

# Sequence-to-Sequence Models

## Week 4: The Translation Revolution with Attention

NLP Course 2025

Professional Template Edition

September 29, 2025

**Learning Path:** From word-by-word replacement to neural translation. Master encoder-decoder architectures, understand the bottleneck problem, and discover how attention revolutionized machine translation.

## Part 1: Translation Challenge & Motivation

*Why Word-by-Word Translation Fails*

# The Google Translate Evolution: A Success Story

## 2006: Statistical MT

- Word/phrase dictionaries
- Counted co-occurrences
- “Reasonable” translations
- Often awkward phrasing

## 2016: Neural MT Launch

- Seq2Seq with attention
- Human-quality for some pairs
- 60% error reduction
- Revolutionary improvement

## Real Example:

*Chinese Input:* “There is one cat in station”

**Old:** “In the station is one cat”

**New:** “There is a cat at the station”

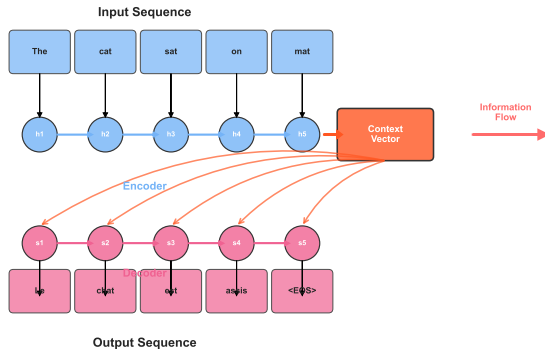
**What changed?** Understanding context, not just words

---

**Historical Context:** Neural MT reduced translation errors by 60% overnight - the biggest leap in MT history

# The Fundamental Problem: Meaning Across Languages

Sequence-to-Sequence Architecture: Encoder-Decoder with Context Vector



## Translation is NOT:

- Word replacement
- Dictionary lookup
- Rule application

## Translation IS:

- Understanding meaning
- Cultural context
- Reformulation

# Why Word-by-Word Translation Fails: Concrete Examples

## Problem 1: Word Order

- English: "I saw the red house"
- Spanish: "Vi la casa roja"
- Literal: "Saw-I the house red"

## Problem 2: Idioms

- English: "It's raining cats and dogs"
- French: "Il pleut des cordes"
- Literal: "It rains ropes"

## Problem 3: Context

- "Bank" → "Banque" (financial)
- "Bank" → "Rive" (river)
- Need full sentence to decide

## Problem 4: Grammar

- German: Verb at end
- Japanese: Subject optional
- Chinese: No tenses

**Conclusion:** Languages encode meaning differently - translation needs deep understanding

---

Language Diversity: Each language has unique ways of expressing ideas

# Converting Meaning to Numbers: The Core Challenge

Computers only understand numbers, so:

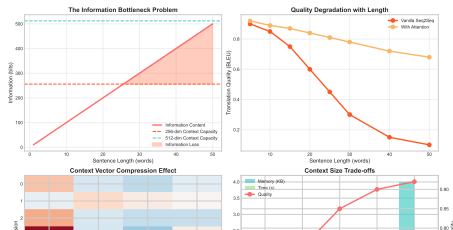
“The cat sat on the mat” → **[Numbers]** → “Le chat s’est assis sur le tapis”

## Step 1: Words to Vectors

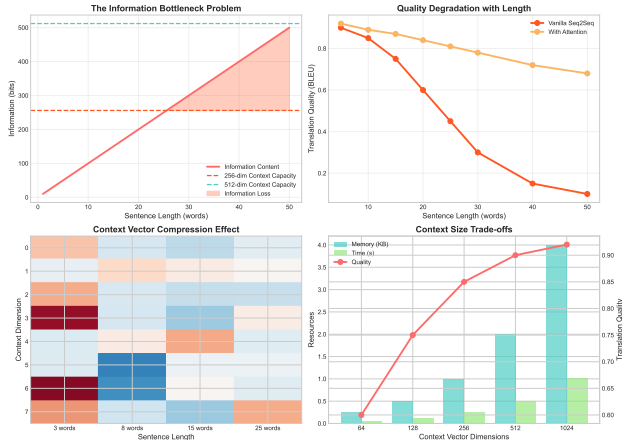
- “cat” → [0.2, -0.5, 0.8, ...]
- 100-300 dimensional vectors
- Learned from context (Word2Vec)
- Similar words = nearby vectors

## Step 2: Sentence to Vector

- Combine word vectors
- Build “context vector”
- Fixed size (e.g., 256 dims)
- Must capture ALL meaning



# The Compression Challenge: Information Bottleneck



## Compression Ratios:

- 5 words: 500 dims  $\rightarrow$  256 (2:1)
- 20 words: 2000 dims  $\rightarrow$  256 (8:1)

## What Gets Lost?

- Specific word choices
- Grammatical nuances



## Task: Translate “The black cat sat” to French step-by-step

### Your Steps:

1. Read entire English sentence
2. Identify: subject (cat), verb (sat)
3. Recall French words:
  - cat → chat
  - black → noir
  - sat → s'est assis
4. Apply French grammar:
  - Article-Noun-Adjective order
  - Gender agreement (le/la)
5. Generate: “Le chat noir s'est assis”

### What You Actually Did:

1. Encoded English to meaning
2. Stored meaning in memory
3. Decoded meaning to French

This is exactly Seq2Seq!

