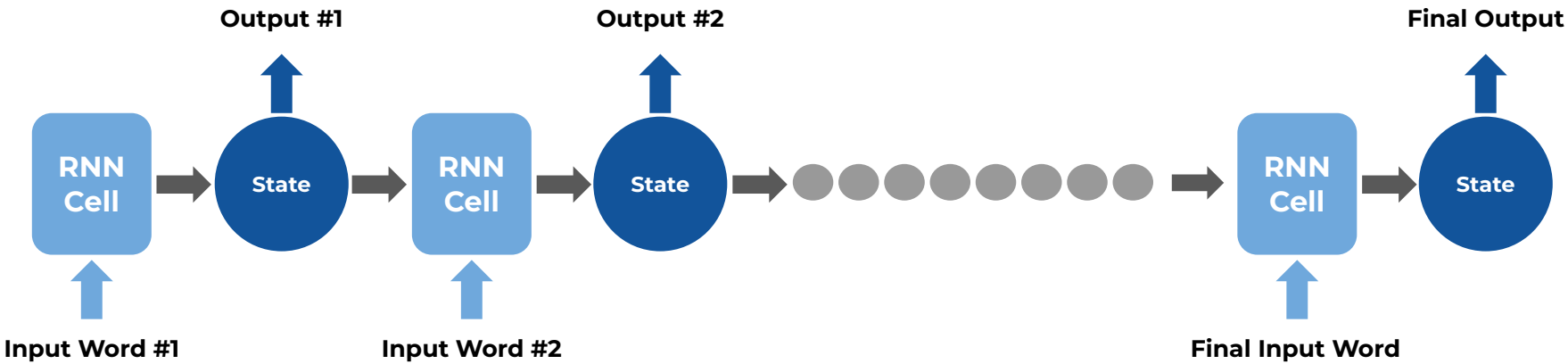
LVC 8: Transformers for Natural Language Processing

Generative AI for Natural Language Processing

Agenda

- Sequential Deep Learning - Recap
- Sequential Deep Learning - Pros & Cons
- The Need for Transformer Models
- Basics of Transformer Models
- Attention Mechanism
- Popular Transformer Architectures

Sequential Deep Learning - A Recap



Sequential Data Processing: The model processes the data one time step at a time, using the previous time steps' data to inform the current time step's predictions. This is different from batch processing, where the model processes all the data at once

Encoder-Decoder Architecture: The next step is to use an encoder-decoder architecture. Encoder encodes the input data into a latent representation, which is passed to the decoder. The decoder generates the output data, one time step at a time, using the latent representation and the previous time steps' output

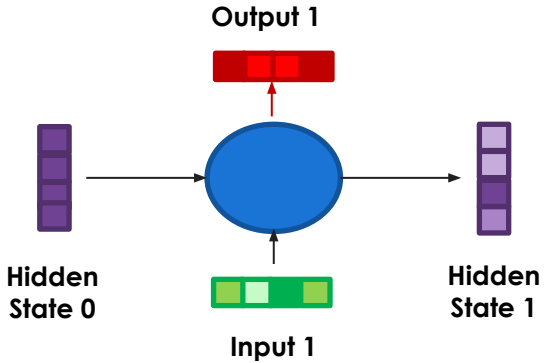
Training and Prediction: The final step is to train the model on the sequential data and use it to make predictions. The model is trained to minimize the difference between the predicted output and the actual output. Once the model is trained, it can be used to make predictions on new, unseen data.

Sequential Deep Learning - Pros and Cons

Pros:

RNNs can process input sequences of fairly large length without increasing or changing the model size.

RNNs encode all their previous states and use that to produce the current output. Therefore, RNNs represent the first DL attempt towards implementing memory.



Cons:

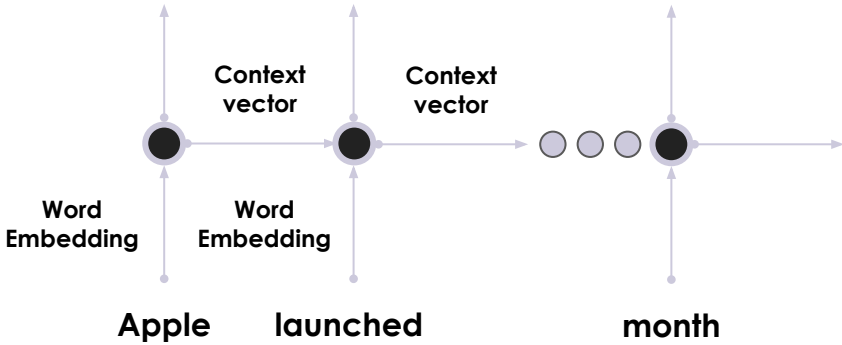
The Long-Term Dependency Problem : RNNs however, have difficulty capturing long-range dependencies and context-dependent information, because the hidden state that is updated at each time step only captures information from the previous time step.

In other words, the model can only capture relationships between elements in a sequence that are within a certain distance from each other. This can make it difficult to capture long-range dependencies, or relationships between elements that are far apart in the sequence.

Sequential Deep Learning - Pros and Cons

Pros:

Sequential NLP models maintain context vectors, which continually get updated as they receive new words in sequence. These context vectors will vary based on the context in which the word occurs. So the word 'Apple' will be captured differently (as a fruit or as an organization, for example) in different contexts through Sequential Deep Learning.



"Apple launched a new iPhone this month."

Cons:

However, RNNs can also be slow to train, because the hidden state must be updated at each time step for each element in the sequence, and sentences are taken in only one word at a time. This can result in slow convergence and increased computational costs.

The Need for Transformer Models

The need for transformers arises from the limitations of sequential models, & the desire for models that can capture long-range dependencies and relationships in sequential data, while also being efficient & scalable.

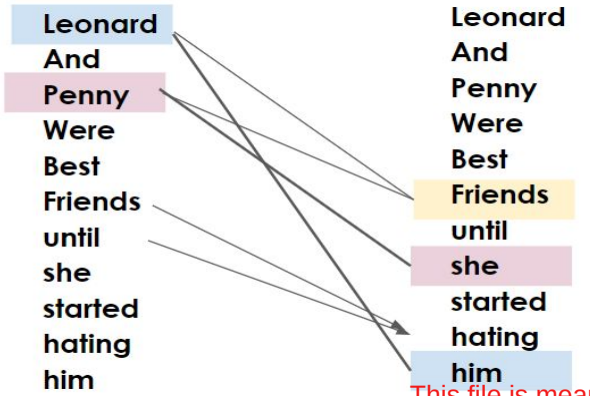
1. They accept the entire corpus of data at once and process it altogether using an attention and positional encoding mechanism, which expresses how one word within the sentence at a particular position is related to all the other words within the same sentence.

2. Due to success of Self attention and Multi-head attention normally massive Transformer models are trained with huge amount of Data. This can perform most diverse tasks mostly in the field of conversational AI while conventional RNNs were limited due to their limitation with parallelizing computations.

The Need for Transformer Models

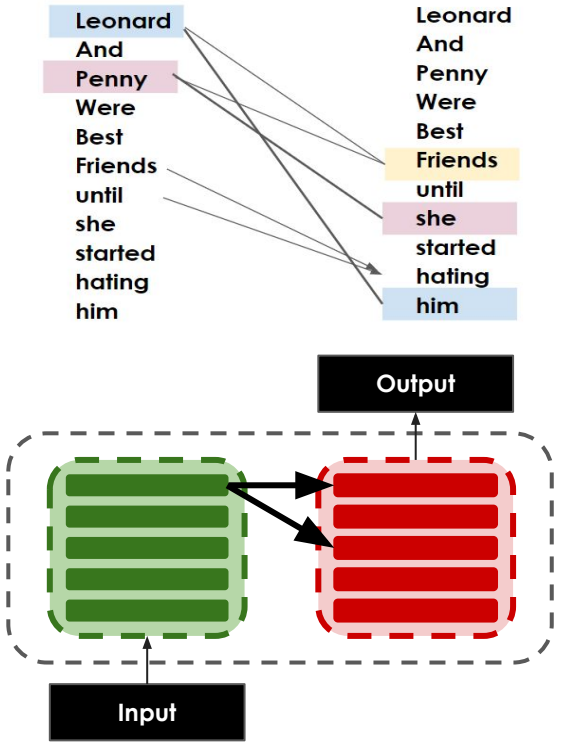
3 They also convert not just words into word embeddings, but sentences into sentence embeddings as well. This is a step-up from word embeddings - these sentence embeddings take care of context and semantic meaning in the sentence.

4 The Transformer uses self-attention and multi-head attention mechanisms to selectively attend to different parts of the input sequence when generating each output token, allowing it to capture complex relationships more effectively.



