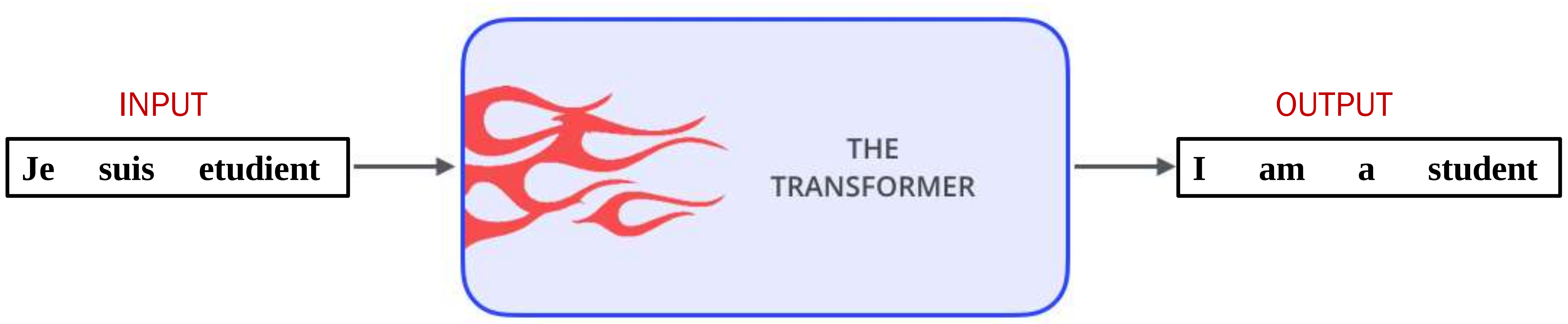
Introduction

Transformer neural networks were first proposed by a Google-led team in 2017 in a widely cited paper titled ***“Attention Is All You Need”***

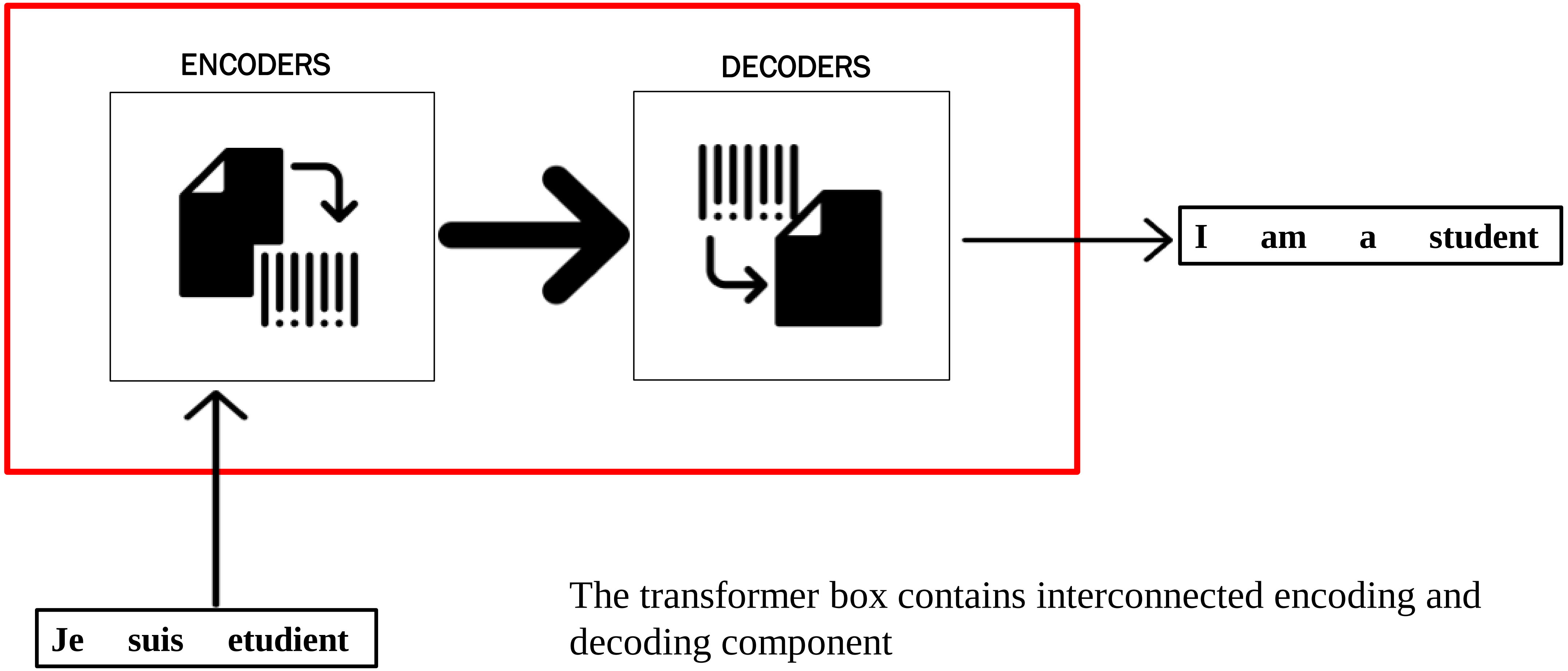
A transformer neural network can take an input sentence in the form of a sequence of vectors, and converts it into a vector called an encoding, and then decodes it back into another sequence.

Continue



In a machine translation application, it would take a sentence in one language, and output its translation in another.

Continue



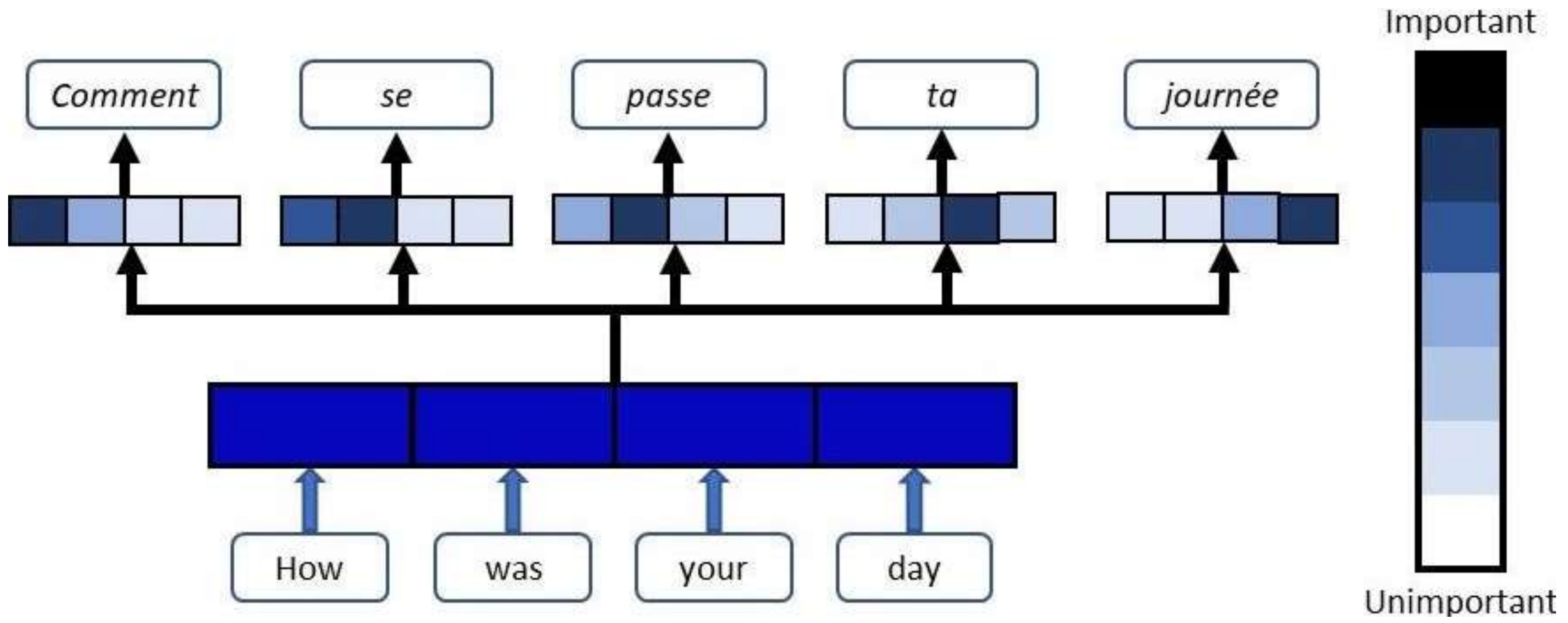
The transformer box contains interconnected encoding and decoding component

Continue

The Transformer neural network is architecture based on a self-attention mechanism that is mainly designed for language understanding.

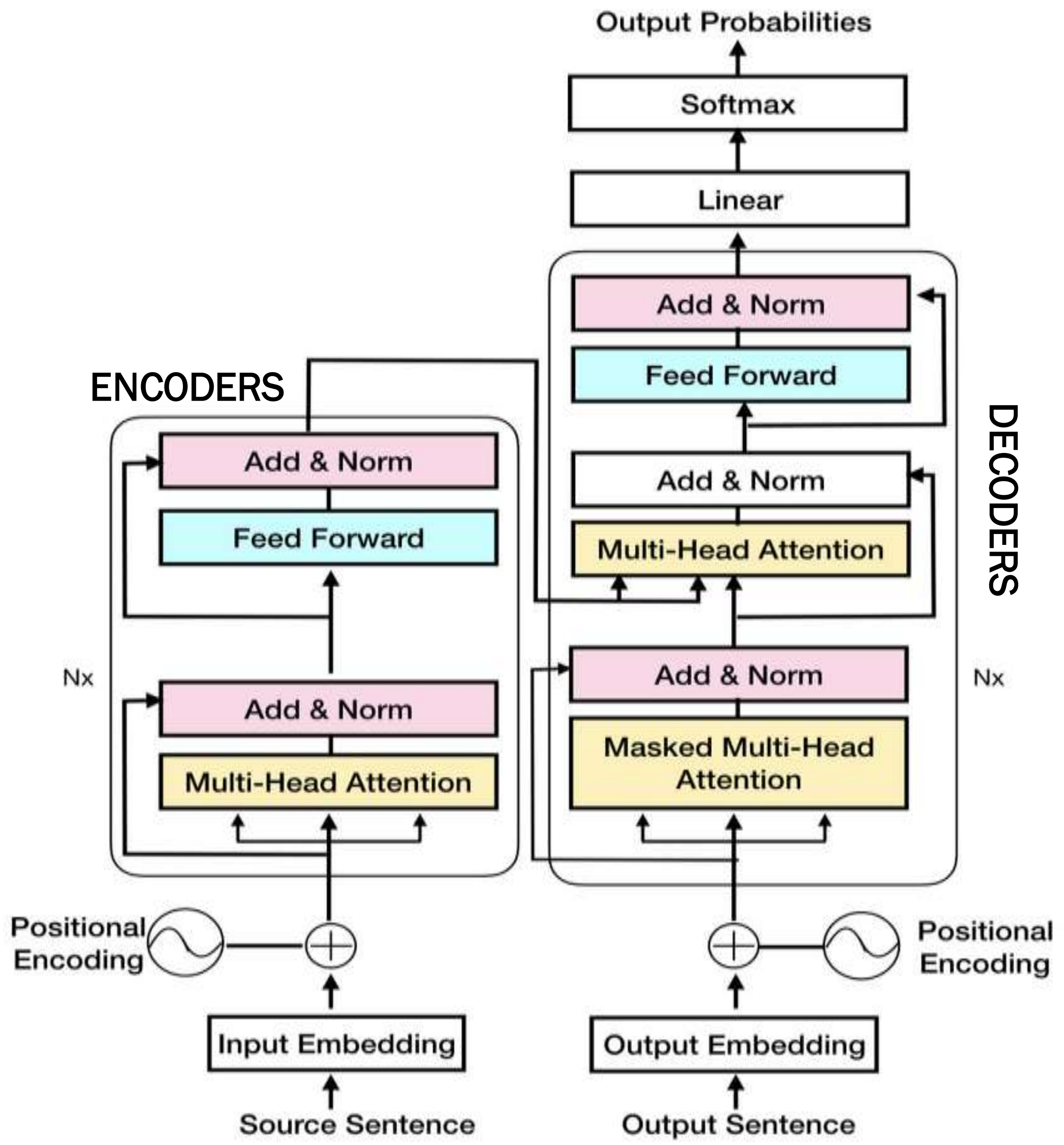
During the time of encoding, attention mechanism decides how important other tokens in an inpiut are .

