



The Illustrated Transformer

A Complete Step-by-Step Explanation

Based on Jay Alammar's Illustrated Transformer

1. Origin of Attention & The Problem with RNN / LSTM

The attention mechanism was first introduced by Bahdanau et al. (2014) for neural machine translation. The Transformer architecture — which made attention the central mechanism — was introduced by Google researchers in the 2017 paper "Attention Is All You Need."

Why RNN and LSTM Failed for Long Sequences

In RNN and LSTM, the entire input sentence is compressed into a single fixed-size context vector called the hidden state. This created two fundamental problems:

```
Sentence: "The cat that sat on the mat was fat"
```

```
Step 1: process "The" → hidden state h1
```

```
Step 2: process "cat" → hidden state h2
```

```
...
```

```
Step 9: process "fat" → hidden state h9 (final)
```

```
Problem 1 — SEQUENTIAL PROCESSING:
```

```
Word 9 can only be processed AFTER word 8.
```

```
No parallelism possible → very slow training.
```

```
Problem 2 — MEMORY LOSS:
```

```
By step 9, h9 has mostly forgotten "cat" from step 2.
```

```
The whole sentence meaning is squished into one small  
vector → context from early words is lost.
```

```
Transformer Fix:
```

```
All words processed IN PARALLEL.
```

```
Every word attends to every other word directly.
```

```
No information loss regardless of sentence length.
```

2. Attention vs Self-Attention in Transformers

Transformers use both Self-Attention and Cross-Attention to solve the context problem:

Self-Attention:

ONE sequence attending to ITSELF.

Example: "The animal didn't cross because IT was tired"

→ "it" looks at all other words in the SAME sentence

→ learns that "it" refers to "animal"

Cross-Attention (original attention mechanism):

TWO DIFFERENT sequences attending to each other.

Example: English sentence ↔ French sentence

→ decoder (French side) looks at encoder (English side)

→ "chiens" focuses on "dogs" in the source sentence

3. Self-Attention — Complete Step-by-Step

Step 1 — Create Q, K, V Matrices

Each word embedding is multiplied by three separately learned weight matrices (WQ, WK, WV) to produce a Query, Key, and Value vector for every word.

Dimensions at Each Stage

Component	Shape	Notes
Word Embedding (X)	(seq_len × 512)	e.g., 2 words → (2 × 512)
Weight Matrix WQ	(512 × 64)	Learned during training
Weight Matrix WK	(512 × 64)	Learned during training
Weight Matrix WV	(512 × 64)	Learned during training
Query Matrix Q	(seq_len × 64)	$X * W_q$
Key Matrix K	(seq_len × 64)	$X * W_k$
Value Matrix V	(seq_len × 64)	$X * W_v$

Why dimension 64? The paper uses 8 attention heads.

$512 \text{ dimensions} \div 8 \text{ heads} = 64 \text{ dimensions per head}$.

This keeps the total computation constant regardless of how many heads are used.

For words "Thinking" and "Machines":

```
x1 (Thinking, 512-dim) × WQ (512×64) = q1 (64-dim)
x2 (Machines, 512-dim) × WQ (512×64) = q2 (64-dim)
```

Same multiplication done with WK → k1, k2

Same multiplication done with WV → v1, v2

Step 2 — Calculate Attention Scores

The score tells us how much focus word i should place on word j while encoding. It is calculated using the dot product between query and key vectors.

Calculating scores for word 1 "Thinking":

```
Score(Thinking vs Thinking) = q1 · k1
Score(Thinking vs Machines) = q1 · k2
```

Dot product measures similarity between vectors.

High score → these words are highly relevant to each other

Low score → less relevant

Full score matrix for 2 words:

	Thinking	Machines
Thinking	[q1 · k1 q1 · k2]	
Machines	[q2 · k1 q2 · k2]	

Result shape: (2 × 2)

In matrix form: $\text{scores} = Q \times K^T$

Steps 3 & 4 — Scale then Softmax

Scale: Divide by $\sqrt{64} = 8$

When vectors have 64 dimensions, dot products naturally grow very large. Large scores cause softmax to become extremely sharp — the model attends to only one word and learns nothing useful. Dividing by $\sqrt{d_k}$ keeps values in a moderate range.

What are Stable Gradients?

