AI Boom ∝ TRANSFORMERS Attention is all you need.

- **Also a Seq2Seq model**, but with a **different architecture**.
- Uses **self-attention** instead of recurrence.
- Encoder produces a **context matrix** (hidden states for all tokens).
- Decoder generates outputs step by step, attending to the **whole context matrix** (no bottleneck).
- Handles long-range dependencies much better.

Goal of Transformer = Encode Decode with self attention

1. Word Embeddings ; Dense vector Representation that are semantic aware (similar words = similar vectors).

- Dense Vector Representation = continuous words
- Semantic Awareness = words with similar embedding have similar vectors.

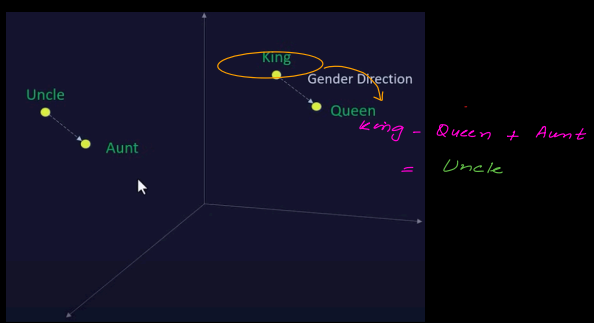
e.g., word2vec by Google → Google News = 100 bn words
Glove by Stanford

Imp Each word is Represented by 300d vector.

	battle	horse	king	man	queen	..	woman
authority	0	0.01	1	0.2	1		0.2
event	1	0	0	0	0		0
has tail?	0	1	0	0	0		0
rich	0	0.1	1	0.3	1		0.2
gender	0	1	-1	-1	1		1

king − man + woman = Queen vector.

We actually don't know what are the features as they've been represented by DL.



Similar words are close to each other e.g.,

Semantic Aware
Not Context Aware.

Imp

These are STATIC embeddings → fixed embeddings

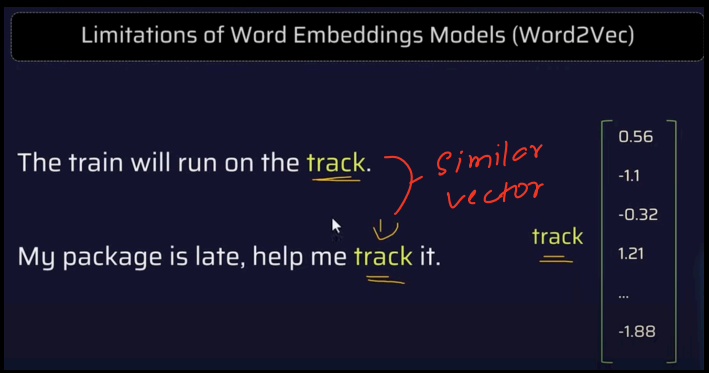
(Fast Text)

Word2Vec GloVe

These static embeddings aren't enough for GPT

🔷 Static Embeddings (Word2Vec, GloVe, FastText)

- Each word has **one fixed vector**, regardless of context.
- Example:
  - `"train"` in *"the train is late"* and *"I train daily"* → **same vector**.
  - `"track"` in *"the train is on track"* vs *"I am on track"* → **same vector**.
- They are **semantic-aware**, meaning *similar words have similar vectors* (king/queen, man/woman).
- But they **cannot adapt to context** → "bank" (river bank vs. financial bank) has the **same embedding**.



NOTE; Static embeddings are not enough for GPT, we need contextual embeddings.

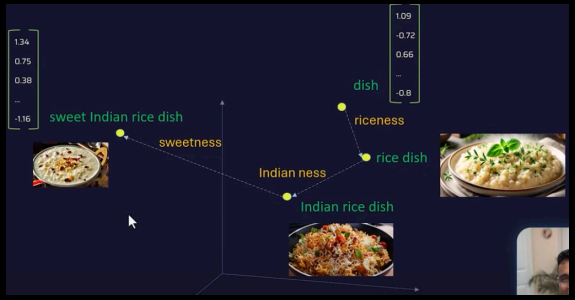
CONTEXTUAL EMBEDDING

🔷 Contextual Embeddings (ELMo, BERT, GPT)

- Embeddings depend on **the surrounding words** (context).
- `"track"` in:
  - *"train is on track"* → embedding shifts towards "railway".
  - *"I'm on track for success"* → embedding shifts towards "progress".
- Generated by **transformers** that use self-attention to look at the whole sentence.
- That's why models like **GPT** don't use Word2Vec/GloVe — they *learn embeddings dynamically at runtime*.

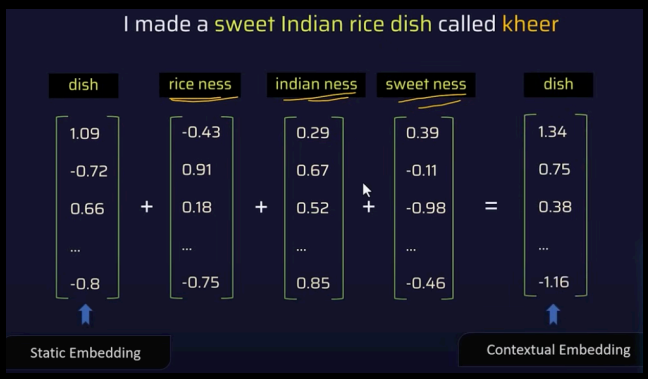
✅ So the truth is:

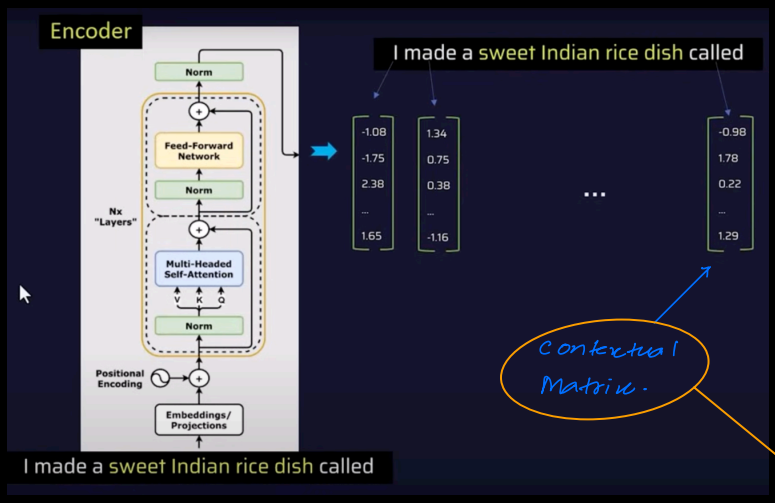
- Word2Vec / GloVe = **Static, semantic-aware** (fixed meaning per word).
- BERT / GPT embeddings = **Dynamic, contextual** (meaning changes with sentence).

