Transformer

🧠 Core Idea of the Transformer

Before Transformers, models like **RNNs** and **LSTMs** were the go-to for sequence tasks (translation, summarization, etc.).

However, they processed sequences **sequentially**, which caused two big problems:

1. **No parallelization** — training was slow.
2. **Difficulty capturing long-range dependencies** — information faded over long sequences.

👉 **Transformers** solved both using one radical idea:

> 🔑 Use attention (especially self-attention) instead of recurrence to model dependencies — and process all tokens in parallel.

This allows:

- Parallel training (faster computation)
- Global context awareness (long-range relationships)
- Better scalability with depth and data

⚙️ Transformer Architecture Overview

The Transformer is composed of two main parts:

Encoder → Decoder

Each is made up of **repeated blocks (layers)**.

🔷 Encoder–Decoder Overview

Component	Function
Encoder	Reads the input sentence and produces contextualized representations.
Decoder	Uses those representations to generate the output sequence (e.g., translated sentence).

For example:

> Input: “I love dogs” (English)
>
> Output: “J’aime les chiens” (French)

🧩 ENCODER ARCHITECTURE

Each encoder block has **two main sublayers**:

1️⃣ **Multi-Head Self-Attention**

2️⃣ **Feed-Forward Network (FFN)**

Plus two important add-ons:

- **Residual connection** around each sublayer
- **Layer normalization**

Step-by-Step in Encoder:

1. Input Embedding

Each word/token is first mapped to a dense vector:

$x_i \to e_i \in \mathbb{R}^{d_{model}}$

2. Positional Encoding

Since Transformers have **no recurrence or convolution**, they don’t know word order.

So we add **positional encodings** (sine & cosine patterns) to embeddings.

$z_i = e_i + PE_i$

This tells the model the position of each token.

3. Multi-Head Self-Attention

Every token **attends to all tokens**, learning contextual meaning.

Output = weighted sum of all token representations (as explained in self-attention).

4. Add & Norm

Output of attention is added back to the input (residual connection) → normalized:

$$\mathrm{LayerNorm}(x + \mathrm{Attention}(x))$$

5. Feed Forward Network (FFN)

A simple MLP applied independently to each position:

$$\mathrm{FFN}(x) = \mathrm{ReLU}(xW_1 + b_1)W_2 + b_2$$

6. Add & Norm again

Another residual connection + normalization.

✅ Final output = encoded representations of each token with full context awareness.

🧩 DECODER ARCHITECTURE

Each decoder block has **three sublayers**:

1️⃣ **Masked Multi-Head Self-Attention**

2️⃣ **Encoder–Decoder Attention**

3️⃣ **Feed-Forward Network**

Again with residuals and normalization.

Step-by-Step in Decoder:

1. Masked Multi-Head Self-Attention

The decoder can only attend to **previous tokens** (to prevent "cheating" during generation).

Masking ensures attention weights for future positions = 0.

2. Encoder–Decoder Attention

Now the decoder attends to **the encoder's output** — this is how it aligns with the input sentence.

(e.g., while generating "chiens," it attends to "dogs.")

3. Feed Forward + Add & Norm

Same as encoder.

4. Linear + Softmax

Finally, the decoder outputs probabilities for the next token.

$P(y_t|y_{<t}, X)$

🧱 The Complete Transformer Flow

```
Input sentence → [Encoder stack] → Context vectors
Decoder (auto-regressive) → uses context + previous outputs → generates tar
get sequence
```

⚡ Key Concepts Recap

Concept	Role
Self-Attention	Lets each word see all others for context
Multi-Head Attention	Captures multiple relationships in parallel
Positional Encoding	Injects sequence order information
Feed Forward Network	Adds non-linearity and depth
Residual Connections + LayerNorm	Stabilize and speed up training
Masked Attention (Decoder)	Ensures autoregressive (left-to-right) generation

Concept	Role
Encoder–Decoder Attention	Connects source and target sequences

---

🧩 Visual Summary

```
        +----------------------------+
Input  →→→ |  Encoder (N layers)      |
        | [Self-Attn + FFN]         |
        +----------------------------+
                  ↓
        +----------------------------+
Output ←←← |  Decoder (N layers)      |
        | [Masked Self-Attn +       |
        |  Enc-Dec Attn + FFN]      |
        +----------------------------+
```

🎯 Intuition Summary

Step	Analogy
Encoder	Reads and understands the full sentence (like a human listening carefully).
Decoder	Writes the translation step-by-step, looking back at both the source (encoder) and what it has already written.
Attention	The mechanism that decides *what to focus on* in each step.

