Natural Language Processing

Week 4: Sequence-to-Sequence Models

Breaking the Fixed-Length Barrier

Learning Objectives

By the end of this lecture, you will:

1. Understand why translation is hard for neural networks
2. Design encoder-decoder architectures
3. Identify the information bottleneck problem
4. Master the attention mechanism
5. Implement your own seq2seq model

Prerequisite

Required Knowledge:

- RNNs and LSTMs (Week 3)
- Backpropagation basics
- Softmax function
- Python/NumPy

Time Allocation:

- Part 1: 15 min
- Part 2: 20 min
- Part 3: 15 min
- Part 4: 20 min
- Exercises: 20 min

Week 4 Overview

Build Your Intuition: The Translation Problem

Build Your Intuition

Imagine you're translating a book from English to French. Would you:

- A) Translate word-by-word in order?
- B) Read the whole sentence, understand it, then write in French?
- C) Look at chunks of 5 words at a time?

Think: Why doesn't option A work?

Example - Word-by-word translation fails:

- English: "I gave her the book yesterday"
- French: "Je lui ai donné le livre hier"
- Word-by-word back: "I her have given the book yesterday"

The word order completely changes between languages!

Why Can't We Just Use RNNs?

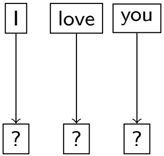
Key Question: *You learned RNNs last week. Why can't we use them for translation?*

The Fundamental Problem:

- RNNs expect: Input length = Output length
- Translation needs: Input length $\neq$ Output length

Concrete Example:

- EN: "I love you" (3 words)
- FR: "Je t'aime" (2 words)
- JP: "Aishiteru" (1 word)
- Which output position gets which input?



Fixed mapping!

The Length Mismatch: Real Data

Let's look at actual translation pairs:

English	Target Language	EN Words	Target Words
I love you	Je t'aime (French)	3	2
I love you	Ich liebe dich (German)	3	3
I love you	Aishiteru (Japanese)	3	1
I love you	Wo ai ni (Chinese)	3	3
I love you	Te amo (Spanish)	3	2
Average length ratio: 3:2.2 (varies by 40%!)			

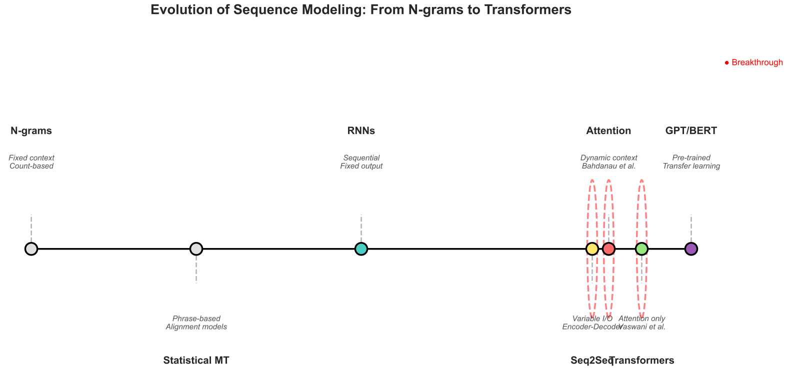
Common Mistake: "Just pad shorter sequences"

- Where to pad? Beginning? End? Middle?
- Model doesn't know target length beforehand
- "Je [PAD] t'aime" $\neq$ "Je t'aime [PAD]"

Check Your Understanding

If padding doesn't work, what's the solution?
Hint: How do human translators handle this?

Evolution of Translation Approaches



Key Insights

- **1950s-1990s**: Rule-based (dictionaries + grammar)

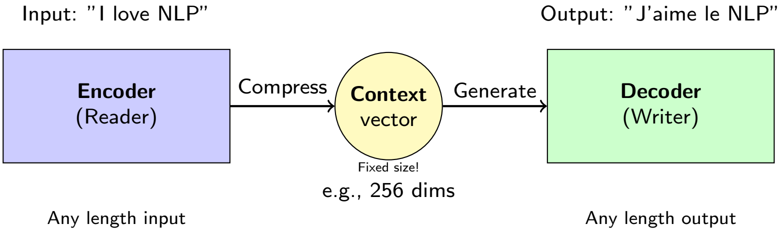
The Brilliant Insight: Two-Stage Process

Build Your Intuition

Think about how YOU translate:

1. **Read** and **understand** the entire sentence
2. Form a mental **representation** of the meaning
3. **Generate** the translation from that understanding

