Natural Language Processing

Week 4: The Compression Journey

From Meaning to Numbers and Back Again

Before We Begin: A Thought Experiment

Imagine You're Designing a Translation System

Your Challenge:
Translate this 40-word sentence into French:

"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."

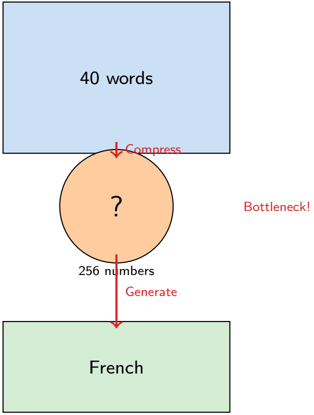
Design Constraints:

1. You can only write down 256 numbers total
2. These numbers must capture ALL the meaning
3. Then translate from just those 256 numbers
4. You cannot look back at the original!

Question: Can 256 numbers really hold 40 words of meaning? What happens to:

- "International" (important detail)
- "premier venues" (significance)
- "researchers around the world" (scale)

Your Design:



Think about it:

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" $\rightarrow$ 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

Checkpoint: Understanding Embeddings

Quick Check: Why do we need embeddings?

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" $\rightarrow$ You know: article, something coming
2. After "The black" $\rightarrow$ You know: a dark-colored thing
3. After "The black cat" $\rightarrow$ You know: a specific animal
4. After full sentence $\rightarrow$ 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

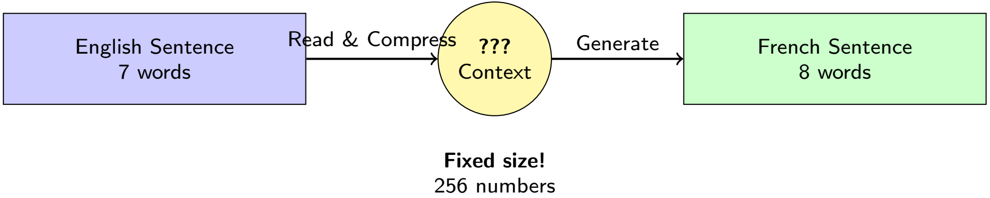
Technical name: When we update understanding word-by-word, we call this a "Recurrent Neural Network" (RNN from Week 3)

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

Intuition: Why Two Networks?

Just like human translation: FIRST fully understand the source sentence (encoder), THEN generate the target (decoder).

Encoder: Building Understanding Step-by-Step

Concrete example: Encoding "The cat sat"

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

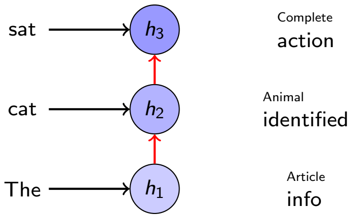
- Convert to embedding: [0.2, 0.5, -0.1, ...] (100d)
- Initial understanding: $h_0 = [0, 0, 0, ...]$ (256d)
- Update: $h_1 = \text{RNN}(\text{"The"}, h_0)$
- New understanding: [0.1, -0.05, 0.03, ...] (256d)

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

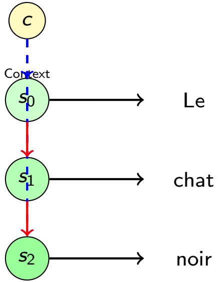
- Input: ¡START¿ token
- Context: $c = h_3$ from encoder (256d)
- Generate: s_0 = RNN(¡START¿, c)
- Predict probabilities: P("Le") = 0.6, P("Un") = 0.3, ...
- **Choose "Le"**

Step 1: Continue

- Input: "Le" (what we just generated)
- Context: still c (same context!)
- Generate: s_1 = RNN("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 = RNN("chat", s_1, c)



Key observations:

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

The First Success: Short Sentences Work Great!

Initial results (5-10 word sentences):

English	French Translation	Result
The cat sat	Le chat s'est assis	Perfect!
I love you	Je t'aime	Perfect!
Hello world	Bonjour le monde	Perfect!
Good morning	Bonjour	Perfect!
See you later	A plus tard	Perfect!

Performance on short sentences:

- 5-10 words: **BLEU 35.2** (excellent quality)
- Captures meaning correctly
- Word order appropriate
- Grammar correct

Breakthrough moment: For the first time, neural networks can translate sentences end-to-end!
No hand-crafted rules, no phrase tables - just learned from data.

Key Question: If it works so well for short sentences, what happens with long ones?

