Natural Language Processing

Week 4: The Compression Journey

From Meaning to Numbers and Back Again

Learning Objectives

By the end of this lecture, you will understand:

1. Why we need numbers to represent words (from first principles)
2. How compression creates an information bottleneck
3. What "context" and "hidden state" actually mean
4. How attention solves the compression problem
5. Why this matters for all modern NLP

Prerequisites from Week 3:

- Basic understanding that neural networks process numbers
- Concept of sequential processing (RNN idea)
- Backpropagation intuition

Table of Contents

The Core Problem: Computers Don't Understand Words

Start with the fundamental challenge:

You want to translate: "The black cat sat on the mat" → French

The Computer's Dilemma:

- Computer sees: `['T', 'h', 'e', ' ', 'b', 'l', 'a', ...]`
- These are just character codes (bytes)
- **No meaning, no relationships, no structure**

What the computer sees:

- "cat" = [99, 97, 116]
- "dog" = [100, 111, 103]
- "sat" = [115, 97, 116]

Problem:

- "cat" and "sat" share [97, 116]
- Does that mean they're similar?
- No! Character overlap $\neq$ meaning

Key Question: How do we give computers a "numerical understanding" of word meaning?

From Words to Numbers: The Embedding Idea

The solution: Represent each word as a vector of numbers

Build intuition with simple example:
Imagine describing animals with just 3 properties:

- Size (0=tiny, 1=huge)
- Cuteness (0=scary, 1=adorable)
- Speed (0=slow, 1=fast)

Word	Size	Cute	Speed
cat	0.3	0.9	0.6
dog	0.5	0.8	0.5
mouse	0.1	0.7	0.8
elephant	0.95	0.4	0.2

Now computers can compute:

- Similarity: cat $\approx$ dog (vectors are close)
- Difference: cat $\neq$ elephant (vectors are far)
- **This is called a "word embedding"**

Reality: We use 100-300 dimensions (not just 3), learned from data

What is a "Hidden State"? Building Intuition

Now we have numbers for words. Next problem: Understanding sentences

Human analogy - Reading comprehension:
As you read "The black cat sat on the mat":

1. After "The" → You know: article, something coming
2. After "The black" → You know: a dark-colored thing
3. After "The black cat" → You know: a specific animal
4. After full sentence → You know: complete scene

Your "understanding" evolves as you read!

Neural network equivalent:

- Network maintains a vector that represents "current understanding"
- This vector updates with each new word
- **This evolving vector is called the "hidden state"**
- Final hidden state = complete understanding of sentence

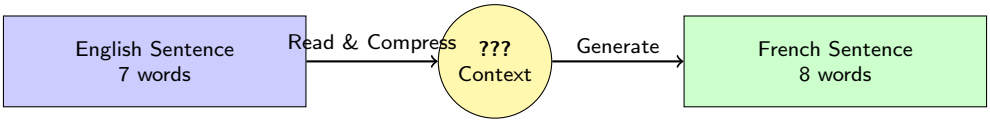
Technical name: When we update understanding word-by-word, we call this a "Recurrent Neural Network" (RNN from Week 3)

The Compression Problem Emerges

Now the real challenge appears:

We need to translate: "The black cat sat on the mat" → "Le chat noir s'est assis sur le tapis"

Two-stage process (like human translation):



The bottleneck:

- 7 words of meaning → compressed to 256 numbers
- Then generate 8 words from just those 256 numbers
- **Can 256 numbers hold all the information?**

Key Question: What happens with longer sentences? 100 words → still 256 numbers?

Quantifying the Compression Problem

Let's calculate how much compression we're doing:

Information content (rough estimate):

- Each word embedding: 100 dimensions (numbers)
- 7-word sentence: $7 \times 100 = 700$ numbers of information
- Context vector: **only 256 numbers**
- **Compression ratio: 700:256 $\approx$ 2.7:1**

What about longer sentences?

Length	Input Dims	Context Dims	Ratio	Quality
5 words	500	256	2:1	Good
20 words	2000	256	8:1	Mediocre
50 words	5000	256	20:1	Poor
100 words	10000	256	40:1	Very Poor

The Information Bottleneck: Longer sentences lose more information!
Like trying to fit a whole book into a single paragraph - something must be lost.

Next question: Can we avoid this bottleneck? (Spoiler: Yes, with attention!)

