Generative AI

22AIM72

Course Outcomes

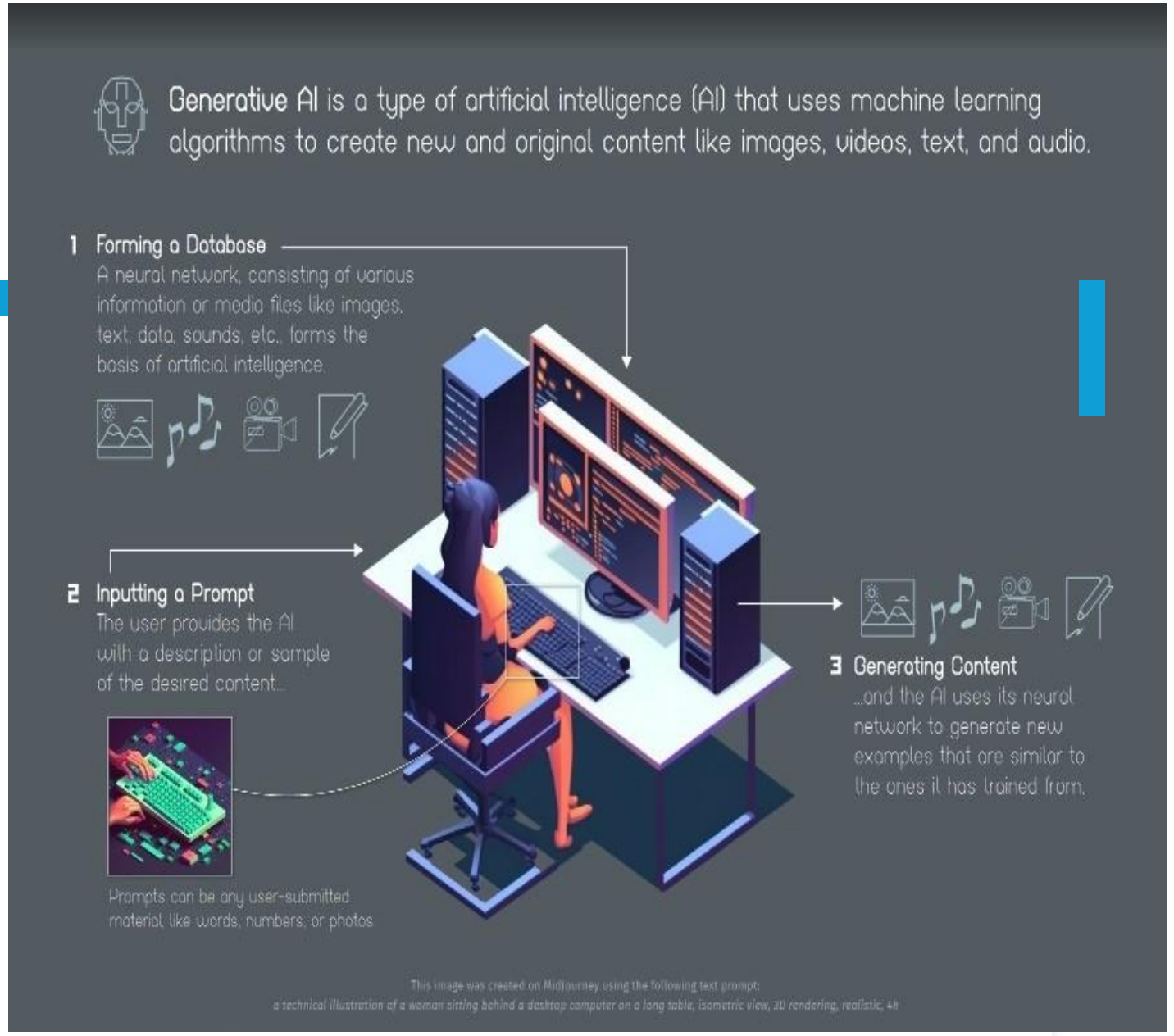
Syllabus

- Module 1

Transformer overview: attention, encoder–decoder structure. BERT, T5, Mistral 7B, Phi-3 – comparison & use cases. Tokenization: BPE, WordPiece. Embeddings and positional encodings. Pre-training vs fine-tuning (instruction/domain tuning). Parameter-efficient tuning: LoRA, QLoRA, PEFT.Prompt engineering: zero-shot, few-shot, CoT, ReAct.

Generative AI

- **Generative AI refers to a category of AI algorithms that generate new outputs based on the data they have been trained on.**
- **It uses a type of deep learning called generative adversarial networks and has a wide range of applications, including creating images, text and audio.**
- **VAEs(Variational Auto Encoders) built with encoders and decoders, opened the floodgates to deep generative modeling by making models easier to scale**



Examples of Generative AI Technology

ChatGPT
text generation AI

INPUT PROMPTS

"Write a limerick about data taking over the world. Include the word egg"

OUTPUT

"Data taking over the world, it's true
It's growing like an egg that's due
It's everywhere we go
It's taking over slow
But it's the future, there's nothing we can do."

Real-Life Applications of this Platform
• communications, journalism, publishing, creative writing, writing assistance

MidJourney
image generation AI

INPUT PROMPTS

- a rabbit wearing a sunhat
- Santa Claus in a blue outfit
- a watercolor of a red bicycle in front of a waterfall

OUTPUT

Real-Life Applications of this Platform
• advertisements, published illustrations, corporate visuals, novel image generation

Codex
computer coding generation AI

INPUT PROMPTS

"Write script to change a website's background color in Python"

OUTPUT

Here's an example script that uses the `requests` and `BeautifulSoup` libraries in Python to change the background color of a website:

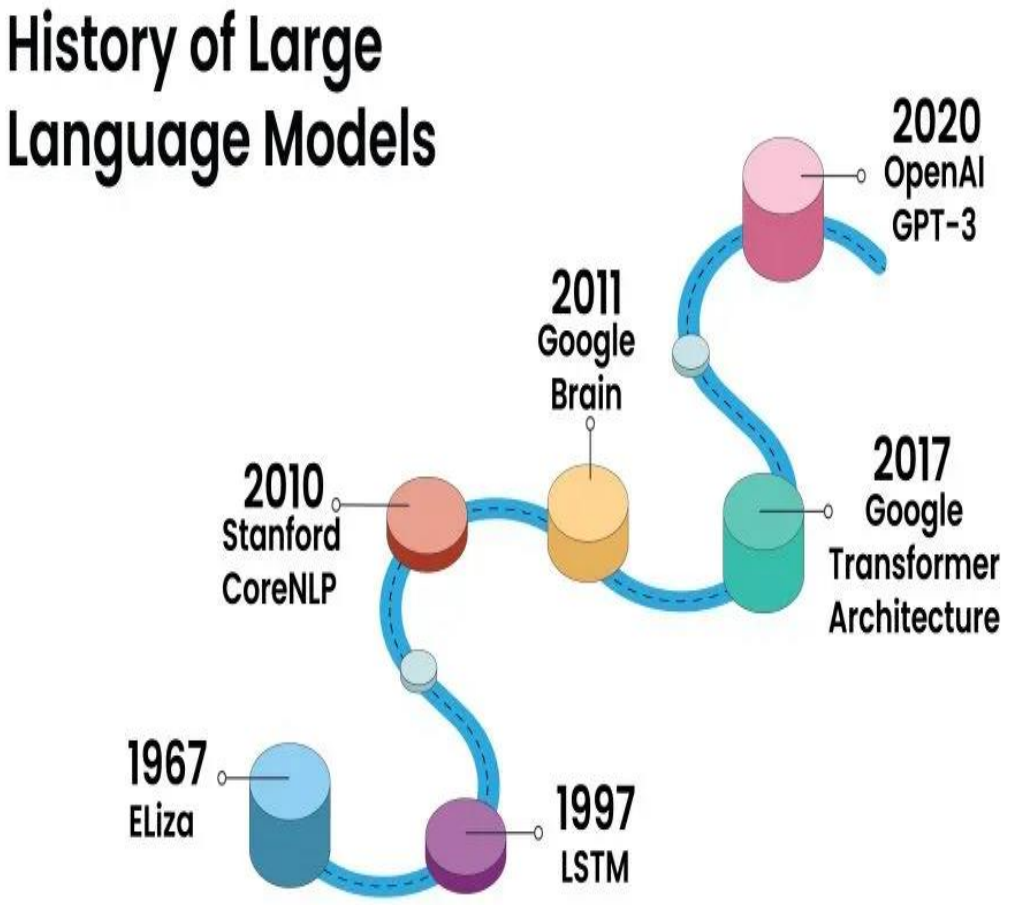
```
import requests
from bs4 import BeautifulSoup

# send a GET request to the website
response = requests.get("https://yourwebsite.com")

# parse the HTML content
soup = BeautifulSoup(response.content, 'html.parser')

# locate the element you want to change the background color of
element = soup.find('body')

# add a new style attribute to the element
element['style'] = 'background-color: pink;'

# print the modified HTML
print(soup.prettify())
```

Real-Life Applications of this Platform
• web design, software development, coding/scripting, technology

LLMs

- Large language models (LLMs) are a category of foundation models trained on immense amounts of data making them capable of understanding and generating natural language and other types of content to perform a wide range of tasks.eg: Meta’s Llama models and Google’s bidirectional encoder
- These models are typically based on a transformer architecture.
- Large language models typically use a transformer architecture, which consists of an encoder and a decoder. The encoder processes the input text, while the decoder generates the output text. Both the encoder and decoder are made up of multiple layers of self-attention mechanisms, which help the model understand the relationships between words in a sequence.

LLMs

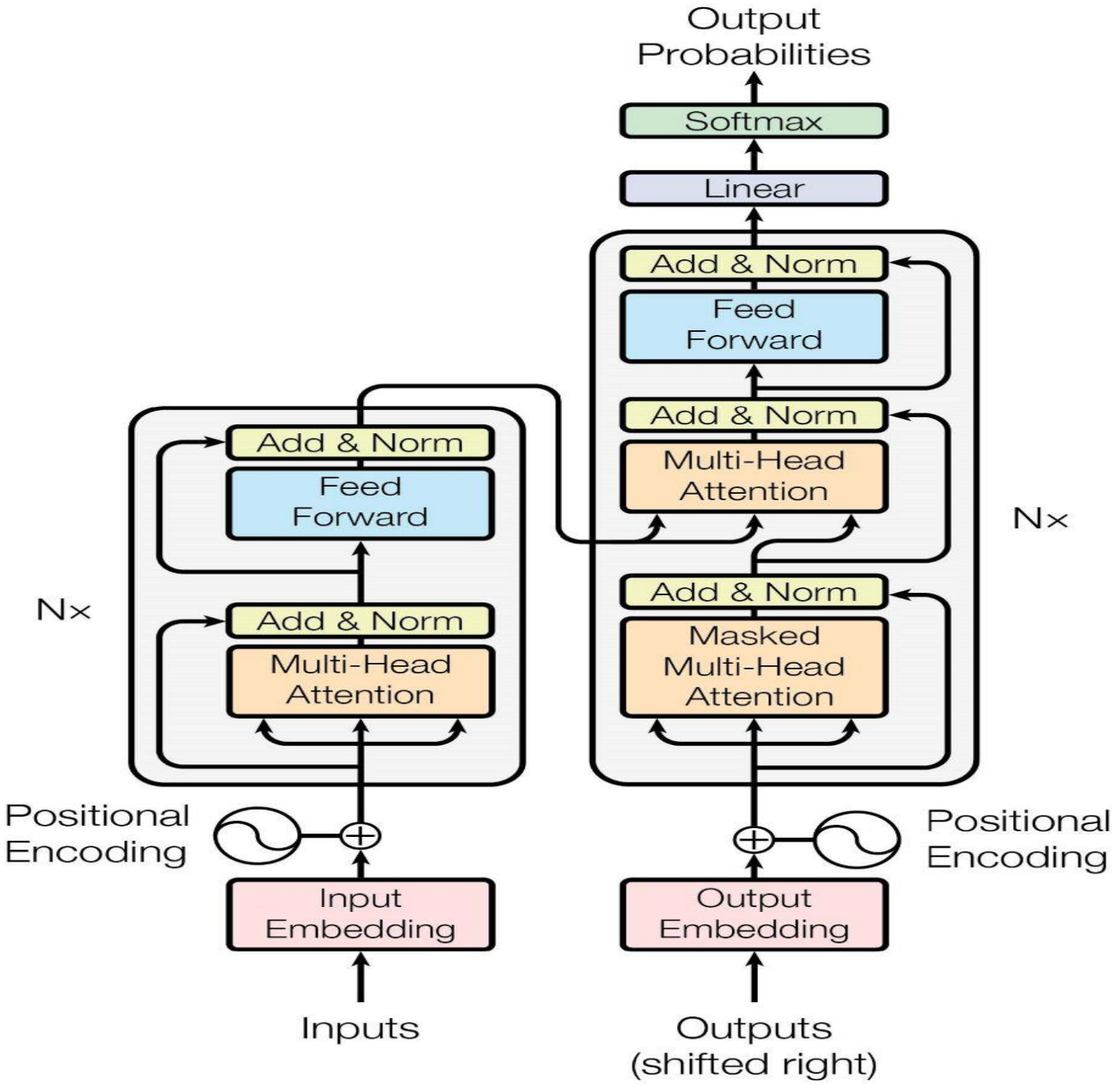


- Types
- **GPT-3 (OpenAI)-** OpenAI, 175 billion parameters
- **BERT (Google)-**processes text bi-directionallyBidirectional Encoder Representations from Transformers
- **T5 (Google)-**Text-to-Text Transfer Transformer
- **RoBERTa (Facebook)-**Robustly Optimized BERT Pretraining Approach

Transformers

- Introduced in the 2017 paper "Attention is All You Need" by Vaswani et al., Transformers eliminate recurrence by relying entirely on attention mechanisms.
- This allows for parallel processing of entire sequences
- **Key Innovation**: Self-attention, which enables the model to weigh the importance of different parts of the input sequence simultaneously.
- • **Advantages**:
- • **Parallelization:** Processes all tokens at once, significantly speeding up training.
- • **Long-range dependencies:** Directly connects any two positions, improving performance on tasks requiring context over long sequences.