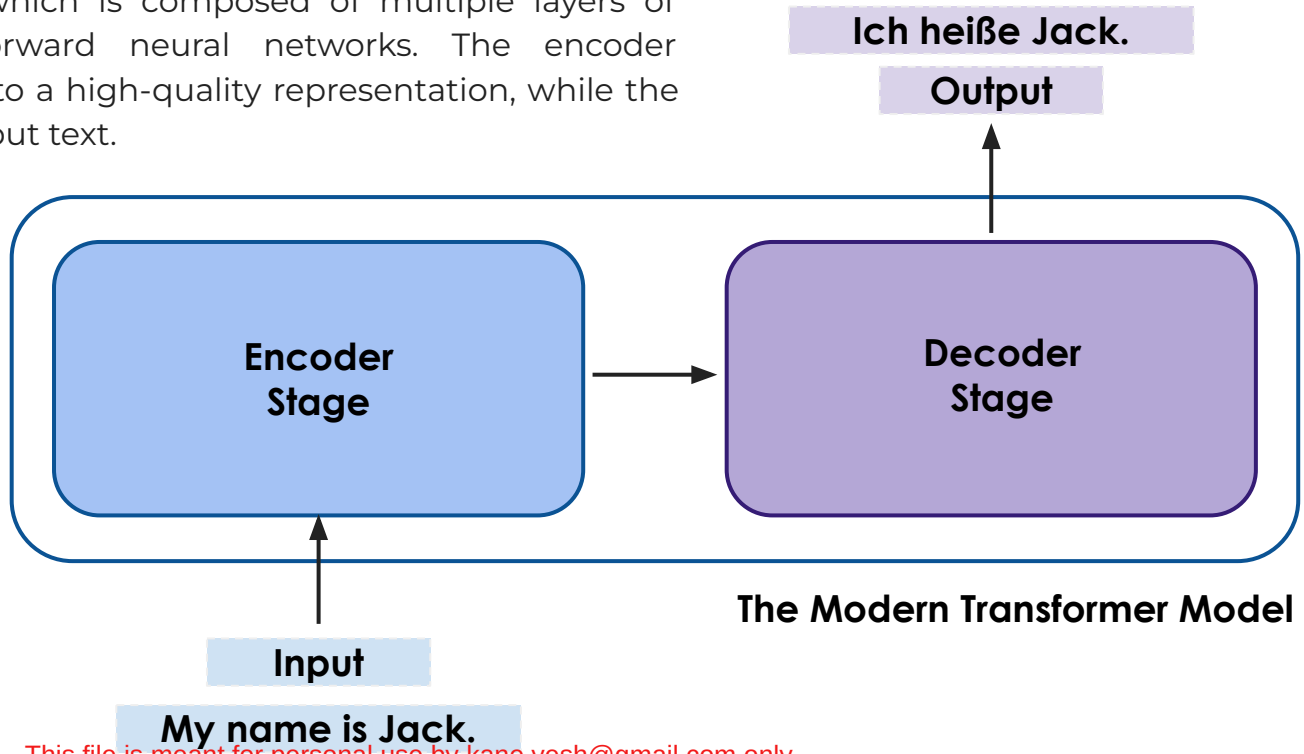
The Basics of Transformer Models

- Transformers are a **type of neural network architecture**
- Transformers were introduced in a paper by **Vaswani et al. in 2017** and have since become widely used in the NLP community.
- Transformers consist of an encoder and a decoder. The encoder takes in a sequence of tokens (e.g. words or characters) and outputs a sequence **of vectors, called "keys," "values," and "queries."** The decoder then takes these vectors as input and outputs a sequence of tokens.
- Transformers are based on the **idea of self-attention**, which allows the model to consider the entire input sequence when computing the output for each element in the sequence.
- The self-attention mechanism in transformers allows the model to compute the **weighted sum of the values based on the similarity between the queries and keys.** This allows the model to selectively focus on different parts of the input sequence as it processes it.
- BERT GPT, XLNet, T5, Electra are popular transformers



Basic Transformer Architecture

The basic transformer architecture consists of **an encoder and a decoder stage,** each of which is composed of multiple layers of self-attention and feedforward neural networks. The encoder processes the input text into a high-quality representation, while the decoder generates the output text.



The Transformer Model - High-level Flow

An **Encoder-Decoder** style architecture is typically used in this type of NLP task, where an input sequence and output sequence are both required, and the output may be very different from the input. This is the case for tasks like Translation and Question Answering.

The way this would work is, **an input sequence is first passed to the Encoder stage** of the Transformer.

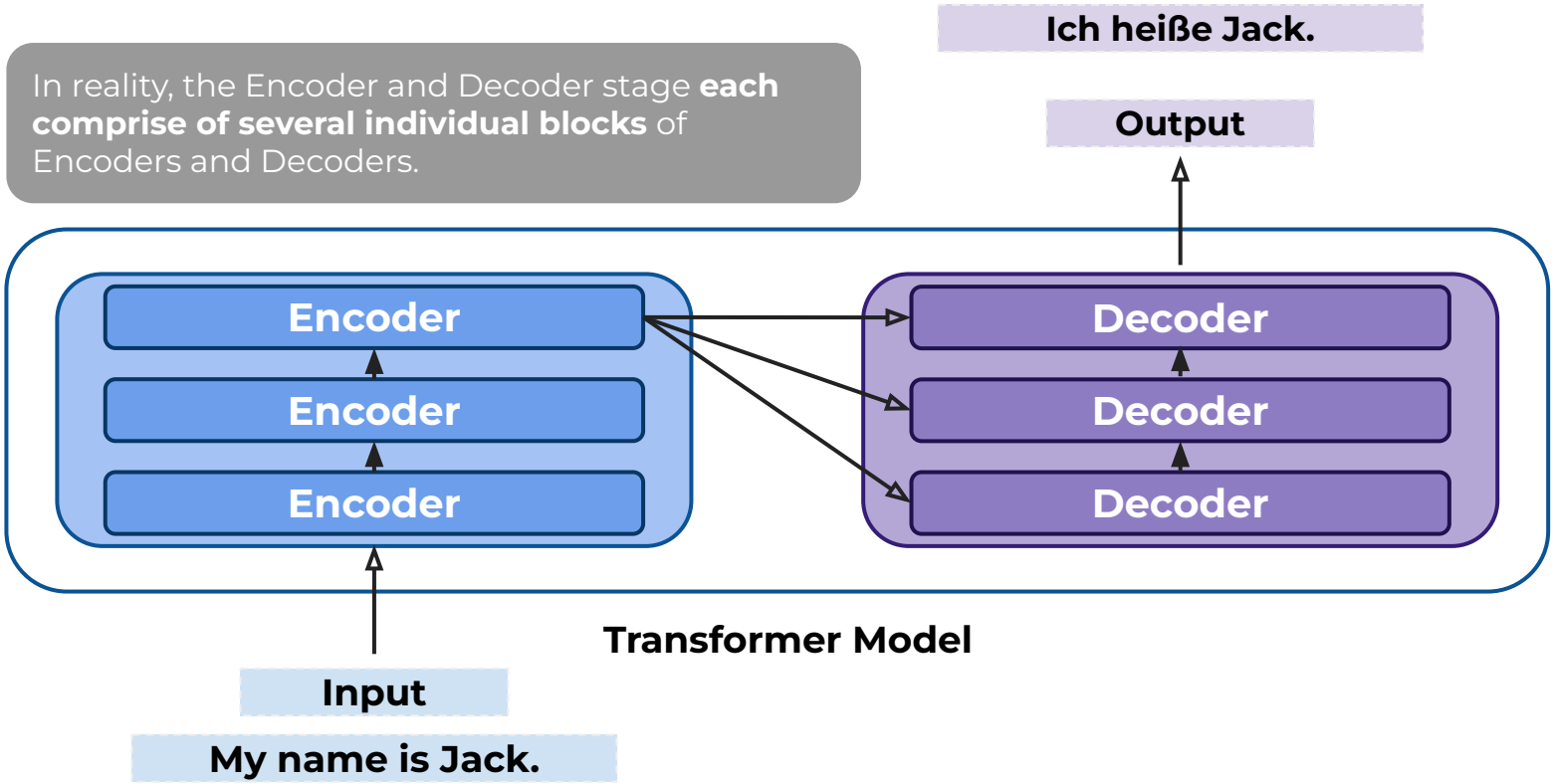
The Encoder stage's operations eventually compute a **high-quality representation of the input sequence**, which has captured its syntactical & semantic meaning.

The Decoder stage is responsible for eventually **"decoding" this representation to a different sentence**, in other words, converting it to the output needed for a task like Machine Translation or Question Answering.

But now, let's go deeper into what the "stages" here refer to.

The Transformer Model - Multiple Blocks

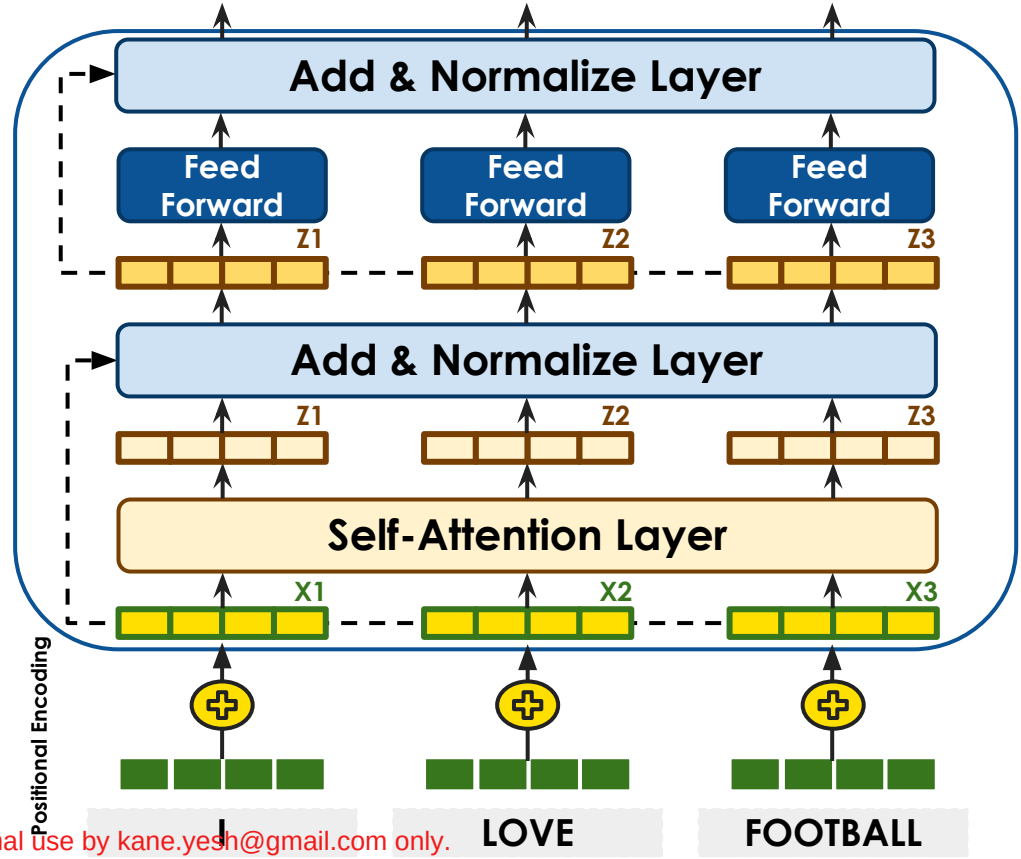
In reality, the Encoder and Decoder stage **each comprise of several individual blocks** of Encoders and Decoders.



