrive/sentence_pair_classification_data/train.csv')
train.sample(5)
```

- You used the **load_datset()** function to automatically convert the given data into a dictionary.

```
dataset=load_dataset('csv',data_files={'train':'/content/drive/MyDrive/sentence_p
air_classification_data/train.csv',\'valid':'/content/drive/MyDrive/sentence_pair
_classification_data/val.csv','test':
'/content/drive/MyDrive/sentence_pair_classification_data/test.csv'},)
```

- To **pre-process** the inputs, we initialized the **model_name/checkpoint** and loaded the tokenizer such that it could automatically load the tokenizer for it.

```
model_checkpoint = "bert-base-cased"
from transformers import AutoTokenizer
tokenizer = AutoTokenizer.from_pretrained(model_checkpoint)
```

- After the tokenizer is loaded, we applied it to the entire data set as shown below.

```
def preprocess_function(records):
     return tokenizer(records['question1'], records['question2'], truncation=True,
return_token_type_ids=True, max_length = 75)
encoded_dataset = dataset.map(preprocess_function, batched=True )
```

Output

```
DatasetDict({
    train: Dataset({
        features: ['question1', 'question2', 'label', 'idx', 'input_ids',
'token_type_ids', 'attention_mask'],
        num_rows: 363846
    })
    valid: Dataset({
        features: ['question1', 'question2', 'label', 'idx', 'input_ids',
'token_type_ids', 'attention_mask'],
        num_rows: 40430
    })
    test: Dataset({
        features: ['question1', 'question2', 'label', 'idx', 'input_ids',
'token_type_ids', 'attention_mask'],
        num_rows: 390965
    })
})
```

- Since the transformed data set consists of few original features, we must remove them once the data set is encoded.

```
pre_tokenizer_columns = set(dataset["train"].features)
tokenizer_columns=list(set(encoded_dataset["train"].features) -
pre_tokenizer_columns)
print("Columns added by tokenizer:", tokenizer_columns)
```

- The next step is to split the data into train and validation sets.

```
from transformers import DataCollatorWithPadding
```

```
data_collator = DataCollatorWithPadding(tokenizer=tokenizer,return_tensors="tf",)

tf_train_dataset = encoded_dataset["train"].to_tf_dataset(
columns=tokenizer_columns,label_cols=["labels"],shuffle=True,
batch_size=batch_size,collate_fn=data_collator,)
tf_validation_dataset = encoded_dataset["valid"].to_tf_dataset(

columns=tokenizer_columns,label_cols=["labels"],shuffle=False,batch_size=batch_size,
    collate_fn=data_collator,)
```

- By now, the data is set into a format that is compatible with the chosen Tensorflow framework. This was done using the **to_tf_dataset**() function.
- Next, you download the model and define hyperparameters for it.

```
model = TFAutoModelForSequenceClassification.from_pretrained(model_checkpoint,
num_labels = num_labels)

num_epochs = 3
num_train_steps = len(tf_train_dataset) * num_epochs
lr_scheduler = PolynomialDecay(
    initial_learning_rate=5e-5, end_learning_rate=0.0,
decay_steps=num_train_steps, power = 2
)
opt = Adam(learning_rate=lr_scheduler)
loss = SparseCategoricalCrossentropy(from_logits=True)
```

- You can save the model training history in your drive and use the pre-trained model when needed.
- After loading the model, you can apply a custom function to infer the model's performance on a custom input.

```
def check_similarity(question1, question2):
  tokenizer_output = tokenizer(question1, question2, truncation=True,
return_token_type_ids=True, max_length = 75, return_tensors = 'tf')
  logits = trained_model(**tokenizer_output)["logits"]
  predicted_class_id = int(tf.math.argmax(logits, axis=-1)[0])
  if predicted_class_id == 1:
    return "Both questions mean the same"
  else:
    return "Both the questions are different."
```

- Once the function is defined, you can verify the results.

This wraps up our use case of fine-tuning a BERT model for finding sentence-pair similarity.