Transformers (Encoder Block)

Zeham Management Technologies BootCamp by SDAIA

September 9th, 2024

Objectives

By the end of this module, trainees will have a comprehensive understanding of:

- ✓ Understand Transformer Architecture
  Apply Transformers to NLP Tasks
- ✓ Differentiate Between Recurrent-Based and Transformer Attention
- ✓ Evaluate Model Performance
- ✓ Fine-Tune Pre-trained Models
- ✓ Implement Custom Transformers
- ✓ Interpret Attention Mechanisms
- ✓ Optimize Hyperparameters

Agenda

- Introduction to transformers Architecture
- Tokenization
- Word Embeddings
- Positional Encoding
- Tokenization & Word Embedding in Transformers
- Multi-Head Self-Attention

Agenda

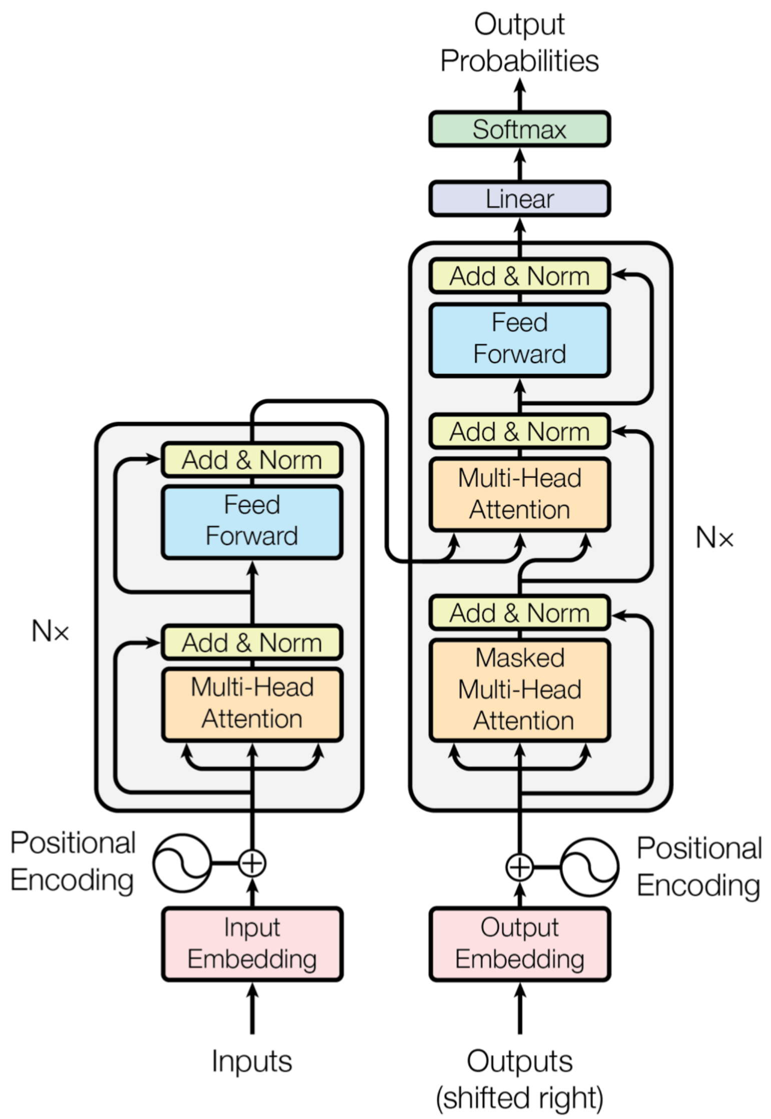
- Add&Norm
- Feed Forward
- Encoder
- Hello Transformer!

What are Transformers?

Transformers are special types of neural network architecture designed to handle sequential data, primarily used in natural language processing (NLP) tasks. Therefore, everything you have learned in the Neural Networks will apply here.

History and Development

Transformers were first introduced by Vaswani et al. in the seminal paper "Attention is All You Need" (2017), revolutionizing NLP by enabling models to handle longer contexts more effectively. It led to advanced models like BERT (2018), GPT (2018), and T5 (2019).



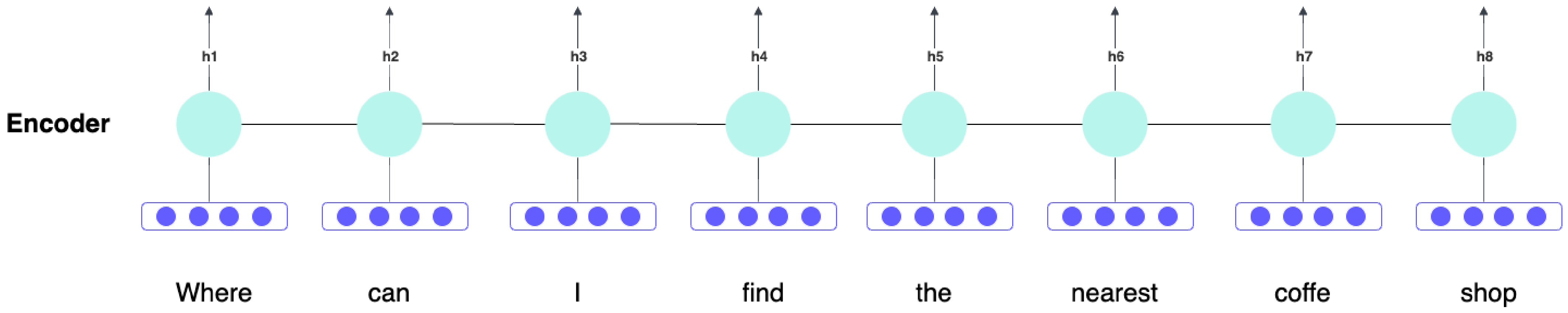
The Problem with Traditional Models

Limitations of RNNs and LSTMs:

- Struggle with long-range dependencies due to vanishing gradient problem.
- Sequential processing leads to slower training times.
- Difficulty in capturing contextual information over long distances in text.

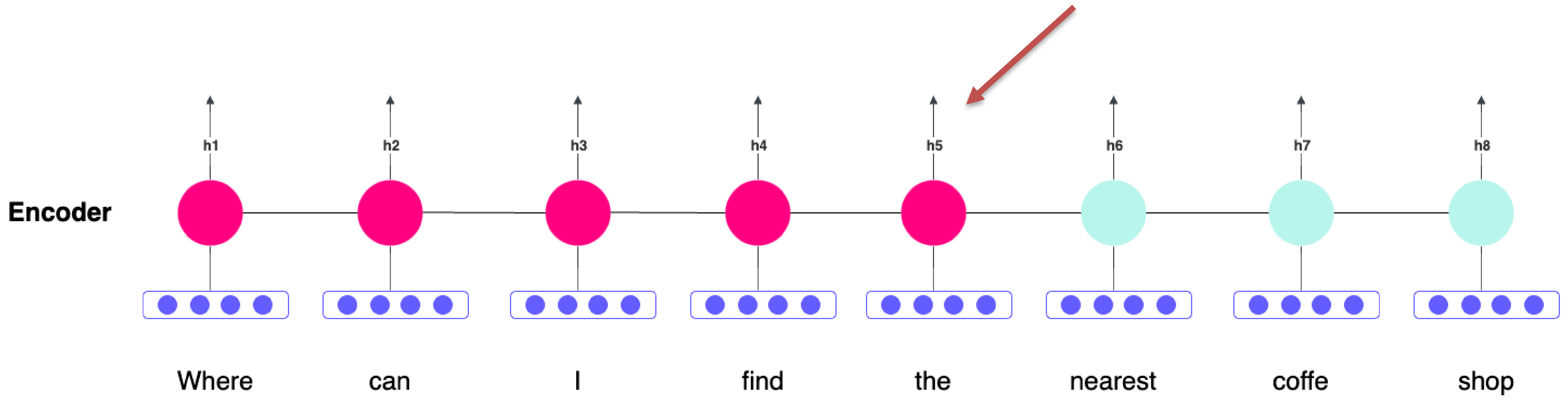
Challenges of Recurrent-Based Attention

To determine the current state of a time-step, RNN-based models must revisit all previous time steps, resulting in slower processing times, particularly with longer sequences due to their sequential nature.



Challenges of Recurrent-Based Attention

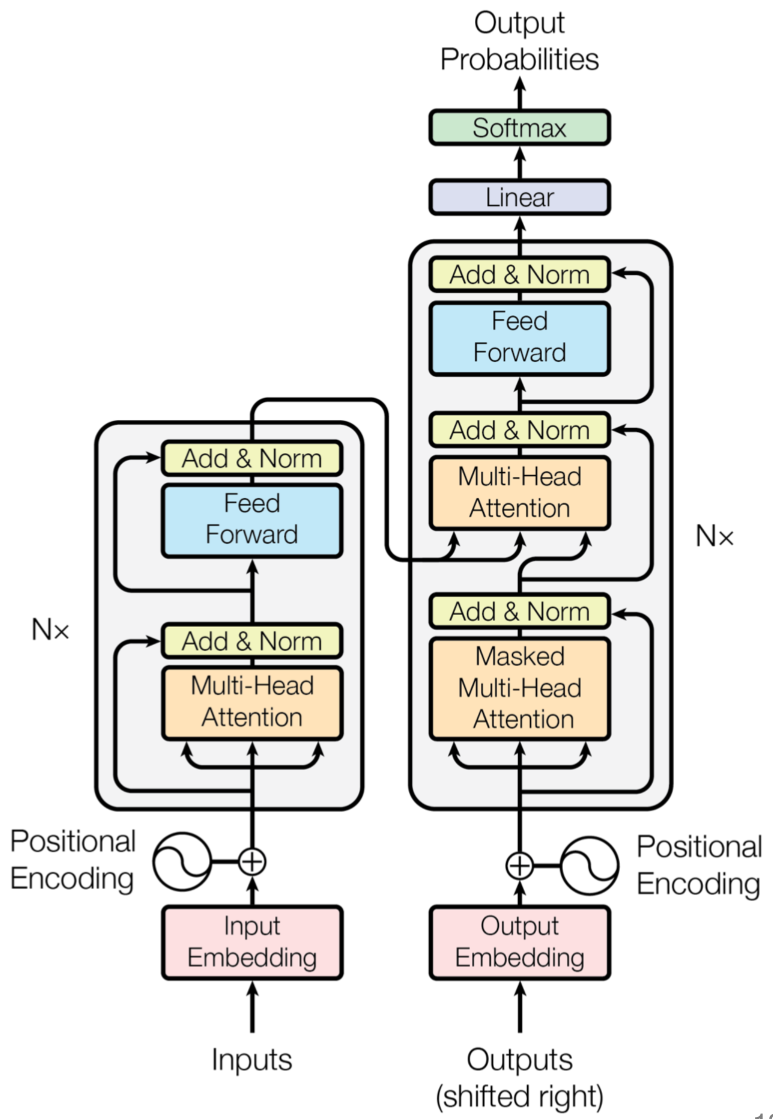
To compute h5 cell in this example, computations for all previous time-steps must be completed first.



Introduction to Transformer Architecture

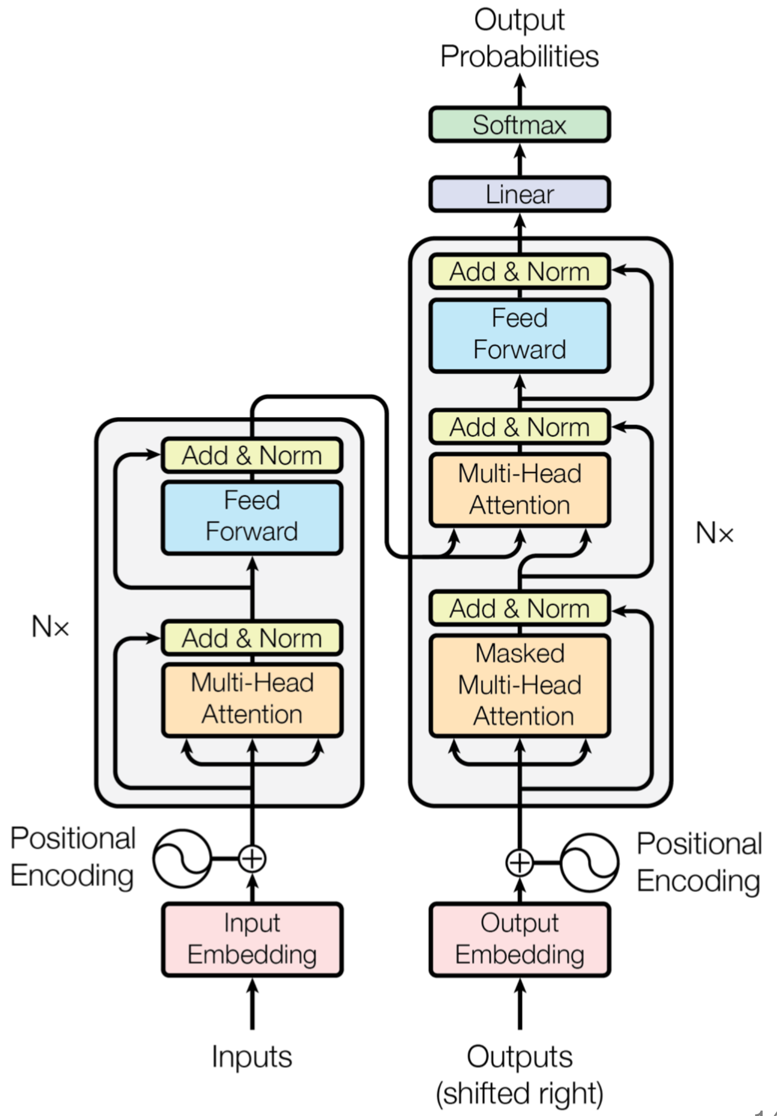
Overview of Transformer:

- **Input Embeddings**: Transforms input tokens into continuous vectors for the model's use.
- **Encoder**: Comprises multiple layers, each with a self-attention mechanism and a feed-forward neural network, to process input text.
- **Decoder:** Also contains multiple layers, each with a self-attention mechanism, an encoder-decoder attention mechanism, and a feed-forward neural network, to generate output text.



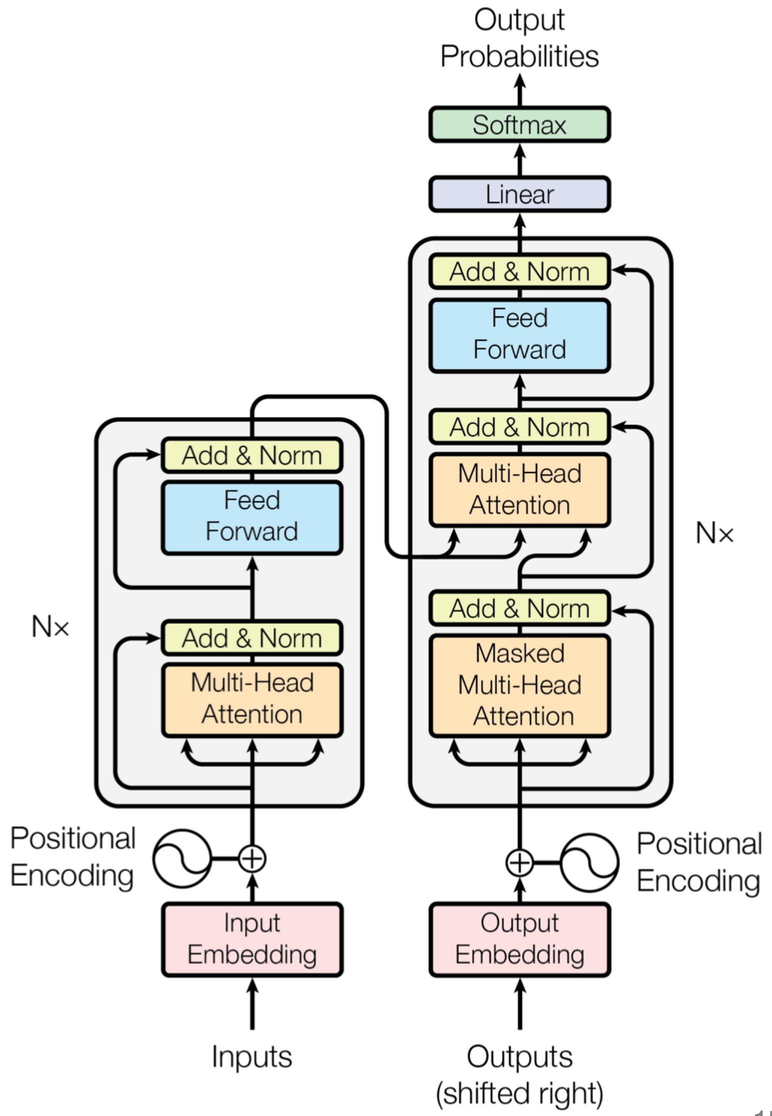
Overview of Transformer:

- **Self-attention mechanism:** Aids the model in understanding word relationships in a sentence, even across long distances.
- **Positional encoding:** Added to input embeddings to provide the model with a sense of word order in the sequence.
- **Layer normalization:** Technique used within encoder and decoder layers to stabilize the training process.
- **Residual connections:** Used in encoder and decoder layers to improve gradient flow during training and address the vanishing gradient problem.



Overview of Transformer:

- **Residual connections:** Used in encoder and decoder layers to improve gradient flow during training and address the vanishing gradient problem.
- **Encoder-decoder attention mechanism**: Used in the decoder to focus on relevant parts of the input when generating the output.
- **Output linear layer**: Converts decoder output into logits for each token in the target vocabulary.
- **SoftMax layer:** Applied to logits to produce probabilities for each token in the target vocabulary.



Tokenization

Tokenization recap

Tokenization is the process of breaking down text into smaller units called tokens. It enables machines to process and understand text data.

Character Tokenization:

Character Tokenization treats each character as a separate token.
Example: "Hello" becomes ["H", "e", "l", "l", "o"]

- Pros:
    - Useful for languages with complex scripts.
    - Maintains information about character-level details.
    - It handles out of vocabulary words.
- Cons:
    - Increases sequence length, leading to higher computational complexity.

Tokenization recap

Tokenization is the process of breaking down text into smaller units called tokens. It enables machines to process and understand text data.

Word Tokenization:

Word Tokenization handles text at the word level, often using white space as a delimiter.

- Example: "Hello, world!" becomes ["Hello", ",", "world", "!"]
- Pros:
  - Preserves most semantic meaning of words.
  - Straightforward for many languages.
- Cons:
  - Can be ambiguous with certain punctuation (e.g., "don't" -> ["do", "n't"]).
  - Doesn’t handle out of vocabulary words including slang, new words, misspellings etc..

Tokenization recap

Tokenization is the process of breaking down text into smaller units called tokens. It enables machines to process and understand text data.

Subword Tokenization:

Subword Tokenization breaks rare words into smaller units but keep the frequent words as is (such as she, he, up etc..) , useful for languages with complex morphology. With the transformer this was the middle ground approach used.
Example: "annoyingly" becomes ["annoying", ”ly"] where “annoyingly” composes two words and it is not frequently used.

- Pros:
  - Handles out-of-vocabulary words well.
  - Captures morphological information.
- Cons:
  - Increases vocabulary size, leading to higher computational costs.

Tokenization recap

Tokenization is the process of breaking down text into smaller units called tokens. It enables machines to process and understand text data.

Byte Pair Encoding (BPE):
Byte Pair Encoding (BPE) is a subword tokenization technique that splits words into smaller subword units based on frequency. It’s widely used in NLP models like GPT and BERT to handle out-of-vocabulary words by breaking them down into subwords that can be pieced together.

Tokenization recap cont.

Tokenization is the process of breaking down text into smaller units called tokens. It enables machines to process and understand text data.

Byte Pair Encoding (BPE):
Example:
Let’s consider a small toy vocabulary with a few words:
- low
- lower
- newest
- widest

We start by treating each character as a separate symbol, and we continuously merge the most frequent pair of symbols.

Tokenization recap cont.

Tokenization is the process of breaking down text into smaller units called tokens. It enables machines to process and understand text data.

Byte Pair Encoding (BPE):
Step 1: Initialize the vocabulary
Each word is represented as a sequence of characters with an end-of-word marker (_):
l o w _
l o w e r _
n e w e s t _
w i d e s t _

Tokenization recap cont.

Tokenization is the process of breaking down text into smaller units called tokens. It enables machines to process and understand text data.

Byte Pair Encoding (BPE):
Step 2: Count symbol pairs
Next, we count the occurrences of every consecutive symbol pair across the entire corpus.
For example:
('l', 'o') -> 2 times
('o', 'w') -> 2 times
('e', 's') -> 2 times
('w', 'i') -> 1 time
...

Tokenization recap cont.

Tokenization is the process of breaking down text into smaller units called tokens. It enables machines to process and understand text data.

Byte Pair Encoding (BPE):
Step 3: Merge the most frequent pair
We find the most frequent pair and merge it into a new symbol. In this case, let's assume ('l', 'o') is the most frequent pair. After merging, the vocabulary looks like this:
lo w _
lo w e r _
n e w e s t _
w i d e s t _

Tokenization recap cont.

Tokenization is the process of breaking down text into smaller units called tokens. It enables machines to process and understand text data.

Byte Pair Encoding (BPE):
Step 4: Repeat the process
We repeat the process by counting pairs again and merging the next most frequent one. Let’s say the next most frequent pair is ('l', 'o'), which becomes low. After merging, the vocabulary updates like this:
low _
low e r _
n e w e s t _
w i d e s t _

Tokenization recap cont.

Tokenization is the process of breaking down text into smaller units called tokens. It enables machines to process and understand text data.

Byte Pair Encoding (BPE):
Final Result
This process repeats until we reach a predetermined number of merges or all frequent pairs are exhausted. The final result would be a set of subword units that can capture frequent patterns, enabling the model to deal with words that aren't in the original vocabulary by combining these subword units.

Tokenization recap cont.

