## What is Attention?

When we read a sentence, we don't understand each word in isolation — we look at the surrounding words to figure out what each word means. For example:

- "The **crane** flew away." → You think of a bird.
- "The **crane** lifted a car." → You think of a machine.

This is what attention allows AI models to do. It lets them focus on **relevant parts of the sentence** to understand what a word or token really means in context.

In transformers, this is called **self-attention**. Every word looks at every other word in the sentence and decides how important each one is. That importance is then used to build a richer, more meaningful representation of the word — one that captures its role in the full sentence.

## How is attention computed?

Self-attention is computed by taking all N input vectors in the rows and columns of an N x N matrix.

Each cell becomes a dot product of the row and column vector. Think of this cell showing the similarity between the row and column vector, with high values for closely aligned vectors and low values for dissimilar vectors.

This matrix is then multiplied for the input vectors again, resulting in a perturbation of the original vectors.

$$V_i = V_1 * \text{Similarity}_{(i,1)} + V_2 * \text{Similarity}_{(i,2)} + V_3 * \text{Similarity}_{(i,3)} + ... + V_N * \text{Similarity}_{(i,N)}$$

The resultant vector is pulled towards others based on how similar they are. Effectively, the crane vector is pulled towards the meaning of flew (the action of a bird) in the first sentence, and car (a machine) in the second sentence.

In this way, we are able to add contextual meaning to every vector through attention. As we stack 12 or 96 transformers together, we see higher-level context (sarcasm, situational awareness, innuendo, etc…) detected by transformers.

---

## What is KV Caching?

Let’s say you're using a language model to generate a sentence, one word at a time.

At every step, the model runs the attention mechanism for all the words until now.

Example:

I … *(1 x 1 attention matrix)*

I am … *(2 x 2 attention matrix)*

I am going … *(3 x 3 attention matrix)*

I am going to … *(4 x 4 attention matrix)*

…

This becomes **very slow** for long texts.

KV caching reuses previous computations by storing them in memory.

- Instead of redoing the full attention calculation every time, the model **stores the cell values** (from attention) for past tokens.
- When it generates the next word, it only needs to calculate the query for that word, and reuse the previously saved cell values.

This speeds things up massively. Most production-grade LLMs (like GPT-4, LLaMA, Mistral, etc.) use KV caching.

---

## Key Takeaways

- Attention helps a model understand what matters and how words relate to each other.
- KV caching helps the model do this efficiently, especially when generating longer texts.