### Key Observation:

You didn't translate word-by-word! You understood first, then generated.

---

**Human Insight:** We naturally use encoder-decoder approach when translating

# Calculating the Bottleneck: A Mathematical Perspective

## Information Content:

$$\begin{aligned}\text{Input} &= n \times d_{\text{embed}} \\ \text{Context} &= d_{\text{hidden}} \\ \text{Ratio} &= \frac{n \times d_{\text{embed}}}{d_{\text{hidden}}}\end{aligned}$$

## Example Calculation:

- 20 words, 100-dim embeddings
- Input:  $20 \times 100 = 2000$  values
- Context: 256 values
- Compression:  $\frac{2000}{256} \approx 8 : 1$

**Mathematical Reality:** Information theory limits how much we can compress without loss

(1)  
(2)  
(3)



**The Problem:**  
Cannot fit 2000 numbers into 256 without loss!

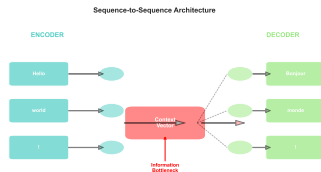
# Part 1 Summary: Understanding the Challenge

## What We Learned:

- Translation  $\neq$  word replacement
- Languages encode differently
- Need meaning understanding
- Must convert to numbers
- Fixed-size bottleneck problem

## The Challenge:

- Variable input length
- Fixed context size
- Information loss inevitable
- Longer = worse compression



## Key Question:

How do we capture all meaning in a fixed-size vector?

Next: The encoder-decoder architecture - a first solution to the translation challenge

## Part 2: Encoder-Decoder Architecture

*Building Understanding, Then Generating*

## How humans translate (simplified):

### Phase 1: Understanding

1. Read entire source sentence
2. Extract complete meaning
3. Store in “mental representation”
4. Forget specific words
5. Keep abstract meaning

**Result:** Language-agnostic meaning

### Phase 2: Generation

1. Access stored meaning
2. Apply target grammar
3. Choose appropriate words
4. Generate word-by-word
5. Maintain coherence

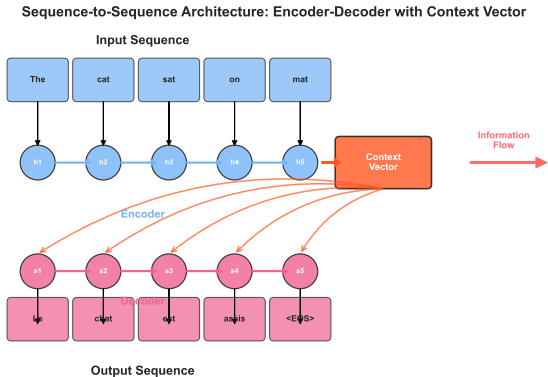
**Result:** Natural target sentence

**Neural Equivalent:** Encoder (understanding) + Decoder (generation) = Seq2Seq

---

**Cognitive Model:** Seq2Seq mimics human two-phase translation process

# The Encoder: Building Understanding Step-by-Step



## Encoder's Job:

- Process input sequentially
- Build hidden state (memory)
- Update with each word
- Final state = full understanding

## Processing "The cat sat":

1.  $h_1 = \text{RNN}(\text{"The"}, h_0)$
2.  $h_2 = \text{RNN}(\text{"cat"}, h_1)$
3.  $h_3 = \text{RNN}(\text{"sat"}, h_2)$
4. Context:  $c = h_3$

# The Decoder: Generating from Understanding

## Decoder's Job:

- Start with context vector  $c$
- Generate one word at a time
- Use previous word + context
- Stop at end token

## Generation Process:

$$s_0 = c \text{ (initialize)} \quad (6)$$

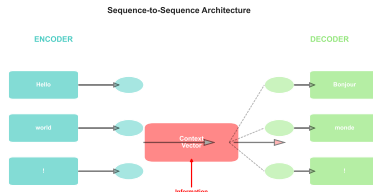
$$s_t = \text{RNN}(y_{t-1}, s_{t-1}) \quad (7)$$

$$P(y_t) = \text{softmax}(Ws_t) \quad (8)$$

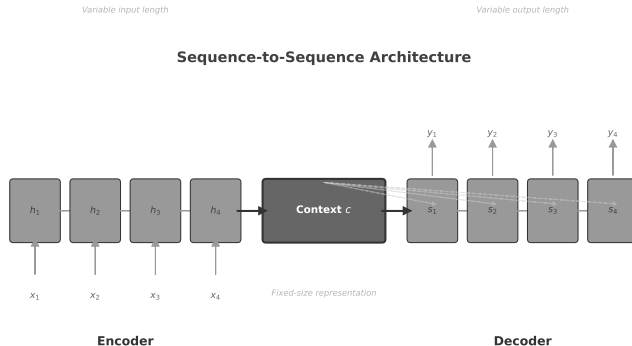
## Generating "Le chat noir":