Architecture



- The Transformer architecture consists of an encoder and decoder, each composed of multiple identical layers

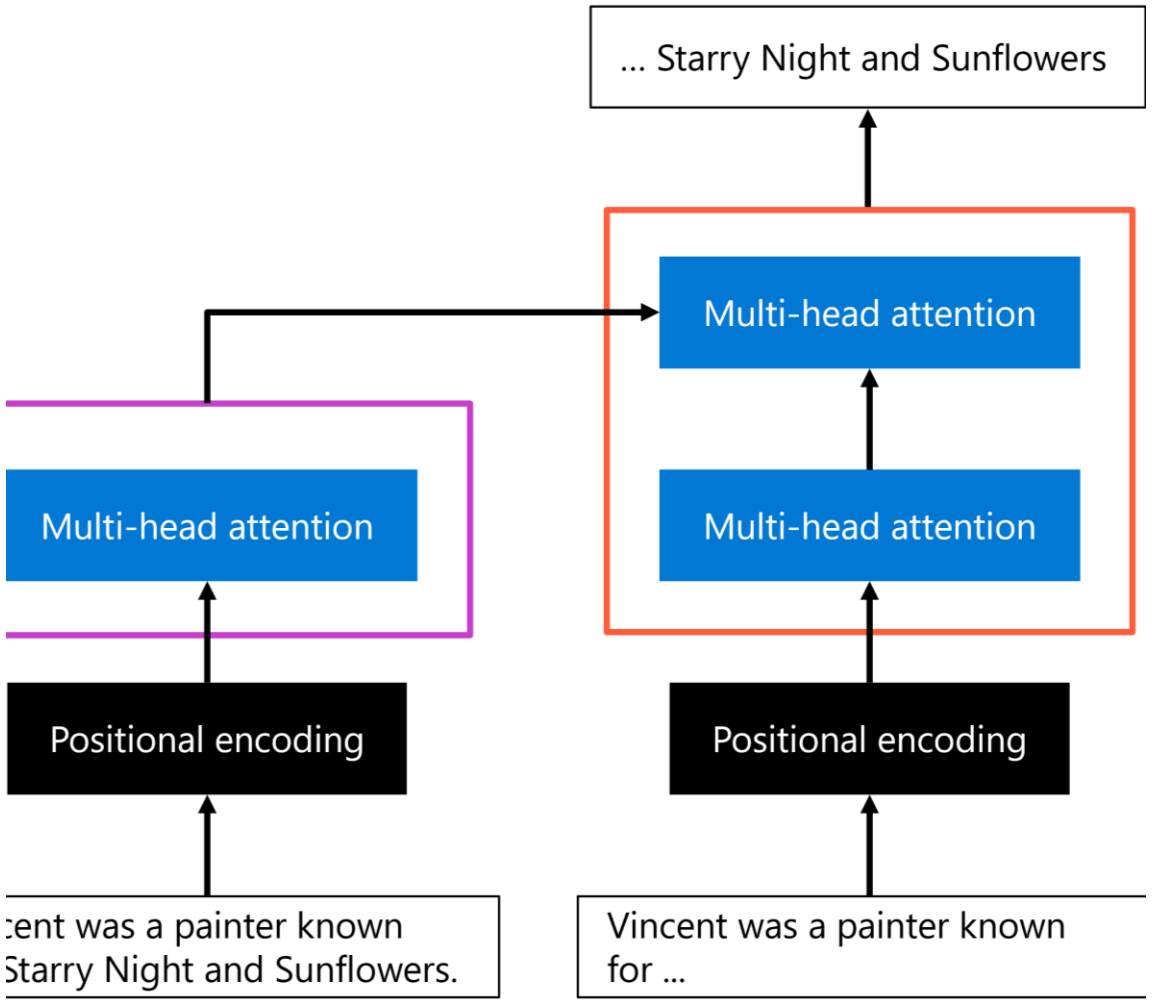
Component	Purpose	When to Use	Example
Input Embeddings	Convert tokens (words or subwords) into dense vectors of fixed size (e.g., 512 dimensions).	Essential for all Transformer models to represent input data numerically.	The word "cat" is mapped to a 512-dimensional vector capturing its semantic meaning.
Positional Encoding	**Add information about token positions** using sine and cosine functions, as Transformers lack inherent sequence order.	Critical for tasks where word order matters, like translation or summarization.	For "The cat is on the mat," positional encodings ensure "cat" is distinguished as the second word

Multi-Head Self-Attention	**Allows each token to attend to all others, capturing diverse relationships** (e.g., syntactic, semantic) via multiple attention heads (e.g., 8 heads).	Used in encoders for bidirectional context and decoders for autoregressive tasks.	In "The animal didn't cross the street because it was tired," one head may focus on "animal" and "it," another on "tired" and "didn't."
Masked Self-Attention	**Prevents the decoder from attending to future tokens,** ensuring autoregressive generation.	Essential for generative tasks like text generation or translation.	When generating "is" in "The cat is," the decoder cannot see "on the mat."
Encoder-Decoder Attention	Allows the decoder to attend to encoder outputs, **aligning source and target sequences.**	Used in sequence-tosequence tasks like translation.	In translating to French, the decoder attends to "cat" in the encoder to generate "chat."
Feed-Forward Neural Networks	Apply position-wise non-linear transformations to **enhance model capacity**.	Used in every layer to process attention outputs	Transforms the attention output for "cat" to capture complex patterns.

Add and Norm (Residual Connections and Layer Normalization)	Stabilize training by adding input to output of sub-layers (residual connections) and normalizing layer outputs to maintain **consistent scales.**	Essential in every layer to prevent vanishing gradients and ensure stable training.	Adds the input of selfattention to its output, then normalizes to stabilize training for "cat."
Linear Layer	**Maps the decoder's final hidden states to the vocabulary size for token prediction.**	Used in the final decoder layer for generative tasks.	Maps a 512-dimensional hidden state to a 50,000-token vocabulary for predicting the next word.
Softmax	**Converts linear layer outputs into probabilities over the vocabulary for token selection.**	Used in the final decoder layer to produce a probability distribution for generative tasks.	Outputs probabilities over a 50,000-token vocabulary, selecting "is" as the next word in "The cat."

Transformers

- There are two main components in the original Transformer architecture:
- The encoder: Responsible for processing the input sequence and creating a representation that captures the context of each token.
- The decoder: Generates the output sequence by attending to the encoder's representation and predicting the next token in the sequence.
- The most important innovations presented in the Transformer architecture were *positional encoding* and *multi-head attention*.



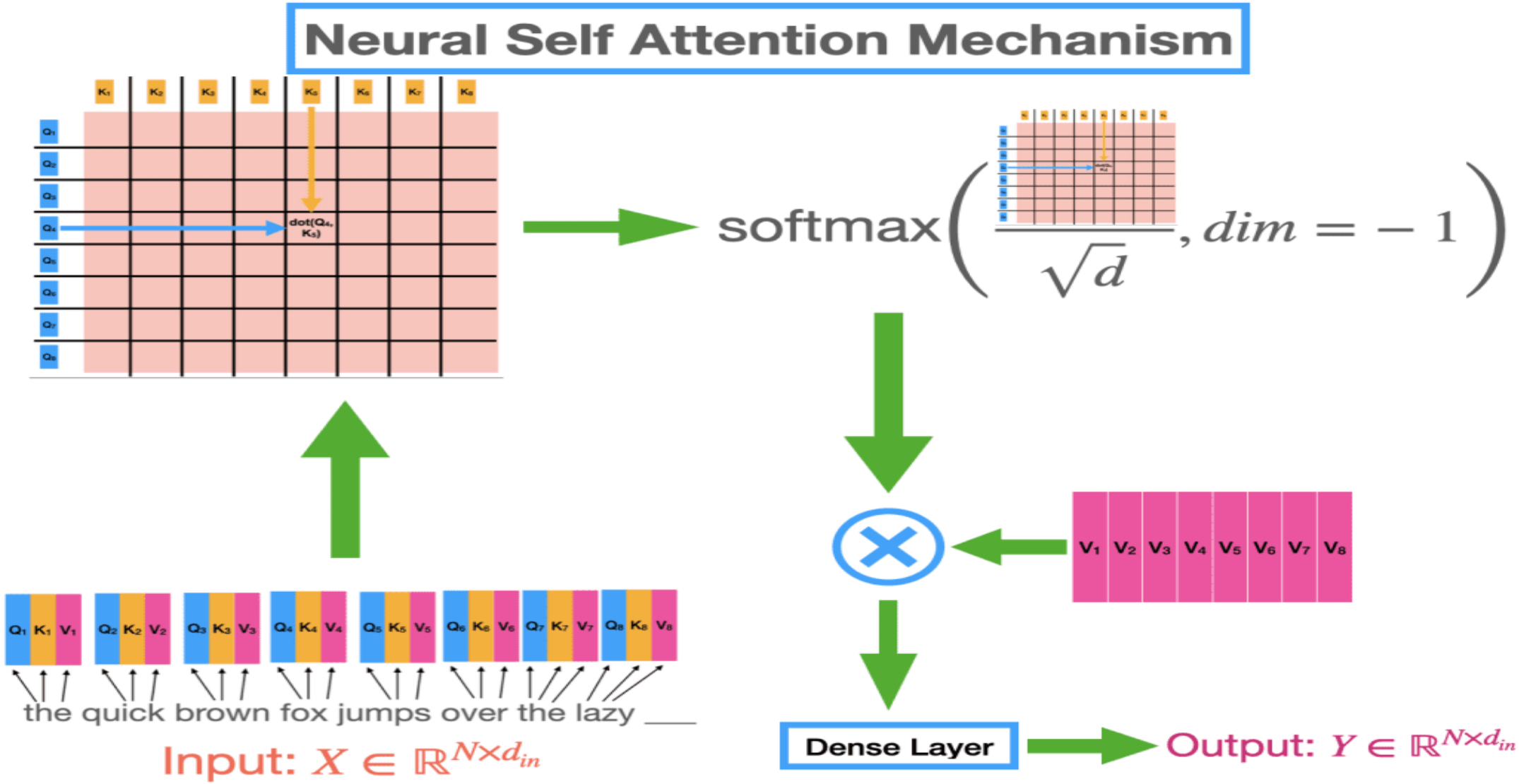
Transformer

- Encoder layer: An input sequence is encoded with **positional encoding, after which multi-head attention is used to create a representation of the text.**
- Decoder layer: An (incomplete) output sequence is encoded in a similar way, by first using **positional encoding and then multi-head attention**. Then, the multi-head attention mechanism is used a second time within the decoder to combine the output of the encoder and the output of the encoded output sequence that was passed as input to the decoder part. As a result, the output can be generated.
- **Positional encoding**
- Positional encoding is used to encode text into vectors. It is the sum of word embedding vectors and positional vectors Eg: The word of William Shakespeare is encoded into 0,1,2,3,4. *Attention is all you need* paper uses sine and cosine functions, where pos is the position and i is the dimension

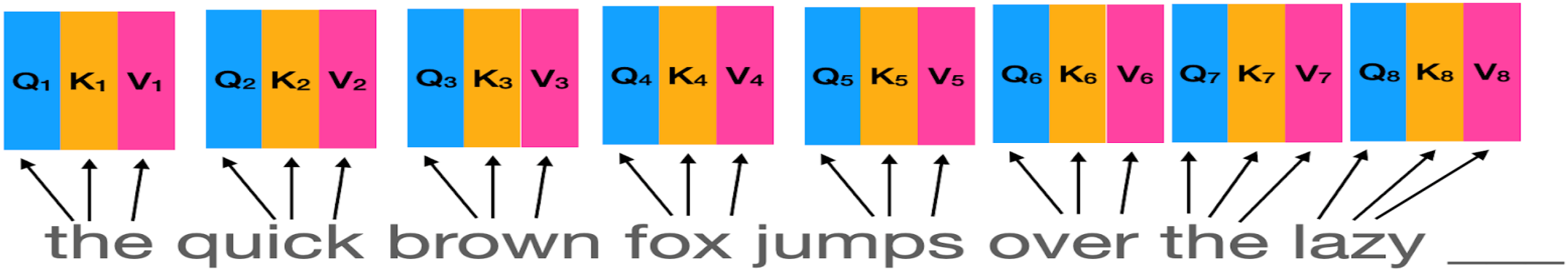
$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$

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

Attention Mechanism



Self Attention



Step 1, Query, key value: Each word ask questions and receive answers. A word asks the same question to all words in the sequence using the 'query' vector. Similarly, it provides the same answer to all words using the 'key' vector. Vector is also used to allow the model to combine the outputs of query and key vectors non-linearly.