Tokenization is the process of breaking down text into smaller units called tokens. It enables machines to process and understand text data.

Byte Pair Encoding (BPE):

Pros:

- **Efficient handling of OOV words:** BPE enables the model to break down rare or unknown words into smaller, recognizable subwords.
- **Compact vocabulary:** The vocabulary size is smaller than character-level tokenization while still capturing useful subword information.

Cons:

- **Limited interpretability:** Breaking down words into subword units can reduce the interpretability of tokenized outputs.
- **Vocabulary growth:** While smaller than character-level vocabularies, BPE can still result in a larger vocabulary than pure word-based tokenization, increasing computational overhead.

Tokenizer Tutorial

```
Tokenizer.ipynb
Tokenizer_Exercise.ipynb
```

Word Embedding Recap

Word Embedding

Word Embeddings in NLP is a technique where individual words are represented as real-valued vectors in a lower-dimensional space and captures inter-word semantics. These representations are essential in Natural Language Processing (NLP) tasks, as they enable algorithms to process and understand language more effectively.

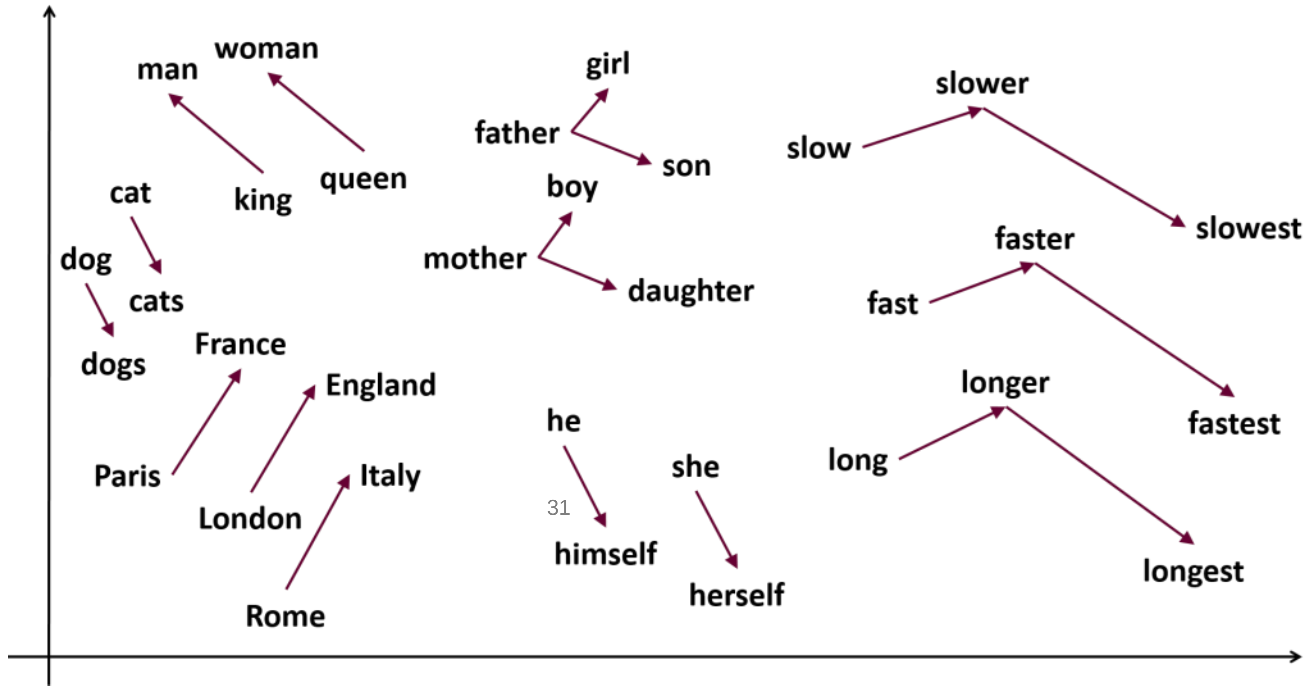
Word Embedding

Word Embedding is a language modeling technique for mapping words to vectors of real numbers. It represents words or phrases in vector space with several dimensions.

- Word embeddings can be generated using various methods like neural networks, co-occurrence matrices, probabilistic models, etc.

- Word2Vec consists of models for generating word embedding. These models are shallow two-layer neural networks having one input layer, one hidden layer, and one output layer.

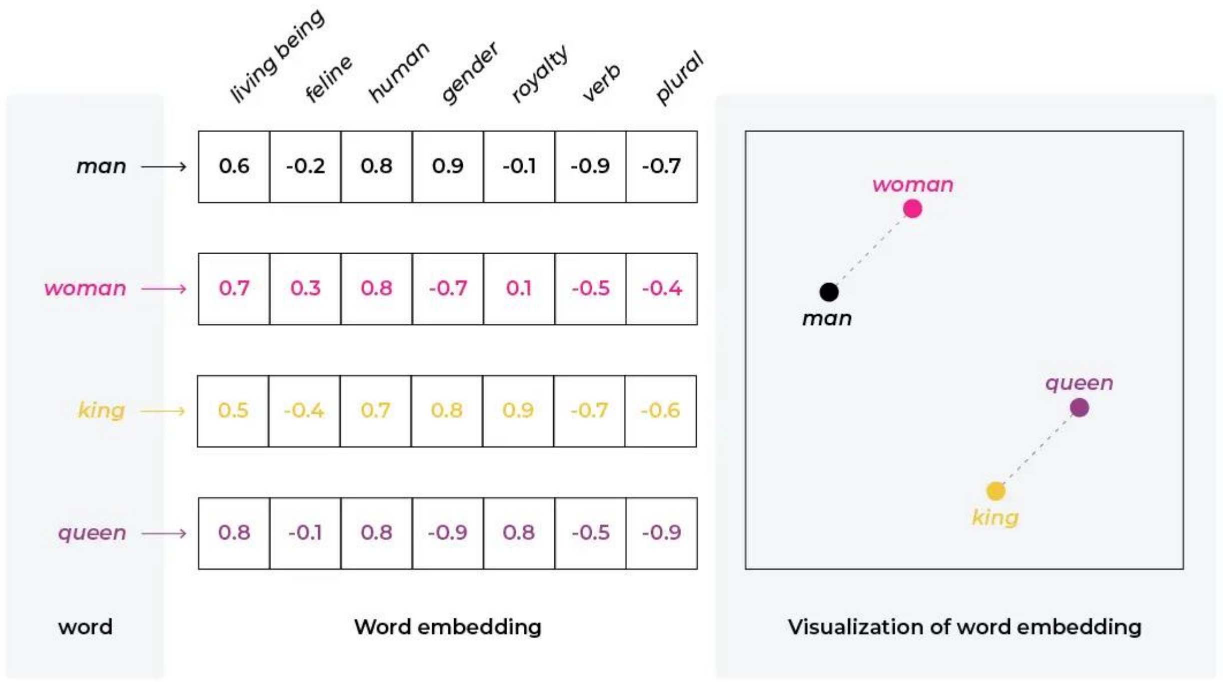
- **Word2vec was created, patented, and published in 2013 by a team of researchers led by Tomas Mikolov at Google.**



Need for Word Embedding

Word Embedding is needed in NLP and Transformers for several reasons including:

- Lower dimensions
- To use word to predict the surrounded words
- Capture semantic meanings



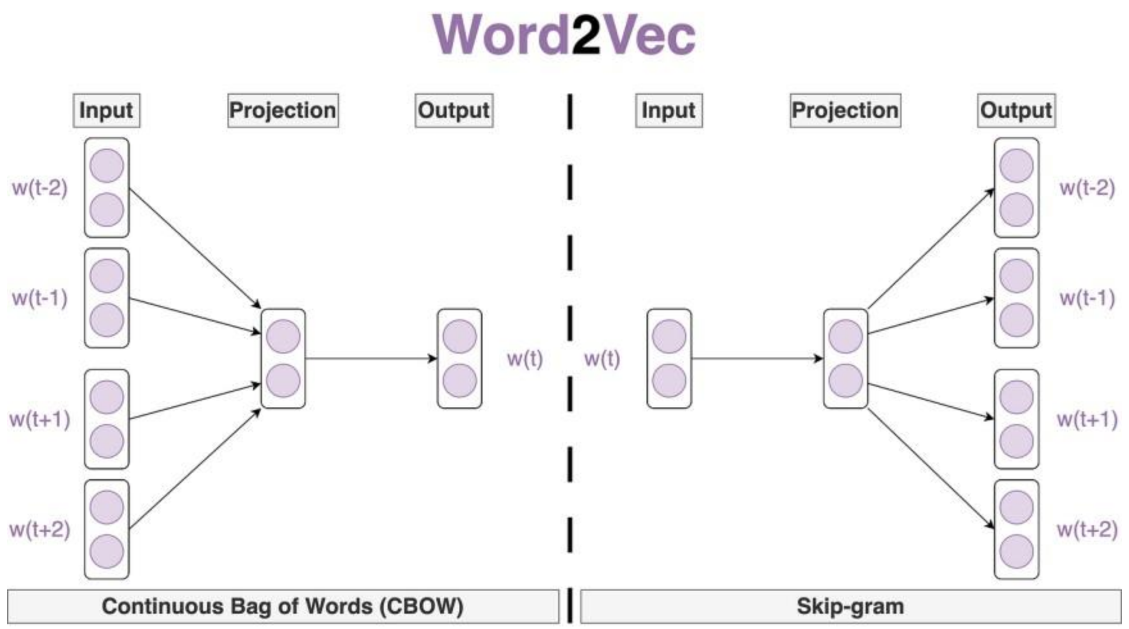
word	living being	feline	human	gender	royalty	verb	plural
man	0.6	-0.2	0.8	0.9	-0.1	-0.9	-0.7
woman	0.7	0.3	0.8	-0.7	0.1	-0.5	-0.4
king	0.5	-0.4	0.7	0.8	0.9	-0.7	-0.6
queen	0.8	-0.1	0.8	-0.9	0.8	-0.5	-0.9

Word2Vec

Word2Vec is a widely used method in natural language processing (NLP) that allows words to be represented as vectors in a continuous vector space.

- Word2Vec is an effort to map words to high-dimensional vectors to capture the semantic relationships between words, developed by researchers at Google.
- For example, it can identify relations like country-capital over larger datasets showing us how powerful word embeddings can be.
- Words with similar meanings should have similar vector representations, according to the main principle of Word2Vec.
- There are two neural embedding methods for Word2Vec:

1. **CBOW (Continuous Bag of Words)**
2. **Skip Gram**

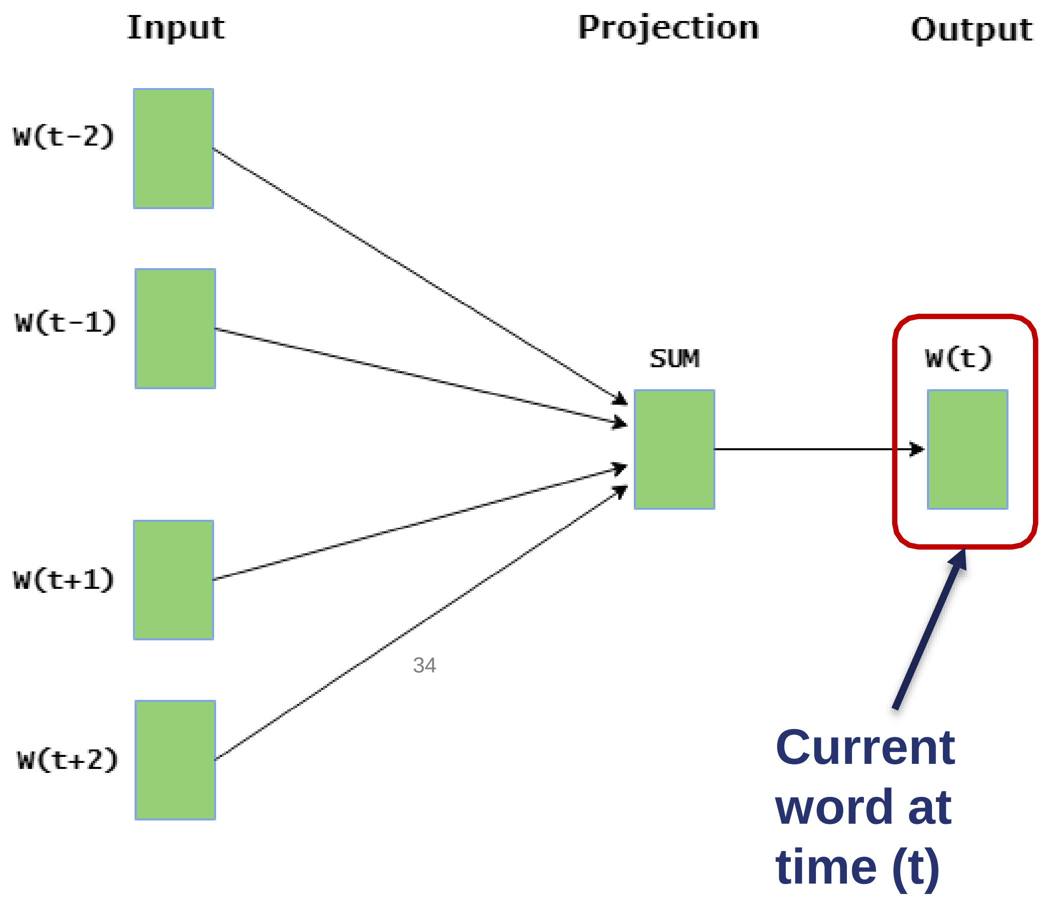


Word2Vec: CBOW (Continuous Bag of Words)

1. **CBOW (Continuous Bag of Words)**

The CBOW model predicts the current word given context words within a specific window.

- The **input layer** contains the context words.
- The **output layer** contains the current word.
- The **hidden layer** contains the dimensions we want to represent the current word present at the output layer.

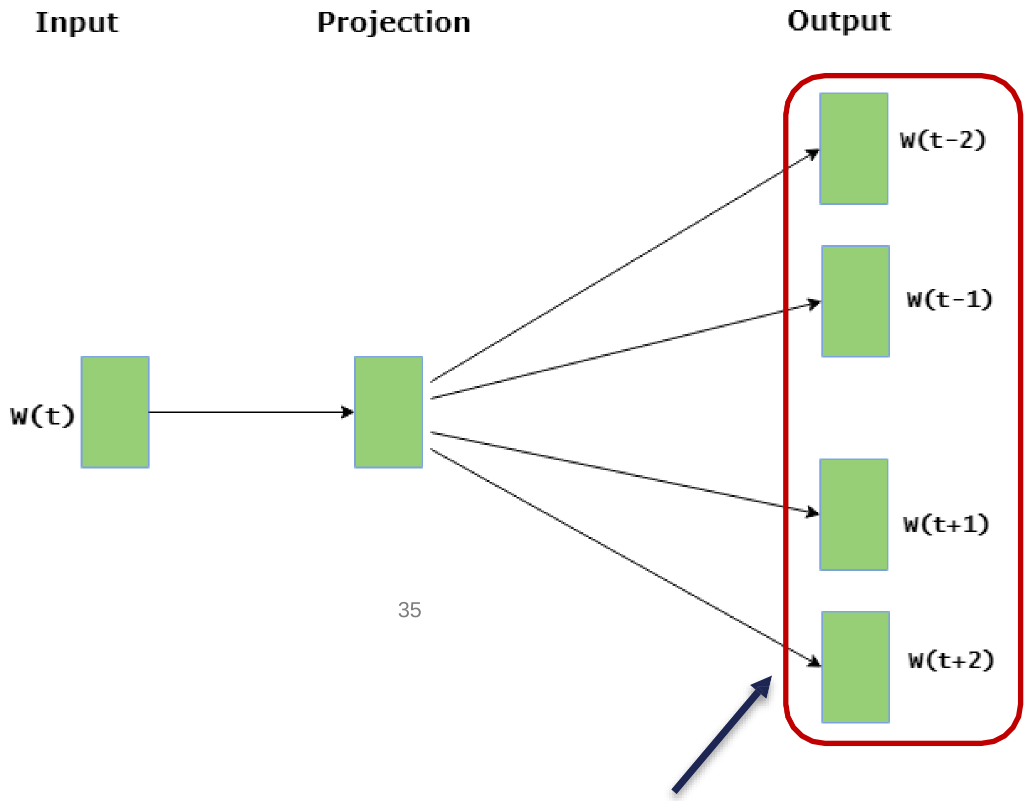


Word2Vec: Skip Gram

2. **Skip Gram**

Skip gram predicts the surrounding context words within specific window given current word.

- The **input layer** contains the current word.
- The **output layer** contains the context words.
- The **hidden layer** contains the number of dimensions in which we want to represent current word present at the input layer.

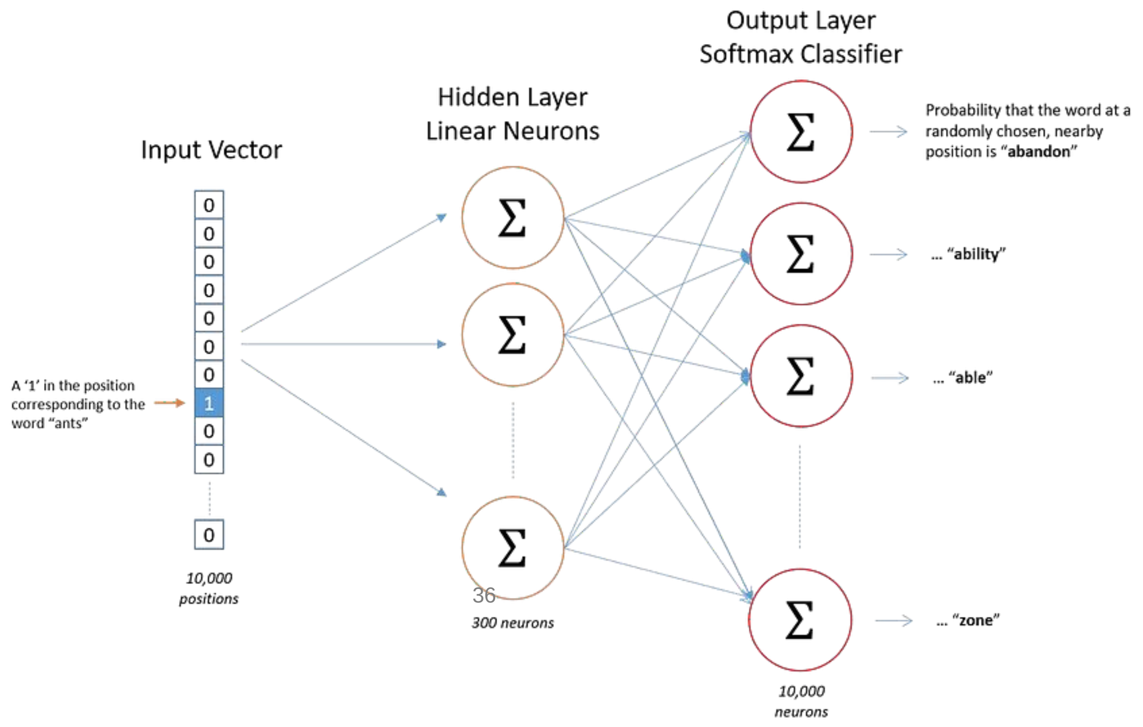


Predict words before (t-1, t-2) and after (t+1, t+2) word at time (t)

Word2Vec Model Architecture

Word2Vec essentially is a shallow 2-layer neural network trained.

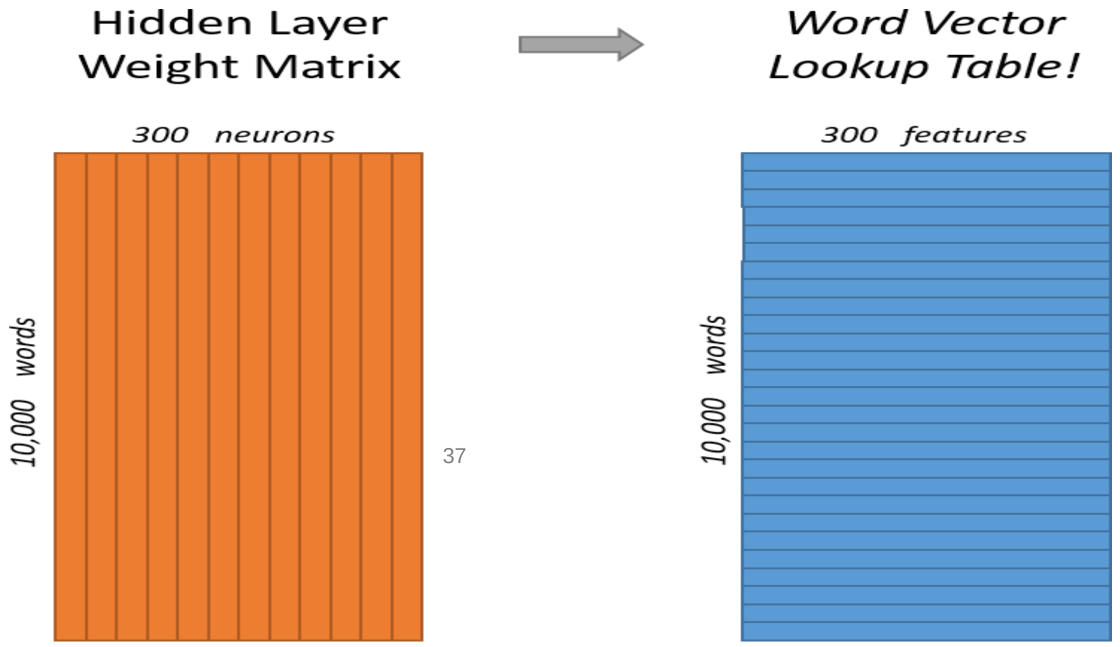
- The input contains all the documents/texts in our training set.
- For the network to process these texts, they are represented in a 1-hot encoding of the words.
- The number of neurons present in the hidden layer is equal to the length of the embedding you want.
- That is, if we want all our words to be vectors of length 300, then the hidden layer will contain 300 neurons.