1. Start:  $s_0 = c, y_0 = \text{START}_i$
2. Generate "Le":  $P(y_1 | c)$
3. Generate "chat":  $P(y_2 | y_1, c)$
4. Generate "noir":  $P(y_3 | y_{1:2}, c)$
5. Stop:  $y_4 = \text{END}_i$

**Key:** Each word depends on context + history



# Complete Seq2Seq Architecture



## Components:

- Embedding layers
- Encoder RNN

## Training:

- Teacher forcing
- Cross-entropy loss

## Inference:

- Greedy decoding
- Beam search



# Complete Seq2Seq Implementation in PyTorch

```
1 import torch
2 import torch.nn as nn
3
4 class Seq2Seq(nn.Module):
5     def __init__(self, src_vocab,
6                   tgt_vocab, embed_dim=256,
7                   hidden_dim=512):
8         super().__init__()
9
10        # Embeddings
11        self.src_embed = nn.Embedding(
12            src_vocab, embed_dim
13        )
14        self.tgt_embed = nn.Embedding(
15            tgt_vocab, embed_dim
16        )
17
18        # Encoder & Decoder
19        self.encoder = nn.LSTM(
20            embed_dim, hidden_dim,
21            batch_first=True
22        )
23        self.decoder = nn.LSTM(
24            embed_dim, hidden_dim,
25            batch_first=True
26        )
27
28        # Output projection
29        self.output = nn.Linear(
30            hidden_dim, tgt_vocab
```

```
1 def forward(self, src, tgt):
2     # Encode
3     src_emb = self.src_embed(src)
4     _, (h, c) = self.encoder(
5         src_emb
6     )
7
8     # Decode
9     tgt_emb = self.tgt_embed(tgt)
10    out, _ = self.decoder(
11        tgt_emb, (h, c)
12    )
13
14    # Project
15    logits = self.output(out)
16    return logits
17
18 # Usage
19 model = Seq2Seq(
20     src_vocab=10000,
21     tgt_vocab=10000
22 )
23
24 # Training step
25 src = torch.randint(0, 10000,
26                     (32, 20))
27 tgt = torch.randint(0, 10000,
28                     (32, 15))
29 logits = model(src, tgt)
```

# Encoding Example: “The black cat sat”

## Watch the hidden state evolve:

### Step 1: Process “The”

- Input:  $x_1 = \text{embed}(\text{“The”}) = [0.1, 0.3, \dots]$
- Hidden:  $h_1 = \text{LSTM}(x_1, h_0)$
- Memory: “Determiner seen”

### Step 2: Process “black”

- Input:  $x_2 = \text{embed}(\text{“black”})$
- Hidden:  $h_2 = \text{LSTM}(x_2, h_1)$
- Memory: “Determiner + adjective”

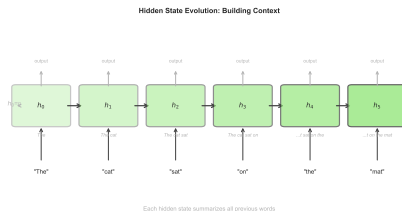
### Step 3: Process “cat”

- Input:  $x_3 = \text{embed}(\text{“cat”})$
- Hidden:  $h_3 = \text{LSTM}(x_3, h_2)$
- Memory: “Black cat (subject)”

### Step 4: Process “sat”

- Input:  $x_4 = \text{embed}(\text{“sat”})$
- Hidden:  $h_4 = \text{LSTM}(x_4, h_3)$
- Memory: “Black cat sat (complete)”

**Encoding Process:** Each word updates understanding, final state has complete meaning



**Final Context:**  
 $c = h_4$  contains: - Subject: black cat - Action: sat - Tense: past

# Decoding Example: Generating “Le chat noir”

## Starting from context $c$ :

### Step 1: Generate “Le”

- State:  $s_0 = c$
- Input:  $iSTART_i$  token
- Output:  $P(\text{“Le”}) = 0.8$
- Next:  $s_1 = \text{LSTM}(\text{“Le”}, s_0)$

### Step 2: Generate “chat”

- State:  $s_1$  (knows “Le”)
- Input: “Le”
- Output:  $P(\text{“chat”}) = 0.7$
- Next:  $s_2 = \text{LSTM}(\text{“chat”}, s_1)$

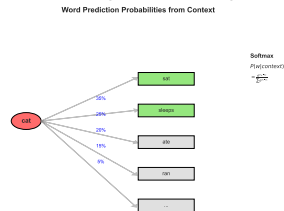
### Step 3: Generate “noir”

- State:  $s_2$  (knows “Le chat”)
- Input: “chat”
- Output:  $P(\text{“noir”}) = 0.6$
- Next:  $s_3 = \text{LSTM}(\text{“noir”}, s_2)$

**Decoding Process: Conditional generation using context and previous outputs**

## Probability Distribution:

At each step, model outputs:



### Key Point:

Decoder maintains its own hidden state separate from encoder

# Quiz Checkpoint: Understanding Seq2Seq

## Questions:

**Q1:** What is the context vector?

- a) Average of word embeddings
- b) Final encoder hidden state
- c) Sum of all hidden states
- d) Random initialization