🧠 The Goal of Transformer Training

Transformers are trained to **predict the next token** in a sequence —

That's how they learn language understanding and translation.

🎯 Objective (Training Goal)

Given:

- An **input sequence** ($X = [x_1, x_2, ..., x_n]$)
- A **target sequence** ($Y = [y_1, y_2, ..., y_m]$)

We train the Transformer to **predict each token (** y_t **)**

Given the **previous tokens** and the **input**:

$P(y_t | y_{<t}, X)$

The model is **auto-regressive** — it predicts tokens one by one.

🧩 Step-by-Step: Transformer Training Pipeline

1️⃣ Input Processing

1. **Input tokens** → embedded into vectors.
2. **Positional encoding** was added to keep word order.
3. Passed into the **encoder** stack (N layers).

Encoder outputs **context vectors** that summarize the input sentence.

2️⃣ Decoder Operation During Training

During training, we already know the **target sentence**.

So we feed the **ground-truth tokens** into the decoder — this is called **teacher forcing**.

Example:

If the target sentence is

> "I love pizza 🍕"

Then during training:

- Input to decoder = "I love"
- Target output = "pizza"

Masking ensures that the model only "sees" previous tokens ("I", "love") when predicting the next one.

3️⃣ Output Prediction

The **decoder's last layer** outputs a vector of size equal to the model dimension (say 512).

This goes through a **linear layer + softmax** to produce a probability distribution over the vocabulary.

$$P(y_t|y_{<t}, X) = \text{softmax}(W_{out}h_t)$$

where (h_t) is the decoder's output at time (t).

4️⃣ Loss Function — Cross-Entropy Loss

We use **cross-entropy loss**, which compares predicted probabilities to the true token.

$$\mathcal{L} = -\sum_{t=1}^{m} \log P(y_t^{true}|y_{<t}, X)$$

👉 This penalizes the model more when it assigns a low probability to the correct word.

🔁 5️⃣ Backpropagation Through the Transformer

This is where training happens!

🔷 Step 1: Compute Loss Gradient

The loss gradient flows **backward** from the softmax output to the final decoder layer.

🔷 Step 2: Gradient Through Decoder Layers

Each decoder layer has:

1. Masked Self-Attention
2. Encoder–Decoder Attention
3. Feed-Forward Network (FFN)

Gradients pass backward through each of these sub-layers.

- The gradient updates **attention weights** (W_Q, W_K, W_V)
- Updates **FFN weights**
- And adjusts **layer norm parameters**

Residual connections ensure **smooth gradient flow —** preventing vanishing gradients even in deep stacks.

🔷 Step 3: Gradient Through Encoder Layers

The gradient also flows backward into the **encoder stack**, because decoder attention depends on encoder outputs.

Each encoder layer gets updates for:

- Self-attention weights (captures better token dependencies)
- FFN weights (refines non-linear transformations)
- Layer norm and residuals

Thus, the encoder gradually learns to represent input sentences more effectively.

🔷 Step 4: Gradient to Embeddings and Positional Encoding

At the end of backpropagation:

- Word embedding matrix (W_E) gets updated → better representations of words.
- Positional encoding (if learned, not fixed sinusoidal) can also be adjusted.

🔧 6️⃣ Optimization

Transformers typically use the **Adam optimizer** (or AdamW) with a special **learning rate schedule**:

$$\mathrm{lr} = d_{model}^{-0.5} \cdot \min(step^{-0.5}, step \cdot warmupsteps^{-1.5})$$

This means:

- LR increases linearly during early "warm-up" steps
- Then decays proportionally to ($step^{-0.5}$)

This helps stabilize training in the beginning and improve convergence later.

📊 7️⃣ Training Objective in Practice

The model's training objective over the full dataset:

$$\text{Minimize } \mathcal{L}total = -\sum(X,Y) \in D \sum_{t=1}^{|Y|} \log P(y_t|y_{<t}, X; \theta)$$

where (θ) includes **all learnable parameters**:

- Embedding weights
- Attention weights ((W_Q, W_K, W_V, W_O))
- FFN weights ((W_1, W_2))
- Layer norms
- Output projection (W_{out})

⚡ How Each Layer Learns (Conceptually)

Layer	What It Learns
Embedding	Word meaning (distributed representation)
Positional Encoding	Word order information
Encoder Self-Attention	Relationships among input tokens
Decoder Self-Attention	Relationships among generated tokens
Encoder–Decoder Attention	Alignments between input and output (like translation pairs)
Feed-Forward Networks	Complex nonlinear transformations of contextual info
Output Softmax Layer	Vocabulary-level probability mapping

All layers are trained **jointly** — the loss from the final output is backpropagated to all components.