Transformer Architecture - Encoder Block

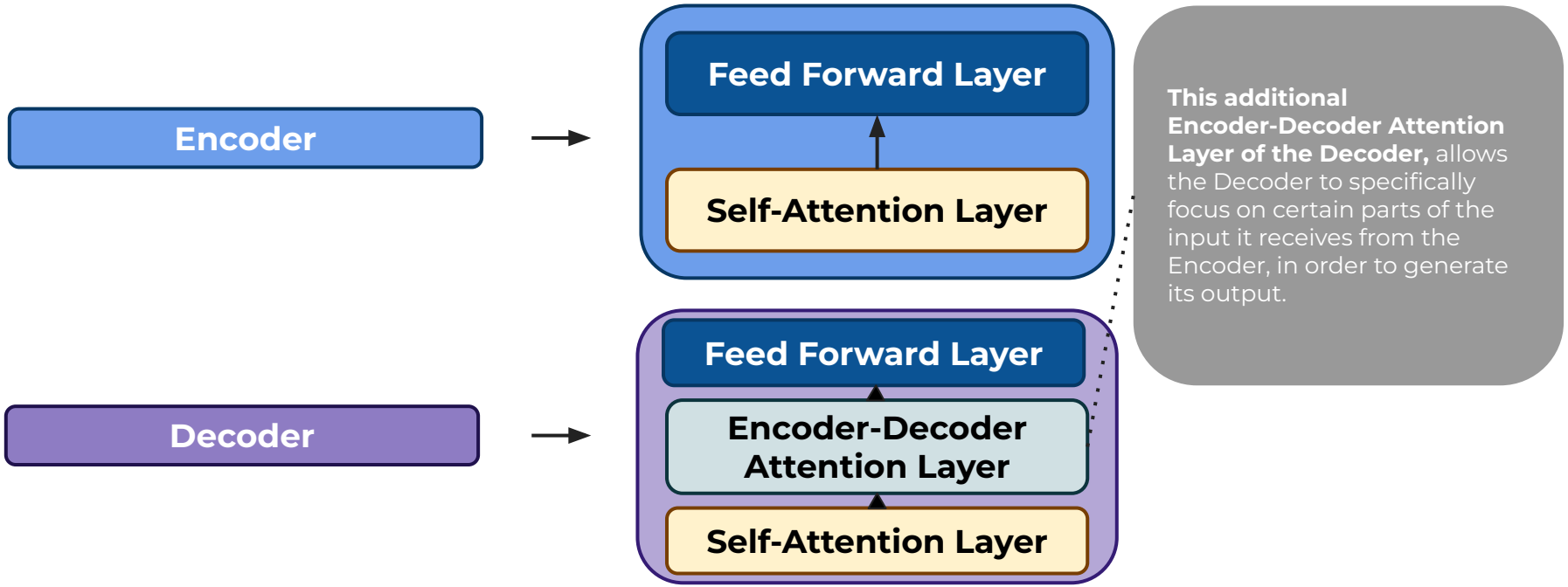
The Encoder block of a Transformer architecture consists of the following components:

1. **Multi-head Attention**: This allows the Encoder to attend to different parts of the input sequence simultaneously.
2. **Feedforward Neural Network**: This component processes the outputs of the Multi-head Attention layer using a standard fully connected neural network with activations like ReLU.
3. **Residual Connections and Layer Normalization**: To improve the flow of information through the Encoder and avoid the vanishing gradient problem, residual connections and layer normalization are added after each sub-layer.
4. **Positional Encoding**: Positional Encoding is typically added to the input embeddings of the Encoder to provide word positional information, using a set of learned sinusoidal functions.



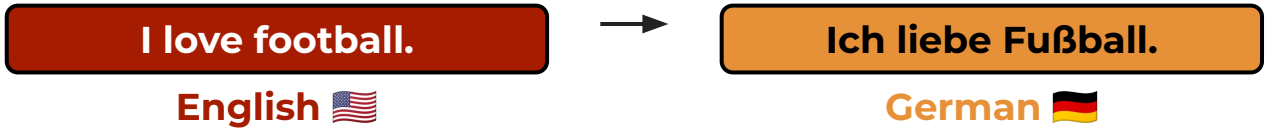
The Encoder vs. The Decoder

At a high level, the Decoder **only slightly differs** from the constitution of the Encoder.

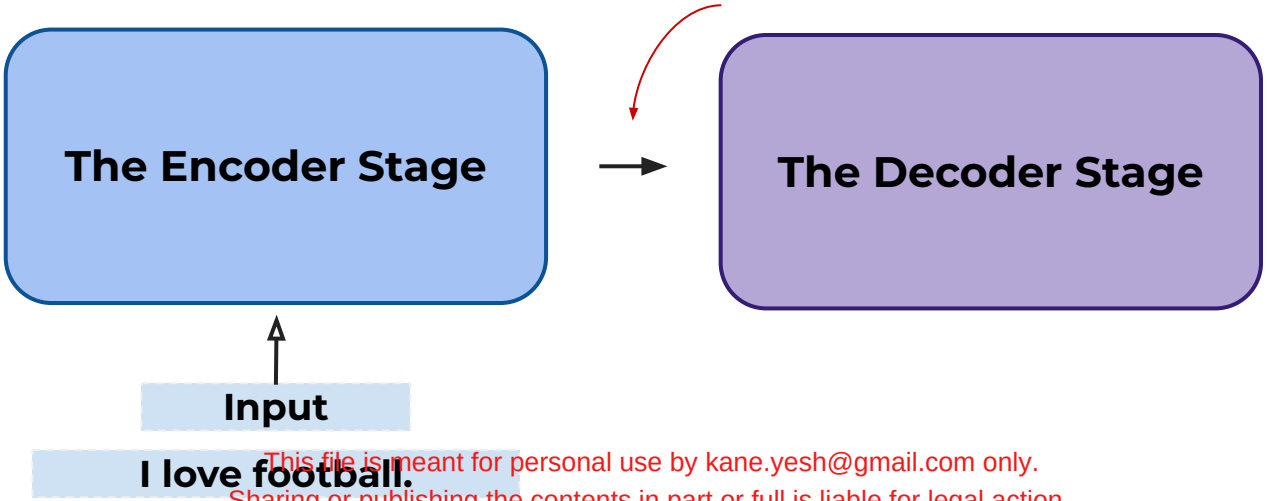


A Peek into the Decoder

Let's assume we're creating this Encoder-Decoder architecture for an **English-to-German Machine Translation task.**



Also, let's remember the Decoder operations start at the point **where the pass through the Encoder Stage has been completed.**