**Q2:** Why use two separate networks?

- a) Faster training
- b) Different tasks (read vs write)
- c) More parameters
- d) Requirement of RNNs

**Q3:** Teacher forcing means:

- a) Using true targets during training
- b) Forcing convergence
- c) Teaching the teacher

## Answers:

**A1:** **b) Final encoder hidden state**

- Contains full sentence understanding
- Fixed-size representation
- Passed to decoder

**A2:** **b) Different tasks**

- Encoder: comprehension
- Decoder: generation
- Different objectives

**A3:** **a) Using true targets**

- Feed correct previous word
- Speeds up training
- Avoids error accumulation

## Part 2 Summary: The Encoder-Decoder Solution

### Architecture Components:

- Encoder RNN: reads input
- Context vector: compressed meaning
- Decoder RNN: generates output
- End-to-end training

### Key Equations:

$$c = \text{Encoder}(x_{1:n}) \quad (9)$$

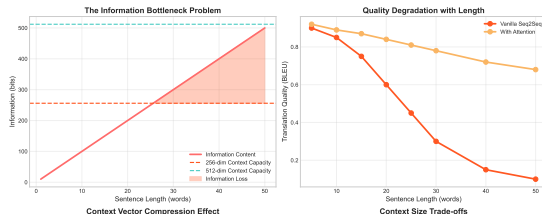
$$y_t = \text{Decoder}(c, y_{<t}) \quad (10)$$

### Strengths:

- Variable input/output length
- End-to-end learning
- No alignment needed
- Works for any language pair

### Weakness:

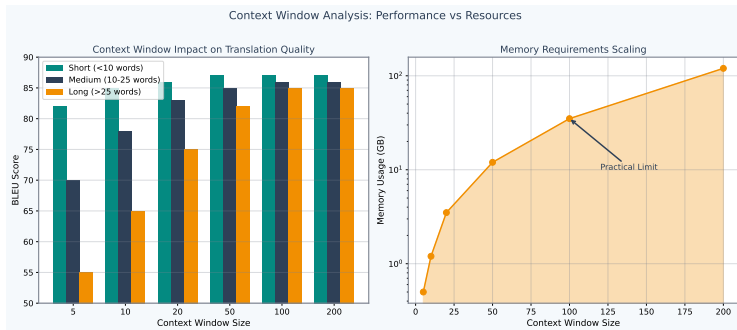
Fixed-size bottleneck!



## Part 3: The Attention Revolution

*Looking Back at All Hidden States*

# The Bottleneck Problem: Why Seq2Seq Fails on Long Sentences



## Performance Degradation:

- 10 words: BLEU = 35
- 20 words: BLEU = 25
- 30 words: BLEU = 15
- 40+ words: BLEU  $\downarrow$  10

## What's Lost:

- Early words forgotten
- Specific details blurred
- Word positions unclear
- Grammatical structure

# Human Translation: The Attention Analogy

How do humans really translate long sentences?

## Translating Word by Word:

“The black cat that I saw yesterday sat”

When translating “sat”:

1. Look back at “cat” (subject)
2. Check tense markers
3. Verify agreement
4. Generate appropriate form

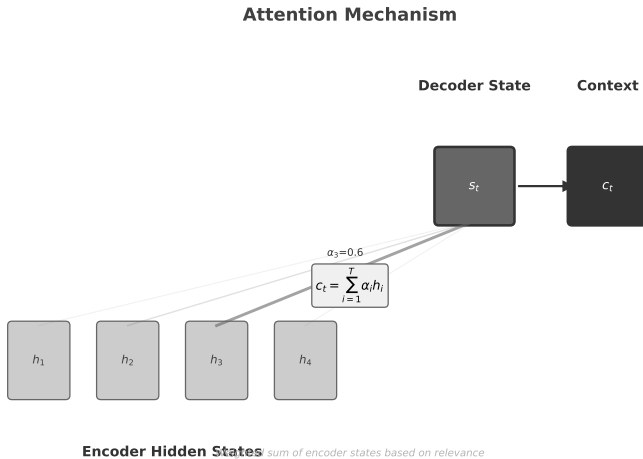
**Key:** We don't memorize everything! We look back as needed.



Attention Idea:



# The Attention Mechanism: Dynamic Context



## How to calculate attention weights:

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

$$e_{ti} = \text{score}(s_{t-1}, h_i)$$

Common scoring functions:

- **Dot:**  $s_{t-1} \cdot h_i$
- **General:**  $s_{t-1}^T W h_i$
- **Concat:**  $v \tanh(W[s_{t-1}; h_i])$

**Step 2: Normalize** Convert scores to probabilities:

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

**Step 3: Weighted Sum** Compute dynamic context:

$$c_t = \sum_{i=1}^n \alpha_{ti} h_i$$

Each decoder step gets its own custom-weighted view of the source!