The Seq2Seq Solution (mimics human process):



Check Your Understanding

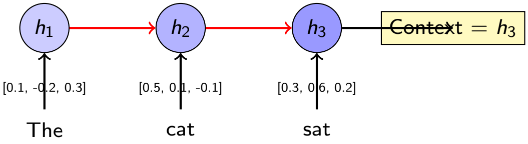
Building Intuition: The Encoder

Build Your Intuition

The encoder is like a **reader** that builds understanding:

- Reads words one by one (like you reading this)
- Updates its understanding with each word
- Final understanding = complete meaning

Step-by-step encoding of "The cat sat":



Try It Yourself

Track how the hidden state changes:

- "The" → General/article context
- "The cat" → Animal/subject identified

Encoder Mathematics (With Intuition)

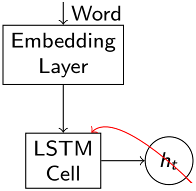
What happens at each step:

For each input word x_t at time t:

$$h_t^{enc} = \text{LSTM}(x_t, h_{t-1}^{enc})$$

Breaking this down:

- x_t = current word (embedded as vector)
- h_{t-1}^{enc} = what we understood so far
- h_t^{enc} = updated understanding



Recurrent connection

Concrete dimensions:

- Word embedding: $x_t \in \mathbb{R}^{100}$
- Hidden state: $h_t \in \mathbb{R}^{256}$
- Context: $c = h_T^{enc} \in \mathbb{R}^{256}$

Quick note: Processing 10 words → 10 hidden states (one per word)

Encoder Implementation (Simplified)

Let's implement what we just learned - it's simpler than you think!

```
1  class Encoder:
2      def __init__(self, vocab_size, hidden_dim):
3          # Two components only!
4          self.embedding = Embedding(vocab_size, 100)
5          self.lstm = LSTM(100, hidden_dim)
6
7      def forward(self, sentence):
8          # sentence = ["I", "love", "NLP"]
9
10         # Start with zero understanding
11         hidden = zeros(hidden_dim)  # [0,0,...,0]
12
13         # Process each word
14         for word in sentence:
15             # Convert word to vector
16             embed = self.embedding[word]  # 100d
17
18             # Update our understanding
19             hidden = self.lstm(embed, hidden)  # 256d
20
21         # Final understanding
22         context = hidden
23         return context  # This is all decoder gets!
```

Line-by-line walkthrough:

- **Lines 3-4**: Just 2 components!
- **Line 10**: Start knowing nothing
- **Lines 13-17**: Core loop
  - Get word vector
  - Update understanding
  - Keep only latest
- **Line 20**: Final state = context

Key Insight

Context size is **always the same**:

- 3 words → 256 dims
- 100 words → Still 256 dims!

This fixed size is both a strength and weakness...

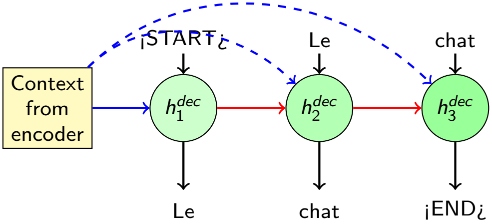
Building Intuition: The Decoder

Build Your Intuition

The decoder is like a **writer** that generates from understanding:

- Starts with the context (understanding)
- Generates one word at a time
- Each word depends on context + previous words

Generation process for "Le chat":



Context used at EVERY step!

Decoder Mathematics (With Intuition)

Generation at each step:

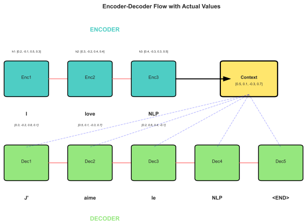
For each output position t:

$$h_t^{dec} = \text{LSTM}(y_{t-1}, h_{t-1}^{dec}, c)$$

$$P(y_t | y_{<t}, c) = \text{softmax}(W \cdot h_t^{dec} + b)$$

Breaking this down:

- y_{t-1} = previous word we generated
- c = context from encoder (always same!)
- h_t^{dec} = decoder's current state
- $P(y_t | \ldots)$ = probability of each word



Concrete example:

- Generating "chat" after "Le"
- Previous: y_{t-1} = "Le" $\rightarrow$ [0.2, 0.1, ...]
- Context: c = [0.3, 0.6, 0.2, ...] (256d)
- Output: P("chat") = 0.7, P("chien") = 0.2, ...

Training Trick: Teacher Forcing

Problem: How do we train when the model makes mistakes early on?