A Peek into the Decoder

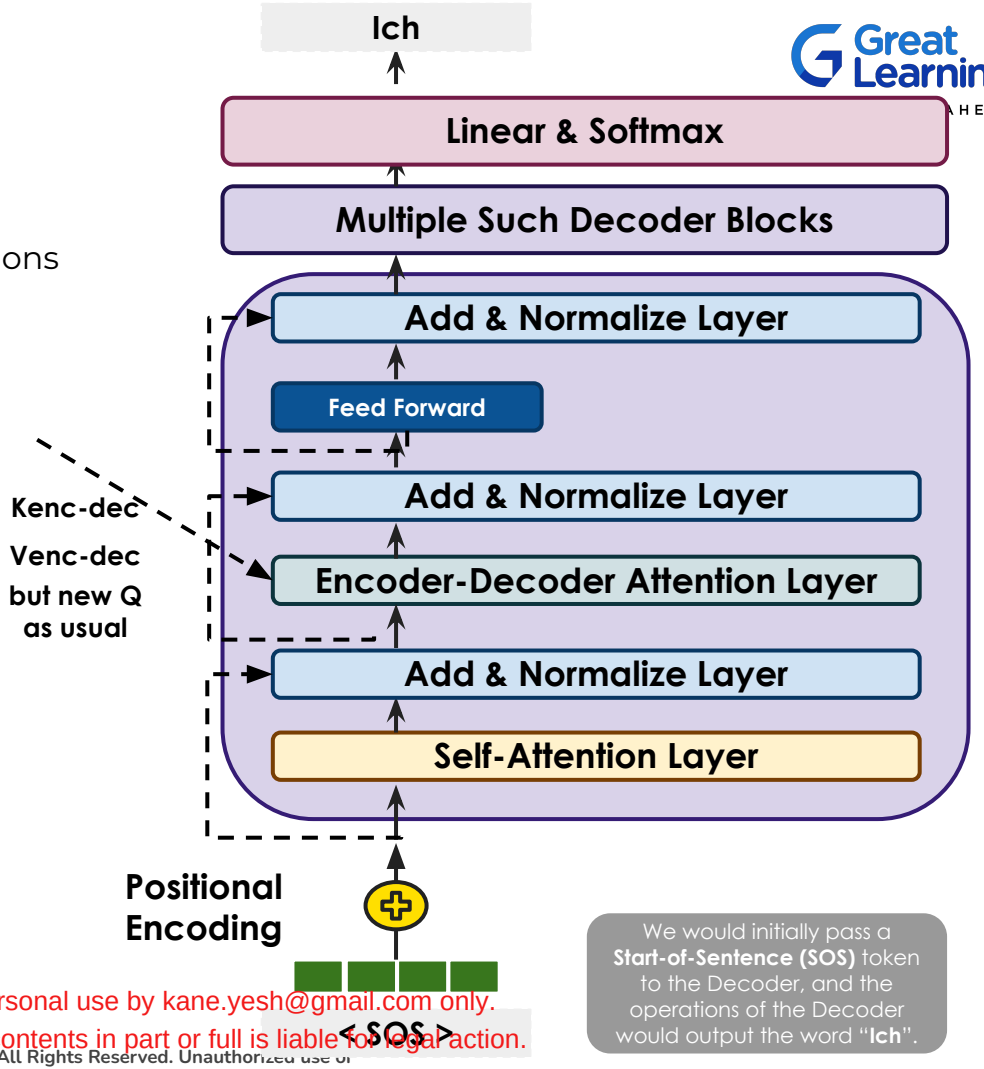
We see immediately that most of these operations are identical to the Encoder.

- **Self-Attention Layer**
- **Add & Normalize Layer**
- **Feed Forward**

But there are a few other operations **unique to the Decoder.**

- **Encoder-Decoder Attention Layer**
- **Linear & Softmax**

Let's understand these differences in some more detail.



We would initially pass a **Start-of-Sentence (SOS)** token to the Decoder, and the operations of the Decoder would output the word "**Ich**".

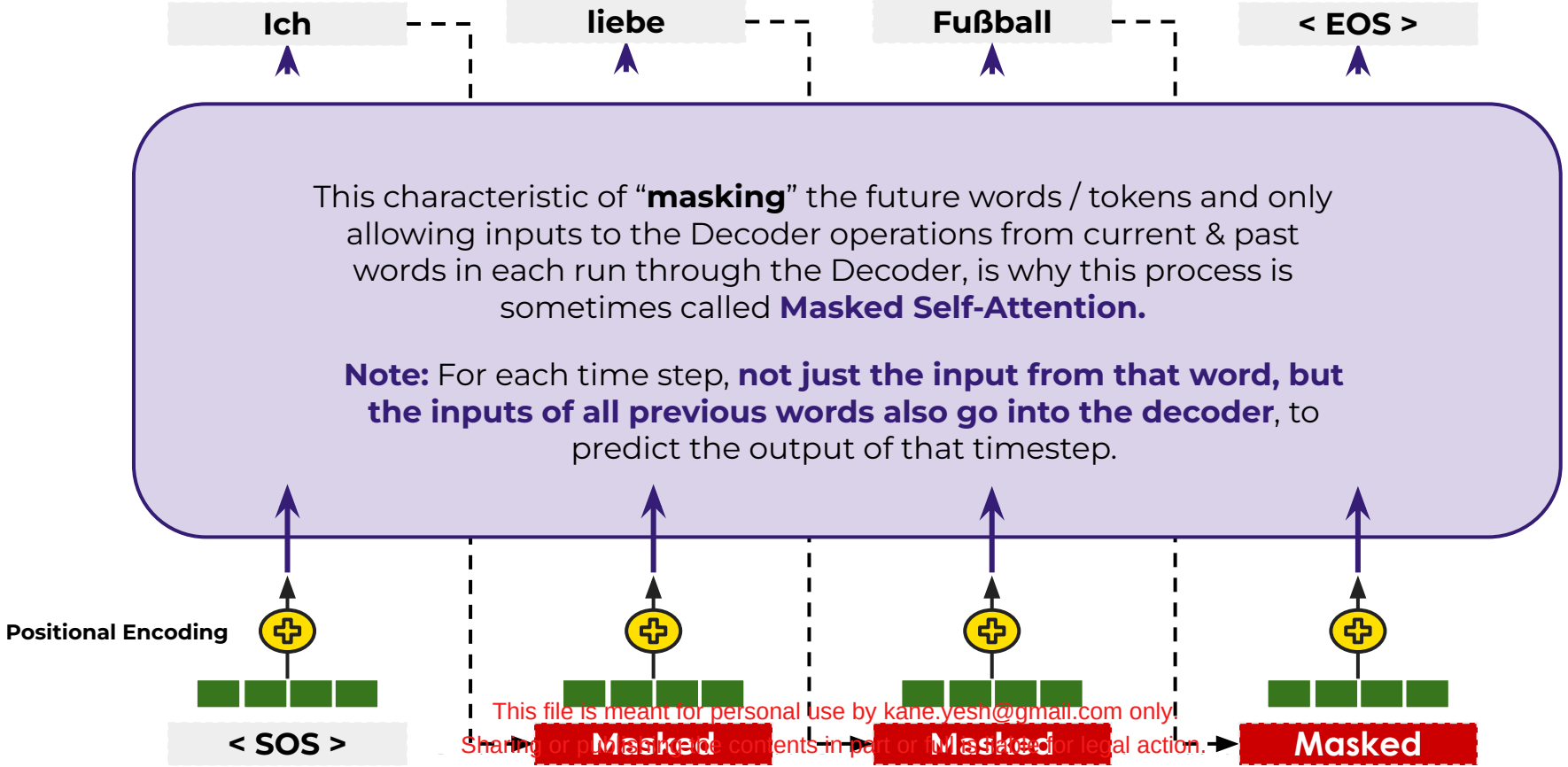
The Decoder's Sequential Nature (Masked Self-Attention)

The first difference to note is that unlike the Encoder, where all the words pass through the Encoder block in parallel, **the Decoder is Sequential in nature**, similar to how we know RNNs and LSTMs operate.

Starting with the <SOS> token, **the Decoder takes a previous word & generates one word at a time**, until it understands it has generated the last word of the sentence, in which case it generates the End of Sentence <EOS> token.

This sequential word-by-word process of the Decoder's text generation makes **the Decoder training stage much more time consuming than that of the Encoder**, and more difficult to parallelize as well.

The Decoder's Sequential Nature (Masked Self-Attention)



The Encoder-Decoder Attention Layer

The other major difference is, of course, the **Encoder-Decoder Attention Layer.**

The difference from normal Self-Attention is that in this layer, **the K and V vectors are not generated from the input embeddings to this layer**, the way they were in the normal Self-Attention layer.

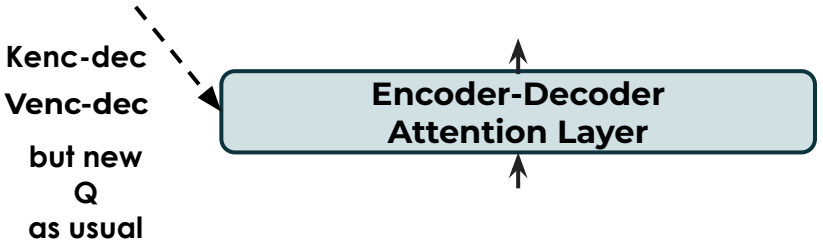
In fact, we utilize a **K encoder-decoder (K enc-dec)** and a **V encoder-decoder (V enc-dec)** in this layer, whose source is from the **final output of the Encoder stage**.

The Encoder-Decoder Attention Layer

We directly utilize **the final embedding vectors** generated at the end of the Encoder stage, and multiply those with weight matrices to get **K enc-dec** & **V enc-dec**. These get used as K and V in this Encoder-Decoder Attention Layer.

