AMERICAN UNIVERSITY OF BEIRUT

Fall 2024

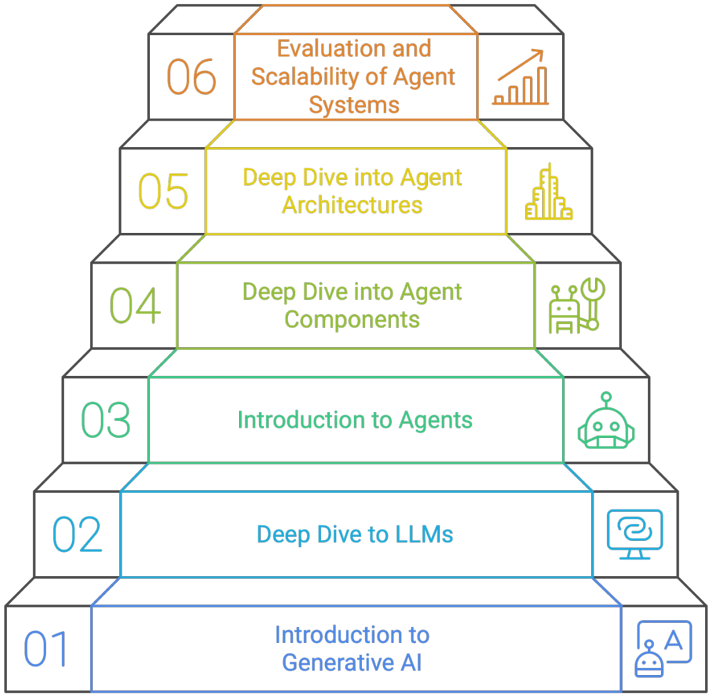
# EECE 503P/798S: Agentic Systems

## C1 - Introduction to GenAI

# Lesson Objectives

- Differentiate between **discriminative** and **generative** AI.
- Explore **real-world applications** of generative AI.
- Learn about the **evolution of generative AI** models (from Markov chains to LLMs).
- Introduce **foundational generative models** such as GANs and VAEs.
- Understand the **Transformer architecture** and the attention mechanism that enabled modern GenAI

# Course Timeline

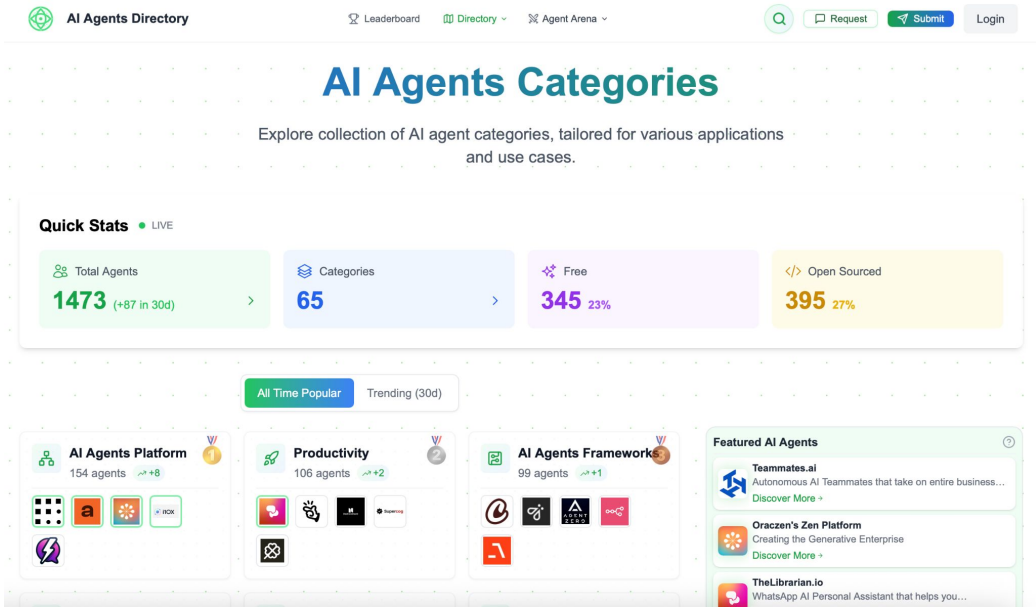



We are here

# Introduction

# What are Agents? Why are they important?

- AI Agents are computer programs that can **execute** tasks
- These tasks can be **broad** or **specialized**
- Using AI to **automate** tasks, workflows, or even jobs has become increasingly **popular**



# AI Agent Popularity

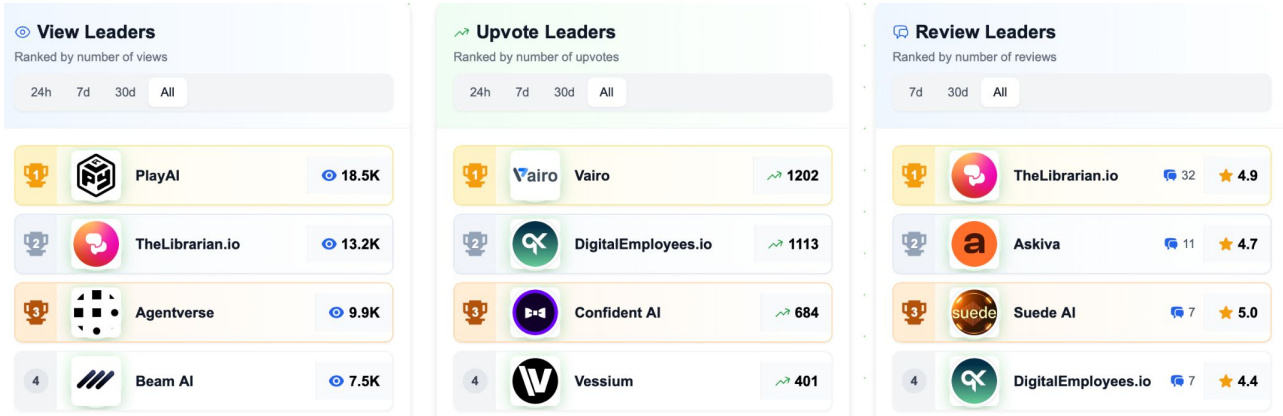
- This fascination with AI Agents is not only a hype, it is a **paradigm shift**.
- AI agents can be integrated in almost **every industry**

The AI Agents Market size was valued at USD 5.25 billion in 2024 and is projected to grow from USD 7.84 billion in 2025 to USD 52.62 billion by 2030 at a CAGR of 46.3% during the forecast period. This rapid growth is fueled by the increasing role of foundational models in improving AI agents.

*https://www.marketsandmarkets.com/Market-Reports/ai-agents-market-15761548.html*

# Where did it come from?

- AI Agents are not a new concept. They have been around for **decades**.
- What shifted the game was not the **concept** or **idea** itself, it was the underlying technology
- As **generative AI** models algorithms **increased** in size, power, and capability, we were able to equip agents with a much more powerful ‘brain’.



# Discriminative VS Generative AI

# Reminder: Types of AI

**GenAI**: A class of deep learning algorithms that can generate new content based on its training data

**DL**: A sub-division of ML that uses neural networks as the statistical model

**ML**: A division of artificial intelligence that uses statistical models as the program

**AI**: Any program that performs a task traditionally only humans could do

# Discriminative Vs Generative AI

## Discriminative AI

Focuses on **classifying** or **labeling** data. It **predicts** outputs from given inputs.

- **Examples:** Spam detection, fraud detection
- Learn **boundaries** between classes

## Generative AI

Creates **new content** by learning data patterns and structures. Produces novel data instances.

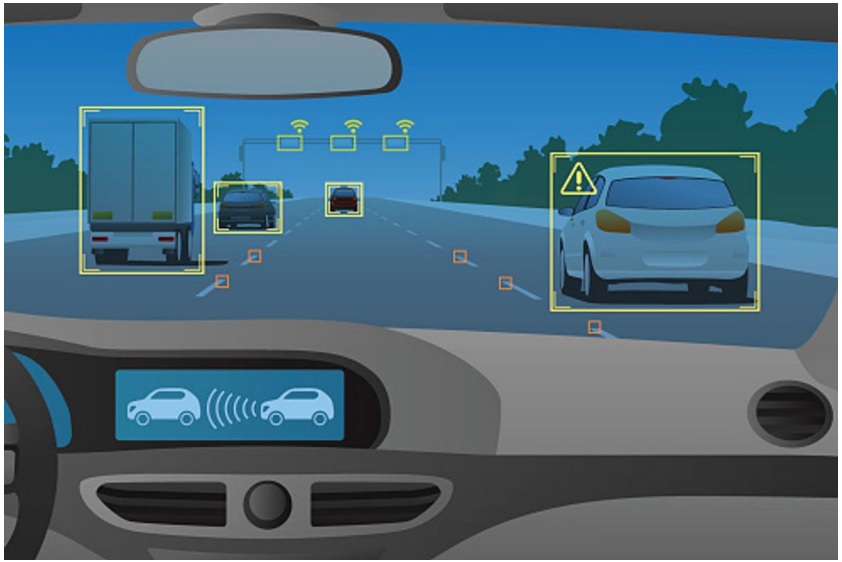
- **Examples:** Text, images, music generation
- Models **data distribution** itself

# Recap: Examples of Discriminative AI Models

1. **Linear Regression**
   Predicts continuous outcomes. It fits a linear equation to data.
2. **Logistic Regression**
   Classifies data into categories. It uses a sigmoid function.
3. **Support Vector Machines**
   Finds optimal hyperplanes. Separates data points into classes.
4. **Decision Trees**
   Uses a tree-like model. Classifies by splitting data on features.
5. **Feedforward Neural Networks**
   Processes data through layers. Learns complex non-linear relationships.

# Discriminative AI in the real world

**Self driving** cars use discriminative computer vision models to **comprehend** their surroundings and make more accurate **decisions** on their next move

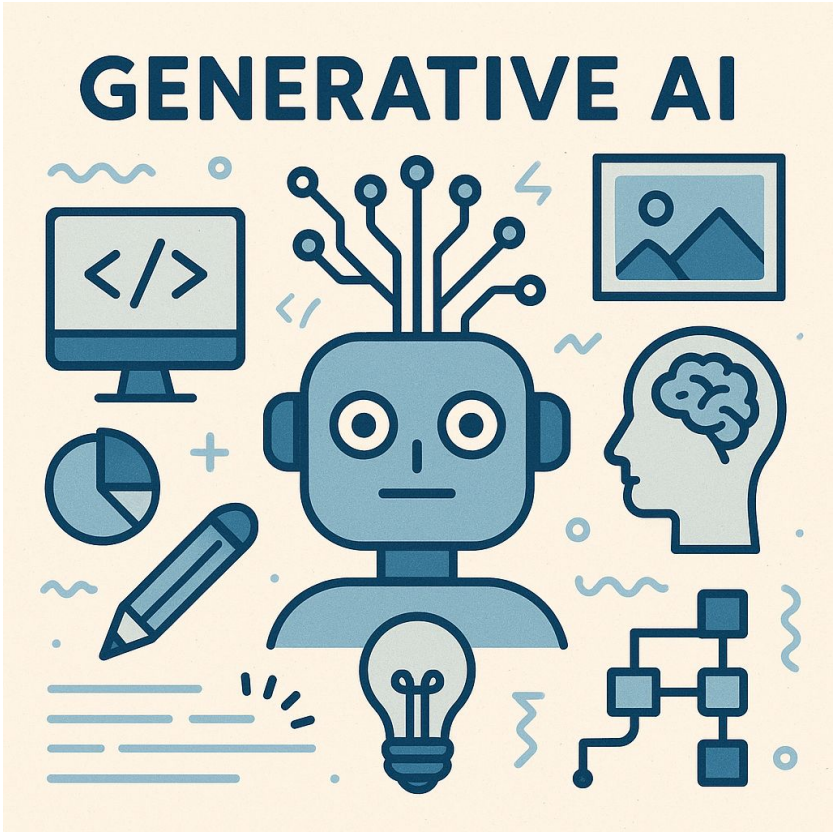


# Generative AI In the Real World

# Generative AI

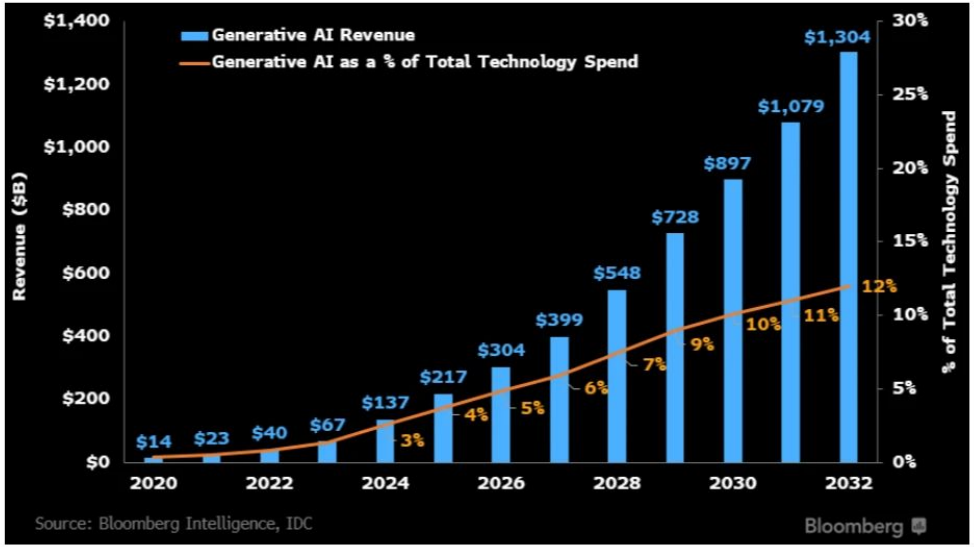
# Generative AI

**Definition**: Generative AI is a field of artificial intelligence where models can create new content, such as text, **images**, **audio**, or **video**, based on the data it has been trained on.



# The GenAI Hype

- The generative AI market is experiencing **rapid growth**
- The global market size was valued at $43.87 billion in **2023** and is projected to reach $967.65 billion by **2032**

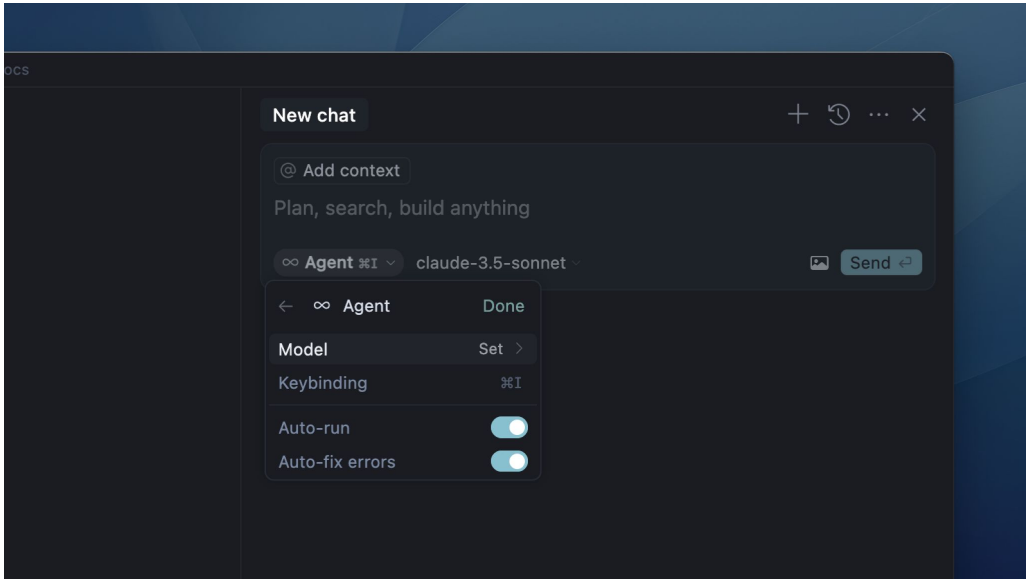


# Applications of Generative AI

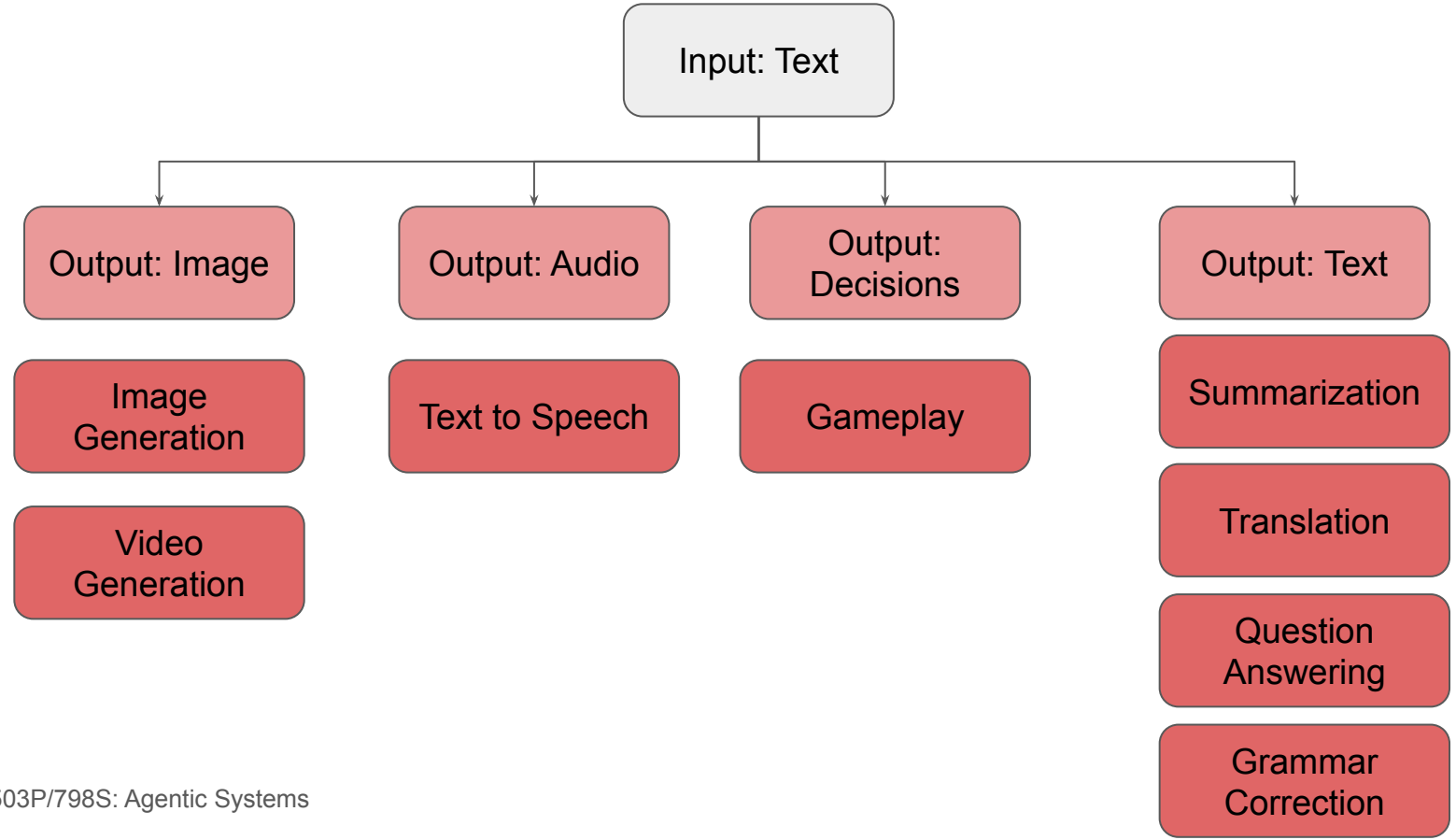
- **Agentforce** by Salesforce is an **AI-powered platform** designed to **create** and **manage** autonomous agents that enhance various business operations
- These agents operate **24/7**
- Providing **proactive support** to both employees and customers