During Training (Teacher Forcing):

- Feed the TRUE previous word
- Not the model's prediction
- Speeds up training dramatically

Example: Teaching "Le chat noir"

1. Input: ¡START¿ → Predict: "Le"
2. Input: "Le" (true) → Predict: "chat"
3. Input: "chat" (true) → Predict: "noir"

During Testing (No Teacher):

- Feed MODEL's previous prediction
- No true words available!
- Errors can accumulate

Example: Generating translation

1. Input: ¡START¿ → Generates: "Le"
2. Input: "Le" (generated) → Generates: "chat"
3. Input: "chat" (generated) → Generates: "noir"

Common mistake: "Teacher forcing at test time"
You CAN'T use teacher forcing during testing - you don't have the true translations!

The Compression Problem

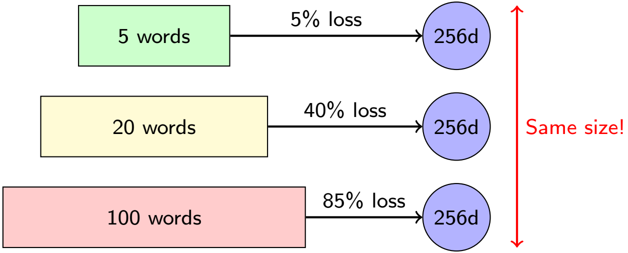
Build Your Intuition

Imagine compressing a book into a single paragraph:

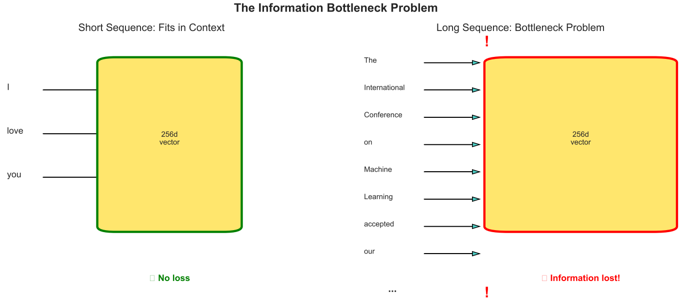
- Short story (5 pages) → Paragraph: Works well!
- Novel (300 pages) → Paragraph: Loses details
- Encyclopedia → Paragraph: Impossible!

Same problem with seq2seq: Longer input → More information loss

The Bottleneck Visualization:



Information Theory Analysis



Key Insights

Information capacity calculation:

- Each word $\approx$ 10 bits of information
- 256-dim vector $\approx$ 256 bits capacity
- 5 words = 50 bits $\rightarrow$ Fits! (20% utilization)
- 20 words = 200 bits $\rightarrow$ Fits (78% utilization)
- 100 words = 1000 bits $\rightarrow$ Overflow! (Need 4x capacity)

Where Information Gets Lost

Let's trace what happens to a long sentence:
"The International Conference on Machine Learning, which is one of the premier venues for presenting research in machine learning and attracts submissions from researchers around the world, accepted our paper."

What the context vector captures:

Y Preserved:

- General topic (ML conference)
- Sentiment (positive - accepted)
- Basic structure (statement)

X Lost:

- "International" detail
- "premier venues" specificity
- "researchers around the world"
- Exact conference name

Experimental Results (Bahdanau et al., 2015):

Sentence Length	BLEU Score	Quality
¡ 10 words	35.2	Excellent
10-20 words	28.5	Good
20-30 words	19.3	Mediocre
¿ 30 words	9.7	Poor

Performance drops 72% for long sentences!

How Humans Translate (The Key Insight)

Build Your Intuition

When you translate "The black cat sat on the mat" to French:

- For "Le" → You look at "The"
- For "chat" → You look at "cat"
- For "noir" → You look at "black"
- You DON'T look at all words equally!

Let's track what we look at:

Generating	Looking at	Attention Weight
"Le"	Mainly "The"	0.8 on "The", 0.2 others
"chat"	Mainly "cat"	0.7 on "cat", 0.3 others
"noir"	Mainly "black"	0.6 on "black", 0.4 others
"s'est assis"	Mainly "sat"	0.9 on "sat", 0.1 others
"sur"	Mainly "on"	0.8 on "on", 0.2 others
"le"	Mainly "the"	0.7 on "the", 0.3 others
"tapis"	Mainly "mat"	0.85 on "mat", 0.15 others

Check Your Understanding

The Attention Solution

Instead of one context vector:

- Keep ALL encoder hidden states
- Let decoder choose what to look at
- Different focus for each output word

The 3-step attention process:

1. **Score:** How relevant is each encoder state?