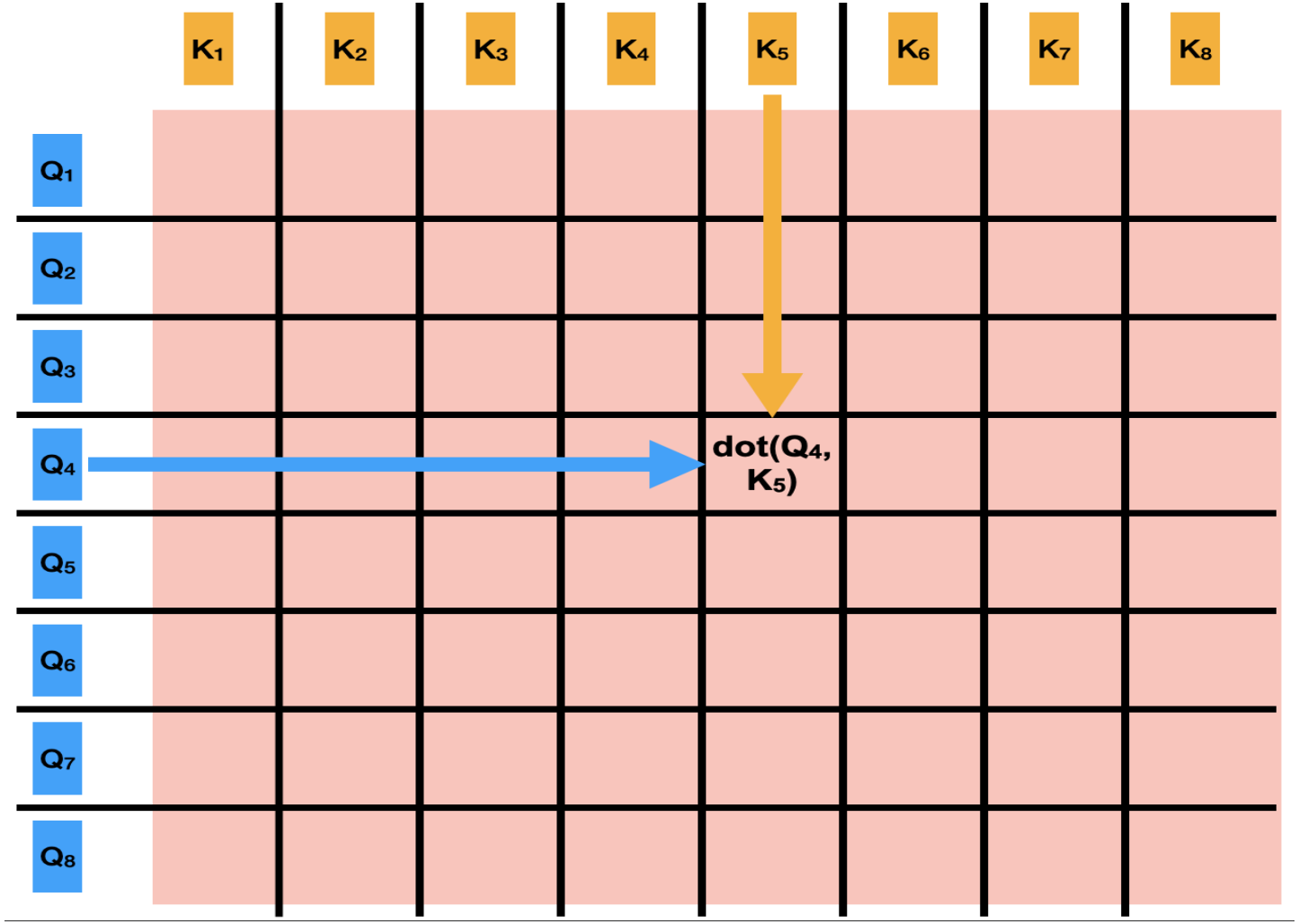
query vector represents the information a given token is seeking.

key vectors represent the information that each token contains.

***value* (or *value vector*)** applies the attention-weighted information from the key vectors.

In Step 1. each word in the sentence emits three separate representations of itself, called the query Q, key K, and value V vectors using three separate dense layers.

- **Step 2, Computing the Attention Matrix:**



For a sentence with N words, after step 1, we have N query, N key and N value vectors.

We take the dot product of all the N query vectors with all the N key vectors, as shown in figure

- **Step 3, Normalization And Attention Scores**
- The attention matrix, as computed above, will contain some large and some small numbers. We will interpret the large numbers as representing high relevance and the small relevance as low relevance.
- Divide the attention matrix by the square root of the dimension of each query vector.
- Take a row-wise softmax of the attention matrix, resulting in an NxN matrix whose each row is a probability distribution.

- **Step 4, Attend:** We take the dot product of the normalized attention scores obtained in the previous step with the value vectors. Mathematically, this is represented as

$$H = V \cdot softmax(QKd)$$

Transformers

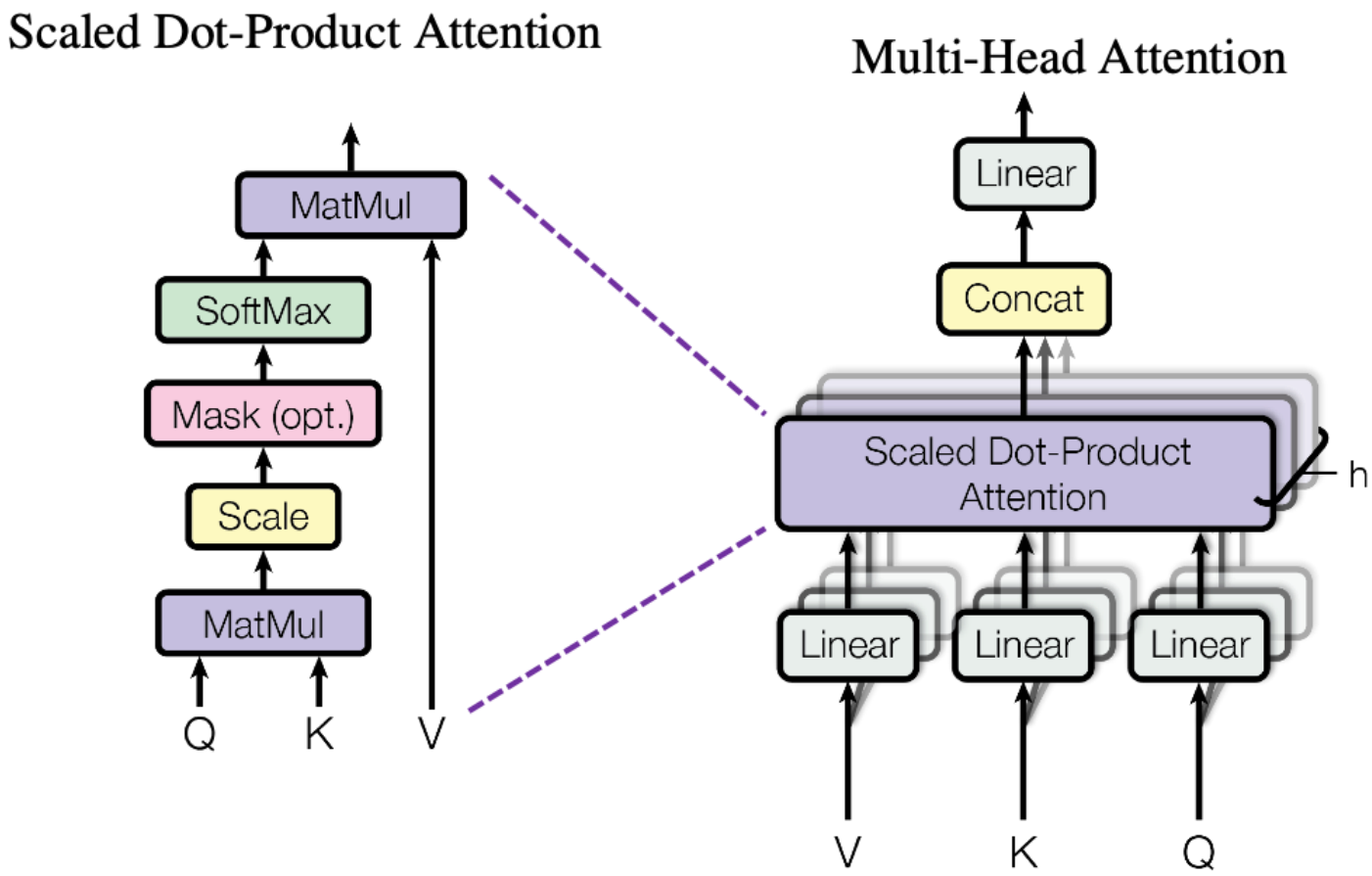
Attention

- The self attention mechanism allows transformers to determine which words in a sentence are most relevant to each other. This is done using a scaled dot-product attention approach.Each word in a sequence is mapped to three vectors Query (Q) ,Key (K),Value (V).
- To calculate the attention function, the query, keys, and values are all encoded to vectors. The attention function then computes the scaled dot-product between the query vector and the keys vectors.

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

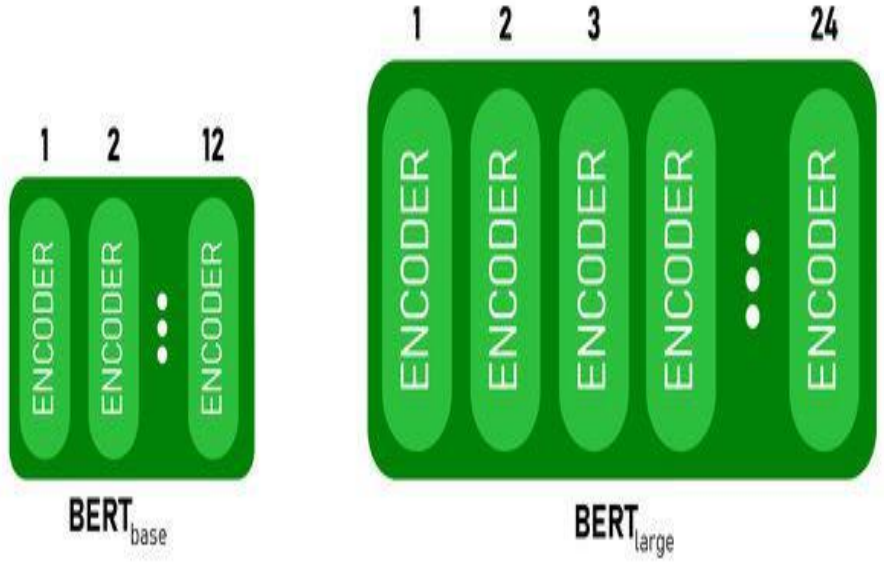
- The dot-product calculates the angle between vectors representing tokens, with the product being larger when the vectors are more aligned.
- The softmax function is used within the attention function, over the scaled dot-product of the vectors, to create a probability distribution with possible outcomes. The key with the highest probability is then selected, and the associated value is the output of the attention function.

- The Transformer architecture uses multi-head attention, which means tokens are processed by the attention function several times in parallel.
- By doing so, a word or sentence can be processed multiple times, in various ways, to extract different kinds of information from the sentence.



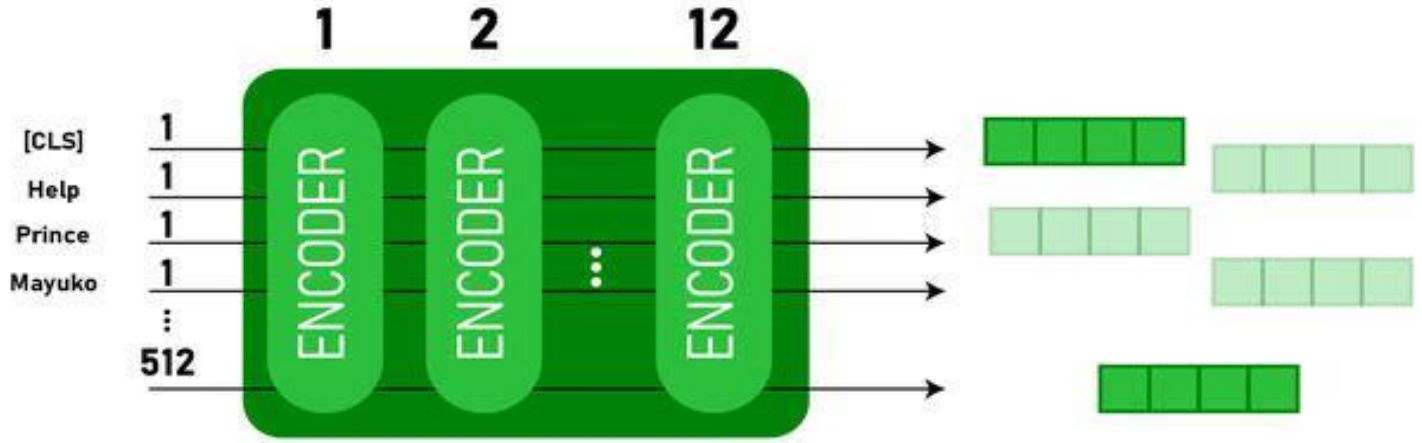
Variants of Transformers- BERT,T5,Mistral 7b,Phi-3 Comparisons

- **BERT**
- BERT (Bidirectional Encoder Representations from Transformers) . BERT employs an encoder-only architecture. BERT uses a bi-directional approach considering both the left and right context of words in a sentence, instead of analyzing the text sequentially. Primarily it has two model sizes: BERT BASE and BERT LARGE.
- BERTBASE has 1*2 layers in the Encoder stack* while BERTLARGE has *24 layers in the Encoder stack.*



BERT

- This model takes the CLS token as input first, then it is followed by a sequence of words as input. Here CLS is a classification token. It then passes the input to the above layers. Each layer applies self-attention and passes the result through a feedforward network after then it hands off to the next encoder. The model outputs a vector of hidden size (*768* for BERT BASE). If we want to output a classifier from this model we can take the output corresponding to the CLS token.