# Applications of Generative AI

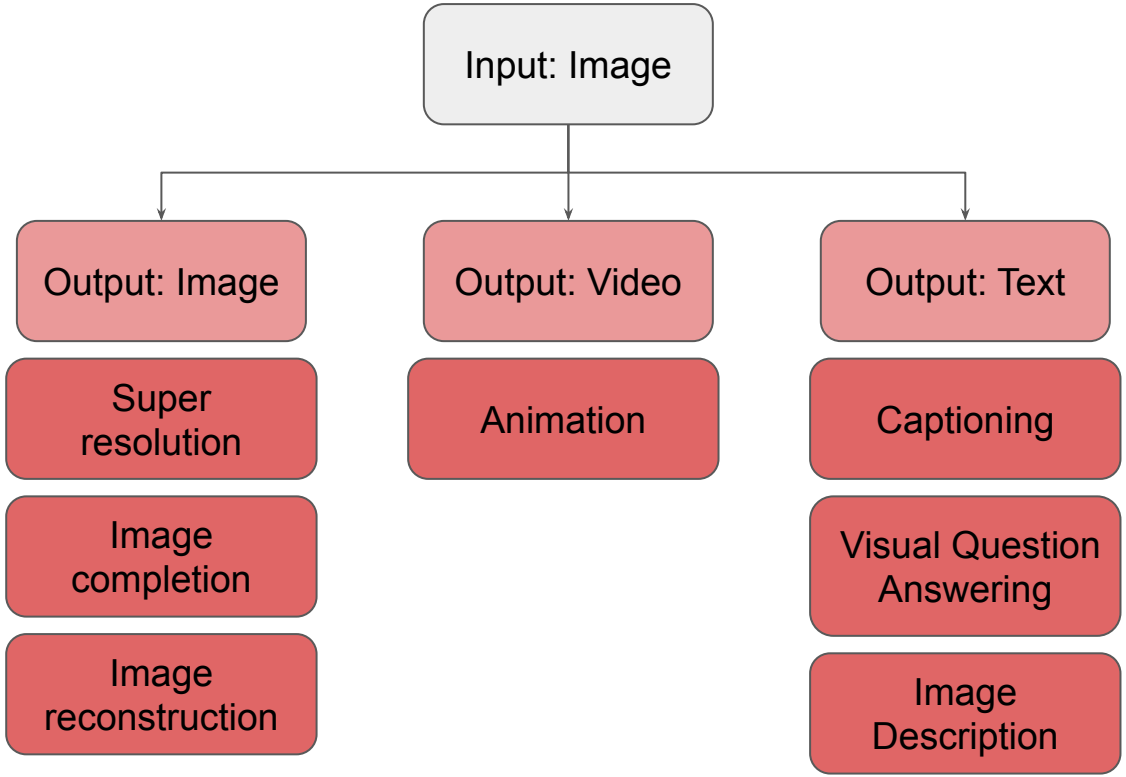
- Agent is the **default** and **most autonomous** mode in Cursor
- It has all tools enabled to **autonomously** explore your codebase, read documentation, and run terminal commands to complete tasks efficiently



# Generative AI: Text Input

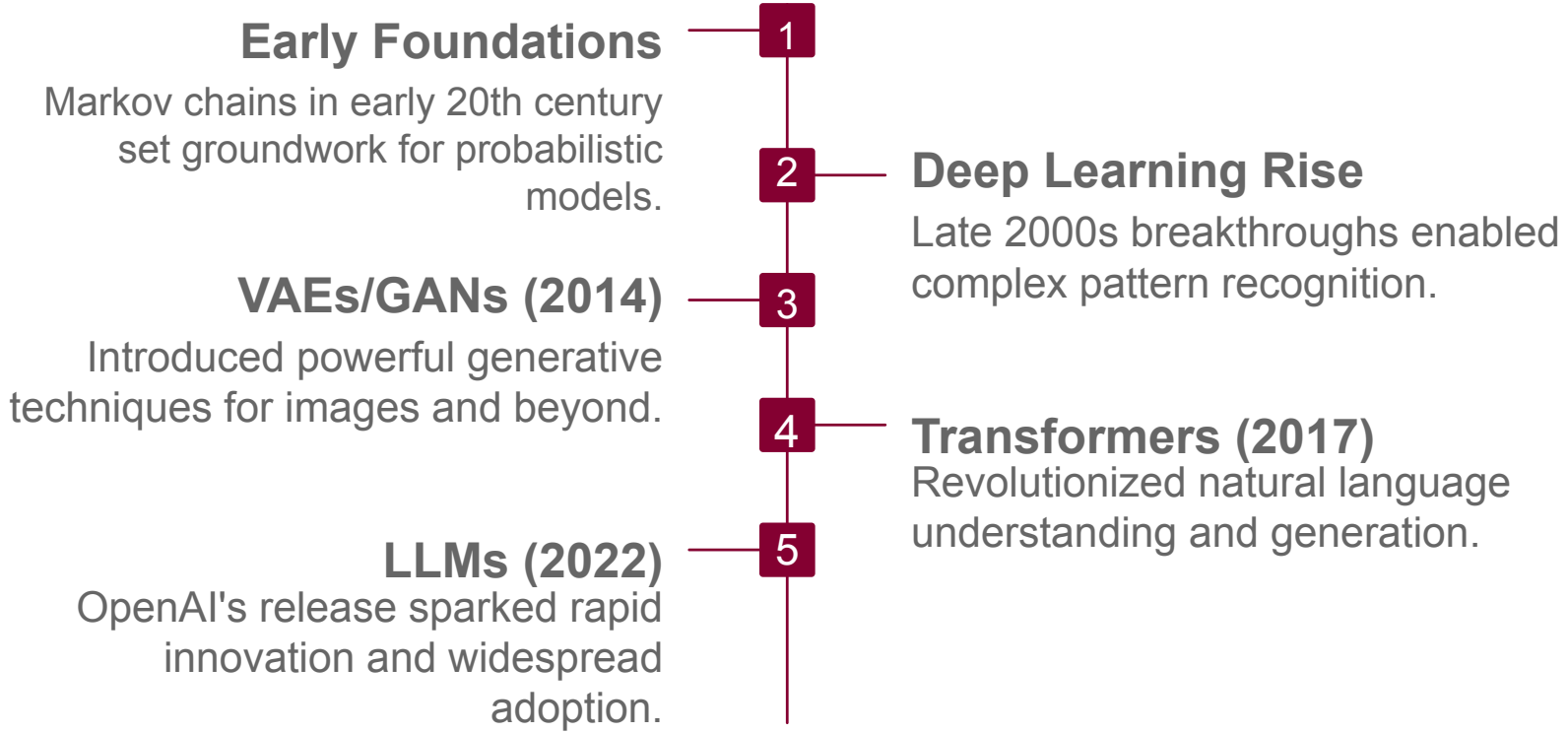


# Generative AI: Image Input

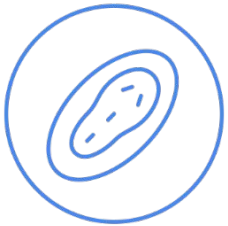


# Generative Models

# Evolution of Generative AI Models



# Types of Generative Models



**Explicit Models**

An explicit model learns the data distribution from the sample distribution, and generates a new type of data. These types of models have access to a probability distribution, and they're trained using maximum likelihood.

VS

**Implicit Models**

Implicit models don't learn the distribution of the data, but rather learn the statistical properties of the data, so it can generalise and generate new samples of data without depending on the probability distribution.

# Explicit Models

Explicit models learn an **explicit probability distribution** of the data. They try to model the likelihood directly and can sample from that distribution.

Examples:

- Gaussian Mixture Models (GMMs): They explicitly assume data comes from a mixture of Gaussian distributions and estimate their parameters.
- Hidden Markov Models (HMMs): Explicitly model the probability distribution of sequences with hidden states.
- Naive Bayes Classifier: Uses Bayes' theorem and explicitly models conditional probabilities of features given a class.
- Variational Autoencoders (VAEs): Use an explicit latent variable model with a likelihood function, trained via maximum likelihood (variational inference).

# Implicit Models

Implicit models don’t define an explicit probability distribution. Instead, they learn **how to generate data that looks like the training data**, without ever specifying the exact underlying distribution.

Examples:

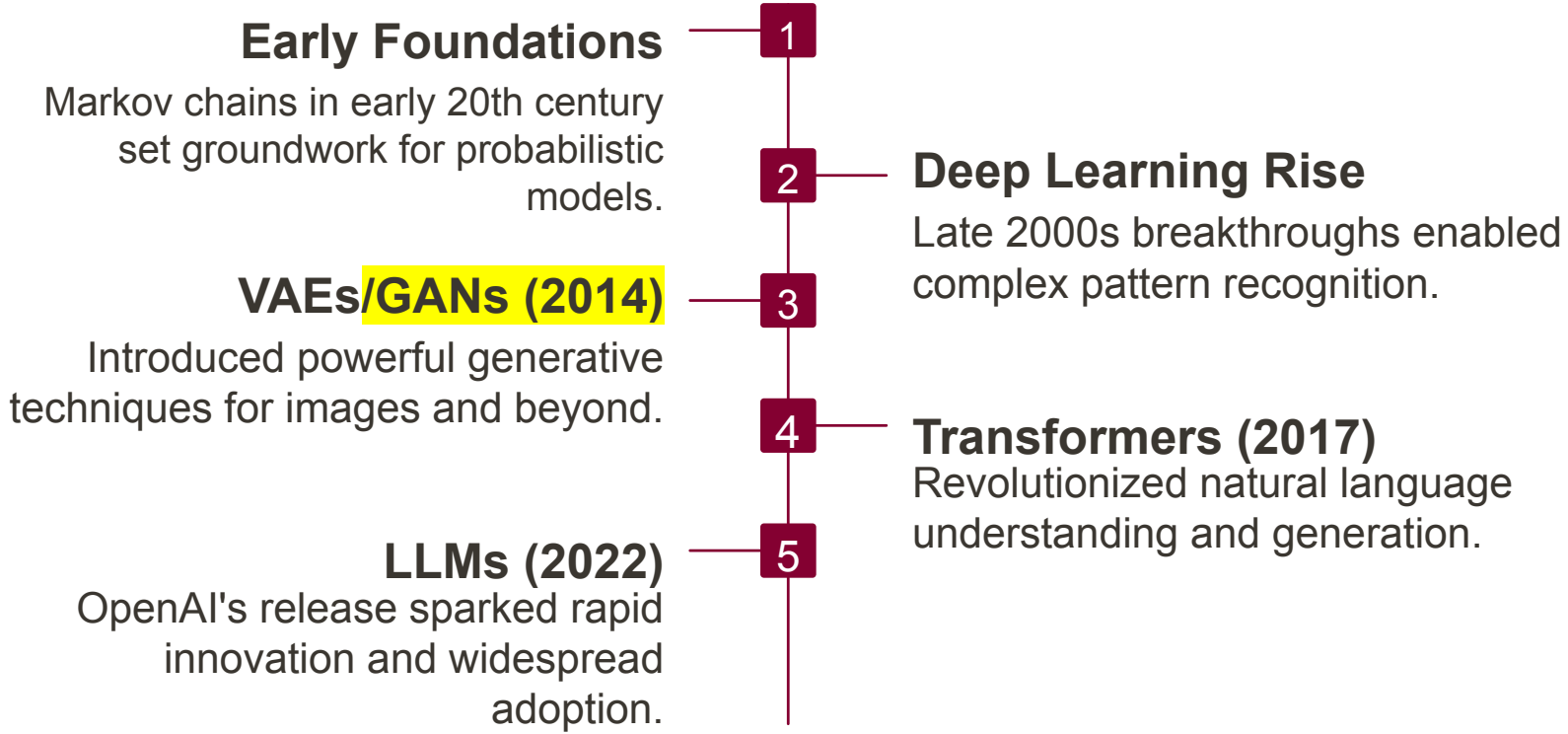
- Generative Adversarial Networks (GANs): The generator learns to produce samples that fool the discriminator, without modeling a probability density.
- Neural Style Transfer Models: Generate realistic outputs (e.g., styled images) without defining an explicit distribution.
- Energy-Based Models (some forms): Learn a scoring function instead of explicit likelihood.
- Diffusion Models (Denoising Diffusion Probabilistic Models): They iteratively denoise random noise into data samples; while probabilistic in construction, they are often considered implicit since they don’t always provide a tractable explicit likelihood.

# Implicit vs Explicit

In summary:

- If the model defines and optimizes a probability distribution (e.g., p(x)), it's explicit.
- If it just learns to generate realistic samples without modeling p(x), it's implicit.

# Evolution of Generative AI Models



# Generative Adversarial Networks (GANs)

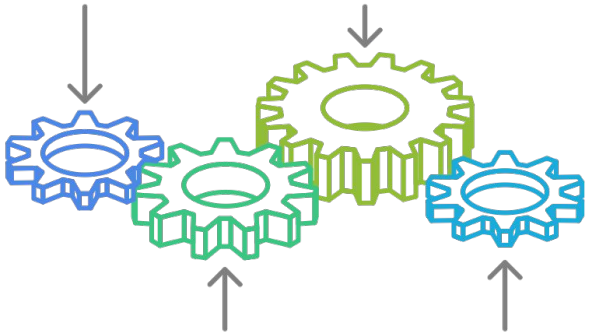
# What are GANs?

**Introduction of GANs**

Ian Goodfellow and colleagues introduce GANs in 2014.

**Adversarial Setup**

GANs use a competitive setup for learning.



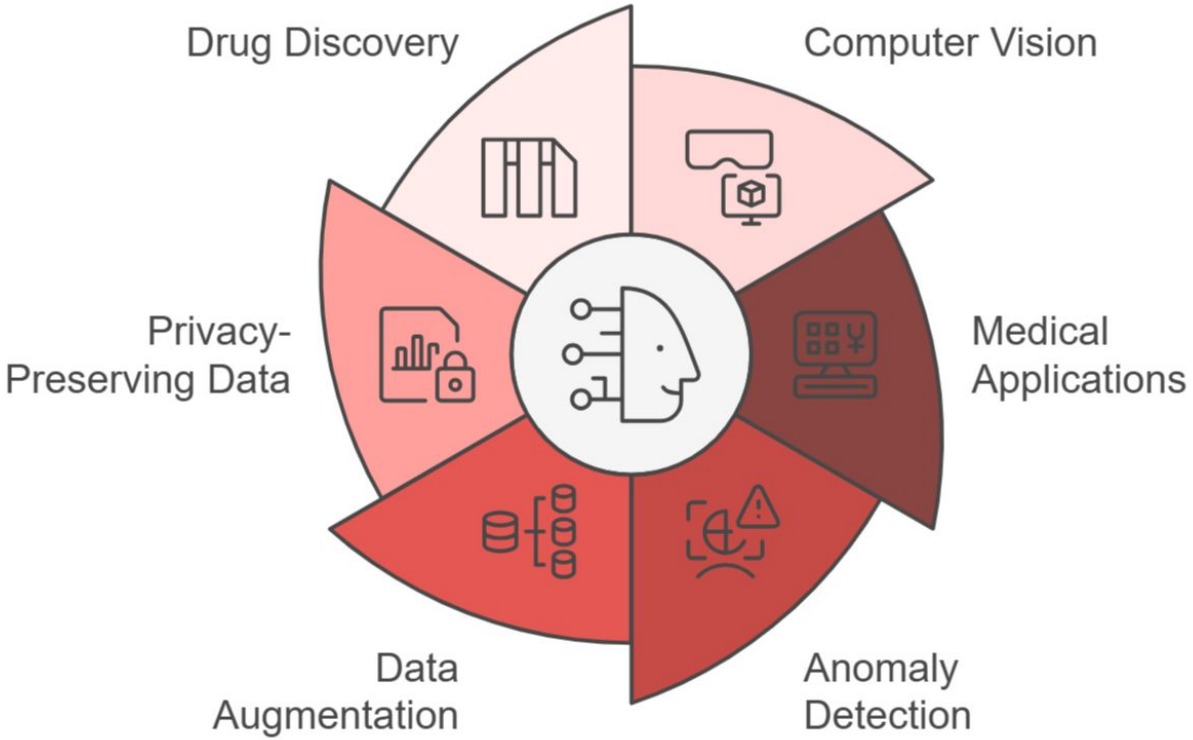
**Paradigm Shift**

GANs represent a new approach to generative modeling.

**Realistic Data Generation**

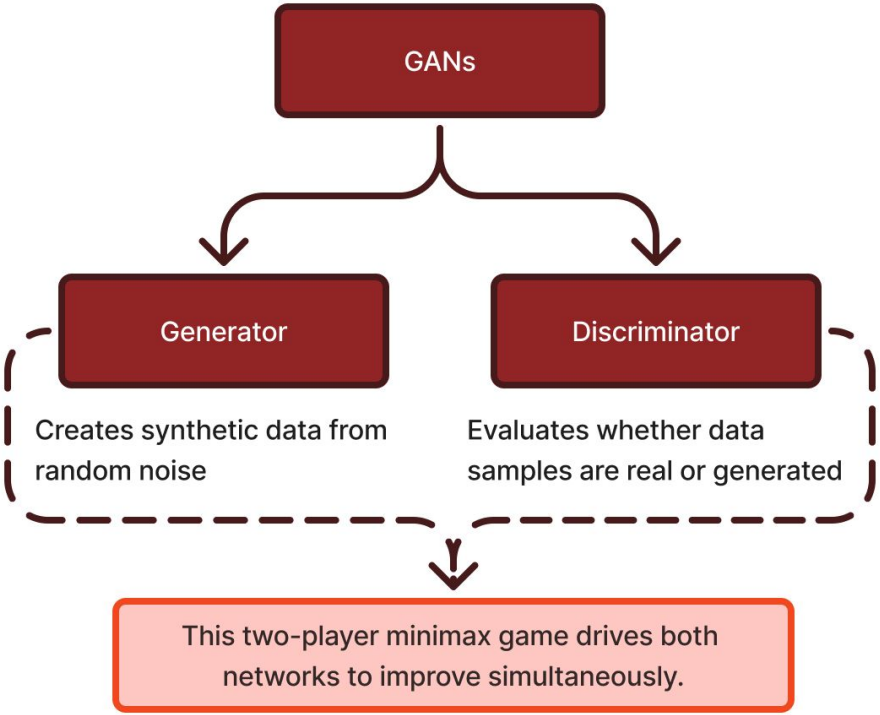
GANs produce synthetic data that is nearly indistinguishable from real data.

# Applications of GANs



# How do GANs work?

- Generative adversarial networks are implicit **likelihood** models that **generate data samples** from the **statistical distribution** of the data.
- The GAN architecture consists of two primary components that work in opposition: the **Generator** and the **Discriminator**



# GAN’s Generator

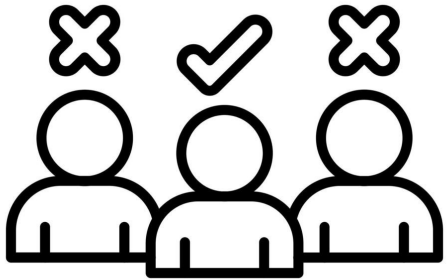
Transforms random noise vectors (z) into outputs that **mimic the training distribution**. Its objective is to produce samples that the Discriminator cannot distinguish from real data.

- Typically implemented using **deconvolutional** layers
- Transforms **latent space vectors** into **structured data**
- Trained to maximize the Discriminator error rate

# GAN's Discriminator

Evaluates both **real and generated** samples, outputting a **probability** that the input belongs to the real dataset. It functions as a **binary classifier** trained to correctly identify data sources.