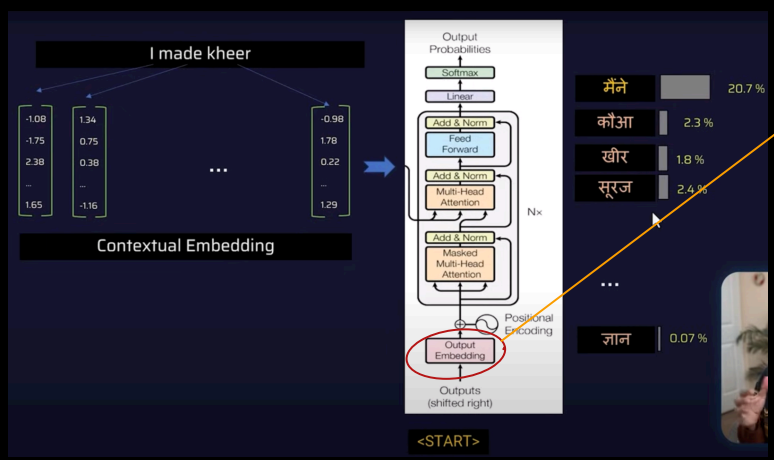


Different
1. Semantic Aware
2. Context Aware.

→ Mathematically, these static embeddings are added together to create Contextual embedding

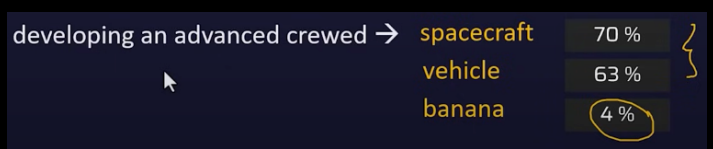






At each stage we have two inputs contextual matrix and last output.

example ;



Imp

Simple Explanation

"In Transformers, the encoder generates a context matrix (all hidden states for input tokens). The decoder uses this matrix along with its previous outputs. At each step, the attention mechanism allows the decoder to selectively focus on the most relevant parts of the context matrix, rather than relying on a single fixed vector."

Context Matrix + Last Output

Bi Directional Encoder Representations of Transformer

Google

BERT

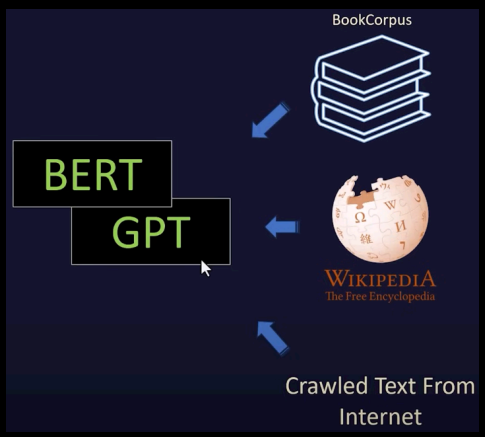
Open AI.

GPT

Generative Pretrained Transformer

Encoder only

Decoder only



These models are fed with huge amounts of data.

Model	Embedding Type	Vector Size (typical)	Context-Aware?	Architecture Type	Owner
Word2Vec	Static word embeddings	100–300	❌ No	Not a transformer (Shallow NN)	Google (2013)
GloVe	Static word embeddings	50–300	❌ No	Not a transformer (Matrix Factorization)	Stanford (2014)
ELMo	Contextual (LSTM-based)	1024	✅ Yes	RNN (Bi-directional LSTM)	Allen Institute (2018)
BERT	Contextual (Transformer Encoder)	768 (Base), 1024 (Large)	✅ Yes	Transformer **Encoder-only**	Google (2018)
GPT-3/4	Contextual (Transformer Decoder)	12,288 (GPT-3), ~8k–32k (GPT-4)	✅ Yes	Transformer **Decoder-only**	OpenAI (2020/2023)
T5	Contextual	768–1024 (Base–Large), up to 11B	✅ Yes	Transformer **Encoder–Decoder**	Google (2019)

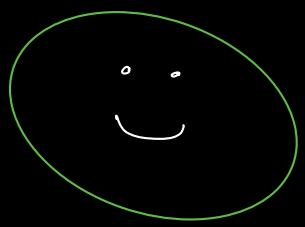
V. Imp !!

✅ Encoder–Decoder in Transformers

Researcher @ Google brain

Yes — the **original Transformer model (Vaswani et al., 2017)** was an **encoder–decoder architecture**, designed mainly for **sequence-to-sequence tasks** (like machine translation).

Let's Learn in Detail



QKV

TRANSFORMERS → Self Attention

→ Attention is all you need

1 Core Concept

- Introduced in **"Attention is All You Need" (Vaswani et al., 2017)**.
- Transformers completely **remove recurrence (RNNs/LSTMs/GRUs)**.
- They rely **only on attention mechanisms** (especially **self-attention**) to model relationships between tokens.
- This makes them **parallelizable, scalable, and better at capturing long-range dependencies**.

👉 One line: *Transformers use attention to understand sequences, without needing recurrence.*

2 Architecture

A standard Transformer has **two parts**:

- **Encoder:** Processes input sequence → produces contextual representations for each token.
- **Decoder:** Generates output sequence, attending to both its own past outputs and encoder outputs.

