Transformers

Transformers were introduced to overcome the limitations of older sequence models like CNNs and RNNs, especially in handling:

* Long-term dependencies – relationships between distant points in a sequence.
* Parallelization – ability to train quickly by processing all tokens simultaneously.
* Flexibility – capturing both local and global context.

## Limitations

CNNs:

* Excellent at detecting local patterns (e.g, edges in images, short-term trends in sequences).
* Limitation: Fixed receptive field; cannot naturally capture long-range dependencies or temporal relationships beyond the filter size.
* Workarounds like stacking more layers or using dilated convolutions exist, but they are less flexible than sequence-aware models.

RNNs:

* Process sequences one step at a time, updating the hidden state at each step.
* Challenges:
  + Hard to parallelize: Each step depends on the previous, making training slower.
  + Long-term dependencies: Vanilla RNNs suffer from vanishing or exploding gradients, making it difficult to learn relationships over long sequences.
* Variants like LSTM or GRU mitigate some of these issues via gating mechanisms but still cannot match the full parallelism of Transformers.

Transformers:

* Use attention mechanisms to directly compare every input token with every other token.
* Advantages:
  + No recurrence or convolution fully parallelizable over sequence length.
  + Can model long-range dependencies efficiently, capturing global context.
  + Scale well to large datasets and long sequences.
  + Flexible architecture for diverse tasks (text, images, time series).
* Trade-offs: Computational cost grows quadratically with sequence length (though mitigations exist).

## Core Idea: Attention

The core idea is that each token **attends** to others to build a richer representation.

* Input is a sequence of tokens.
* Each token is projected into three vectors:
  + Query (q) – “What am I looking for?”
  + Key (k) – “What information do I hold?”
  + Value (v) – “What content can I pass on?”

Attention computes similarity between queries and keys to weight values: each token updates its representation based on all other tokens in the sequence.

## Attention Formula

* : queries, : keys, : values
* : scaling factor for numerical stability.
* Softmax ensures weights sum to 1.
* Output: each token’s new representation incorporates context from all tokens.
* : similarity score between tokens.
* Softmax: converts scores into probabilities (weights sum to 1).
* Multiplying by : combines token content weighted by these probabilities.

## Numerical example

### Step 1: Inputs

Suppose we have 3 tokens, each represented by a 2D vector:

We define **trainable projection matrices** (here fixed for illustration):

## # Step 2: Compute Q, K, V

Queries:

Keys:

Values:

### Step 3: Compute Attention for Token 1

Query for token 1:

Softmax normalization:

Output vector for token 1:

$$

$$

### Final Attention Output

Attention weights () & updated token representations ():

### Interpreting the Attention Weight Matrix

is a 3×3 matrix where:

* Rows = query token (the “focusing” token).
* Columns = key token (the “source” token being attended to).

Example:

* First row: [0.140, 0.284, 0.576]
* Token 1 is looking at itself, token 2, and token 3.
* It places 14% of its focus on itself, 28% on token 2, and 58% on token 3.

Key Properties:

1. Each row sums to 1 the softmax normalization ensures attention is a weighted average:
2. These weights are **learned relationships** between tokens:
   * Higher weight = stronger relationship or influence
   * E.g. token 3 has a high weight of 0.620 in its own row, meaning it primarily relies on itself but also incorporates information from others

## Interpreting — Updated Token Representations

The matrix contains the **final output vectors** from the attention mechanism.

* Each row corresponds to one token’s new embedding
* Computed by combining the value vectors () using the attention weights from that token’s row:

Example:

For Token 1:

This vector now encodes **Token 1’s original meaning + contextual information** from Tokens 2 and 3, weighted by their relevance

### Why is More Powerful than

1. The original input tokens are **isolated**, containing no information about other tokens
2. After self-attention:
   * Each is **context-aware**, enriched with information from the **entire sequence**
   * Example use cases:
     + In language models: could represent a word’s meaning **in context**
     + In time series: could combine signals from past, present, and future steps

* answers: *“Which tokens matter most to me?”*
  + It’s like a **map of relationships** between tokens
* answers: *“Given those relationships, what is my updated understanding?”*
  + It’s a **contextual embedding** ready for downstream tasks like prediction or classification

This interpretation is crucial because in deep transformers:

* **Multiple attention heads** produce multiple and matrices in parallel
* Later layers refine and combine them
* Resulting in extremely rich, hierarchical representations of the input data

## What happens to the

Each goes through:

1. Feed-forward network: nonlinear transformation to refine features.
2. Residual connection & Layer normalization helps stabilize deep training.
3. Decoder output layer that projects back into vocabulary space (for language tasks):

Result: a **probability distribution over next possible tokens**

## Training Target and Loss

* Goal: Predict the **next word** given a sequence.
* Input: tokens
* Target: the next token
* Prediction: from the final

Loss function: Cross-entropy between and the true token.

Gradients flow backward through:

1. Output projection,
2. Transformer layers,
3. Embedding layer.

## Multi-Head Attention

Instead of a single attention computation:

* **Multiple heads** learn different aspects of relationships - run in parallel
* Each head has its own .

### Analogy:

* CNN: each filter detects a different feature (edges, shapes).
* Transformer: each head focuses on different token relationships.

## Full Transformer Block

Architecture of one encoder block:

1. Input tokens + positional encoding (to add order info).
2. Multi-head attention combined outputs.
3. Residual connection + layer norm.
4. Feed-forward network.
5. Residual connection + layer norm.

Stacking many blocks = deep Transformer.

## Key Advantages

1. Parallelism: Train on all tokens at once → huge speedup.
2. Long-range dependencies: Global context captured easily.
3. Specialization: Different attention heads focus on different relationships.
4. Scalability: Foundation of modern LLMs like GPT, BERT.

## Summary

Transformers revolutionized sequence modeling by:

* Replacing recurrence and convolution with **attention**.
* Allowing **efficient parallel training**.
* Enabling large-scale models that power modern NLP and beyond.

The core is **self-attention**, which lets each token “see” every other token and decide what matters most.