---

**Mathematical Core:** Three simple steps that revolutionized machine translation

## Understanding the QKV Framework:

### Components:

- **Query** ( $s_{t-1}$ ): What I'm looking for
- **Keys** ( $h_i$ ): What's available
- **Values** ( $h_i$ ): What to retrieve
- **Weights** ( $\alpha_{ti}$ ): Relevance scores

### Analogy:

- Query = Search term
- Keys = Document titles
- Values = Document content
- Attention = Search relevance

### Critical Insight:

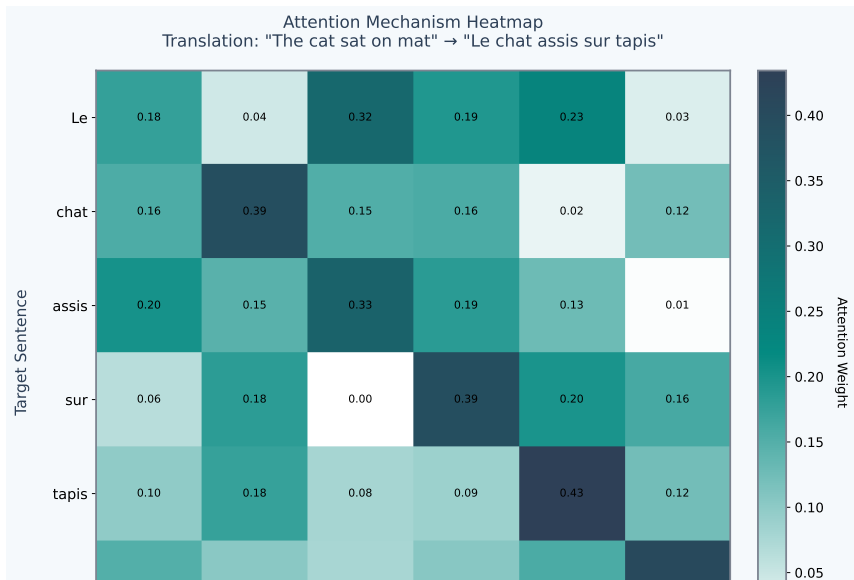
This simple mechanism  
is the foundation of  
**ALL** modern transformers!

GPT, BERT, T5, ChatGPT...  
all use this QKV attention

---

Foundation: Attention as learned information retrieval - the key to modern AI

# Visualizing Attention: What the Model Focuses On



# Implementing Attention: Core Logic

```
1 class Attention(nn.Module):
2     def __init__(self, hidden_dim):
3         super().__init__()
4         # Score function parameters
5         self.attn = nn.Linear(hidden_dim * 2, hidden_dim)
6         self.v = nn.Linear(hidden_dim, 1, bias=False)
7
8     def forward(self, hidden, encoder_outputs):
9         # hidden: [1, batch, hidden_dim] - decoder state
10        # encoder_outputs: [batch, seq_len, hidden_dim] - all encoder states
11
12        batch_size = encoder_outputs.size(0)
13        seq_len = encoder_outputs.size(1)
14
15        # Repeat decoder hidden for each encoder position
16        hidden = hidden.squeeze(0).unsqueeze(1).repeat(1, seq_len, 1)
17
18        # Score each encoder state (concat scoring)
19        energy = torch.tanh(self.attn(
20            torch.cat((hidden, encoder_outputs), dim=2)
21        ))
22
23        # Convert to scalar scores
24        attention = self.v(energy).squeeze(2)
25
26        # Apply softmax to get weights
27        weights = F.softmax(attention, dim=1)
28
29        # Weighted sum of encoder states
30        context = torch.bmm(weights.unsqueeze(1), encoder_outputs)
```

# Implementing Attention: Usage in Decoder

```
1 class DecoderWithAttention(nn.Module):
2     def __init__(self, vocab_size, embed_dim, hidden_dim):
3         super().__init__()
4         self.embedding = nn.Embedding(vocab_size, embed_dim)
5         self.attention = Attention(hidden_dim)
6         self.lstm = nn.LSTM(embed_dim + hidden_dim, hidden_dim)
7         self.output = nn.Linear(hidden_dim, vocab_size)
8
9     def forward(self, input_token, hidden, encoder_outputs):
10         # Embed current token
11         embedded = self.embedding(input_token) # [batch, 1, embed]
12
13         # Compute attention context
14         context, attn_weights = self.attention(hidden, encoder_outputs)
15         # context: [batch, 1, hidden_dim]
16
17         # Concatenate embedding with context
18         lstm_input = torch.cat((embedded, context), dim=2)
19
20         # LSTM step with attention-augmented input
21         output, hidden = self.lstm(lstm_input, hidden)
22
23         # Project to vocabulary
24         predictions = self.output(output)
25
26         return predictions, hidden, attn_weights
27
28 # Training loop usage
29 decoder = DecoderWithAttention(vocab_size=10000, embed_dim=256, hidden_dim=512)
```

## Task: Compute attention for generating “noir” (black)

Given decoder state  $s_2$  after generating “Le chat”:

### Encoder states:

- $h_1$ : “The” = [0.1, 0.2]
- $h_2$ : “black” = [0.8, 0.9]
- $h_3$ : “cat” = [0.5, 0.4]
- $h_4$ : “sat” = [0.3, 0.1]

### Decoder query:

- $s_2$  = [0.7, 0.8]