Each block is built from:

- **Self-Attention Layer** → lets each token "look at" all other tokens in the sequence.
- **Feedforward Network** → adds non-linearity & learning power. → Softmax
- **Layer Normalization + Residual Connections** → stabilizes training.
- **Positional Encoding** → adds order information (since there's no recurrence).

3 Key Mechanism: Self-Attention

For each token, self-attention calculates:

- **Query (Q), Key (K), Value (V) vectors**
- Attention score = similarity(Q, K) → tells how much one token attends to another
- Weighted sum of values → gives new representation

👉 This allows a word like "bank" to attend differently depending on context ("river bank" vs "bank account").

4 Advantages Over RNN/LSTM/GRU

Feature	RNN/LSTM/GRU	Transformer
Processing	Sequential (slow)	Parallelizable (fast)
Long dependencies	Hard to capture	Handled easily via self-attention
Information bottleneck	Fixed context (improved by attention)	Each token attends to all tokens directly
Scalability	Struggles with large data	Scales to billions of parameters

5 Applications

- **NLP:** Translation (Google Translate), Summarization, Question Answering, ChatGPT 😉
- **Vision:** Vision Transformers (ViT) for image classification.
- **Multimodal:** Models like CLIP, DALL·E, Gemini.

Imp : Transformer was initially introduced to translate one language to other

6 Analogy (Easy to Explain)

- RNN = reading a book word by word (slow, memory fades).
- LSTM = smarter note-taking but still sequential.
- Encoder–Decoder with Attention = translator with a spotlight that can look back at pages.
- **Transformer = everyone in the class can talk to everyone else instantly (global communication at once).**

✅ Quick One-Liner for Interview:

"Transformers are deep learning models that replace recurrence with self-attention, allowing every token to directly attend to all others in parallel. This makes them faster, better at handling long dependencies, and the backbone of modern NLP models like BERT and GPT."

GPT → 175 B parameters.

Generative — Generates Data

Pre Trained — Trained on Large Data

Transformer: (self attentive) — A neural network that is based on self attention

Models

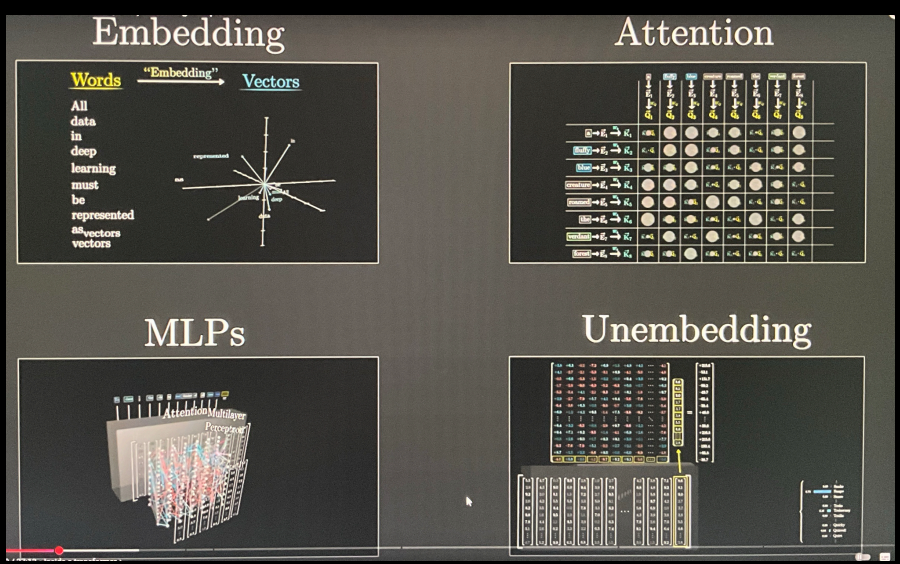
Text to Voice

Text to Image (Dalle, Midgourney)

Text to Text (GPT)

Basic Concept is Prediction Model.

Inside the Transformer.



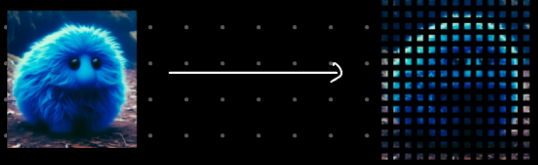
Process :

Step 1.

1. **Tokens** ; Breaking sentences, images, voice into smaller parts → text into words.

a. **Text** → sentence in words

b. **Image** → small vectors



c. **Voice** → into smaller parts

Step II ; **Vectorization** : Embedding matrix. (semantic aware)

Each of these tokens is then associated with vectors.

→ The model works with matrix (same words same vectors)

embedding matrix → [vocab size x Embedding Dimension]

example in GPT5 = 50k x 12k ≅ 600m parameters

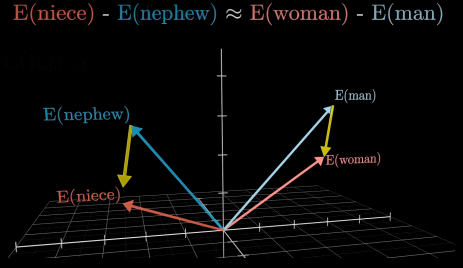
Embedding Dimension = 300, then each word is represented by vector of 300 D.

v. Imp.

Remember ; The embeddings are still static @ this stage.

e.g., **Train** will always have similar vector.

example ; These static embeddings are semantic aware i.e., they understand word meaning



example ;

plural will have higher values than singular ones

Octopus = -2.36

Octopi = +1.38

Imp.

king − man + woman ≅ Queen

Step III Positional Encoding

embeddings for position

🔷 Why do we need positional embeddings?

Transformers only use **self-attention**, which treats the input as a **set of vectors**.
But sets don't have order!

Example:

```
css

I love cats
cats love I
```

Copy code

If we only use embeddings without position info, both would look the same bag of vectors.
⚡ The model wouldn't know the difference between subject and object.