Word2Vec Model Architecture *(cont'd)*

The output layer contains probabilities for a target word (given an input to the model, what word is expected) given a particular input.

- At the end of the training process, the hidden weights are treated as the word embedding.
- Intuitively, this can be thought of as each word having a set of n weights *(300 considering the example)* "weighing" their different characteristics.



Implementing Word2Vec

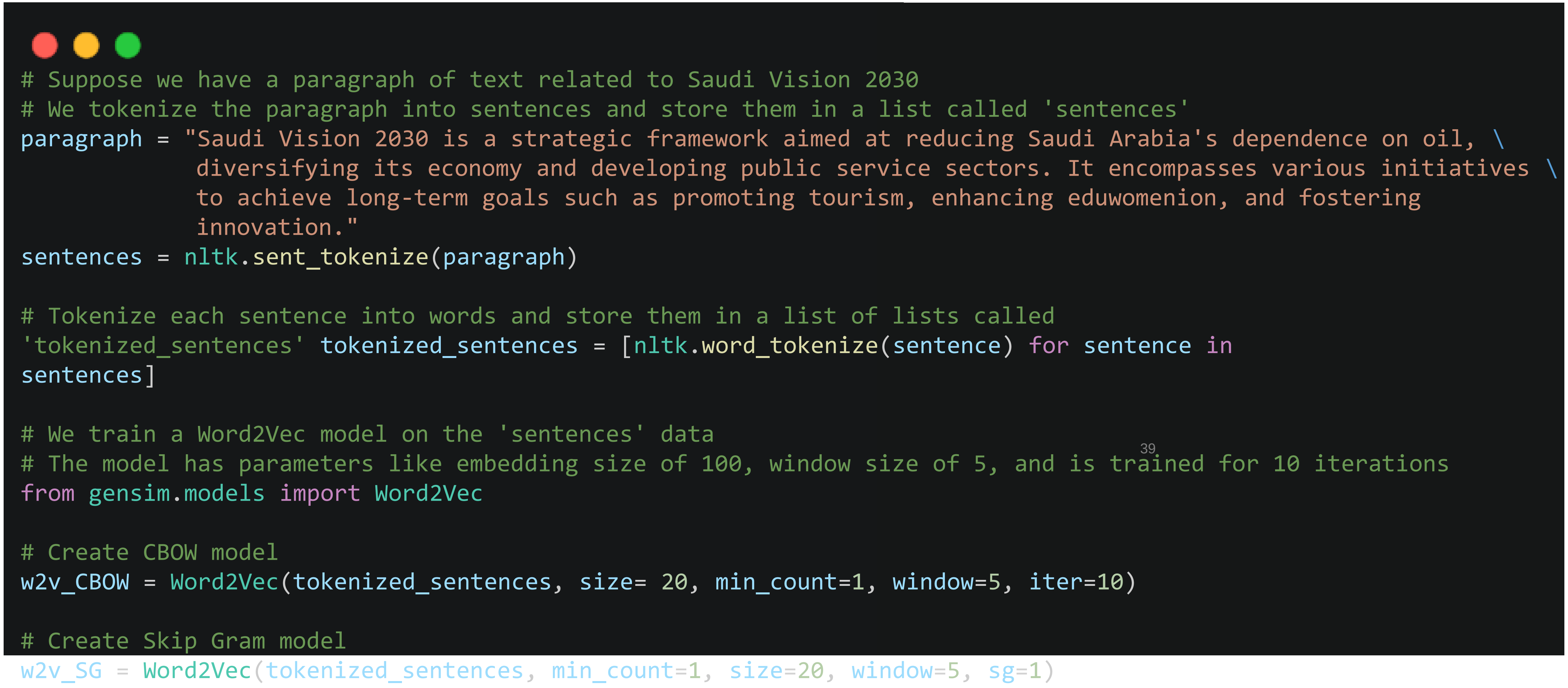
1. **Training the embeddings:**

- Word2Vec is imported from the gensim.model library. Each input to the model must be a list of phrases, so the input to the model is generated by using the split() function on each line in the corpus of texts.
- The model is then set up with various parameters, with a brief explanation of their meanings:

    1. **Size** refers to the size of the word embedding that it would output.
    2. **Window** refers to the maximum distance between the current and predicted word within a sentence.
    3. The **min_count** parameter is used to set a minimum frequency for the words to be a part of the model, i.e. it ignores all words with count less than **min_count**.
    4. **iter** refers to the number of iterations for training the model.

Implementing Word2Vec *(cont'd)*

1. **Training the embeddings:**



```
# Suppose we have a paragraph of text related to Saudi Vision 2030
# We tokenize the paragraph into sentences and store them in a list called 'sentences'
paragraph = "Saudi Vision 2030 is a strategic framework aimed at reducing Saudi Arabia's dependence on oil, \
             diversifying its economy and developing public service sectors. It encompasses various initiatives \
             to achieve long-term goals such as promoting tourism, enhancing eduwomenion, and fostering
             innovation."
sentences = nltk.sent_tokenize(paragraph)

# Tokenize each sentence into words and store them in a list of lists called
'tokenized_sentences' tokenized_sentences = [nltk.word_tokenize(sentence) for sentence in
sentences]

# We train a Word2Vec model on the 'sentences' data
# The model has parameters like embedding size of 100, window size of 5, and is trained for 10 iterations
from gensim.models import Word2Vec

# Create CBOW model
w2v_CBOW = Word2Vec(tokenized_sentences, size= 20, min_count=1, window=5, iter=10)

# Create Skip Gram model
w2v_SG = Word2Vec(tokenized_sentences, min_count=1, size=20, window=5, sg=1)
```

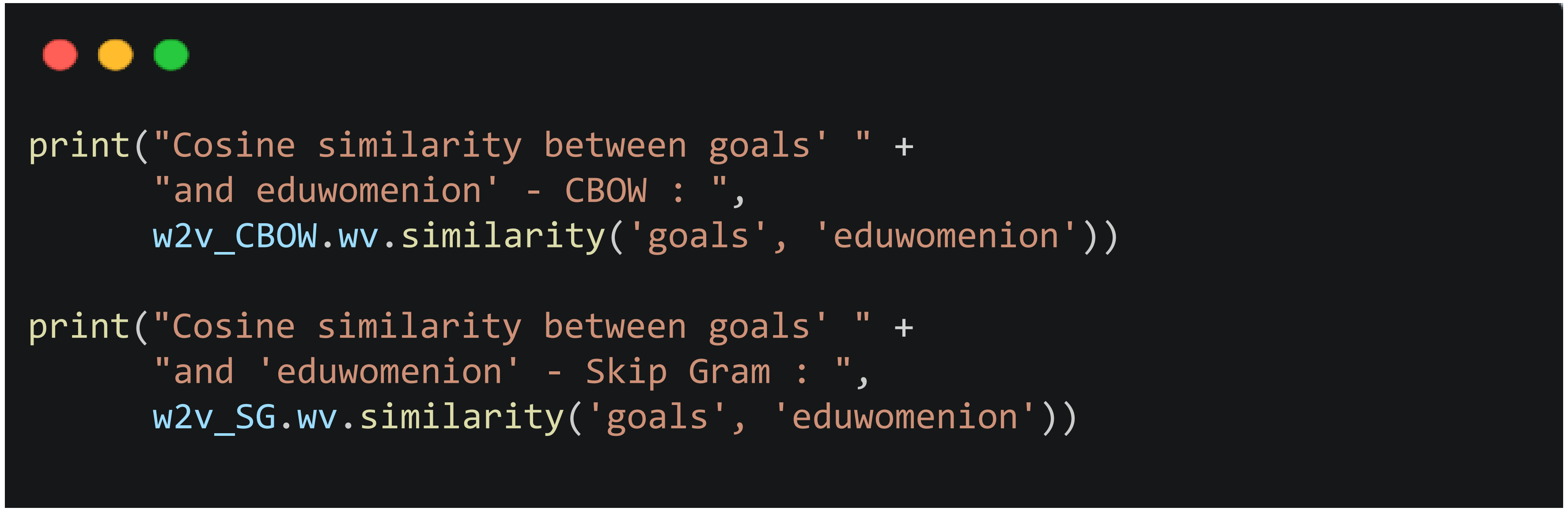
Implementing Word2Vec *(cont'd)*

2. **Using the word2vec model:**

- Finding the vocabulary of the model can be useful in several general applications, and in this case, provides us with a list of words we can try and use other functions.
- Finding the embedding of a given word can be useful when we're trying to represent sentences as a collection of word embeddings, like when we're trying to make a weight matrix for the embedding layer of a network. I included this so that it can help your intuition of what the word vector looks like.
- We can also find out the similarity between given words *(the cosine distance between their vectors)*. Here we have tried **'goals'** and **'eduwomenion'** and compared it with CBOW and Skip-Gram seeing how a stark distinction exists.

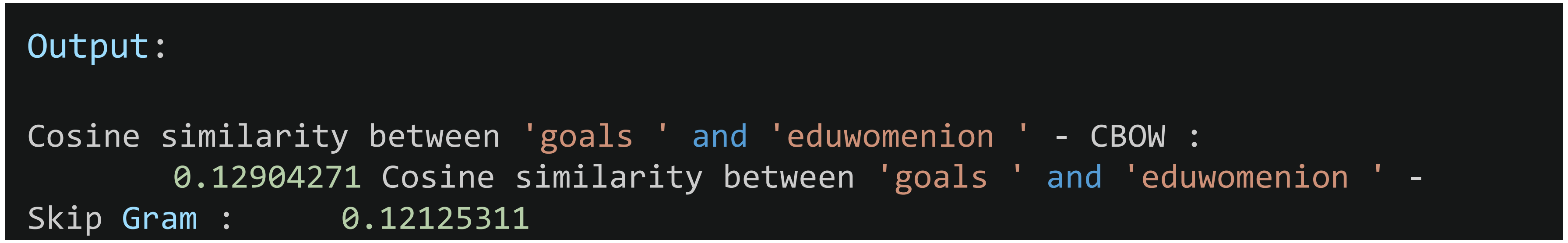
Implementing Word2Vec *(cont’d)*

2. **Using the word2vec model:**



```
print("Cosine similarity between goals' " +
      "and eduwomenion' - CBOW : ",
      w2v_CBOW.wv.similarity('goals', 'eduwomenion'))

print("Cosine similarity between goals' " +
      "and 'eduwomenion' - Skip Gram : ",
      w2v_SG.wv.similarity('goals', 'eduwomenion'))
```



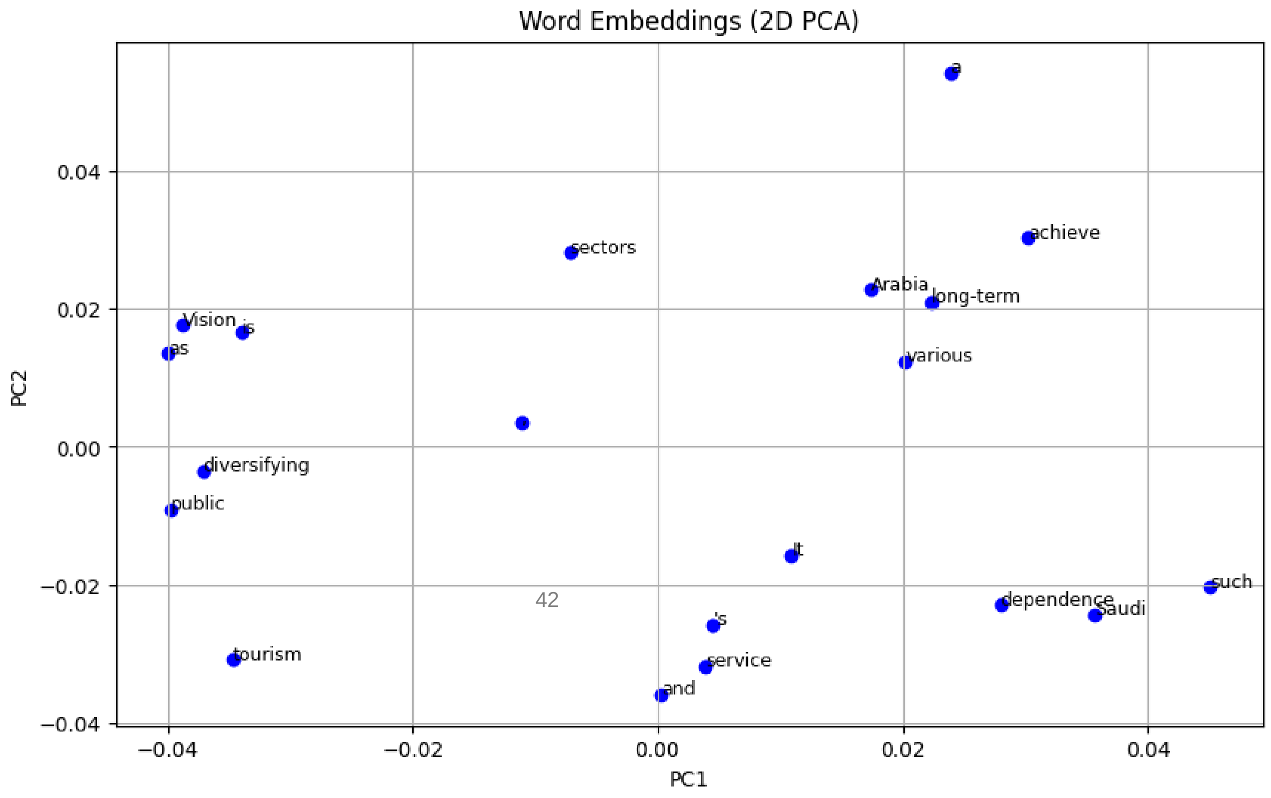
```
Output:

Cosine similarity between 'goals ' and 'eduwomenion ' - CBOW :
       0.12904271 Cosine similarity between 'goals ' and 'eduwomenion ' -
Skip Gram :      0.12125311
```

Implementing Word2Vec *(cont'd)*

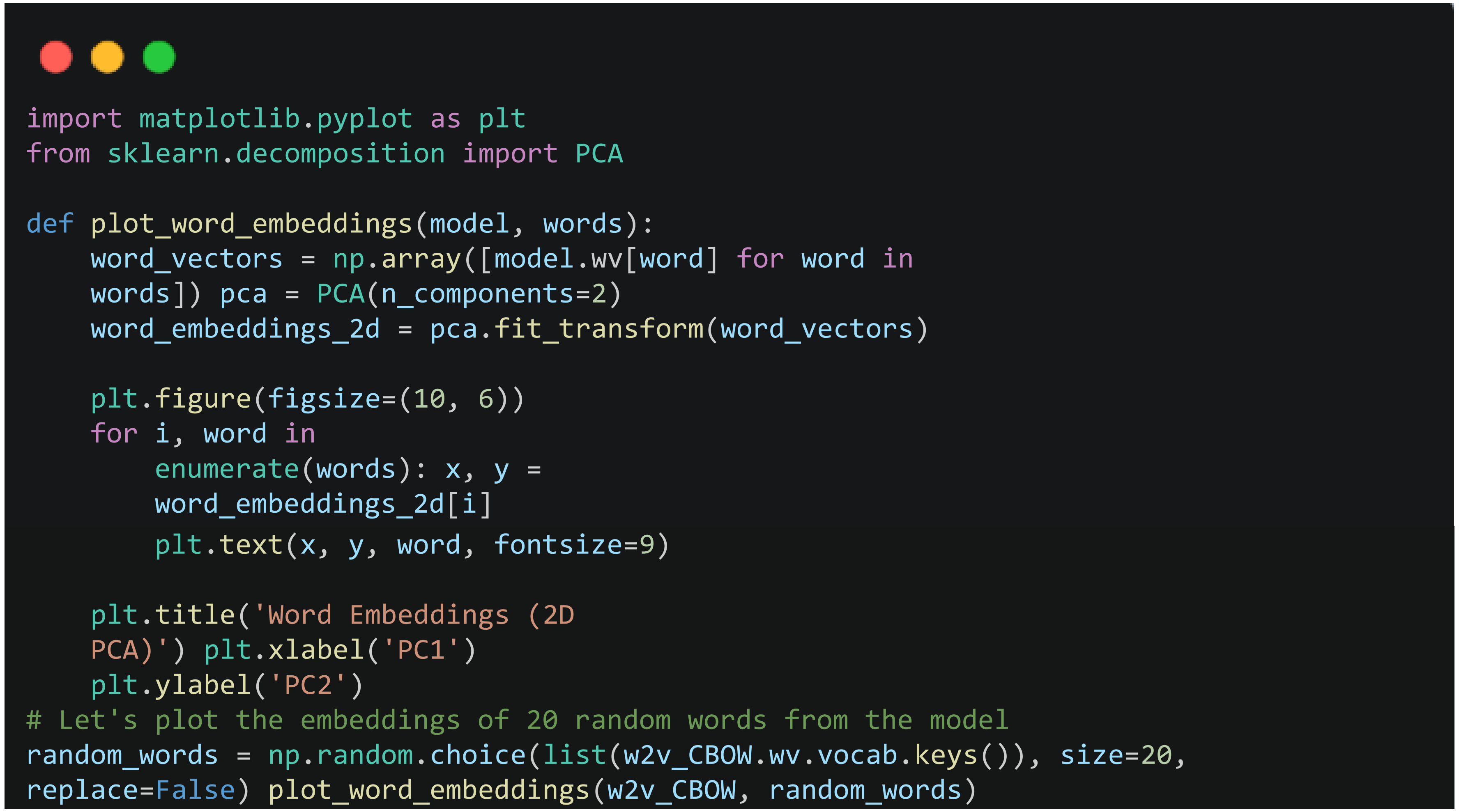
3. **Visualizing word embeddings:**

- Word2Vec word embedding can usually be of sizes 100 or 300, and it is practically not possible to visualize a 300- or 100- dimensional space with meaningful outputs.
- In either case, it uses PCA to reduce the dimensionality and represent the word through their vectors on a 2-dimensional plane.
- **The actual values of the axis are not of concern as they do not hold any significance, rather we can use it to perceive similar vectors with respect to each other.**



Implementing Word2Vec *(cont'd)*

3. **Visualizing word embeddings:**



```
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA

def plot_word_embeddings(model, words):
    word_vectors = np.array([model.wv[word] for word in
    words]) pca = PCA(n_components=2)
    word_embeddings_2d = pca.fit_transform(word_vectors)

    plt.figure(figsize=(10, 6))
    for i, word in
        enumerate(words): x, y =
        word_embeddings_2d[i]
        plt.text(x, y, word, fontsize=9)

    plt.title('Word Embeddings (2D
    PCA)') plt.xlabel('PC1')
    plt.ylabel('PC2')
# Let's plot the embeddings of 20 random words from the model
random_words = np.random.choice(list(w2v_CBOW.wv.vocab.keys()), size=20,
replace=False) plot_word_embeddings(w2v_CBOW, random_words)
```

Implementing Word2Vec *(cont'd)*

3. Word Embeddings DataFrame

```
vector = pd.DataFrame(w2v_SG.wv.vectors)
vector.columns = list(w2v_SG.wv.vocab.keys())
vector.index = list(w2v_SG.wv.vocab.keys())
vector.head()
```

	Saudi	Vision	2030	is	a	strategic	framework	aimed	at	reducing	...
Saudi	-0.010134	-0.005619	-0.000848	-0.005767	-0.005605	-0.011439	-0.006764	0.005265	-0.008947	0.003225	...
Vision	-0.009512	0.002805	-0.002119	-0.006504	-0.008407	-0.001648	-0.000456	-0.004755	-0.002717	-0.002775	...
2030	0.011752	0.008253	-0.003374	0.006506	-0.007851	-0.010551	-0.011443	0.012018	-0.002569	-0.009787	...
is	-0.004154	-0.000653	-0.012131	-0.001399	-0.001966	0.004273	0.001662	0.003115	-0.011905	-0.006971	...
a	0.000704	0.007078	-0.012078	-0.00483	0.000472	0.002312	-0.010609	0.005597	0.001601	-0.011495	...

Word2Vec: **Drawbacks**

1. **Challenges with Phrase Representations:**
   - Combining word vector representations to obtain phrases like "hot potato" or "Boston Globe" is problematic.
   - Longer phrases and sentences pose even greater complexity.
2. **Limitations in Window Sizes:**
   - Smaller window sizes can lead to similar embeddings for contrasting words (e.g., "good" and "bad").
   - This similarity is undesirable, particularly in tasks like sentiment analysis where differentiation is crucial.
3. **Task Dependency:**
   - Word embeddings' effectiveness is dependent on the specific application.
   - Re-training embeddings for every new task is computationally expensive.

4. **Neglecting Polysemy and Biases:**
   - Word2Vec models may not adequately account for polysemy (multiple meanings of a word) and other biases present in the training data.
   - This can lead to skewed representations and biased outputs in downstream applications.

