Sequence-to-Sequence With Attention

🧠 1. What is Sequence-to-Sequence With Attention (Seq2Seq)?

A **Sequence-to-Sequence** model is the classic neural approach used for tasks like:

- **Machine Translation** (English → French),
- **Summarization**,
- **Question answering**, etc.

It consists of two main parts:

Component	Role
Encoder	Reads the input sequence (e.g., English sentence) and converts it into a **context vector** — a numerical summary of meaning.
Decoder	Takes that context vector and generates the output sequence (e.g., the French translation).

🧩 Example

Input: "I love deep learning."

Output: "J'aime l'apprentissage profond."

👉 The encoder processes each English word into a hidden state, and after reading the whole sentence, produces a **fixed-length vector** (the *context vector*).

👉 The decoder then uses this single vector to generate each word in the French language.

⚠️ 2. The Bottleneck Problem

The **bottleneck problem** arises because the **entire input sequence** must be compressed into **one fixed-length vector** before decoding begins.

Let's visualize it:

```
Encoder:     I → love → deep → learning → [context vector]
                                   ↓
Decoder:                     J' → aime → l' → apprentissage → profond
```

That **[context vector]** is the *only bridge* between the input and output.

🔍 Why It's a Problem

1️⃣ Fixed-Length Compression

- Regardless of whether the input is 5 words or 50 words long, the encoder must compress all meaning into a **single fixed-size vector** (e.g., 512 dimensions).
- This creates an **information bottleneck** — the longer or more complex the input, the more meaning gets lost.

> 💬 Think of it like trying to summarize a whole book into one sentence — a lot of context is inevitably lost.

2️⃣ Vanishing Context

- During decoding, the model depends entirely on this one vector to generate all outputs.
- As decoding progresses, the model gradually "forgets" earlier parts of the input since it has no direct access to the encoder's hidden states.

3️⃣ Performance Drops with Long Sentences

- Seq2Seq models with a single context vector work *okay* for short sentences.

- But for long sentences or paragraphs, translation quality drops sharply because:
  - Important words or phrases are lost in the compression.
  - The model can't recall fine-grained relationships between distant tokens.

🧮 3. Mathematical Intuition

In basic Seq2Seq (without attention):

- Encoder produces hidden states:

  $h_t = f(x_t, h_{t-1})$

- The **final hidden state** h_T (after the last word) becomes the **context vector**:

  $c = h_T$

- Decoder then uses c at every time step to predict output tokens:

  $s_t = g(y_{t-1}, s_{t-1}, c)$

🧩 The issue: c is fixed → no dynamic access to individual encoder states → **information loss**.

🚧 4. Visual Intuition

Without Attention (Bottleneck Present)

```
Input: [ I ] → [ love ] → [ deep ] → [ learning ]
                    ↓
              [ context vector ]
                    ↓
Output: [ J' ] → [ aime ] → [ l' ] → [ apprentissage ]
```

Only one arrow (the context vector) connects input and output — creating the bottleneck.

💡 5. The Solution — Attention Mechanism

🎯 1. The Core Idea of Attention

> Attention is a mechanism that lets a neural network focus on the most relevant parts of the input when producing each part of the output.

Instead of treating all input words equally, the model **learns to assign different weights (importance)** to different words **depending on the current output step**.

💡 In Simple Terms

Imagine you're translating this sentence:

> "The cat sat on the mat."

When generating the French word for "cat" (`chat`),

You mainly need to focus on **"cat"** in the input — not on "mat" or "on."

So the model "attends" more to **"cat"** than to other words at that decoding step.

That selective focusing is **Attention**.

🧩 2. Why We Needed Attention

In the earlier **Seq2Seq models**, we saw the **bottleneck problem**:

The entire input sentence had to be squashed into one fixed vector before decoding.

That meant the decoder had **no direct access** to the encoder's hidden states — leading to **information loss**, especially in long sentences.

> ⚠️ The model had to "remember everything" in one memory slot — very inefficient.

Attention removes this bottleneck by giving the decoder **direct, weighted access** to *all* encoder outputs.

🧮 3. How Attention Works (Conceptually)

Let's say:

- The encoder produces hidden states $h_1, h_2, \dots, h_T$ for input words.
- Decoder has its own hidden state at time step t, denoted s_t.

We want to generate the next output word y_t.

The decoder does the following:

1️⃣ **Compare** the current decoder state s_t with each encoder state h_i

→ this gives a **score** (how relevant each input word is to this step).

$e_{t,i} = \text{score}(s_t, h_i)$

2️⃣ **Normalize** these scores into probabilities (via softmax):

$$\alpha_{t,i} = \frac{\exp(e_{t,i})}{\sum_j \exp(e_{t,j})}$$

→ called **attention weights**.

3️⃣ **Compute Context Vector:**

$c_t = \sum_i \alpha_{t,i} h_i$

→ this is a *weighted average* of all encoder states, where weights reflect relevance.

4️⃣ **Use Context to Generate Output:**

The decoder combines c_t and s_t to predict the next word y_t.

✨ Visual Intuition

```
Input: [ I ] → [ love ] → [ deep ] → [ learning ]
            ↖    ↘     ↑      ↗
Output:  [ J' ] → [ aime ] → [ l' ] → [ apprentissage ]
```

The decoder "looks at" — i.e., **attends** — to specific input words (with varying strengths) at each decoding step.

🧮 4. Types of Attention (Basic Overview)

Type	Description
Additive Attention (Bahdanau, 2015)	Uses a small neural network to compute the score between encoder and decoder states.
Dot-Product (Multiplicative) Attention (Luong, 2015)	Computes similarity via dot product between states — faster and simpler.
Scaled Dot-Product Attention	Used in Transformers; divides by √d to stabilize gradients.

🧠 5. Intuition: Attention = Soft Search

You can think of attention as a **soft lookup table** or **soft search mechanism**:

- The model doesn't "pick" one word — it computes a **weighted mix** of all words.
- It learns **where to look** and **how much to look** at each input part automatically.

> It's like your eyes moving over words as you read a sentence — you don't process all words equally at all times.

⚙️ 6. Why Attention Was Revolutionary

Problem Before	Attention Fix
Fixed-length bottleneck	Accesses all encoder states dynamically
Lost context in long sentences	Focuses selectively on relevant words
Poor interpretability	Attention weights show *what the model looked at*
Slow learning	Makes gradient flow smoother via direct connections

---

🧾 6. Summary Table

Problem Aspect	Seq2Seq (No Attention)	With Attention
Input representation	Fixed-length vector	Variable-length context

Problem Aspect	Seq2Seq (No Attention)	With Attention
Information retention	Limited (compression)	Preserved via dynamic attention
Long sentence handling	Poor	Good
Interpretability	Low	High (attention weights show focus)

🚀 7. Key Takeaway

> The bottleneck problem in sequence-to-sequence models arises from compressing an entire input sequence into one fixed-size vector.
>
> This limits the model's ability to handle long or complex inputs.
>
> **Attention mechanisms** (and later Transformers) overcome this by allowing the model to directly access all input states — removing the bottleneck entirely.
>
> Attention allows the model to dynamically focus on relevant parts of the input when generating each output token.
>
> It replaces a single "memory bottleneck" with a **flexible, differentiable weighting system** that decides what to remember — and what to ignore — at every step.