So → we inject **position information** into embeddings.

Example:

```
scss

Pos(1) → [0.11, 0.05, -0.02, ...]
Pos(2) → [-0.03, 0.07, 0.01, ...]
Pos(3) → [0.02, -0.06, 0.04, ...]
```

When embedding a token, you **add** the word embedding + positional embedding:

```
scss

Embed("I") + Pos(1)
Embed("love") + Pos(2)
Embed("cats") + Pos(3)
```

Types ⟶ Learned positional Embeddings

Sinosidal positional Embeddings

🔷 Learned positional embeddings

Data based fixed length

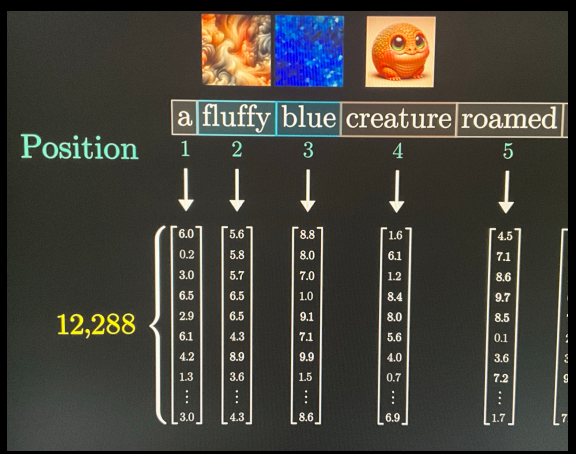
- **Trainable** (just like word embeddings).
- Each position index (1, 2, 3, ... up to max length) gets its own vector that is updated during training.
- **Advantage**: The model can learn **data-specific position signals**.
- **Disadvantage**: Limited to the **maximum sequence length** you trained with — can't naturally extend to unseen longer inputs.

🔷 Sinusoidal positional embeddings

Sin cosine formula.

Generic

- **Fixed** (formula-based, not trainable).
- Based on sine & cosine functions at different frequencies.
- **Advantage**: Can generalize to **longer/unseen sequence lengths** (since you can always compute new values).
- **Disadvantage**: The model cannot adapt positional signals to the dataset — it just gets the generic sinusoidal pattern.



✅ So your summary is correct:

- **Sinusoidal** = fixed formula, good for unseen lengths.
- **Learned** = data-dependent, flexible, but capped at training length.

v.v.imp.

Step IV Contextualization.

Self attention

🔷 Contextualization = Self-Attention

At this stage, embeddings are **static** (context-free).
We want them to become **dynamic** (context-aware).
This happens in **self-attention layers.**

```
I love cats
```

Right now:

```
ini

"I" = [vector1]
"love" = [vector2]
"cats" = [vector3]
```

Self-attention lets **each word look at other words** and decide how much they matter to its meaning.

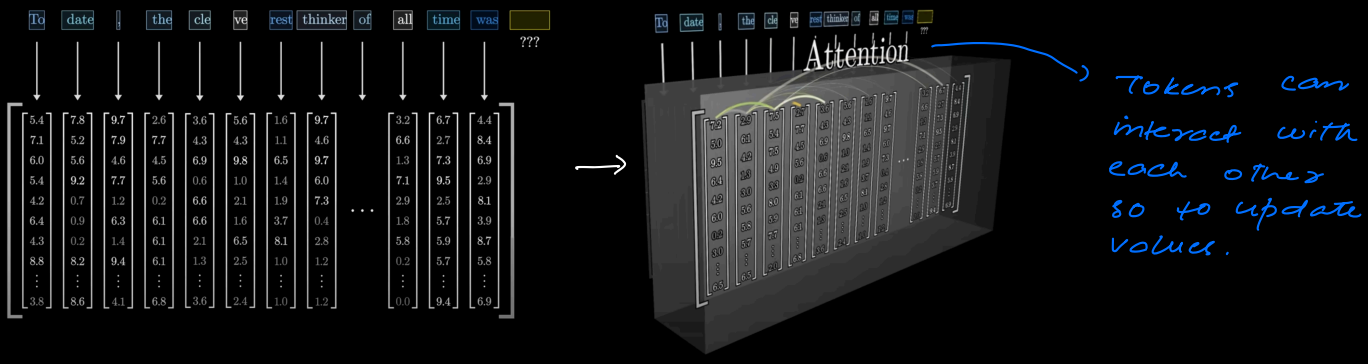
- For "love", maybe it pays attention mostly to "I" (subject) and "cats" (object).
- For "I", it might care mostly about "love".
- For "cats", it might care about "love" more than "I".

So each embedding is updated to a **weighted mix of all embeddings.**

Imp. Static Embedding Semantic Aware Not content Aware.

imp

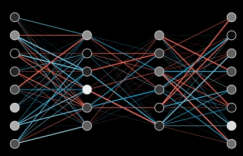
ATTENTION; allows embeddings to interact with each other and pass on information.

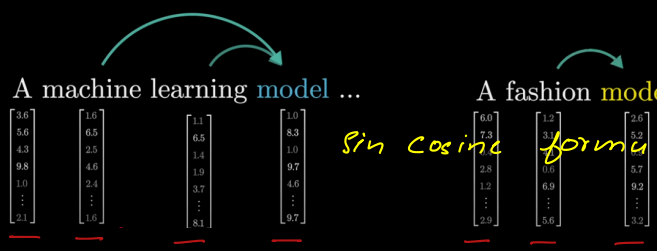


imp

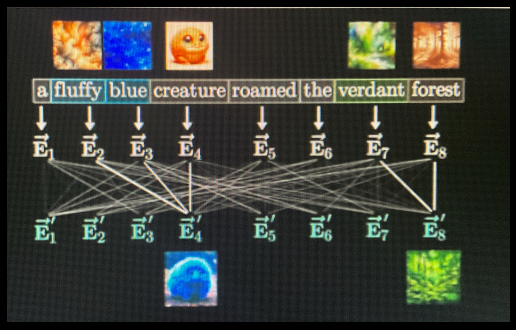
Attention helps understanding context

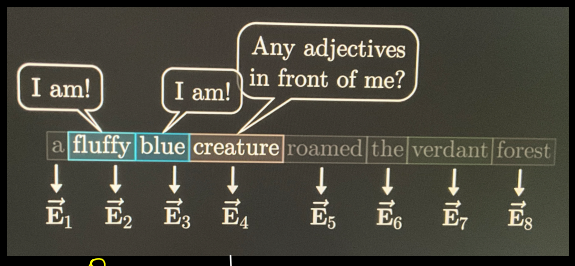
example.