- Often uses **convolutional architecture** for images
- Acts as an **adaptive loss** function for the Generator
- Must balance between **overconfidence and underconfidence**

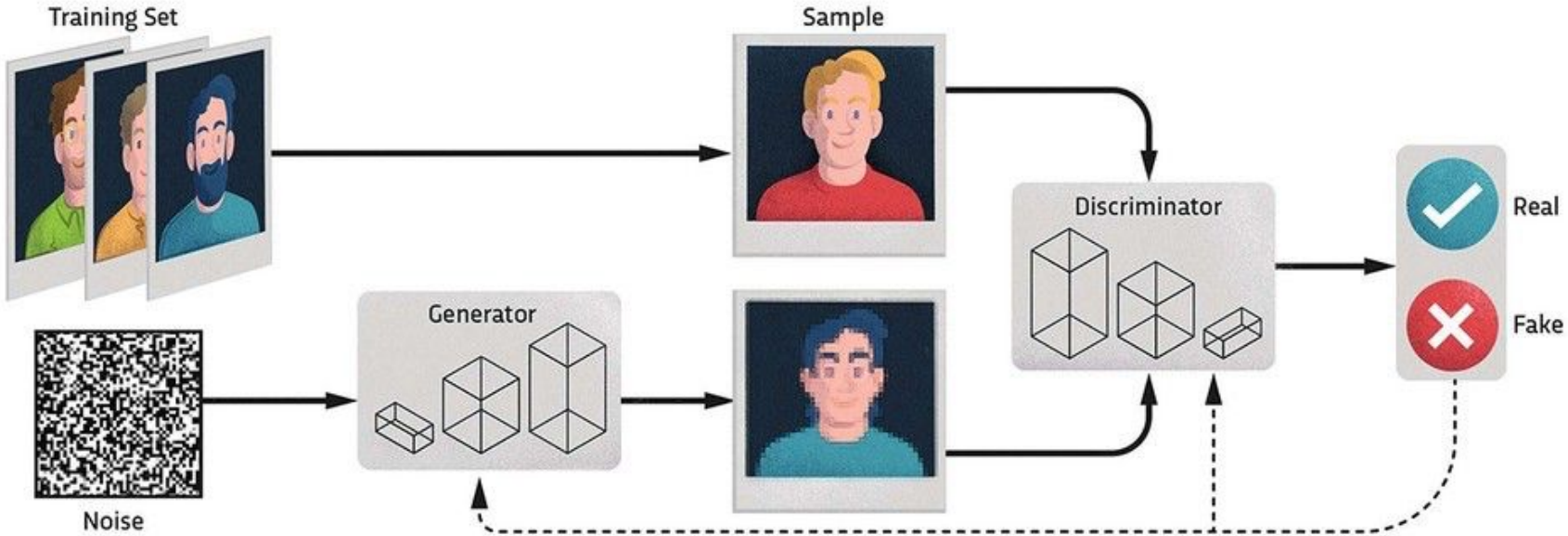


# Adversarial Relationship

This adversarial relationship drives both networks toward improvement until reaching an **equilibrium** where generated samples are **indistinguishable** from real data.

- **Zero-sum game** where one network's gain is the other's loss
- **Competition** leads to continuous improvement
- **Equilibrium** represents optimal generative capability

# GAN Pipeline



# GAN Training Process

## Initialization

Both networks are initialized with random weights. The Generator creates obviously fake samples, and the Discriminator easily identifies them.

## Discriminator Training

The Discriminator is trained on a batch containing both real examples and Generator-created samples, learning to distinguish between them.

## Generator Training

With the Discriminator weights frozen, the Generator is trained to produce samples that maximize the Discriminator error rate.

## Alternating Optimization

Steps 2 and 3 are repeated, with each network improving to counter the other's advancements until equilibrium is reached.

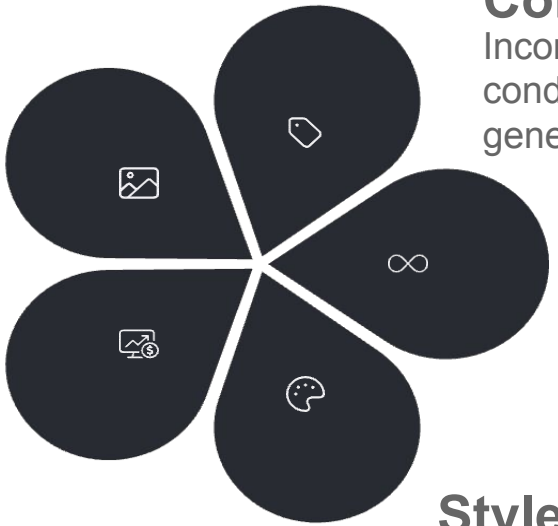
# Types of GANs

## DCGAN

Deep Convolutional GAN uses convolutional and batch normalization layers for stable image generation.

## CycleGAN

Performs unpaired image-to-image translation using cycle consistency loss.



## Conditional GAN

Incorporates class labels or other conditional information to control the generation process.

## Wasserstein GAN

Uses Wasserstein distance to improve training stability and prevent mode collapse.

## StyleGAN

Enables fine-grained control over generated features and disentangled representation.

# Limitations of GANs

## Training Instability

GANs often suffer from oscillating loss values, sensitivity to hyperparameters, and difficulty reaching convergence. Advanced regularization techniques and careful initialization are required for stable training.

## Evaluation Difficulty

Quantifying GAN performance remains challenging, with metrics like Inception Score and FID providing incomplete measures. Human evaluation is still necessary for many applications.

## Mode Collapse

Generators frequently produce limited varieties of samples, ignoring entire modes of the target distribution. This results in reduced diversity and is particularly challenging in complex, multimodal distributions.

## Architectural Sensitivity

GAN performance depends heavily on specific architectural choices and hyperparameter settings, making reproducibility difficult and requiring extensive experimentation.

# Hands on: Build and train a simple GAN using Tensorflow in ‘Train a GAN.ipynb’

# Variational AutoEncoders (VAEs)

# Evolution of Generative AI Models

AMERICAN UNIVERSITY OF BEIRUT

1

## Early Foundations

Markov chains in early 20th century set groundwork for probabilistic models.

2

## Deep Learning Rise

Late 2000s breakthroughs enabled complex pattern recognition.

3

## VAEs/GANs (2014)

Introduced powerful generative techniques for images and beyond.

4

## Transformers (2017)

Revolutionized natural language understanding and generation.

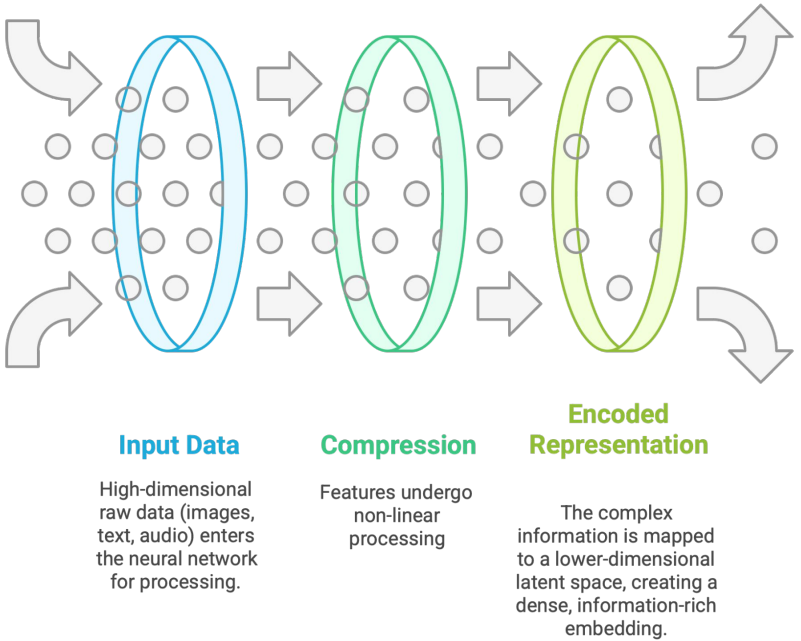
5

## LLMs (2022)

OpenAI's release sparked rapid innovation and widespread adoption.

# Encoders

- Before we dive into Variational AutoEncoders, let's first understand what is an **encoder**.
- An encoder is a neural network (or part of a network) whose sole purpose is to **map high-dimensional** input data ***x*** into a typically **lower-dimensional** latent representation ***z***



# Encoders: When/Why Use a Standalone Encoder?

1. **Dimensionality Reduction**
   To compress features (e.g., reduce 784-dimensional MNIST images to a 32-dimensional code) before feeding into another module (e.g., classifier).
2. **Feature Extraction**
   To extract a fixed-size embedding (e.g., in Transformer-based encoders for NLP, or CNN encoders in vision tasks).
3. **Pretrained Representations**
   You might use a pre trained encoder (e.g., ResNet, BERT) to obtain embeddings for downstream tasks.

# AutoEncoders

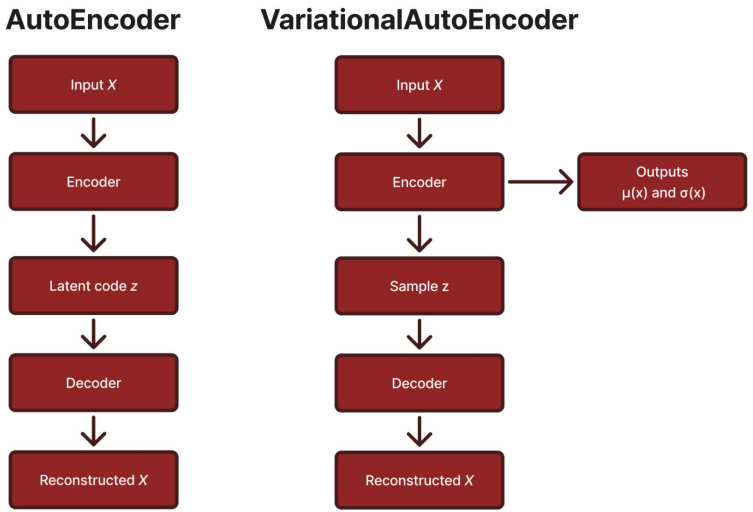
- An autoencoder consists of two parts: an **Encoder** (Maps input x to latent code z) and a **Decoder** (Attempts to reconstruct x from z).
- Using the **decoder**, the autoencoder attempts to **reconstruct the input** to create a **compact representation** without losing any information.

# AutoEncoders: When do we use them?

1. **Dimensionality Reduction / Data Compression**
   The bottleneck forces the network to learn a compact representation
2. **Denoising Autoencoders (DAE)**
   Encoder–decoder architecture trained to reconstruct “clean” data from corrupted input
3. **Feature Learning**
   Use the latent code z as “learned features” for classification, clustering, etc
4. **Data Visualization**
   Projecting high-dimensional data into a 2D or 3D latent space (when the bottleneck has that size)

# Variational AutoEncoders

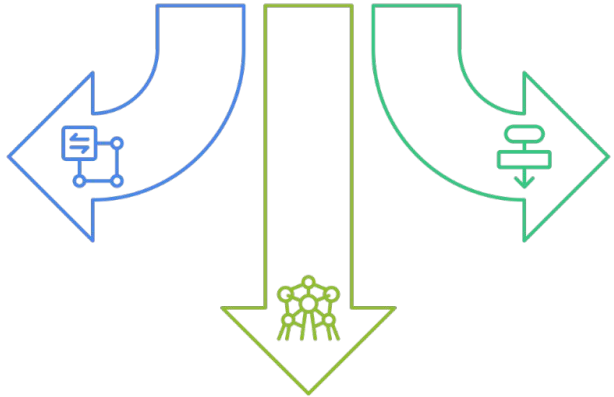
- Unlike classical autoencoders that simply compress and reconstruct data, VAEs learn a **probabilistic mapping** from inputs to a continuous latent space
- VAEs encode inputs as **probability distributions**, typically multivariate Gaussians.
- This probabilistic approach enables VAEs to generate **diverse, novel outputs** that maintain the statistical properties of the training data



# Variational Autoencoders: When do we use them?

1. **Generative Modeling**
   Since we know the prior distribution p(z)=(usually standard normal), we can sample z, feed it into the decoder, and generate new data.
2. **Latent-Space Interpolation**
   The latent space is smooth and continuous, so moving between two points in it gives smooth transitions between their reconstructions.
3. **Semi-Supervised Learning**
   The encoder gives a compact representation z that can be used for other tasks, especially when combined with some labels.
4. **Anomaly Detection**
   If an input x is far from the normal latent space, it's likely an outlier or anomaly.

# Encoders to AutoEncoders: Recap



**Encoder**

Maps input to latent code but does not reconstruct input.

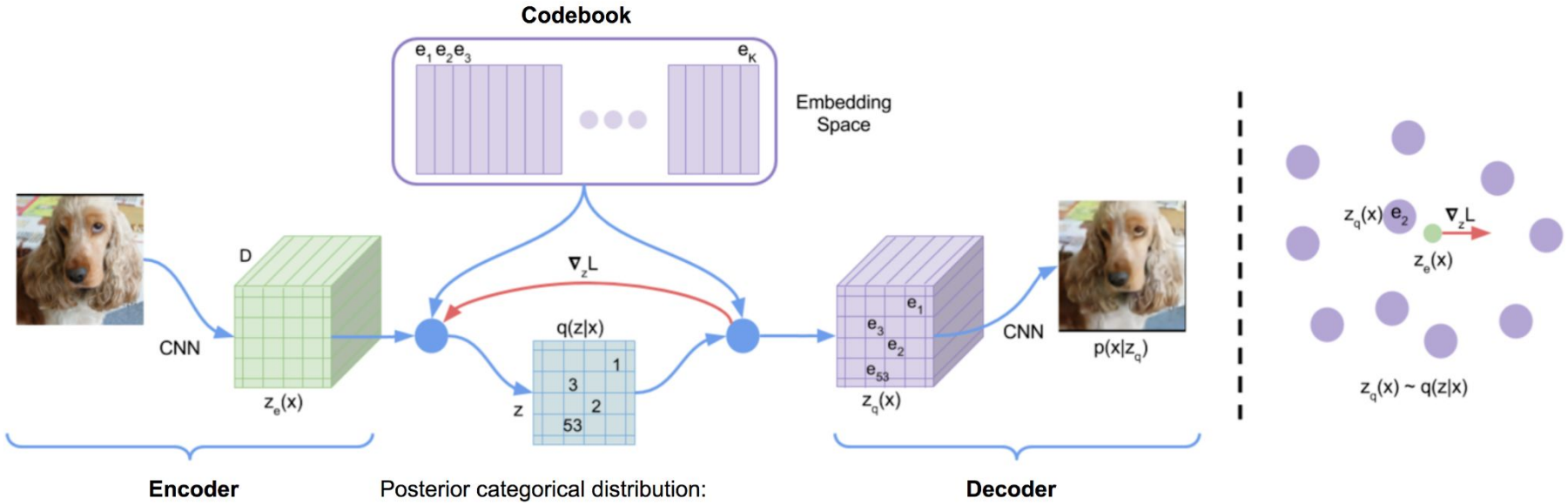
**Autoencoder**

Reconstructs input with a deterministic latent code.

**VAE**

Enables true generative modeling with a probabilistic latent and KL-divergence.

# VAE Pipeline




$$q(\mathbf{z} = \mathbf{e}_k|\mathbf{x}) = \begin{cases} 1 & \text{if } k = \arg\min_i \|\mathbf{z}_e(\mathbf{x}) - \mathbf{e}_i\|_2 \\ 0 & \text{otherwise.} \end{cases}$$

# Hands on: Build and train a simple VAE using Tensorflow in ‘Train a VAE.ipynb’

# Evolution of Generative AI Models

1

## Early Foundations

Markov chains in early 20th century set groundwork for probabilistic models.

2

## Deep Learning Rise

Late 2000s breakthroughs enabled complex pattern recognition.

3

## VAEs/GANs (2014)

Introduced powerful generative techniques for images and beyond.

4

## Transformers (2017)

Revolutionized natural language understanding and generation.

5

## LLMs (2022)

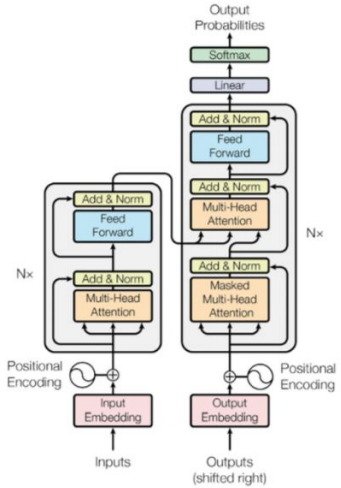
OpenAI's release sparked rapid innovation and widespread adoption.

# The Famous Transformer Architecture

- In 2017, researchers at google released the famous **Transformer** architecture in the paper ‘Attention is all you need’
- Although the Transformer architecture was not originally designed for generative tasks, it became the breakthrough that enabled generative models to achieve their current level of power

**Transformer**

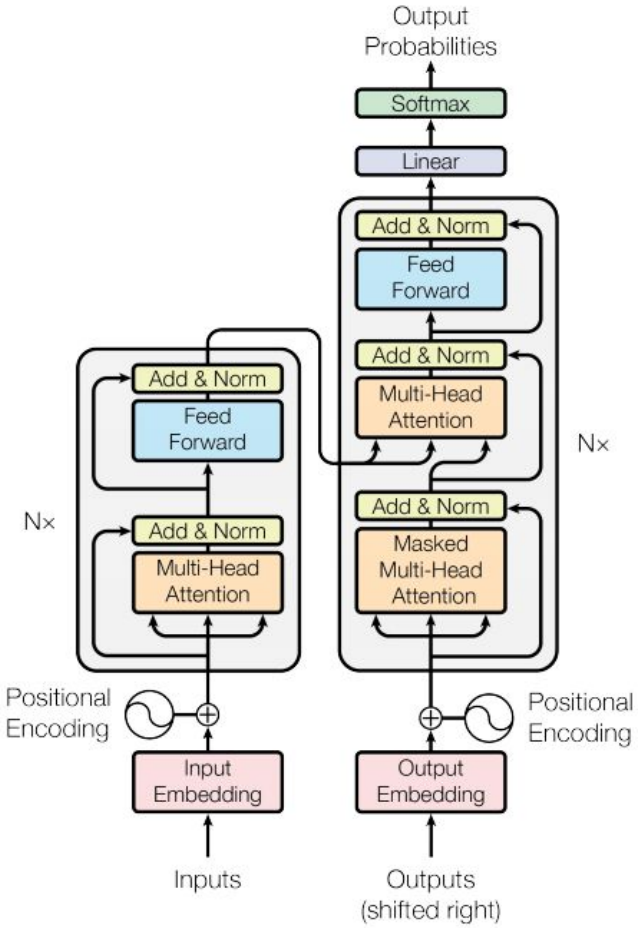
**Attention Is All You Need**



# What is a Transformer?

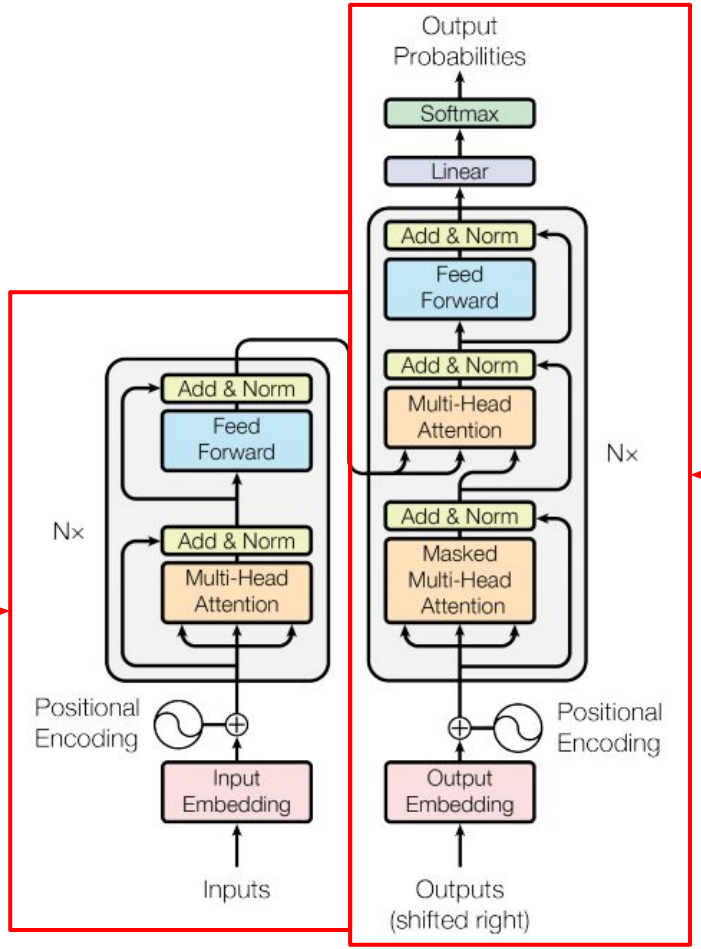
This is the transformer:

Let's dissect it



# What is a Transformer?

This is called an encoder block




This is called a decoder block

# Encoders and Decoders: Not a new concept

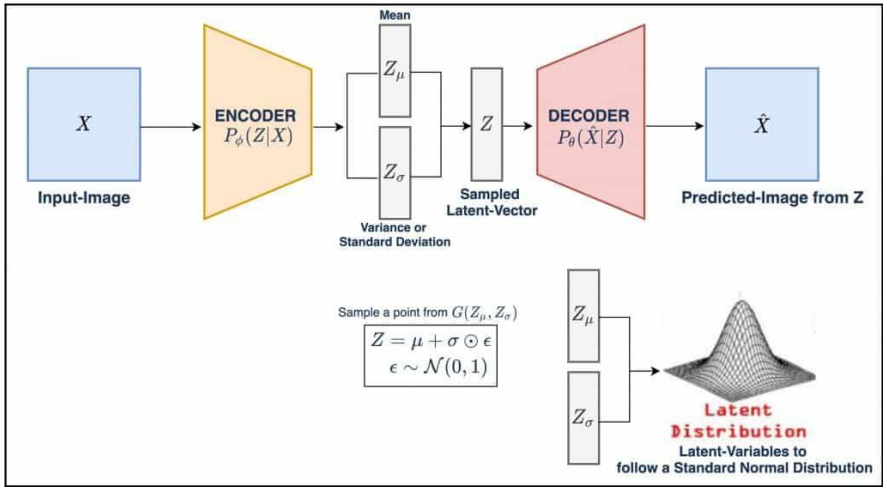
In Variational Autoencoders, we saw the concept of **encoders and decoders** before:

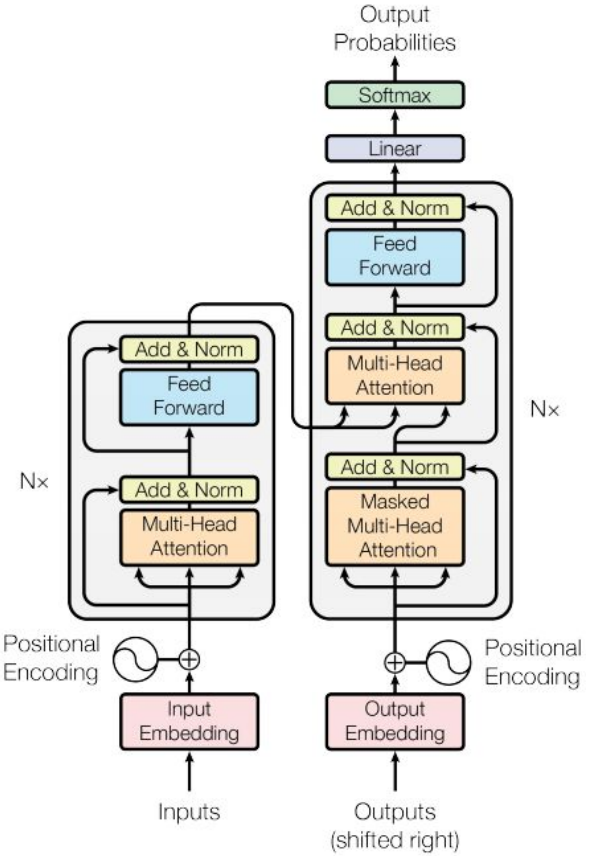
1. The **encoder** outputted a condensed representation of the input that retained all the important information.
2. The **decoder** generated a new version the input from the encoder's representation.

Are the encoder and decoder in VAEs the same in Transformers? Yes… and **no**.

# Encoders and Decoders in Transformers

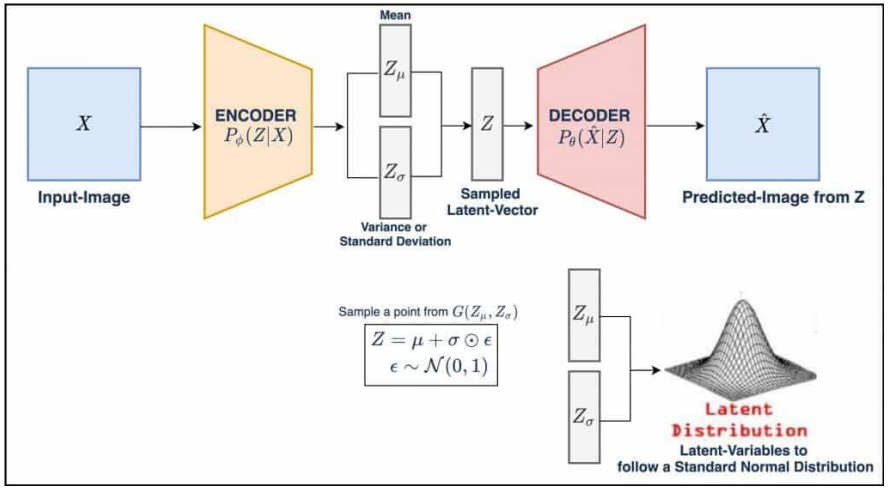
While the encoder and decoder in **VAEs** and **Transformers** have the same functionality, they work **completely differently** under the hood

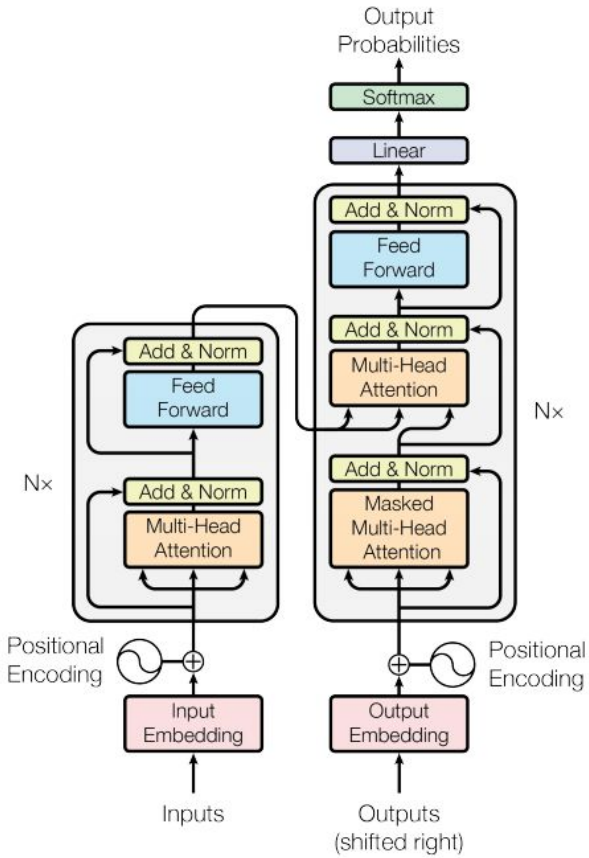


# Encoders and Decoders in Transformers

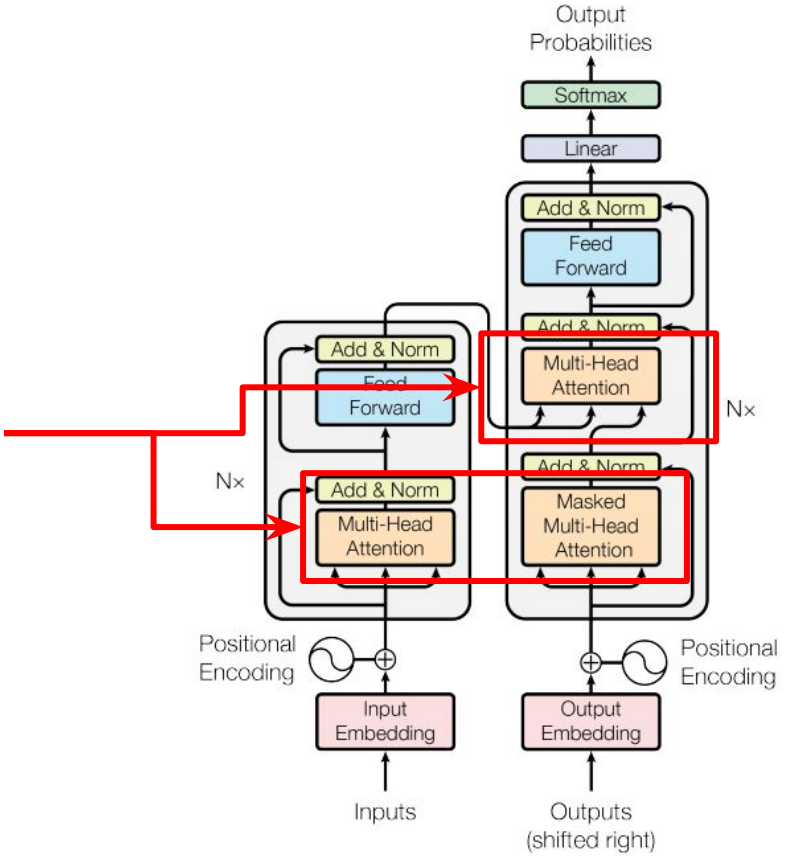
**Variational Autoencoders (VAEs)** typically use simpler neural network architectures, such as fully-connected (dense) or convolutional neural networks (CNNs), for their encoders and decoders



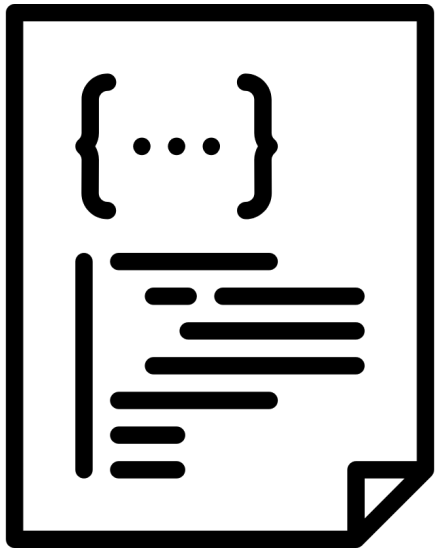
# Encoders and Decoders in Transformers

Instead of feed forward neural networks and CNNs, the transformer uses **attention-based** encoders and decoders



# Encoders and Decoders in Transformers

- Why was a completely new architecture of encoders and decoders proposed anyway? What was wrong with NN + CNNs?
- While the VAE architecture preformed impressively well on image data, it always underperformed when dealing with **text or sequence data.**



# The need for Transformers

The transformer architecture, or the attention mechanism in specific, came as a solution to many problems:

1. **RNNs and LSTMs** always had a problem (the problem being vanishing/exploding gradients) with processing **long sequences.**
2. The authors of the AIAYN paper noticed that the solution to unlocking robust text generation is not trying to fit all the context in the memory (hidden state).

The authors approached sequence understanding in AI by emulating **how humans comprehend text.**

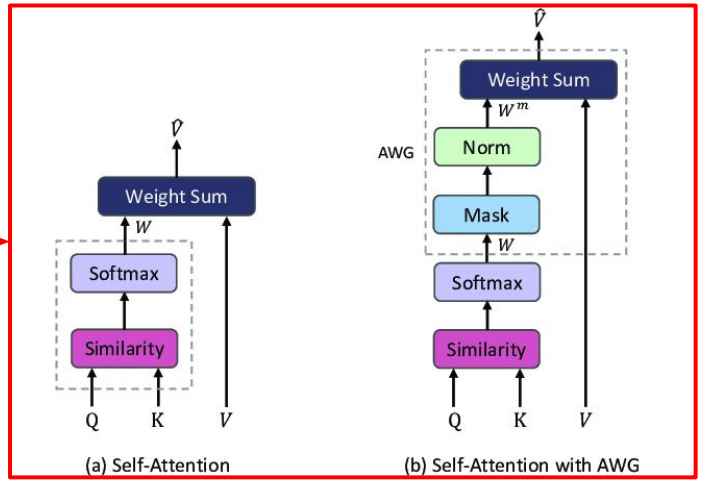
# How do humans process text?

- If you were asked to summarize a book you've read, how would you complete this task?
- What you WOULDN'T do is recall every single word in the book before summarizing it.
- What you WOULD do is recall **only** the most important events and characters to piece together a shorter version of the story.
- In other words, your brain would **pay attention** to the most important information instead of every single detail. This is the high level concept of the attention mechanism.

# The Attention Mechanism

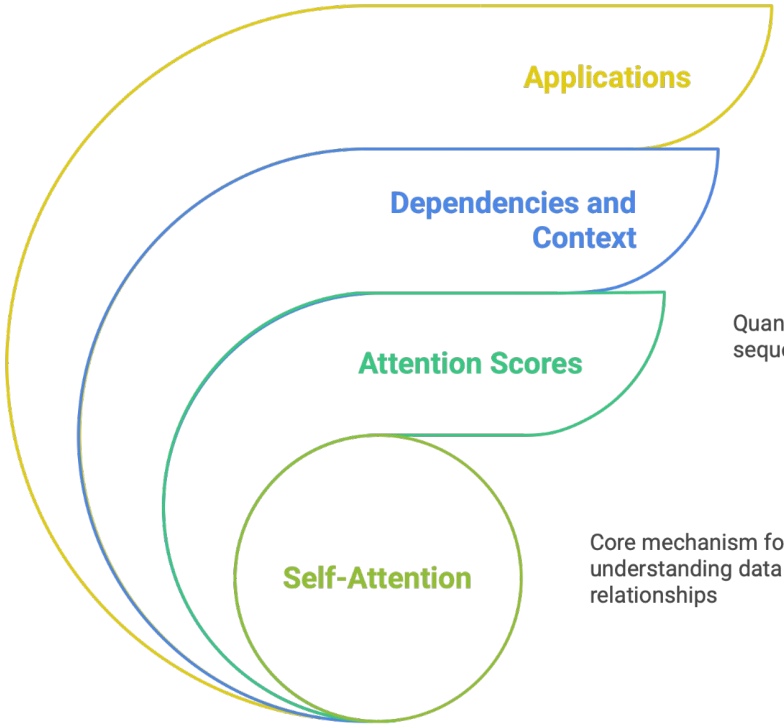
- Earlier, we claimed that the transformer architecture is what kickstarted the focus on generative AI.
- Well, that's partially true. Technically, it is the **attention mechanism** that allowed deep learning algorithms to reach the power they have today.

Attention

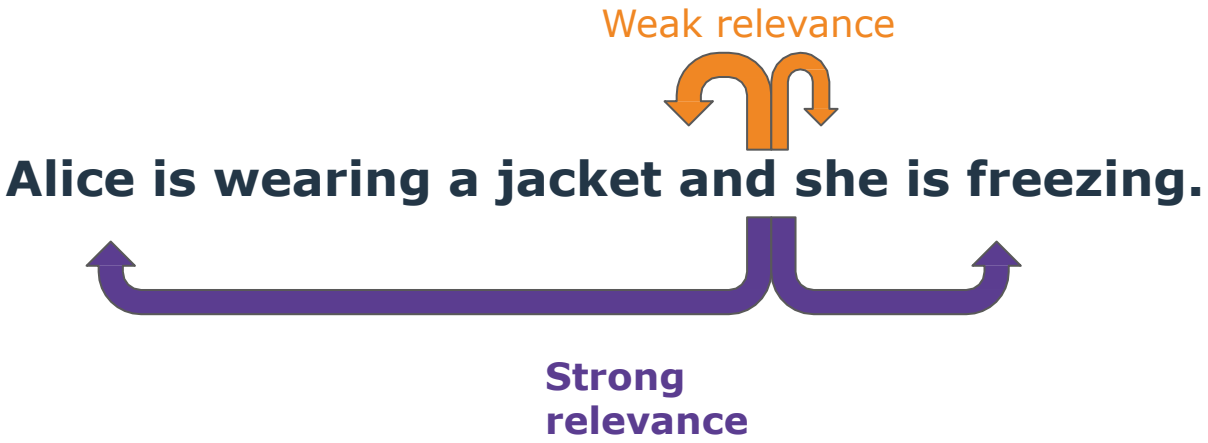


# The Attention Mechanism

- In the transformer architecture, we saw that the attention block was called **‘Masked MultiHead Attention’**.
- Before we dive into this variation, we will understand the the main building block: **self-attention.**



# How does Attention work?



# Step 1: Vectorization

**Alice is wearing a jacket and she is freezing.**

$t_0$ $t_1$ $t_2$ $t_3$ $t_4$ $t_5$ $t_6$ $t_7$ $t_8$

Vectorization refers to creating embeddings that encompass both the positional and semantic context of the words.

$$t_0, t_1, \ldots, t_n \longrightarrow V_0, V_1, \ldots, V_n$$

# Step 2: Creating Weights

To introduce context to our vectors $V$, We select **one token** at a time. Let's say we want to apply attention for $V_{0,}$ we first get the **dot product** of the vector $V_0$ with each of the vectors including itself.

