AdTrans project, 2023-1-PL01-KA220-HED-000158917

ADTRANS
ADVANCED DATA ANALYSIS FOR TRANSPORTATION AND MOBILITY ACADEMIC TEACHERS

Session 2

Large Language Models Basics

Paraskevi Fasouli

Co-funded by
the European Union

Co-funded by the European Union. Views and opinions expressed are however those of the author or authors only and do not necessarily reflect those of the European Union or the Foundation for the Development of the Education System. Neither the European Union nor the entity providing the grant can be held responsible for them.

Content

1. Intro to LLM Characteristics
2. Transformer Architecture
3. Data Collection & Preprocessing
4. Feature Engineering

1. Introduction to Large Language Model Characteristics

Importance of LLMs in AI

“Large Language Models are advanced AI systems that use deep learning, particularly transformer architectures, to process, understand, and generate human language. They are trained on vast datasets and can perform a wide range of natural language processing (NLP) tasks.”

Key Role in AI:

LLMs drive breakthroughs in AI by enabling machines to:

- **Understand context** in text-based inputs.
- **Generate human-like responses**, such as in chatbots and virtual assistants.
- **Support decision-making** through text summarization, translation, and information retrieval.
- **Enhance automation** in applications like content creation, sentiment analysis, and more.

Impact:

LLMs are at the forefront of AI applications, transforming industries like healthcare, finance, and transportation by automating complex language-based tasks.

Famous Examples of Large Language Models

GPT-4 (OpenAI):

- Known for natural language generation, used in tasks like content creation, summarization, and conversational AI.

BERT (Google):

- Excels at understanding context in text, particularly in search queries and natural language understanding.

T5 (Text-to-Text Transfer Transformer by Google):

- Versatile model that frames all NLP tasks as text generation, making it adaptable across a wide range of applications.

LLaMA (Meta):

- Designed for high performance and resource efficiency in AI research and applications, advancing NLP research.

Key Characteristics

- **Scale & Data**: Trained on massive datasets with billions of parameters, allowing them to understand and generate human-like text.
- **Deep Learning Architecture**: Built using advanced neural network architectures, primarily **transformers**, which excel at processing sequences of text and maintaining context.
- **Contextual Understanding**: Capable of understanding nuanced context across long pieces of text, making them effective for tasks like summarization, translation, and conversation.
- **Generative Abilities**: Able to generate coherent and contextually relevant text, used in applications like chatbots, content creation, and text completion.
- **Multi-Task Learning**: Versatile across various natural language processing (NLP) tasks, including text classification, question answering, and language translation.

Key Components of LLMs

1. Neural Network Architecture: Transformers

- LLMs are primarily built on transformer architectures, which are deep-learning models designed to handle sequential data like text. Transformers use layers of self-attention mechanisms to process input data simultaneously, rather than sequentially as in older models like RNNs (Recurrent Neural Networks) or LSTMs (Long Short-Term Memory Networks).

Key Components of LLMs

2. Attention Mechanism:

- **Self-Attention**: This is a key component of transformers that allows the model to weigh the importance of different words in a sentence relative to each other. It helps the model focus on the most relevant parts of the input when making predictions or generating text.
- **Multi-Head Attention**: Instead of computing a single attention score, transformers use multiple attention heads that each focus on different parts of the sentence. This allows the model to capture various relationships between words simultaneously.

Key Components of LLMs

3. Tokenization:

- **Subword Tokenization**: Text is broken down into smaller units called tokens, which can be entire words, subwords, or even characters. This allows LLMs to handle large vocabularies efficiently and manage rare or out-of-vocabulary words.

Key Components of LLMs

4. Positional Encoding:

- **Positional Information**: Since transformers process all tokens simultaneously, positional encoding is added to the tokens to give the model information about the order of words in the sentence, which is crucial for understanding context.

Key Components of LLMs

5. Pre-training & Fine-tuning:

- **Pre-training**: LLMs are first trained on large, diverse datasets to learn general language patterns. This training is unsupervised, meaning the model learns by predicting missing or future words in the text.
- **Fine-tuning**: After pre-training, LLMs are fine-tuned on smaller, task-specific datasets. This process tailors the model to perform specific tasks like translation, summarization, or sentiment analysis.

Key Components of LLMs

6. Layer Normalization & Residual Connections:

- **Layer Normalization**: This technique is used to stabilize and speed up training by normalizing the output of each layer.
- **Residual Connections**: These are shortcuts that skip one or more layers, helping to prevent issues like vanishing gradients and allowing for deeper networks.

Key Components of LLMs

7. Inference Mechanism:

- **Text Generation**: During inference, LLMs generate text by predicting the next token in a sequence based on the context provided by the previous tokens. This process continues until the model produces the desired output.

What are Transformers?

A transformer model is a neural network that learns the context of sequential data and generates new data out of it. To put it simply:

“A transformer is a type of artificial intelligence model that learns to understand and generate human-like text by analyzing patterns in large amounts of text data.”

Transformers are a current state-of-the-art NLP model and are considered the evolution of the encoder-decoder architecture. However, while the encoder-decoder architecture relies mainly on Recurrent Neural Networks (RNNs) to extract sequential information, Transformers completely lack this recurrency.

2. Transformer Architecture

Transformer Architecture

Transformers were first developed to solve the problem of sequence transduction, or neural machine translation, which means they are meant to solve any task that transforms an input sequence into an output sequence. This is why they are called "Transformers".

They are specifically designed to comprehend context and meaning by analyzing the relationship between different elements, and they rely almost entirely on a mathematical technique called attention to do so.

Transformer Architecture

- The transformer's ability to process data in parallel and focus on important parts of the input has led to significant advances in NLP.
- The **transformer architecture** is the backbone of most modern **large language models** (LLMs) and has revolutionized natural language processing (NLP).

Here's how it works:

Self-attention allows the model to focus on important words.

Positional encoding maintains word order and structure.

Multi-head attention processes multiple aspects of the input.

Feed-forward layers refine the model's understanding.

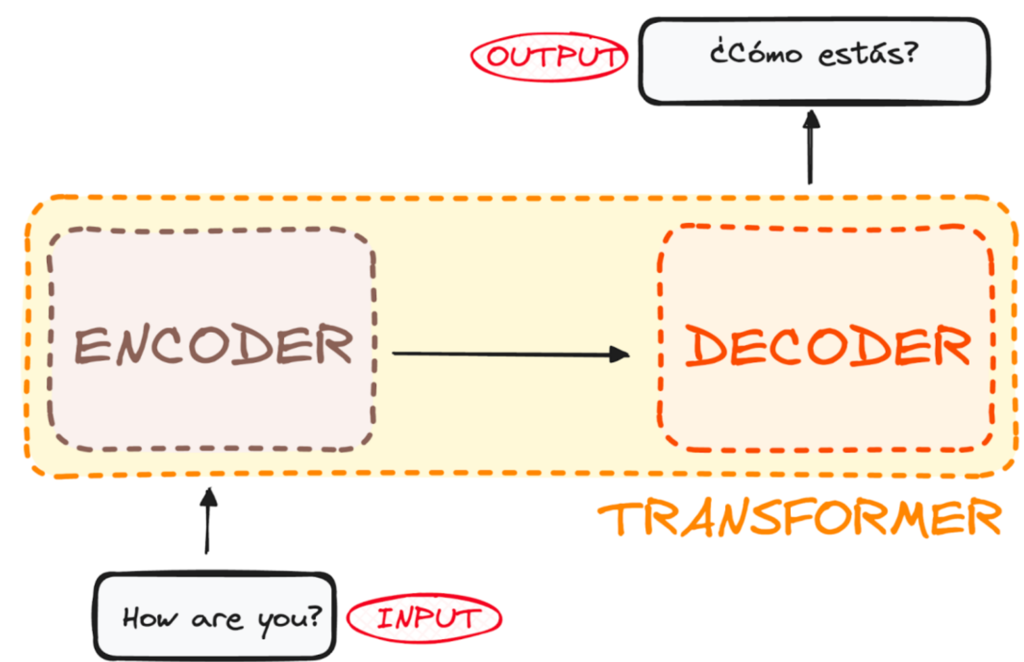
Parallel processing makes transformers faster and more scalable.

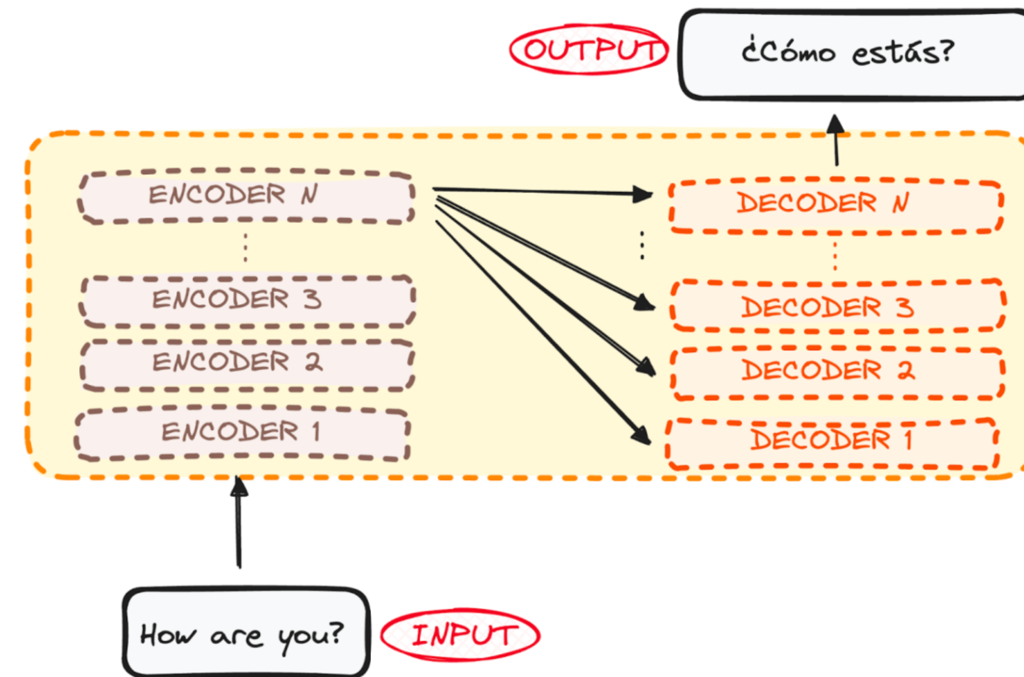
Transformer Architecture

- Transformers excel in converting input sequences into output sequences. It is the first machine translation model relying entirely on self-attention to compute representations of its input and output.
- The main core characteristic of the Transformers architecture is that they maintain the **encoder-decoder** model.
- If we start considering a Transformer for language translation as a simple black box, it takes a sentence in one language, English, for instance, as an input and outputs its translation in English.