Besides, it allows the transformer to focus on certain words on both the left and right of the current word to decide how to translate it.



Architecture

The core architecture consists of a stack of encoders fully connected to a stack of decoders. Each encoder consists of two blocks: a self-attention component, and a feed forward network. Each decoder consists of three blocks: a self-attention component, an encoder-decoder attention component, and a feed forward component.

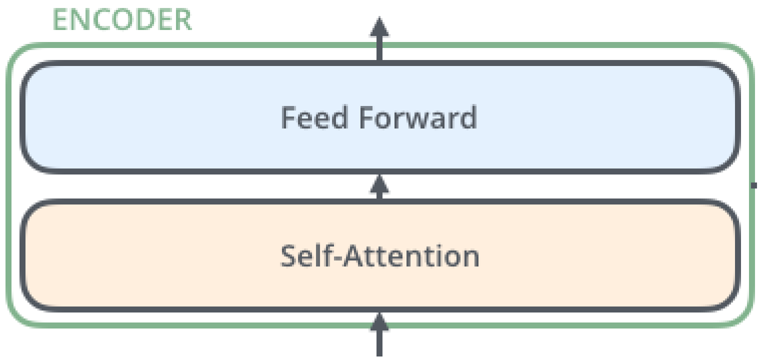


Architecture

Encoder stack contains six encoder layers on top of each other.

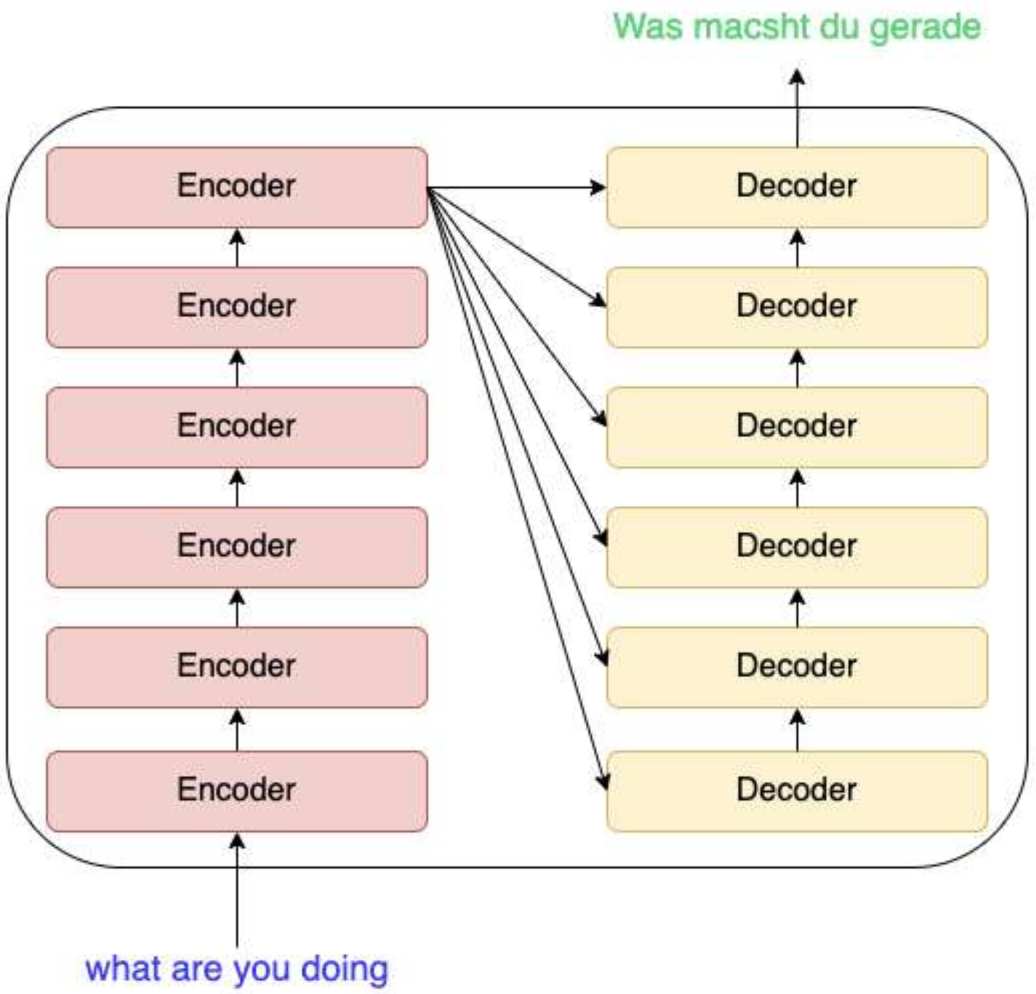
The encoders are all identical in structure (yet they do not share weights). Each one is broken down into two sub-layers:

- ❑ **A multi-head self-attention Layer, and**
- ❑ **A position-wise fully connected feed-forward network**



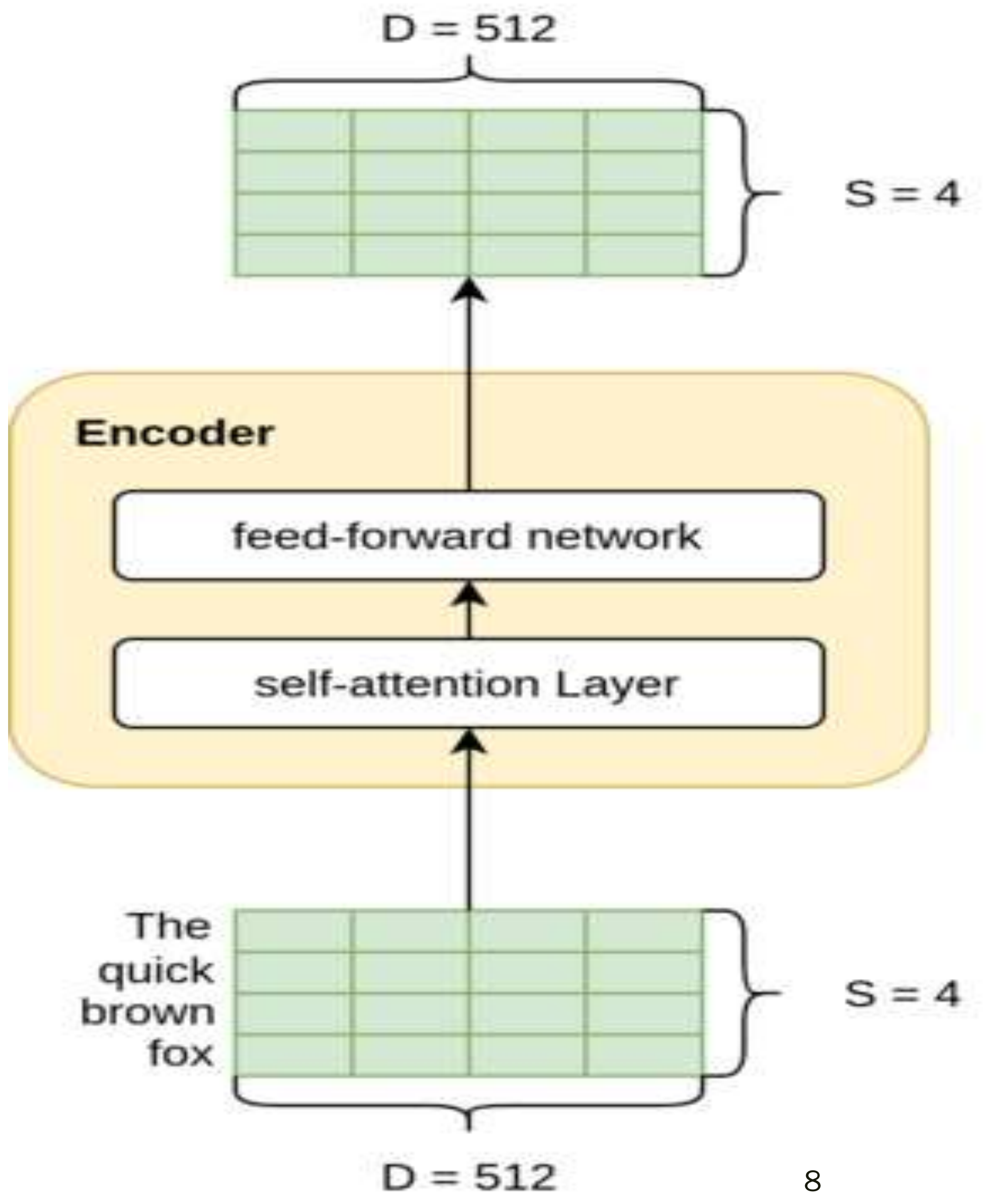
Architecture

The self-attention mechanism takes in a set of input encodings from the previous encoder and weighs their relevance to each other to generate a set of output encodings. The feed-forward neural network then further processes each output encoding individually. These output encodings are finally passed to the next encoder as its input, as well as the decoders.



Architecture

Since the encoder layers are stacked on top of each other, the output needs to be of the same dimensions as the input so it can flow easily into the next encoder.
Therefore, the output is also of the shape, SxD.

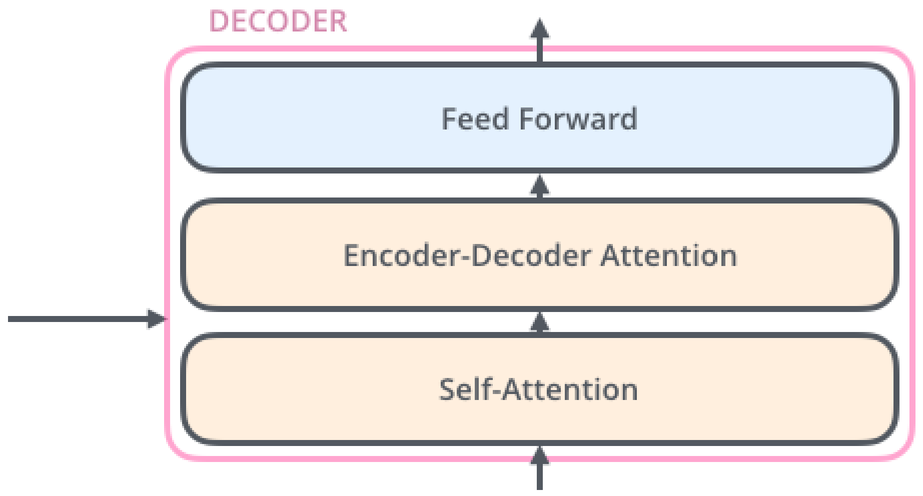


Architecture

Each decoder consists of three major components:

- ❑ **A self-attention mechanism**
- ❑ **An attention mechanism over the encodings and**
- ❑ **A feed-forward neural network.**

The decoder functions in a similar fashion to the encoder, but an additional attention mechanism is inserted which instead draws relevant information from the encodings generated by the encoders



Self Attention Mechanism in the Transformer

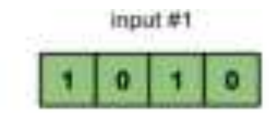
A self-attention module takes in *n* inputs, and returns *n* outputs. In **layman's** terms, the self-attention mechanism allows the inputs to interact with each other ("self") and find out who they should pay more attention to ("attention"). The outputs are aggregates of these interactions and attention scores.