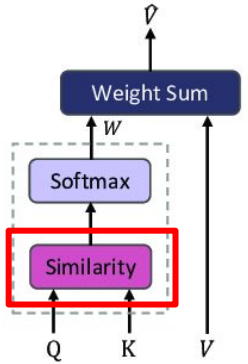
$$V_0, V_1, V_2, V_3, \ldots, V_n$$

$$. \quad . \quad . \quad . \quad .$$

$$V_0, V_0, V_0, V_0, \ldots, V_0$$

$$=$$

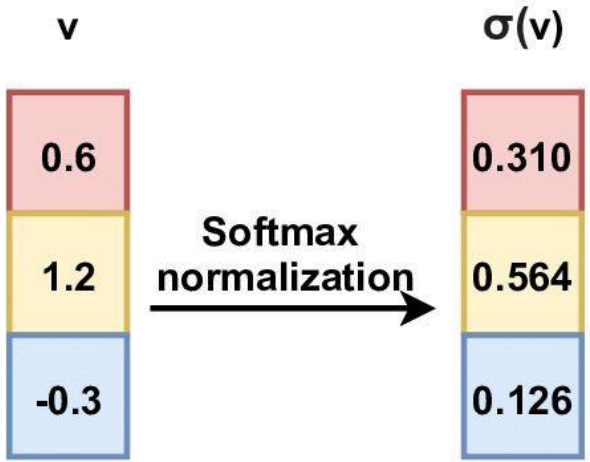
$$W_{00}, W_{01}, W_{02}, W_{03}, \ldots, W_{0n}$$

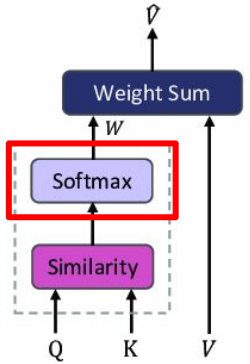


# Step 3: Normalization of Weights

Now we have the **attention weights** : $W_{00}, W_{01}, W_{02}, W_{03}, \ldots, W_{0n}$

We need to **normalize** them to control the main differences in their values by using the **softmax** function, which normalizes the values between 0 and 1



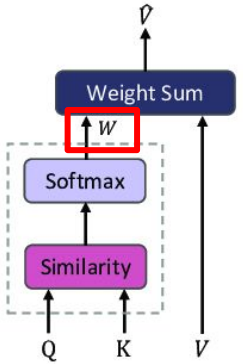
# Step 4: Contextualize the Initial Embeddings

We now **multiply** the initial embedding vectors by the similarity scores (weights) after normalization

$$V_0, V_1, V_2, V_3, \ldots, V_9$$

$$x \quad x \quad x \quad x \quad x$$

$$W_{00}, W_{01}, W_{02}, W_{03}, \ldots, W_{09}$$

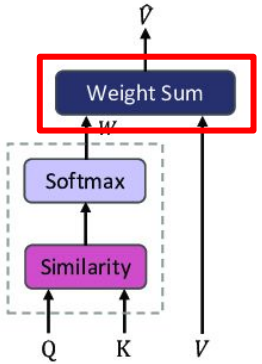


# Step 5: The Final Embedding Y

- After obtaining all the weighted vectors, we **sum up** the weighted vectors to obtain a final vector Y:

$$V_0 W_{00} + V_1 W_{01} + V_2 W_{02} + V_3 W_{03} + \ldots + V_n W_{0n} = Y_0$$

- $Y_0$ replaces the vector $V_0$ as the **contextualized embedding**



# Self-Attention

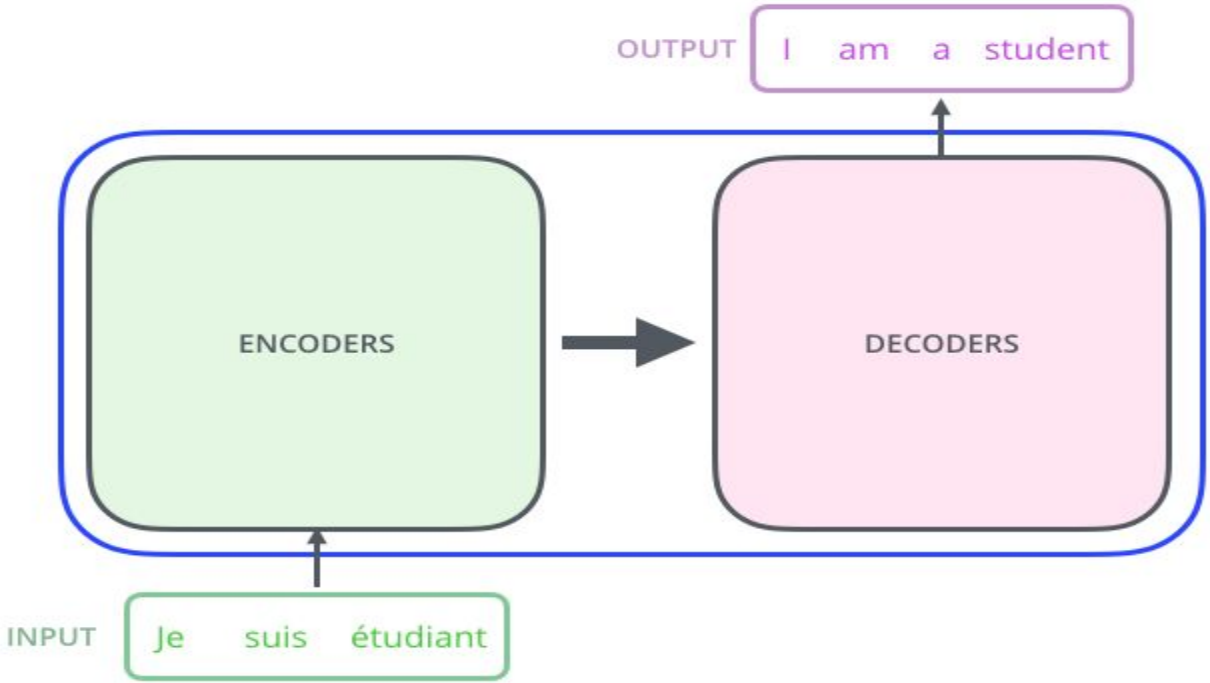
- The process that we saw earlier describes the **basic steps of self-attention**.
- However, there is **one component** missing in the calculation...
- If the attention mechanism works only by multiplication and dot products, how is it being **trained**? What is learned during the **ingestion of the corpus**?

# Wait, let’s make sure we’re aligned first

# A High-Level Look

Let’s begin by looking at the model as a single black box. In a machine translation application, it would take a sentence in one language, and output its translation in another.

# A High-Level Look

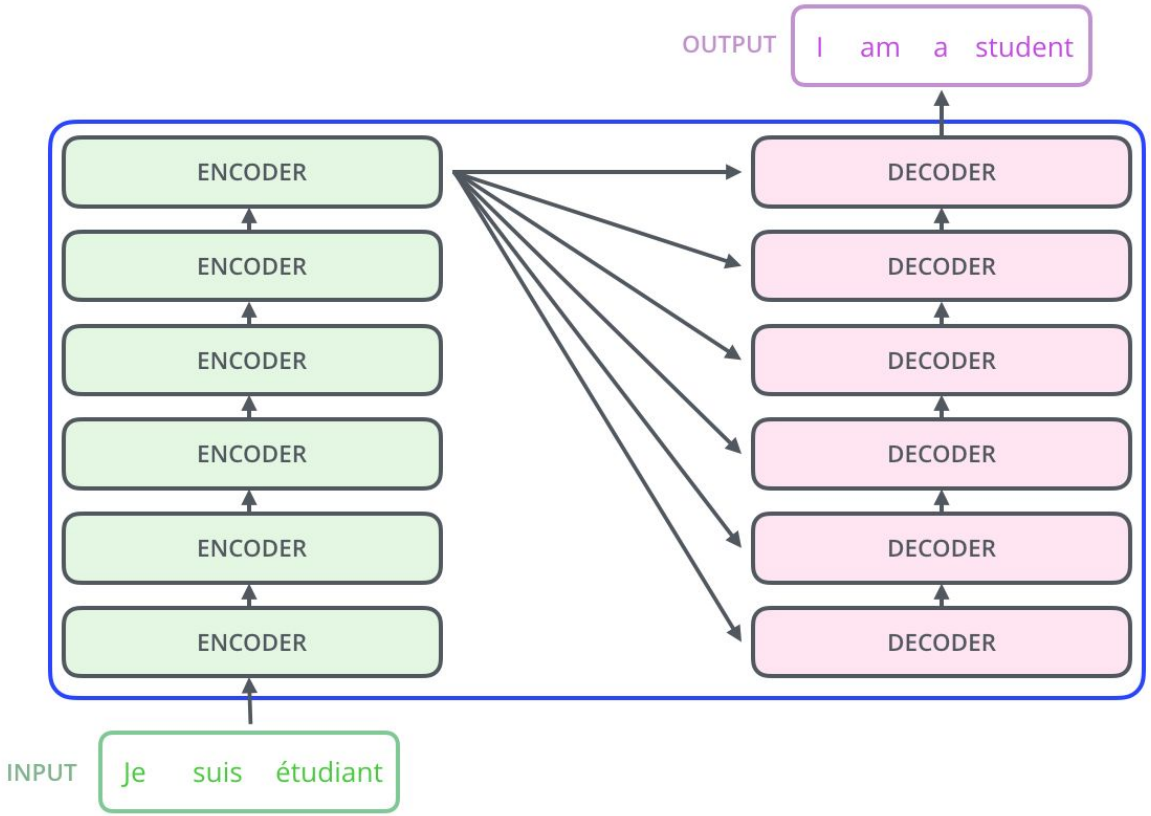


# A High-Level Look

The encoding component is a stack of encoders (the paper stacks six of them on top of each other – there's nothing magical about the number six, one can definitely experiment with other arrangements).

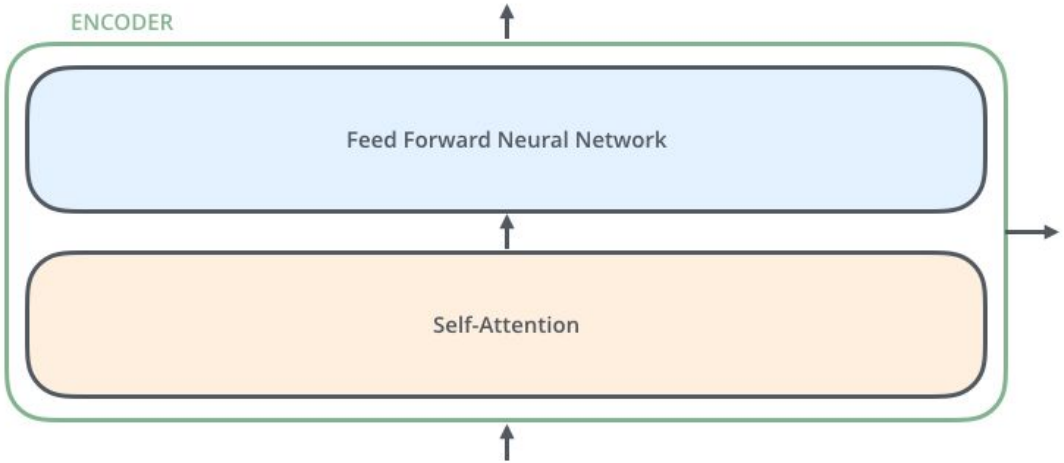
The decoding component is a stack of decoders of the same number.

# A High-Level Look



# A High-Level Look [Encoder]

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



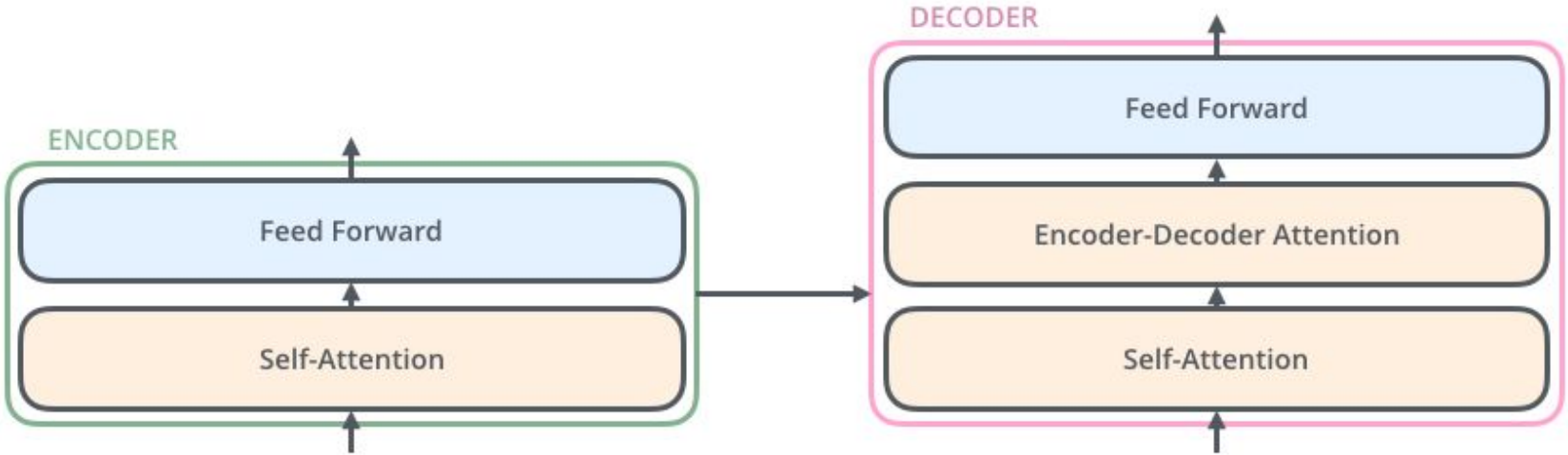
# A High-Level Look [Encoder]

The Encoder's inputs first flow through a self-attention layer – **a layer that helps the encoder look at other words in the input sentence as it encodes a specific word**. We'll look closer at self-attention later.

The outputs of the self-attention layer are fed to a feed-forward neural network. The exact same feed-forward network is independently applied to each position.

# A High-Level Look [Decoder]

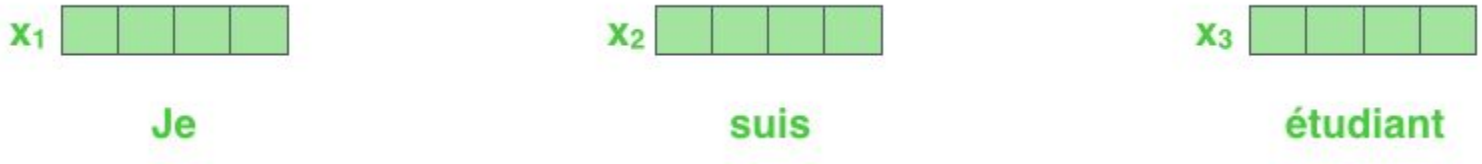
The decoder has both those layers, but between them is an attention layer that helps the decoder focus on relevant parts of the input sentence (similar what attention does in seq2seq models).



# Bringing The Tensors Into The Picture

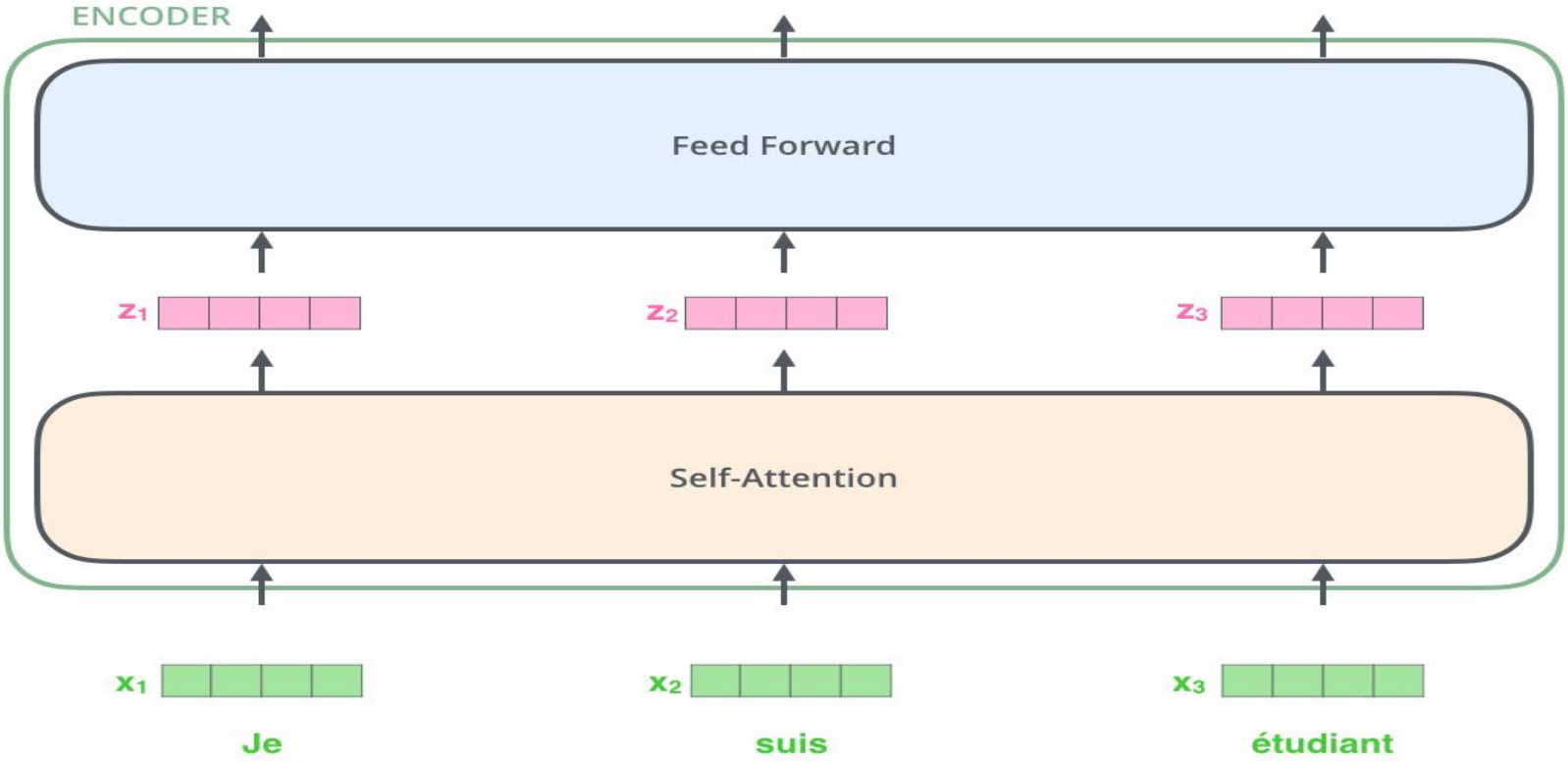
Now that we've seen the major components of the model, let's start to look at the various vectors/tensors and how they flow between these components to turn the input of a trained model into an output.

As is the case in NLP applications in general, we begin by turning each input word into a vector using an embedding algorithm.



# Bringing The Tensors Into The Picture

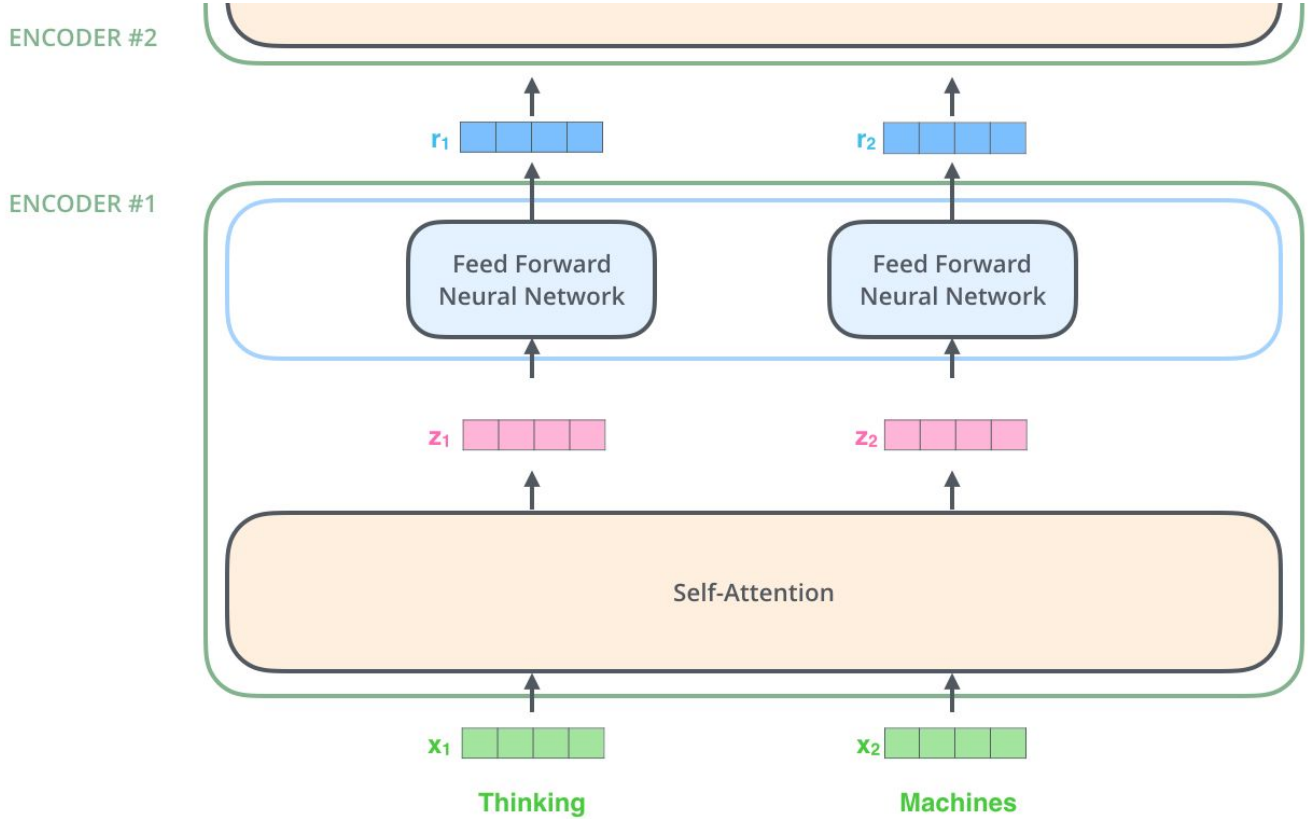
After embedding the words in our input sequence, each of them flows through each of the two layers of the encoder.