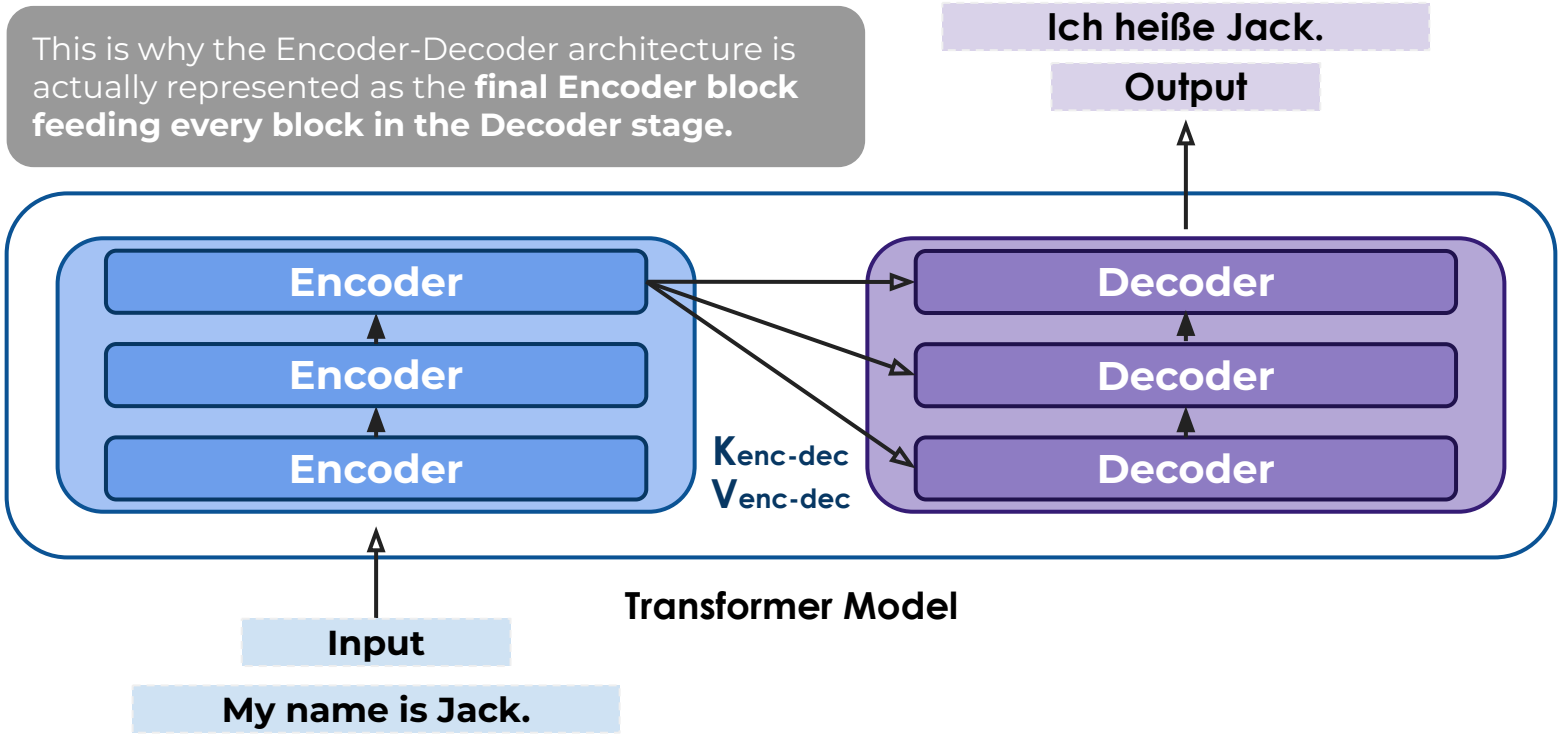
It is only the Q vector that this layer creates from the input to it, the way that normally happens in the Self-Attention Layer (where all three of K, Q & V are directly created from the input embeddings to the layer).

It is also important to mention, that the **Q for < SOS >** (Dec Pos 0) for example, **only relies on the K enc-dec & V enc-dec of the word "I"** (Enc Pos 1) from the input, to predict the word "Ich".
This happens for every Decoder word.



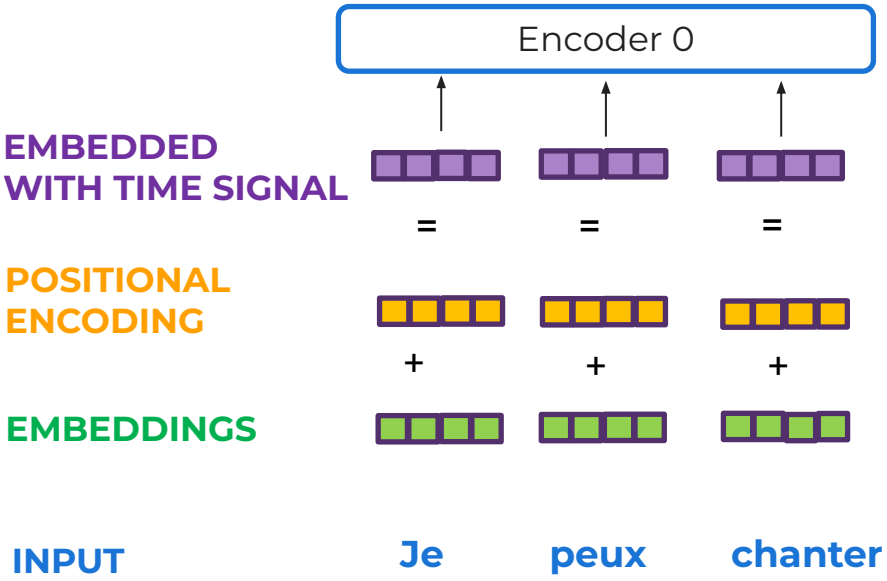
The Encoder-Decoder Attention Layer

This is why the Encoder-Decoder architecture is actually represented as the **final Encoder block feeding every block in the Decoder stage.**



The arrows from the final Encoder block to each Decoder block represent the **K enc-dec & V enc-dec from the final Encoder layer being used in the Encoder-Decoder Attention Layer of each Decoder block** in the Decoder stage.

Word Embedding & Positional Encoding



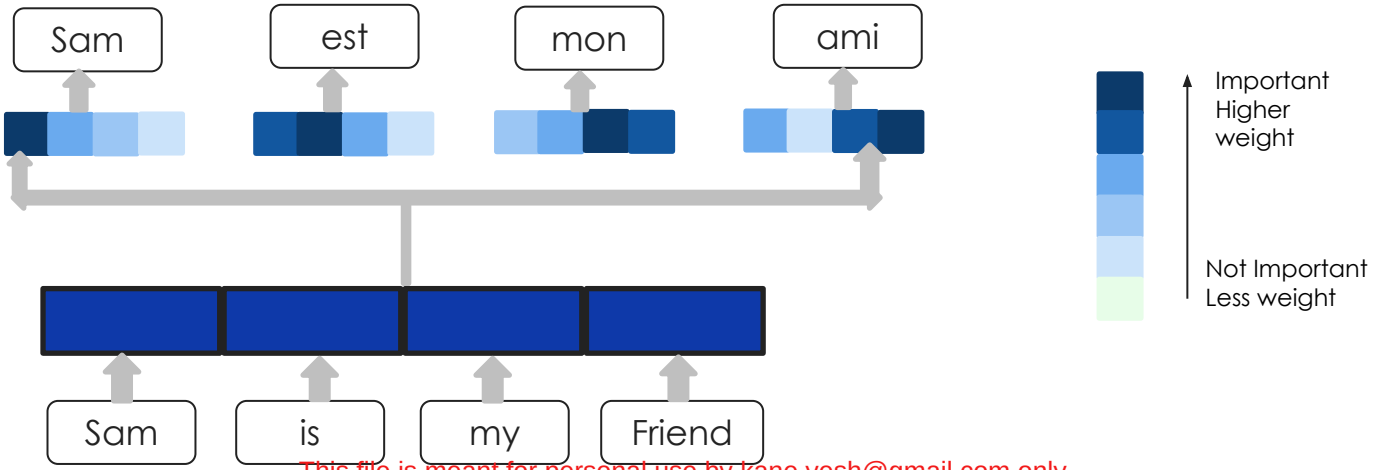
Word embedding is the process of representing each word in the input sequence as a vector in a high-dimensional space

Positional Encoding is a way to account for **the order of the words in the input sequence.**

Positional Encoding is a vector added to each input embedding. These vectors follow a specific pattern that the model learns, which helps it determine the position of each word, or the distance between different words in the sequence.

Attention Mechanism Intuition

The attention mechanism in neural networks is a technique used to dynamically weigh and sum up different elements of a vector, called "**attention weights**", in order to produce a **context-aware output representation**. The idea is to assign higher attention weights to the more important elements of the input, **allowing the network to focus on relevant parts & automatically identify important features.**



English to French context based translation

The Attention Weights

In a neural network, the attention mechanism is used to selectively focus on certain parts of the input data when processing it. This is done by computing a set of attention weights, which are values that indicate how important each element of the input data is for the current task.

The attention weights are typically computed using a dot product attention mechanism, which compares the

- **query vector** (representing the context in which the input data is being processed)
- **the key vector** (representing the input data).

The resulting attention weights are then used to compute a weighted sum of the value vector (representing the input data), which produces the output representation.

The KQV Matrix

The KQV matrix is a crucial component in the self-attention mechanism of the Transformer architecture. Here are the steps for computing the KQV matrix

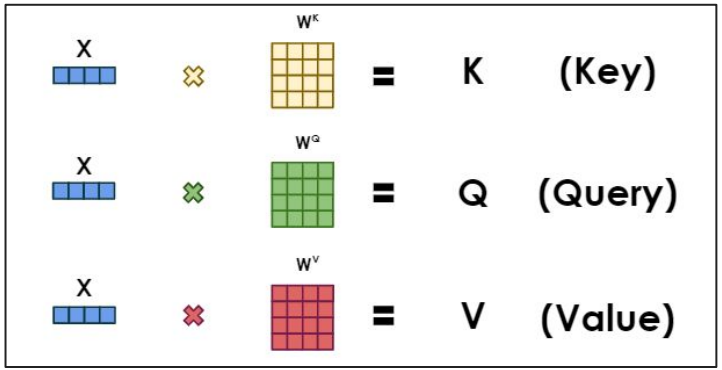
Compute the query (Q), key (K), and value (V) matrices for the input sequence: The Q, K, and V matrices are computed by multiplying the input sequence with learned weight matrices

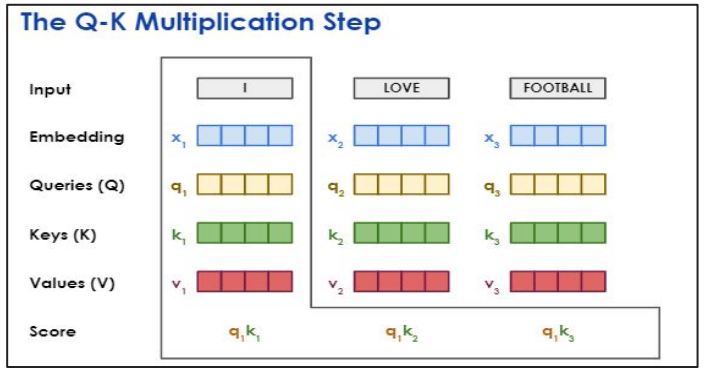
Compute the scaled dot-product attention scores: The attention scores between each query and key are computed by taking the dot product of Q and K, and dividing the result by the square root of the dimension of the key vectors.