The illustrations are divided into the following steps:

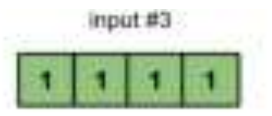
1. Prepare inputs
2. Initialize weights
3. Derive **key**, **query** and **value**
4. Calculate attention scores for Input 1
5. Calculate softmax
6. Multiply scores with **values**
7. Sum **weighted values** to get Output 1
8. Repeat steps 4–7 for Input 2 & Input 3

Self Attention Mechanism in the Transformer

Step-1 (Arrange inputs)
Assume we have 3 inputs, each with dimension 4



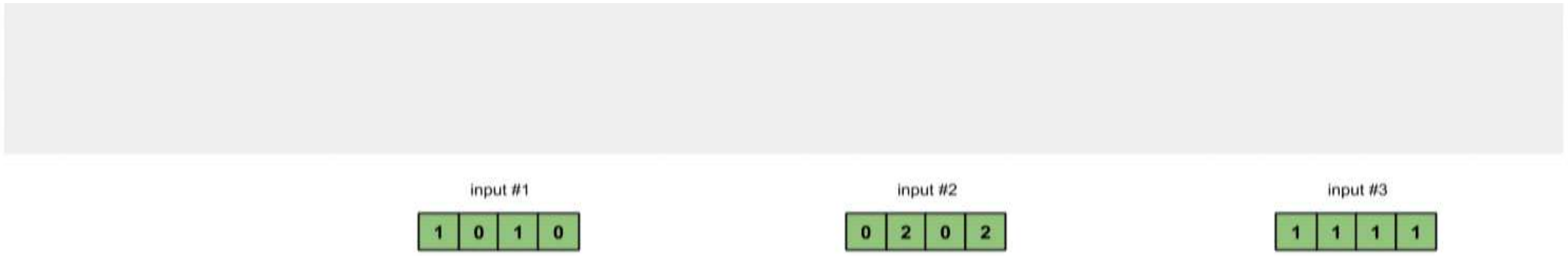
input #2
0 2 0 2



Input 1: [1, 0, 1, 0] ; Input 2: [0, 2, 0, 2] ; Input 3: [1, 1, 1, 1]

Step-2 (Initialize weights)
Now we have to derive **key** (orange), **query** (red), and **value** (purple). For this example, let’s take that we want these representations to have a dimension of 3. Because every input has a dimension of 4, this means each set of the weights must have a shape of 4×3.

Self Attention Mechanism in the Transformer



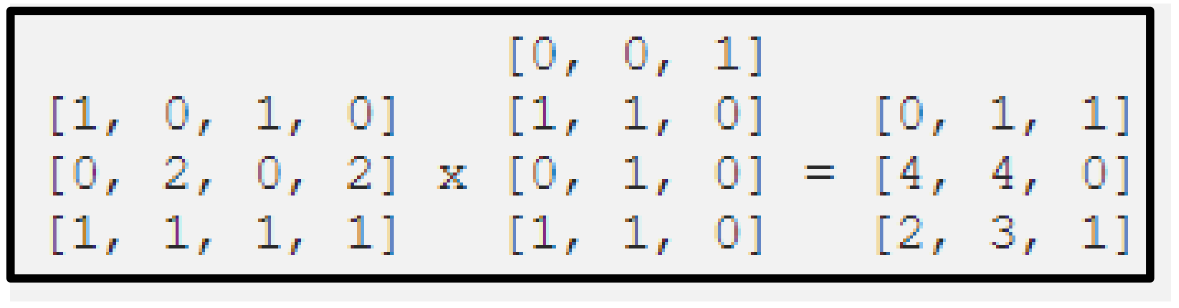
In order to obtain these values, every input (green) is multiplied with a set of weights for **keys**, a set of weights for **queries** , and a set of weights for **values**. In our example, we initialize the three sets of weights as follows.

Weights for **key**:	Weights for **query**:	Weights for **value**:
[[1, 0, 1], [1, 0, 0], [0, 0, 1], [0, 1, 1]]	[[0, 0, 1], [1, 1, 0], [0, 1, 0], [1, 1, 0]]	[[0, 2, 0], [0, 3, 0], [1, 0, 3], [1, 1, 0]]

Self Attention Mechanism in the Transformer

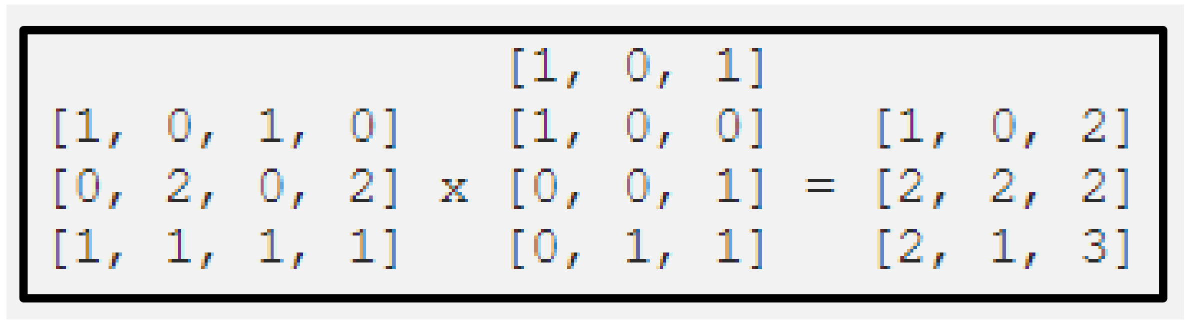
Step-3 (Derive key, query and value)

Key outputs for 3 different inputs,K $\longrightarrow$



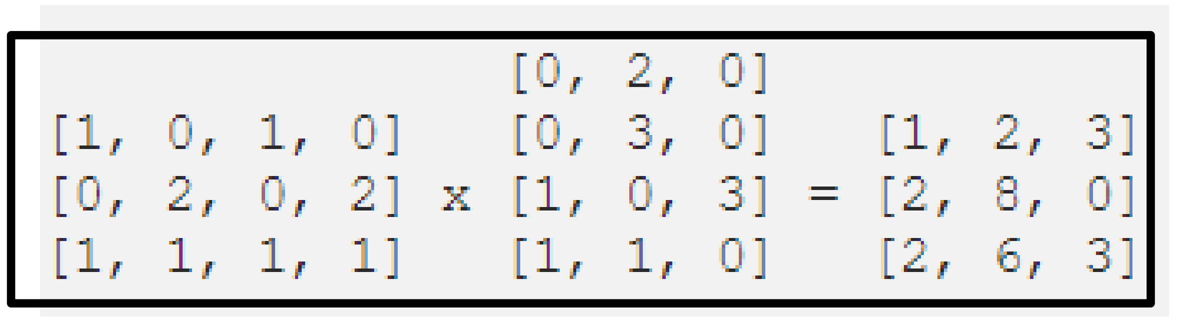
```
                      [0, 0, 1]
[1, 0, 1, 0]          [1, 1, 0]         [0, 1, 1]
[0, 2, 0, 2]    x     [0, 1, 0]    =    [4, 4, 0]
[1, 1, 1, 1]          [1, 1, 0]         [2, 3, 1]
```

Query outputs for 3 different inputs,Q $\longrightarrow$



```
                      [1, 0, 1]
[1, 0, 1, 0]          [1, 0, 0]         [1, 0, 2]
[0, 2, 0, 2]    x     [0, 0, 1]    =    [2, 2, 2]
[1, 1, 1, 1]          [0, 1, 1]         [2, 1, 3]
```

Value outputs for 3 different inputs,V $\longrightarrow$

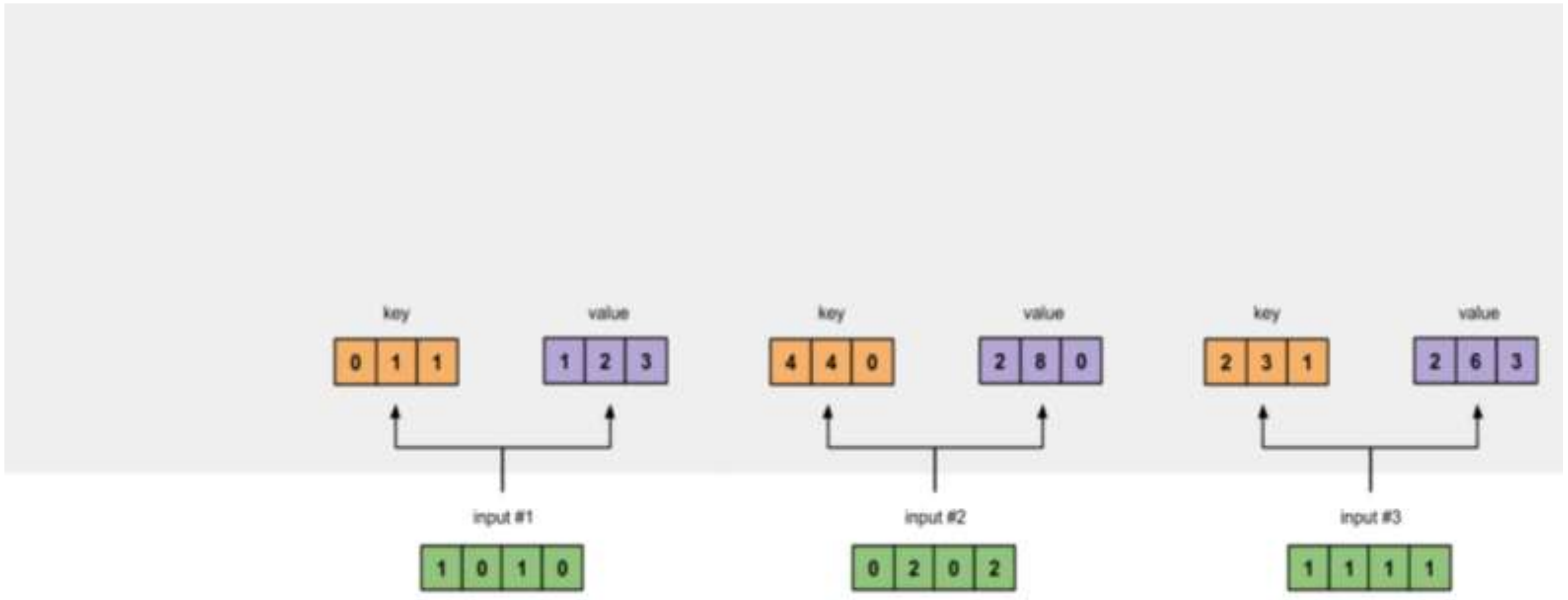


```
                      [0, 2, 0]
[1, 0, 1, 0]          [0, 3, 0]         [1, 2, 3]
[0, 2, 0, 2]    x     [1, 0, 3]    =    [2, 8, 0]
[1, 1, 1, 1]          [1, 1, 0]         [2, 6, 3]
```