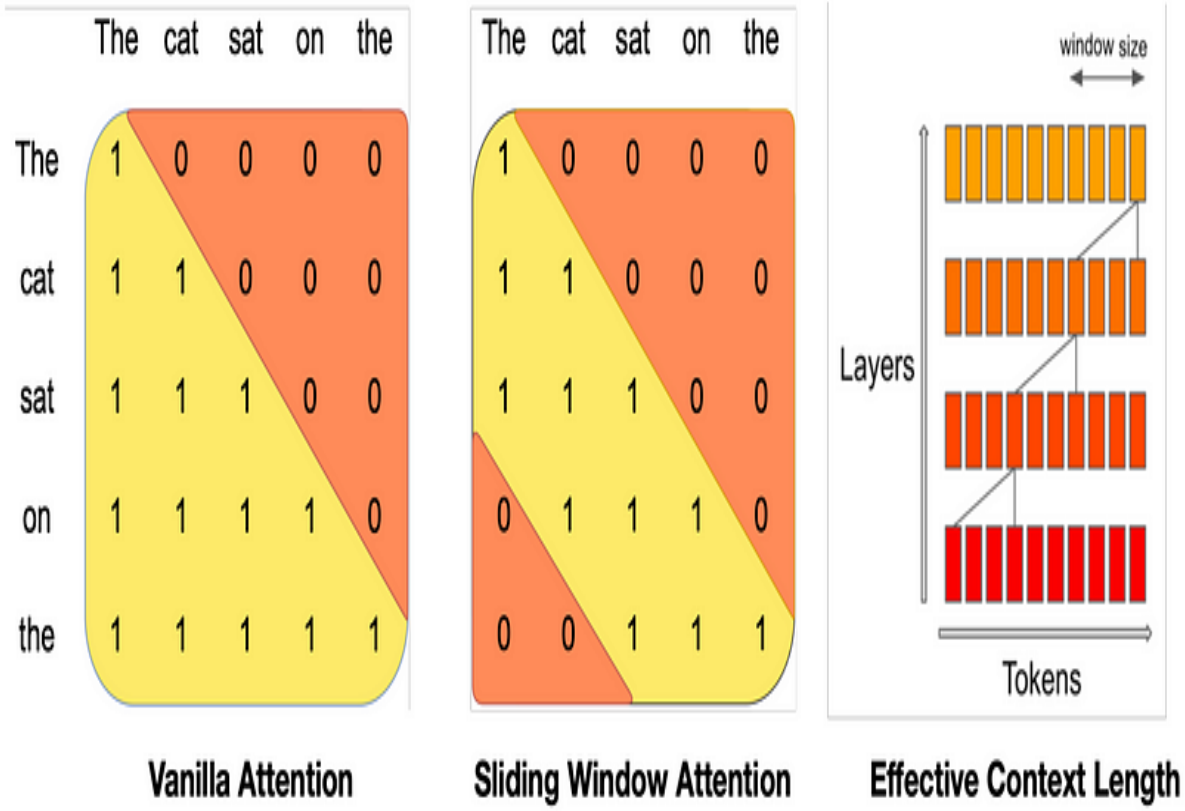


T5 (Text-to-Text Transfer Transformer)

- T5 (Text-to-Text Transfer Transformer) is a transformer-based model developed by Google Research.
- This can be applied to various tasks such as translation, summarization and question answering.
- Model uses encoder-decoder architecture similar to Transformer-based sequence-to-sequence models. It works by :
- Task Formulation as Text-to-Text: Instead of treating different NLP tasks separately it reformulates each problem into a text-based input and output.
- Encoding the Input: The input text is tokenized using SentencePiece, then passed through the encoder which generates a contextual representation.
- Decoding the Output: The decoder takes the encoded representation and generates the output text in a autoregressive manner.
- Training the Model: T5 is pre-trained using a denoising objective where portions of text are masked and the model learns to reconstruct them. It is then fine-tuned for various tasks.

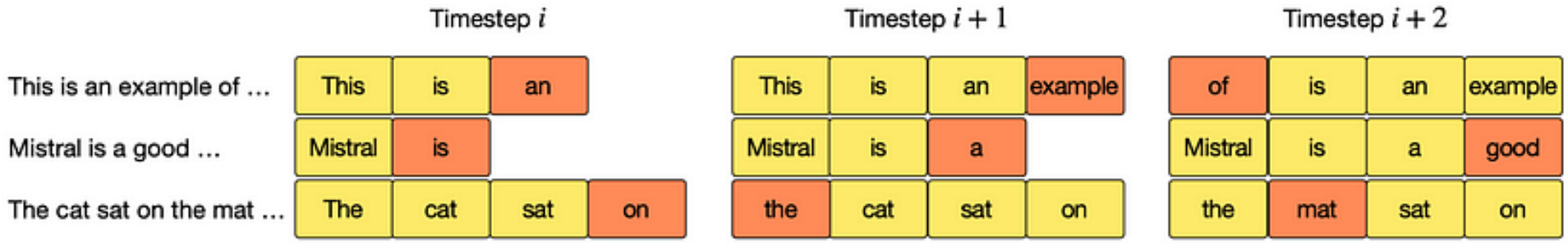
Mistral 7B

- Mistral 7B is an LLM engineered for superior performance and efficiency. It leverages grouped-query attention (GQA) for faster inference, coupled with sliding window attention (SWA) to effectively handle sequences of arbitrary length with a reduced inference cost.
- Sliding Window Attention leverages the layers of a transformer model to extend its attention beyond a fixed window size, denoted as W. In SWA, the hidden state at position i in layer k can attend to hidden states from the preceding layer within the range of positions i — W to i, allowing access to tokens at a distance of up to W * k tokens



Mistral 7B

- Rolling Buffer Cache
- A Rolling Buffer Cache, employs a fixed attention span to limit cache size. The cache is of fixed size W, and it stores keys and values for timestep i at position i mod W in the cache. When i exceeds W, earlier values are overwritten, halting cache size growth.



Mistral 7B

- **Pre-fill and chunking**
- In sequence generation, tokens are predicted sequentially based on prior context. To optimize efficiency, a (k, v) cache is pre-filled with the known prompt. If the prompt is very long, it is chunked into smaller segments using a chosen window size. Each chunk is used to pre-fill the cache.
- Mixture of Experts
- Mixture of Experts is an ensemble technique in which we have multiple “expert” models, each trained on a subset of the data such that each model specializes on it and then the output of the experts are combined to produce one single outp

Tokenization

- Tokenization is a fundamental process in Natural Language Processing (NLP) that involves breaking down a stream of text into smaller units called tokens.
- These tokens can range from individual characters to full words or phrases.
- ln deep learning–based models (e.g., BERT), each token is mapped to an embedding vector to be fed into the model.

Tokenization Techniques in NLP

- **Whitespace Tokenization**

It tokenizes based on spaces between words. Example: "madhusudhan reddy gone." → ["madhusudhan", "reddy", "gone."]

- **Rule-based / Regex Tokenization**

It tokenizes text based on predefined rules or regular expressions (e.g., split by punctuation or specific patterns). Example: "madhusudhan, reddy gone!" → ["madhusudhan", ",", "reddy", "gone", "!"]

- **Character-level Tokenization**

It tokenizes text at the individual character level. Example: "madhu" → ["m", "a", "d", "h", "u"]

Word-level Tokenization

It tokenizes text by separating words (typically using whitespace and punctuation). Example: "madhusudhan reddy gone." → ["madhusudhan", "reddy“,”gone”,”.”]

Tokenization Techniques in NLP

- **Subword Tokenization**
- Breaks words into smaller units (subwords) to handle unknown or rare words. It includes the following techniques:

a)Byte-Pair Encoding (BPE) It merges frequent character pairs to form subwords. Example: "unbelievable" → ["un", "believ", "able"]

b) Byte-level BPE Same as BPE, but operates at the byte level instead of characters. Used in GPT-2, GPT-3. "hello!"

→ ['h', 'e', 'l', 'l', 'o', '!']

→ [104, 101, 108, 108, 111, 33]

c)WordPiece It splits words into subwords based on frequency and greedy matching. Used in BERT. == Example: "unbelievable" → ["un", "##believable"]

Visualize: https://platform.openai.com/tokenizer

Example-

```
import tiktoken
# Load tokenizer (use model-specific encoding)
enc = tiktoken.get_encoding("cl100k_base")  # can also use 'gpt2', 'p50k_base', etc.
# Encode text → token IDs
text = "madhusudhan reddy"
token_ids = enc.encode(text)
print("Token IDs:", token_ids)
# Decode token IDs → text
decoded = enc.decode(token_ids)
print("Decoded text:", decoded)
```

Output

Token IDs: [20920, 13092, 664, 10118, 312, 54610] Decoded text: madhusudhan reddy

Byte Pair Encoding

- Used by OpenAI for tokenization when pretraining the GPT model. It's used by a lot of Transformer models, including GPT, GPT-2, RoBERTa, BART, and DeBERTa.
- Example:m i s s i s s i p p i
- Initial vocabulary = { "m", "i", "s", "p" }

Step 2: Count frequent pairs

- ("m", "i") → 1
- ("i", "s") → 2
- ("s", "s") → 2
- ("s", "i") → 2
- ("i", "p") → 1
- ("p", "p") → 1
- ("p", "i") → 1
- The most frequent pairs are ("i","s"), ("s","s"), and ("s","i"), all with 2. Let's pick **("s","s")** first.

BPE Example

- Step 3: Merge ("s","s") → "ss“
- m i ss i ss i p p i
- Vocabulary = { "m", "i", "s", "p", "ss" }
- Step 4: Count pairs again

•("m", "i") → 1
•("i", "ss") → 2
•("ss", "i") → 2
•("i", "p") → 1
•("p", "p") → 1
•("p", "i") → 1

- Most frequent: ("i","ss") and ("ss","i"), both 2. Pick **("i","ss")**.

BPE

Step 5: Merge ("i","ss") → "iss“

m iss iss i p p I

Vocabulary = { "m", "i", "s", "p", "ss", "iss" }

Step 6: Count pairs
- ("m", "iss") → 1
- ("iss", "iss") → 1
- ("iss", "i") → 1
- ("i", "p") → 1
- ("p", "p") → 1
- ("p", "i") → 1

Pick ("iss","i") → merge → "issi".

m issi ss i p p i → actually better: m issi ppi

Depending on which pairs we choose next, results may differ, but the idea is the same: gradually merge common substrings

WordPiece

- WordPiece is the subword tokenization algorithm used for BERT, DistilBERT, and Electra and is very similar to BPE.
- In contrast to BPE, WordPiece computes a score for each pair, using the following formula:
  score=(freq_of_pair)/(freq_of_first_element×freq_of_second_element)
- By dividing the frequency of the pair by the product of the frequencies of each of its parts, the algorithm prioritizes the merging of pairs where the individual parts are less frequent in the vocabulary.
- WordPiece first initializes the vocabulary to include every character present in the training data and progressively learns a given number of merge rules

How WordPiece Tokenization Works

- The algorithm follows a data-driven approach to build its vocabulary. It starts with individual characters and gradually merges the most frequently occurring pairs until reaching a target vocabulary size.
- **Algorithm steps:**
- Initialize vocabulary with all individual characters
- Count frequency of all adjacent symbol pairs in the corpus
- Merge the most frequent pair into a single token
- Update the corpus with the new merged token
- Repeat until reaching desired vocabulary size (typically 30K-50K tokens)

Example

Example: Corpus = **“unhappiness unhappy unhappiness”**

Step 1: Start with characters u n h a p p i n e s s

Vocabulary = { "u", "n", "h", "a", "p", "i", "e", "s" }

Step 2: Count possible merges

- un" → 3 times
- "hap" → 3 times
- "pi" → 2 times
- "ness" → 2 times

Score("h","ap")=Count("hap") / Count("h")×Count("ap")

= 3 / 3×3

=0.33

- During actual tokenization, WordPiece uses a greedy longest-match strategy. For each word, it finds the longest possible subword that exists in its vocabulary then marks it as a token and repeats for the remaining characters.

Word Piece Tokenization

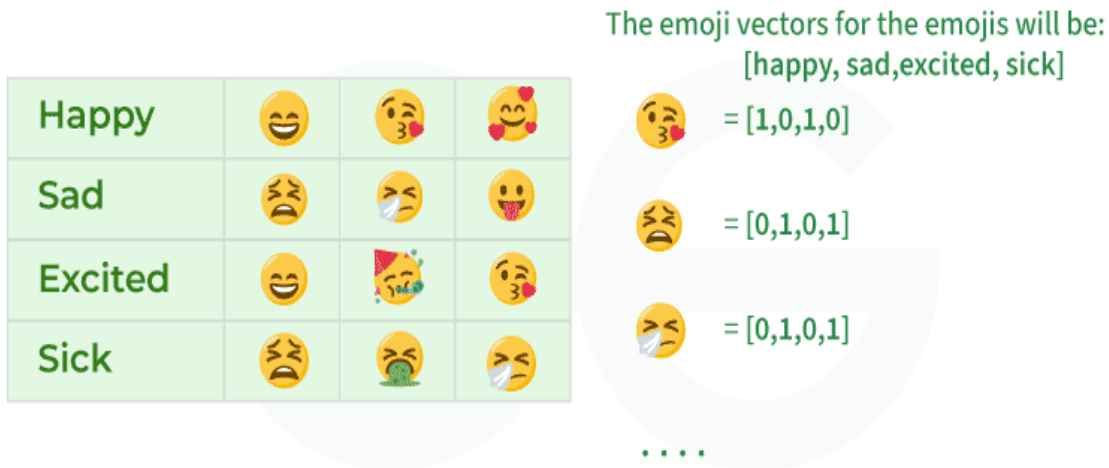
- **Tokenization process:**
- Start from the beginning of each word
- Find the longest matching subword in vocabulary
- Add it to the token list with appropriate prefix
- Move to the next unprocessed characters
- Continue until the entire word is processed
- This statistical foundation ensures that common patterns naturally emerge as single tokens while rare combinations get broken into more familiar components.

Embeddings

- Embeddings are dense vector representations of tokens that capture their **semantic and syntactic meaning** in a continuous vector space. They allow models to understand relationships between words .

e.g., "king" and "queen" are close in vector space.