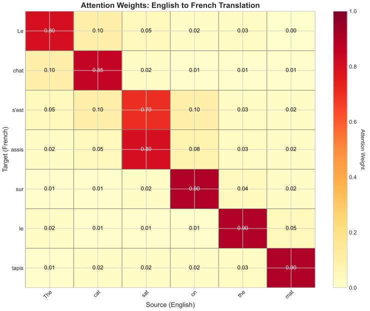
$$e_{ti} = \text{score}(h_t^{dec}, h_i^{enc})$$

2. **Normalize:** Convert to probabilities

$$\alpha_{ti} = \frac{\exp(e_{ti})}{\sum_j \exp(e_{tj})}$$

3. **Combine:** Weighted sum

$$c_t = \sum_i \alpha_{ti} \cdot h_i^{enc}$$



Example: For "chat", attention weights = [0.1, 0.7, 0.2]

- 10% focus on "The"
- **70% focus on "cat"** ← Makes sense!
- 20% focus on "sat"

Attention Calculation: Step by Step

Let's calculate attention for generating "chat":

Step 1: Score each source word

Current decoder state: $h_2^{dec} = [0.5, -0.2, 0.8]$

Word	Hidden State	Score
"The"	[0.1, 0.2, 0.1]	0.09
"cat"	[0.8, 0.1, 0.7]	0.94
"sat"	[0.2, 0.3, 0.2]	0.20

Step 2: Apply softmax

$$\alpha_1 = \frac{e^{0.09}}{e^{0.09} + e^{0.94} + e^{0.20}} = 0.27$$

$$\alpha_2 = \frac{e^{0.94}}{\dots} = \color{red}{0.63}$$

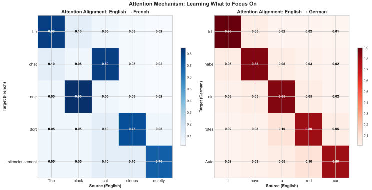
$$\alpha_3 = \frac{e^{0.20}}{\dots} = 0.10$$

Step 3: Weighted combination

$$c_2 = 0.27 \cdot h_1^{enc} + 0.63 \cdot h_2^{enc} + 0.10 \cdot h_3^{enc}$$

Result:

- 63% attention on "cat"
- Correct word alignment!
- Context is mostly "cat"



Attention Implementation

```
1  def attention(decoder_hidden, encoder_outputs):
2      """
3      decoder_hidden: current state [256]
4      encoder_outputs: all states [seq_len, 256]
5      """
6      scores = []
7
8      # Step 1: Score each encoder output
9      for enc_out in encoder_outputs:
10         # Dot product similarity
11         score = dot(decoder_hidden, enc_out)
12         scores.append(score)
13
14     # Step 2: Normalize with softmax
15     scores = array(scores)
16     exp_scores = exp(scores - max(scores))
17     weights = exp_scores / sum(exp_scores)
18
19     # Step 3: Weighted combination
20     context = zeros_like(decoder_hidden)
21     for i, enc_out in enumerate(encoder_outputs):
22         context += weights[i] * enc_out
23
24     return context, weights
25
26 # Usage in decoder:
27 for t in range(max_length):
28     context, attn = attention(hidden, all_enc)
29     # Use context instead of fixed vector!
```

Key improvements:

- **Line 9-11**: Score relevance
- **Line 15-16**: Softmax for probabilities
- **Line 19-21**: Custom context

Try It Yourself

With 10 source words:

- 10 attention weights
- Sum to 1.0
- Different for each output!

Common Misconception

Q: "Does attention look at future words?"
A: No! Only at encoder (source) states, never future target words.

Types of Attention Mechanisms

Three ways to compute attention scores:

1. Dot Product (Luong)

$$e_{ti} = h_t^{dec} \cdot h_i^{enc}$$

- Simplest and fastest
- No parameters to learn
- Works well in practice

2. Scaled Dot Product

$$e_{ti} = \frac{h_t^{dec} \cdot h_i^{enc}}{\sqrt{d}}$$

- Used in Transformers
- Prevents large values
- More stable gradients

3. Additive (Bahdanau)

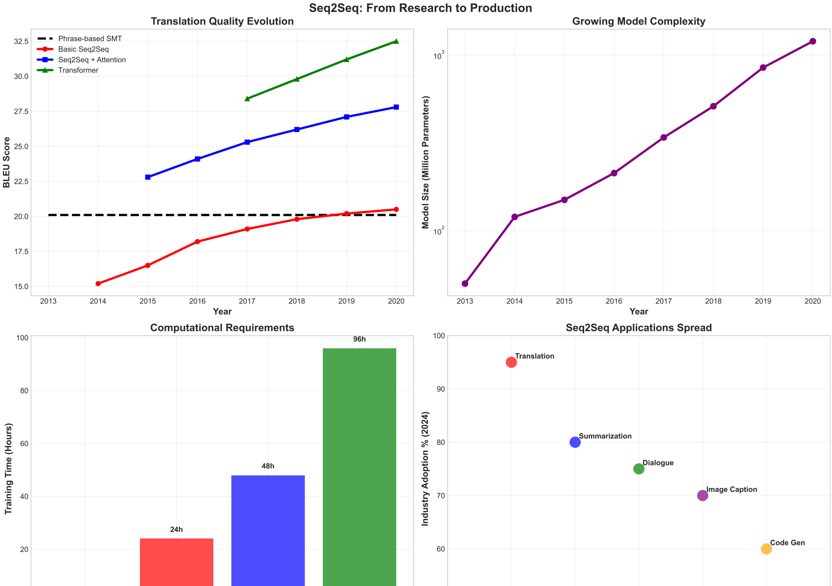
$$e_{ti} = v^T \tanh(W_1 h_t^{dec} + W_2 h_i^{enc})$$

- Original attention paper
- More parameters
- More flexible

Performance comparison:

Type	BLEU	Speed
Dot Product	31.2	Fast
Scaled	31.5	Fast
Additive	31.7	Slower

The Impact of Attention



Complete Mathematical Formulation

Encoder Equations:

$$h_t^{enc} = \text{LSTM}^{enc}(E^{enc}(x_t), h_{t-1}^{enc}) \quad \text{(Process each word)} \tag{1}$$

$$H^{enc} = [h_1^{enc}, h_2^{enc}, \ldots, h_T^{enc}] \quad \text{(Keep all states)} \tag{2}$$

Decoder with Attention:

$$e_{ti} = \text{score}(h_{t-1}^{dec}, h_i^{enc}) \quad \text{(Relevance scores)} \tag{3}$$

$$\alpha_{ti} = \frac{\exp(e_{ti})}{\sum_{j=1}^{T} \exp(e_{tj})} \quad \text{(Attention weights)} \tag{4}$$

$$c_t = \sum_{i=1}^{T} \alpha_{ti} h_i^{enc} \quad \text{(Context vector)} \tag{5}$$

$$h_t^{dec} = \text{LSTM}^{dec}([E^{dec}(y_{t-1}); c_t], h_{t-1}^{dec}) \quad \text{(Decode)} \tag{6}$$

$$P(y_t | y_{<t}, X) = \text{softmax}(W_o[h_t^{dec}; c_t] + b_o) \quad \text{(Output probs)} \tag{7}$$

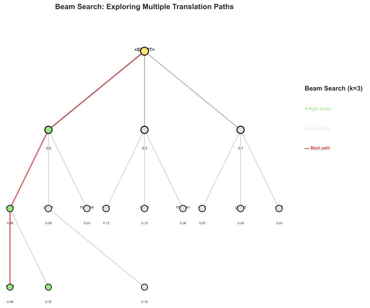
Training Objective:

$$\mathcal{L} = -\sum_{t=1}^{T'} \log P(y_t^* | y_{<t}^*, X) \text{(Cross-entropy loss)} \tag{8}$$

Beam Search Algorithm

Problem: Greedy decoding (always pick highest probability) is suboptimal
Solution: Keep top-k hypotheses at each step

```python
def beam_search(encoder_outputs, beam_size=3):
    # Start with single hypothesis
    beams = [([<START>], 0.0)]

    for t in range(max_length):
        new_beams = []

        for sequence, score in beams:
            if sequence[-1] == <END>:
                completed.add((sequence, score))
                continue

            # Get probabilities for next word
            probs = decode_step(sequence, encoder_outputs)

            # Keep top k words
            top_words = top_k(probs, beam_size)

            for word, prob in top_words:
                new_seq = sequence + [word]
                new_score = score + log(prob)
                new_beams.append((new_seq, new_score))

        # Keep top k beams overall
        beams = sorted(new_beams, key=score)[:beam_size]

    return best_completed()
```