🧩 Training Example (Simplified)

Suppose your training pair is:

> Input: "I love dogs"

> Output: “J’aime les chiens”

At step `t=3`:

- Model predicts token = “chiens”
- True token = “chiens”
- Cross-entropy loss is low → minimal update

  If model predicted “chat” (cat):

- Loss is high → gradient adjusts encoder-decoder attention so “dogs” better maps to “chiens”

Over many samples, these gradients train:

- Encoders to encode semantic meaning
- Decoders to decode contextually correct translations

🎯 Intuitive Summary

Step	What Happens
Forward pass	Model predicts next token using all attention layers
Compute loss	Compare predicted vs. true token (cross-entropy)
Backward pass	Gradients flow through decoder → encoder → embeddings
Update weights	Optimizer adjusts parameters
Repeat	Until the model converges and can generate accurate sequences

---

🧠 Big Picture: The Transformer Has Two Main Parts

[Encoder Stack] → [Decoder Stack]

But depending on the **task**, we can use:

- only the **Encoder** part,

- only the **Decoder** part,
- or both together (**Encoder–Decoder**).

1️⃣ Encoder-Only Models

🧩 Architecture

Use **only the encoder stack** from the Transformer.

Each layer contains:

- Multi-head **self-attention**
- Feed-forward network
- Add & Norm connections

🔍 How It Works

- The encoder takes an input sequence (like a sentence or document).
- Each token attends to *all other tokens* (bidirectionally).
- The model learns **contextual representations** of the entire input.

$$h_i = f(x_1, x_2, ..., x_n)$$

So each token's vector (h_i) knows the meaning of all words around it.

🧠 Use Cases

Encoder-only models are used for **understanding tasks** (not generation).

Examples:

- Sentence classification (e.g., sentiment analysis)
- Named Entity Recognition (NER)
- Question answering (extractive)
- Similarity and embedding generation

📘 Examples of Encoder-Only Models

Model	Description
BERT	"Bidirectional Encoder Representations from Transformers" — learns deep bidirectional context.
RoBERTa	Robustly optimized version of BERT.
DistilBERT	Smaller, faster version of BERT.

⚡ Key Property

Bidirectional attention: each token can see all other tokens on both sides — left and right.

This gives rich contextual understanding but makes **text generation impossible** (since the model "sees the future").

2️⃣ Decoder-Only Models

🧩 Architecture

Use **only the decoder stack**, but **without encoder–decoder attention**.

Each layer includes:

- **Masked self-attention**
- **Feed-forward network**
- Add & Norm connections

🔍 How It Works

- The decoder predicts the next token one step at a time.
- Masked attention ensures each token only attends to **previous tokens** (not future ones).

$$P(y_t | y_{<t})$$

This creates a **causal**, left-to-right generation process.

🧠 Use Cases

Decoder-only models are used for **generation tasks** such as:

- Text completion
- Dialogue and chatbots
- Story generation
- Code generation
- Autoregressive modeling

🤖 Examples of Decoder-Only Models

Model	Description
GPT (1, 2, 3, 4, 5)	"Generative Pre-trained Transformer" — trained to predict next word (language modeling).
LLaMA, **Falcon**, **Mistral**	Open-source GPT-style models.
CodeGen, **StarCoder**	Specialized for code generation.

⚡ Key Property

Unidirectional attention: each token can only see tokens **to its left**, preserving causality.

This makes it perfect for **autoregressive generation**.

3️⃣ Encoder–Decoder (Seq2Seq) Models

🧩 Architecture

Uses **both encoder and decoder stacks** — the *full* Transformer.

Encoder → produces context → Decoder → generates output

🔍 How It Works

- The **encoder** processes the input sequence → context representations.
- The **decoder** uses:
    - Masked self-attention (to generate outputs step-by-step)

- Encoder–decoder attention (to focus on relevant input tokens)

- This allows **conditional generation** (output depends on input).

$$P(y_t|y_{<t}, X)$$

🧠 Use Cases

Used for **sequence-to-sequence tasks**, where input and output are different sequences:

- Machine translation
- Summarization
- Text-to-text transformation
- Question answering (generative)
- Paraphrasing

🌟 Examples of Encoder–Decoder Models

Model	Description
T5	"Text-To-Text Transfer Transformer" — converts all NLP tasks into a text-to-text format.
BART	Combines BERT-style encoder + GPT-style decoder for text generation and denoising.
MarianMT	Specialized for machine translation.
mT5, **Flan-T5**	Multilingual or instruction-tuned versions.