Self Attention Mechanism in the Transformer

Step 4: Calculate attention scores

we start off with taking a dot product between 1st Input's **query** (red) with all **keys** (orange), including itself

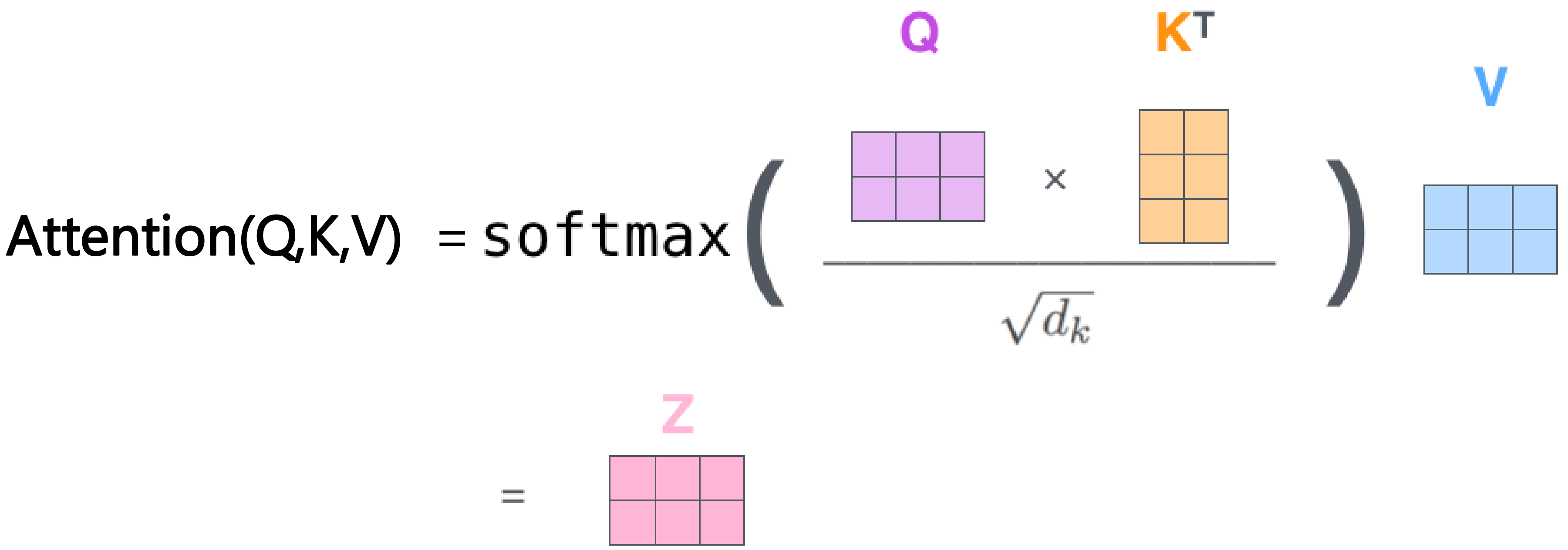


Attention Score Matrix
$Q.K^T$

```
          [0, 4, 2]
[1, 0, 2] x [1, 4, 3] = [2, 4, 4]
          [1, 0, 1]
```

Self Attention Mechanism in the Transformer

Step 5: Calculate softmax

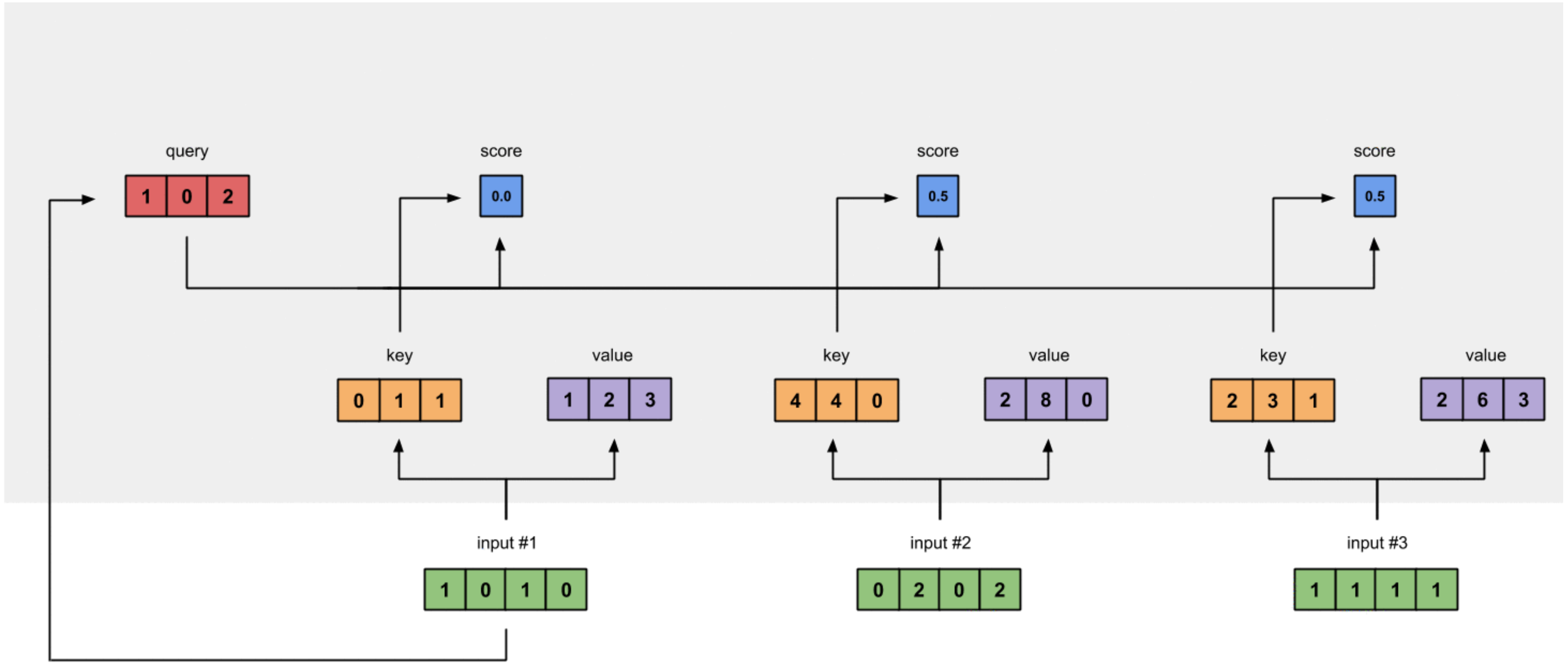


$\boxed{\text{softmax([2, 4, 4]) = [0.0, 0.5, 0.5]}}$

Self Attention Mechanism in the Transformer

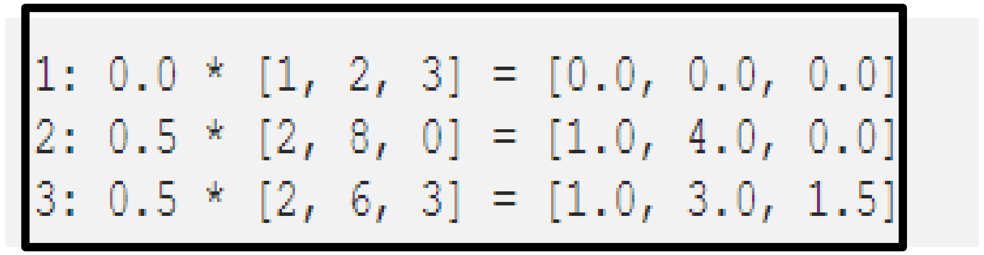
Step 6: Multiply scores with values

The softmax attention scores for each input (blue) is multiplied with its corresponding **value** (purple). This results in 3 *alignment vectors* (yellow).



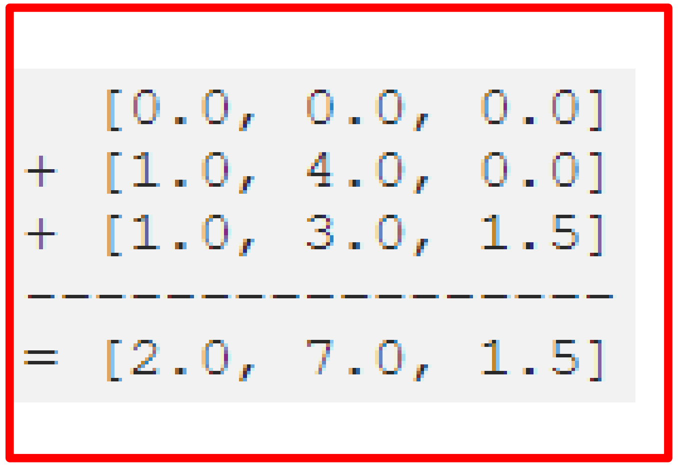
Self Attention Mechanism in the Transformer

Weighted values from step-6 $\longrightarrow$



```
1: 0.0 * [1, 2, 3] = [0.0, 0.0, 0.0]
2: 0.5 * [2, 8, 0] = [1.0, 4.0, 0.0]
3: 0.5 * [2, 6, 3] = [1.0, 3.0, 1.5]
```

Step 7: Sum weighted values to get Output

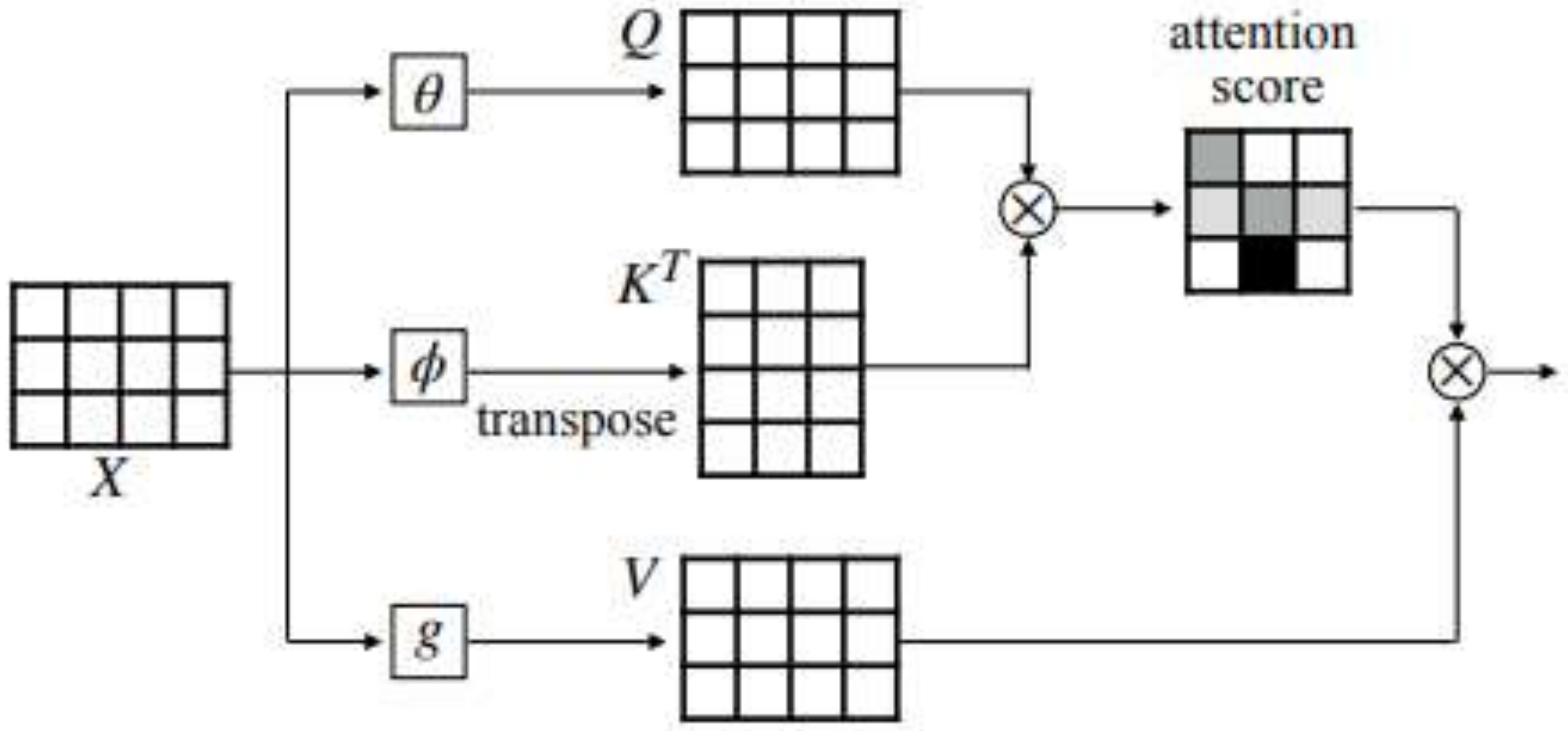


```
  [0.0, 0.0, 0.0]
+ [1.0, 4.0, 0.0]
+ [1.0, 3.0, 1.5]
-----------------
= [2.0, 7.0, 1.5]
```