Understanding Attention mechanism





imp

1. Intuition First

- **Query (Q):** What I'm *looking for* (like my question).
- **Key (K):** What each token *offers as an answer*.
- **Value (V):** The actual information to pass along once relevance is decided.

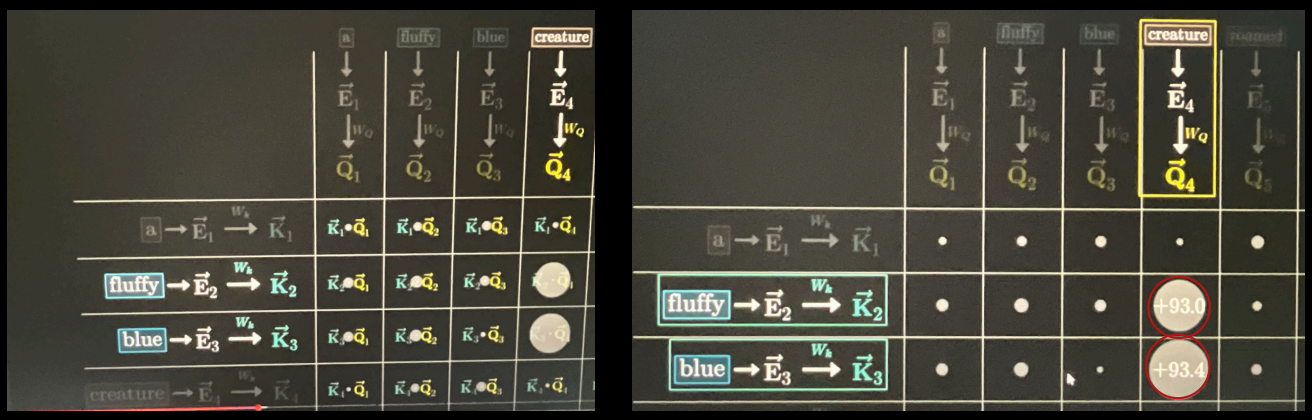
So:
👉 Query asks → Keys are checked → Values are returned (weighted by how relevant each Key is to the Query).

2. The Dot Product Q · K

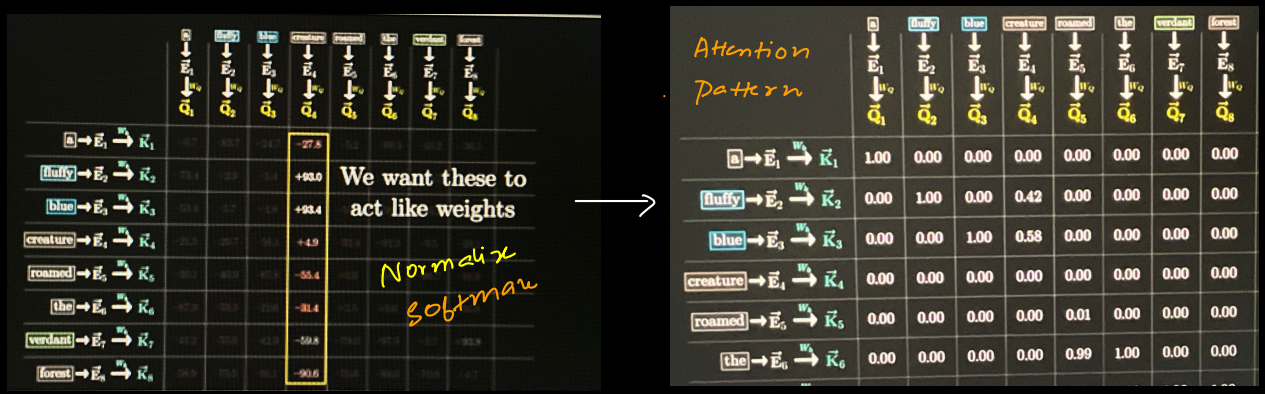
- We take the dot product between a **query vector (Q)** and each **key vector (K)**.
- Why? Because dot product is a measure of **similarity** (higher = more aligned).
- If **Q · K is large** → the token is *highly relevant* to what we're asking.
- If **Q · K is small/negative** → not relevant.

Example:

- Query = "What is the capital of France?"
- Keys = embeddings for tokens like *Paris, London, dog, table*.
- The dot product between Q and K(*Paris*) wil highest → so "Paris" gets most attention.



In simple, fluffy & Blue attend to Creature



3. Softmax Step

After computing all Q·K scores for tokens:

- We apply **softmax** to turn them into probabilities (attention weights).
- So tokens sum to 1, with the most relevant ones getting higher weight.

---

4. Weighted Sum with Values (V)

- Multiply attention weights with the Value vectors.
- Then take the **weighted sum** → that gives the new contextual embedding for this token.

5. Formula

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

- QK^T: dot product similarity between queries and keys
- divide by $\sqrt{d_k}$: scaling (prevents exploding values in high dimensions)
- softmax: converts scores → attention weights
- multiply with V: return the weighted info

- **Q · K^T** = similarity
- **softmax(Q · K^T)** = attention distribution (weights)
- Multiply those weights with **V** = updated contextual embedding

imp In GPT 3 ; 96 heads

Keys ; Dimensions = 12K
Rows = 128
Parameters = 1572864

Step by Step (Self-Attention for one query)

1. **Dot product:**
   - $Q \cdot K^T$ → similarity scores.
   - These are raw numbers (not yet probabilities).
2. **Normalization with softmax:**
   - Apply softmax → now they **are weights** (probabilities that sum to 1).
   - Example: [0.1, 0.7, 0.2] means the query pays 70% attention to token 2.
3. **Apply weights to Values (V):**
   - Multiply these weights with the corresponding **Value vectors**.
   - Example: $0.1V_1 + 0.7V_2 + 0.2V_3$.
   - This gives the **new contextual embedding** for that query token.