Transformer Architecture

- We observe that this black box is composed of two main parts:
  - The **encoder** takes in our input and outputs a matrix representation of that input. For instance, the English sentence "How are you?"
  - The **decoder** takes in that encoded representation and iteratively generates an output. In our example, the translated sentence "¿Cómo estás?"
- However, both the encoder and the decoder are a stack with multiple layers (the same number for each). All encoders present the same structure, and the input gets into each of them and is passed to the next one. All decoders also present the same structure and get the input from the last encoder and the previous decoder.
- The original architecture consisted of 6 encoders and 6 decoders, but we can replicate as many layers as we want. So, let's assume N layers of each.

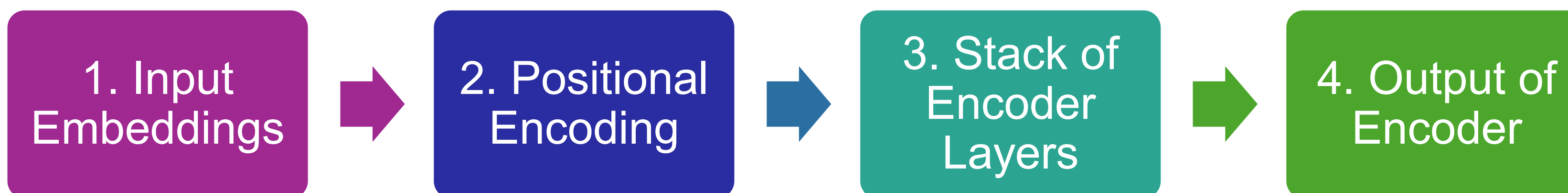


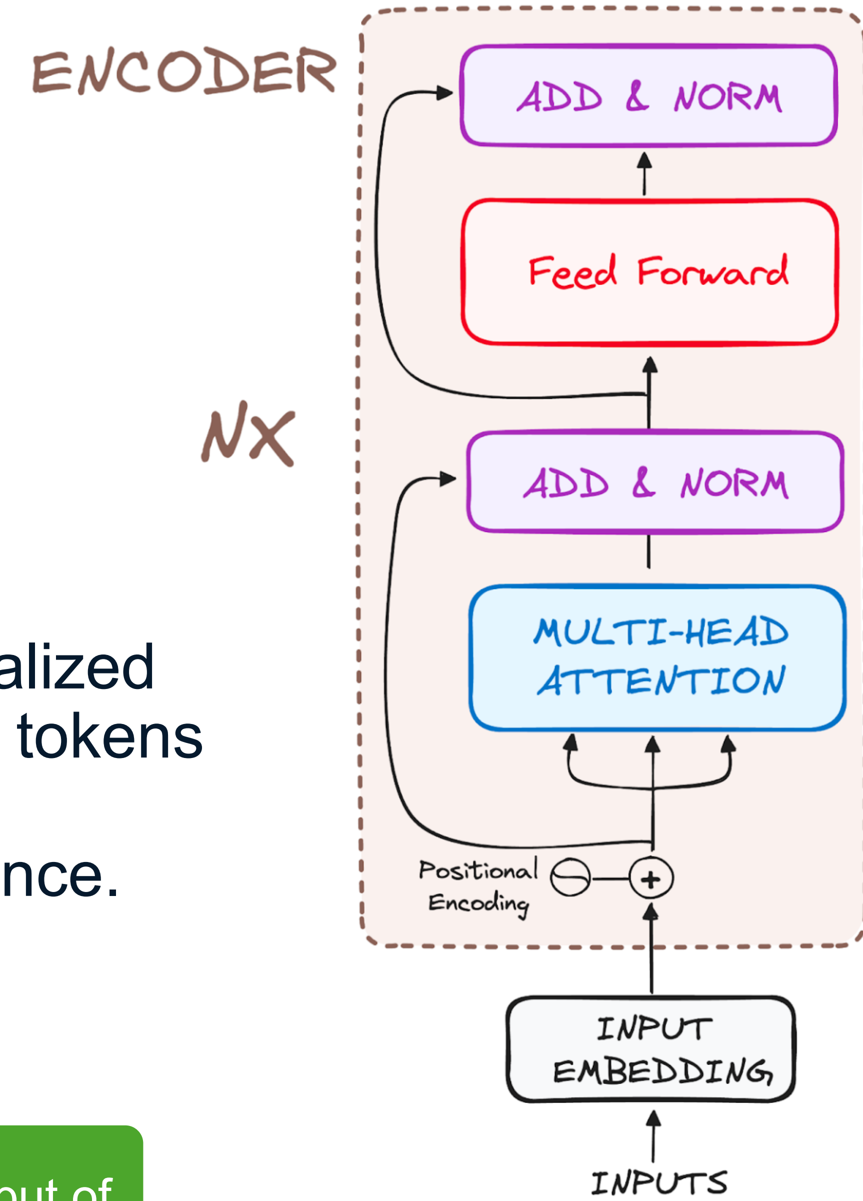


The Encoder Workflow

The encoder is a fundamental component of the Transformer architecture. The primary function of the encoder is to transform the input tokens into contextualized representations. Unlike earlier models that processed tokens independently, the Transformer encoder captures the context of each token with respect to the entire sequence.

The encoder workflow consists of 4 steps:

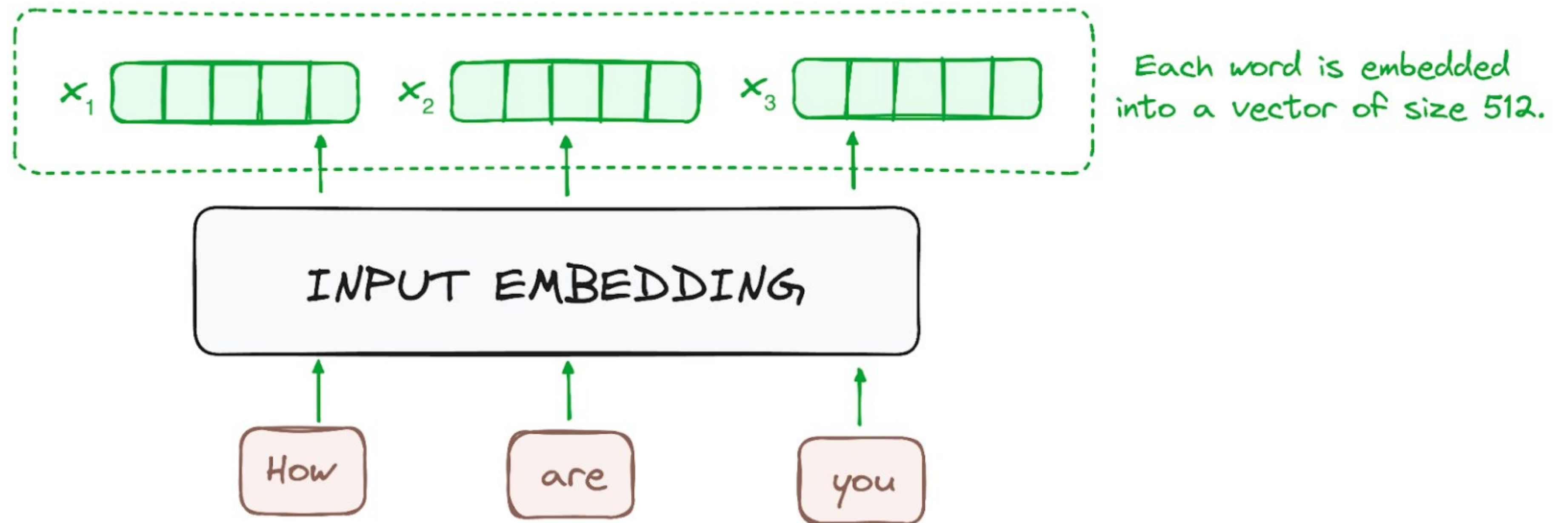




1. Input Embeddings

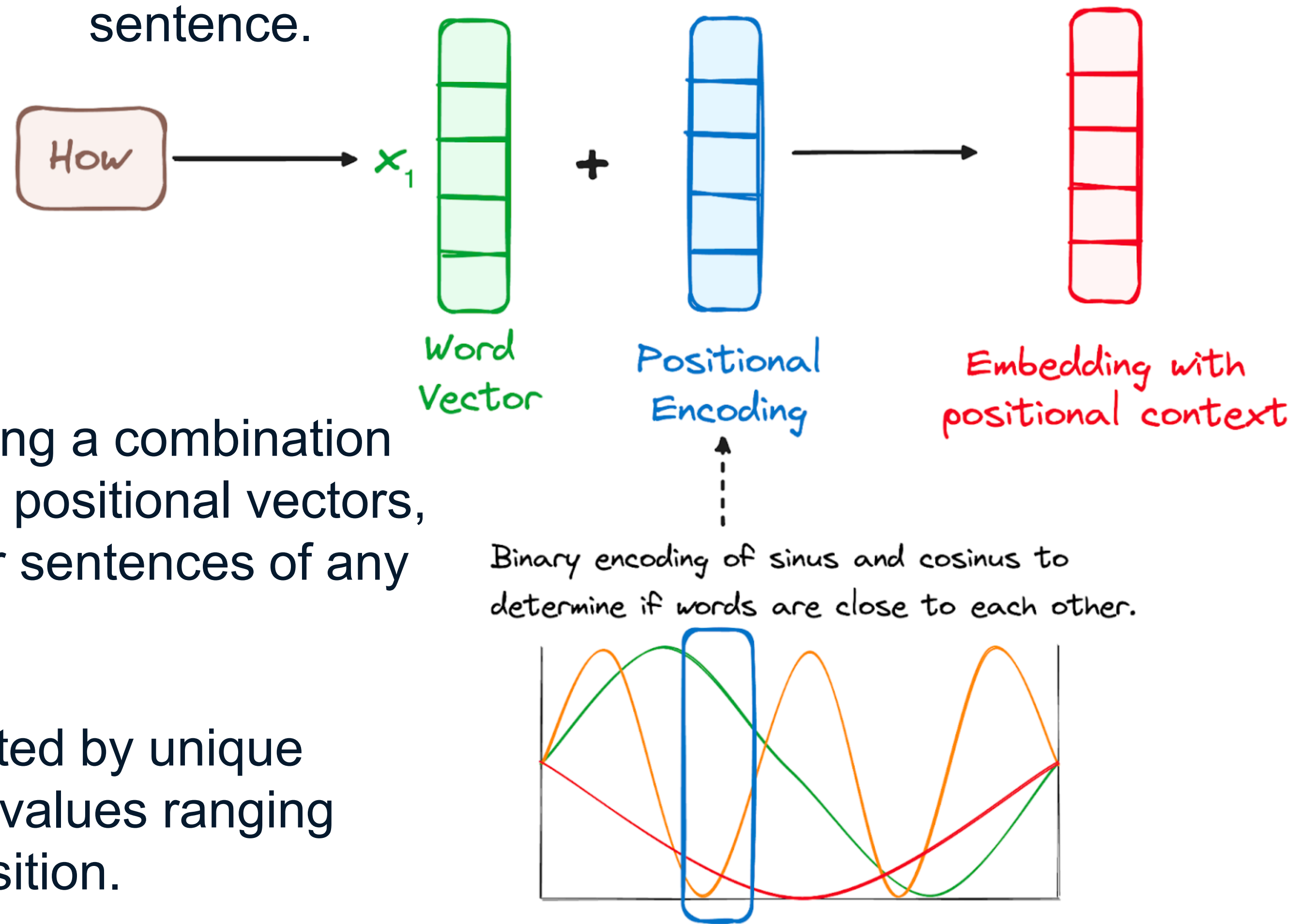
The embedding only happens in the bottom-most encoder. The encoder begins by converting input tokens - words or subwords - into vectors using embedding layers. These embeddings capture the semantic meaning of the tokens and convert them into numerical vectors.

All the encoders receive a list of vectors, each of size 512 (fixed-sized). In the bottom encoder, that would be the word embeddings, but in other encoders, it would be the output of the encoder that's directly below them.



2. Positional Encoding

Since Transformers do not have a recurrence mechanism like RNNs, they use positional encodings added to the input embeddings to provide information about the position of each token in the sequence. This allows them to understand the position of each word within the sentence.



To do so, the researchers suggested employing a combination of various sine and cosine functions to create positional vectors, enabling the use of this positional encoder for sentences of any length.

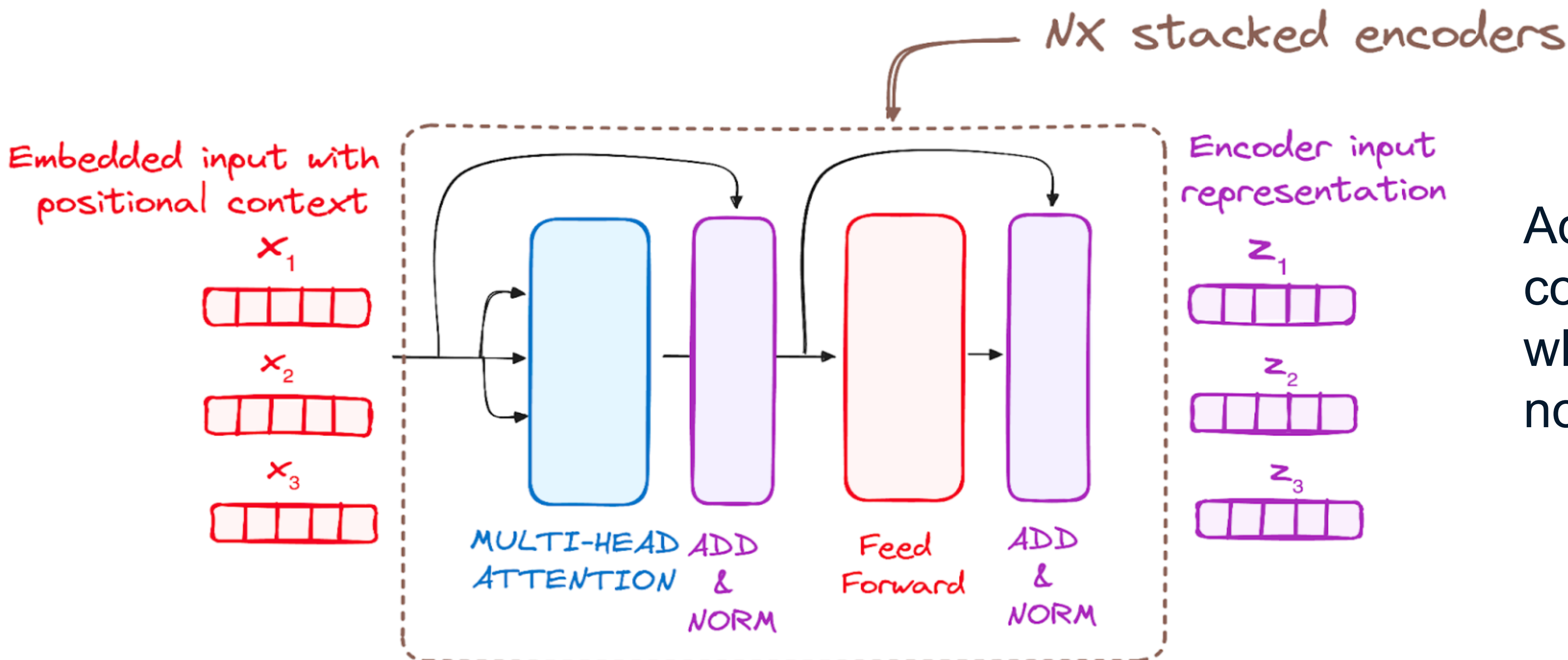
In this approach, each dimension is represented by unique frequencies and offsets of the wave, with the values ranging from -1 to 1, effectively representing each position.

3. Stack of Encoder Layers

The Transformer encoder consists of a stack of identical layers (6 in the original Transformer model).

The encoder layer serves to transform all input sequences into a continuous, abstract representation that encapsulates the learned information from the entire sequence. This layer comprises two sub-modules:

1. A multi-headed attention mechanism.
2. A fully connected network.



Additionally, it incorporates residual connections around each sublayer, which are then followed by layer normalization.

3.1 Multi-head Attention

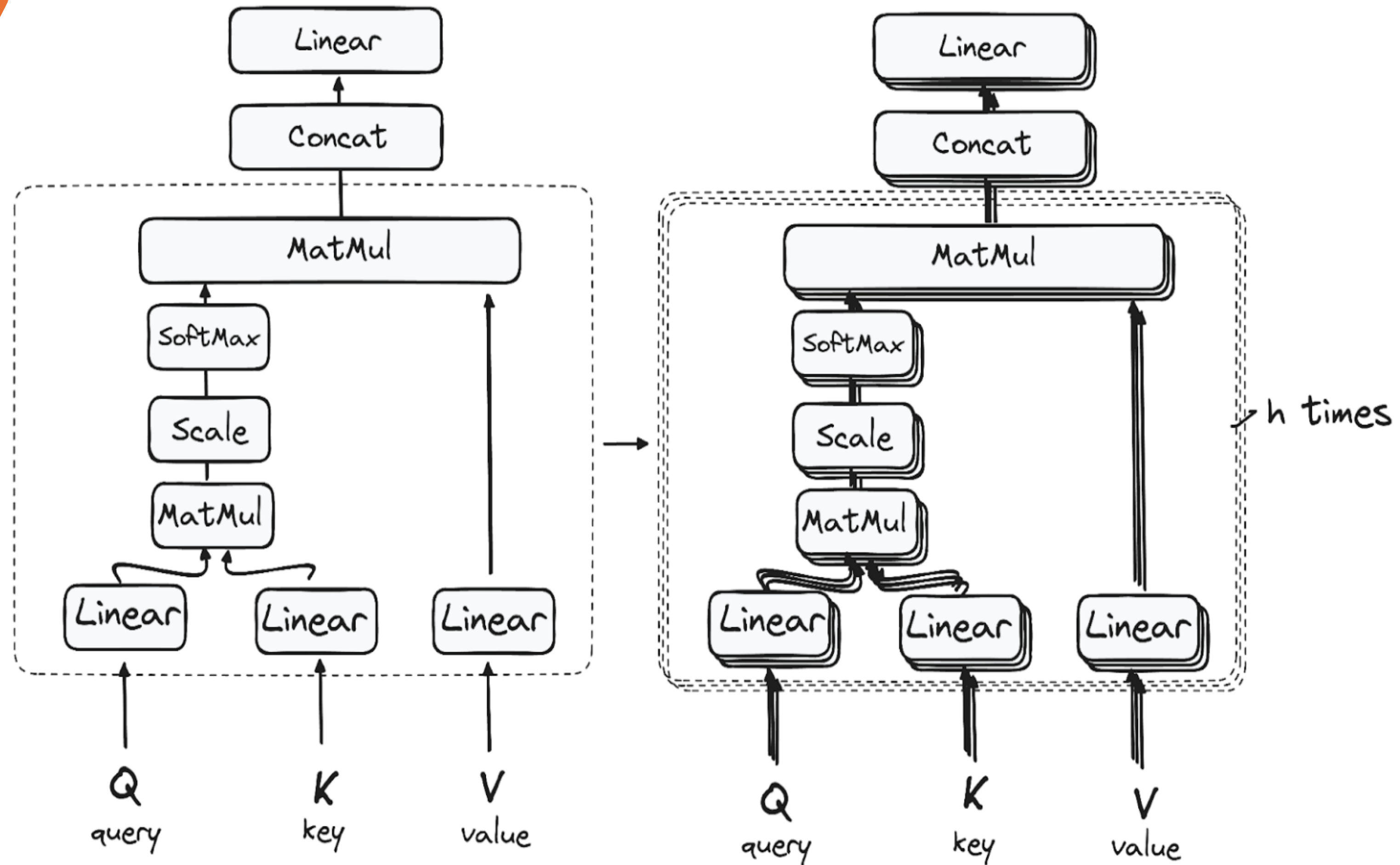
In the encoder, the multi-headed attention utilizes a specialized attention mechanism known as self-attention. This approach enables the models to relate each word in the input with other words. For instance, in a given example, the model might learn to connect the word “are” with “you”.