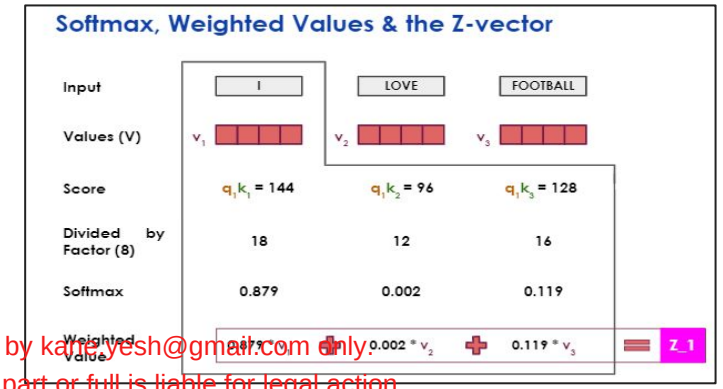
Apply the softmax function: The attention scores are passed through a softmax function, which scales the scores so that they sum to one and represent the weights to be applied to the corresponding values.

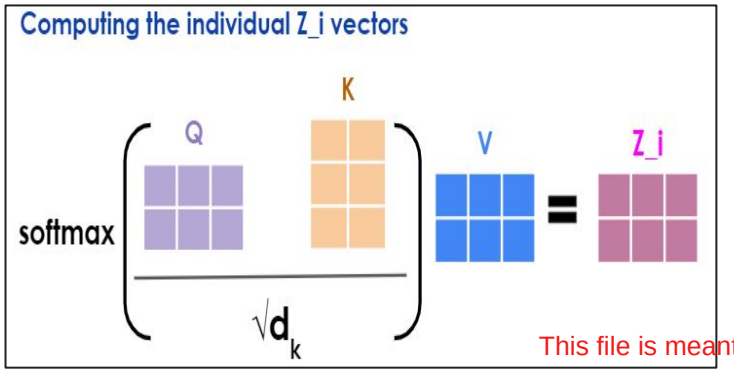
Compute the output of the self-attention layer: Output of the self-attention layer is computed by taking the weighted sum of the values V, using the attention weights obtained in step 3. This can be efficiently computed using matrix multiplication.

The KQV Matrix Computation Step









The Linear & Softmax Layers

At the end of the Decoder stage, there's a **Linear and Softmax** layer that performs a fairly simple operation needed to get the final word prediction.

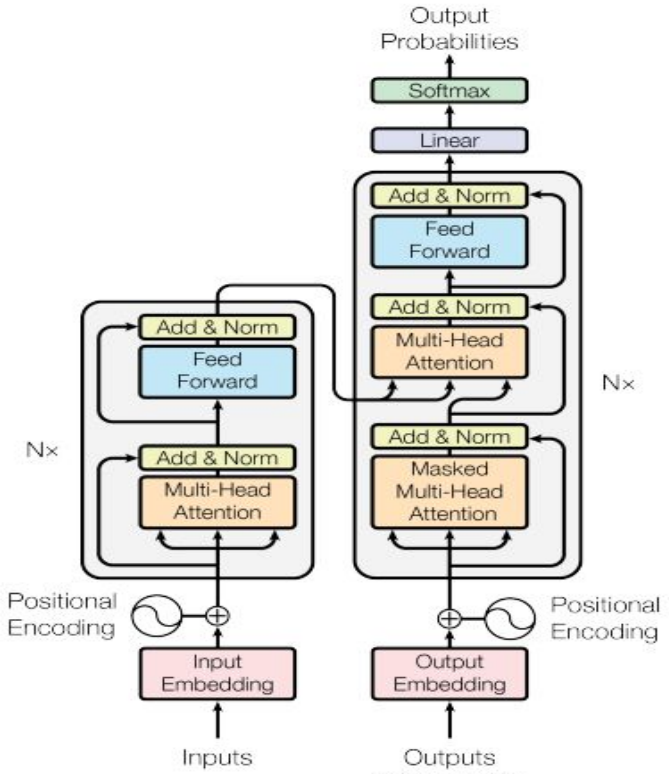
The **Linear** layer is merely a fully-connected layer of neurons, with a number of nodes equalling the size of the entire vocabulary, in addition to some special tokens.

This is then fed to the final Softmax layer, which converts the numerical outputs into probabilities, so that **the word with the highest probability can be selected as the output of the Decoder**, in the style of a **multi-class Classification problem.**

Finally, **Categorical Cross-Entropy** is the loss function used for backpropagation.

This construct is called the **Language Model Head**, and this is how **the Decoder eventually generates a word at each sequential time step**!

Bringing It All Together



The Transformer follows this overall architecture using stacked self-attention and point-wise, fully connected layers for both the encoder and decoder

Source: Image from original research paper
Attention is all you need. 1706.03762.pdf (arxiv.org)

The Evolution of Large Language Models

2018

The BERT Transformer

The first popular modern Transformer architecture. A favorite among practitioners.

2018-2019

RoBERTa, XLNet & DistilBERT

Popular BERT variations which improved on BERT's training data, procedure, performance and computational cost

2018-2021

The GPT Family

OpenAI's autoregressive Decoder-only models (GPT, GPT-2, GPT-3)

2023+

ChatGPT & Beyond

Generative Transformer models applied to Conversational AI with increasingly exhaustive text corpuses for training

BERT

Bidirectional Encoder Representations from Transformers

Although the original Transformer paper had come out in 2017, it wasn't until the **release of BERT in 2018** that the industry truly started to take notice of Transformer architectures for NLP.

BERT was released by a team from Google, and has become the de-facto industry baseline in NLP due to its ubiquitous good performance on all categories of NLP tasks.

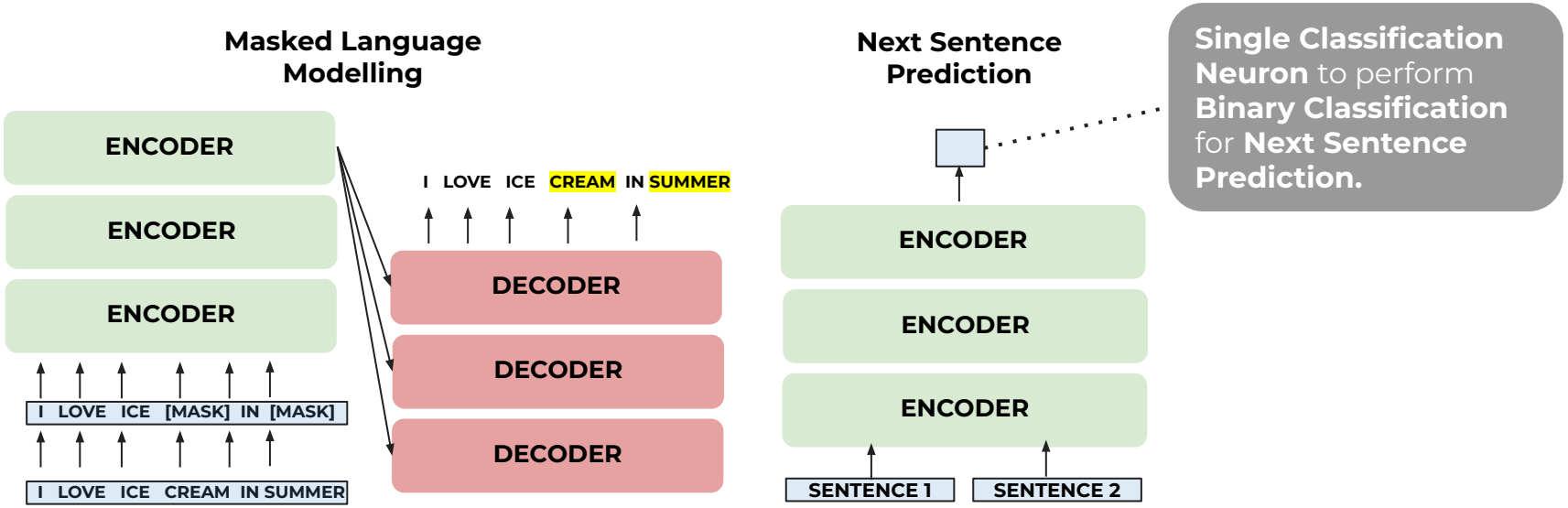
BERT's training essentially consists of two stages: **A Pre-training Stage** and a **Fine-Tuning Stage**

Pre-training Stage + **Fine-tuning Stage**

BERT - Pre-training + Fine-tuning

The **Pre-training** of BERT was done using an Encoder-Decoder architecture on **Masked Language Modeling (MLM).**

Then, **only the Encoder blocks** are taken for **Fine-tuning** on **Next Sentence Prediction (NSP).** The fine-tuned Encoders are what **eventually become BERT.**



BERT - Variants

The BERT architecture has **2 variants**, both of which are **Encoder-only architectures.**

1. **$BERT_{BASE}$** has **12 Encoder blocks**, hidden dimension of 768, 110 M parameters & **12 Attention heads**.
2. **$BERT_{LARGE}$** has **24 Encoder blocks**, hidden dimension of 1024, 340 M parameters & **16 Attention heads**.