Total weights: 175,181,291,520
Organized into 27,938 matrices

GPT-3

Embedding	12,288 (d_embed) * 50,257 (n_vocab) = 617,558,016
Key	128 (d_query) * 12,288 (d_embed) = 1,572,864 per head
Query	128 (d_query) * 12,288 (d_embed) = 1,572,864 per head
Value↓	128 (d_value) * 12,288 (d_embed) = 1,572,864 per head
Value↑	12,288 (d_embed) * 128 (d_value) = 1,572,864 per head
Up-projection	6,291,456
Down-projection	
Unembedding	50,257 (n_vocab) * 12,288 (d_embed) = 617,558,016

SINGLE HEAD VS MULTI HEAD ATTENTION

⚖️ Single-Head vs Multi-Head Attention

Aspect	Single-Head Attention	Multi-Head Attention
How many projections?	One set of Q, K, V	Multiple sets of Q, K, V (e.g., 8 heads)
Focus	Captures *one type* of relationship	Captures *multiple relationships* in parallel
Example Sentence	*"The cat sat on the mat."*	*"The cat sat on the mat."*
What it learns?	Learns 1 pattern: e.g., **"cat ↔ sat"** (subject–verb)	Head 1: **"cat ↔ sat"** (subject–verb)Head 2: **"on ↔ mat"** (preposition–object)Head 3: **"The ↔ cat"** (article–noun)
Output Representation	One contextual embedding per token	Richer embedding (concat of multiple heads) per token
Analogy	One spotlight → sees one relationship	Many spotlights → see different relationships

e.g., GPT 3 has 96 heads

✅ **In short:**

- **Single-Head** = one perspective.
- **Multi-Head** = multiple perspectives combined

So, in multi headed, Original Embedding will have many changes (from multi heads)

$$\text{Original embedding } \vec{\mathbf{E}}_i + \Delta\vec{\mathbf{E}}_i^{(1)} + \Delta\vec{\mathbf{E}}_i^{(2)} + \Delta\vec{\mathbf{E}}_i^{(3)} + \Delta\vec{\mathbf{E}}_i^{(4)} + \cdots$$

Step V Add and Norm

1. Residual Add 2. Normalization

🔷 What is Add & Norm?

Each Transformer sub-layer (like **self-attention** or **feed-forward**) is wrapped like this:

$$\text{Output} = \text{LayerNorm}(X + \text{Sublayer}(X))$$

Normalization

- `X` = input to the sublayer.
- `Sublayer(X)` = what the sublayer does (e.g., self-attention, FFN).
- We add `X` back (residual/skip connection).
- Then apply **Layer Normalization**.

So Residual add is like keep original inputs (input embeddings = static embed + position) and also keep sublayer (output of self attention)

Imp Residual Add = Input Layer + Sub layer

↳ so that our model doesn't stop

🔷 Why Residual Connection (Add)?

Residual = **skip connection** (idea borrowed from ResNets in computer vision).

- Prevents information loss: the model can always "fall back" on input if the sublayer doesn't help.
- Helps gradient flow (avoids vanishing gradients in deep networks).
- Lets deeper networks train more easily.

→ Layer Norm = to normalize the sum,

mean = 0 var = 1

Solves vanishing / exploding gradients

🔷 What does Norm do?

- Imagine your numbers (embeddings) sometimes explode (too large) or vanish (too small).
- **LayerNorm** rescales them so they're balanced → mean = 0, variance = 1 (more stable).
- It prevents some tokens from overpowering others and helps training converge smoothly.

Imp

🔷 Analogy

Think of it like cooking:

- **Add** = combine old recipe (original input) + new spices (attention output).
- **Norm** = taste-test and adjust seasoning so no ingredient is too salty/spicy → balanced flavor.

Step VI Feed Forward Layer (MLP)

In the **feed-forward block of a Transformer** (after attention), the process is:

1. **Input (hidden states)** → come from the attention layer.
2. **First Linear Layer (Perceptron 1)** → expands the dimension (e.g., from `d_model` → `4 * d_model`).
3. **Activation Function (GELU / ReLU)** → adds non-linearity so the model can capture complex patterns.
4. **Second Linear Layer (Perceptron 2)** → projects back down (from `4 * d_model` → `d_model`).

512 → 2048

So yes, you can say:

↳ contraction

- The feed-forward is **two perceptrons (linear layers) with GELU/ReLU in between.**
- These are applied to each token's hidden state **independently** (no mixing across tokens here — mixing happens in attention).

👉 In short: **FFN = Linear → GELU → Linear.**

FFN = Expansion - Activate - Contraction

Q// why do we expand.

Expansion is like giving wider space to learn complex patterns by ReLU or GELU

✅ So in short:
The FFN expands to **learn richer transformations**, then de-expands to **maintain manageable model size and consistency across layers.**

Q// ReLU vs GELU → preffered for modern NLP Transformers.

↓ Rectified Linear unit

→ Gaussian Error Linear Unit

imp.

It's an **activation function** (like ReLU, sigmoid, tanh) but smoother and works better in Transformers.

🔷 **ReLU** says:

- If x < 0 → output 0
- If x ≥ 0 → output x

(Problem: it's sharp, creates a hard cutoff at 0).

🔷 **GELU** is softer:

- Instead of making a hard cutoff, it "smoothly gates" the negative values.
- Small negative values are not completely killed — they are just **downweighted.**
- Formula involves the Gaussian distribution (hence the name).

👉 Intuition:
Think of GELU as saying *"If input is positive, let it pass; if it's slightly negative, maybe keep a little of it; if very negative, then drop it."*

Step VII : Add & Norm again

🔑 Why Add & Norm after FFN?