⚡ Key Property

Bidirectional in the encoder, **unidirectional in the decoder**.

→ Model *understands* input deeply, and *generates* conditioned on it.

🔍 Comparison Summary Table

Feature	Encoder-Only	Decoder-Only	Encoder–Decoder
Attention Direction	Bidirectional	Unidirectional (causal)	Encoder: bidirectionalDecoder: unidirectional
Main Purpose	Understanding	Generation	Translation / Seq2Seq
Inputs	Single text	Previous tokens	Input + Generated output
Examples	BERT, RoBERTa	GPT, LLaMA	T5, BART
Use Cases	Classification, QA (extractive)	Text completion, chatbots	Summarization, translation
Training Objective	Masked LM (fill missing words)	Next-token prediction	Conditional generation
Context Flow	All tokens see each other	Each token sees past only	Decoder attends to encoder outputs

🧩 Visual Summary

```
1️⃣ Encoder-Only
Input → [Encoder Stack] → Output Representation
    ↳ Understanding task (BERT)

2️⃣ Decoder-Only
Input → [Masked Decoder Stack] → Generated Output
    ↳ Generation task (GPT)

3️⃣ Encoder–Decoder
Input → [Encoder] → Context → [Decoder] → Output
    ↳ Translation / Summarization (T5, BART)
```

🎯 Intuition Summary

Type	Analogy
Encoder-Only	Like reading and *understanding* a sentence deeply.

Type	Analogy
Decoder-Only	Like *writing* a story word-by-word.
Encoder–Decoder	Like *translating* — read a source sentence, then generate its version in another language.

🧠 What Is Pretraining in Transformers?

Before a Transformer can perform tasks like translation, summarization, or sentiment analysis, it needs to *understand* language.

To gain this understanding, it undergoes a **pretraining phase** — learning from massive amounts of unlabeled text using **self-supervised objectives** (like predicting missing words).

These objectives are called **Pretraining Strategies**.

⚙️ Main Pretraining Strategies in Transformers

Below are the most common and influential pretraining strategies used in Transformer models:

1. Masked Language Modeling (MLM) — Used by BERT

Idea:

- Randomly mask (hide) some words in a sentence.
- Ask the model to predict those masked words from the surrounding context.

Example:

> Input: "The cat sat on the [MASK]."
>
> Target: "mat"

Goal:

Learn *bidirectional context* — i.e., understand words based on *both left and right* neighbors.

Used in:

✅ BERT, RoBERTa, ALBERT

---

2. Next Sentence Prediction (NSP)

Idea:

- Alongside MLM, the model also learns whether two sentences logically follow each other.

Example:

> Sentence A: "The cat sat on the mat."
>
> Sentence B: "It started to purr." ✅ (Next sentence)
>
> Sentence C: "Apples grow on trees." ❌ (Not the next sentence)

Goal:

Learn *relationships between sentences* — helpful for question answering and natural language inference.

Used in:

✅ Original BERT

Limitation:

Later research (e.g., RoBERTa) showed NSP doesn't help much and can be removed.

---

3. Causal Language Modeling (CLM) — Used by GPT

Idea:

- Predict the *next word* given all previous words.
- Only uses *left-to-right* context (unidirectional).

Example:

> Input: "The cat sat on the"
>
> Target: "mat"

Goal:

Learn *generative* modeling — essential for text generation and completion.

Used in:

✅ GPT, GPT-2, GPT-3, GPT-4, LLaMA

---

4. Permutation Language Modeling (PLM) — Used by XLNet

Idea:

- Instead of masking words, predict tokens in a *random permutation order*.
- This combines the benefits of MLM and CLM (bidirectional context + generative ability).

Used in:

✅ XLNet

Goal:

Capture bidirectional context *without using masks*.

---

5. Denoising Autoencoder (DAE) — Used by BART / T5

Idea:

- Corrupt the input sentence (by masking, deleting, shuffling words).
- Ask the model to *reconstruct* the original sentence.

Example:

> Corrupted: "The [MASK] on mat cat the."
>
> Target: "The cat sat on the mat."

Goal:

Learn to recover meaning from noisy input — great for summarization, translation, etc.