BERT's pre-training allows it to learn **high-quality latent representations** of words and sentences, **including context**.

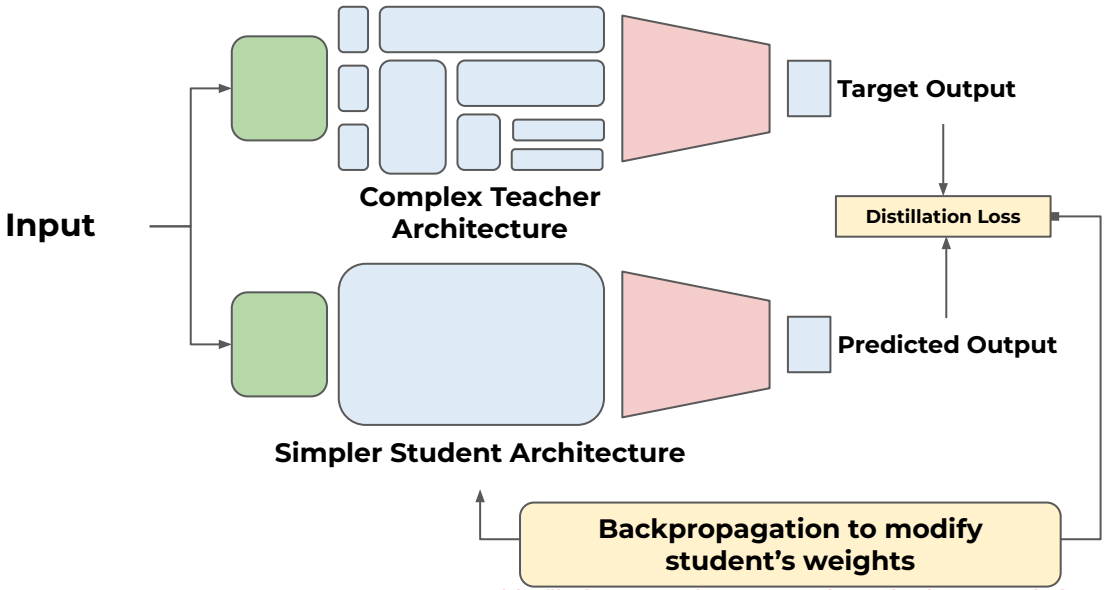
BERT can be further **fine-tuned with fewer computational resources on smaller datasets** specific to various NLP tasks.

The contextualized embeddings BERT returns means that it will return **different embeddings even for the same word** appearing in different contexts.

The success of BERT has made it a staple of **Google's NLP operations on its Search Engine** in multiple languages.

DistilBERT - The Idea of Knowledge Distillation

DistilBERT goes down a more important direction, and applies an idea to improve BERT's computational cost that is generalizable to any Deep Learning model - **Knowledge Distillation**.



In DistilBERT, **a simpler student model is simply trained to replicate the BERT (teacher's) performance** through the mechanism described.

DistilBERT - The Idea of Knowledge Distillation

Great Learning
POWER AHEAD

This simple idea, also called **Teacher-Student Training**, allows us to reduce the sometimes unnecessary size and complexity of Large Language Models like BERT, and is a boon to resource-constrained settings such as individual laptops, mobile phones and other edge devices that still need to deploy heavy Deep Learning applications.

With its much smaller size, **DistilBERT** was shown to **reduce the size of BERT by 40%**, improve on compute speed by 60% and yet, still retain **97% of BERT's Natural Language Understanding abilities.**

Training a Student Model on the output of a Teacher Model in this manner, forces it to become more efficient, **allowing us to build a much simpler model** that can adapt its parameters to perform nearly as well as a more complex model.

This idea can of course be applied to any architecture as the Teacher, or even recursively applied to continually try to create simpler models.

The GPT Family - Decoder-only Models

Generative Pre-trained Transformer

OpenAI, on the other hand, went down a different path in the pursuit of creating Large Language Models (LLMs) that are capable of generating high-quality text - **Decoder-only Architectures.**

Decoder-only models generate Natural Language text in an **autoregressive fashion** (the output of one timestep is the input to the next) based on a **prompt**.

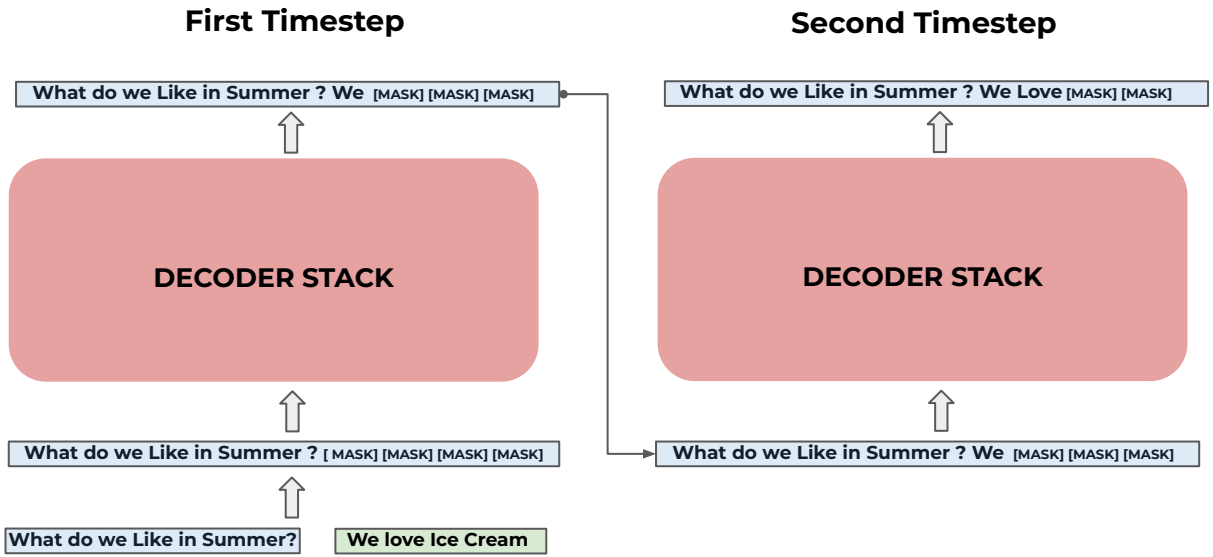
The difference from the earlier Encoder-Decoder style architecture is that in a Decoder-only model, **the initial input prompt is also used as input into the Decoder.**

This is the key architectural innovation behind the GPT series of models:
GPT (2018), GPT-2 (2019) and GPT-3 (2020)

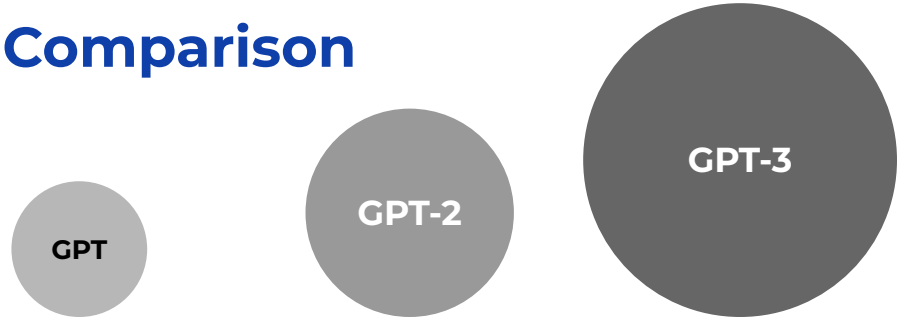
Outside of this, the GPT models have shown an emergent better understanding of Natural Language simply as the size of their training dataset has increased.

The GPT Family - Training Process

The following is an example of how the GPT models (GPT, GPT-2 & GPT-3) achieve this autoregressive text generation **by utilizing the input prompt text as well.**



The GPT Family - A Comparison



	GPT	GPT-2	GPT-3
Parameters	117 Million	1.5 Billion	175 Billion
Decoder Blocks	12	48	96
Context Token Size	512	1024	2048
Hidden Layers	768	1600	12288
Batch Size	64	512	3.2 Million

Types of Transformers

There are broadly **three-types** of Transformer-based Large Language Models today, based on their usage of Encoder and Decoder blocks:

1

Encoder-only Transformers only aim to use Encoders to generate continuous vectors / embeddings of their input text, and do not generate directly-usable output words.
BERT and RoBERTa are examples of such Encoder-only models.
Encoder-only LLMs find use in Discriminative tasks that only involve the use of Embeddings, such as Text Classification or Semantic Search.

2

Encoder-Decoder Transformers utilize the Encoder and Decoder blocks in tandem, similar to the original Transformer architecture. These **Encoder-Decoder** Hybrid Models are typically used in Generative tasks where the output heavily relies on the input and cross-attention is needed to get a good mapping, such as **Machine Translation** and **Text Summarization**.
T5 and FLAN-T5 are examples of Encoder-Decoder Hybrid Transformer LLMs.

3

Decoder-only Transformers are currently the most popular type of Transformer powering modern LLMs, and they utilize only Decoder blocks to auto-regressively predict the next token from large unsupervised corpuses of text prompts, making them simpler to train & more general-purpose. They are suited for other generative tasks like **Completion and Q&A**.
The GPT and Llama models are good examples of Decoder-only LLMs.