During training the model learns by calculating how much to adjust each weight — this signal is called a gradient.

If scores are too large → softmax becomes near 0 or near 1
→ gradients become vanishingly small or explosively large
→ both cases break training, the model stops learning.

Dividing by \sqrt{dk} keeps scores in a moderate range so gradients flow smoothly, and the model actually converges.

```
scaled_scores = Q x KT /  $\sqrt{64}$  = Q x KT / 8
```

Example:

```
Raw dot product:    q1 · k1 = 960  
After scaling:      960 / 8 = 120    ← much more stable
```

Softmax

Softmax converts the scaled scores into probabilities — all values become positive and sum to exactly 1.0.

```
Raw scaled scores:  [112, 96]  
After softmax:      [0.88, 0.12]
```

```
Interpretation:  
"Thinking" places 88% of its attention on itself  
and 12% of its attention on "Machines"
```

Steps 5 & 6 — Weighted Sum of Values (How Z is Computed)

Each value vector is multiplied by its softmax score, then all are summed. This produces one Z vector per word — a contextual blend of all value vectors weighted by relevance.

Computing Z1 — for word "Thinking"

```
Attention weights for "Thinking" row:  [0.88, 0.12]
```

```
z1 = (0.88 × v1) + (0.12 × v2)
```

```

v1 = value vector of "Thinking" → shape (64,)
v2 = value vector of "Machines" → shape (64,)

z1 = mostly v1 (Thinking) + a little v2 (Machines)
    = shape (64,)

```

Computing Z2 — for word "Machines"

Attention weights for "Machines" row: [0.15, 0.85]

```

z2 = (0.15 × v1) + (0.85 × v2)

    = a little v1 (Thinking) + mostly v2 (Machines)
    = shape (64,)

```

Key insight:

Each WORD gets its OWN row in the attention weight matrix.

That row is used as the weights for its own weighted sum of V.

Attention weight matrix (2 × 2):

	Thinking	Machines	
Thinking	[0.88	0.12]	← weights used for z1
Machines	[0.15	0.85]	← weights used for z2

$Z = \text{attention_weights} @ V \rightarrow \text{shape}(\text{seq_len} \times 64)$

Row 1 = z1, Row 2 = z2

Full Self-Attention — Matrix Form

In practice all six steps are combined into one matrix formula for speed and parallelism:

```

Z = softmax( Q × KT / √dk ) × V

```

Step by step:

$Q \times K^T$	→	(seq_len × seq_len)	all scores at once
$/ \sqrt{dk}$	→	scale for stable gradients	
$\text{softmax}(\dots)$	→	(seq_len × seq_len)	attention probabilities
$@ V$	→	(seq_len × 64)	final Z matrix

All words computed simultaneously — full parallelism!

4. Positional Encoding

Since all words are processed in parallel, the model has no built-in sense of word order. Positional encoding adds position information to each embedding before it enters the encoder stack.

Formula from the paper:

```
PE(pos, 2i)    = sin( pos / 10000^(2i / 512) )    ← even dimensions
PE(pos, 2i+1)  = cos( pos / 10000^(2i / 512) )    ← odd dimensions
```

```
pos = position of word in sequence (0, 1, 2, ...)
i   = dimension index
```

Final input to encoder:

```
X_final = Word Embedding + Positional Encoding
```

```
"Thinking" at position 0: embedding_1 + PE(0) → (512,)
"Machines" at position 1: embedding_2 + PE(1) → (512,)
```

Shape stays (seq_len, 512) — no dimension change.

Now the model knows "Thinking" comes before "Machines".

5. Multi-Head Attention — The Beast With Many Heads

Level 1 — WITHIN one encoder layer: All 8 attention heads run in PARALLEL (processing the same input simultaneously)

Instead of running self-attention once, the Transformer runs it 8 times in parallel. Each run (called a head) uses its own independent set of WQ , WK , WV weight matrices, so each head learns a different type of relationship.

```
head_1 might learn: "it" → "animal"    (coreference)
head_2 might learn: "cross" → "street"  (verb-object relation)
head_3 might learn: subject-verb dependencies
head_4 might learn: adjective-noun relationships
... and so on for all 8 heads
```

Combining the 8 Heads

Each head produces its own Z matrix: `shape (seq_len, 64)`

Step 1 — Concatenate all 8 heads:

```
[Z1, Z2, Z3, Z4, Z5, Z6, Z7, Z8]
→ shape (seq_len, 512)      (8 × 64 = 512)
```

```
Step 2 – Multiply by WO projection matrix:  
(seq_len, 512) × WO (512 × 512) → (seq_len, 512)
```

Output shape = Input shape ✓

The model can now stack multiple encoder blocks cleanly.