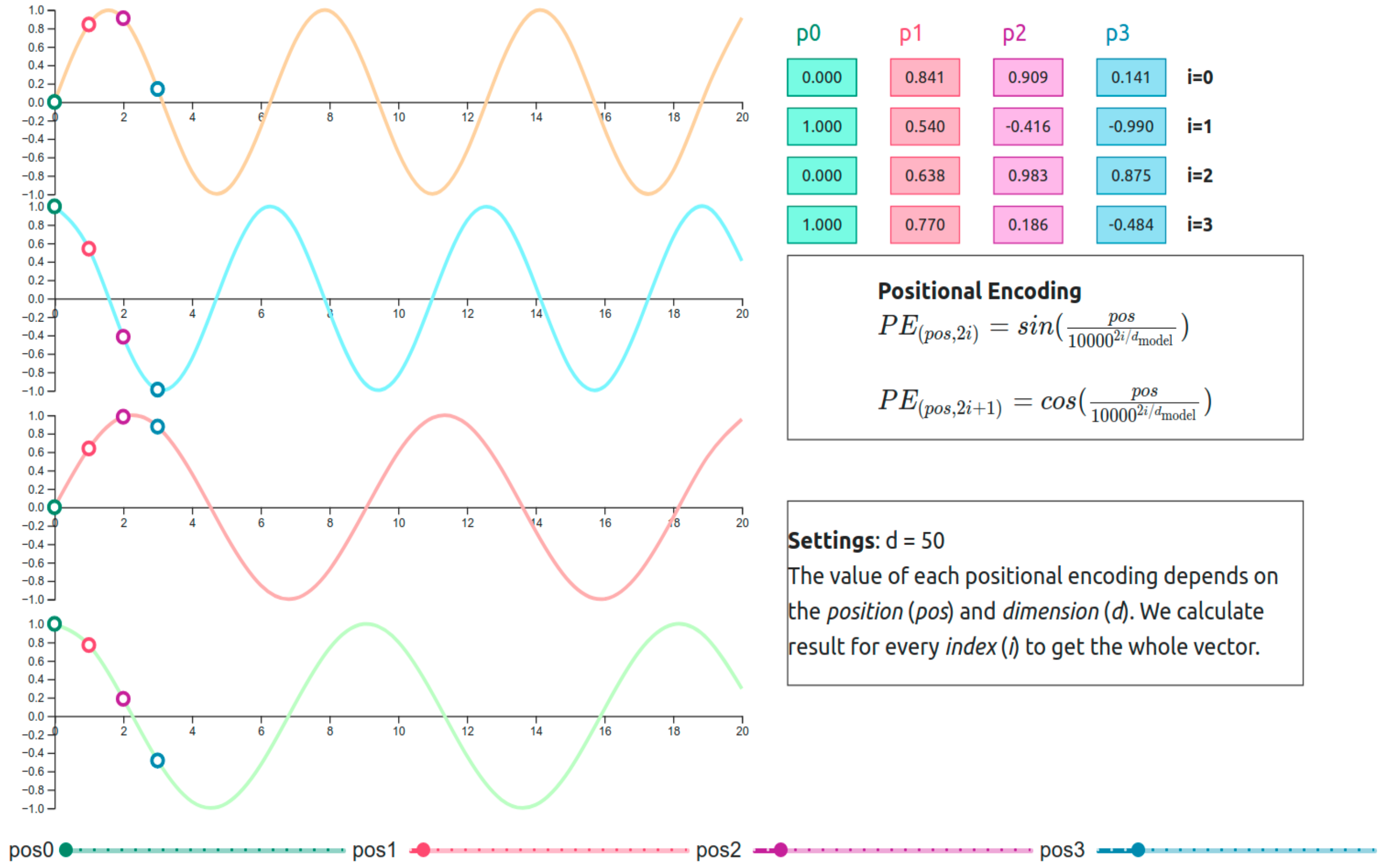
Embedding Tutorial

```
Embedding.ipynb
Embedding_Exercise.ipynb
```

Positional Encoding

Positional Encoding

Positional encoding is used in Transformer architectures to inject information about the position of words in a sequence (sentence) into the model.



p0	p1	p2	p3	
0.000	0.841	0.909	0.141	i=0
1.000	0.540	-0.416	-0.990	i=1
0.000	0.638	0.983	0.875	i=2
1.000	0.770	0.186	-0.484	i=3

Positional Encoding

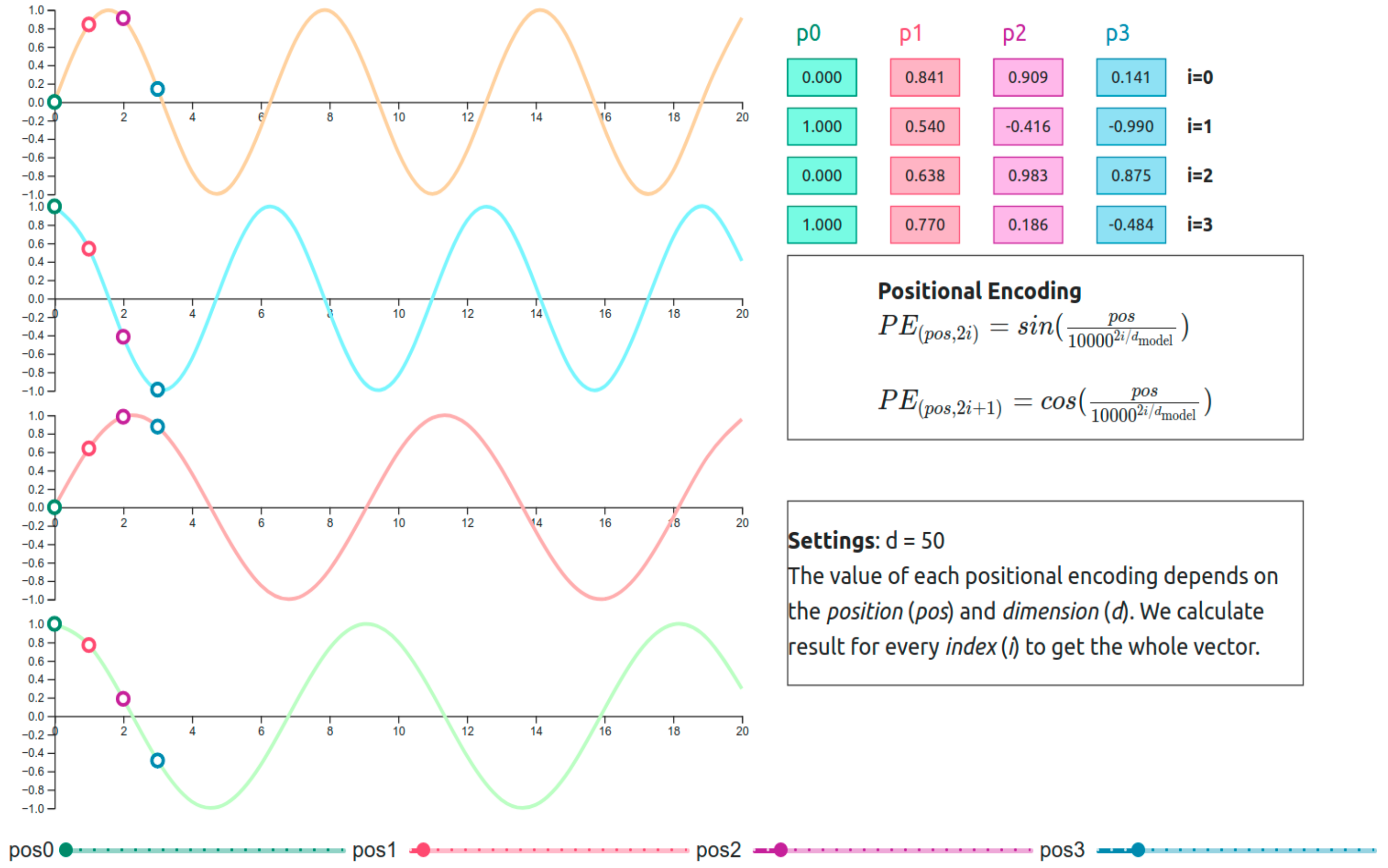
$$PE_{(pos,2i)} = sin(\frac{pos}{10000^{2i/d_{\text{model}}}})$$

$$PE_{(pos,2i+1)} = cos(\frac{pos}{10000^{2i/d_{\text{model}}}})$$

Settings: d = 50
The value of each positional encoding depends on the *position* (*pos*) and *dimension* (*d*). We calculate result for every *index* (*i*) to get the whole vector.

Positional Encoding

It helps the model understand the order of words in a sequence, which is important for tasks like machine translation and text generation.



p0	p1	p2	p3	
0.000	0.841	0.909	0.141	i=0
1.000	0.540	-0.416	-0.990	i=1
0.000	0.638	0.983	0.875	i=2
1.000	0.770	0.186	-0.484	i=3

Positional Encoding

$$PE_{(pos,2i)} = sin(\frac{pos}{10000^{2i/d_{\text{model}}}})$$

$$PE_{(pos,2i+1)} = cos(\frac{pos}{10000^{2i/d_{\text{model}}}})$$

Settings: d = 50
The value of each positional encoding depends on the *position* (*pos*) and *dimension* (*d*). We calculate result for every *index* (*i*) to get the whole vector.

Positional Encoding

Positional encoding is important for Transformer models as these models do not inherently understand the order of words in a sequence.

Formula

$$PE_{(pos,2i)} = sin(\frac{pos}{10000^{2i/d_{\text{model}}}})$$

$$PE_{(pos,2i+1)} = cos(\frac{pos}{10000^{2i/d_{\text{model}}}})$$

Positional Encoding

The formula for positional encoding uses *sine* and *cosine* functions to encode the position of each word based on its position in the sequence and the dimension of the model.

Formula

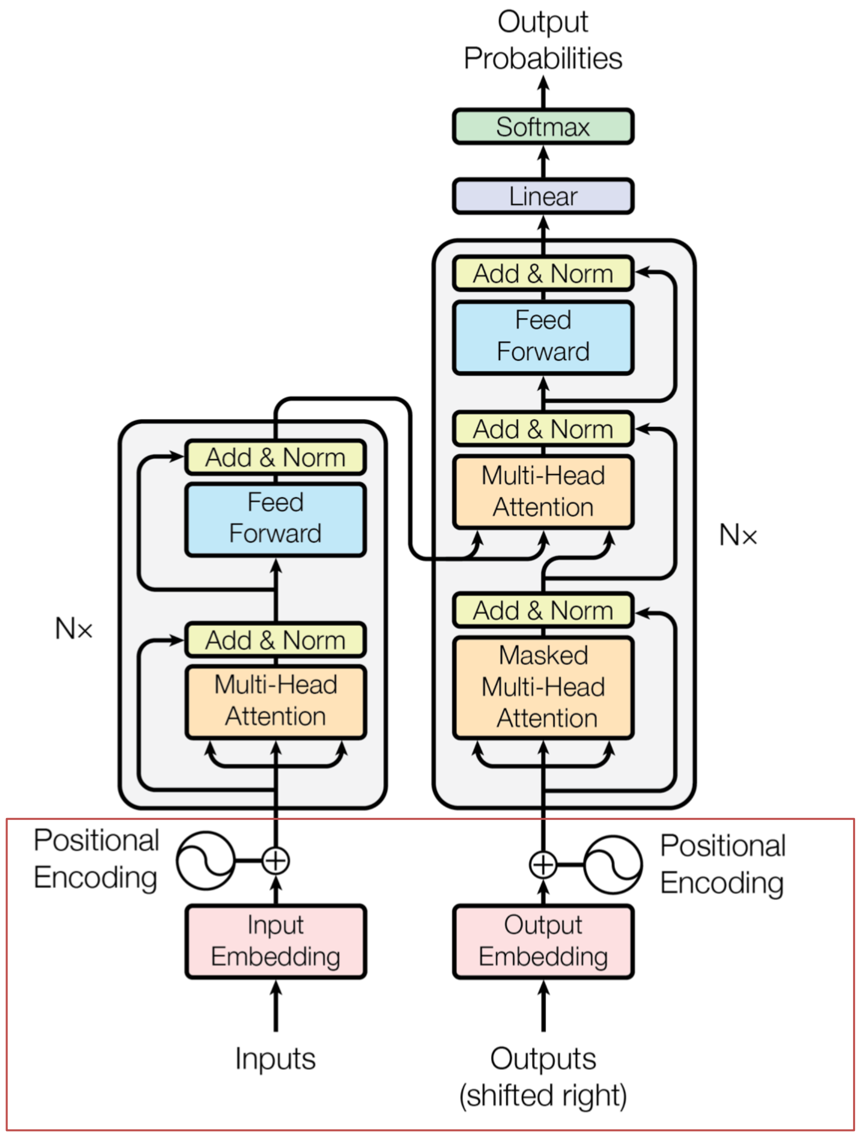
$$PE_{(pos,2i)} = sin(\frac{pos}{10000^{2i/d_{\text{model}}}})$$

$$PE_{(pos,2i+1)} = cos(\frac{pos}{10000^{2i/d_{\text{model}}}})$$

Tokenization & Word Embedding in Transformers

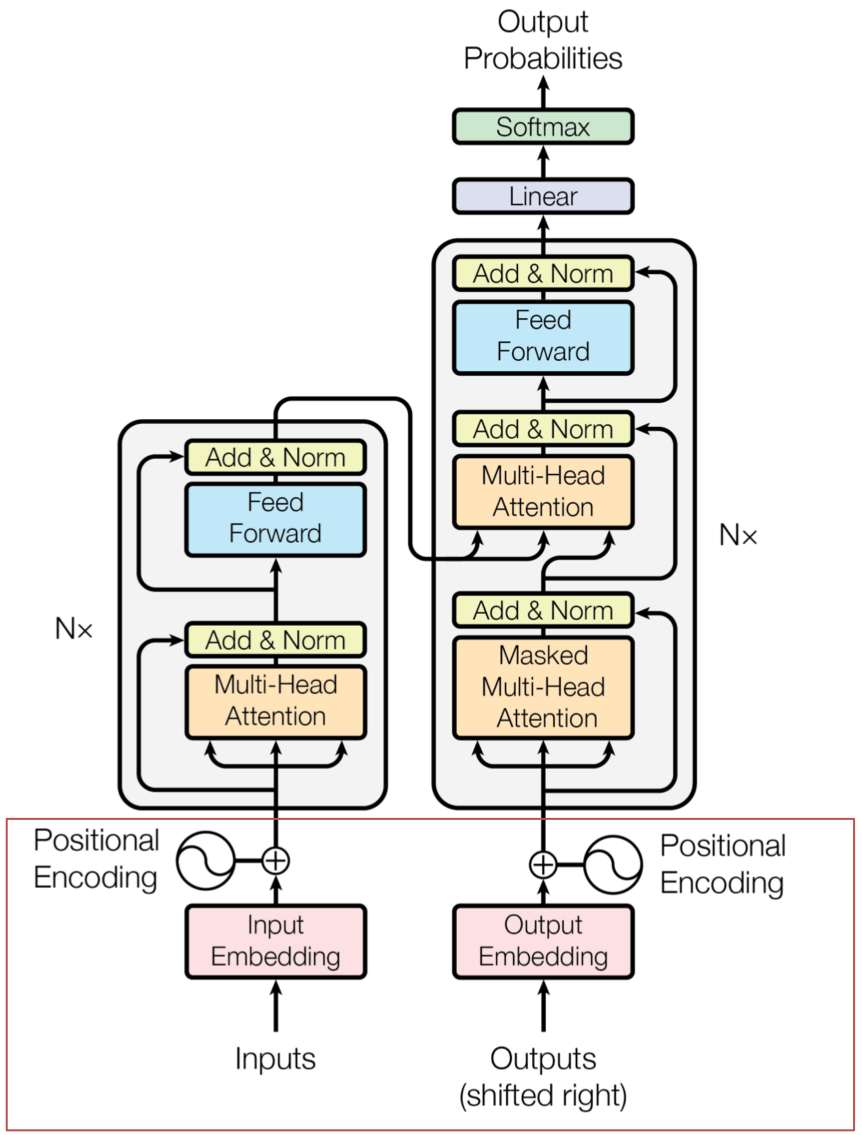
Tokenization & Word Embedding in Transformers

In the original transformer, each token was mapped to 512-dimensional embedding.



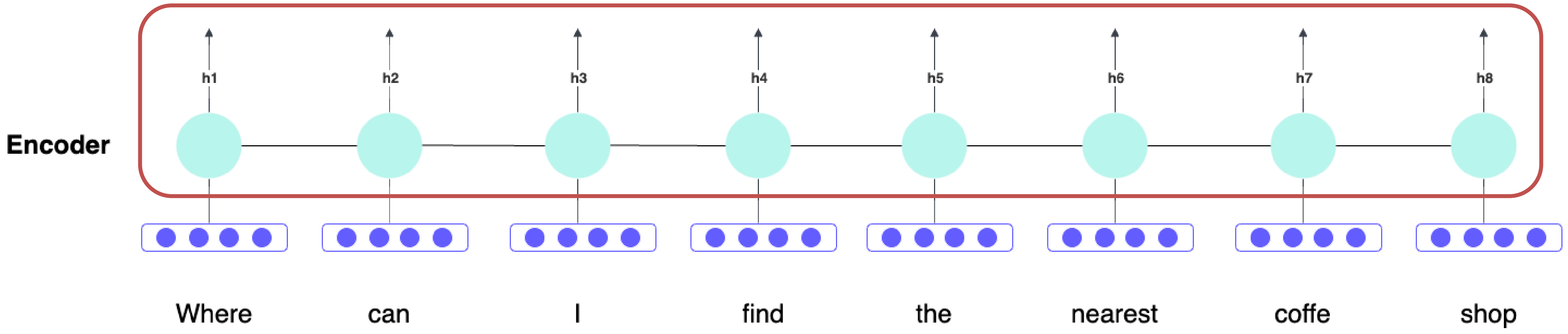
Tokenization & Word Embedding in Transformers

Unlike Recurrent-Based Attention the transformer attention is parallelized, information loss is not a problem anymore!



Tokenization & Word Embedding in Transformers

This is achieved by removing the recurrent layer to eliminate the sequential nature and replacing it with an alternative solution.



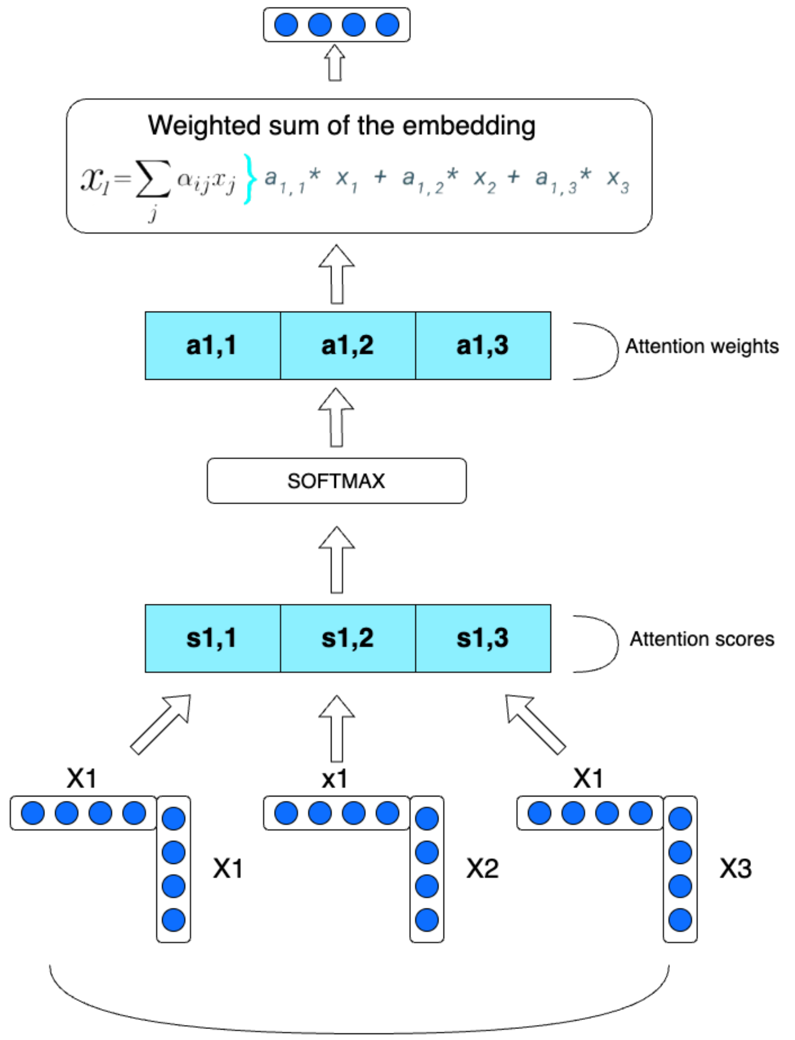
Tokenization & Word Embedding in Transformers

The alternative solution is basing each encoder output on all the encoder inputs, so each encoder output will take in count all the encoder inputs.

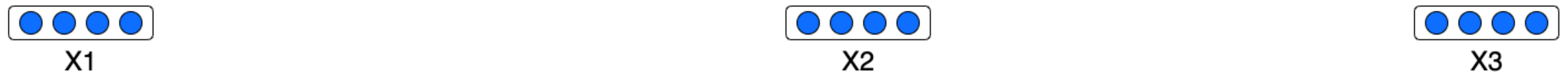
This is done via making the encoder applies attention on itself which is a mechanism called “**Self Attention**”.