Discriminative AI/NLP Problems now becoming a Special Use-Case of Decoder-only Generative AI/NLP

Because Decoder-only LLMs are able to generate any output that is a logical continuation of the input text in some way, they can also tackle the kind of problem statements typical of classical NLP, such as Text Classification, Text Completion, Named Entity Recognition or Part-of-Speech Tagging.

System Prompt

You are a helpful Text Classifier. Your job is to look at the text provided (which will be a customer review of a business) and only output a sentiment “Positive”, “Neutral” or “Negative”. Do not output anything else. I repeat - you are not allowed to output any other words, just one of the three options provided above with respect to what you feel is the most likely sentiment of the customer review provided.

User Query

I ordered a waffle, and was shocked to find hair on the food. I asked for a refund but was told I couldn’t get one, I would only be given a replacement, however I would need to wait 30 minutes for the replacement - terrible customer service. Please do not visit this place.

LLM Response

Negative

LLM behaving as a Discriminative Classifier by “generating” Classification labels

This is how **Decoder-only LLMs** have allowed Generative AI to subsume the earlier methods and problem statements of NLP and **create a more generalizable AI solution**, that uses the same model to adapt to any NLP task required, whether it’s Classification, Question Answering or Translation.

Fine-tuning Transformers & LLMs
The Goal of Open Source End-to-End LLM Solutioning

In the examples we have seen in the course so far, we have interacted with Transformers in the following two ways:

1	**Prompting / In-context Learning**	Providing system prompts & user queries with few-shot examples to modify the outputs of the LLMs in tune with business requirements
2	**Indexing / Vector Database RAG**	Providing a vector database of text data for additional context to help the LLM generate contextual and more factual answers minimizing hallucinations

However, *neither of these ways truly modifies the weights & parameters of the core Transformer model behind the LLM*, and as such, the above 2 ways are limited in their flexibility and ability to give high-quality outputs for specialized domains / tasks outside the purview of the generalist web-based text corpuses LLMs are trained on.

Adapting LLMs to specific domains such as Healthcare, Legal or Finance, which have specialized text corpuses where the meaning and relationship of various words may significantly differ, requires us to tweak the core reasoning abilities of the LLMs and change the values of the weights / parameters they have learnt from pre-training - in other words **LLMs need to be fine-tuned to meet the needs of specific domains / industries.**

Later in this course, we will see examples of how the industry has learnt to fine-tune LLMs, specifically the famous Parameter-Efficient Technique called **Low-Rank Adaptation (LoRA)** and a variant called **QLoRA**.

Great Learning

Happy Learning !

APPENDIX

From BERT to RoBERTa

Robustly Optimized BERT pre-training Approach

The **RoBERTa** model, released by a Facebook AI research team in 2019, was just a modification to key hyperparameters in the way BERT was trained.

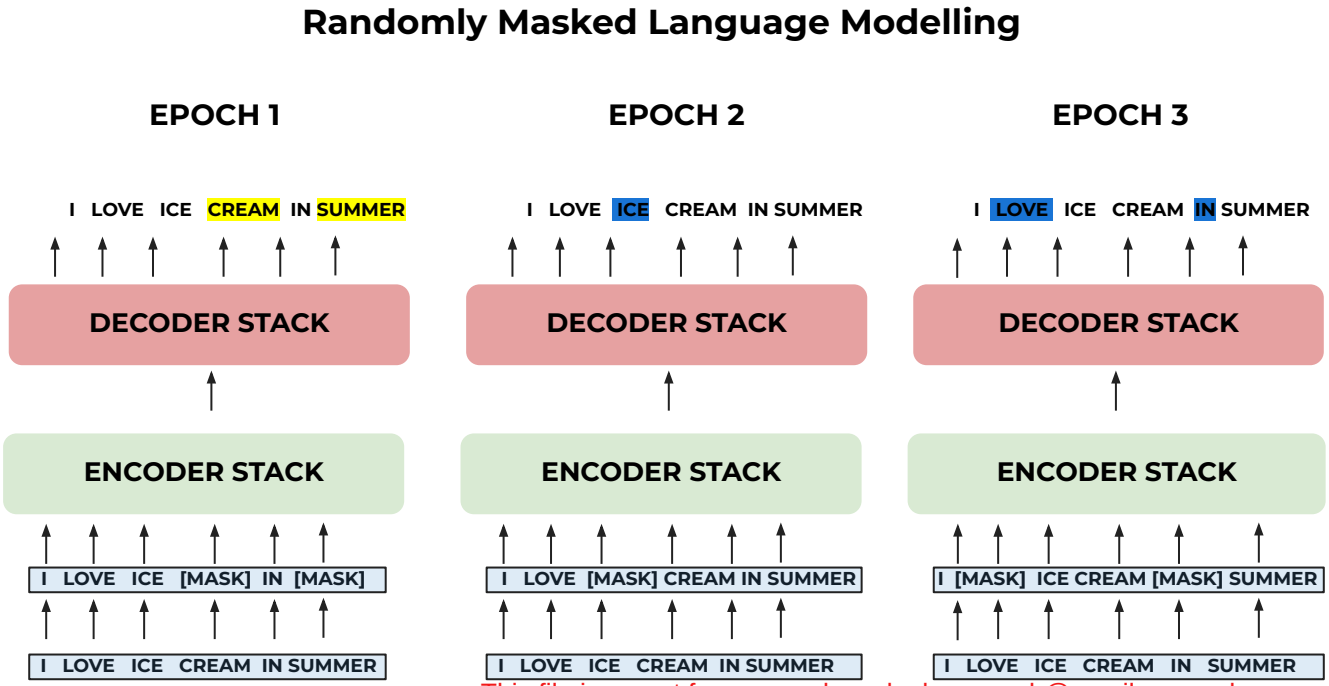
RoBERTa has the exact **same architecture as BERT**, but the Pre-training Approach was changed to **Randomly Masked Language Modeling**, where **the choice of word being masked is randomly changed** in each epoch of training.

In addition, **RoBERTa does not perform fine-tuning using Next Sentence Prediction**, as the authors postulated that the procedure does not yield much difference to the quality of representations.

Despite just these minor tweaks (together with a much bigger training dataset) **RoBERTa was able to significantly outperform BERT on several NLP benchmarks**, and is one of the state-of-the-art NLP models available today.

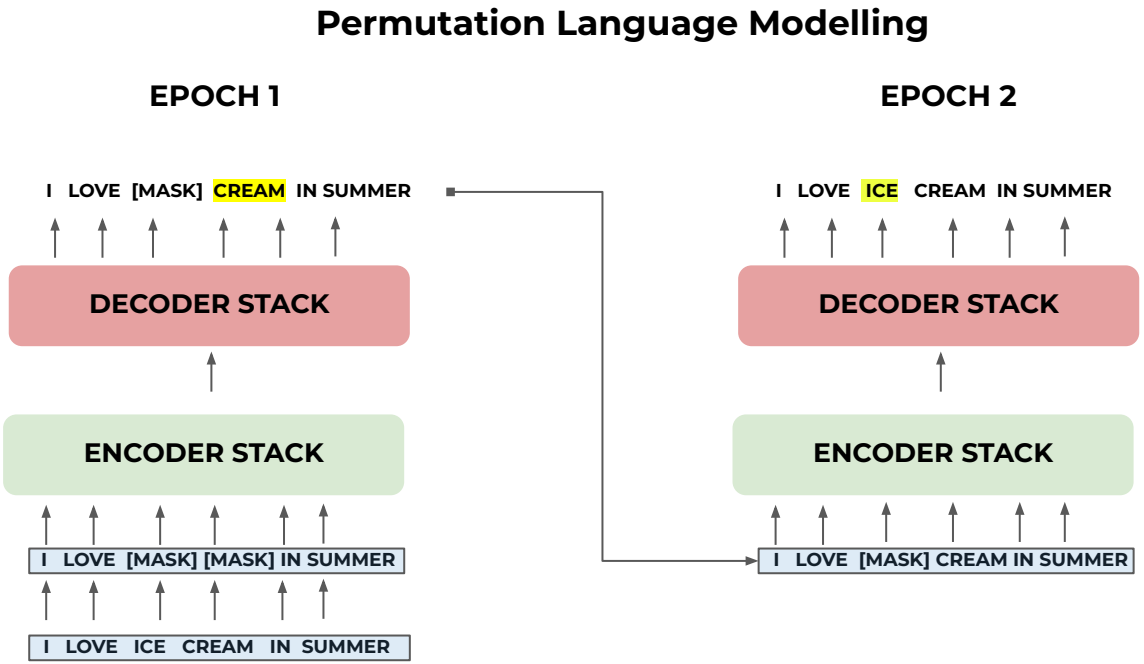
RoBERTa - Training Process

The following is an example of this process for RoBERTa:



XLNet - Training Process

Here's the XLNet training process illustrated with the same example:



XLNet - Another Alternative

XLNet, another variation on BERT which was released around the same time as RoBERTa in 2019, uses a different Pre-training technique to improve on BERT's Masked Language Modeling.

XLNet again uses the exact **same architecture as BERT**. But the XLNet authors believed that predicting all the masked tokens with the Decoder in one go **neglects the sequential dependencies between tokens** that exist in Natural Language.

So XLNet proposes a different idea - **Permutation Language Modeling**.

Rather than predicting both "Ice" and "Cream" in the same run, for instance, the model predicts **one masked token in one run** (let's say "Ice"), then uses this to predict "Cream" **in the next run**.

And just like with RoBERTa, this minor tweak also leads to a **substantial improvement over BERT**, often by a large margin, on several NLP tasks.

Great Learning

Happy Learning !