ENCODER
Feed Forward
z1
z2
z3
Self-Attention
x1
x2
x3
Je
suis
étudiant

# Now We’re Encoding!

As we’ve mentioned already, an encoder receives a list of vectors as input. It processes this list by passing these vectors into a ‘self-attention’ layer, then into a feed-forward neural network, then sends out the output upwards to the next encoder.

# Now We're Encoding!



# Self-Attention at a High Level

Say the following sentence is an input sentence we want to translate:

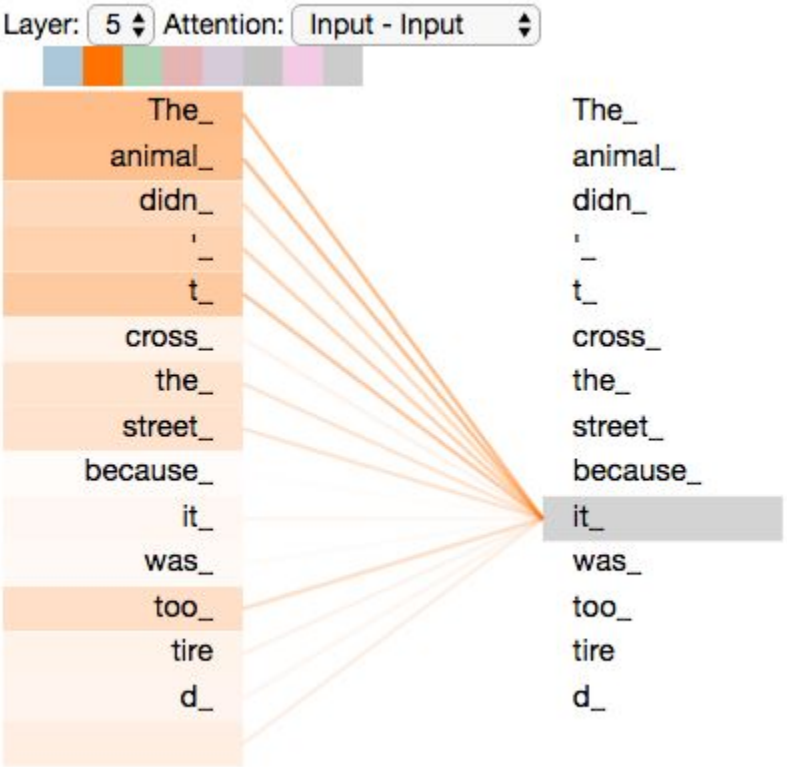
”The animal didn't cross the street because it was too tired”

What does “it” in this sentence refer to? Is it referring to the street or to the animal? It’s a simple question to a human, but not as simple to an algorithm.

When the model is processing the word “it”, self-attention allows it to associate “it” with “animal”.

As the model processes each word (each position in the input sequence), self attention allows it to look at other positions in the input sequence for clues that can help lead to a better encoding for this word.

# Self-Attention at a High Level



# Two Kinds of Weights - What is Learned vs. What is Computed?

**Learned Parameters:**

- WQ, WK, WV (projection matrices)
- Feedforward layers inside each encoder/decoder block
- Updated via gradient descent on the training loss

**Computed Each Forward Pass:**

- Attention scores = $\text{softmax}(QK^T/\sqrt{d_k})$
- These depend on the current input sequence and change dynamically

**Key Idea:** The matrices learn how to project embeddings so that attention scores capture meaningful relationships.

# How Training Shapes Attention - How Do Attention Weights Become Meaningful?

- Start: Random queries/keys → weak or noisy attention
- Training: Model predicts next word; loss compares with ground truth
- Backpropagation: Adjusts WQ,WK,WV so useful associations get higher scores

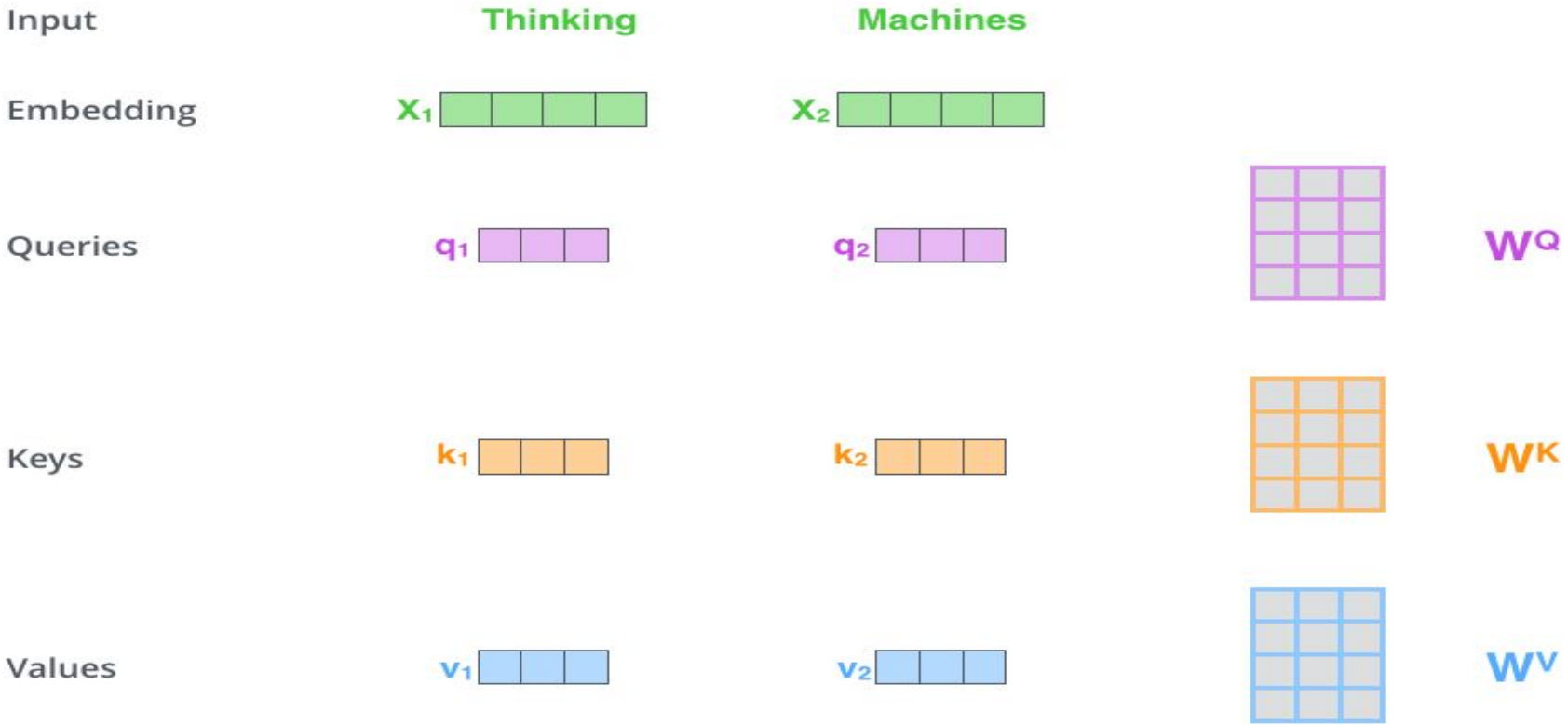
- Result: Attention aligns tokens that help prediction (e.g., “it” → “animal”)

# Self-Attention in Detail

Let's first look at how to calculate self-attention using vectors, then proceed to look at how it's actually implemented – using matrices.

The first step in calculating self-attention is to create **three vectors** from each of the encoder's input vectors (in this case, the embedding of each word). So for each word, we create a Query vector, a Key vector, and a Value vector. These vectors are created by multiplying the embedding by three matrices that we trained during the training process.

# Self-Attention in Detail



# What are the “query”, “key”, and “value” vectors?


AMERICAN UNIVERSITY OF BEIRUT

They’re abstractions that are useful for calculating and thinking about attention. Once you proceed with reading how attention is calculated below, you’ll know pretty much all you need to know about the role each of these vectors plays.

# Self-Attention Score

The second step in calculating self-attention is to calculate a score.

Say we're calculating the self-attention for the first word in this example, "Thinking". We need to score each word of the input sentence against this word.

The score determines how much focus to place on other parts of the input sentence as we encode a word at a certain position.

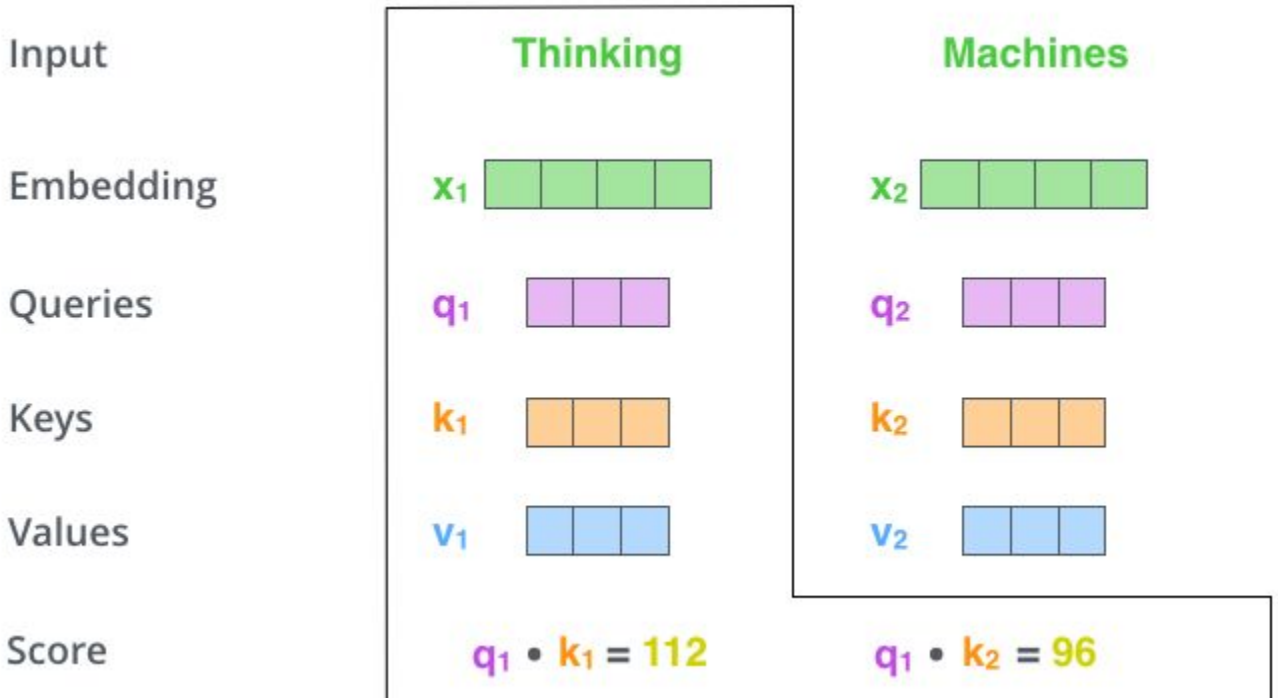
# Self-Attention Score

The score is calculated by taking the dot product of the query vector with the key vector of the respective word we're scoring.

So if we're processing the self-attention for the word in position #1, the first score would be the dot product of q1 and k1.

The second score would be the dot product of q1 and k2.

# Self-Attention Score



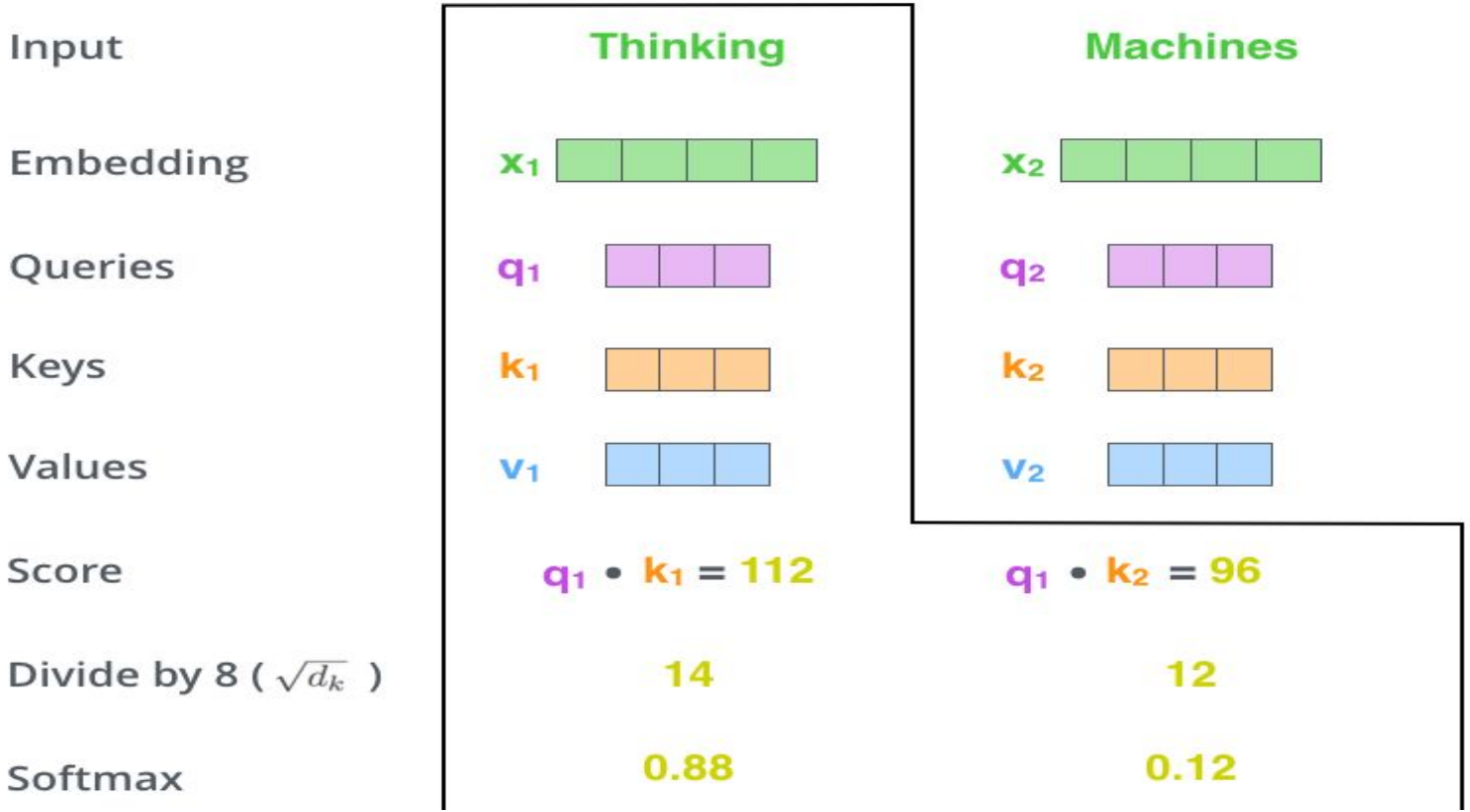
# Self-Attention Score

The third and fourth steps are to divide the scores by 8 (the square root of the dimension of the key vectors used in the paper – 64.

This leads to having more stable gradients.

There could be other possible values here, but this is the default, then pass the result through a softmax operation. Softmax normalizes the scores so they’re all positive and add up to 1.

# Self-Attention Score



# Softmax Score

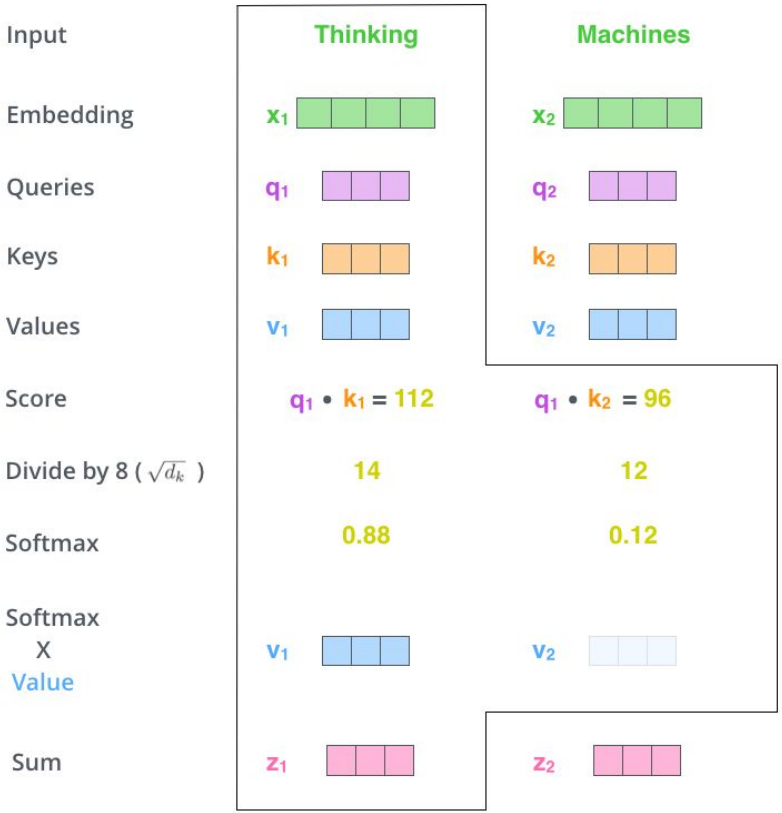
This softmax score determines how much each word will be expressed at this position. Clearly the word at this position will have the highest softmax score, but sometimes it’s useful to attend to another word that is relevant to the current word.

# Self-Attention Output

The fifth step is to multiply each value vector by the softmax score (in preparation to sum them up). The intuition here is to keep intact the values of the word(s) we want to focus on, and drown-out irrelevant words (by multiplying them by tiny numbers like 0.001, for example).

The sixth step is to sum up the weighted value vectors. This produces the output of the self-attention layer at this position (for the first word).

# Self-Attention Output



# Self-Attention Output

That concludes the self-attention calculation.