This mechanism allows the encoder to focus on different parts of the input sequence as it processes each token. It computes attention scores based on:

- A **query** is a vector that represents a specific word or token from the input sequence in the attention mechanism.
- A **key** is also a vector in the attention mechanism, corresponding to each word or token in the input sequence.
- Each **value** is associated with a key and is used to construct the output of the attention layer. When a query and a key match well, which basically means that they have a high attention score, the corresponding value is emphasized in the output.

This first Self-Attention module enables the model to capture contextual information from the entire sequence. Instead of performing a single attention function, queries, keys and values are linearly projected h times. On each of these projected versions of queries, keys and values the attention mechanism is performed in parallel, yielding h-dimensional output values.

3.1 Multi-head Attention

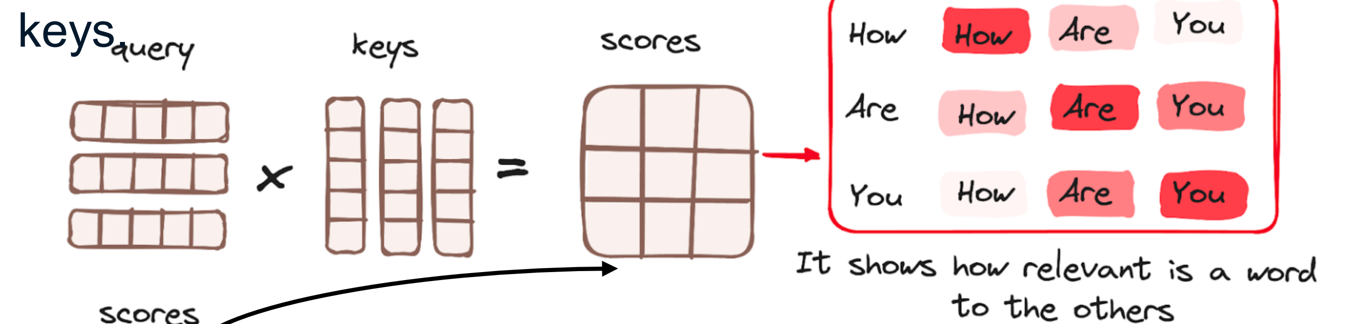


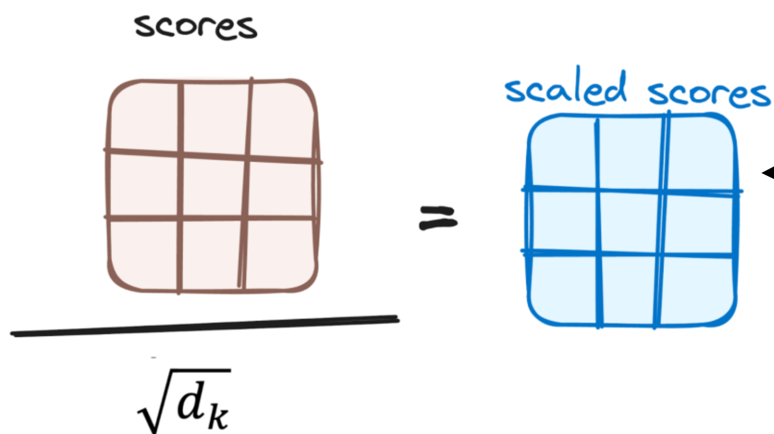
3.1 Multi-head Attention: *MatMul & Reducing Magnitude of Attention Scores*

Once the query, key, and value vectors are passed through a linear layer, a dot product matrix multiplication is performed between the queries and keys, resulting in the creation of a score matrix.

The score matrix establishes the degree of emphasis each word should place on other words. Therefore, each word is assigned a score in relation to other words within the same time step. A higher score indicates greater focus.

This process effectively maps the queries to their corresponding keys.





The scores are then scaled down by dividing them by the square root of the dimension of the query and key vectors. This step is implemented to ensure more stable gradients, as the multiplication of values can lead to excessively large effects.

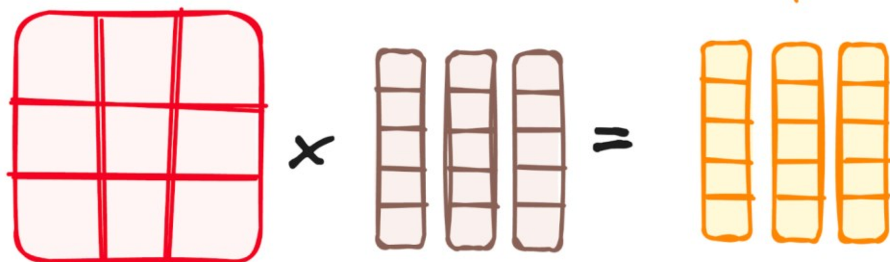
3.1 Multi-head Attention: *Applying Softmax & Making Output*

Subsequently, a softmax function is applied to the adjusted scores to obtain the attention weights. This results in probability values ranging from 0 to 1.

The softmax function emphasizes higher scores while diminishing lower scores, thereby enhancing the model's ability to determine which words should receive more attention effectively.

Higher scores get heighten, and lower scores are depressed

Softmax( ) =



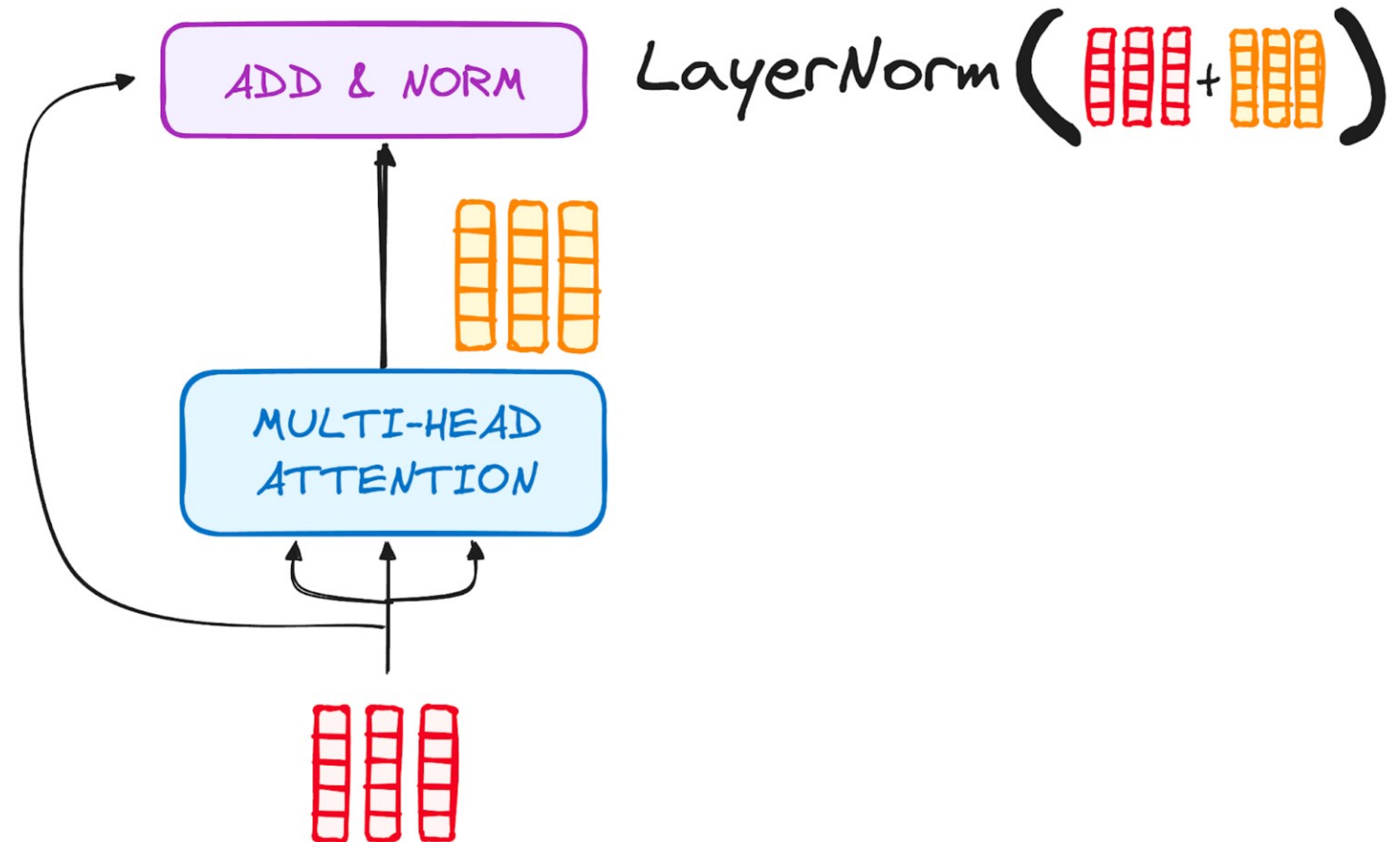
The following step of the attention mechanism is that weights derived from the softmax function are multiplied by the value vector, resulting in an output vector.