Level 2 — ACROSS encoder layers: Encoder 1 → Encoder 2 → Encoder 3 ... SEQUENTIAL (each encoder waits for the previous one to finish)

These two are **completely independent concepts** — one being parallel does not affect the other being sequential.

6. Add & Normalize — Why It Is Needed

After every sub-layer (self-attention and feed-forward), the Transformer applies a Residual Connection followed by Layer Normalization.

```
Formula:  Output = LayerNorm( X + SubLayer(X) )
```

Applied twice in each encoder block:

```
After Self-Attention:  LayerNorm( X + attention_output )
```

```
After Feed Forward:   LayerNorm( X + ff_output )
```

Residual Connection — the "Add" Part

The Transformer stacks 6 encoders deep.

In deep networks, gradients shrink as they travel backward through many layers → early layers stop learning (vanishing gradient).

Adding the original X directly creates a shortcut path for gradients to bypass multiple layers. Early layers keep learning.

It also preserves the original word embedding — even if the attention layer learned nothing useful, the original signal is still intact in the output.

Layer Normalization — the "Norm" Part

After $X + \text{attention_output}$, values can be on wildly different scales. LayerNorm brings them to mean = 0, std = 1.

This keeps inputs to the next layer consistent and stable,

leading to faster and more reliable training.

7. What Happens Inside the Feed-Forward Network

After attention, each word's vector passes through the same feed-forward network independently. Attention gathered information from other words; the feed-forward network processes and transforms that information.

```
FFN(x) = ReLU( x @ W1 + b1 ) @ W2 + b2
```

Dimensions:

Input: (seq_len, 512) → W1: (512, 2048)

Hidden: (seq_len, 2048) ← 4× expansion gives more capacity

ReLU: zeros out all negative values

Output: (seq_len, 512) → W2: (2048, 512) compresses back

Why expand to 2048 then compress back to 512?

Self-Attention answers: "Which words are relevant to each other?"

Feed-Forward answers: "Given this context, what should the final representation of this word be?"

The 2048 hidden dimension gives the model extra "thinking space"
— more capacity to learn complex non-linear transformations
before compressing back to 512.

Applied to each word INDEPENDENTLY using the SAME weights.

All word positions are processed in parallel.

8. How the Encoder Output Becomes K and V for the Decoder

After all 6 encoders finish processing, the top encoder produces a matrix of shape (seq_len, 512) — one rich contextual vector per input word. This is the encoder memory.

```
Encoder final output: shape (seq_len, 512)
```

One 512-dim vector per input word,

containing deep contextual understanding of that word.

To create K and V for the decoder's Cross-Attention:

```
K = encoder_output @ WK_cross    shape (seq_len, 64)
V = encoder_output @ WV_cross    shape (seq_len, 64)
```

WK_cross and WV_cross are NEW weight matrices learned specifically for cross-attention – different from the WK, WV used inside the encoder's own self-attention layers.

These K and V are computed ONCE after encoding is complete and then shared with ALL 6 decoder layers.

Inside each decoder's Cross-Attention:

Q → comes from the decoder itself (what we are generating)

K → comes from the encoder output (what the input was)

V → comes from the encoder output (what the input was)

The decoder queries the encoder memory:

Q asks: "What input word is relevant to what I am generating?"

K answers: "Here are all input words to compare against."

V delivers: "Here is the actual content of each input word."

9. Why Linear Before Softmax in the Decoder Output

The final decoder output has shape (seq_len, 512). Softmax needs one score per vocabulary word to output a probability distribution. The vocabulary can contain 10,000 to 50,000 words.

Problem:

```
Decoder output:    (seq_len, 512)
Vocabulary size:   10,000 words
Softmax requires:  one score per vocabulary word
512 ≠ 10,000 → dimensions do not match
```

Linear layer bridges this gap:

```
output @ W_final = logits
(512)   × (512 × 10000) = (10000,)
```

Each of the 10,000 output values is a raw score (logit) for one word in the vocabulary.