The Two-Network Architecture

The key insight: Separate "reading" from "writing"

Why two networks? Build from human behavior:
When YOU translate:

1. **Phase 1 (Reading):** Read and understand the English sentence
   - Process word-by-word
   - Build complete understanding
   - Store meaning in your memory
2. **Phase 2 (Writing):** Generate the French translation
   - Start from your understanding
   - Generate word-by-word in French
   - Use grammar and vocabulary of target language

Neural equivalent:

- **Encoder network:** Reads input, builds "hidden state" (understanding)
- **Context vector:** Final hidden state (compressed meaning)
- **Decoder network:** Generates output from context

Technical names you now understand:

- "Sequence-to-Sequence" (Seq2Seq) = this two-network setup
- "Encoder-Decoder architecture" = same thing

Encoder: Building Understanding Step-by-Step

Concrete example: Encoding "The cat sat"

Step-by-step processing:
Step 1: Read "The"

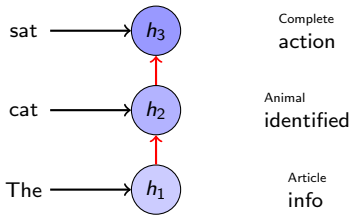
- Convert to embedding: [0.2, 0.5, -0.1, ...] (100d)
- Initial understanding: $h_0 = [0, 0, 0, ...]$ (256d)
- Update: $h_1 = \text{RNN}(\text{"The"}, h_0)$
- New understanding: [0.1, -0.05, 0.03, ...] (256d)

Step 2: Read "cat"

- Embedding: [0.7, -0.3, 0.4, ...] (100d)
- Previous understanding: h_1
- Update: $h_2 = \text{RNN}(\text{"cat"}, h_1)$
- New understanding: [0.3, 0.2, -0.1, ...] (256d)

Step 3: Read "sat"

- Embedding: [-0.2, 0.6, 0.1, ...]
- Update: $h_3 = \text{RNN}(\text{"sat"}, h_2)$
- **Final understanding: h_3 = context vector**



Key insight:

- Each h_t = accumulated understanding
- Always 256 dimensions
- Final h_3 goes to decoder

Decoder: Generating from Understanding

Now generate French from the context vector

Generation process:

Step 0: Start

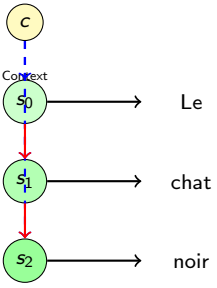
- Input: ¡START¿ token
- Context: $c = h_3$ from encoder (256d)
- Generate: $s_0 = \text{RNN}(\text{¡START¿}, c)$
- Predict probabilities: P("Le") = 0.6, P("Un") = 0.3, ...
- **Choose "Le"**

Step 1: Continue

- Input: "Le" (what we just generated)
- Context: still c (same context!)
- Generate: $s_1 = \text{RNN}(\text{"Le"}, s_0, c)$
- Predict: P("chat") = 0.7, P("chien") = 0.2, ...
- **Choose "chat"**

Step 2: Continue until ¡END¿

- Input: "chat"
- Generate: $s_2 = \text{RNN}(\text{"chat"}, s_1, c)$
- Keep going until model outputs ¡END¿



Key observations:

- Context c used at every step
- Previous word fed back in
- Generate one word at a time
- Stop when ¡END¿ predicted

What Does "Dot Product Similarity" Mean?

Building geometric intuition from scratch

Start with 2D vectors (easy to visualize):

Two vectors in 2D space:

- $\vec{a} = [3, 4]$
- $\vec{b} = [4, 3]$

Dot product calculation:

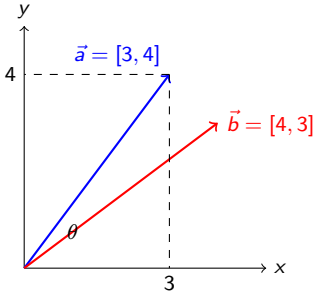
$$\vec{a} \cdot \vec{b} = (3 \times 4) + (4 \times 3)$$
$$= 12 + 12 = 24$$

Geometric meaning:

- Vectors point similar direction → Large positive value
- Perpendicular → Zero
- Opposite directions → Large negative value

Why this matters for attention:

- Decoder state: $h_t^{dec} = [0.5, -0.2, 0.8, ...]$ (256d)
- Encoder state: $h_i^{enc} = [0.6, -0.1, 0.7, ...]$ (256d)
- Dot product $h_t^{dec} \cdot h_i^{enc}$ = how aligned they are
- **High value = encoder state i is relevant for decoder step t**



Small angle θ = similar direction

The Bottleneck Returns: Experimental Evidence

Does the encoder-decoder actually work well?