In this process, only the words that present high softmax scores are preserved. Finally, this output vector is fed into a linear layer for further processing.

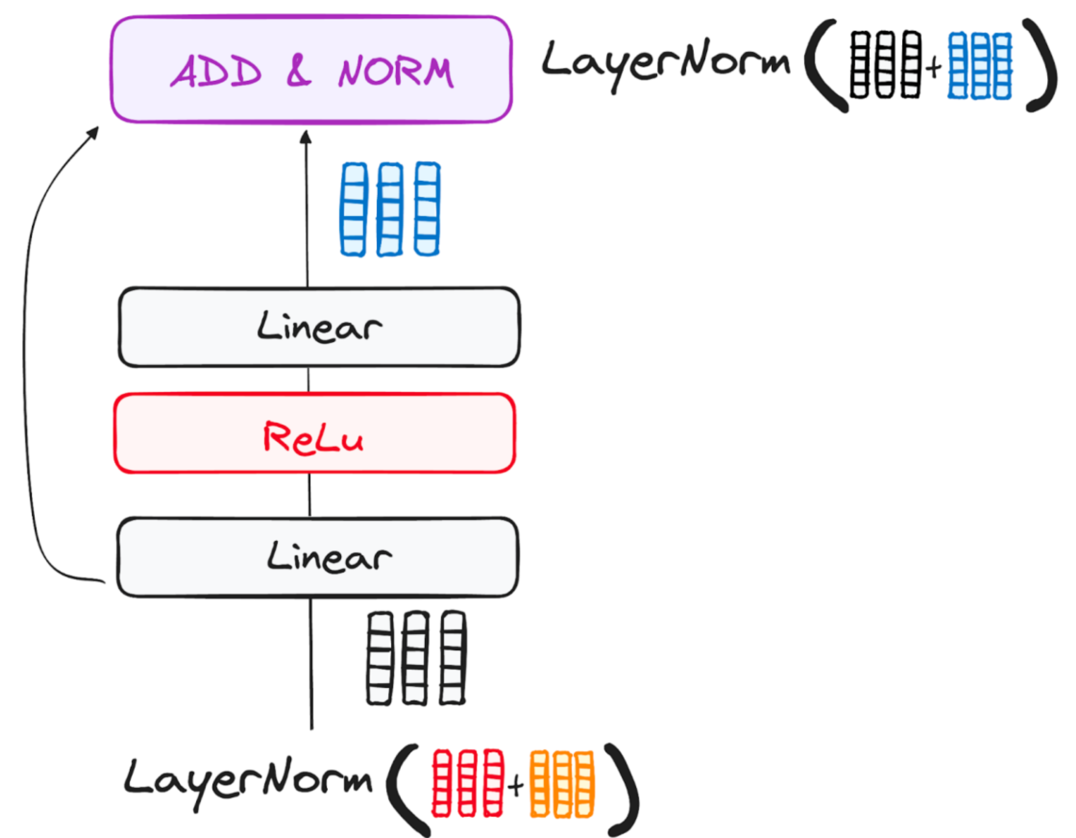
Finally, we get the output of the Attention mechanism!

3.2 Normalization & Residual Connections

Each sub-layer in an encoder layer is followed by a normalization step. Also, each sub-layer output is added to its input (residual connection) to help mitigate the vanishing gradient problem, allowing deeper models. This process will be repeated after the Feed-Forward Neural Network too.



3.3 Feed-Forward Neural Network



The journey of the normalized residual output continues as it navigates through a pointwise feed-forward network, a crucial phase for additional refinement.

Picture this network as a duo of linear layers, with a ReLU activation nestled in between them, acting as a bridge. Once processed, the output embarks on a familiar path: it loops back and merges with the input of the pointwise feed-forward network.

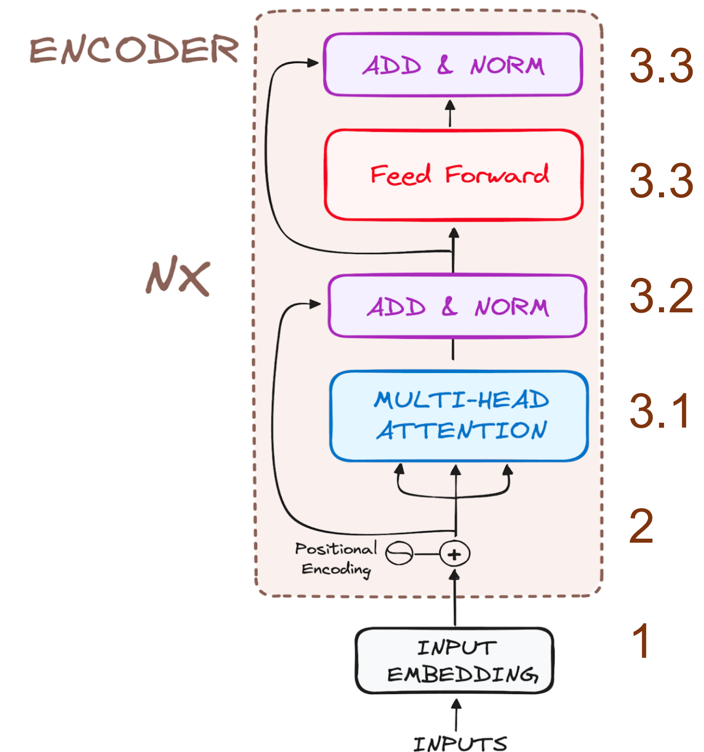
This reunion is followed by another round of normalization, ensuring everything is well-adjusted and in sync for the next steps.

4. Output of the Encoder

The output of the final encoder layer is a set of vectors, each representing the input sequence with a rich contextual understanding. This output is then used as the input for the decoder in a Transformer model.

This careful encoding paves the way for the decoder, guiding it to pay attention to the right words in the input when it's time to decode.

Think of it as building a tower where you can stack up N encoder layers. Each layer in this stack gets a chance to explore and learn different facets of attention, much like layers of knowledge. This not only diversifies the understanding but could significantly amplify the predictive capabilities of the transformer network.



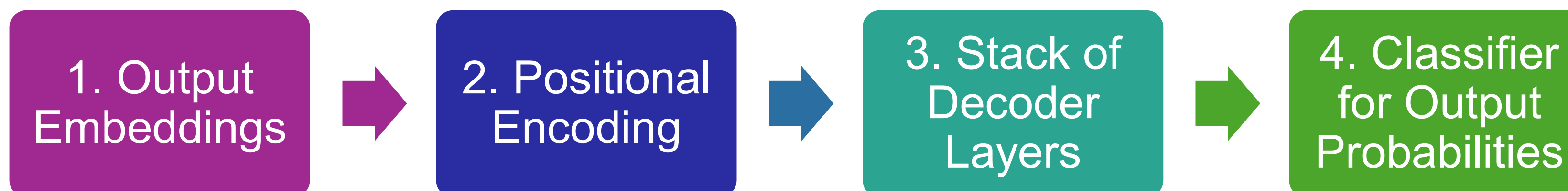
The Decoder Workflow

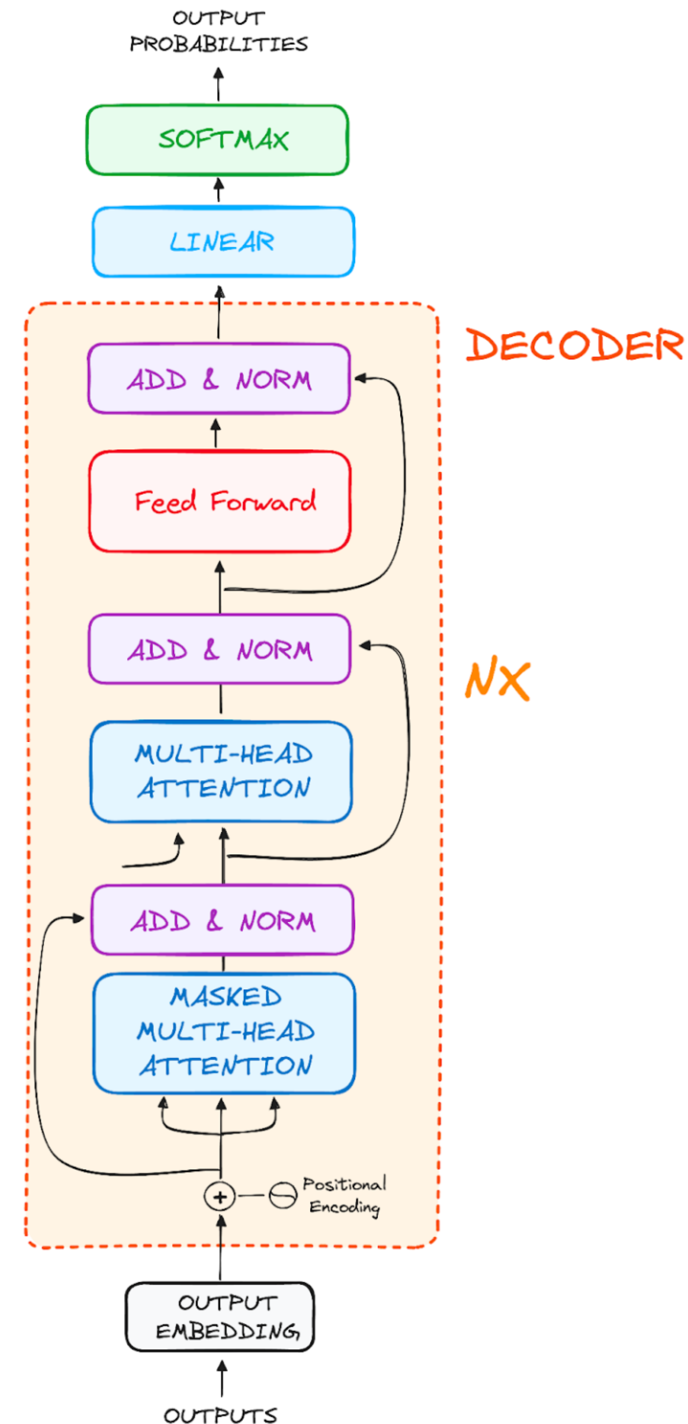
The decoder's role centers on crafting text sequences. Mirroring the encoder, the decoder is equipped with a similar set of sub-layers. It boasts two multi-headed attention layers, a pointwise feed-forward layer, and incorporates both residual connections and layer normalization after each sub-layer.