The resulting vector [2.0, 7.0, 1.5] is Output 1, which is based on the **query representation from Input 1** interacting with all other keys, including itself.

We have to do step-4 to step-7 in the same way to generate output-2 and output-3 from input-1 and input-2

Self Attention Mechanism in a nutshell



Self-attention sub-layer

The multi-head variant of attention allows the model to jointly attend to information from different representation subspaces, and is defined as

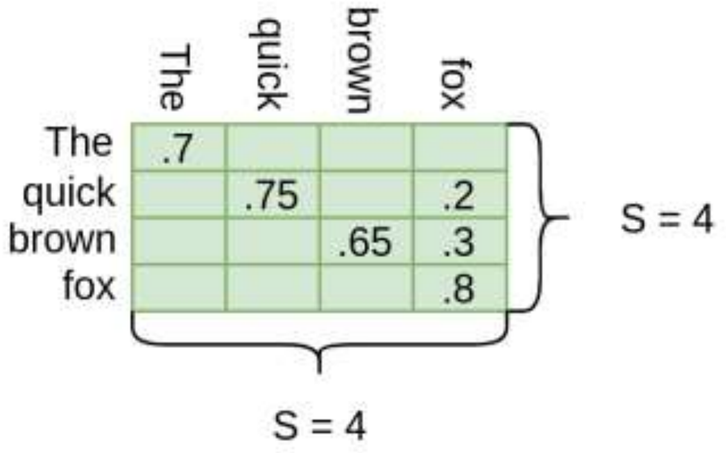
$$\mathrm{MultiHead}(Q, K, V) = \mathrm{Concat}(\mathrm{head}_1, ..., \mathrm{head}_\mathrm{h})W^O$$
$$\text{where } \mathrm{head}_\mathrm{i} = \mathrm{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Where the projections are parameter matrices $W_i^Q \in \mathbb{R}^{d_\text{model} \times d_k}$, $W_i^K \in \mathbb{R}^{d_\text{model} \times d_k}$, $W_i^V \in \mathbb{R}^{d_\text{model} \times d_v}$ and $W^O \in \mathbb{R}^{hd_v \times d_\text{model}}$.

h is the number of heads, and d_K and d_V are the dimensionalities of Key and Value

Continue

- The self-attention layer is initialized with 3 weight matrices — Query (W^Q), Key (W^k), and Value (W^v). Each of these matrices has a size of (Dxd), where d is taken as 64. The weights for these matrices are trained while training the model.
- The resultant would be SxS matrix as the contribution of each word in another word.

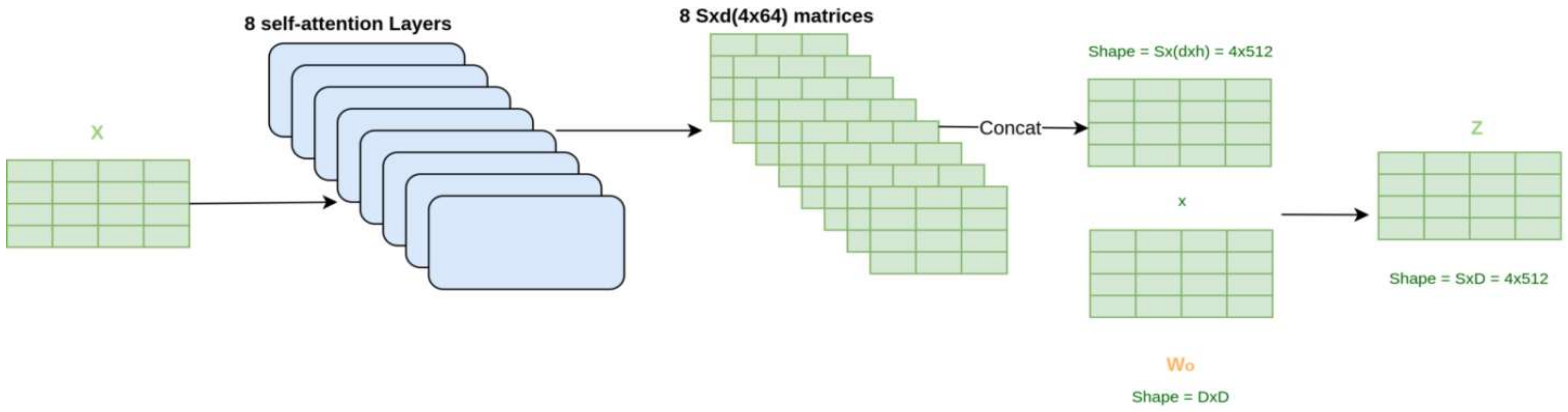


- Many such self-attention layers when are used in parallel, as a result is called multihead. Input X goes through many self-attention layers in parallel, each of which gives a z matrix of shape (Sxd) = 4×64. Then these 8(h) matrices are concatenated and again a final output linear layer, Wo, of size DxD is applied.

Continue

For concatenation operation a size of SxD(4x(64×8) = 4×512) is found. And multiplying this output by Wo, the final output Z with the shape of SxD(4×512) as desired can be obtained.
Also, the relation between h,d, and D is h x d = D

Finally get the output Z of shape 4×512 is found. But before it goes into another encoder it is passed through a Feed-Forward Network.



Continue

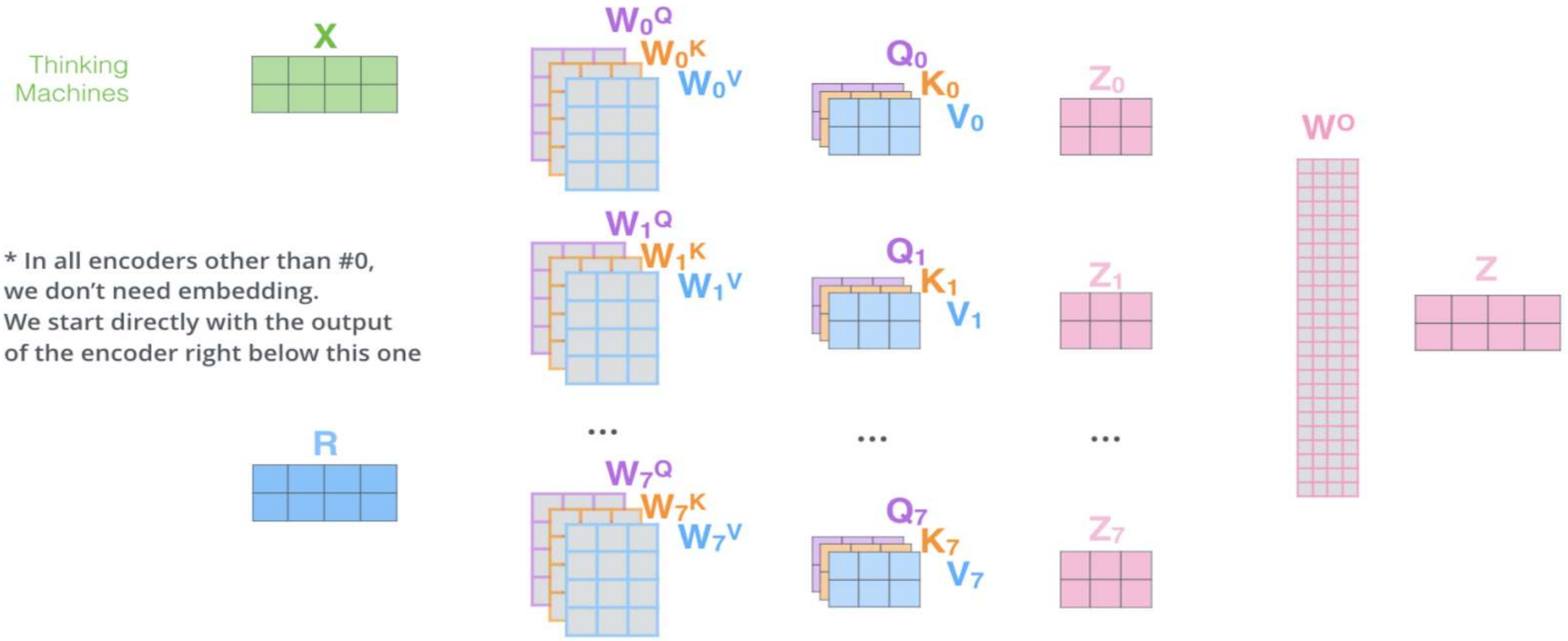
1) This is our input sentence*

2) We embed each word*

3) Split into 8 heads. We multiply X or R with weight matrices

4) Calculate attention using the resulting Q/K/V matrices

5) Concatenate the resulting Z matrices, then multiply with weight matrix W^O to produce the output of the layer



* In all encoders other than #0, we don't need embedding. We start directly with the output of the encoder right below this one

Position-wise Feed-Forward Networks

.It accepts a 3-dimensional input with shape (batch size, sequence length, feature size). The position-wise FFN consists of two dense layers that applies to the last dimension. Since the same two dense layers are used for each position item in the sequence, we referred to it as *position-wise*. Indeed, it is equivalent to applying two 1×1 convolution layers.

$\mathrm{FFN}(h_i) = \sigma(h_i W1 + b1)W2 + b2$

Here W1, W2, b1 and b2 are parameters

The self-attention sub-layer takes the information from all positions as input while the position-wise feed-forward layer is applied to each position separately

Positional Encoding

To better capture the sequential information, the Transformer model uses the *positional encoding* to maintain the positional information of the input sequence.