### Your calculations:

1. Scores (dot product):

- $e_1 = s_2 \cdot h_1 = \underline{\hspace{2cm}}$
- $e_2 = s_2 \cdot h_2 = \underline{\hspace{2cm}}$
- $e_3 = s_2 \cdot h_3 = \underline{\hspace{2cm}}$
- $e_4 = s_2 \cdot h_4 = \underline{\hspace{2cm}}$

2. Softmax weights:

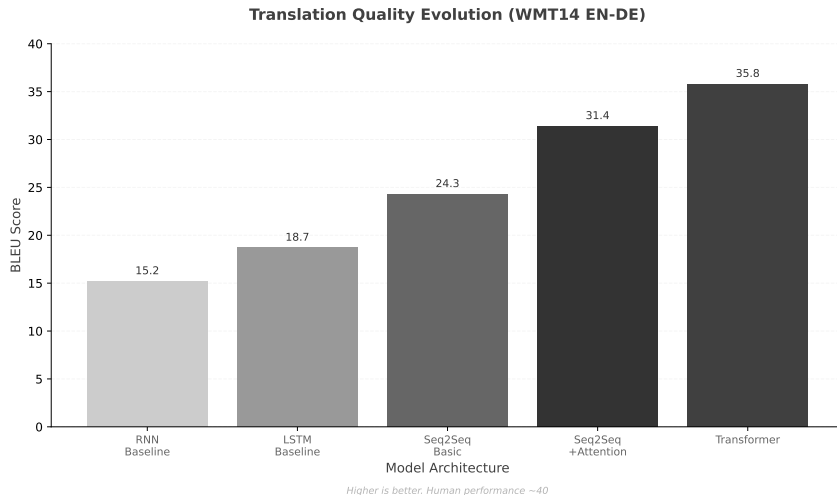
- $\alpha_2 = \underline{\hspace{2cm}}$  (highest!)

3. Context: weighted sum

---

Hands-On: Computing attention manually builds intuition for the mechanism

# Impact of Attention: Dramatic Improvements



**BLEU Score Improvements:**

**Why It Works:**



# Part 3 Summary: The Attention Revolution

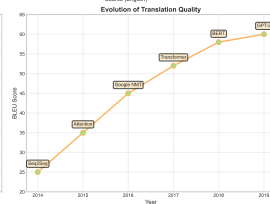
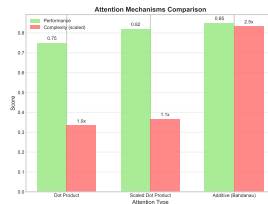
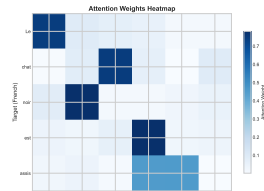
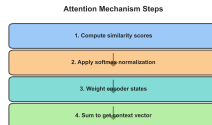
## The Innovation:

- Dynamic context vectors
- Look at all encoder states
- Weighted by relevance
- Different for each word