These components function in a way akin to the encoder's layers, yet with a twist: each multi-headed attention layer in the decoder has its unique mission.

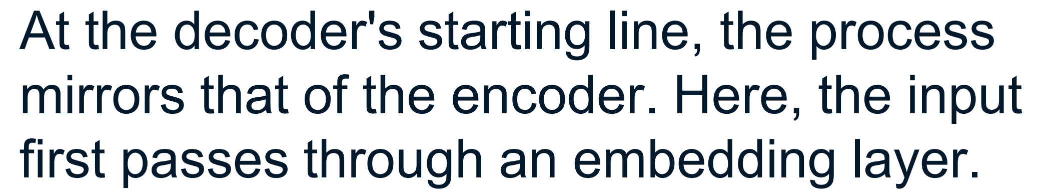
The final of the decoder's process involves a linear layer, serving as a classifier, topped off with a softmax function to calculate the probabilities of different words.

The decoder workflow also consists of 4 steps:

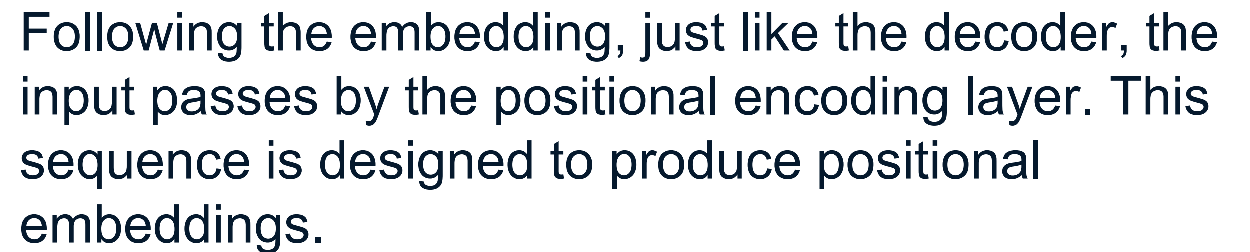




1. Output
Embeddings


At the decoder's starting line, the process
mirrors that of the encoder. Here, the input
first passes through an embedding layer.

2. Positional Encoding

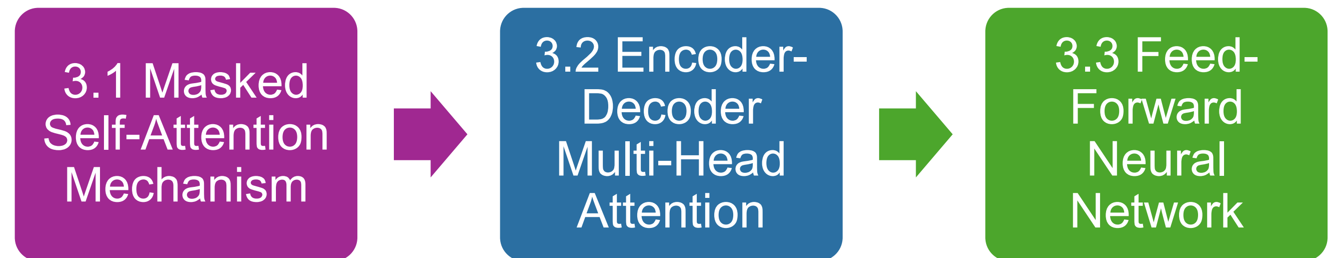


Following the embedding, just like the decoder, the input passes by the positional encoding layer. This sequence is designed to produce positional embeddings.

These positional embeddings are then channeled into the first multi-head attention layer of the decoder, where the attention scores specific to the decoder's input are meticulously computed.

3. Stack of Decoder Layers

The decoder consists of a stack of identical layers (6 in the original Transformer model). Each layer has three main sub-components:

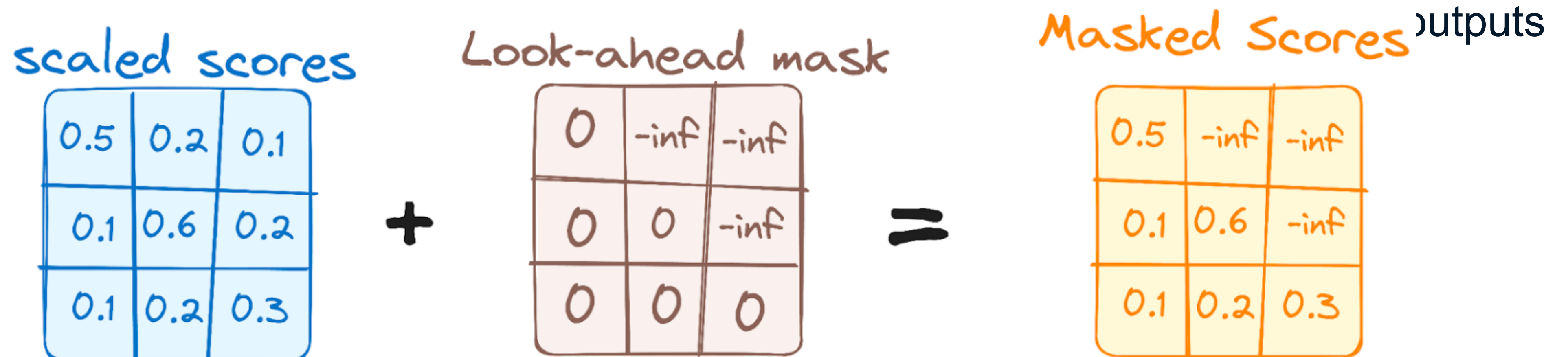


3.1 Masked Self-Attention Mechanism

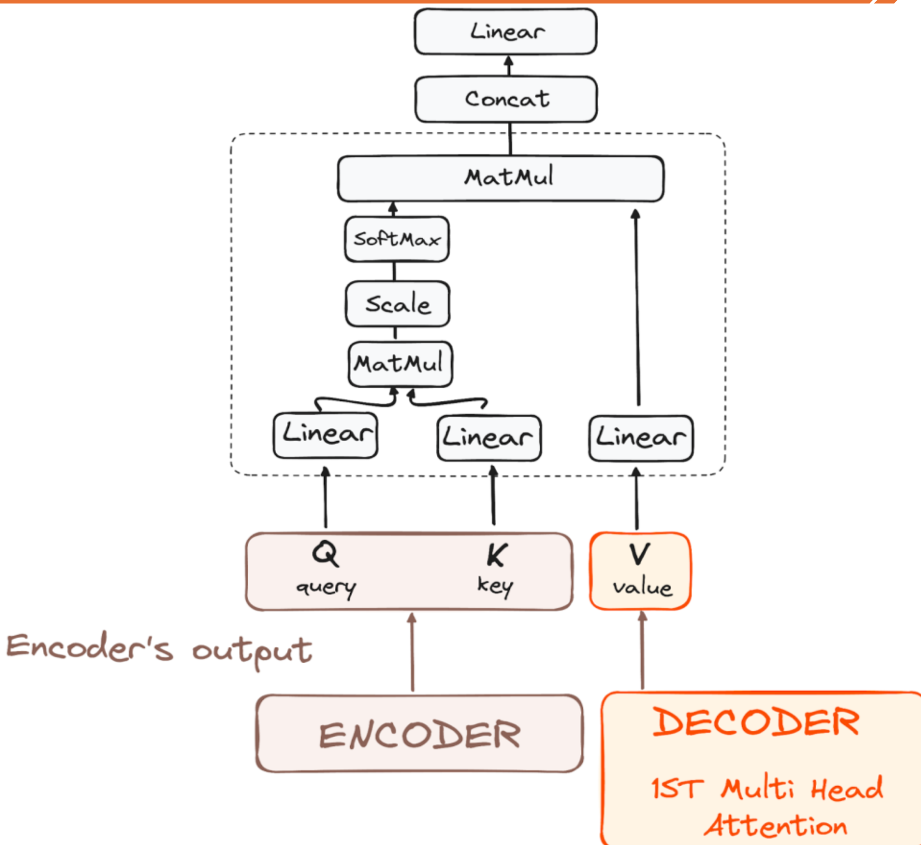
This is like the self-attention mechanism in the encoder but with a crucial difference: it prevents positions from attending to subsequent positions, which means that each word in the sequence isn't influenced by future tokens.