Multi-Head Self-Attention

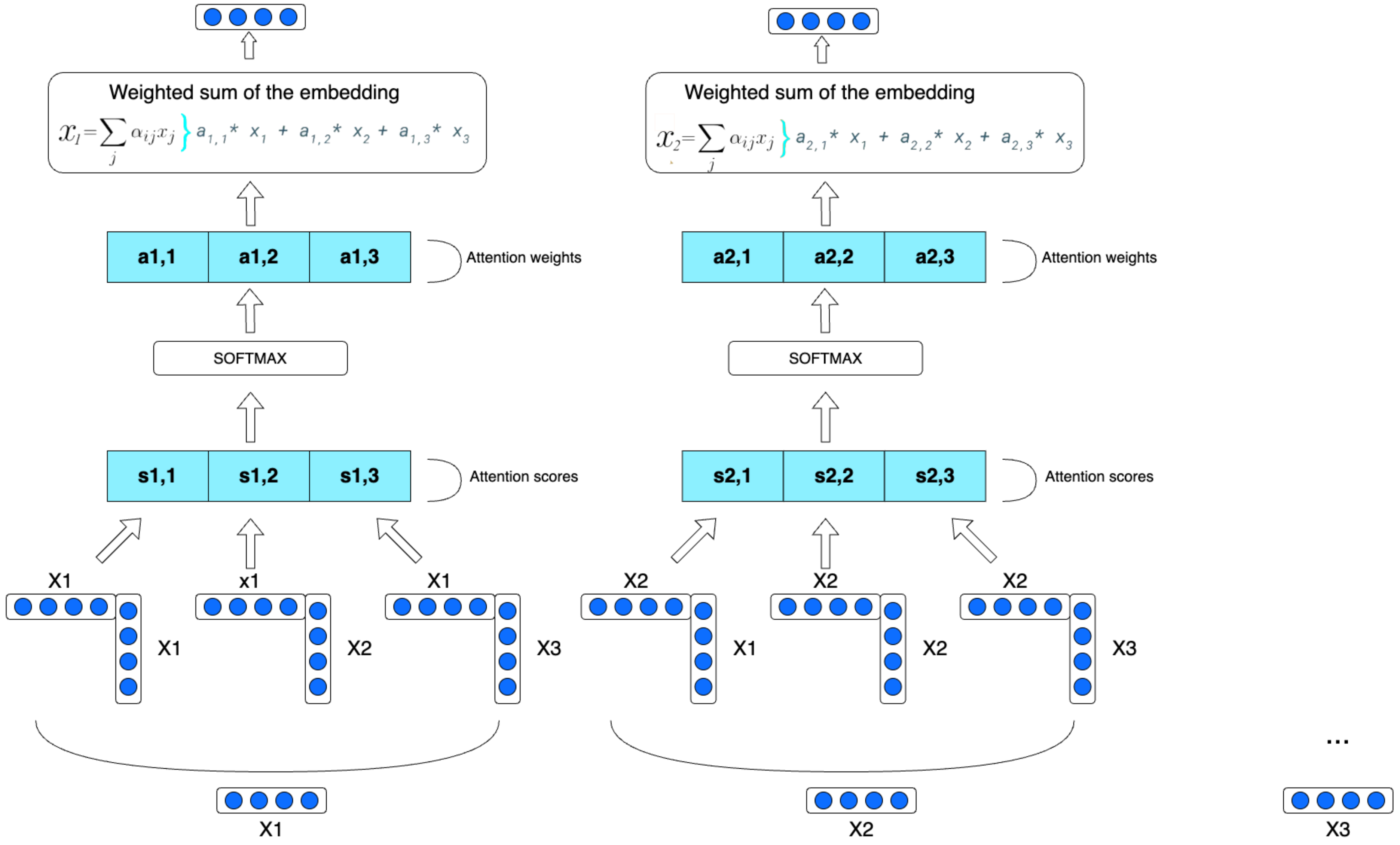
Simple Self Attention



Weighted sum of the embedding
$x_1 = \sum_j \alpha_{ij} x_j$
$a_{1,1} * x_1 + a_{1,2} * x_2 + a_{1,3} * x_3$
a1,1
a1,2
a1,3
Attention weights
SOFTMAX
s1,1
s1,2
s1,3
Attention scores
X1
x1
X1
X1
X2
X3



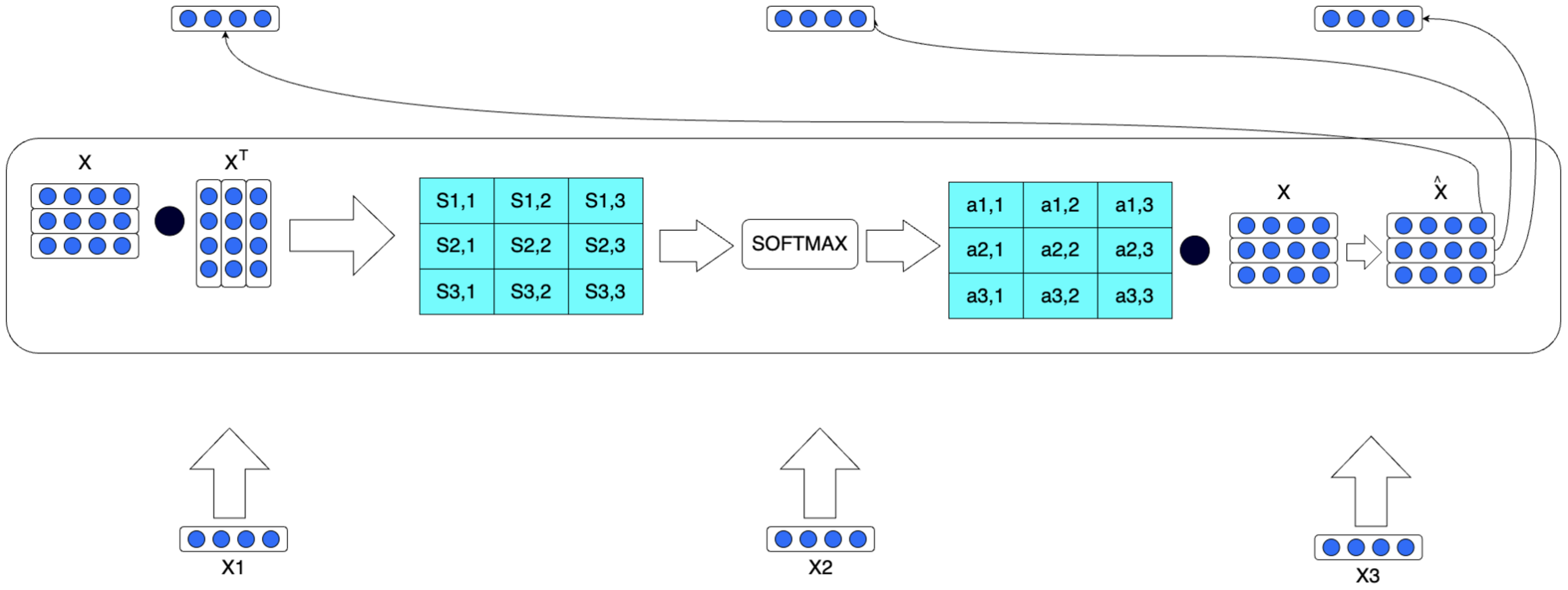
Simple Self Attention



Weighted sum of the embedding
$x_1 = \sum_j \alpha_{ij} x_j$ } $a_{1,1} * x_1 + a_{1,2} * x_2 + a_{1,3} * x_3$
a1,1
a1,2
a1,3
Attention weights
SOFTMAX
s1,1
s1,2
s1,3
Attention scores
X1
x1
X1
X1
X2
X3
X1
Weighted sum of the embedding
$x_2 = \sum_j \alpha_{ij} x_j$ } $a_{2,1} * x_1 + a_{2,2} * x_2 + a_{2,3} * x_3$
a2,1
a2,2
a2,3
Attention weights
SOFTMAX
s2,1
s2,2
s2,3
Attention scores
X2
X2
X2
X1
X2
X3
X2
...
X3

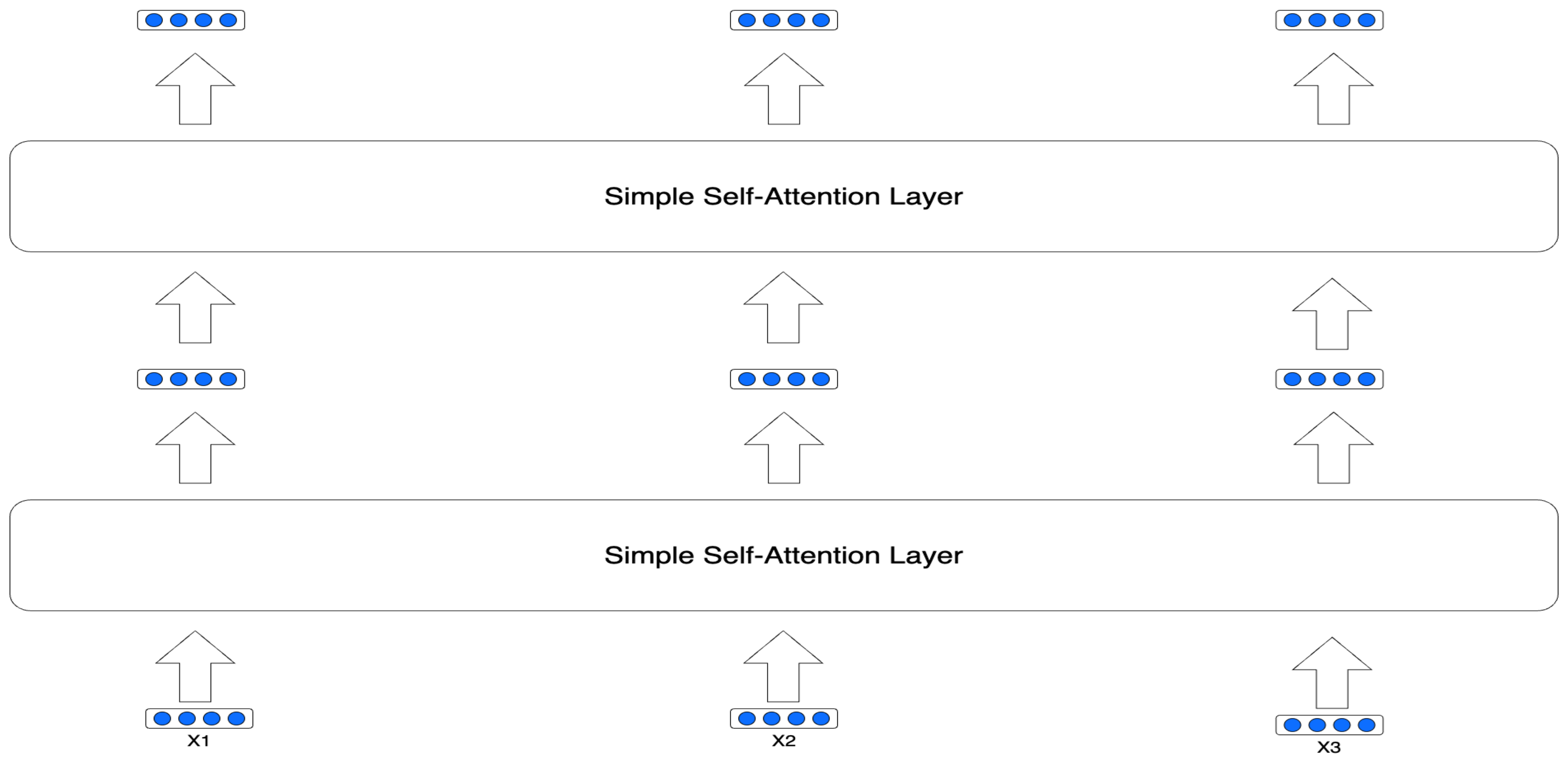
Simple Self Attention

We don't need to do this sequentially; we can achieve it all at once using matrices.



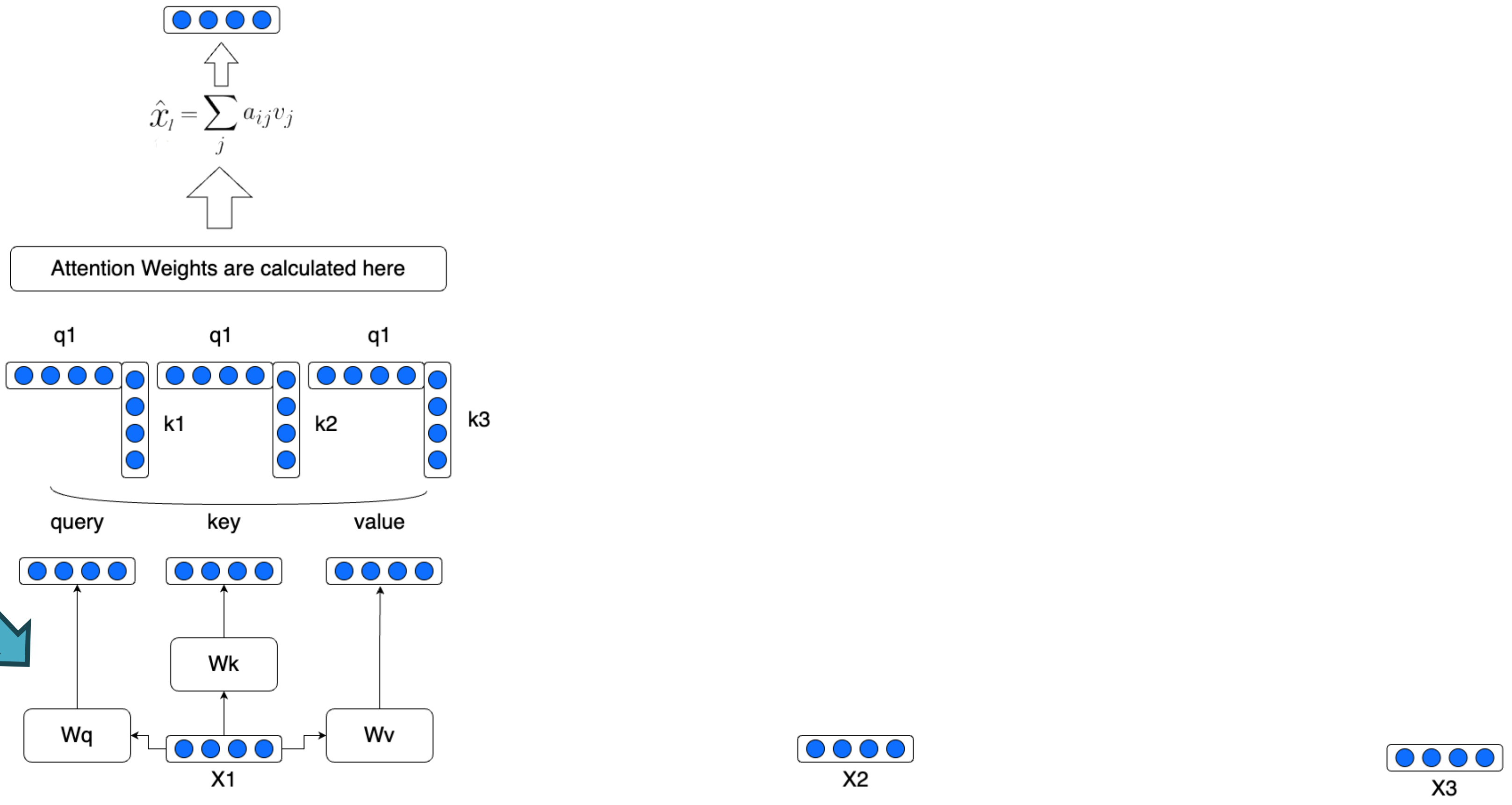
Simple Self Attention

Since the encoder output is based on the encoder inputs, we can stack them up.



Scaled Dot Product Self-Attention

Instead of multiplying the row embeddings to each others, we turn them into queries, keys, and values.



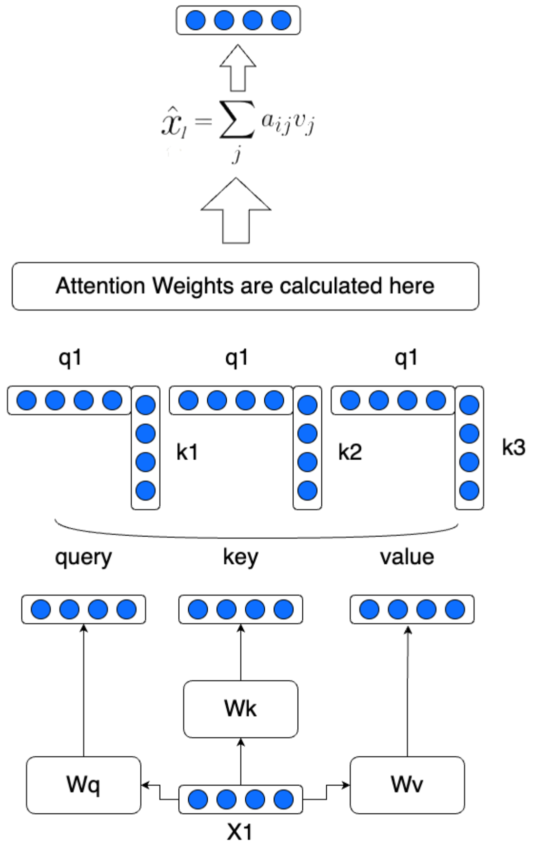
$\hat{x}_i = \sum_j a_{ij} v_j$
Attention Weights are calculated here
q1
q1
q1
k1
k2
k3
query
key
value
Wk
Wq
Wv
X1
X2
X3

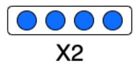
We multiply “x1” to query ‘Wq’ to generate query vector.

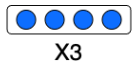
Same also for the key and the value vectors

Scaled Dot Product Self-Attention

Everything is done in the same way for the next encoder input (x2 and x3)

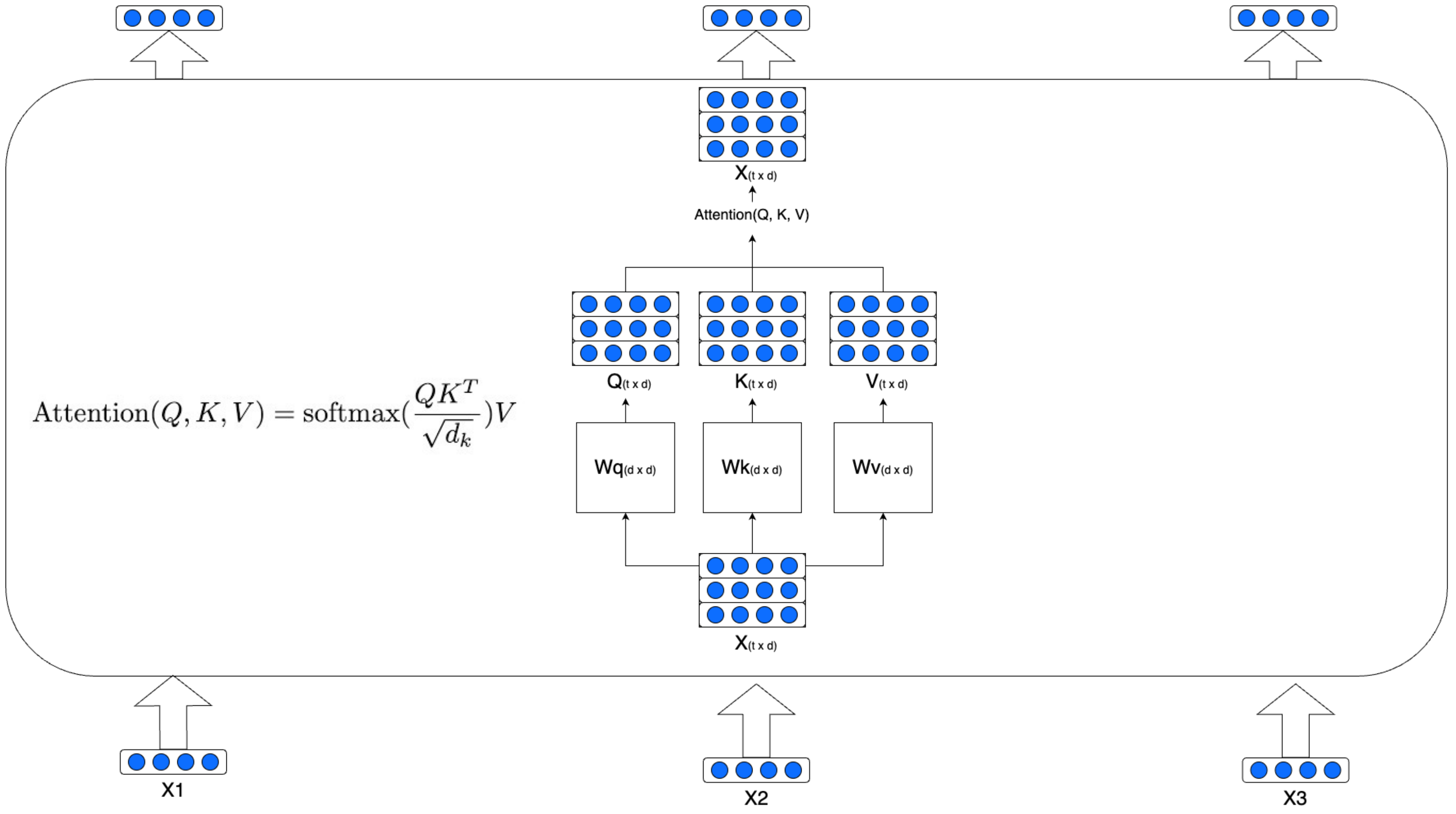

$\hat{x}_i = \sum_j a_{ij} v_j$
Attention Weights are calculated here
q1
q1
q1
k1
k2
k3
query
key
value
Wk
Wq
Wv
X1





Scaled Dot Product Self-Attention

We don't need to do this sequentially; we can achieve it all at once using matrices.