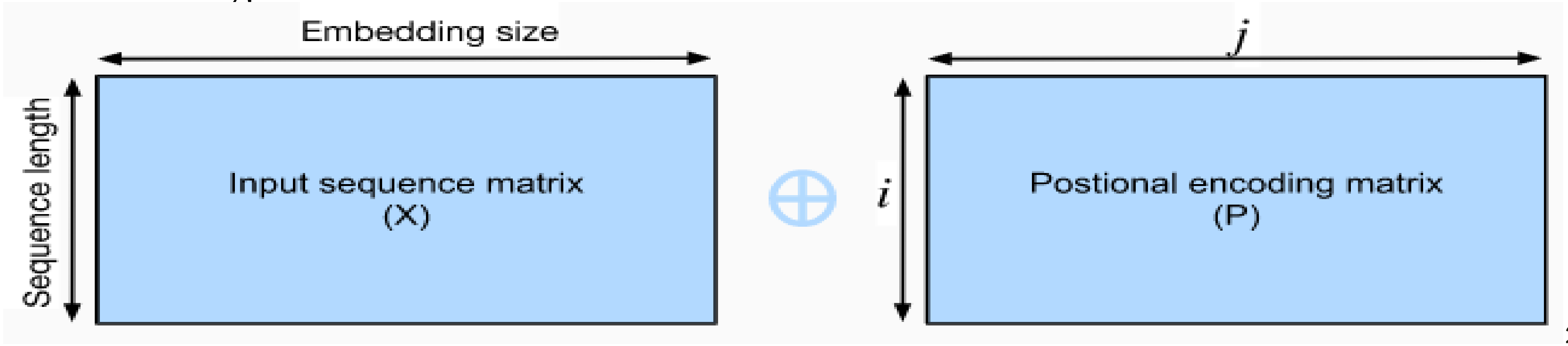
$$P_{i,2j} = \sin(i/10000^{2j/d}),$$

$$P_{i,2j+1} = \cos(i/10000^{2j/d}),$$

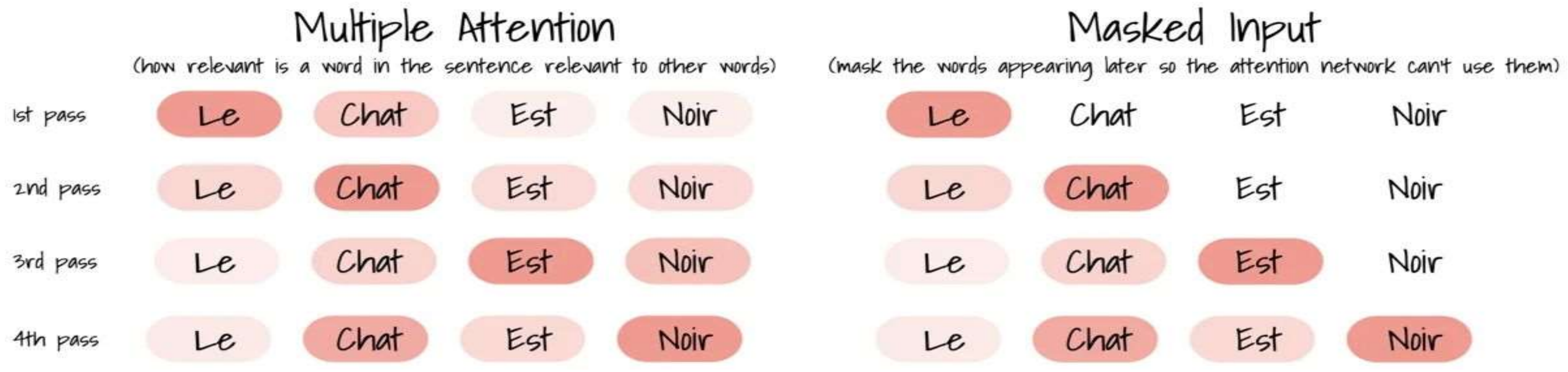
for $i = 0, \ldots, l-1$ and $j = 0, \ldots, \lfloor (d-1)/2 \rfloor$.

Assume that $X \in \mathbf{R}^{l\times d}$ is the embedding of an example, where l is the sequence length and d is the embedding size. This positional encoding layer encodes X's position $P \in \mathbf{R}^{l\times d}$ and outputs $P + X$

The position **P** is a 2-D matrix, where **i** refers to the order in the sentence, and **j** refers to the position along the embedding vector dimension



Decoder's Masked Multi-Head Attention



Multiple Attention
(how relevant is a word in the sentence relevant to other words)
Masked Input
(mask the words appearing later so the attention network can't use them)
1st pass
2nd pass
3rd pass
4th pass
Le
Chat
Est
Noir

Decoder's linear layer and softmax

The purpose of the Decoder is to predict the following word, the output size of this feed-forward layer is the number input words in the vocabulary.

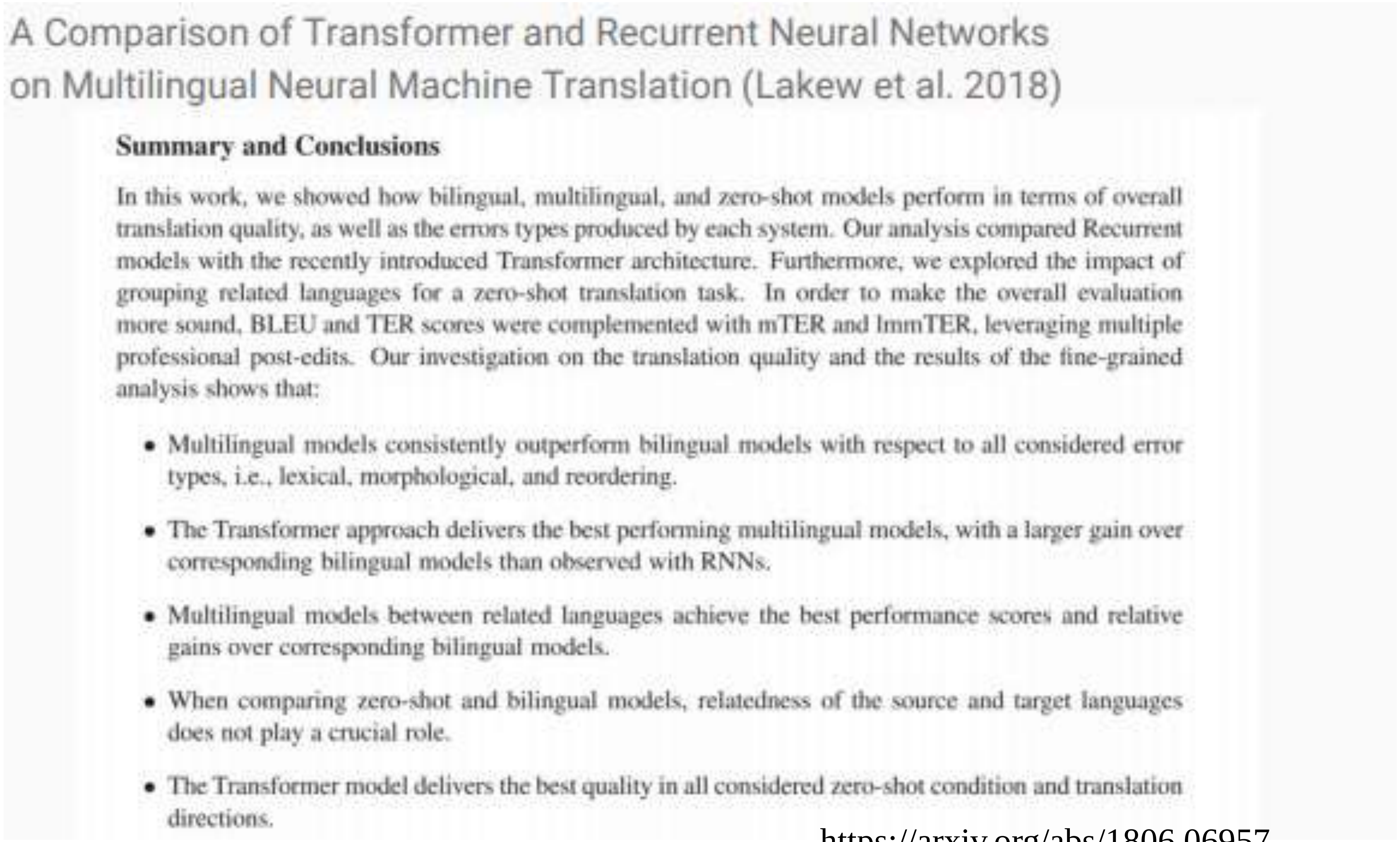
The Final Linear and Softmax Layer

The final Linear layer is followed by a Softmax Layer. The decoder stack outputs a vector of floats and the final Linear layer converts those in to words.

The Linear layer is a simple fully connected neural network that projects the vector produced by the stack of decoders, into a much, much larger vector called a logits vector.

- ❑ If we assume that our model knows 50,000 unique English words that it has learned from its training dataset. This would make the logits vector 50,000 cells wide – each cell corresponding to the score of a unique word. That is how we interpret the output of the model followed by the Linear layer.
- ❑ The softmax layer then turns those scores into probabilities **(all positive, all add up to 1.0)**. The cell with the highest probability is chosen, and the word associated with it is produced as the output for this time step.

Recent Works



A Comparison of Transformer and Recurrent Neural Networks on Multilingual Neural Machine Translation (Lakew et al. 2018)

Summary and Conclusions

In this work, we showed how bilingual, multilingual, and zero-shot models perform in terms of overall translation quality, as well as the errors types produced by each system. Our analysis compared Recurrent models with the recently introduced Transformer architecture. Furthermore, we explored the impact of grouping related languages for a zero-shot translation task. In order to make the overall evaluation more sound, BLEU and TER scores were complemented with mTER and lmmTER, leveraging multiple professional post-edits. Our investigation on the translation quality and the results of the fine-grained analysis shows that:

- Multilingual models consistently outperform bilingual models with respect to all considered error types, i.e., lexical, morphological, and reordering.
- The Transformer approach delivers the best performing multilingual models, with a larger gain over corresponding bilingual models than observed with RNNs.
- Multilingual models between related languages achieve the best performance scores and relative gains over corresponding bilingual models.
- When comparing zero-shot and bilingual models, relatedness of the source and target languages does not play a crucial role.
- The Transformer model delivers the best quality in all considered zero-shot condition and translation directions.

https://arxiv.org/abs/1806.06957

Recent Works

Universal Transformers (Dehghani et al. 2018)