For instance, when the attention scores for the word "are" are being computed, it's important that "are" doesn't get a peek at "you", which is a subsequent word in the sequence.

This masking ensures that the predictions for a
utputs



3.2 Masked Self-Attention Mechanism

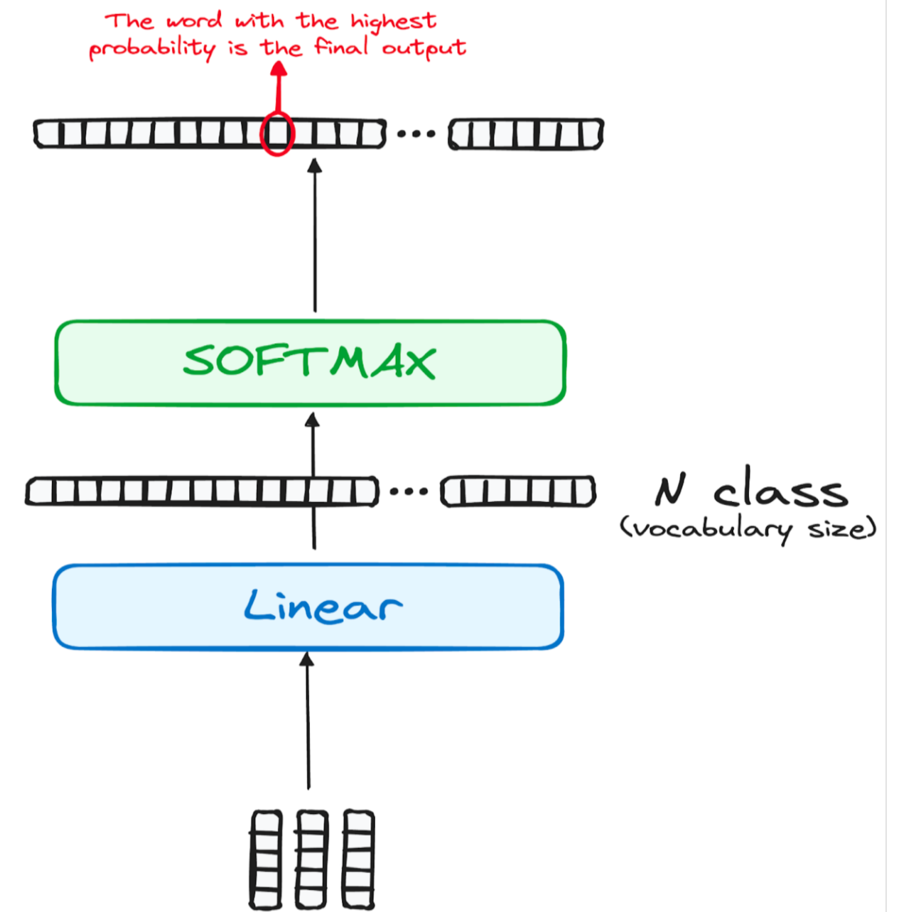


In the second multi-headed attention layer of the decoder, we see a unique interplay between the encoder and the decoder's components.
Here, the outputs from the encoder take on the roles of both queries and keys, while the outputs from the first multi-headed attention layer of the decoder serve as values.

This setup effectively aligns the encoder's input with the decoder's, empowering the decoder to identify and emphasize the most relevant parts of the encoder's input.
Following this, the output from this second multi-headed attention layer is refined through a pointwise feedforward layer, enhancing the processing further.

In this sub-layer, the queries come from the previous decoder layer, and the keys and values come from the output of the encoder. This allows every position in the decoder to attend overall positions in the input sequence, effectively integrating information from the

4. Output of the Decoder



The journey of data through the transformer model culminates in its passage through a final linear layer, which functions as a classifier. The size of this classifier corresponds to the total number of classes involved (number of words contained in the vocabulary). For instance, in a scenario with 1000 distinct classes representing 1000 different words, the classifier's output will be an array with 1000 elements.

This output is then introduced to a softmax layer, which transforms it into a range of probability scores, each lying between 0 and 1. The highest of these probability scores is key, its corresponding index directly points to the word that the model predicts as the next in the sequence.

4. Output of the Decoder

Normalization and Residual Connections

Each sub-layer (masked self-attention, encoder-decoder attention, feed-forward network) is followed by a normalization step, and each also includes a residual connection around it.

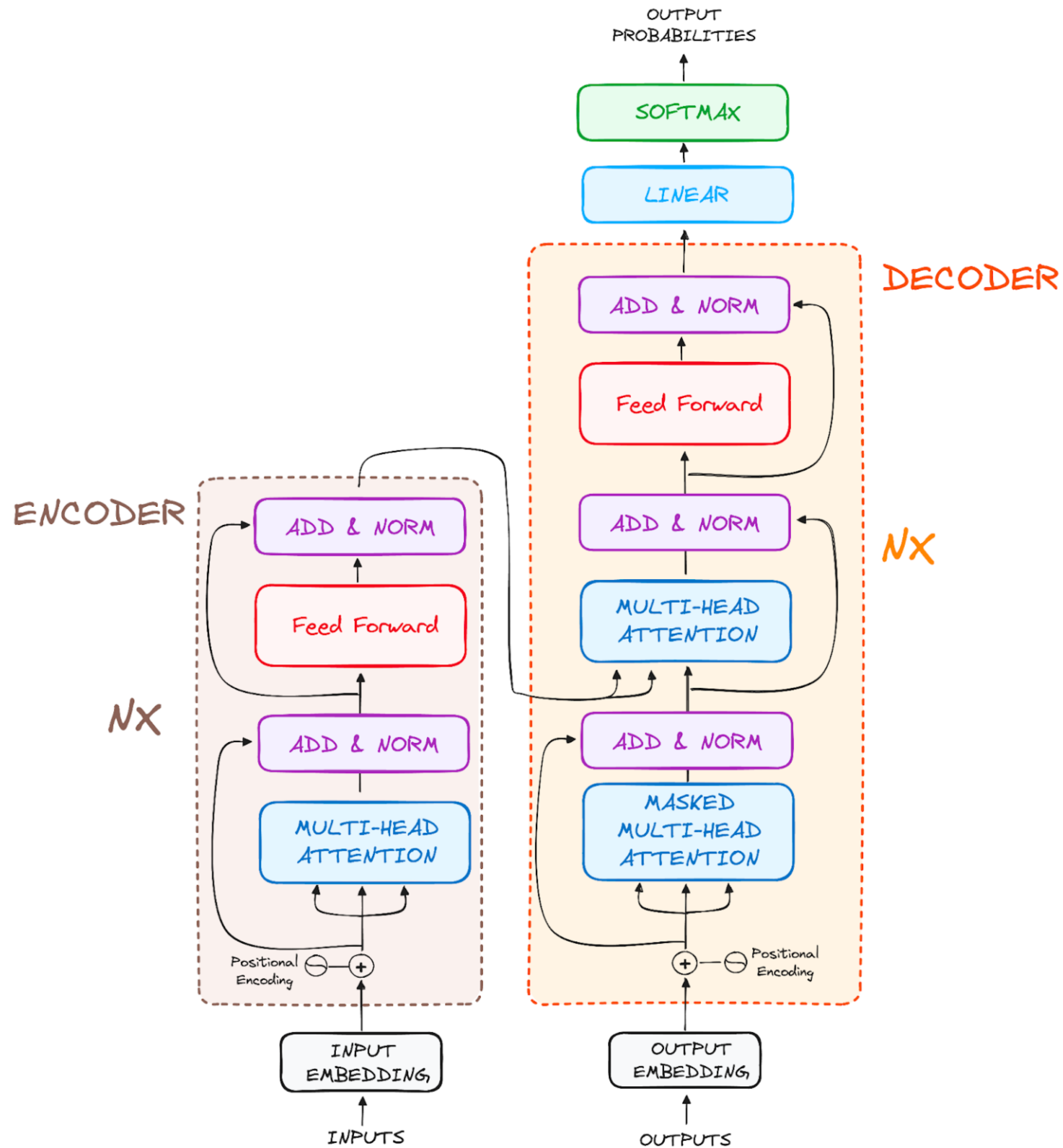
Output of the Decoder

The final layer's output is transformed into a predicted sequence, typically through a linear layer followed by a softmax to generate probabilities over the vocabulary.

The decoder, in its operational flow, incorporates the freshly generated output into its growing list of inputs and then proceeds with the decoding process. This cycle repeats until the model predicts a specific token, signaling completion.
The token predicted with the highest probability is assigned as the concluding class, often represented by the end token.

The decoder isn't limited to a single layer. It can be structured with N layers, each one building upon the input received from the encoder and its preceding layers. This layered architecture allows the model to diversify its focus and extract varying attention patterns across its attention heads. Such a multi-layered approach can significantly enhance the model's ability to predict as it develops a more nuanced understanding of different

Final Architecture



Comparison with Traditional Models

LLMs:

- **Architecture**: Typically based on deep learning techniques, especially transformer models, which handle complex language patterns.
- **Training**: Pre-trained on massive, diverse text datasets and then fine-tuned for specific tasks, allowing them to capture extensive language nuances and knowledge.
- **Capabilities**: Excels in understanding context, generating coherent and contextually relevant text, and handling a wide range of language tasks with minimal task-specific training.
- **Scalability**: Requires significant computational resources and large datasets for training but can generalize well across various domains once trained.

Traditional Models:

- **Architecture**: Often based on simpler techniques such as rule-based systems, statistical models, or earlier machine learning methods like bag-of-words or n-grams.
- **Training**: Typically trained on smaller, domain-specific datasets with a focus on predefined tasks, often requiring manual feature engineering.
- **Capabilities**: Limited in handling complex language understanding and generation compared to LLMs; performance can vary significantly based on task-specific design and feature selection.
- **Scalability**: Generally, requires less computational power and data for training but may not generalize as well across diverse tasks or domains.