1. **Residual helps preserve info**
   - The FFN is just a non-linear transformation.
   - Without residual: the model *only* sees the transformed features.
   - With residual: the model sees **original (attention output) + FFN-transformed version** → richer representation.
2. **Normalization keeps things stable**
   - The FFN output can have a very different scale than the residual input.
   - LayerNorm ensures the sum doesn't explode or vanish, keeping training smooth.
3. **Symmetry across sublayers**
   - Each sublayer (Attention, FFN) gets the *same treatment*: `Input + Sublayer → Norm`.
   - This consistency stabilizes very deep stacks (Transformers often have 12–96 layers).

Step VIII Stacking

- Steps 4-7 are repeated N times
- To refine contextual embeddings more.

STEP IX Output Head

Output head = final step where the Transformer converts embeddings into the actual result (depends on the task).

Decoder only

- **Language Modeling (like GPT):** Softmax - probabilities
  The model takes the embeddings → passes them through a linear layer → applies softmax → gives probabilities of the next word/token.
- **Translation (original Transformer):** encoder → decoded
  The decoder looks at encoder embeddings (from the source language) → generates the next translated word step by step.

encoder only

- **Classification (like BERT fine-tuning):** Generates CLS token
  The special `[CLS]` token's embedding (summary of the whole text) → goes into a classifier → predicts label (e.g., positive/negative, spam/not spam).

Encoder vs Decoder in Transformers

- **Encoder**: Reads the whole input (good for understanding).
- **Decoder**: Generates text step by step (good for writing).

BERT → understands.

- **Only Encoder**.
- Takes in a sentence (or two), processes it, and outputs contextual embeddings.
- Used for **understanding tasks**: classification, sentiment, NER, QA (extractive).
  ✅ Example: "Is this review positive or negative?"

GPT → predict

- **Only Decoder**.
- Predicts the **next word** given previous words.
- Used for **generation tasks**: text completion, story writing, chatbots.
  ✅ Example: "Once upon a time..." → continues writing.

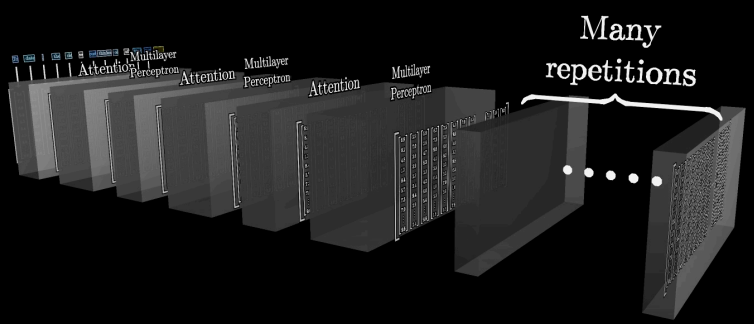
Summary of Transformer

Step	Process	Simple Explanation
1	**Tokenization**	Break input text into tokens (words, subwords, or characters) and map them to IDs.
2	**Embedding + Positional Encoding**	Convert tokens into vectors + add positional info so model knows word order.
3	**Self-Attention** (Q, K, V)	Each token looks at all others to gather context (who relates to whom).
4	**Add & Norm (after attention)**	Add skip connection (residual) and normalize for stable training.
5	**Feed-Forward Network (FFN)** (Non linarity)	Apply small neural network to refine each token's representation.
6	**Add & Norm (after FFN)**	Again add skip connection + normalize to keep info + stability.
7	**Stacking Layers**	Repeat steps 3–6 multiple times (e.g., 6 or 12 layers) for deeper understanding.
8	**Final Contextual Embeddings**	Each token now has a rich, context-aware vector representation.
9	**Output Head (Task-Specific)**	- GPT: embeddings → linear → softmax → predict next word. - BERT: use [CLS] token → classifier. - Translation: decoder attends to encoder → next word.

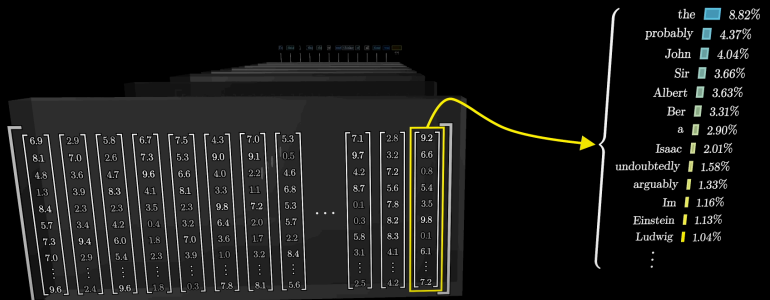
Good To know

Gpt 3 has 175b parameters (Tunable).

Process



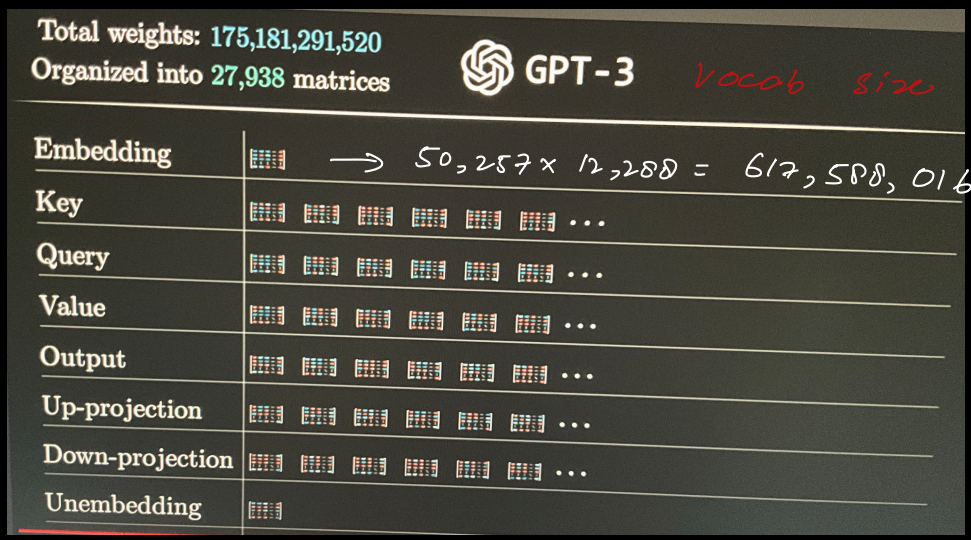
Output



Tunable parameters ⟷ weights

context size = 2048

→ In GPT 3, vector dimension = 12,288 d. ↳ GPT 3.