Experimental results (Bahdanau et al., 2014):

Sentence Length	Compression	BLEU Score	Quality
5-10 words	2:1	31.5	Good
10-20 words	5:1	26.3	Acceptable
20-30 words	10:1	18.7	Poor
30-40 words	15:1	12.4	Very Poor
40+ words	20:1	8.1	Terrible

What gets lost in long sentences?
Input: "The International Conference on Machine Learning, which is one of the premier venues for presenting machine learning research and attracts submissions from researchers around the world, accepted our paper."
Translation loses: "International", "premier venues", "around the world" details
Keeps only: General topic (ML conference), sentiment (positive), main fact (paper accepted)

Problem: Single context vector is the bottleneck - longer sentences lose 70+ percent quality!

The Human Insight: Selective Focus

How do YOU actually translate?

Translating: "The black cat sat on the mat" → "Le chat noir s'est assis sur le tapis"

Honest introspection:

- Writing "Le" → You look back at "The"
- Writing "chat" → You look back at "cat"
- Writing "noir" → You look back at "black"
- Writing "s'est assis" → You look back at "sat"

Key observation:

- You DON'T compress everything into one memory
- You keep the original English visible
- You **selectively attend** to relevant words
- Different output words need different input words

Brilliant idea (Bahdanau et al., 2015): Let the decoder look back at ALL encoder states, not just the final one!

Attention Mechanism: From Intuition to Math

The attention solution in 3 steps:

Step 1: Keep all encoder states (not just last one)

- After encoding "The cat sat": Keep $[h_1^{enc}, h_2^{enc}, h_3^{enc}]$
- Each h_i^{enc} represents understanding up to word i
- **No compression yet - all information preserved!**

Step 2: Compute relevance scores (the "attention" scores)

- For each decoder step t, compute how relevant each encoder state is
- Use dot product (geometric similarity we learned earlier):
- $score_i = h_t^{dec} \cdot h_i^{enc}$
- High score = state i is relevant for current output

Step 3: Weighted combination (dynamic context)

- Convert scores to probabilities (softmax): $\alpha_i = \frac{\exp(score_i)}{\sum_j \exp(score_j)}$
- Weighted average: $context_t = \sum_i \alpha_i \cdot h_i^{enc}$
- **Different context for each decoder step!**

Technical name: These α_i weights are called "attention weights" (where the model is "paying attention")

Attention: Concrete Numerical Example

Let's calculate attention for generating "chat"

Setup:

- Encoder states from "The cat sat": h_1, h_2, h_3 (each 256d)
- Decoder state when generating 2nd word: s_1 (256d)

Step 1: Compute relevance scores

$$
\begin{aligned}
score_1 &= s_1 \cdot h_1^{enc} && = 0.08 && (\text{"The"}) \\
score_2 &= s_1 \cdot h_2^{enc} && = 0.92 && (\text{"cat"}) \\
score_3 &= s_1 \cdot h_3^{enc} && = 0.15 && (\text{"sat"})
\end{aligned}
$$

Step 2: Softmax to get weights

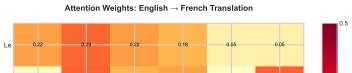
$$
\begin{aligned}
\alpha_1 &= \frac{e^{0.08}}{e^{0.08} + e^{0.92} + e^{0.15}} && = 0.08 && (8 \text{ percent}) \\
\alpha_2 &= \frac{e^{0.92}}{\dots} && = 0.72 && (72 \text{ percent}) \\
\alpha_3 &= \frac{e^{0.15}}{\dots} && = 0.20 && (20 \text{ percent})
\end{aligned}
$$

Step 3: Weighted context

$$
\begin{aligned}
context_1 = 0.08 \cdot h_1^{enc} \\
+ 0.72 \cdot h_2^{enc} \\
+ 0.20 \cdot h_3^{enc}
\end{aligned}
$$

Interpretation:

- 72 percent focus on "cat"
- Makes sense - generating "chat" (French for cat)!
- Model learned alignment



Implementing Attention (Surprisingly Simple)

The complete attention mechanism in code:

```
def attention(decoder_state, encoder_states):
    """
    decoder_state: current decoder hidden state [256]
    encoder_states: list of encoder states, length T
                    each state is [256]
    Returns: context vector [256], attention weights [T]
    """
    scores = []

    for enc_state in encoder_states:
        score = dot(decoder_state, enc_state)
        scores.append(score)

    scores = array(scores)

    exp_scores = exp(scores - max(scores))
    attention_weights = exp_scores / sum(exp_scores)

    context = zeros(256)
    for i, enc_state in enumerate(encoder_states):
        context += attention_weights[i] * enc_state

    return context, attention_weights
```

That's it! Just 3 operations:

1. **Lines 10-12:** Dot products (relevance)
2. **Lines 16-17:** Softmax (probabilities)
3. **Lines 19-21:** Weighted sum (context)

Usage in decoder:

```
for t in range(max_output_length):
    context, attn = attention(
        decoder_state,
        all_encoder_states
    )

    output = decoder_step(
        prev_word,
        decoder_state,
        context
    )
```

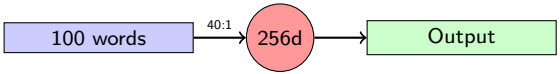
Key difference from before: Context is recomputed at EVERY step, dynamically focusing on relevant inputs!