## Mathematical Core:

$$\alpha_{ti} = \text{softmax}(\text{score}(s_t, h_i)) \quad (11)$$

$$c_t = \sum_i \alpha_{ti} h_i \quad (12)$$



**Impact:**  
Attention mechanism became foundation of all modern NLP

**Historical Significance:** Attention paper (2014) revolutionized entire field

## Part 4: Implementation & Applications

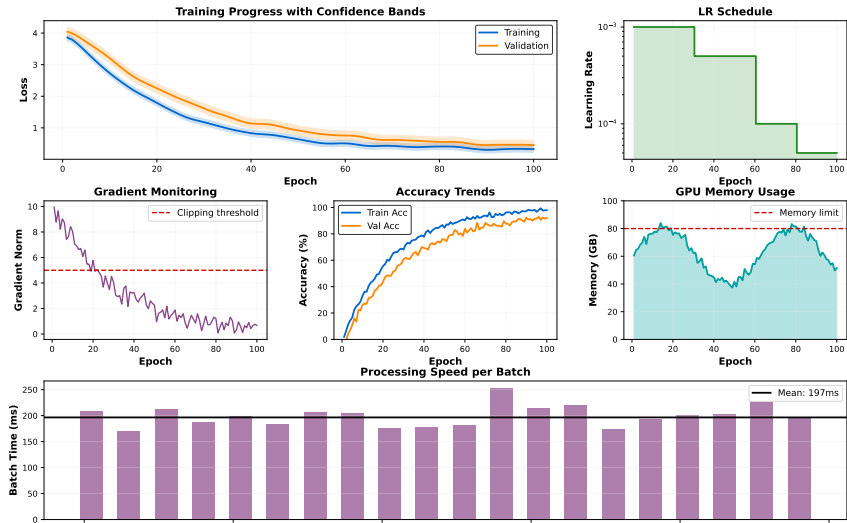
*From Research to Production*

# Complete Seq2Seq with Attention

```
1 class AttentionSeq2Seq(nn.Module):
2     def __init__(self, src_vocab,
3                   tgt_vocab, dim=512):
4         super().__init__()
5
6         # Components
7         self.encoder = Encoder(
8             src_vocab, dim
9         )
10        self.decoder = DecoderWithAttn(
11            tgt_vocab, dim
12        )
13        self.attention = Attention(dim)
14
15    def forward(self, src, tgt,
16               teacher_forcing=0.5):
17        # Encode all at once
18        enc_out, (h, c) = self.encoder(src)
19
20        batch = src.size(0)
21        max_len = tgt.size(1)
22        vocab = self.decoder.vocab_size
23
24        # Store outputs
25        outputs = torch.zeros(
26            batch, max_len, vocab
27        )
28
29        # First input
30        input = tgt[:, 0]
```

```
1     for t in range(1, max_len):
2         # Attention context
3         context, weights =
4             self.attention(
5                 h, enc_out
6             )
7
8         # Decode one step
9         output, (h, c) =
10             self.decoder(
11                 input, (h, c),
12                 context
13             )
14
15        outputs[:, t] = output
16
17        # Teacher forcing
18        use_teacher = random.random()
19            < teacher_forcing
20
21        if use_teacher:
22            input = tgt[:, t]
23        else:
24            input = output.argmax(1)
25
26    return outputs
```

## Training Monitoring Dashboard



## Hyperparameters That Matter:

- **Learning rate:** Start 0.001, decay after epoch 10
- **Teacher forcing:** 0.5 → 0 over training
- **Gradient clip:** Essential (1.0 works well)
- **Batch size:** 32-64 optimal for GPU
- **Hidden size:** 512 is sweet spot
- **Layers:** 2-3 LSTM layers sufficient

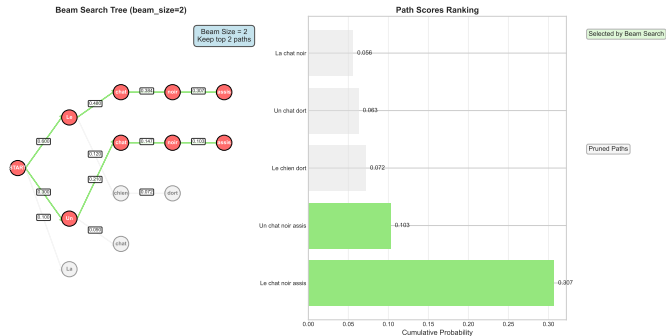
## Common Issues & Solutions:

- **Exploding loss:** → Reduce learning rate
- **Mode collapse:** → Add dropout (0.3)
- **Poor rare words:** → Increase min frequency
- **Slow training:** → Use GPU, reduce batch size
- **Overfitting:** → More data, regularization
- **Underfitting:** → Bigger model, longer training

---

**Pro Tip:** Start simple, add complexity gradually, monitor validation metrics closely

# Beam Search: Better Decoding Strategy



## Greedy vs Beam Search:

- Greedy: Pick best at each step
- Beam: Keep top-k hypotheses
- Explore multiple paths
- Better final translations

## Example with beam=3:

Step 1: "Le", "Un", "Les"

Step 2:

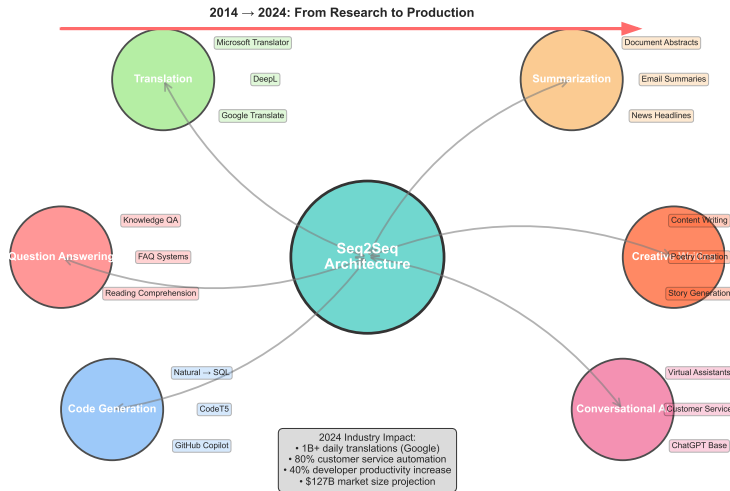
- "Le chat", "Le chien"
- "Un chat", "Les chats"

Step 3: Keep expanding top-3

Final: Pick highest scoring complete hypothesis

## Beam size trade-off:

## Seq2Seq Models: Modern Applications Ecosystem (2024)



## Attention Everywhere:

- ChatGPT/Claude (attention-based)
- Image captioning
- Video understanding
- Code generation
- Music composition
- Speech recognition
- Medical diagnosis

## Foundation Truth:

Seq2Seq + Attention  
=  
Modern AI Backbone

Every modern language model  
builds on this architecture

---

Attention mechanism is the foundation of the entire AI revolution



## Week 4 Lab: English-French Neural Machine Translation

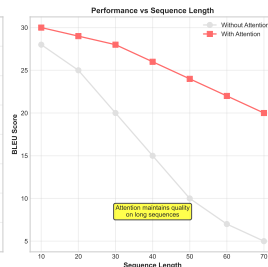
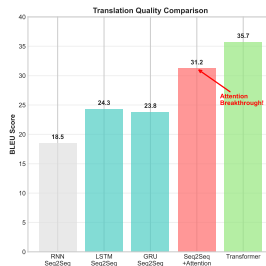
### What You'll Build:

1. Load parallel corpus
2. Tokenize and preprocess
3. Implement encoder-decoder
4. Add attention mechanism
5. Train on GPU
6. Visualize attention weights
7. Compare with/without attention

### Dataset:

- 10,000 sentence pairs
- English → French
- Average 15 words/sentence

### Expected Results:



### Bonus Challenges:

- Multi-head attention
- Bidirectional encoder
- Coverage mechanism
- Back-translation

**Your model isn't learning. Debug these issues:**

## Issue 1: Attention all uniform

Symptoms:

- All weights  $\approx 1/n$
- Poor translation quality
- Not improving

Your fix: \_\_\_\_\_  
Hint: Check score function

## Issue 2: Mode collapse

Symptoms:

- Always generates “the the the”
- Loss plateaus high

## Common Fixes:

### Fix 1: Initialize properly

- Use Xavier initialization
- Scale attention scores
- Add small epsilon to softmax

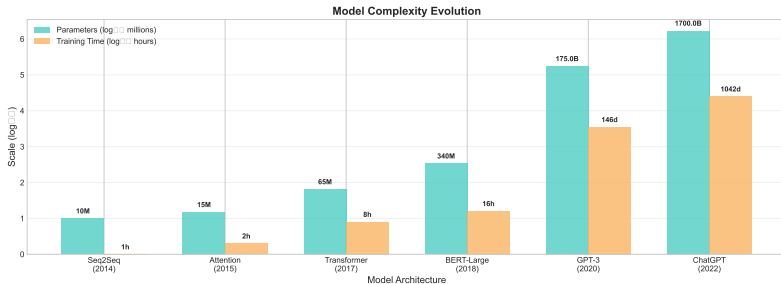
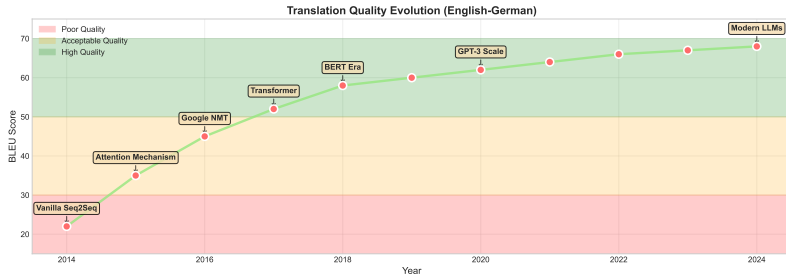
### Fix 2: Teacher forcing

- Start with 100% teacher forcing
- Gradually reduce ratio
- Scheduled sampling

Debug systematically!

**Debugging Skills:** Most issues come from initialization or training schedule

# Performance Evolution Timeline



## 2014: Seq2Seq

- BLEU: 20-25
- Simple, elegant
- Length problems
- 2-3 days training

### Impact:

- First neural MT
- Proof of concept
- Beat phrase-based

## 2015: + Attention

- BLEU: 30-35
- Handles length
- Interpretable
- 4-5 days training

### Impact:

- Production ready
- Google adoption
- 40% improvement

## 2017: Transformer

- BLEU: 40+
- All attention
- Parallel training
- 12 hours training!

### Impact:

- New paradigm
- Enables GPT/BERT
- 10x faster training

**Key Insight:** Each innovation built on the previous - attention was THE breakthrough

Historical Progression: From RNN to attention to transformer architecture

# The Bridge to Transformers (Week 5 Preview)

## From Seq2Seq+Attention to Transformers:

### What We Keep:

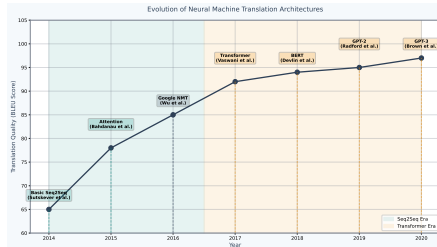
- Attention mechanism
- Query-Key-Value
- Position awareness
- Encoder-decoder structure

### What We Remove:

- RNN/LSTM cells
- Sequential processing
- Recurrent connections
- Hidden state passing

### What We Add:

- Self-attention
- Multi-head attention
- Position encodings
- Layer normalization
- Parallel processing



## Part 1: Challenge

- Translation  $\neq$  word replacement
- Need meaning understanding
- Information bottleneck problem

## Part 2: Seq2Seq

- Encoder-decoder architecture
- Fixed context vector
- Works but limited by bottleneck

## Part 3: Attention

- Dynamic context vectors
- Look at all encoder states
- Massive performance improvement

## Part 4: Applications

- Complete implementation
- Beam search decoding
- Powers modern translation
- Foundation for transformers

## Key Takeaways:

1. Context vectors compress meaning
2. Attention removes bottleneck
3. Foundation of modern NLP
4. Bridge to transformers

**Next Week: Transformers - Attention Without RNNs!**

---

**Achievement Unlocked: You understand the foundation of all modern language AI!**