- Embeddings enable tasks like sentiment analysis by representing words in a way that reflects their meaning and context.



Embeddings

- **Types of Embeddings**
- **Static Embeddings**: Fixed vectors for each word, regardless of context.
- **Examples**: Word2Vec, GloVe
- **Limitation**:bank" always has one vector (e.g., [0.1, 0.2, 0.3]). Struggles with polysemy (e.g., "bank" in "river bank" vs. "bank account" has the same vector).
- **Contextual Embeddings**: Dynamic vectors that change based on surrounding words.
- **Examples**: BERT, GPT
- **Advantage**: Captures nuances (e.g., "bank" has different vectors in different contexts

Embeddings

- **How Are Embeddings Created?**
- **Static Embeddings**: Trained on large corpus using algorithms like:

Word2Vec: Predicts context words or target words (CBOW-Continuous Bag of Words) based on cooccurrences.

GloVe: Uses global word co-occurrence statistics.

- **Contextual Embeddings**: Generated by transformer models using attention mechanisms to consider the entire sentence.

Example

- Example of word embeddings for a very small corpus (6 words), where each word is represented as a 3-dimensional vector:
- **cat** **[0.2, -0.4, 0.7]**
  dog **[0.6, 0.1, 0.5]**
  apple **[0.8, -0.2, -0.3]**
  orange **[0.7, -0.1, -0.6]**
  happy **[-0.5, 0.9, 0.2]**
  sad **[0.4, -0.7, -0.5]**
- The values in the vector represent the word's position in a continuous 3-dimensional vector space. Words with similar meanings or contexts are expected to have similar vector representations. For instance, the vectors for "cat" and "dog" are close together, reflecting their semantic relationship. Likewise, the vectors for "happy" and "sad" have opposite directions, indicating their contrasting meanings.

Positional encoding

- **positional encoding** was introduced which adds information about each token's position in the sequence which helps model to understand relationships and order between tokens. Suppose we have a Transformer model which translates English sentences into French.
- *"The cat sat on the mat.“*
- *After tokenization: ["The", "cat" , "sat", "on", "the" ,"mat"]*
- After that each token is mapped to a high-dimensional vector representation through an embedding layer. These embeddings encode semantic information about the words in the sentence. However they lack information about the order of the words.
- Positional encoding are added to the word embeddings and they assign each token a unique representation based on its position in the sequence.

- To calculate positional encoding,For each position *p* in the sequence and for each dimension 2i and 2i+1 in the encoding vector:

1. **Even-indexed dimensions:** $PE(p, 2i) = \sin\left(\frac{p}{10000^{\frac{2i}{d\text{model}}}}\right)$
2. **Odd-indexed dimensions:** $PE(p, 2i+1) = \cos\left(\frac{p}{10000^{\frac{2i}{d\text{model}}}}\right)$

k : The position of the word in the sentence (for example, 0 for the first word, 1 for the second, and so on.)

i : The dimension index of the embedding vector. maps to column index. 2i will indicate an even position and 2i+1 will indicate an odd position

dmodel : The predefined dimensionality of the token embeddings (for example, 512)

n : user-defined scaler value (for example, 10000)

PE : position function for mapping position k in the input sequence to get the positional mapping

Pre-training vs fine-tuning (instruction/domain tuning)

- **Pre-training** involves training a neural network model on a large corpus of text data in an unsupervised manner. It is an initial phase of machine learning training that is a crucial step to equip an LLM with general language understanding capabilities. **After pre-training it can be fine-tuned to accomplish the desired results.**
- **Fine-tuning i**s a supervised learning process, requiring a relatively small set of labeled data. The use of a model that is already pre-trained is precisely what makes it possible to employ a relatively small set of labeled data for additional training.

Steps in LLM Pretraining

- **Data Collection**

Gather a large and diverse corpus of text data from various sources

- **Cleaning**

Removing or replacing any non-textual elements and duplicated text samples from the raw data to ensure that it is consistent and meaningful

- **Tokenization**

Tokenize the text data into smaller units such as words, subwords, or characters

Architecture Selection

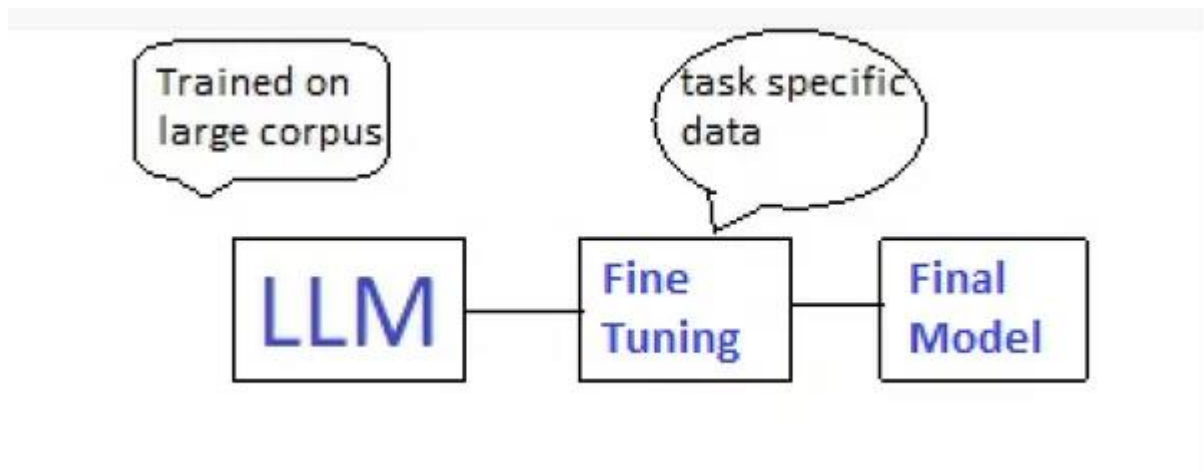
A transformer-based architecture is often chosen for the model, as it works well with sequences in texts.

Pre-training process

Feeding a large dataset as input through the model to make it comprehend and create human-like sequences of text

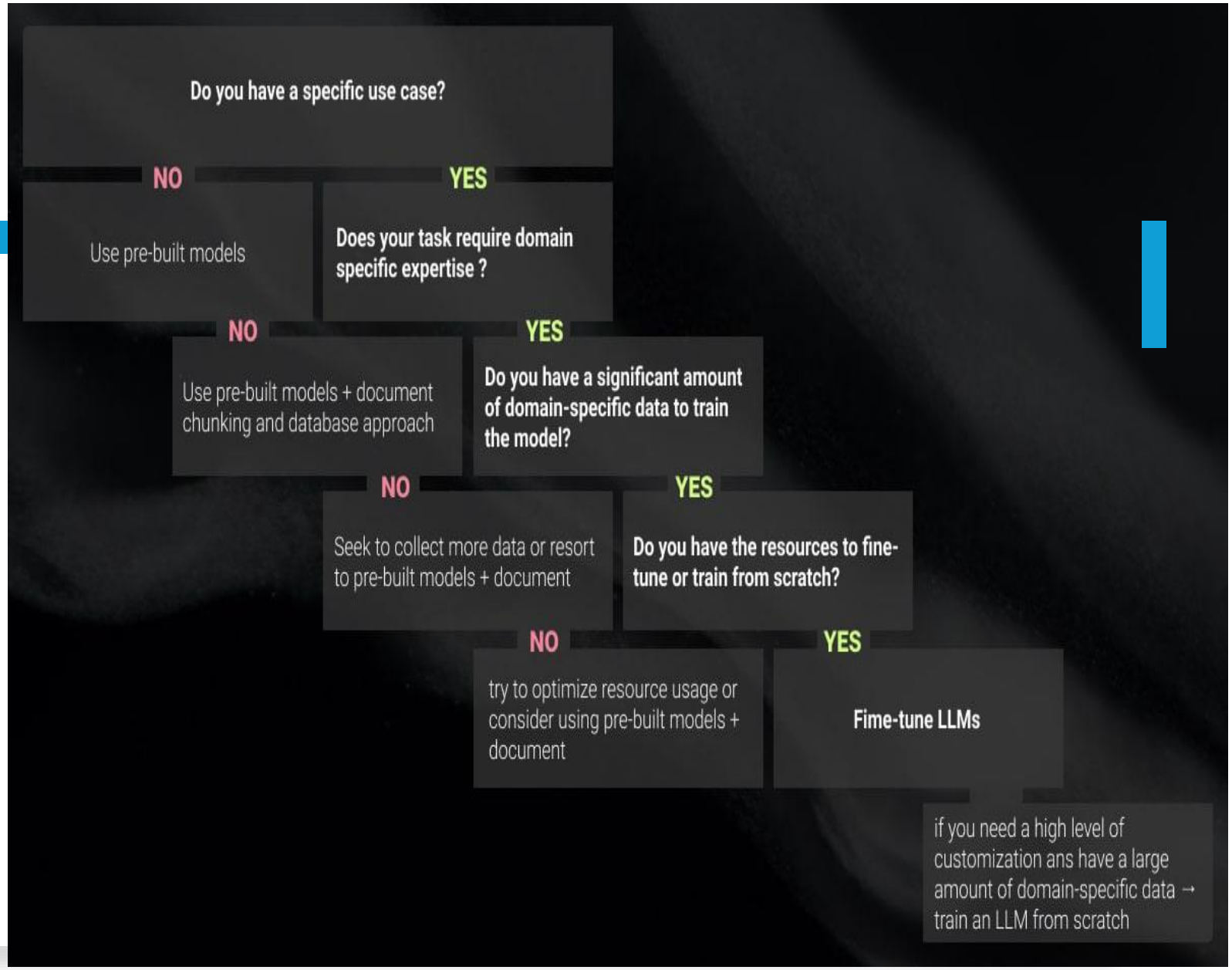
Key Pre-Training Techniques

Technique	Description	Example Models
Masked Language Modeling	Predicts missing words in a given sentence to understand context deeply.	BERT, RoBERTa
Causal Language Modeling	Predicts the next word in a sequence to generate natural, coherent text.	GPT series, GPT-4
Sequence-to-Sequence Learning	Trains models to handle both input and output as text for task flexibility.	T5, BART


Trained on large corpus
task specific data
LLM
Fine Tuning
Final Model

Fine Tuning

- LLM fine-tuning enables models to specific needs.
- While general-purpose LLMs offer broad capabilities, fine-tuning refines these models for particular domains, tasks, or business requirements, providing enhanced accuracy and relevance.



Key Fine-Tuning Techniques

Technique	Description	Example Use Cases
Full Fine-Tuning	Updates all model parameters for task-specific learning.	Sentiment analysis, classification
Parameter-Efficient Methods	Optimizes specific model layers (e.g., LoRA, Adapters) to save computational resources.	Domain-specific tasks, low-resource scenarios
Few-Shot/Zero-Shot Tuning	Leverages minimal (or no) labeled examples, relying on pre-trained model prompts.	Rare language tasks, NLU

Fine Tuning-Instruction Tuning

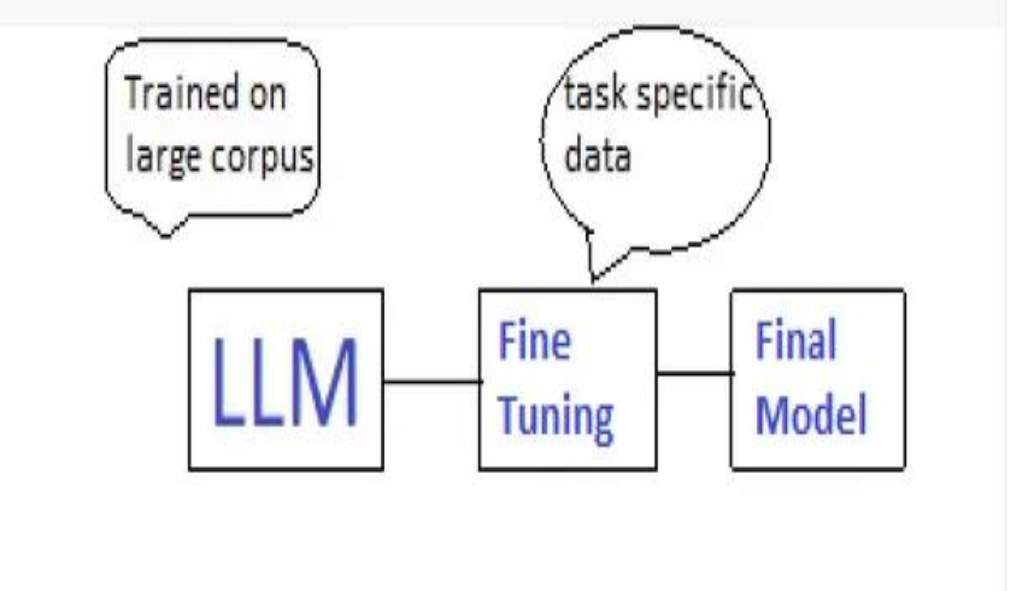
- Instruction tuning emphasizes teaching the model to follow explicit directions and generalize across various tasks
- Eg: *What is the capital of France?,* Teaching a model to summarize an article, translate a sentence, or generate a recipe based on instructions

Fine Tuning-Domain Tuning

- Domain fine-tuning is the process of adapting a pre-trained AI model, such as a large language model (LLM), to a specific domain, like medicine or finance, by further training it on a smaller, domain-specific dataset.
- By grounding the model in domain-specific data, fine-tuning can decrease the likelihood of generating inaccurate or irrelevant information.
- The model learns the terminology, regulations, and stylistic conventions of the target domain.
- Examples : **Legal,Healthcare, Finance**

Parameter efficient Tuning

- PEFT (Parameter-Efficient Fine-Tuning) is a method for adapting large language models (LLMs) to specific tasks by updating only a small fraction of their parameters, rather than the entire model.
- This significantly reduces computational and storage costs, accelerates the fine-tuning process, and prevents "catastrophic forgetting" by preserving the original model's knowledge.
- Techniques like LoRA and QLoRA are popular PEFT methods that introduce small, trainable adapters while freezing most of the pre-trained weights.