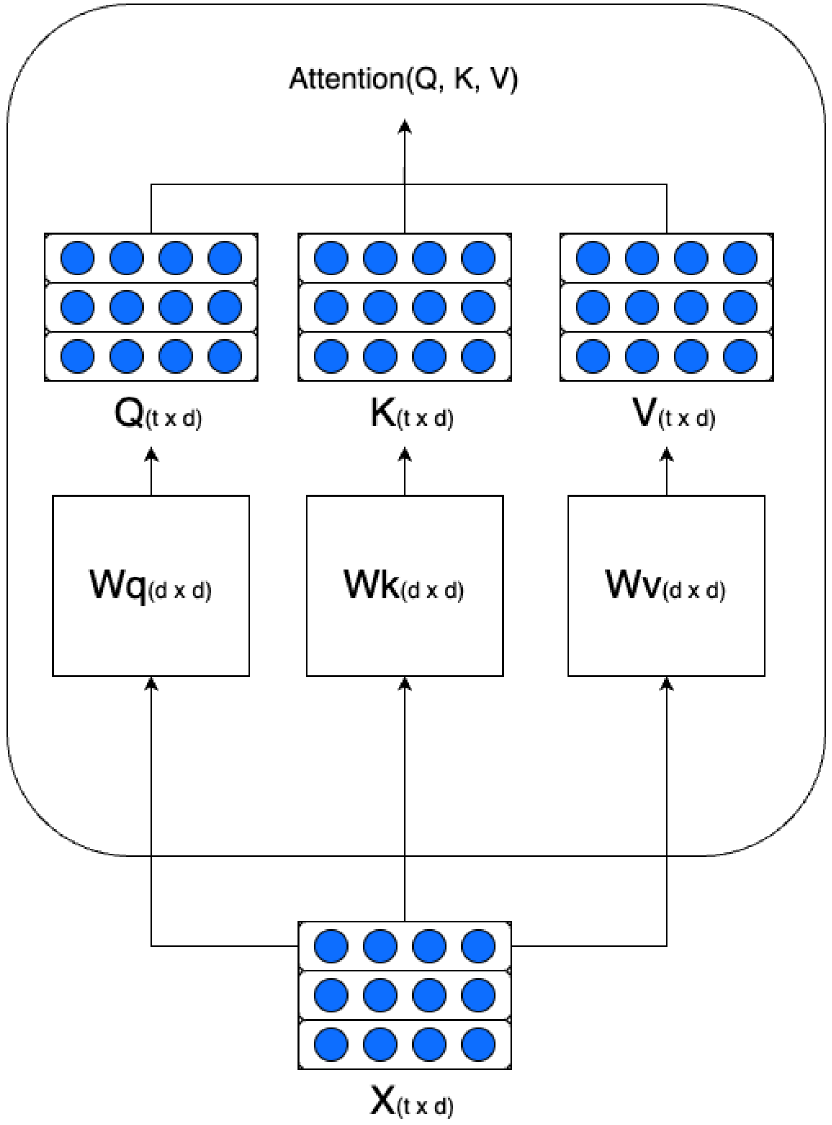


$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Attention Head (single)

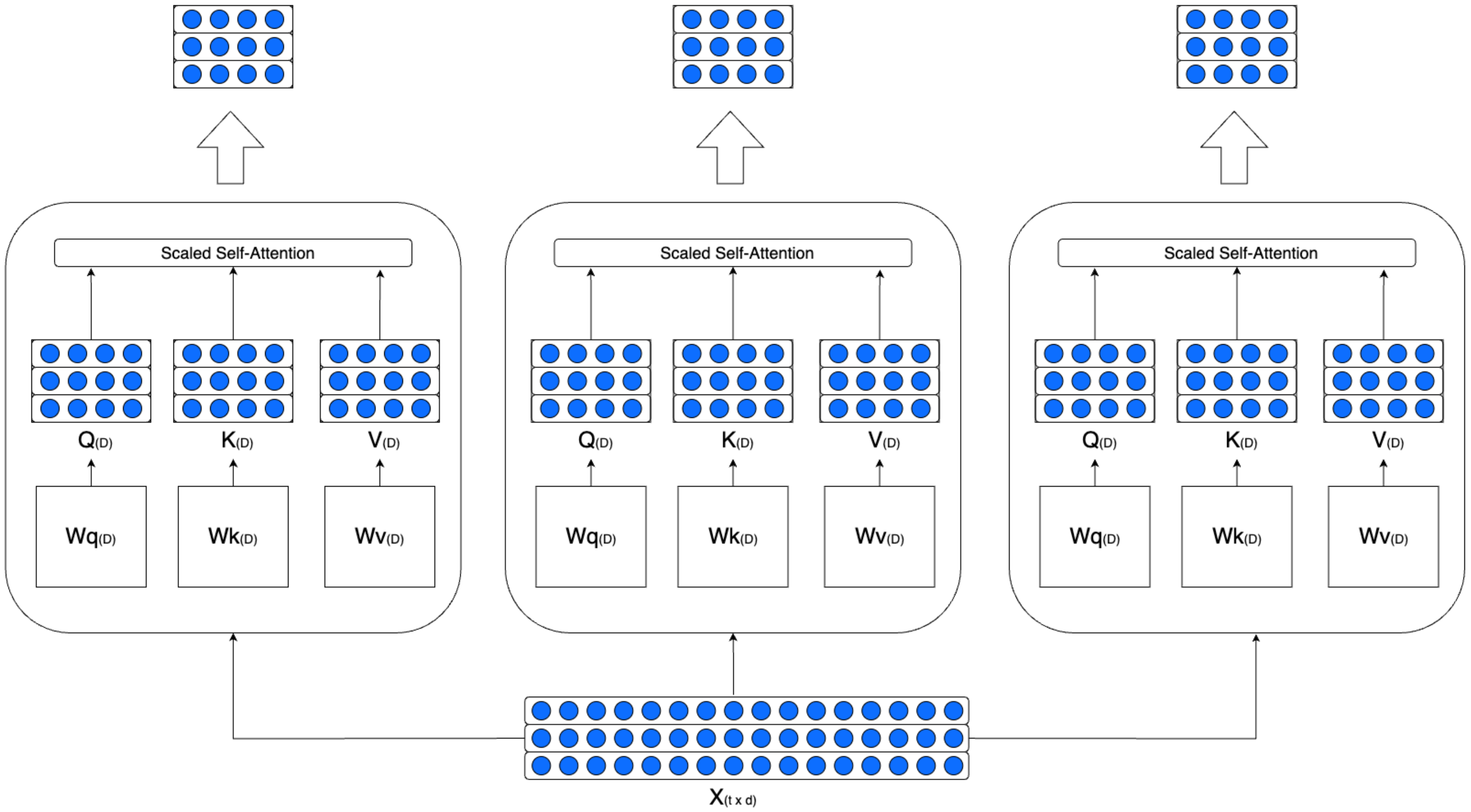
This process, where inputs are transformed into queries, keys, and values before being used in the attention calculation, can be considered as a head.

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



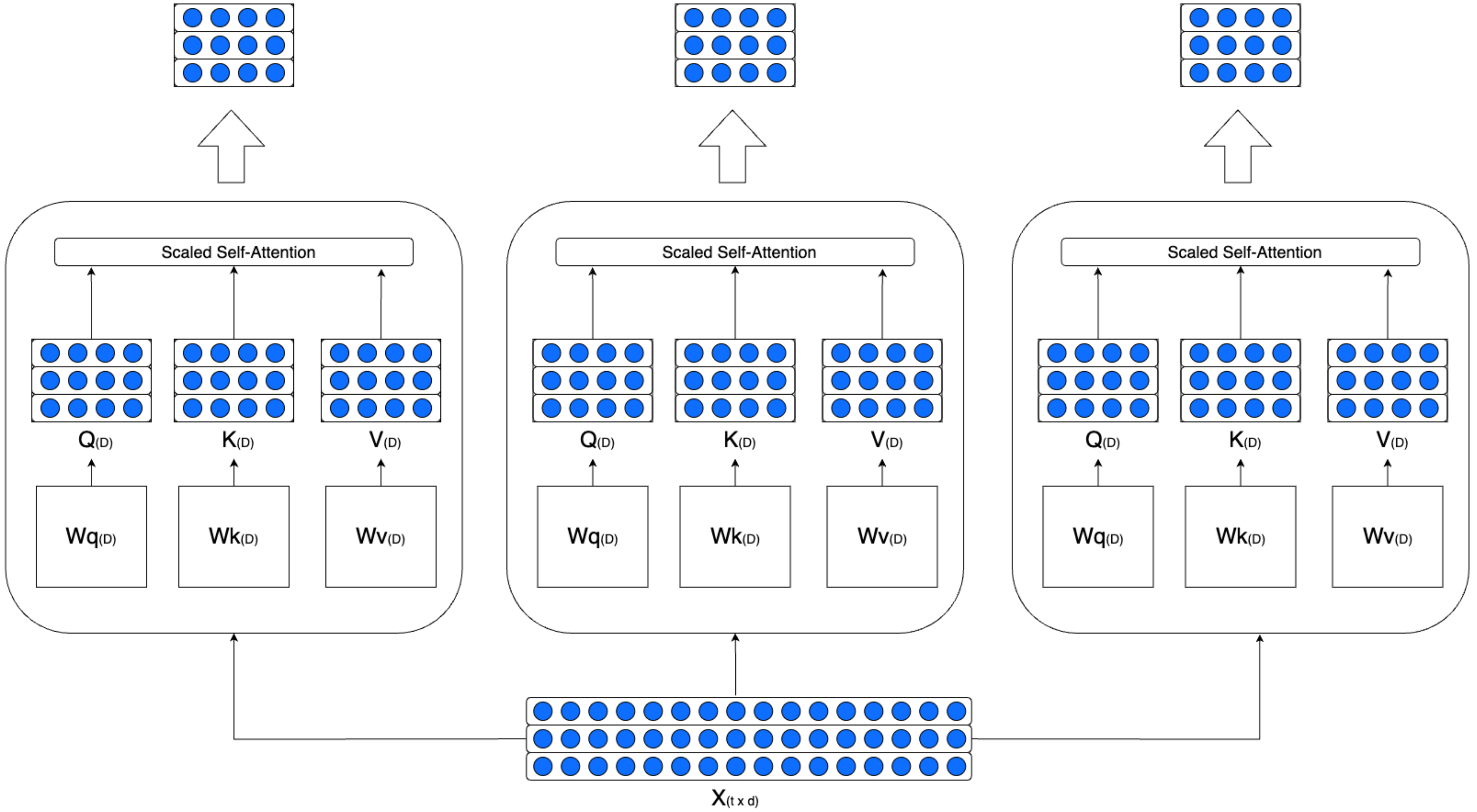
Multi-Head Self-Attention

In Multi-Head Self-Attention, each head has its own of *q*, *k*, and *v* weights.



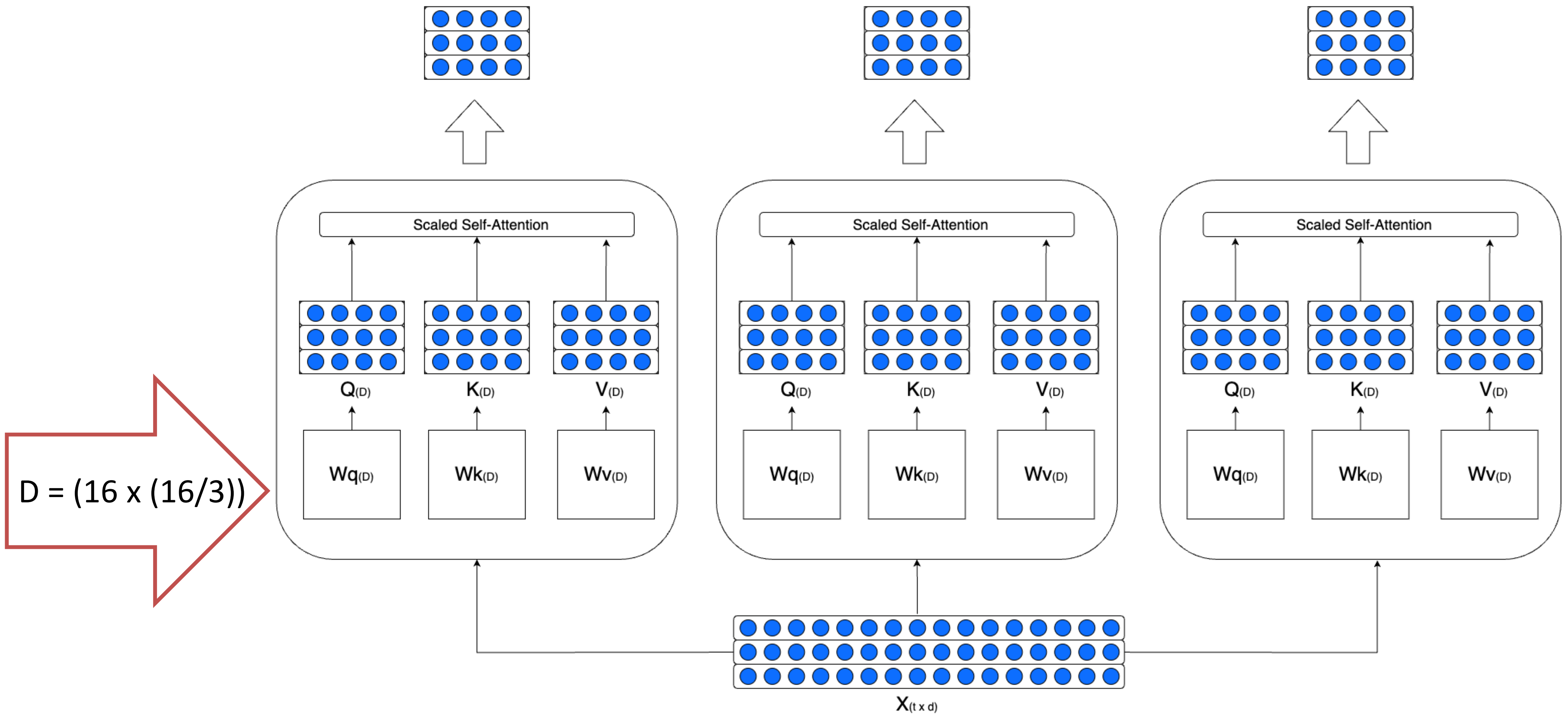
Multi-Head Self-Attention

The dimension of each Q, K, and V will be denoted by D which is (*D = d x d/h*) where *h* is the number of heads



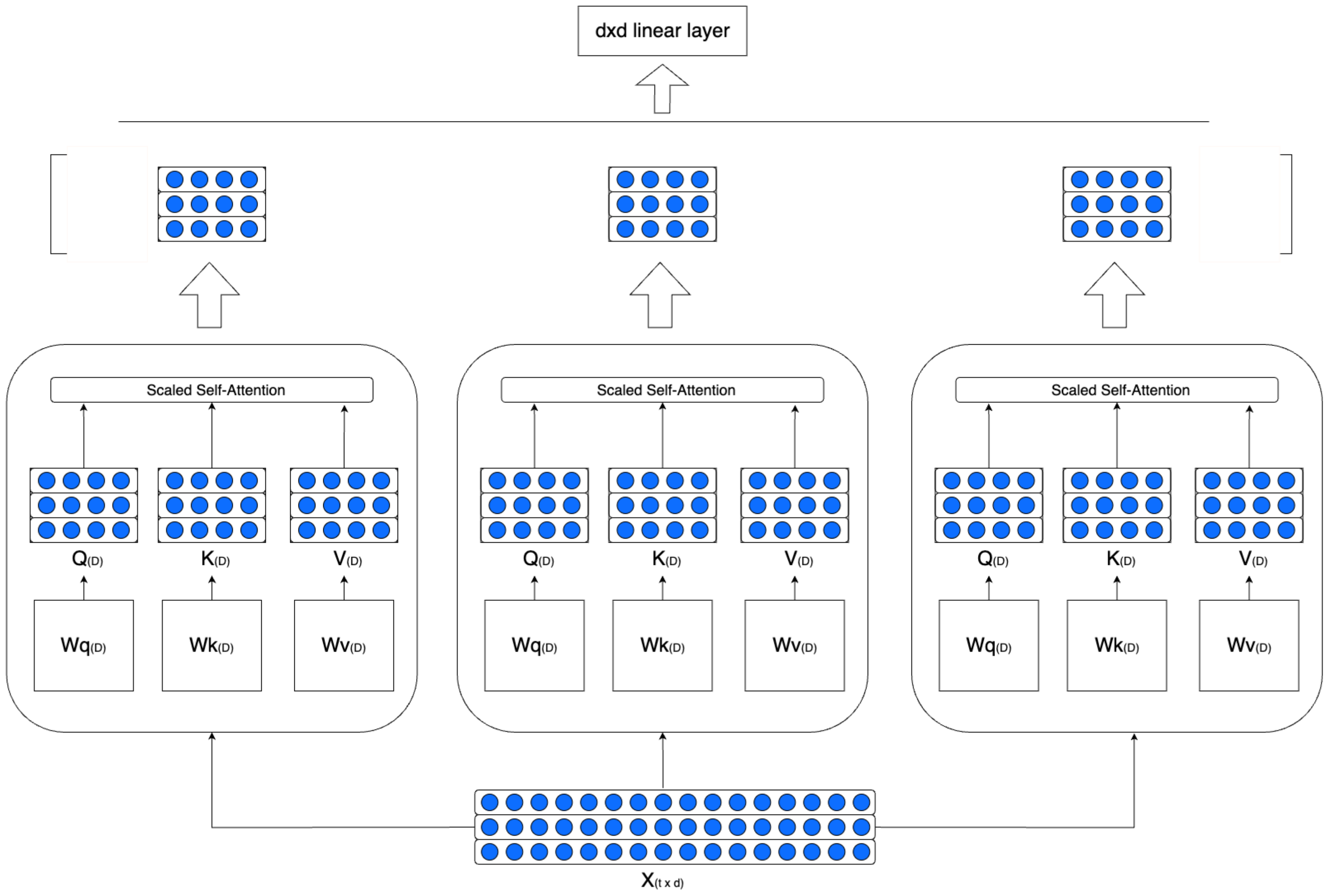
Multi-Head Self-Attention

The dimension of each Q, K, and V will be denoted by D which is (D = d x d/h)



Multi-Head Self-Attention

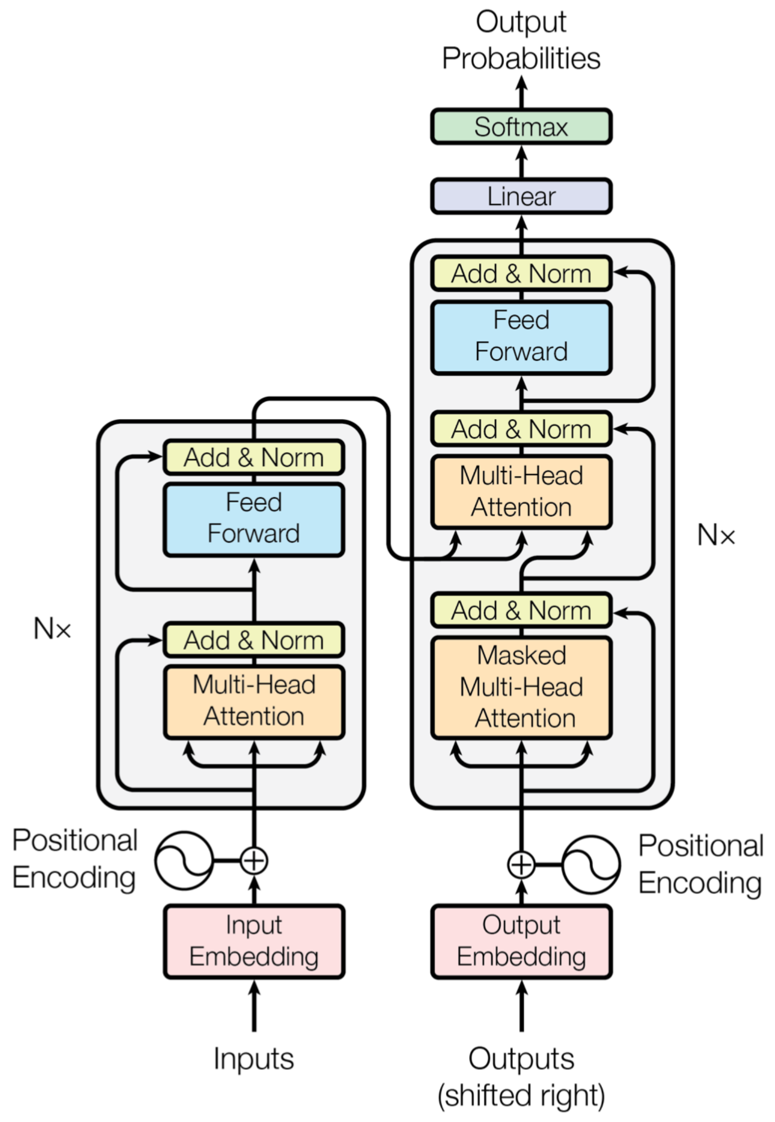
At the end, the 3 outputs are concatenated and then passed through dxd linear layer.



Add & Norm

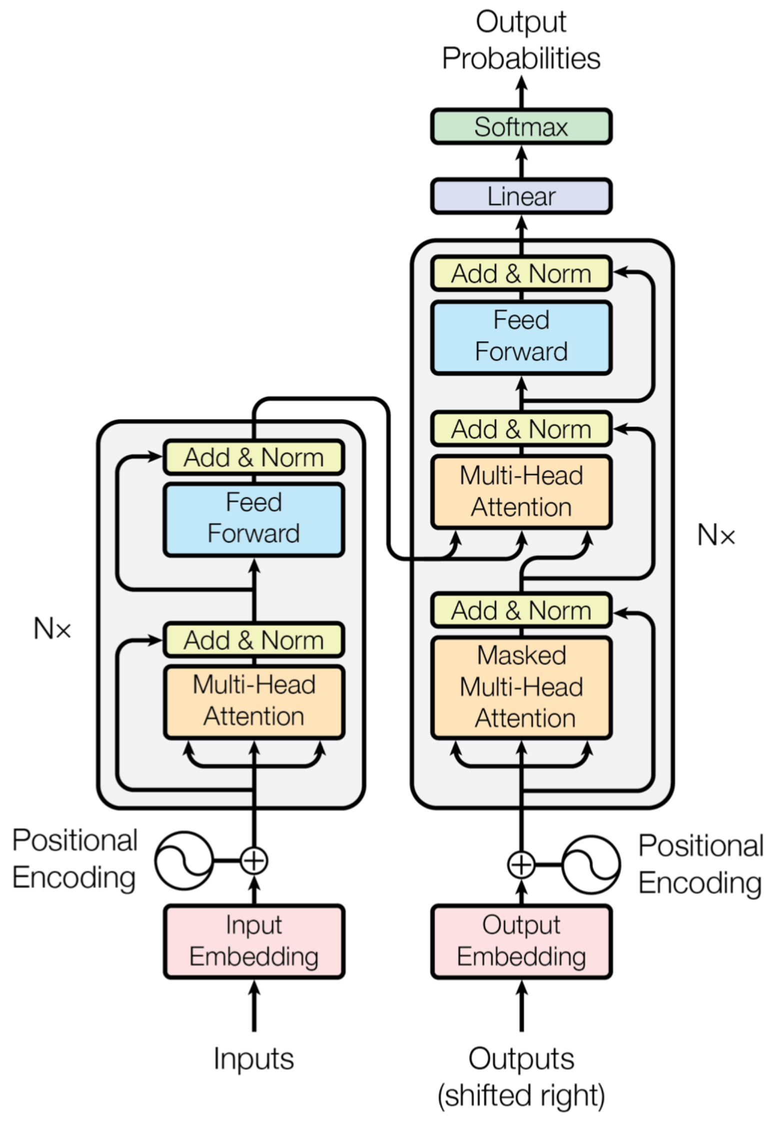
Add & Norm

In the architecture of Transformer, you will notice that there are multiple Add & Norm layers. This layer adds the Residual Connection and performs Normalization Layer.