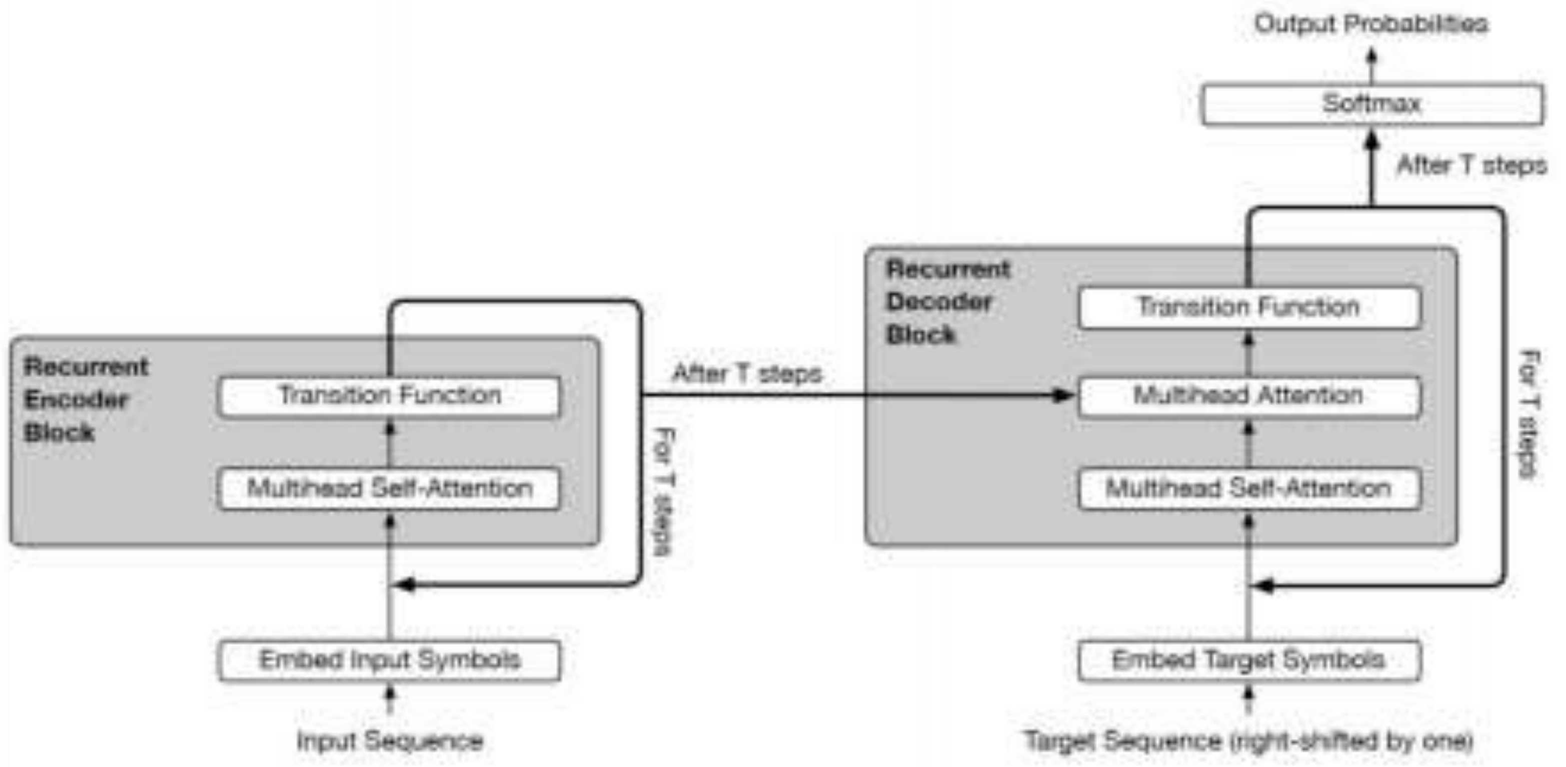


Figure 2: The recurrent blocks of the Universal Transformer encoder and decoder. This diagram omits position and time-step encodings as well as dropout, residual connections and layer normalization. A complete version can be found in the appendix. The Adaptive Universal Transformer dynamically determines the number of steps T for each position using ACT.

https://arxiv.org/abs/1807.03819

Difference with CNN

- The key difference between transformers and convolutional neural networks (CNN), is that transformers can simultaneously attend to every word of their input sequence.

- CNNs do not necessarily help with the problem of figuring out the problem of dependencies when translating sentences. That’s why Transformers were created, they are a combination of both CNNs with attention.

- CNN starts to look very locally and their receptive field becomes more global in each layer and after quite some training time, whereas Transformer succeeds at having large field of view from the very beginning.

- Transformers lack some of the inductive biases inherent to CNNs, such as translation equivariance and locality, and therefore do not generalize well when trained on insufficient amounts of data.

Difference with CNN

- For heavy weight state-of-the-art winners, energy consuming, environmental unfriendly, data hungry kind of AI transformer neural network may takeover CNN. However, in case of light weight machine learning in production, for computation and energy economic settings CNN maybe chosen over transformer neural network.

- Transformer is very data hungry, it needs huge data to learn how to focus and what to focus it’s attention to make the right predictions.

- The given focus pattern can be a limitation in case of CNN, where as due to having large field of view the possibility of making right prediction increases

- Transformer overcomes the limitations of CNN’s that look very narrowly at first, limitations which help when there’s not much training data.

Example of Its Usage

Transformer neural networks are useful for many sequence-related **deep learning** tasks, such as **machine translation**, **text generation, information retrieval**, **text classification**, **document summarization**, **image captioning**, and **genome analysis**.

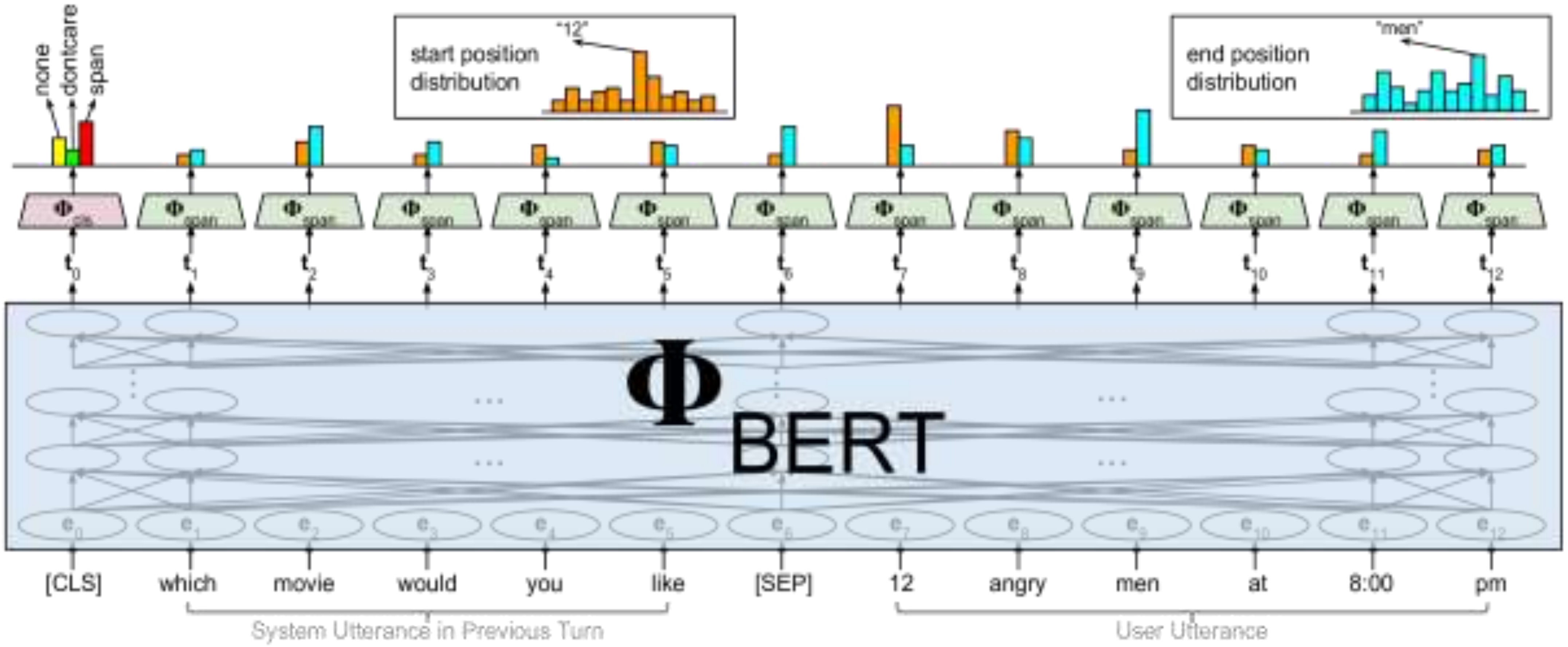
Transformer in Information Retrieval

- ❑ Google Search has begun to use Google's transformer neural network **BERT** from 2019 for search queries in over 70 languages.

- ❑ The introduction of transformer neural networks to Google Search means that queries where words such as **'from'** or **'to'** affect the meaning are better understood by Google. Users can search in more natural English rather than adapting their search query to what they think Google will understand.

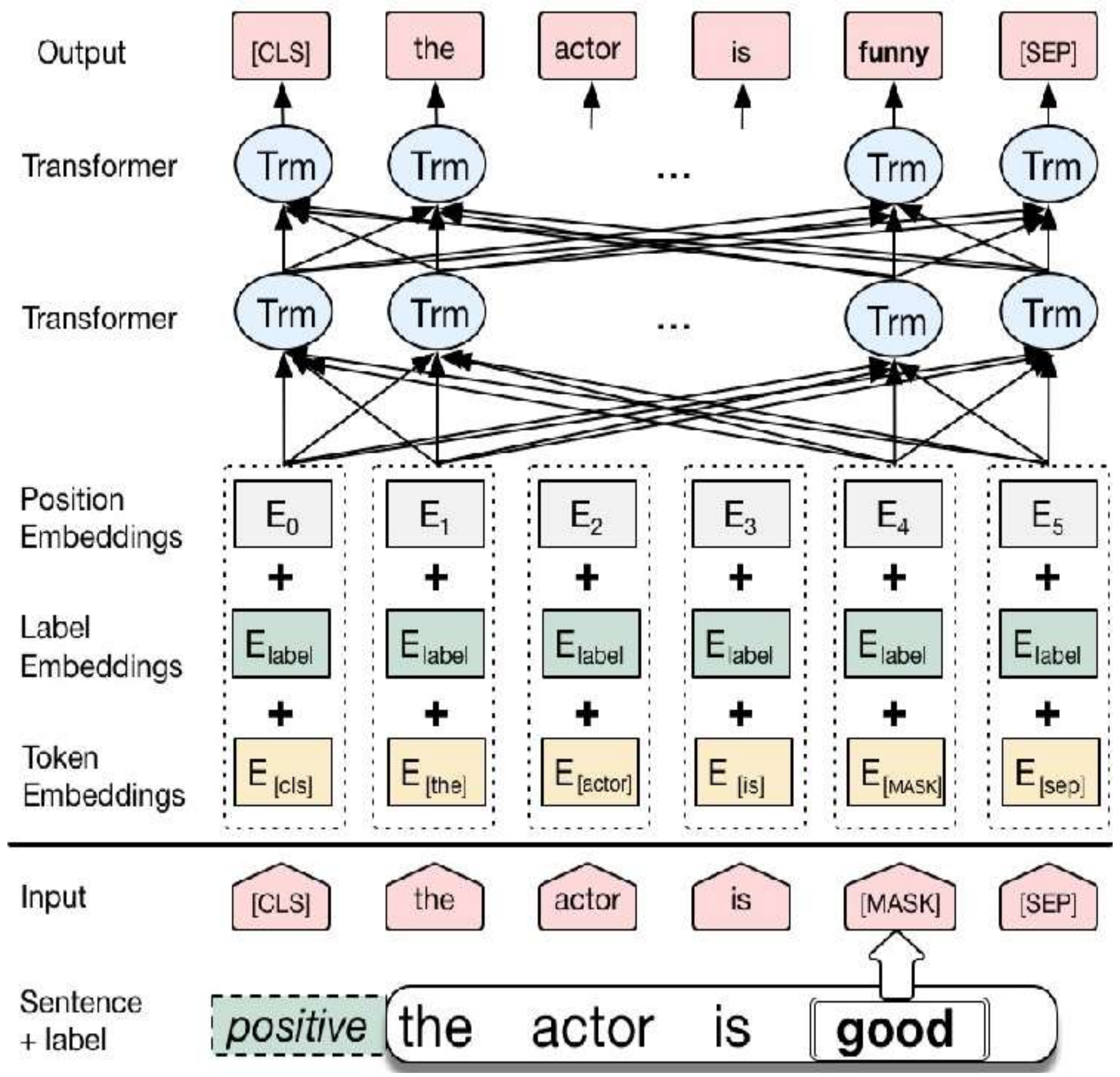
An example from Google's blog is the query "2019 brazil traveler to USA need a visa." The position of the word 'to' is very important for the correct interpretation of the query. The previous implementation of Google Search was not able to pick up this nuance and returned results about USA citizens traveling to Brazil, whereas the transformer model returns much more relevant pages.