Logits can be any real number: -500 to +500

Then Softmax:

Converts 10,000 raw scores → 10,000 probabilities
All positive, all sum to 1.0

Then argmax:

Pick the index with highest probability
vocabulary[4821] → "thanks" ✓

If you skipped the Linear layer and applied softmax directly to the 512-dim decoder output, you would only produce probabilities over 512 buckets — not over 10,000 words. The Linear layer is the essential bridge from model dimension to vocabulary dimension.

10. How Output Vectors Are Converted Back to Words

After softmax produces a probability distribution over the vocabulary, the model maps the predicted index back to a word through the vocabulary lookup table.

Step 1 – argmax:

Pick the index with the highest probability.
e.g. probabilities[4821] = 0.87 ← highest
Predicted token index = 4821

Step 2 – vocabulary lookup:

vocabulary[4821] = "thanks"

Step 3 – detokenization (subword tokenizers):

Tokens: ["thank", "##s"] → merge → "thanks"
Tokens: ["un", "##happy"] → merge → "unhappy"
prefix means continuation piece (WordPiece convention)

Step 4 – repeat until <EOS> token:

The decoder generates one token at a time.
Each new token is fed back as input for the next step.
Generation stops when the <EOS> token is produced.

Full example:

Input: "merci"
Step 1: decoder → probabilities → "thanks" ✓
Step 2: feed "thanks" back → generates → <EOS>
Output: "thanks"

The embedding matrix ($\text{vocab_size} \times 512$) used at the START to convert words into vectors is often reused (transposed) at the END to convert vectors back into word probabilities.

This technique is called weight tying — it saves parameters and works well in practice.

11. Why Modern LLMs Are Decoder-Only

Original Transformer vs Decoder-Only

Original Transformer Decoder — 3 sub-layers:

1. Masked Self-Attention
2. Cross-Attention ← requires encoder output (K, V)
3. Feed Forward

LLM Decoder-Only — 2 sub-layers:

1. Masked Self-Attention ← only this
 2. Feed Forward
- Cross-Attention removed entirely.

How It Works Without an Encoder

In decoder-only models, the input prompt and output tokens are treated as one single continuous sequence. There is no separation between source and target.

Original Transformer thinking:

Source: "Je suis etudiant" → Encoder
Target: "I am a student" → Decoder
Cross-Attention bridges the two.

Decoder-Only thinking:

"Je suis etudiant I am a student"
↑ this entire sequence is ONE stream of tokens.
No encoder. No cross-attention needed.

Causal mask in self-attention does all the work:

Token at position N can attend to positions 0..N-1 only.
When generating "Paris" in response to
"What is the capital of France?"
→ self-attention looks back at ALL previous tokens
→ "France" receives high attention weight

→ model generates "Paris"

This is exactly what cross-attention used to do –
but now self-attention handles it naturally because
the prompt and response share one sequence.

Four Reasons Decoder-Only Won

1. Generation is naturally a decoder task. LLMs predict the next token given all previous tokens. That is exactly what the decoder does. No second sequence is needed, so no encoder is needed.

2. Simpler and scales better. One component instead of two. Fewer design decisions. No need to balance encoder vs decoder size. GPT-4 and LLaMA scale to hundreds of layers and trillions of parameters cleanly.

3. Causal attention is the right constraint. The decoder only looks at past tokens — exactly how language works. You read and write left to right. Encoder bidirectional attention would let the model cheat by seeing future tokens during generation.

4. Emergent abilities from scale. Researchers discovered that scaling decoder-only models unlocked surprising capabilities: few-shot learning, reasoning, instruction following, and code generation — all without any encoder.

Architecture Comparison

Architecture	Models	Attention	Best Used For
Encoder-Only	BERT, RoBERTa	Bidirectional	Classification, Search, Understanding
Decoder-Only	GPT, LLaMA, Claude	Causal (left→right)	Text Generation, Chat, Reasoning
Encoder-Decoder	T5, BART	Both	Translation, Summarization

The shift to decoder-only happened because the goal changed — from translating between two fixed sequences to freely generating language at massive scale.

The prompt itself becomes the memory.

Self-attention over the full context IS the encoder.

No separate component needed.