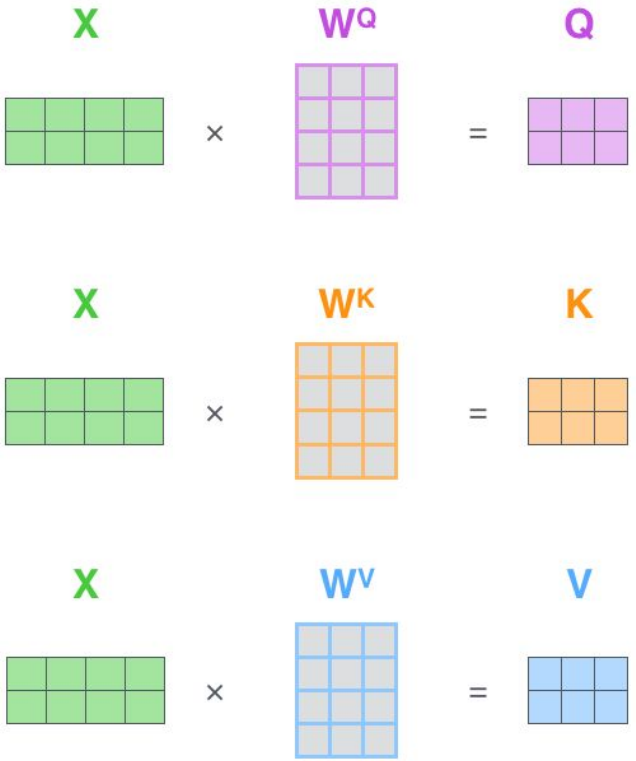
The resulting vector is one we can send along to the feed-forward neural network.

In the actual implementation, however, this calculation is done in matrix form for faster processing.

# Matrix Calculation of Self-Attention

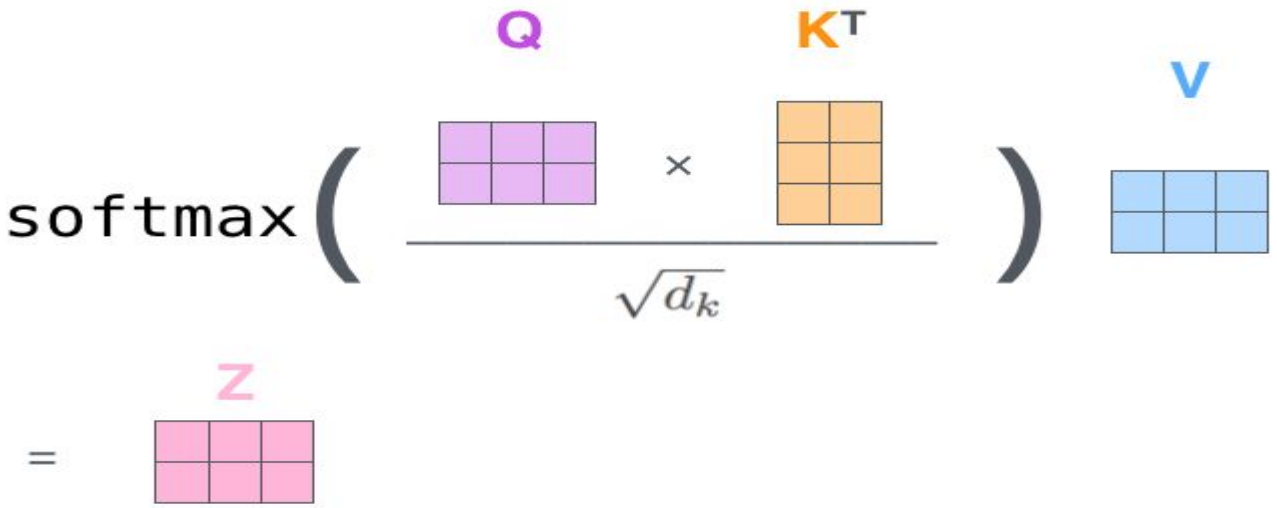
The first step is to calculate the Query, Key, and Value matrices.

We do that by packing our embeddings into a matrix X, and multiplying it by the weight matrices we've trained (WQ, WK, WV).

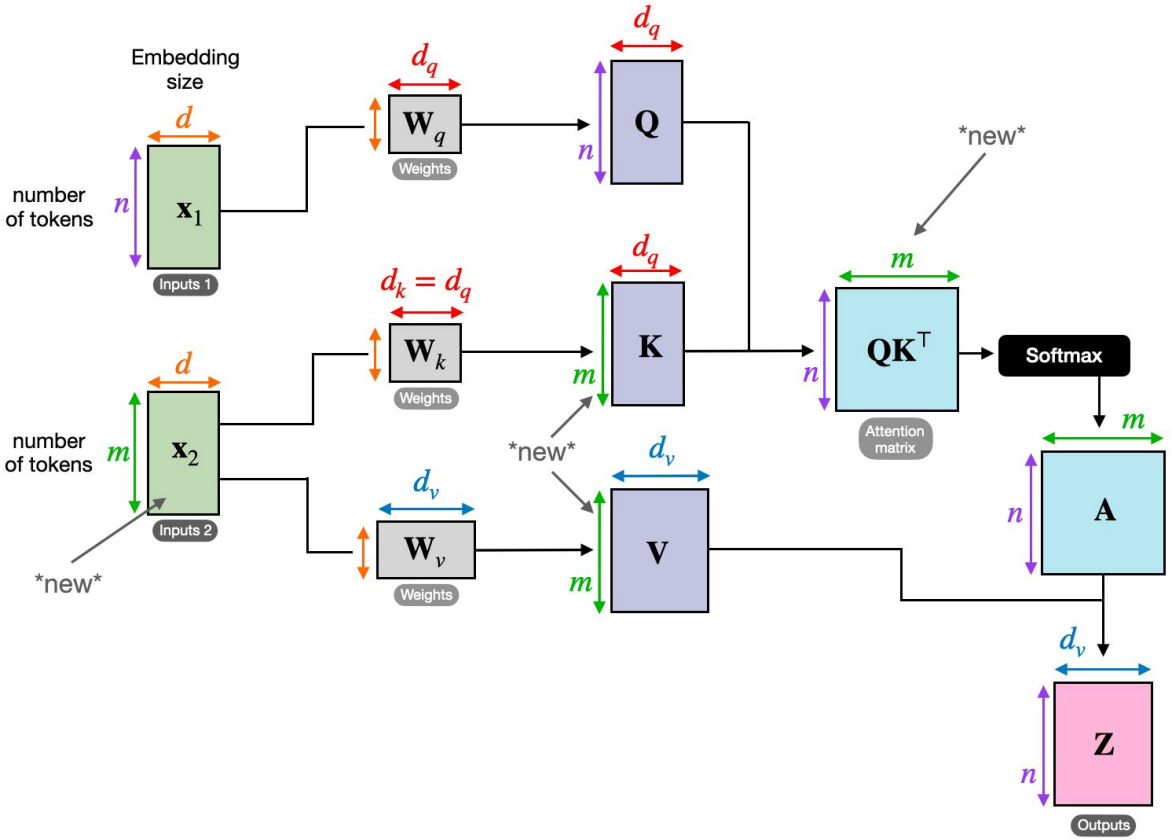


# Matrix Calculation of Self-Attention

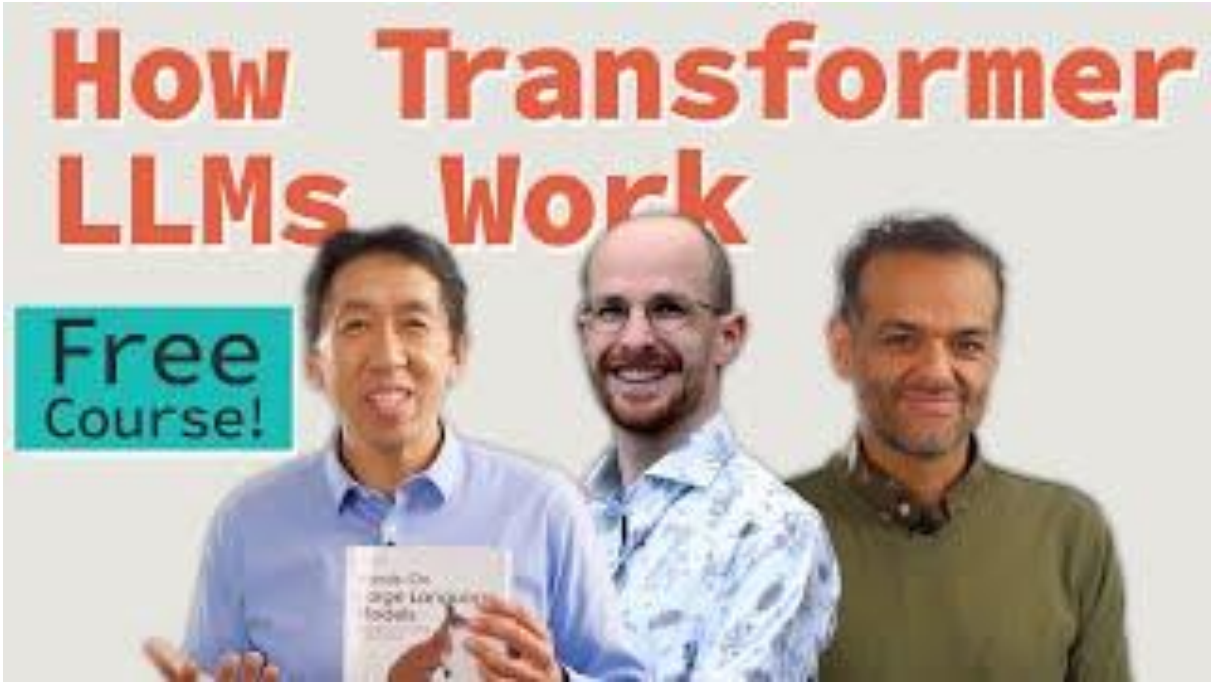
Finally, since we're dealing with matrices, we can condense steps two through six in one formula to calculate the outputs of the self-attention layer.



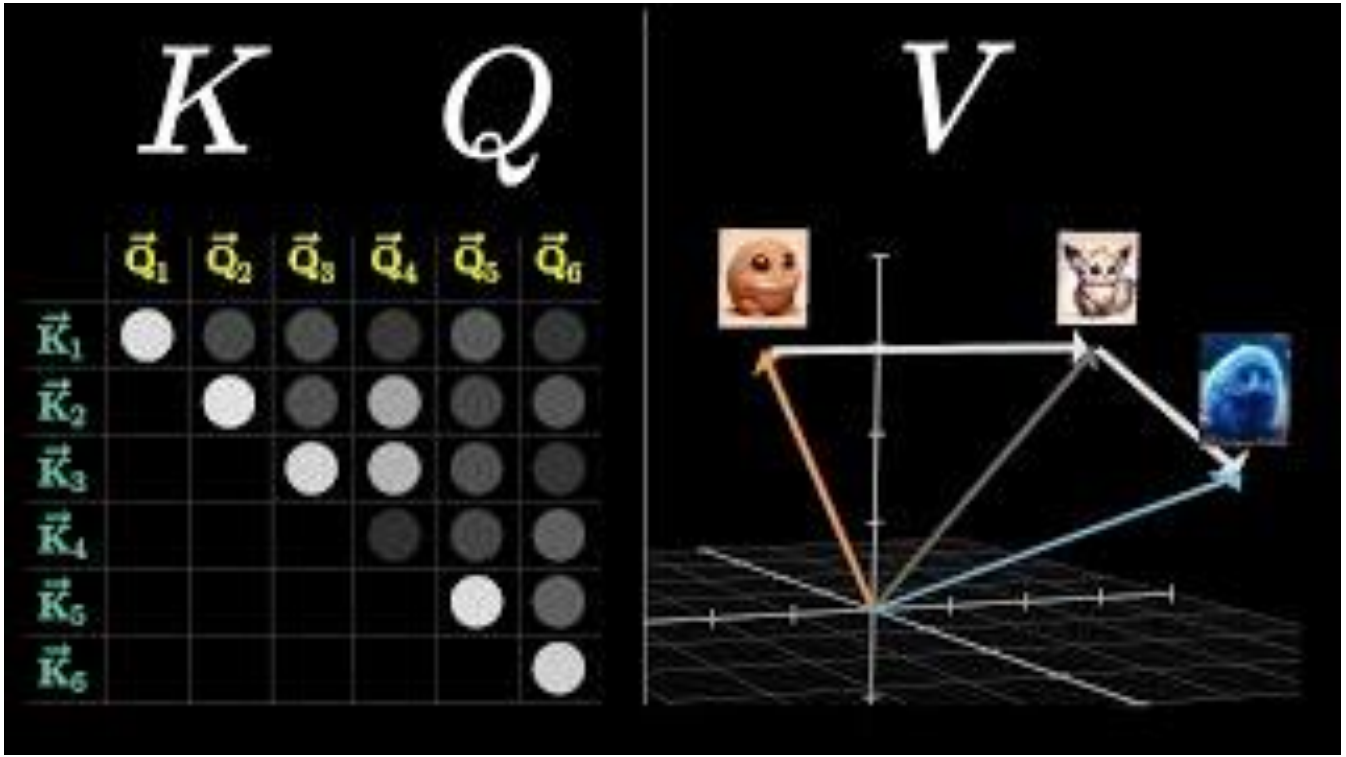
# Self-Attention Recap



# For a detailed explanation

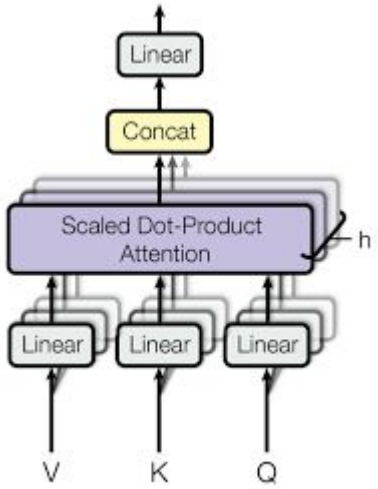


# 3Blue1Brown



# Multi-head Attention

- Multi-head Attention runs **multiple self-attention** mechanisms in **parallel**, allowing the model to focus on **different aspects** of the input **simultaneously**
- Different attention heads can learn to focus on **different linguistic features** (syntax, semantics, entities, etc.)
- After computing attention, there's also a **final output** projection matrix **WO** with dimension **[d_model, d_model]** that combines the outputs from all attention heads back into a single representation.

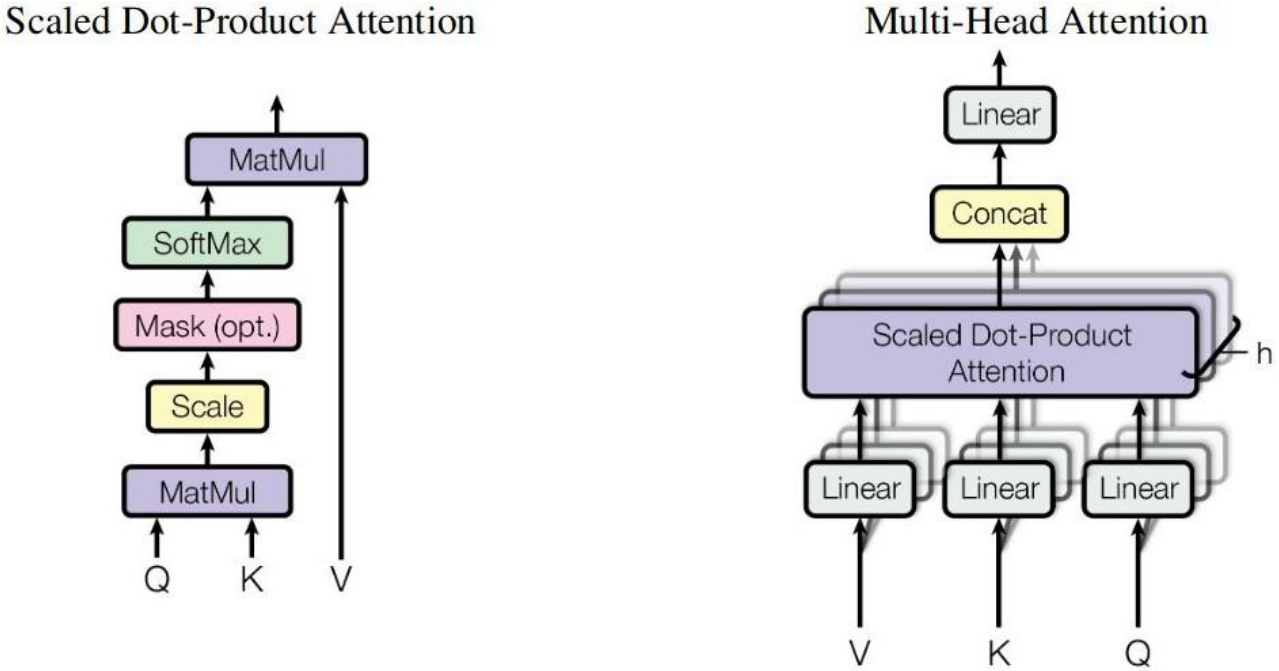


# Multi-head Attention

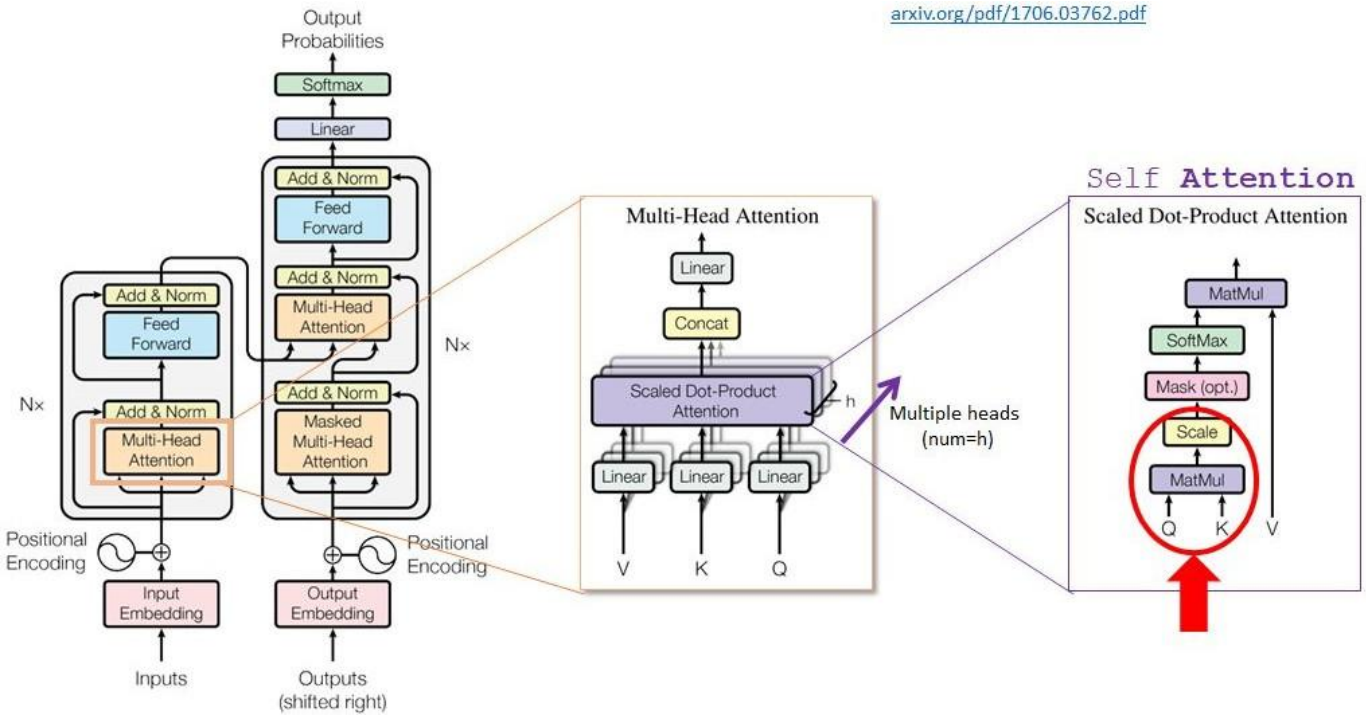
When implementing multi-head attention, we typically:

1. **Project** the input (dimension d_model) to Q, K, and V (each with dimension d_model)
2. **Split** each into num_heads pieces (each with dimension d_model/num_heads)
3. **Compute attention** separately for each head
4. **Concatenate** the results and project back to dimension d_model

# Multi-head Attention



# Where is Self-Attention and Multi-Head attention?

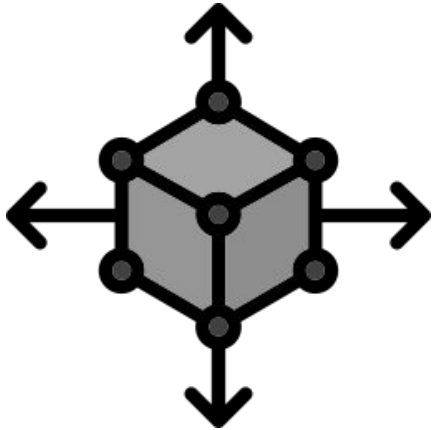


# Let’s Talk Dimensionality

We usually take the numbers suggested by the original Transformer paper as the baseline for our understanding and implementation.