BERT



none
dontcare
span
start position distribution
"12"
end position distribution
"men"
Φ_{cls}
Φ_{span}
t_0 t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9 t_{10} t_{11} t_{12}
Φ_{BERT}
e_0 e_1 e_2 e_3 e_4 e_5 e_6 e_7 e_8 e_9 e_{10} e_{11} e_{12}
[CLS]
which
movie
would
you
like
[SEP]
12
angry
men
at
8:00
pm
System Utterance in Previous Turn
User Utterance

BERT

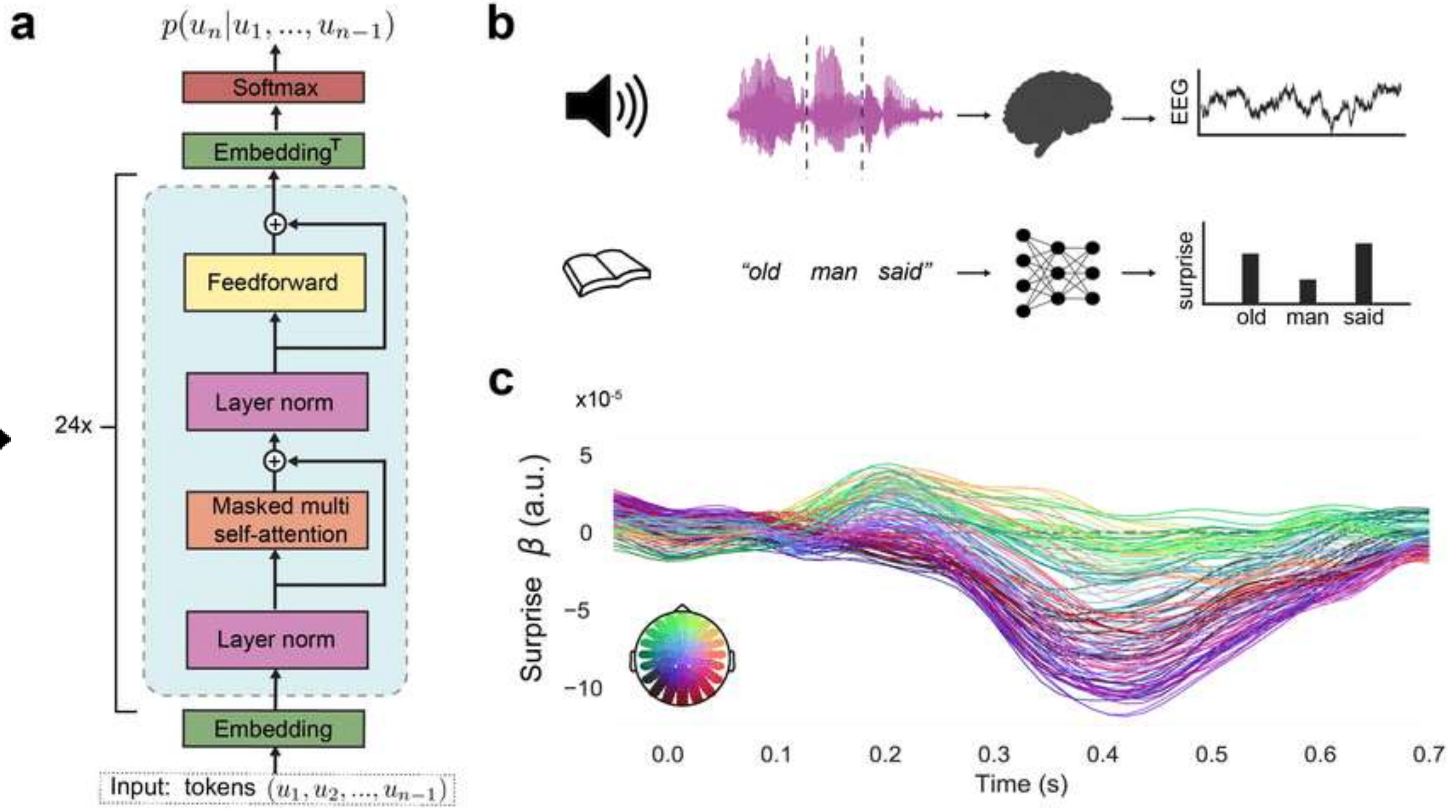

Output
[CLS]
the
actor
is
funny
[SEP]
Transformer
Trm
...
Transformer
Trm
...
Position Embeddings
E_0
E_1
E_2
E_3
E_4
E_5
Label Embeddings
E_{label}
Token Embeddings
$E_{[cls]}$
$E_{[the]}$
$E_{[actor]}$
$E_{[is]}$
$E_{[MASK]}$
$E_{[sep]}$
Input
[CLS]
the
actor
is
[MASK]
[SEP]
Sentence + label
positive
the actor is good

Transformer for Text Generation(GPT)

- OpenAI have demonstrated how their transformer models **GPT-2** and **GPT-3** can generate extremely humanlike texts.

- In their paper "***Fine-Tuning Language Models from Human Preferences***" , OpenAI introduced reinforcement learning instead of supervised learning to train a transformer neural network to generate text. In this set-up, the transformer neural network receives a 'reward' if it generates a continuation of the story which is judged pleasing to human readers.

GPT-2



Text Generation(Markov Transformer)

Deng and Rush proposed a probabilistic approach of text generation using Markov Transformer in their paper tittled "***Cascaded Text Generation with Markov Transformers***"
In this work, they exploit this property by proposing cascaded decoding with a Markov transformer architecture. Their approach centers around a graphical model representation of the output space of text generation.

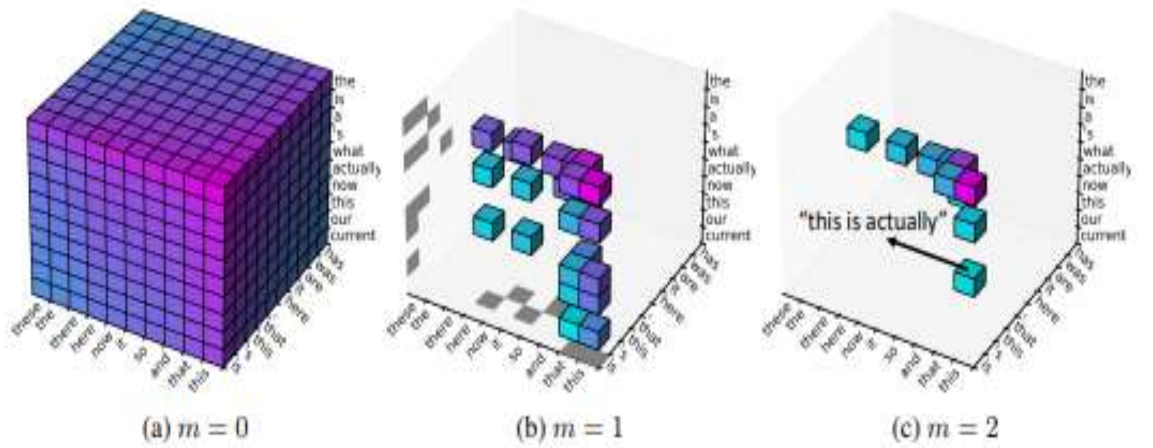


Figure 1: Illustration of cascaded decoding ($K = 10$, iters $= 4$) for $\mathcal{X}_1, \mathcal{X}_2, \mathcal{X}_3$. The axes correspond to x_1, x_2 and x_3. (a) 0th-order (non-autoregressive) model prunes unigrams to produce $\mathcal{X}_1$; (b) 1st-order model prunes bigrams to K per size-2 span (seen in 2D projection); (c) 2nd-order model prunes trigrams to K total in size-3 span. Colors represent max-marginals $\mathrm{MM}^{(m)}_{\mathcal{X}_m}(x_{1:3})$, with pink being higher and blue being lower. Fixed limit K allows for efficient parallel (GPU) implementation.

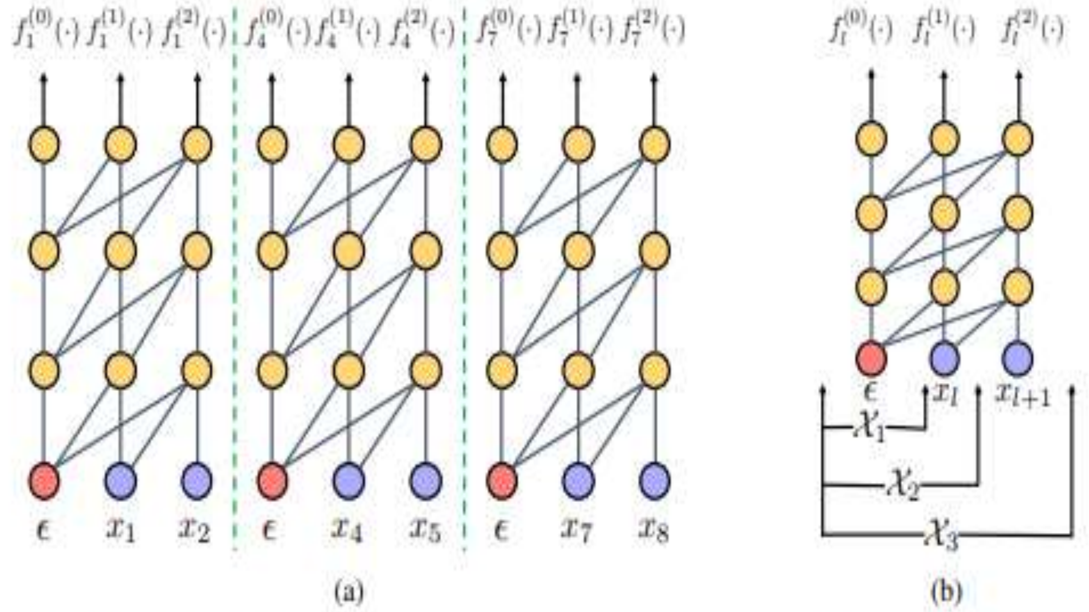

$f_1^{(0)}(\cdot)$ $f_1^{(1)}(\cdot)$ $f_1^{(2)}(\cdot)$
$f_4^{(0)}(\cdot)$ $f_4^{(1)}(\cdot)$ $f_4^{(2)}(\cdot)$
$f_7^{(0)}(\cdot)$ $f_7^{(1)}(\cdot)$ $f_7^{(2)}(\cdot)$
ϵ x_1 x_2 ϵ x_4 x_5 ϵ x_7 x_8
(a)
$f_l^{(0)}(\cdot)$ $f_l^{(1)}(\cdot)$ $f_l^{(2)}(\cdot)$
ϵ x_l x_{l+1}
$\mathcal{X}_1$
$\mathcal{X}_2$
$\mathcal{X}_3$
(b)

Figure 2: Markov transformer with $M = 2$ and $L = 9$. (a) At training, model state is reset with a barrier every $M + 1$ words. (b) At decoding, potential $f_l^{(0)}$ is computed at each position to get $\mathcal{X}_1$, and the dependency order is increased by introducing more columns to compute $\mathcal{X}_2$ and $\mathcal{X}_3$.

Thank You for your patience