Solving the Bottleneck: Before vs After

Comparing the two approaches:

WITHOUT Attention:

- Compress entire input to 256d
- Same context for all outputs
- Information loss grows with length



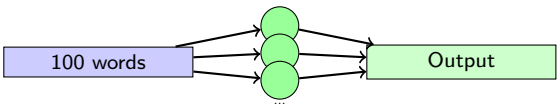
Bottleneck!

Results:

- Short sentences: BLEU 31
- Long sentences: BLEU 8
- 74 percent quality drop!

WITH Attention:

- Keep all encoder states
- Dynamic context each step
- Minimal information loss



All kept!

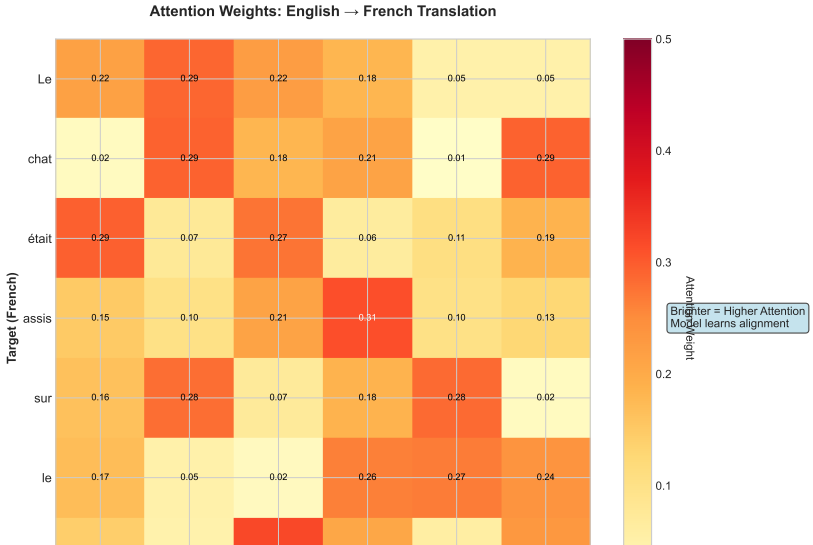
Results:

- Short sentences: BLEU 33 (+2)
- Long sentences: BLEU 24 (+16!)
- Only 27 percent drop

Impact: Attention improves long-sentence quality by **300 percent**!

Visualizing Attention: What the Model Learns

Attention weights reveal learned alignments:



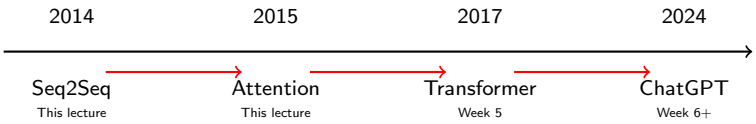
From Seq2Seq to Modern NLP

The foundation you've learned:

Core concepts (now clear to you):

1. **Embeddings:** Words as vectors (numerical meaning)
2. **Hidden states:** Evolving understanding
3. **Encoder-Decoder:** Separate reading from writing
4. **Context vectors:** Compressed representation
5. **Attention:** Selective focus to avoid bottleneck

How this evolved into modern AI:



What's next (Week 5):

- Transformer = "What if we ONLY use attention, remove RNNs entirely?"
- Self-attention = Apply attention to input itself
- This enables: BERT, GPT, ChatGPT, Claude, and all modern LLMs

Summary: The Complete Journey

What you now understand from first principles:

1. **Why embeddings:** Computers need numbers, embeddings give numerical meaning
2. **Why hidden states:** Capture evolving understanding as we process sequences
3. **Why encoder-decoder:** Separate reading (comprehension) from writing (generation)
4. **Why context vectors:** Compress meaning, but creates bottleneck
5. **Why attention:** Solve bottleneck by keeping all states and selectively focusing
6. **Why dot product:** Geometric measure of relevance (vector alignment)

Technical terms you can now explain:

- Word embedding, hidden state, context vector
- Encoder, decoder, seq2seq architecture
- Attention mechanism, attention weights
- Information bottleneck, compression ratio
- Dot product similarity

Next week: Remove the encoder/decoder RNNs, keep only attention = Transformers!