PEFT

Adapter layers are added after multi-head attention and feed-forward layers in the transformer architecture. The parameters of these added layers are only updated during fine-tuning while keeping the rest of the parameters frozen.

- LoRA-Low-Rank Adaptation (LoRA)
- QLoRA-Quantized Low-Rank Adaptation

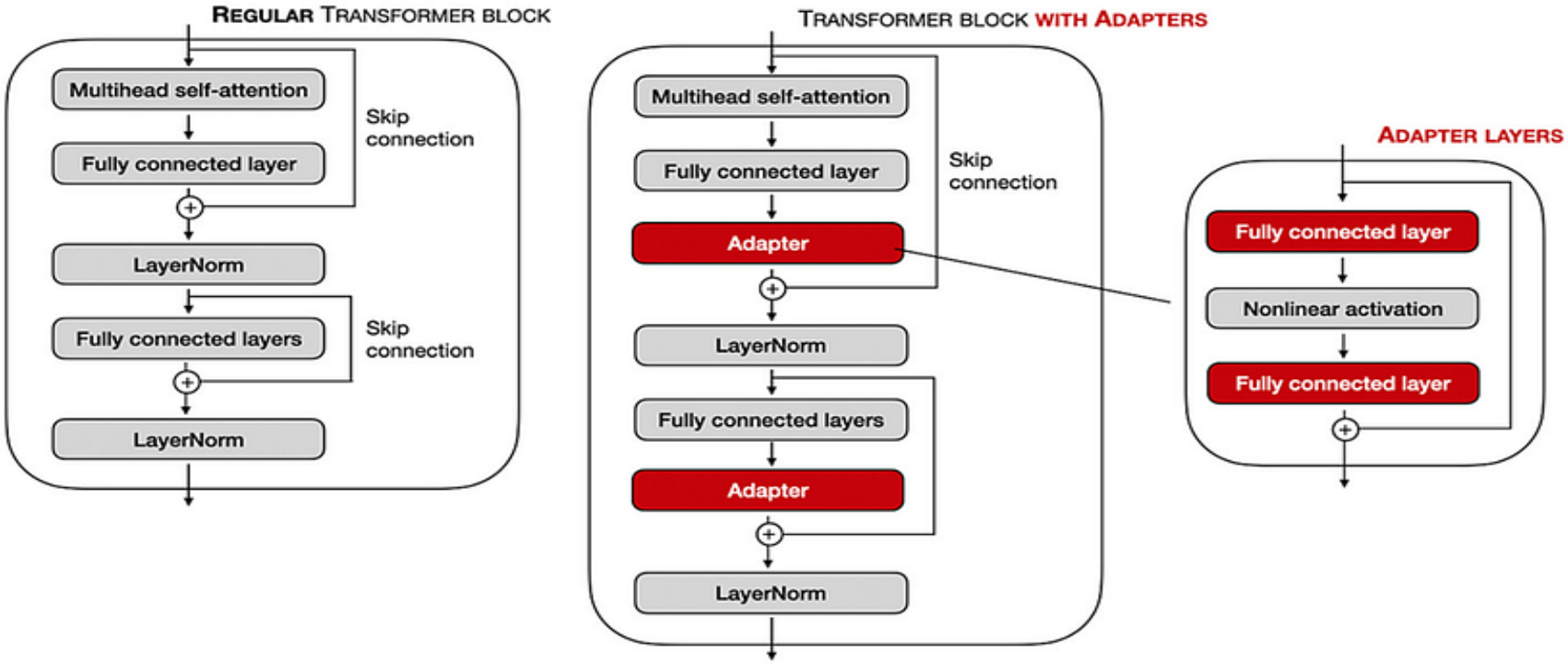


Image source: *https://magazine.sebastianraschka.com/p/finetuning-llms-with-adapters*

steps involved in fine-tuning using PEFT:

- **Data Preparation:** Begin by structuring your dataset in a way that suits your specific task. Define your inputs and desired outputs, especially when working with Falcon 7B.
- **Library Setup:** Install necessary libraries like HuggingFace Transformers, Datasets, BitsandBytes, and WandB for monitoring training progress.
- **Model Selection:** Choose the LLM model you want to fine-tune, like Falcon 7B.
- **PEFT Configuration:** Configure PEFT parameters, including the selection of layers and the 'R' value in LoRA. These choices will determine the subset of coefficients you plan to modify.
- **Quantization:** Decide on the level of quantization you want to apply, balancing memory efficiency with acceptable error rates.
- **Training Arguments:** Define training arguments such as batch size, optimizer, learning rate scheduler, and checkpoints for your fine-tuning process.
- **Fine-Tuning:** Use the HuggingFace Trainer with your PEFT configuration to fine-tune your LLM. Monitor training progress using libraries like WandB.
- **Validation:** Keep an eye on both training and validation loss to ensure your model doesn't overfit.
- **Checkpointing:** Save checkpoints to resume training from specific points if needed.
-

LoRA

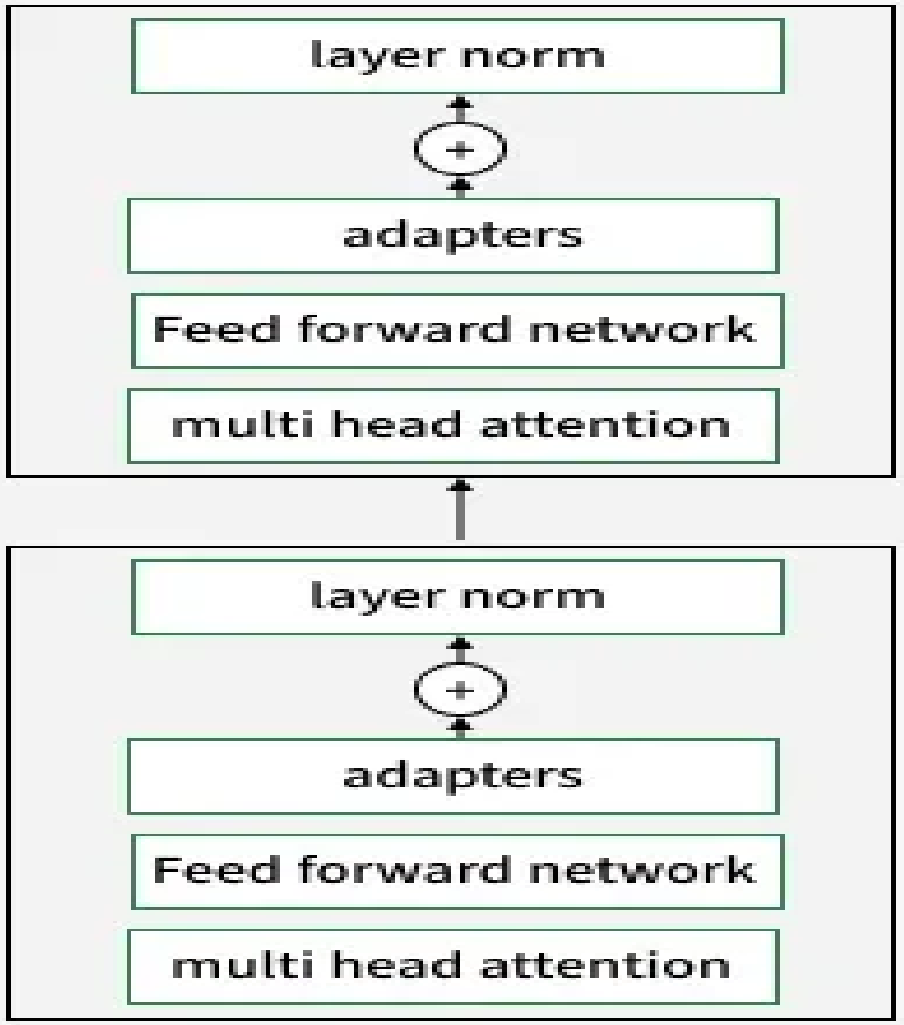
- Low-Rank Adaptation (LoRA) method is a fine-tuning method introduced by a team of Microsoft researchers in 2021.
- **LoRA** is a parameter-efficient fine-tuning method: instead of updating the huge pretrained weights, it introduces small trainable matrices of low rank.

LORA achieves this by:

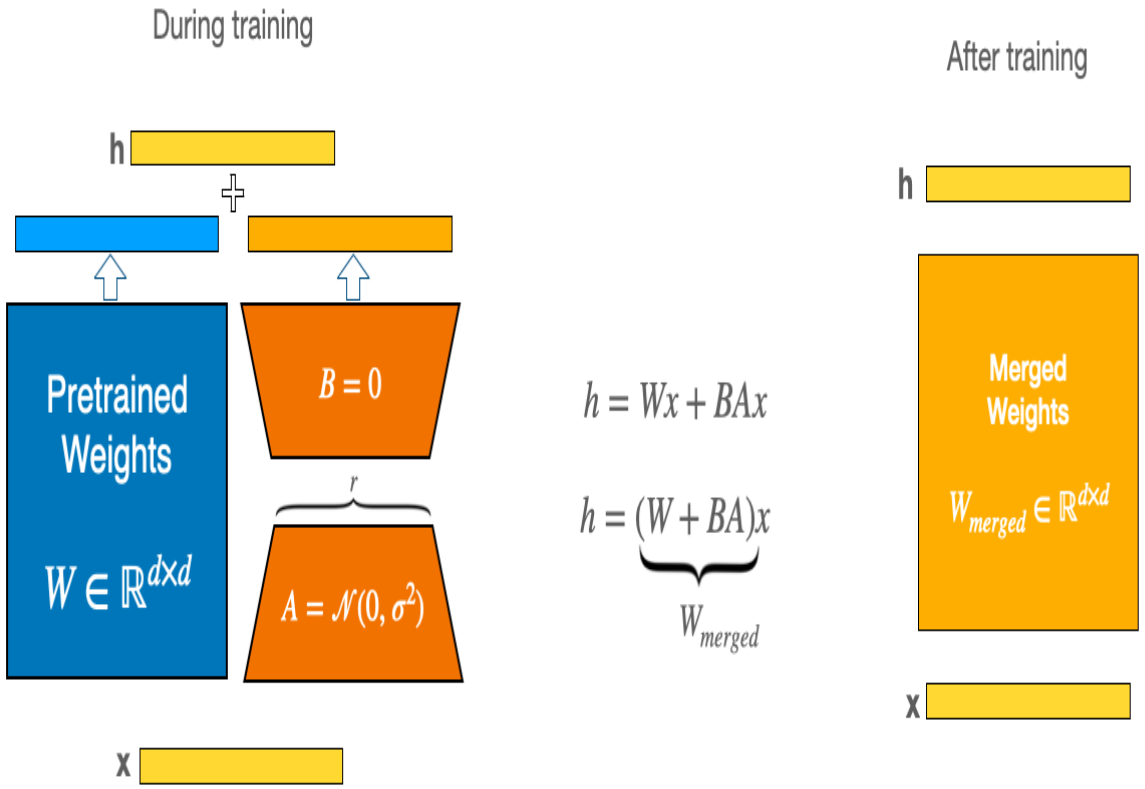
- Tracking changes to weights instead of updating them directly.
- Decomposing large matrices of weight changes into smaller matrices that contain the “trainable parameters.”

LoRA

- Image shows a transformer block architecture where **adapters** are inserted after the feed-forward network and before layer normalization.
- In LoRA, these adapters are implemented as low-rank matrices. During fine-tuning, only these adapter parameters are updated, while the core model weights (multi-head attention, feed-forward network, etc.) stay fixed.