Comparison with Traditional Models

LLMs leverage advanced neural architectures and extensive training data to handle a broad range of language tasks with high flexibility and context awareness, whereas traditional models often rely on more straightforward techniques and are more limited in scope and adaptability.

3. Data Collection & Preprocessing

Role of Data in Training LLMs

- **Foundation of Learning**: LLMs rely on vast and diverse datasets to learn language patterns, grammar, and context.
- **Data Sources**: Includes text from books, websites, articles, and other written content to provide a wide range of linguistic examples.
- **Quality and Quantity**: The size and diversity of the data determine the model's ability to generalize and perform accurately across different tasks.
- **Impact on Output**: High-quality data leads to coherent and meaningful outputs, while biased or incomplete data can result in incorrect or skewed results.
- **Shaping Capabilities**: Data directly influences the model's knowledge, behavior, and ability to apply language understanding in real-world applications.

Crucial for LLMs' performance & accuracy

Sources of Data Used for LLM Training

Web Scraped Text: Large datasets from websites, forums, and blogs (e.g., Common Crawl) provide diverse and up-to-date language examples.

Books & Literature: Digitized books and academic papers offer well-structured, formal language content across various subjects and genres.

Wikipedia & Encyclopedic Sources: High-quality, factual text that covers a wide range of topics, contributing to general knowledge.

News Articles: Current events and journalistic writing provide real-world context and contemporary language use.

Social Media Content: Data from platforms like Twitter or Reddit captures informal, conversational language and slang, enhancing the model's ability to handle everyday interactions.

Code Repositories: In models like Codex, data from platforms like GitHub helps train the model for programming and code generation tasks.

Importance Of Data In Building LLMs

1. Training Quality:

- **Diverse and Large-Scale Data**: LLMs require vast amounts of diverse text data to learn language patterns, understand context, and capture a wide range of topics. The more diverse the data, the better the model can generalize to different tasks and languages.
- **Representation of Real-World Scenarios**: High-quality, varied data ensures that the LLM can understand and generate language that accurately reflects real-world usage, including slang, idioms, technical jargon, and cultural references.

2. Generalization:

- **Contextual Understanding**: Rich data helps LLMs develop a deep understanding of context, enabling them to generate relevant and coherent responses across different scenarios.
- **Transfer Learning**: Pre-training on large datasets allows LLMs to transfer their knowledge to specific tasks (via fine-tuning) with less additional data, improving performance on those tasks.

Importance Of Data In Building LLMs

3. Bias and Fairness:

- **Mitigating Bias**: The quality and diversity of the training data impact the model's ability to avoid or minimize biases. Well-curated data helps in reducing the risk of perpetuating harmful stereotypes or biases.
- **Fair Representation**: Including data from various demographic, geographic, and linguistic sources ensures that the model performs well across different groups and languages, making it more inclusive.

4. Performance and Accuracy:

- **Data Quality**: High-quality, clean data improves the accuracy and reliability of the model's predictions. Noisy or biased data can lead to poor performance and unreliable outputs.
- **Task-Specific Data**: During fine-tuning, task-specific data is used to adapt the model to particular applications, enhancing its effectiveness for those tasks.

1. Data Cleaning

1. Deduplication

- Removing duplicate entries to ensure the dataset has unique, diverse content.
- **Example**: Eliminating identical paragraphs or sentences that appear multiple times across different documents.

2. Removing Noisy Data

- Filtering out irrelevant, incomplete, or erroneous data.
- **Example**: Removing HTML tags, URLs, broken sentences, or text with excessive special characters.

3. Language Filtering

- Ensuring that the dataset is consistent in language, especially for monolingual models.
- **Example**: Identifying and removing text in unintended languages within a primarily English dataset.

4. Stopwords & Rare Words Han...

- Deciding whether to remove common stopwords or rare words, depending on the model's needs.
- **Example**: Removing words like "the," "and," "but," or filtering out extremely rare words that may add noise to the model.

Tokenization:

- Break text into smaller units like words or subwords, enabling the model to process language effectively while capturing meaning across different word forms.

Normalization:

- Standardize text by converting to lowercase, removing special characters, and handling accents or punctuation to ensure consistency across the dataset.

Padding and Truncation:

- Adjust sequence lengths by padding shorter texts with special tokens or truncating longer ones to maintain uniform input sizes for the model.

Text Encoding:

- Convert tokenized text into numerical representations using word embeddings or other encoding techniques that can be processed by the model.

Batching:

- Group data into batches to allow efficient training and parallel processing, balancing batch size to optimize memory use and training speed.

Shuffling:

- Randomize the order of data points in each batch to prevent the model from learning sequence patterns that don't generalize well.

3. Data Annotation & Labeling (Optional)

While large-scale, unsupervised data is sufficient for initial LLM training, **annotation and labeling become important during the fine-tuning phase** to adapt the model for specific, task-oriented applications.

Fine-Tuning for Specific Tasks:

- While LLMs are pre-trained on general text, they are often **fine-tuned** using labeled or annotated data for specific tasks like text classification, sentiment analysis, or question answering.

Supervised Learning:

- For task-specific models (e.g., sentiment analysis, named entity recognition), annotated data is crucial to teach the model the correct relationships between inputs and outputs.

Reinforcement Learning with Human Feedback (RLHF):

- In some cases, human-annotated feedback is used to fine-tune LLMs to generate more accurate and human-like responses.

These criteria ensure that the data used to train LLMs is high-quality and conducive to producing **accurate, reliable, & fair models.**

Criteria For Evaluating Data Quality

1. Diversity: The dataset should cover a wide range of topics, domains, and language styles to ensure the model generalizes well across various tasks and contexts.

2. Cleanliness: Data must be free of noise, errors, duplicates, and irrelevant content like broken text, spam, or incorrect formats to prevent the model from learning incorrect patterns.

3. Balance: Ensure a balanced representation of different languages, dialects, and perspectives to avoid bias and ensure fairness in the model's output.

4. Relevance: Data should be contextually relevant and aligned with the intended use cases or applications for which the LLM is being trained or fine-tuned.

5. Freshness: Data should be up-to-date to reflect current language use, trends, and knowledge, especially for time-sensitive tasks like news or customer support.

6. Completeness: Data should provide enough context and depth in each instance (e.g., full sentences or documents) to allow the model to understand and generate coherent language.

4. Feature Engineering

Introduction to Feature Engineering

“Feature engineering is the process of selecting, transforming, and creating new features from raw data to improve a model’s performance. It helps models better understand the underlying patterns and relationships in the data.”

Significance:

- By enhancing the input data, feature engineering helps simplify complex relationships, increase model accuracy, and reduce overfitting, making it critical in traditional machine learning tasks.
- For LLM tasks, it plays a pivotal role in enhancing accuracy and task-specific performance.

Role of Feature Engineering in LLMs

Contextual Feature Creation: In LLMs, creating meaningful text-based features like word embeddings or incorporating additional metadata (e.g., timestamps, document types) can provide richer context, improving the model's understanding.

Handling Text Structure: Feature engineering helps address specific language structures, such as entity recognition or syntactic parsing, which allows the model to grasp nuanced relationships in the text.

Task-Specific Features: During fine-tuning, task-relevant features, like sentiment or named entities, can be engineered to boost model performance on specialized tasks like classification or translation.

Importance of Features

Features in **Large Language Models (LLMs)** are crucial because they help in capturing relevant patterns and relationships in the data, directly impacting the model's ability to understand and generate meaningful text.

By extracting and using the right features, LLMs can better comprehend context, word relationships, and nuances in language, improving overall performance in tasks like translation, text generation, or sentiment analysis.

Good Example:	Bad Example:
• Using **word embeddings** that capture semantic relationships between words is a good feature. • For example, words like "happy" and "joyful" will have similar embeddings, helping the model understand positive sentiment even if different words are used in the text. This enhances the model's ability to classify text as positive or negative accurately.	• Using the **length of a sentence** as a feature for predicting sentiment might be a poor choice. The length of a sentence doesn't directly correlate with sentiment (e.g., a short sentence like "I hate this!" has a strong negative sentiment, while a long sentence might be neutral). This could lead the model to make incorrect predictions based on irrelevant features.

Methods for Creating Features

1. Tokenization: Breaking down text into smaller units such as words, subwords, or characters, which serve as the basic features for LLMs to process language. Subword tokenization (e.g., Byte Pair Encoding or WordPiece) is commonly used to handle rare words.

2. Word Embeddings: Converting words into dense vector representations, capturing their semantic meaning. Pre-trained embeddings like Word2Vec, GloVe, or model-specific embeddings (e.g., BERT embeddings) help the model understand relationships between words.

3. Positional Encoding: Adding information about the position of words in a sequence. Since transformers process words in parallel, positional encoding helps the model maintain the order and structure of the text.

4.Text Normalization: Lowercasing, removing special characters, and normalizing punctuation to create consistent and cleaner features from raw text.

Methods for Creating Features

5. Named Entity Recognition (NER): Identifying and tagging entities (e.g., people, locations, organizations) within the text, allowing the model to treat named entities differently from regular words.

6. Part-of-Speech (POS) Tagging: Assigning grammatical tags (noun, verb, adjective, etc.) to words can provide additional context about their function in a sentence, enhancing the model's understanding of sentence structure.

7. N-grams: Creating features from sequences of words (bigrams, trigrams) to capture context and word dependencies beyond individual tokens, which is useful for tasks like text classification.

8. Domain-Specific Features: For specialized tasks, creating features like specific jargon or acronyms relevant to the field (e.g., medical terms in healthcare) helps tailor the LLM to a particular domain.

End of Session 2