Used in:

✅ BART, T5

🧩 Summary Table

Strategy	Directionality	Objective	Example Models	Strength
MLM	Bidirectional	Predict masked words	BERT	Strong contextual understanding
NSP	Bidirectional	Predict next sentence	BERT	Sentence-level reasoning
CLM	Unidirectional	Predict next token	GPT series	Natural text generation
PLM	Bidirectional (permuted)	Predict token order	XLNet	Combines BERT & GPT benefits
DAE	Bidirectional	Reconstruct corrupted input	BART, T5	Robust understanding & generation

---

🧩 Effect on Training and Downstream Tasks

- These strategies help the model **learn general language representations** from unlabeled data.
- During **fine-tuning**, the pretrained weights are adjusted slightly for specific tasks like:
  - Classification (sentiment)
  - QA
  - Summarization
  - Translation
- This approach drastically reduces labeled data requirements and training time.

🧠 What is Masked Language Modeling (MLM)?

Definition:

Masked Language Modeling is a **self-supervised learning objective** where a model learns to **predict missing (masked) words** in a sentence based on their surrounding context.

🎯 Goal

Instead of predicting the next word (like GPT), MLM teaches the model to **understand context in both directions** — left and right.

That's why it's called a **bidirectional training objective**.

📖 Example

Original sentence:

> "The cat sat on the mat."

We randomly **mask** one or more tokens (e.g., 15% of them):

> "The cat sat on the [MASK]."

The model must predict the missing word:

> "mat"

So it learns to understand *how words relate to each other in both directions.*

⚙️ How It Works (Step-by-Step)

1️⃣ Input Preparation

- Take a sentence and **randomly mask** 15% of the tokens.
- But not all of them are replaced with `[MASK]`:
    - 80% → replaced with `[MASK]`
    - 10% → replaced with a random word
    - 10% → left unchanged

    This helps prevent the model from overfitting to the `[MASK]` token.

Example:

Original	Masked Input	Target
"I love NLP models."	"I love [MASK] models."	"NLP"

2️⃣ Encoder Processing

The masked sentence is passed through the **encoder** (e.g., in BERT):

- Every token attends to *all* other tokens (including left and right context).
- The encoder produces contextual embeddings for each token.

3️⃣ Prediction Layer

For each masked position, the model predicts the **original word** using a softmax classifier over the vocabulary:

$$P(w_i|context) = \mathrm{softmax}(Wh_i + b)$$

where:

- (h_i) = hidden representation of the masked position
- (W) = output projection matrix
- (b) = bias vector

4️⃣ Loss Function — Cross-Entropy Loss

The model is trained to minimize the negative log-likelihood of the correct token:

$$\mathcal{L} = -\sum_{i \in M} \log P(w_i^{true}|context)$$

where (M) = set of masked positions.

Only masked tokens contribute to the loss.

🧩 Intuitive Understanding

Property	Description
Bidirectional context	Model looks at both left and right sides of the masked word.
Self-supervised	Labels are created from the data itself (no manual annotation needed).
Contextual embeddings	Learns meaning of words *in context* (e.g., "bank" in "river bank" vs "money bank").

🧱 Example in Detail

Sentence:

> "The dog chased the [MASK]."

The model sees:

- Left context: "The dog chased the"
- Right context: (none in this case)

Predicts:

> "ball" (high probability), "cat", "stick" (lower probability)

⚡ Why MLM Works So Well

- It teaches the model to **understand relationships among all words** in a sentence.
- It's **bidirectional** — unlike autoregressive models (like GPT) which only look left-to-right.
- The representations learned can be reused for **many downstream tasks** (transfer learning).

📘 Example: BERT's MLM Training Objective

BERT combines two tasks during pretraining:

Task	Description
Masked Language Modeling (MLM)	Predict masked tokens using bidirectional context.
Next Sentence Prediction (NSP)	Predict if one sentence follows another.

During training, BERT randomly masks 15% of input tokens and learns to predict them.

After pretraining, it's fine-tuned for specific NLP tasks (classification, QA, etc.).

🧩 Comparison to Next Token Prediction (like GPT)

Feature	Masked LM (BERT)	Next-Token LM (GPT)
Context	Bidirectional	Left-to-right (causal)
Masking	Predict missing words	Predict next word
Use Case	Understanding	Generation
Example Task	Fill-in-the-blank	Text continuation

🎯 Summary

Aspect	Masked Language Modeling
What it does	Randomly masks words and trains model to predict them
Why it works	Forces model to learn bidirectional contextual understanding
Loss function	Cross-entropy over masked tokens
Used in	BERT, RoBERTa, ALBERT
Result	Powerful contextual embeddings for downstream NLP tasks

---