Vocab size = 50,257

→ 50,257 × 12,288 = 617,588,016

which means

Total parameters

= 50,257 × 12288

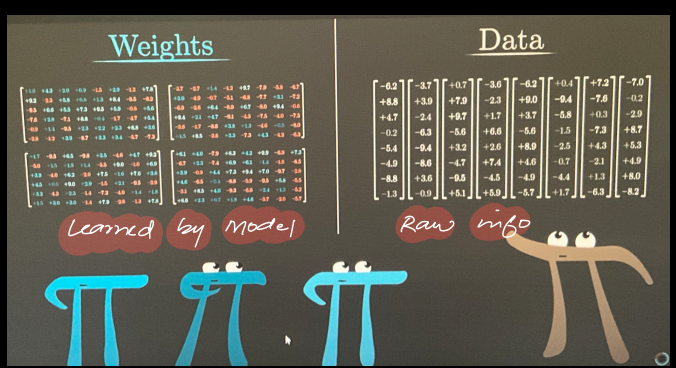
= 617,588,016

Q1 Difference b/w Data & weights

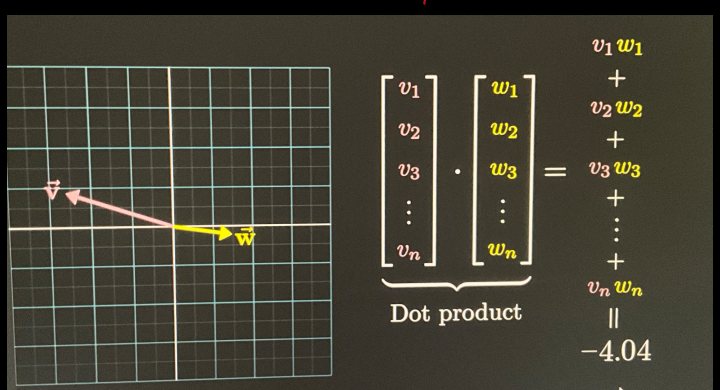
👉 In short:

- Data = what you train on (examples).
- Weights = what the model learns (knowledge extracted from the data).

weights are more imp for model.



Q1 What is DOT product. → multiplication of adjacent no.s



+ve when same direction

-ve when opposite Dir.

0 when perpendicular

Imp. plural one will have higher value

Softmax = Turns results into probability dist of 1 or Normalized

Temp if high → uniform dist
if less → more weight to big values.

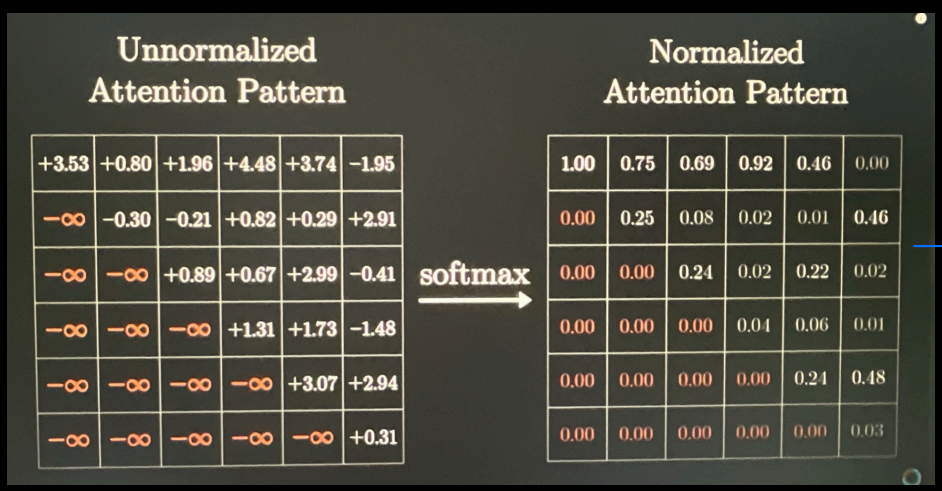
while, temp is less → Dynamic but very fast Nonsensical.

Imp: Api won't allow Temp less than 2

Imp: In GPT 5, Context size is 400,000

Q// How to do Masking in GPT?

Basically avoiding earlier tokens to impact / influence earlier ones. ⟶ Zero them out **$-\infty$**



Unnormalized Attention Pattern					
+3.53	+0.80	+1.96	+4.48	+3.74	−1.95
−∞	-0.30	-0.21	+0.82	+0.29	+2.91
−∞	−∞	+0.89	+0.67	+2.99	-0.41
−∞	−∞	−∞	+1.31	+1.73	-1.48
−∞	−∞	−∞	−∞	+3.07	+2.94
−∞	−∞	−∞	−∞	−∞	+0.31

softmax ⟶

Normalized Attention Pattern					
1.00	0.75	0.69	0.92	0.46	0.00
0.00	0.25	0.08	0.02	0.01	0.46
0.00	0.00	0.24	0.02	0.22	0.02
0.00	0.00	0.00	0.04	0.06	0.01
0.00	0.00	0.00	0.00	0.24	0.48
0.00	0.00	0.00	0.00	0.00	0.03

⟶ This is called masking

Q// Why do we have 2 Norm and Add?

1ST Norm & Add = Normal Inputs + Self attention Output

2ND Norm & Add = Self attention Output + FFN Output.