Example with beam_size=2:

- Start: "Le" (0.7), "Un" (0.3)
- After "Le": "chat" (0.6), "chien" (0.1)
- After "Un": "chat" (0.2), "animal" (0.1)
- Keep: "Le chat" (0.42), "Un chat" (0.06)

BLEU Score: Evaluating Translation Quality

BLEU = Bilingual Evaluation Understudy

$$\text{BLEU} = \text{BP} \cdot \exp\left(\sum_{n=1}^{4} w_n \log p_n\right)$$

Where:

- p_n = precision of n-grams
- w_n = weights (usually 0.25 each)
- BP = brevity penalty (penalizes short translations)

Concrete Example:

- Reference: "The cat sat on the mat"
- Hypothesis: "The cat is on the mat"

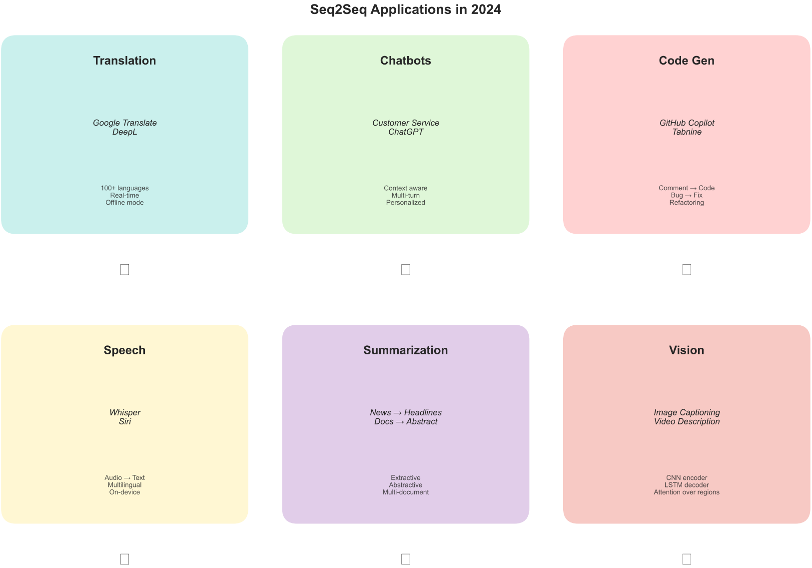
N-gram	Matches	Total	Precision
1-gram	The, cat, on, the, mat	6	$5/6 = 0.83$
2-gram	"The cat", "on the", "the mat"	5	$3/5 = 0.60$
3-gram	"on the mat"	4	$1/4 = 0.25$
4-gram	None	3	$0/3 = 0.00$

Step-by-step calculation:

- Brevity penalty: $BP = e^{1-6/6} = 1.0$ (same length)
- Geometric mean: $\sqrt[4]{0.83 \times 0.60 \times 0.25 \times 0.01} = 0.22$
- Final BLEU: $1.0 \times 0.22 = 0.22$

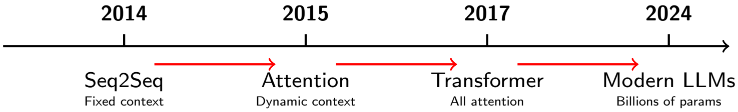
Interpretation: 0.22 means "Understandable but needs improvement"

Seq2Seq in Production Today



From Seq2Seq to Transformers

The Evolution Timeline:



Key Innovations:

1. **Seq2Seq (2014):** Separate encoding and decoding
2. **Attention (2015):** Solve the bottleneck problem
3. **Transformer (2017):** Remove RNNs entirely, use only attention
4. **GPT/BERT (2018+):** Pre-training on massive data

Key Insight: Everything you learned today is the foundation of modern LLMs!
ChatGPT, Claude, and Gemini all build on these seq2seq concepts.

Week 4 Summary: Key Takeaways

Problems Solved:

1. Variable-length I/O
2. Information bottleneck
3. Long-range dependencies
4. Translation alignment

Key Concepts:

- Encoder-Decoder separation
- Context vectors
- Teacher forcing
- Attention mechanism
- Beam search

You Can Now:

- Build a seq2seq model
- Implement attention
- Diagnose bottleneck issues
- Choose attention types
- Evaluate with BLEU

Next Week: Transformers

- "Attention is All You Need"
- Self-attention
- Multi-head attention
- Positional encoding

Check Your Understanding

Quick check: Can you explain why we need TWO networks for translation?
Answer: Because input length $\neq$ output length!