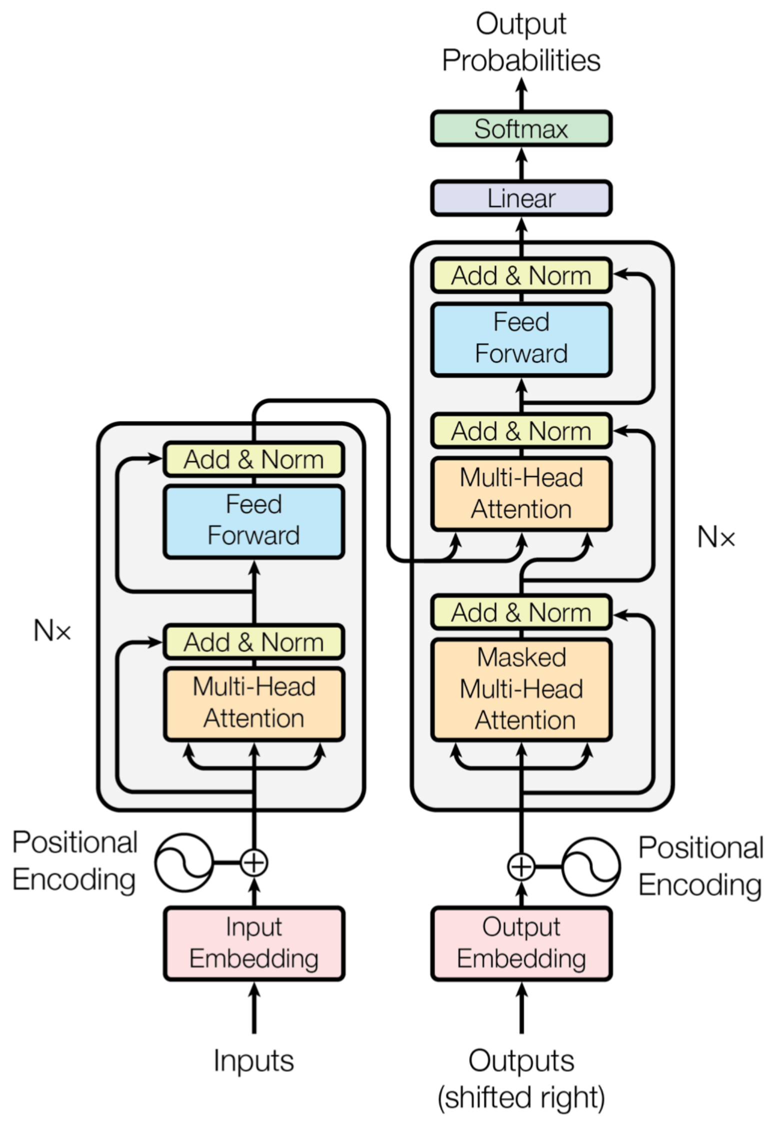
Add & Norm (The Residual Connection)

The Residual Connection creates a path for the gradient. This ensure that the vectors are updated instead of getting replaced by the Attention layer.



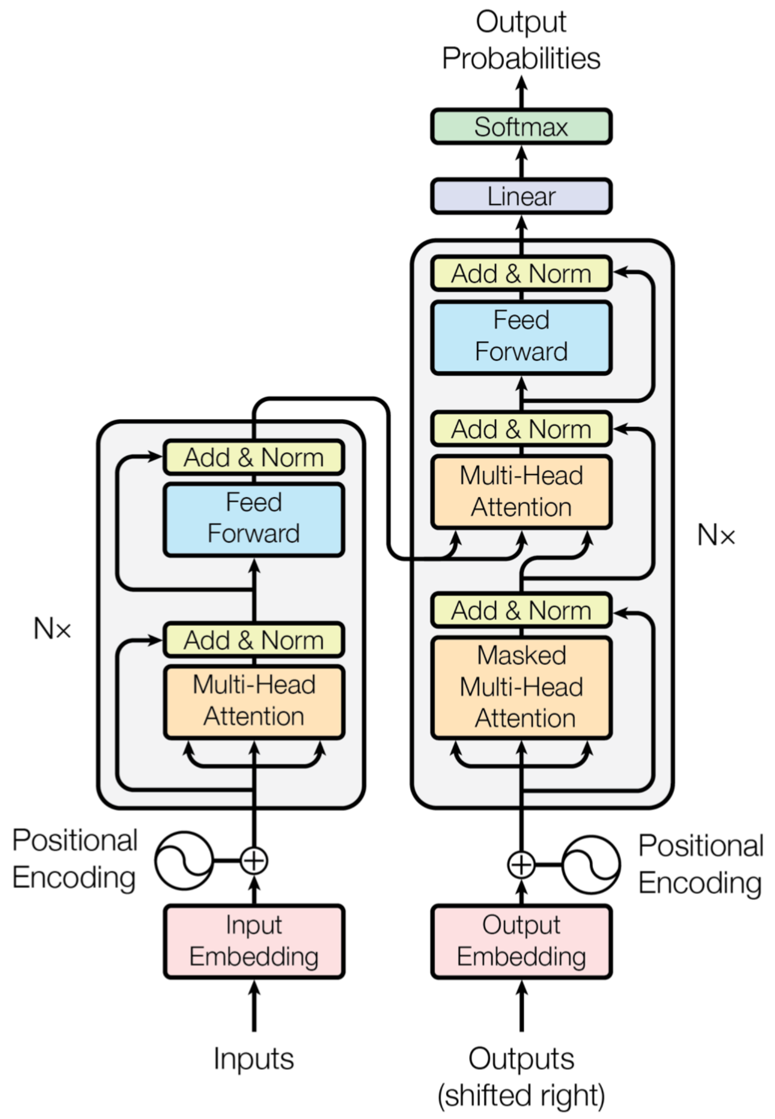
Add & Norm (Normalization Layer)

Normalization Layer simplifies the activation values of the neural network. Typically, these values can span a wide range of positive and negative numbers. Normalization rescales these values to a smaller range, centering them around 0 with a standard deviation of 1.



Add & Norm (Normalization Layer)

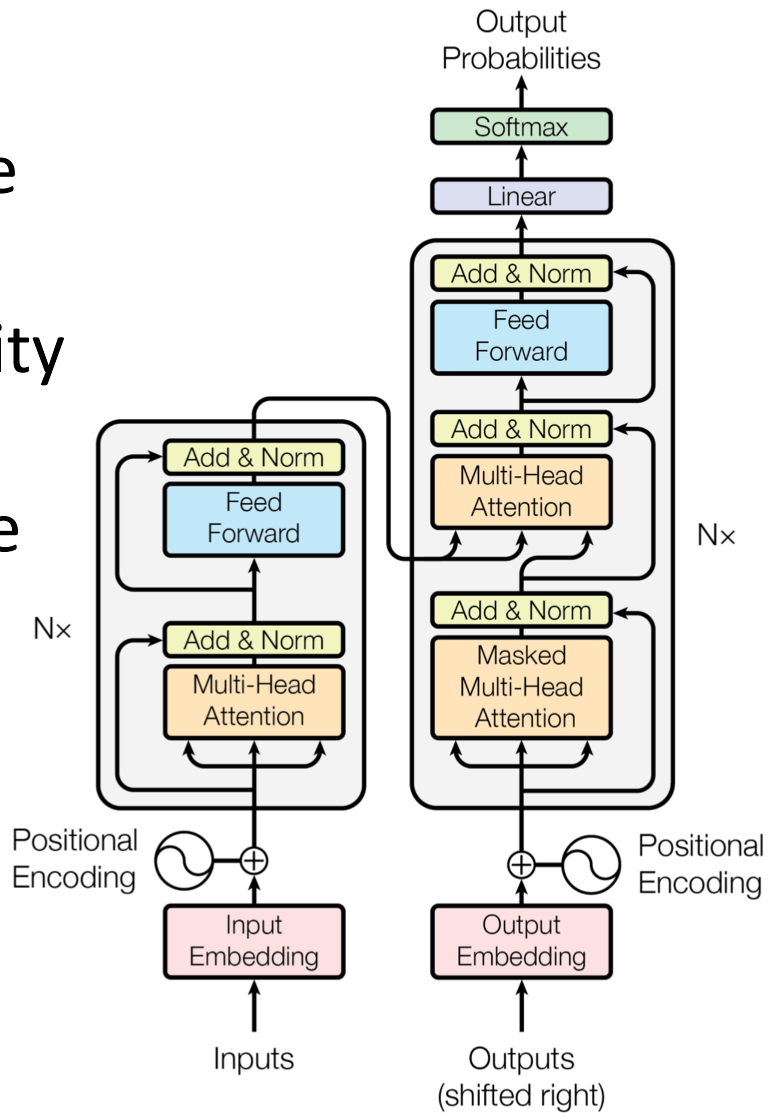
Normalization Layer maintains a reasonable scale for the output, which would enhance the model’s stability and performance .



Add & Norm (Normalization Layer)

- μ is the mean of the last D dimensions. σ^2 is the variance of the last D dimensions.
- ε is a small constant to ensure numerical stability when σ^2 is very small.
- γ and β are learnable parameters that allow the network to adjust the normalized values.

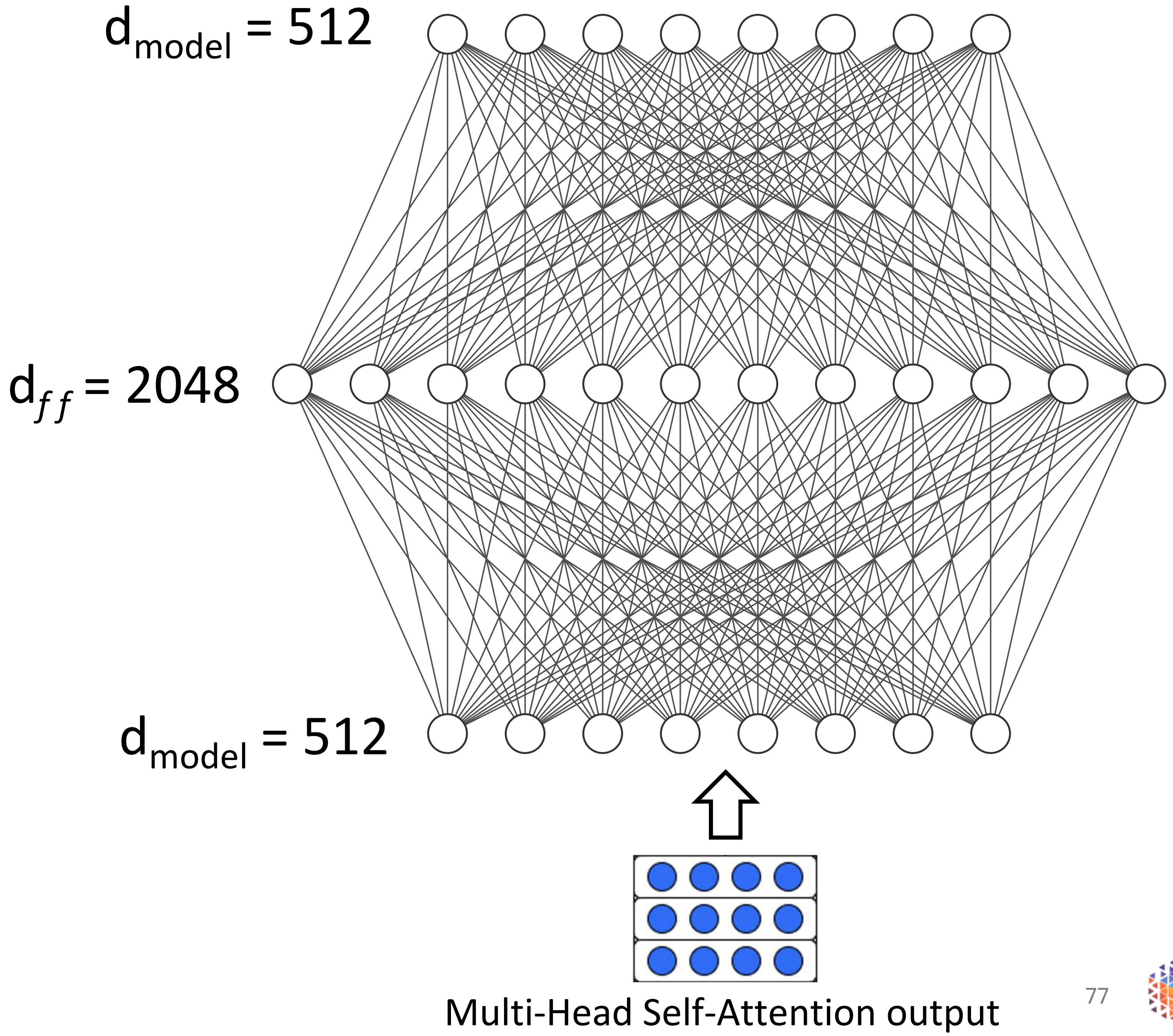
$$y = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} \odot \gamma + \beta$$



Feed Forward

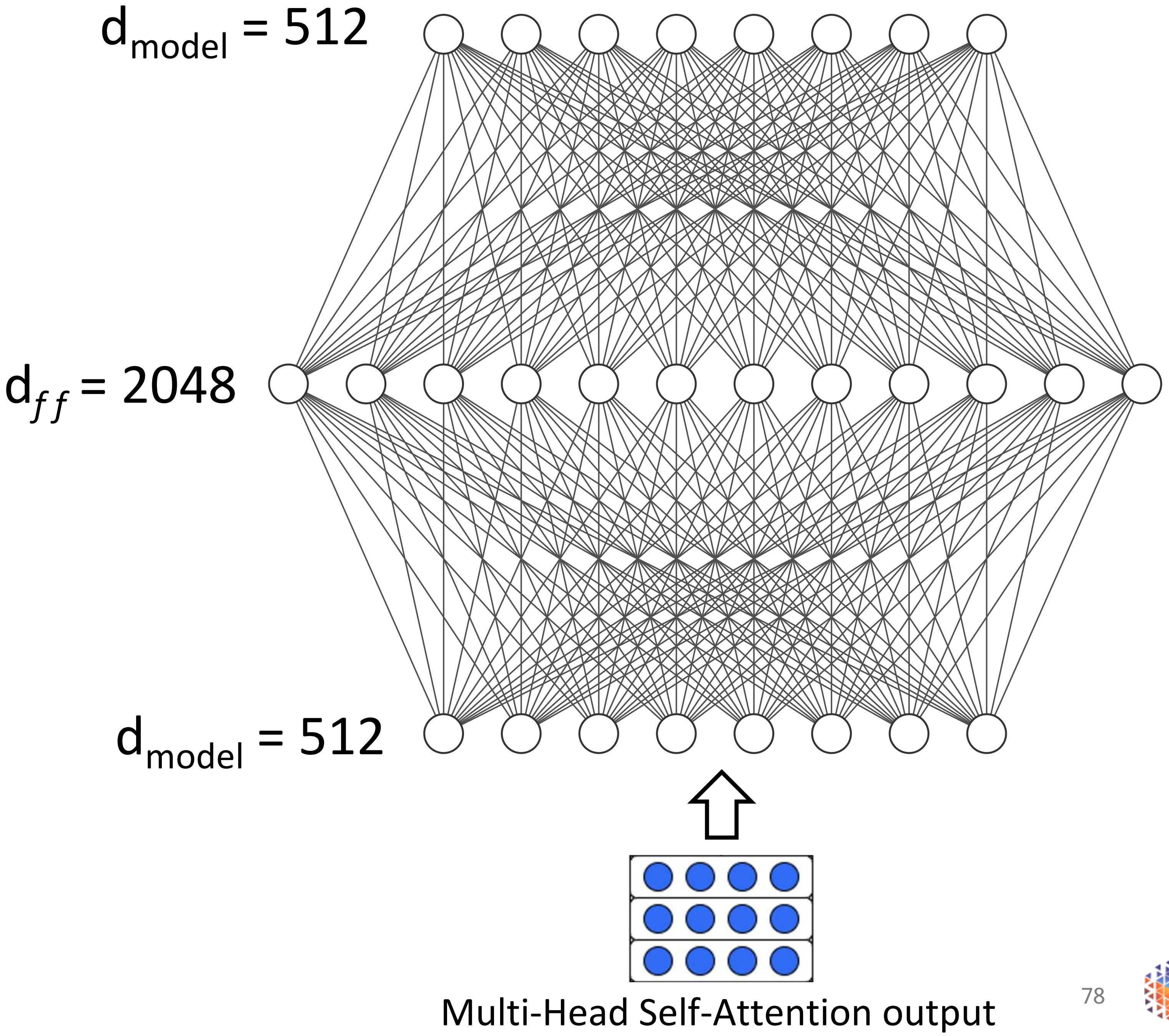
Feed Forward

Each layer in the original Transformer contains a fully connected feed-forward network.



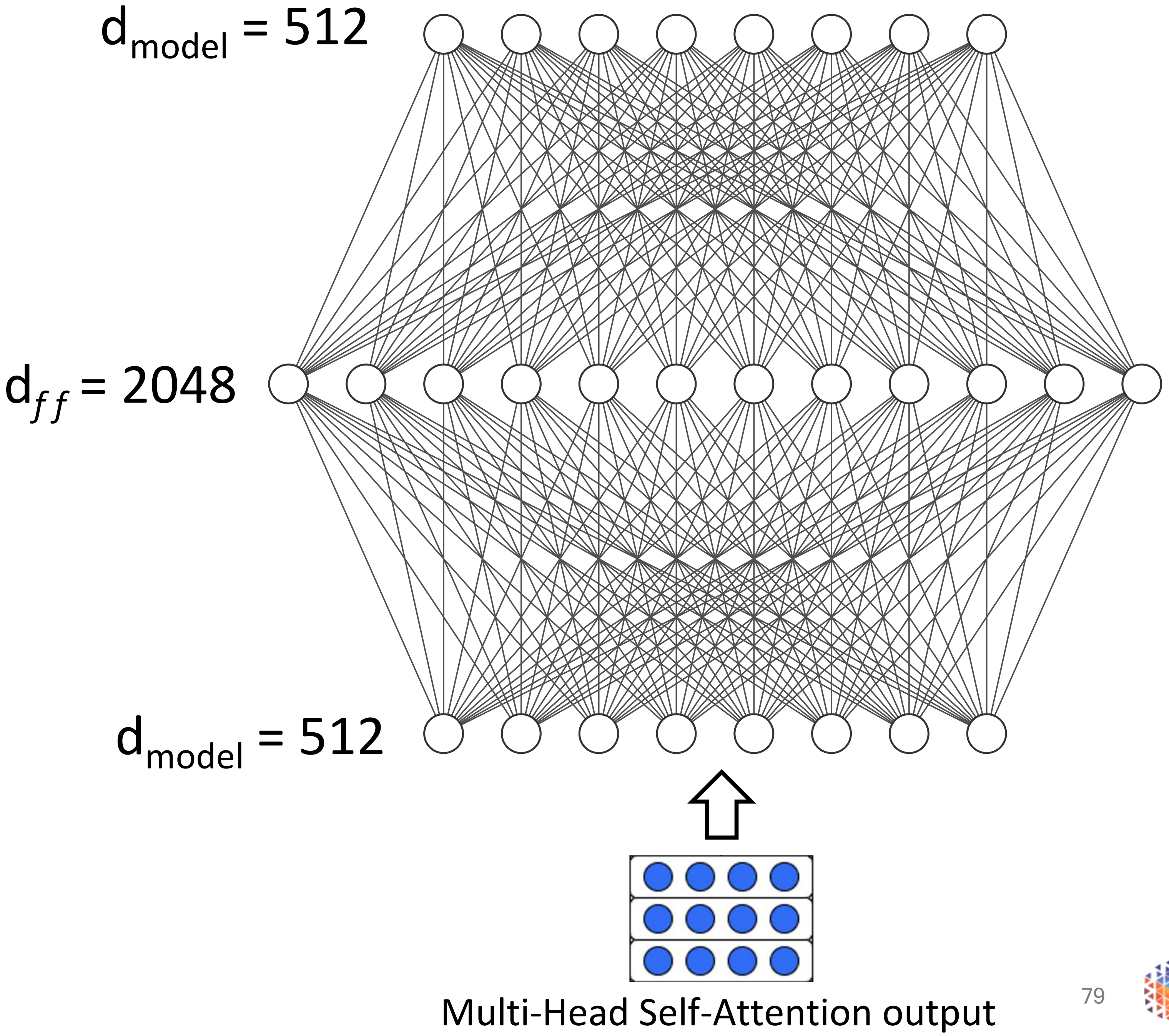
Feed Forward

This consists of two linear transformations with a ReLU activation function in between.