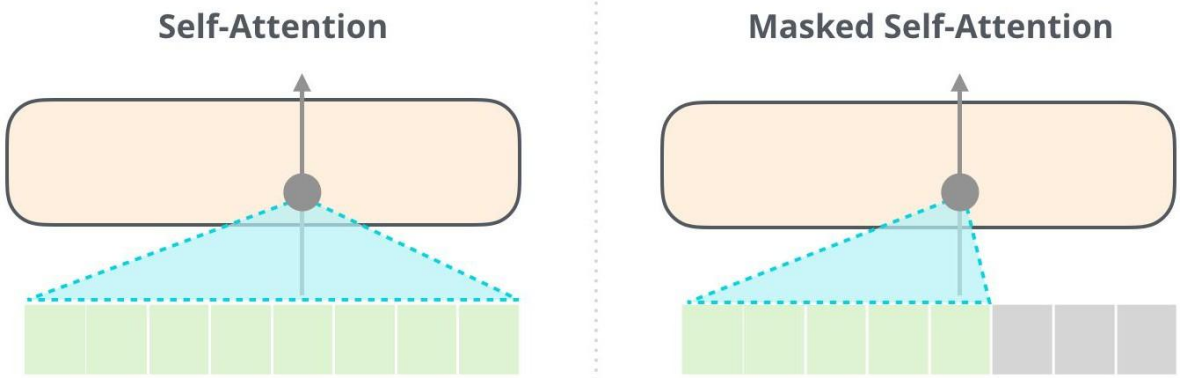
In the original "Attention is All You Need" paper:

- **d_model = 512** (the overall embedding dimension)
- **num_heads = 8**
- So each head had **d_k = d_model/num_heads = d_v = 64**



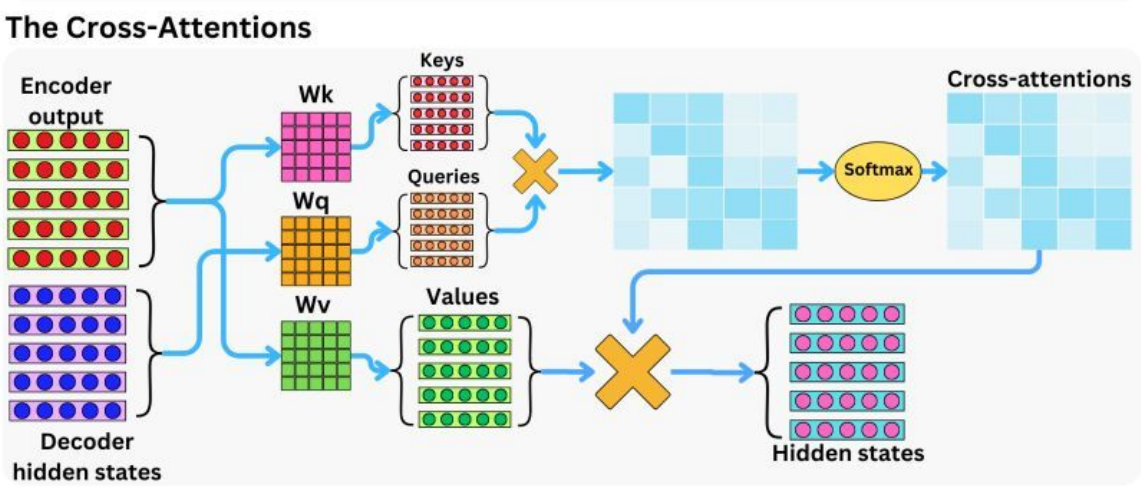
# Masked Attention

- **Concept:** Prevents a token from attending to future tokens in a sequence.
- **Implementation:** Sets attention scores for future tokens to near-zero values.
- **Usage:** Common in transformer decoders and BERT pre-training.
- **Outcome:** Preserves the temporal nature of sequences, ensuring models can't "cheat" by seeing the future.



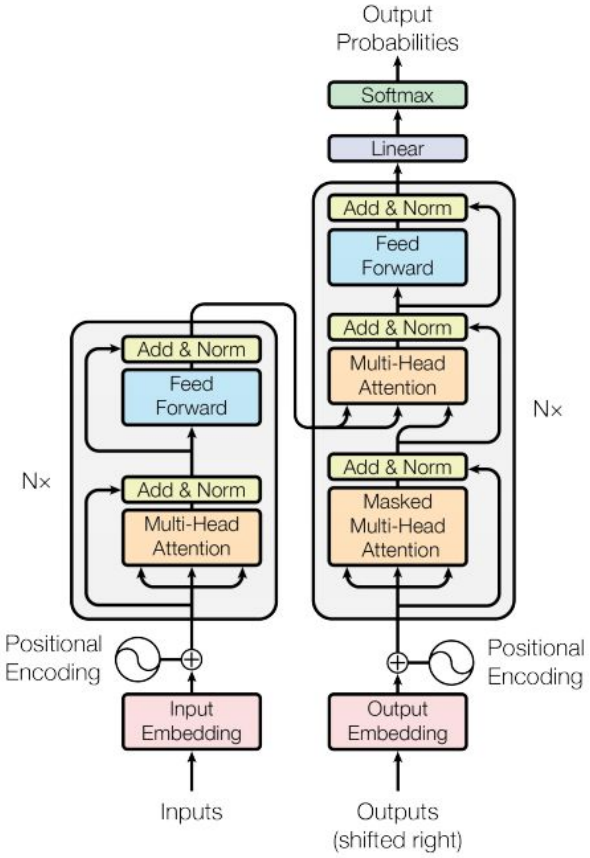
# Cross-Attention

- Present in the decoder block, **cross attention** is where elements from one sequence attend to elements in a different sequence
- **Multiple queries** instead of one (sequence 1), and the keys and values are of a **different sequence** (sequence 2)



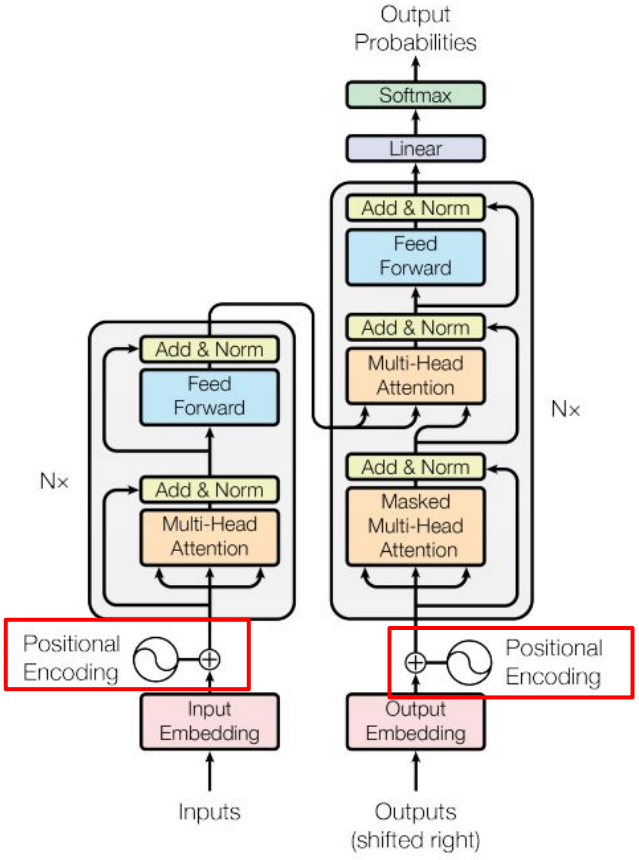
# Transformer Architecture

Now that we have covered the multi-head attention blocks, the rest of the transformer is made up for normalization and feed forward neural network blocks.



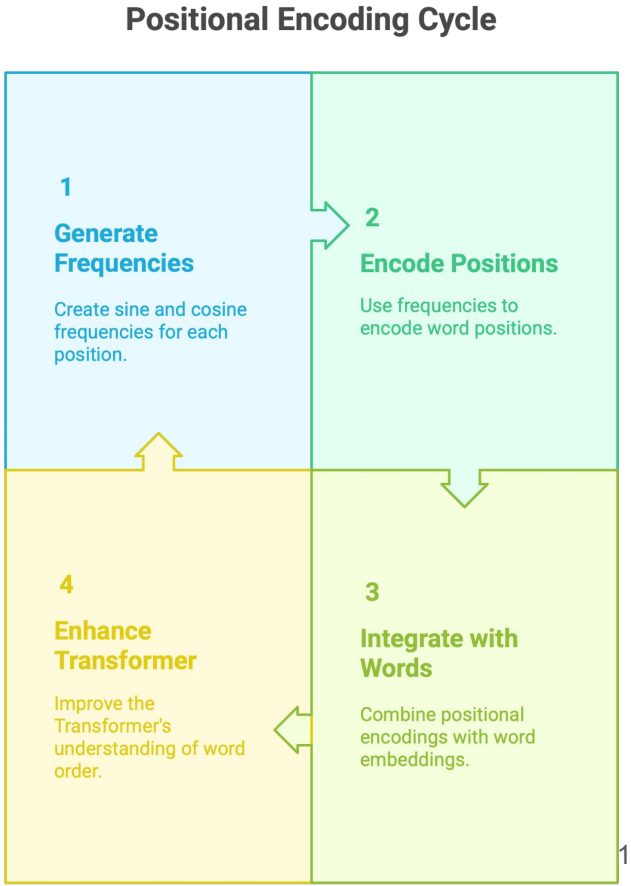
# Transformer Architecture

Except- what are these ‘positional encoding’ additions?



# Positional Encoding

- Since the Transformer architecture does not inherently capture the order of words in a sequence, **positional encoding** is used to provide information about the position of each word.
- Common methods include using sine and cosine functions with different frequencies to represent the position.

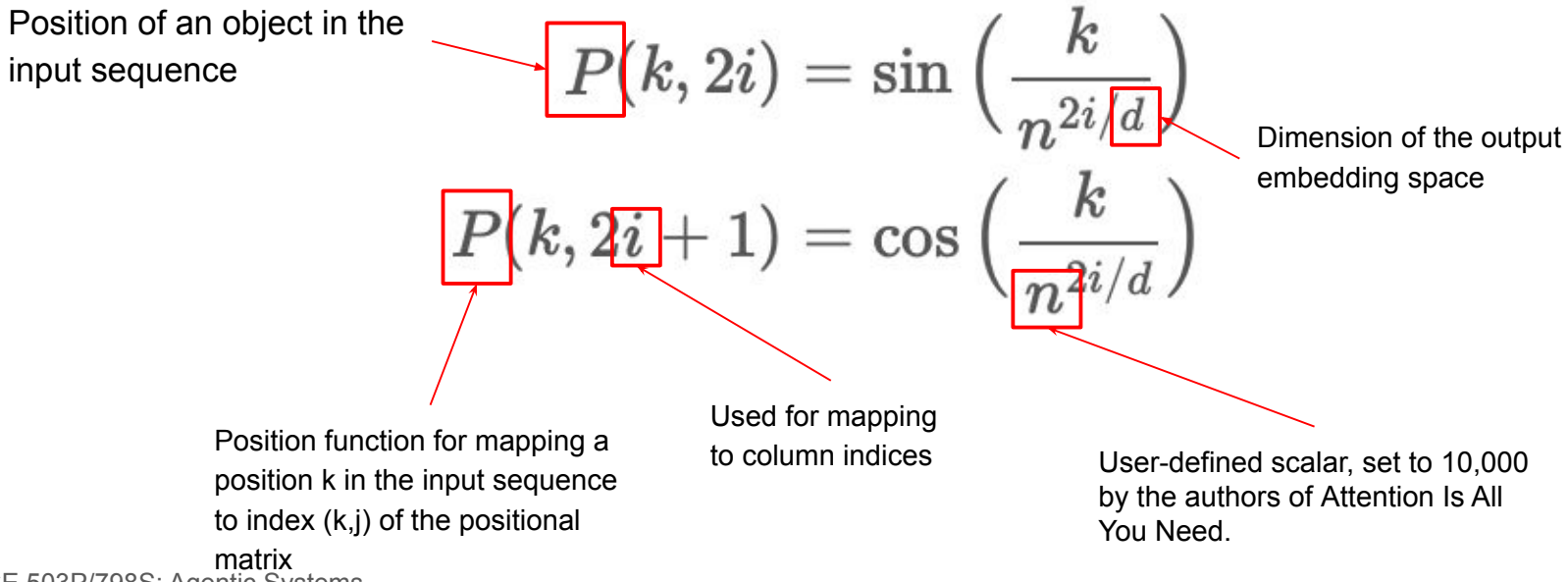


# Positional Encoding

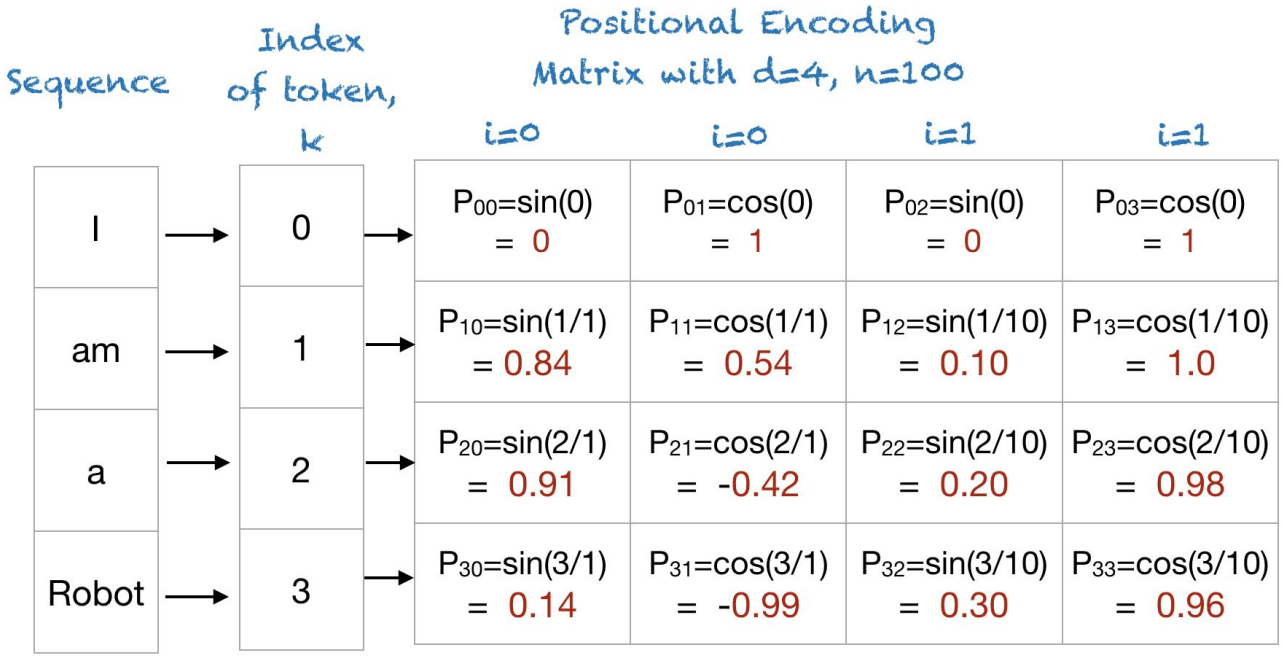
- There are many reasons why a single number, such as the index value, is not used to represent an item's position in transformer models.
- For **long sequences**, the indices can **grow large in magnitude**
- If you **normalize** the index value to lie between 0 and 1, it can create problems for variable length sequences as they would be **normalized differently**.

# Positional Encoding

Suppose you have an input sequence of length  and require the position of the object within this sequence. The **positional encoding** is given by **sine** and **cosine** functions of varying frequencies:



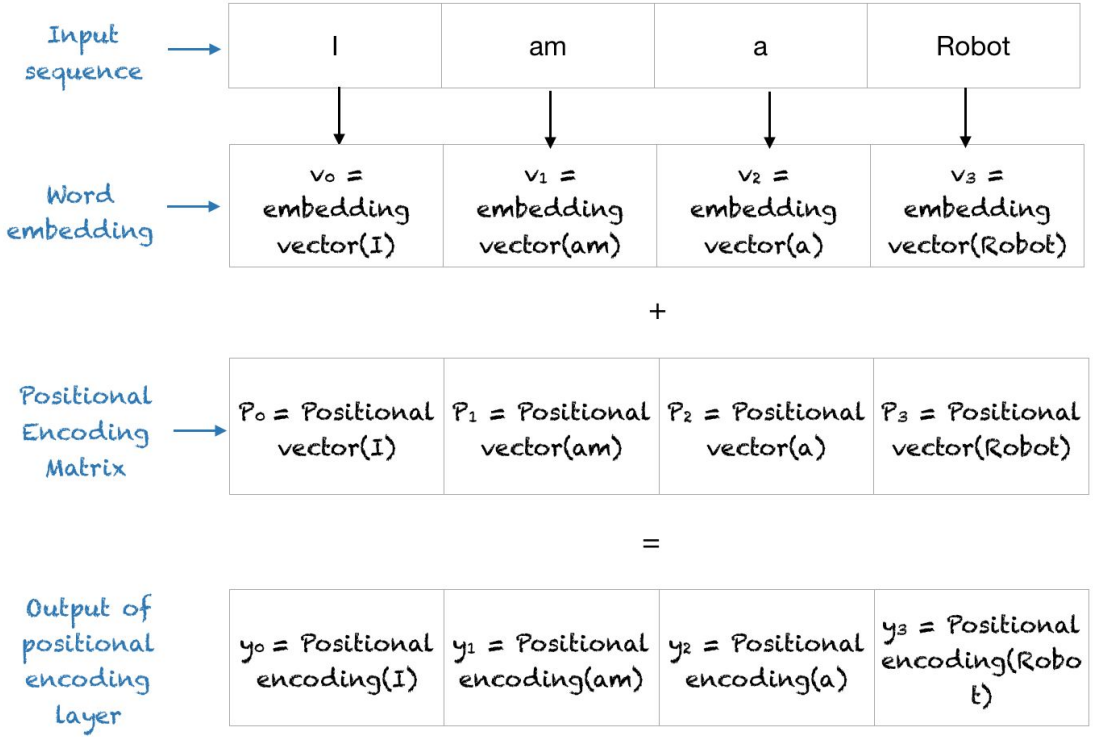
# Positional Encoding




Positional Encoding Matrix for the sequence 'I am a robot'

https://machinelearningmastery.com/a-gentle-introduction-to-positional-encoding-in-transformer-models-part-1/

# Positional Encoding




https://machinelearningmastery.com/a-gentle-introduction-to-positional-encoding-in-transformer-models-part-1/

# Hands on: Explore different transformer-based models in ‘Transformers on HuggingFace.ipynb’


AMERICAN UNIVERSITY OF BEIRUT

# Thank you!