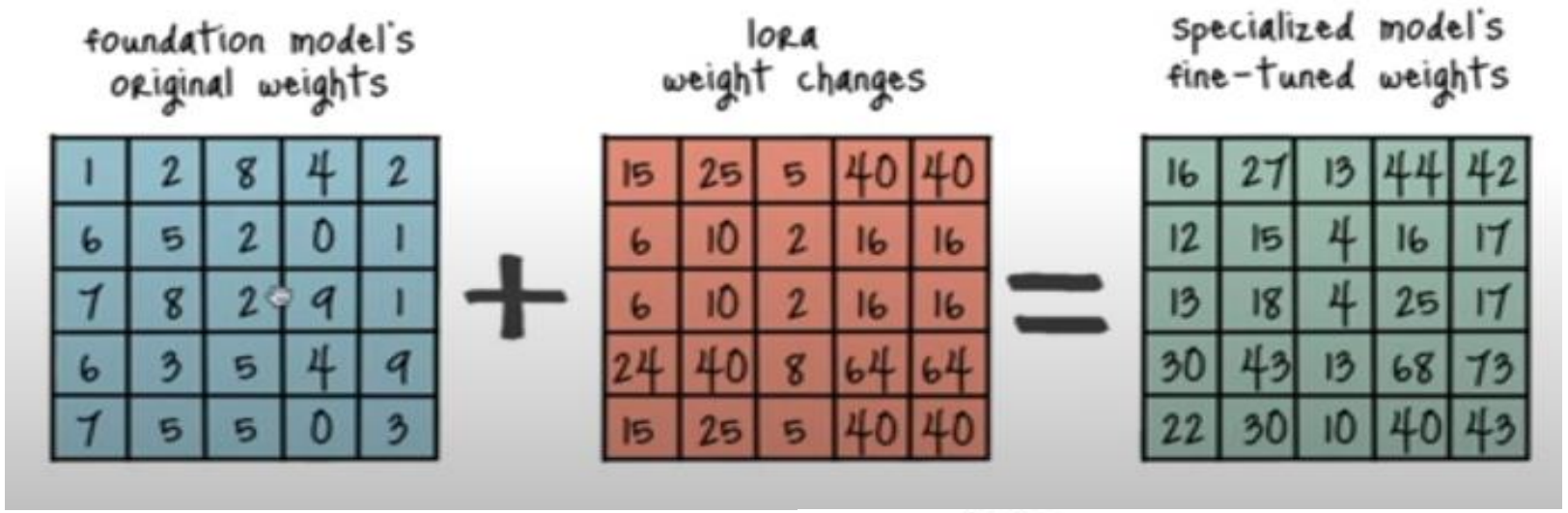


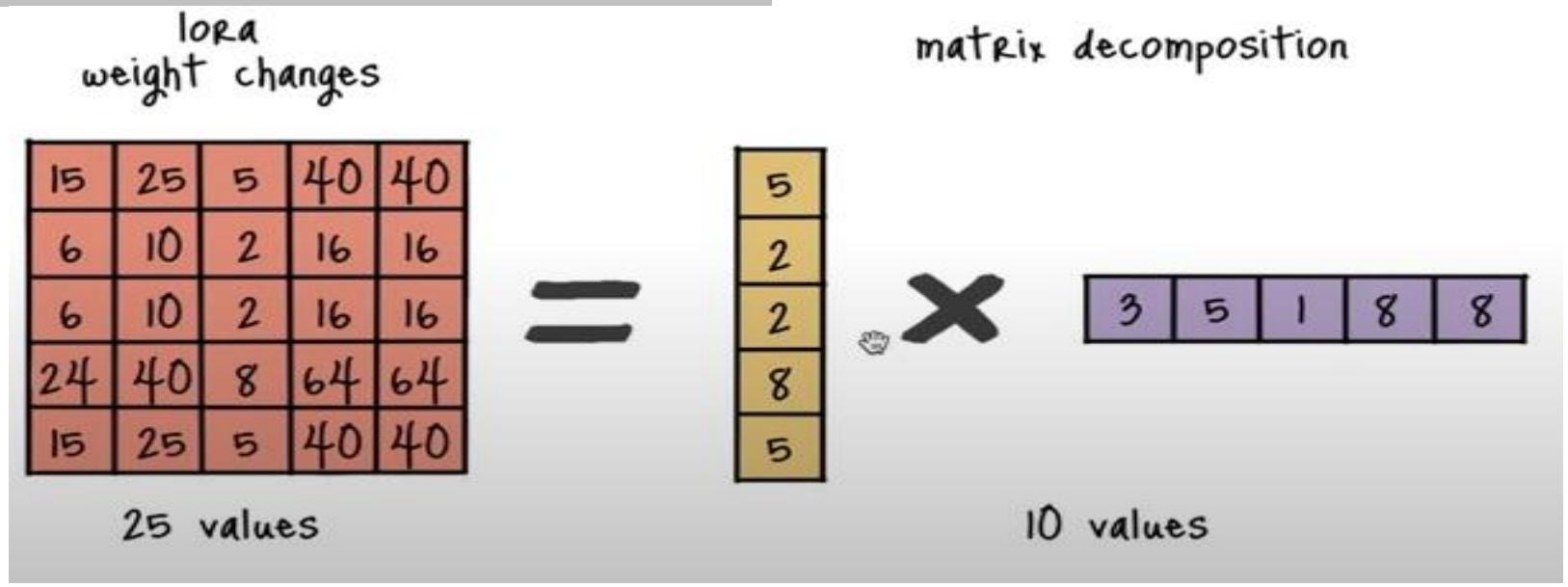
LoRA



- Rather than altering the weight matrix W of a layer in all of its components, LoRA creates two smaller matrices, A and B, whose product roughly represents the modifications to W. The adaptation can be expressed mathematically as Y = W+AB, where A and B are the low-rank matrices.
- If W is an *mxn* matrix A might be *mxr* and B is *rxn* where *r* is rank and much smaller than *m,n*. During fine tuning only A and B are adjusted enabling the model to learn task specific features.

LoRA


foundation model's original weights
lora weight changes
specialized model's fine-tuned weights
1 2 8 4 2
6 5 2 0 1
7 8 2 9 1
6 3 5 4 9
7 5 5 0 3
15 25 5 40 40
6 10 2 16 16
6 10 2 16 16
24 40 8 64 64
15 25 5 40 40
16 27 13 44 42
12 15 4 16 17
13 18 4 25 17
30 43 13 68 73
22 30 10 40 43


lora weight changes
matrix decomposition
15 25 5 40 40
6 10 2 16 16
6 10 2 16 16
24 40 8 64 64
15 25 5 40 40
5
2
2
8
5
3 5 1 8 8
25 values
10 values

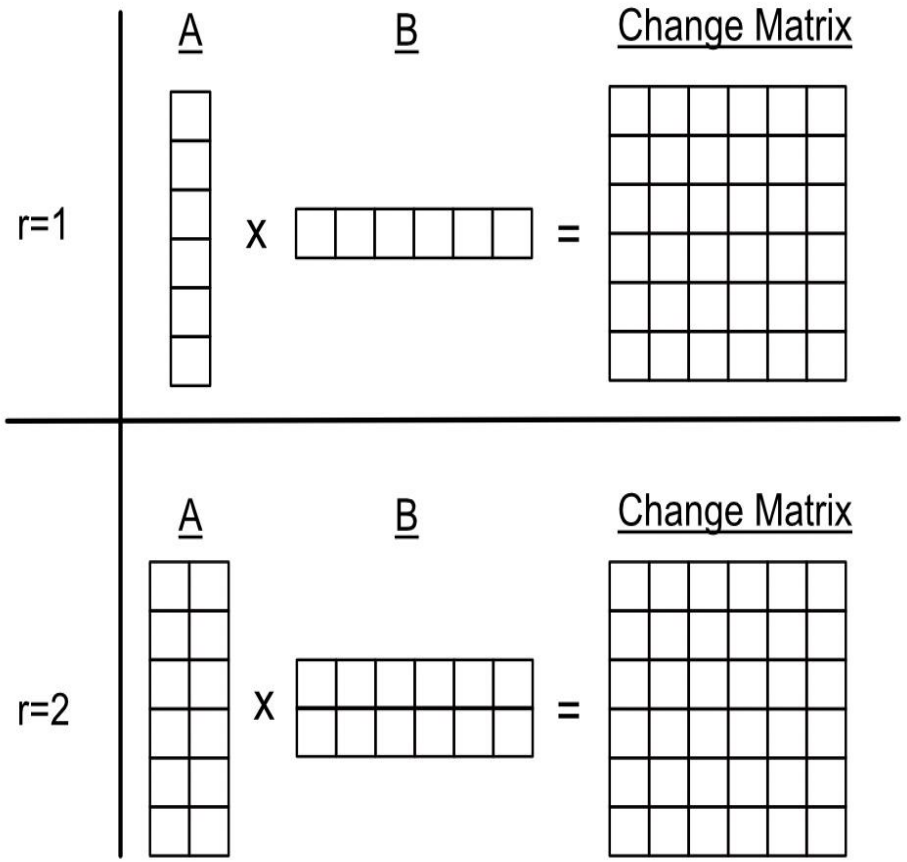
Fine Tuning Using LoRA

number of parameters	original matrix dimensions	lora parameters (rank 1)	savings
25	5 x 5	10	60%
1M	1000 x 1000	2000	99.80%
2B	~ 44k x 44k	~ 89k	99.995%
7B	~ 83k x 83k	~ 167k	99.997%
13B	~ 114k x 114k	~ 228K	99.998%

Research paper Link

chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://arxiv.org/pdf/2106.09685

LoRA



A conceptual diagram of LoRA with an r value equal to 1 and 2. In both examples the decomposed A and B matrices result in the same sized change matrix, but r=2 is able to encode more linearly independent information into the change matrix, due to having more information in the A and B matrices

QLoRA

- QLoRA is the extended version of LoRA which works by quantizing the precision of the weight parameters in the pre trained LLM to 4-bit precision.
- Typically, parameters of trained models are stored in a 32-bit format, but QLoRA compresses them to a 4-bit format.
- This reduces the memory footprint of the LLM, making it possible to finetune it on a single GPU. This method significantly reduces the memory footprint, making it feasible to run LLM models on less powerful hardware, including consumer GPUs.

QLoRA

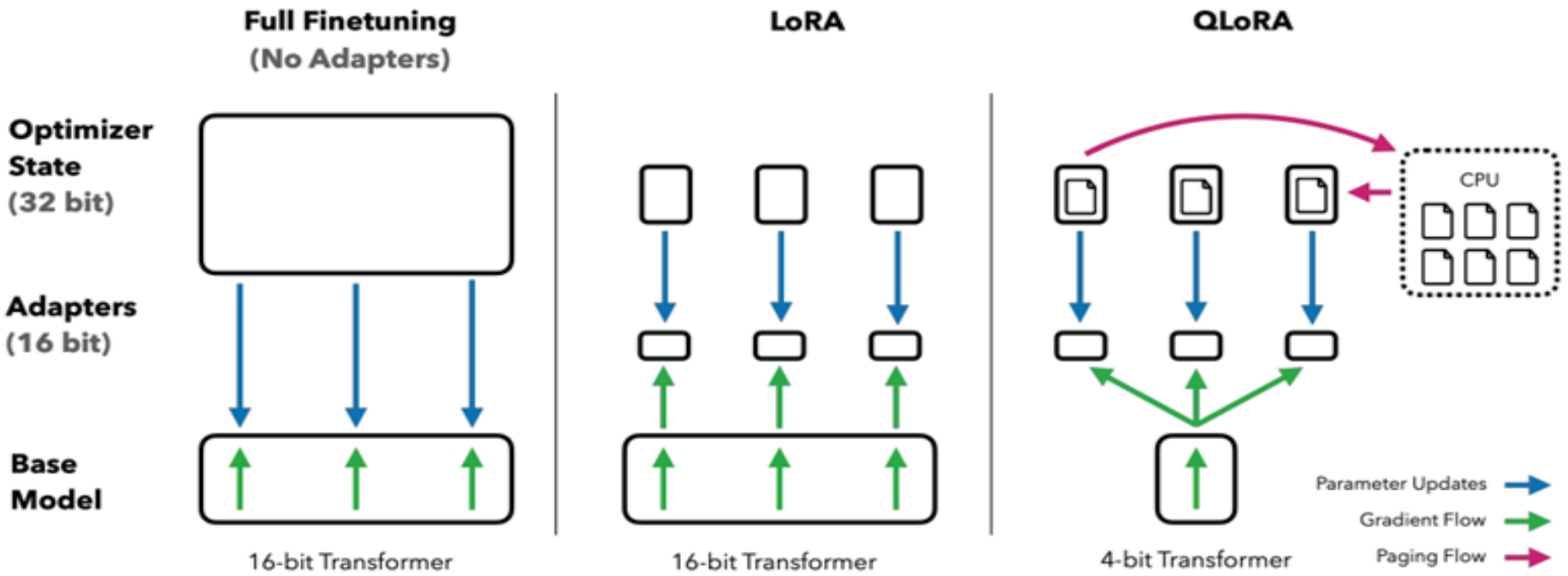


Figure 1: Different finetuning methods and their memory requirements. QLoRA improves over LoRA by quantizing the transformer model to 4-bit precision and using paged optimizers to handle memory spikes.

Working of QLoRA

- QLoRA fine-tunes large language models efficiently by following a structured process. Each step focuses on reducing memory and computation while adapting the model to a specific task

Quantize the Base Model

- Pretrained model is converted from full precision to 4-bit weights.
- This reduces GPU memory usage allowing large models to run on smaller hardware. Quantization methods like NF4 help maintain accuracy during compression.
- Quantization is a technique that is helpful in reducing the size of the model by converting high precision data to low precision. In simple terms, it converts datatype of high bits to fewer bits. For example, converting FP32 to 8-bit Integers is a quantization technique.

Add Low-Rank Adapters

- Small adapter layers are inserted into selected parts of the model, typically the attention layers.
- These adapters remain in higher precision (e.g 16-bit) to ensure stable training.
- The backbone model is kept frozen, so the original weights are not modified.

Working of QLoRA

- **Fine-Tune Only the Adapters**
- During training, only the adapter layers are updated.
- This drastically reduces the number of trainable parameters and the required computation.
- Fine-tuning becomes faster and feasible on a single GPU or low-resource device.
- **Merge or Keep Adapters Separate**
- After training, adapters can be merged into the quantized model for deployment.
- Alternatively, it can be kept separate allowing reuse or swapping for different tasks without retraining the base model.
- Reference:https://towardsdatascience.com/tensor-quantization-the-untold-story-d798c30e7646/

Prompt engineering

- **Prompt engineering** is the art of asking the right question to get the best output from an LLM. It enables direct interaction with the LLM using only plain language prompts.
- . Crafting effective prompts often requires an understanding of underlying architectures of the model.Often used to finetune the model without changing the underlying architecture or training data.
- **Training data and tokenization**. LLMs are trained on vast datasets, tokenizing input data into smaller chunks (tokens) for processing. The choice of tokenization (word-based, byte-pair, etc.) can influence how a model interprets a prompt. For instance, a word tokenized differently might yield varied outputs.
- **Model parameters**. LLMs have millions, if not billions, of parameters. These parameters, fine-tuned during the training process, determine how the model responds to a prompt. Understanding the relationship between these parameters and model outputs can aid in crafting more effective prompts.
- **Loss functions and gradients**. At a deeper level, the model's behavior during prompt response is influenced by its loss functions and gradients. These mathematical constructs guide the model's learning process. While prompt engineers don't typically adjust these directly, understanding their impact can provide insights into model behavior

Types of Prompt

- Lack of prompt engineering leads to incorrect response called as hallucinations
- What you ask is what you get.