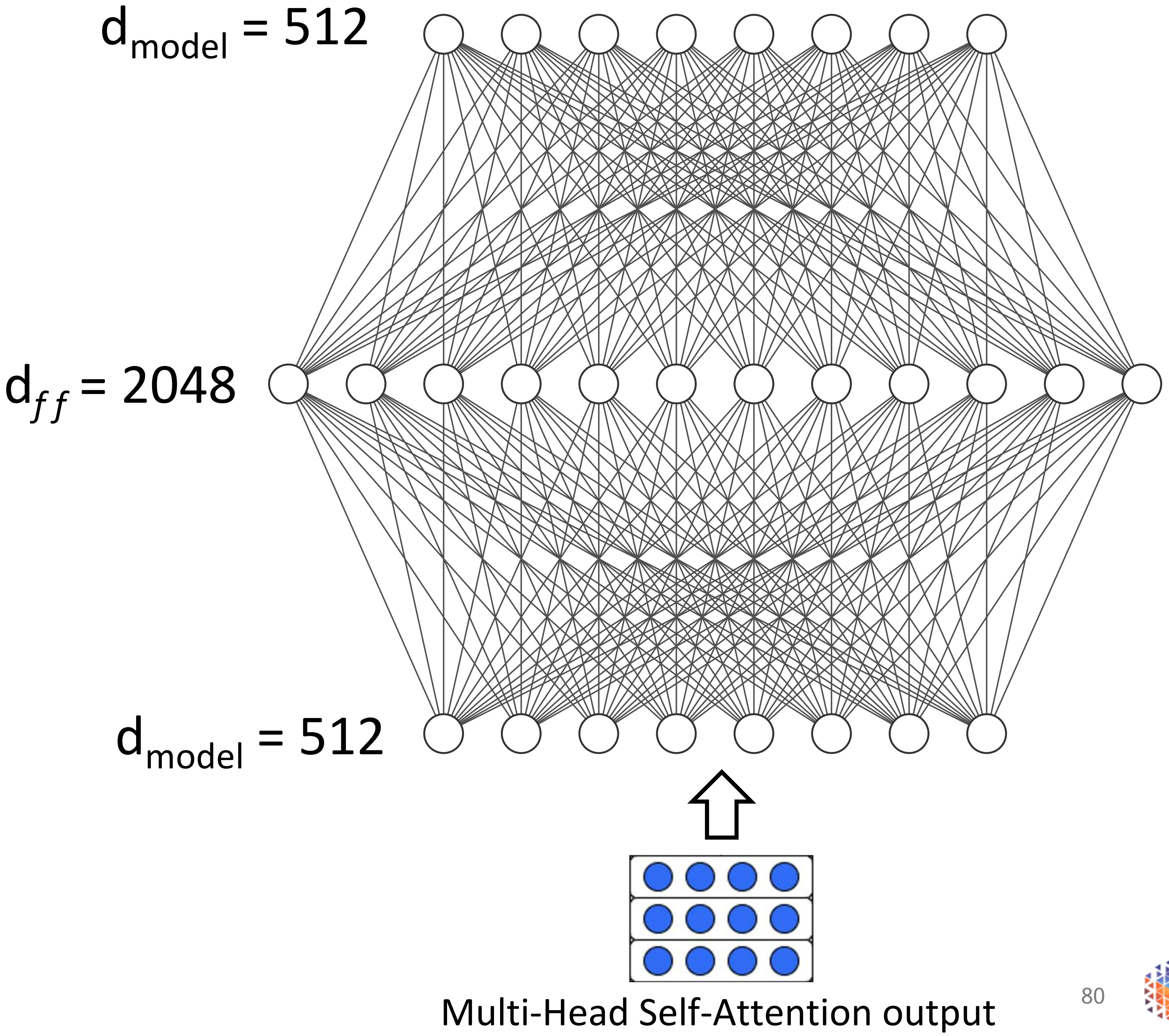
Feed Forward

The input and output dimensionality is the same as the embedding size, which is 512.



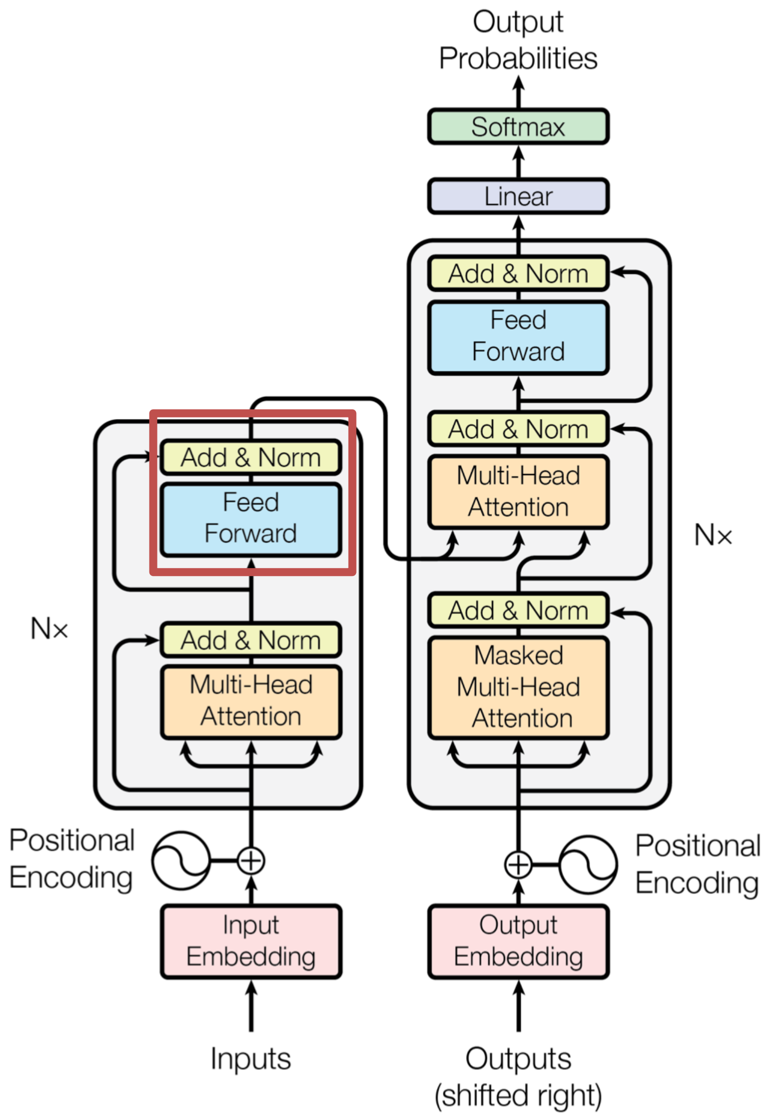
Feed Forward

The inner layer has a higher dimensionality of 2048, which allows the network to learn more complex representations.



Feed Forward

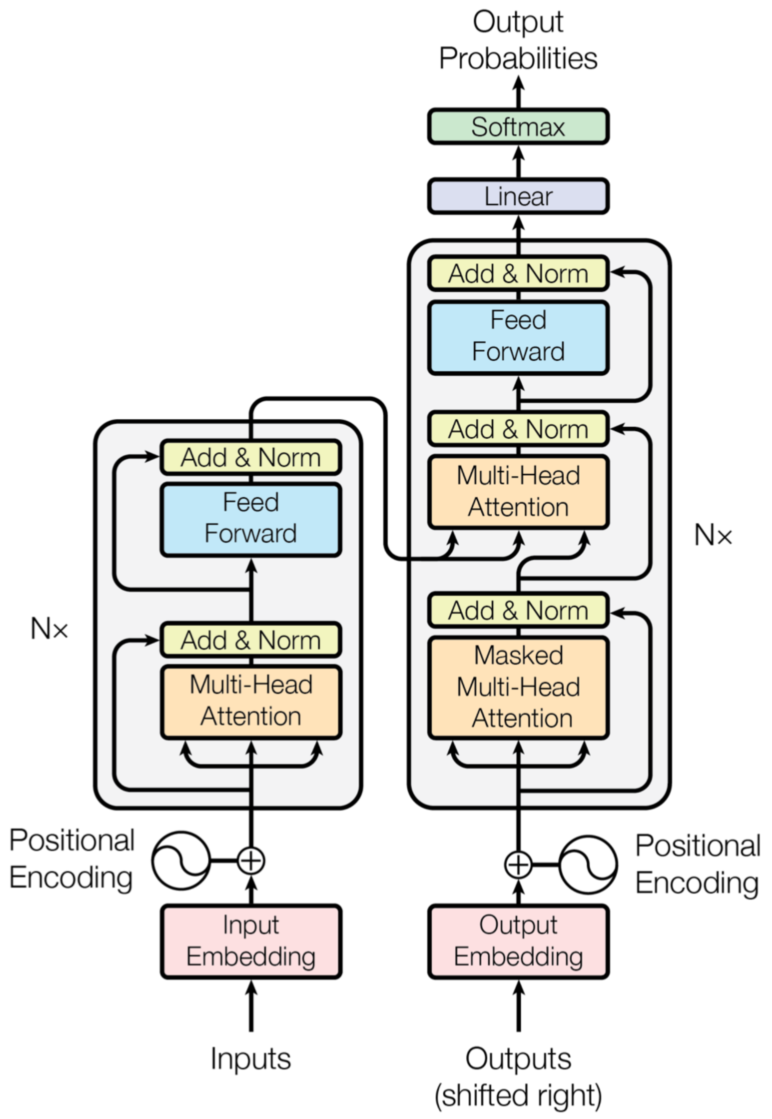
After the feed-forward layer, layer normalization is typically applied to stabilize and accelerate training.



Encoder

Encoder

An encoder is a combination of all that we have learned from the Multi-Head Attention layer to the Add&Norm layer.

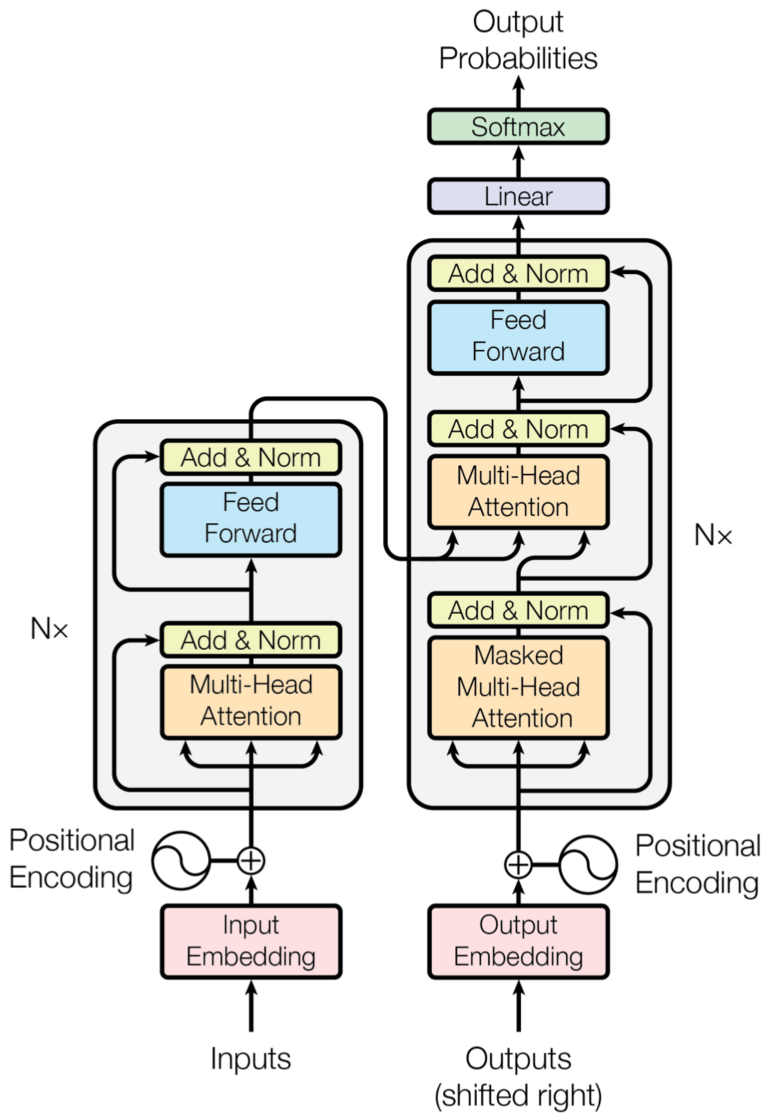


Encoder

The input and output dimensions of the encoder are the same (in the original Transformer, d_{model}=512).

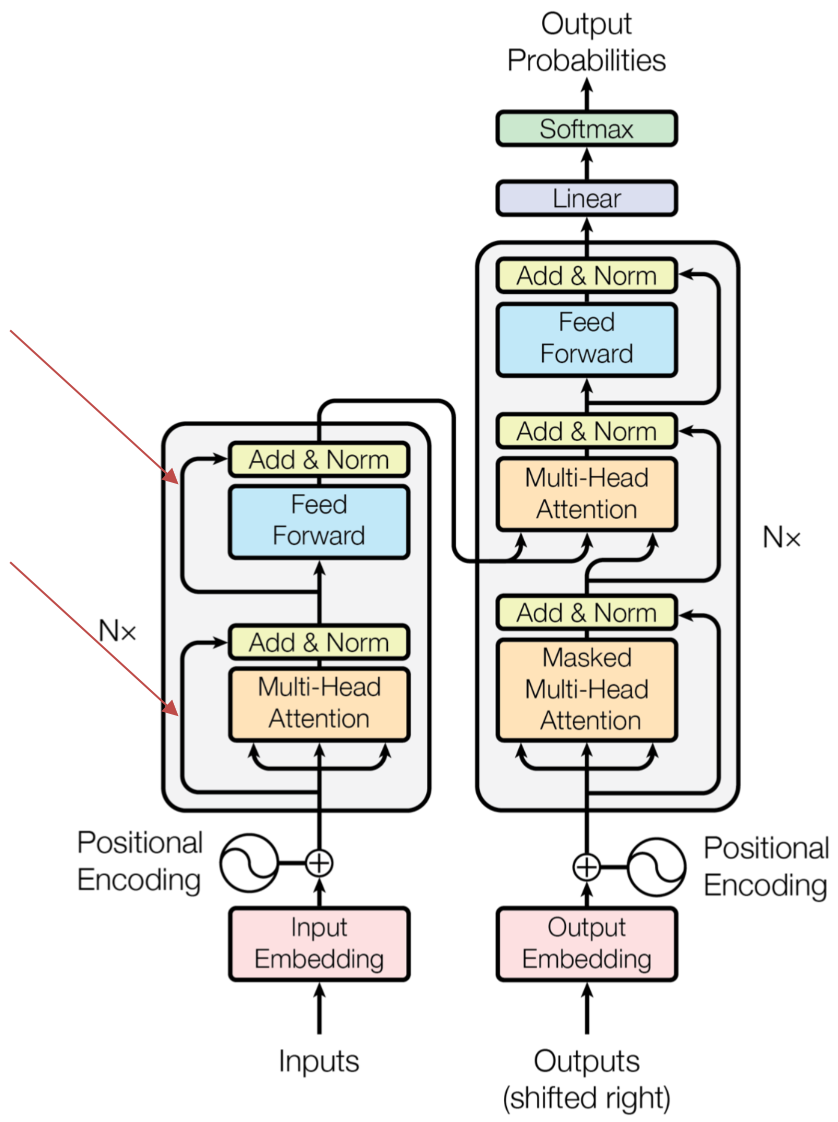
This consistency allows for stacking multiple encoder layers, where the outputs of one layer serve as the inputs to the next layer.

The encoder is composed of a stack of *N=6*



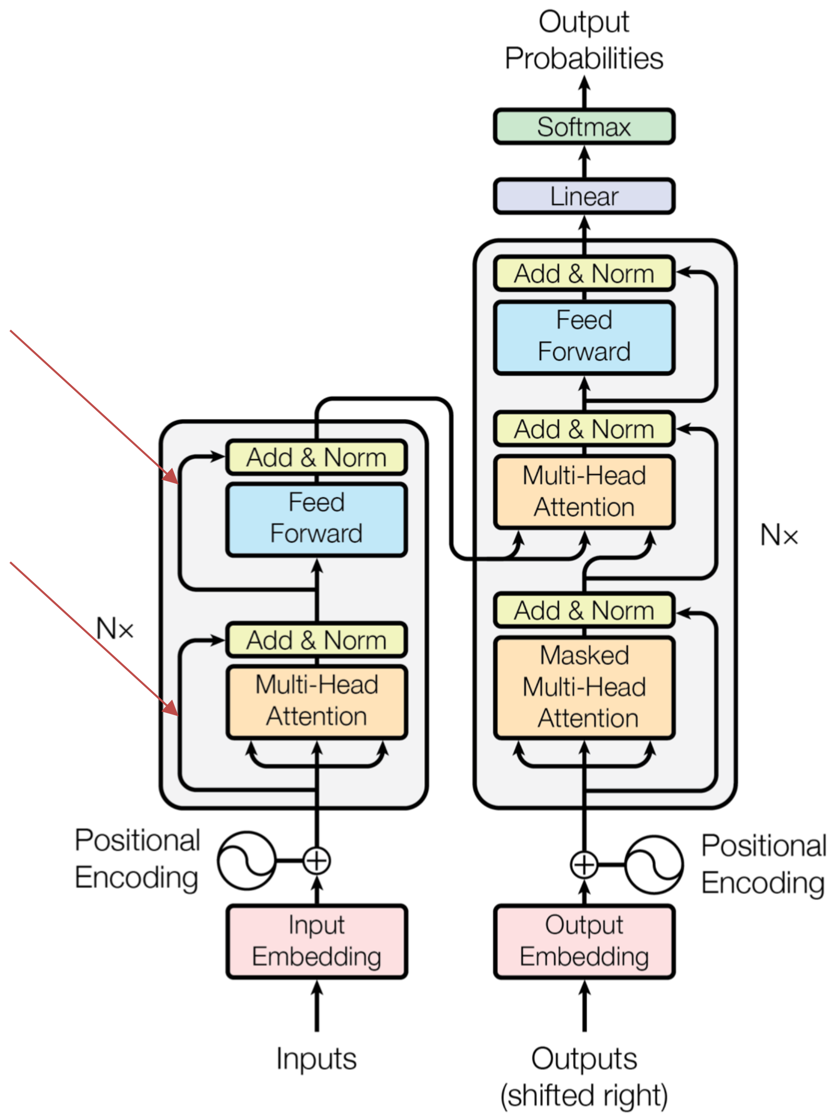
Encoder

- One important thing about the encoder is that it contains a skip connection around each sub-layer.
- The skip connection adds the input of a sub-layer to its output before applying normalization, which helps preserve the gradient



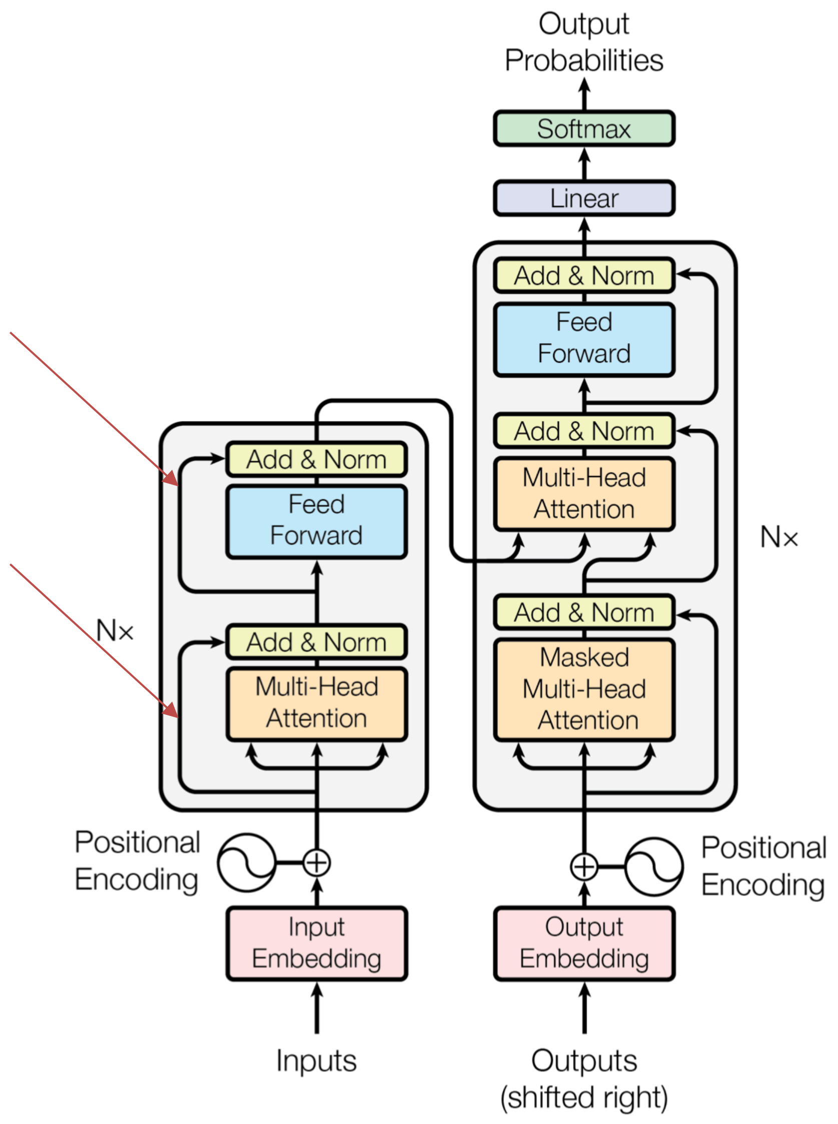
Encoder

This skip connection, also known as a residual connection, allows the model to retain information from the original input, mitigating the vanishing gradient problem and aiding in the training of deep networks.



Encoder

- The skip connection adds the input of a sub-layer to its output before applying normalization.
- This to help preserving the gradient



Hello Transformer!

`Building_A_Transformer_From_Scratch_Encoder.ipynb`