Types of prompt

- Explicit Prompt
- Conversational Prompt(interaction with chatbot)
- Instructional Prompts(give instructions-write in one page)
- Context Based Prompts(suggest tourist attraction in Paris based on my planned trip next month)
- Open Ended Prompts (tell me about impact of AI)
- Bias mitigating prompt(generate a balanced and objective view of student reservation in India, avoid any opinion , favor to any group)
- Code Generation Prompt

Key elements of a prompt

- **Instruction**. This is the core directive of the prompt. It tells the model what you want it to do. For example, "Summarize the following text" provides a clear action for the model.
- **Context.** Context provides additional information that helps the model understand the broader scenario or background. For instance, "Considering the economic downturn, provide investment advice" gives the model a backdrop against which to frame its response.
- **Input data**. This is the specific information or data you want the model to process. It could be a paragraph, a set of numbers, or even a single word.
- **Output indicator**. This element guides the model on the format or type of response desired. For instance, "In the style of Shakespeare, rewrite the following sentence" gives the model a stylistic direction.

Techniques of Prompts

- **Direct prompting (Zero-shot)**
- **One-shot**
- **Few Shot**

Direct prompting (Zero-shot)

- Direct prompting (also known as Zero-shot) is the simplest type of prompt.
- It provides no examples to the model, just the instruction. You can also phrase the instruction as a question, or give the model a "role”.
- Provide: Instruction, Some context, Idea Generation
- Prompt: Write a poem on Universe.

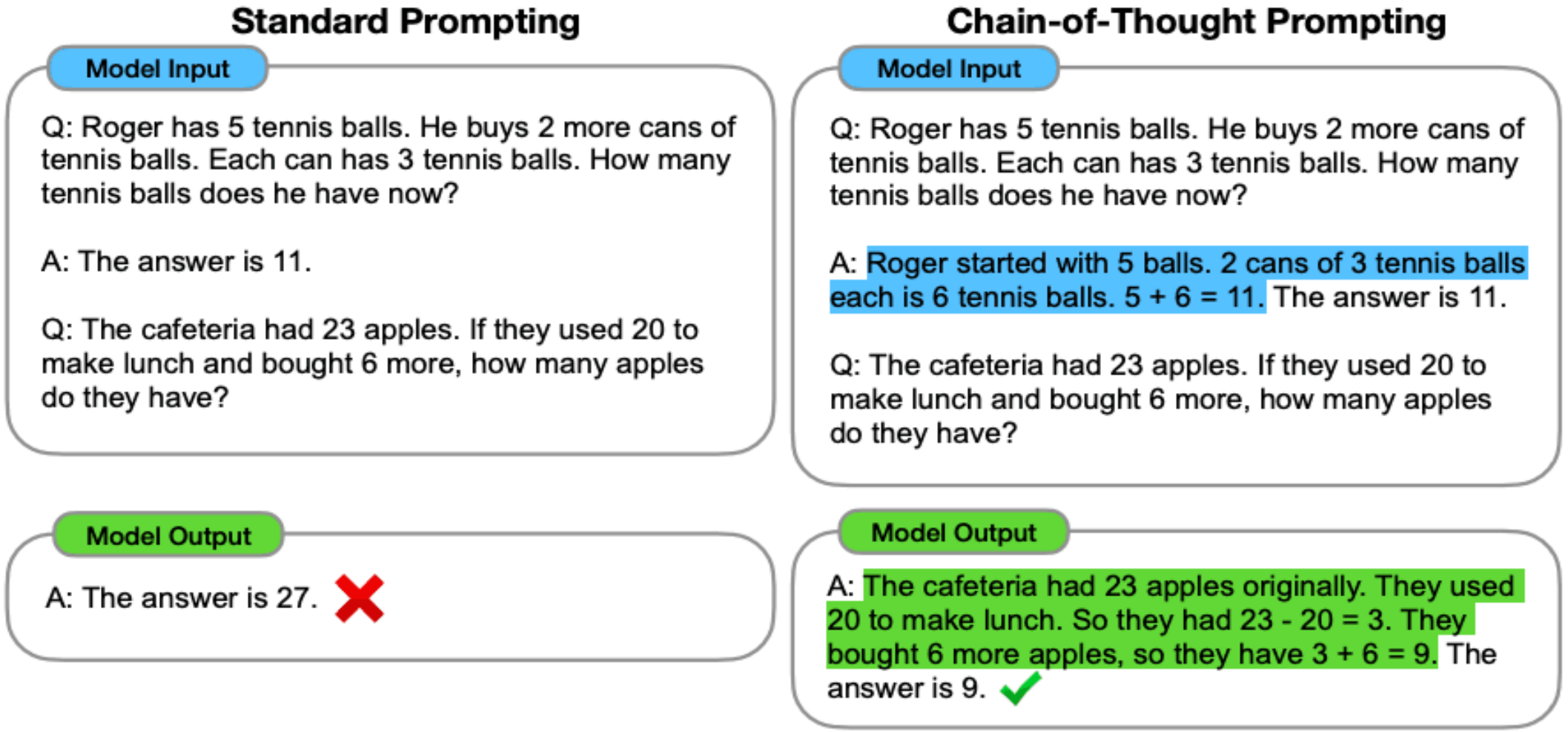
- **Few-shot prompting**/in-context learning. Here, the model is given a few examples (shots) to guide its response. By providing context or previous instances, the model can better understand and generate the desired output.
- For example, showing a model several examples of translated sentences before asking it to translate a new one.

Chain of Thought

- **Chain-of-Thought (CoT) Prompting**
- CoT) prompting enables complex reasoning capabilities through intermediate reasoning steps. We can combine it with few-shot prompting to get better results on more complex tasks that require reasoning before responding.
- Generates intermediate reasoning step before arriving to solution.
- Used in Mathematical problem solving
- Working with code and equation
- Decision making
- It is guiding model with context and intermediate step
- Eg: Describe the answer in step
- Reference:

https://arxiv.org/abs/2201.11903

CoT


Standard Prompting
Model Input
Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?
A: The answer is 11.
Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?
Model Output
A: The answer is 27.
Chain-of-Thought Prompting
Model Input
Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?
A: Roger started with 5 balls. 2 cans of 3 tennis balls each is 6 tennis balls. 5 + 6 = 11. The answer is 11.
Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?
Model Output
A: The cafeteria had 23 apples originally. They used 20 to make lunch. So they had 23 - 20 = 3. They bought 6 more apples, so they have 3 + 6 = 9. The answer is 9.

Zero-shot COT Prompting

- Generates step by step reasoning without prior example
- Steps
- Reasoning Extraction
- Answer Extraction

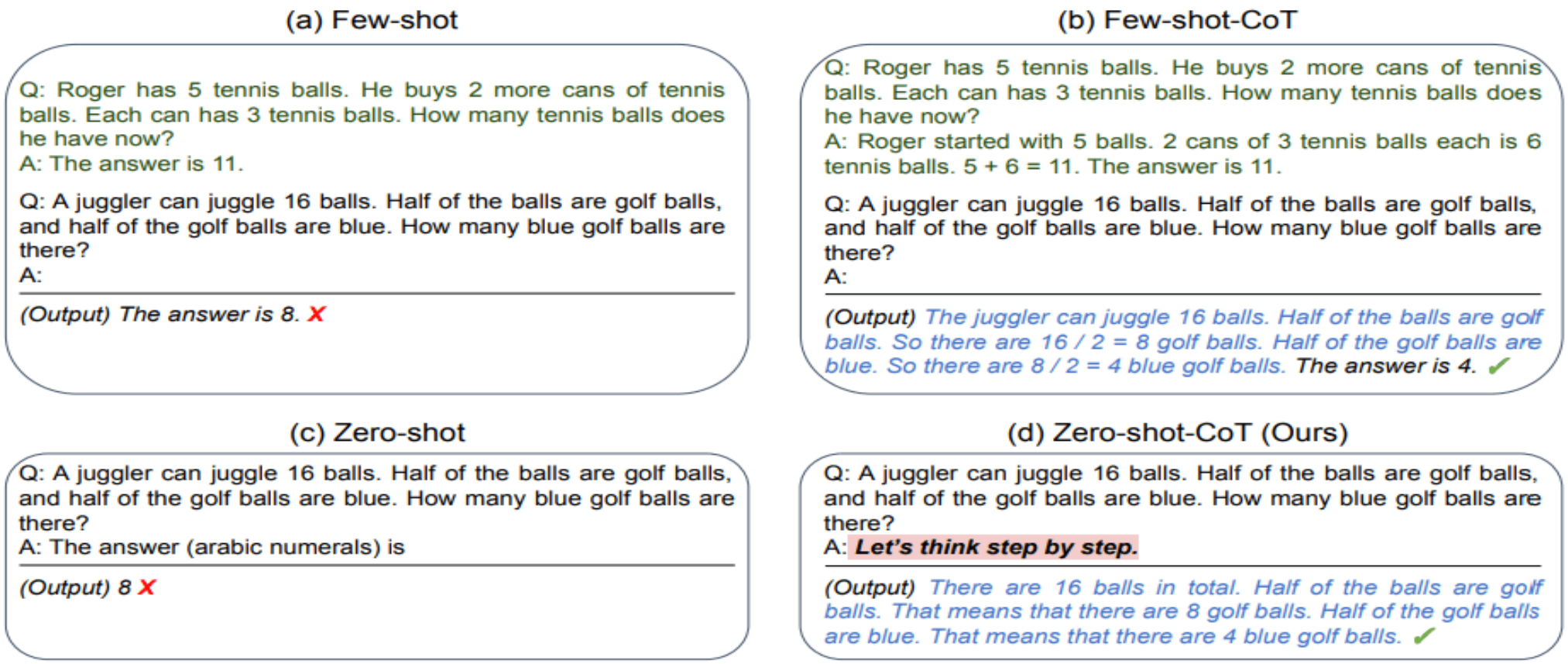
Different Aprroaches

Zero shot prompting

➢Reasoning Extraction

➢Answer Extraction

CoT


(a) Few-shot
Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?
A: The answer is 11.
Q: A juggler can juggle 16 balls. Half of the balls are golf balls, and half of the golf balls are blue. How many blue golf balls are there?
A:
(Output) The answer is 8. ✗
(b) Few-shot-CoT
Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?
A: Roger started with 5 balls. 2 cans of 3 tennis balls each is 6 tennis balls. 5 + 6 = 11. The answer is 11.
Q: A juggler can juggle 16 balls. Half of the balls are golf balls, and half of the golf balls are blue. How many blue golf balls are there?
A:
(Output) The juggler can juggle 16 balls. Half of the balls are golf balls. So there are 16 / 2 = 8 golf balls. Half of the golf balls are blue. So there are 8 / 2 = 4 blue golf balls. The answer is 4. ✓
(c) Zero-shot
Q: A juggler can juggle 16 balls. Half of the balls are golf balls, and half of the golf balls are blue. How many blue golf balls are there?
A: The answer (arabic numerals) is
(Output) 8 ✗
(d) Zero-shot-CoT (Ours)
Q: A juggler can juggle 16 balls. Half of the balls are golf balls, and half of the golf balls are blue. How many blue golf balls are there?
A: Let's think step by step.
(Output) There are 16 balls in total. Half of the balls are golf balls. That means that there are 8 golf balls. Half of the golf balls are blue. That means that there are 4 blue golf balls. ✓

CoT

- **Manual CoT**

➢Manually provide detailed step by step examples(manual demos) reasoning for LLM

➢LLM uses these examples to solve new problems.

➢Used to ensure Logical consistency.Eg: Business, Finance

➢Expertise required to create manual examples. Demands deep subject knowledge

Auto CoT

➢Automatically generate example demonstration that LLM can use , without requiring manual effort.

➢First a)perform **Question clustering**, then perform b)**creating example answer,** then c)**use examples to perform new question**

CoT

- **Multimodal CoT**

➢ **Uses both text and images.**

➢ **Rationale Generation (textual Reasoning)**

➢ **Answer inference**

➢ **Eg: will these magnet attract or repel**

➢ **Context:Two magenet are placed together. Hint. Magnet attract pull together. Repel push apart**

Output:

Rationale: Gives reason

Answer: It will attract

ReAct

- ReAct is a general paradigm that combines reasoning and acting with LLMs.
- ReAct prompts LLMs to generate verbal reasoning traces and actions for a task.
- This allows the system to perform dynamic reasoning to create, maintain, and adjust plans for acting while also enabling interaction to external environments (e.g., Wikipedia) to incorporate additional information into the reasoning.

ReAct

(1d) ReAct (Reason + Act)

Thought 1: I need to search Apple Remote and find the program it was originally designed to interact with.
Act 1: Search[Apple Remote]
Obs 1: The Apple Remote is a remote control introduced in October 2005 by Apple … originally designed to control the Front Row media center program …

Thought 2: Apple Remote was originally designed to control the Front Row media center program. I need to search Front Row next and find what other device can control it.
Act 2: Search[Front Row]
Obs 2: Could not find [Front Row]. Similar: ['Front Row Seat to Earth', 'Front Row Motorsports',' Front Row (software) ', …]

Thought 3: Front Row is not found. I need to search Front Row (software) .
Act 3: Search[Front Row (software)]
Obs 3: Front Row is a discontinued media center software …

Thought 4: Front Row (software) is controlled by an Apple Remote or the keyboard function keys. So the answer is keyboard function keys.
Act 4: Finish[keyboard function keys]

- In the example above, we pass a prompt like the following question from
- Aside from the Apple Remote, what other devices can control the program Apple Remote was originally designed to interact with?
- Note that in-context examples are also added to the prompt but we exclude that here for simplicity. We can see that the model generates *task solving trajectories* (Thought, Act). Obs corresponds to observation from the environment that's being interacted with (e.g., Search engine). In essence, ReAct can retrieve information to support reasoning, while reasoning helps to target what to retrieve next.