The Failure Pattern Emerges

Testing with longer sentences reveals a problem:

Experimental results (Bahdanau et al., 2014):

Sentence Length	Compression Ratio	BLEU Score	Quality Drop
5-10 words	2:1	35.2	Baseline
10-20 words	5:1	28.5	-19%
20-30 words	10:1	18.7	-47%
30-40 words	15:1	12.4	-65%
40+ words	20:1	8.1	-77%
Pattern: Quality drops as compression ratio increases			

The trend is clear:

- Short sentences (¡ 10 words): Excellent
- Medium sentences (10-20 words): Good
- Long sentences (20-30 words): Poor
- Very long (30+ words): Terrible

The Pattern: Performance degrades predictably with sentence length!
Something systematic is failing as inputs get longer.

Diagnosing the Bottleneck

What gets lost in long sentences? Let's trace it:

Input sentence (42 words):
"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."

Compressed to 256 numbers...

What Survived:

- General topic: ML conference
- Sentiment: Positive
- Main fact: Paper accepted
- Basic structure: Conference does X

Capacity used: 200/256 numbers

What Got Lost:

- "International" modifier
- "premier venues" importance
- "researchers around the world" scale
- Exact conference name
- "submissions" detail

Overflow: 42 words $\rightarrow$ need 420 numbers, only have 256!

Root cause analysis:

- Fixed 256-number container for ANY sentence length
- Information overflow gets discarded
- Network keeps only high-level summary
- Details necessarily lost

The Human Translation Insight

Introspection exercise: How do YOU actually translate?

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

Honest observation:

- 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"
- Writing "sur" → You look back at "on"
- Writing "le" → You look back at "the"
- Writing "tapis" → You look back at "mat"

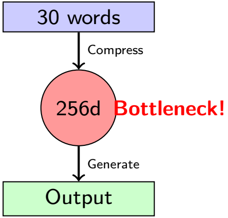
Critical realization:

- 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

Aha Moment: Humans don't compress - they SELECT!

From compression to selection:

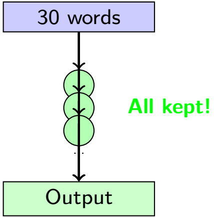
Old Way (Compression):



Problem:

- 30 words $\rightarrow$ 1 vector
- Information loss inevitable
- Same context for all outputs

New Way (Selection):



Solution:

- Keep ALL encoder states
- Select relevant ones per output
- Different context each time

Key Insight: Don't throw away information, just focus on what matters!

Analogy: Instead of summarizing a book, keep the full book and read relevant pages as needed.

Attention = Weighted Relevance (Zero Jargon)

Breaking down "attention" into simple concepts:

Setup:

- 5 source words: [The, black, cat, sat, on]
- Generating French word "chat" (cat)
- Question: How relevant is each source word?

Intuitive relevance scores:

Word	Relevance	Why
The	5%	Generic article
black	15%	Describes cat
cat	70%	Direct match!
sat	5%	Action, not noun
on	5%	Preposition
Total	**100%**	Must sum to 1

What these percentages do:

Context for "chat" = weighted average:

$$
\begin{aligned}
&= 0.05 \times h_{\text{The}} \\
&+ 0.15 \times h_{\text{black}} \\
&+ 0.70 \times h_{\text{cat}} \\
&+ 0.05 \times h_{\text{sat}} \\
&+ 0.05 \times h_{\text{on}}
\end{aligned}
$$

Result: Context is *mostly* "cat" info, with a bit of everything else

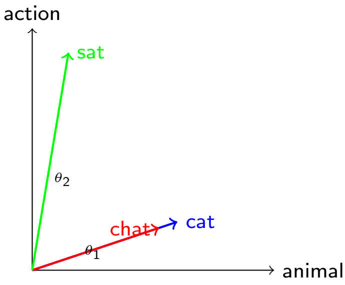
These percentages ARE the attention weights!

Why Dot Product Measures Relevance

Geometric intuition (building from 2D):

Question: How does the network compute those relevance percentages?

Vectors as arrows in space:



Properties of meaning:

- "cat" $=$ [0.8 animal, 0.2 action]
- "chat" $=$ [0.7 animal, 0.2 action]
- "sat" $=$ [0.2 animal, 0.9 action]

Similar meanings $\rightarrow$ similar directions!

Dot product measures alignment:

cat · chat:

$$
\begin{aligned}
&= (0.8 \times 0.7) + (0.2 \times 0.2) \\
&= 0.56 + 0.04 = 0.60
\end{aligned}
$$

cat · sat:

$$
\begin{aligned}
&= (0.8 \times 0.2) + (0.2 \times 0.9) \\
&= 0.16 + 0.18 = 0.34
\end{aligned}
$$

Mathematical property:

- Aligned vectors $\rightarrow$ High value
- Perpendicular $\rightarrow$ Zero
- Opposite $\rightarrow$ Negative

Higher dot product = more relevant!

The 3-Step Attention Mechanism

Now we can understand the full algorithm:

Step 1: Score (Measure Relevance)
For each encoder state, compute dot product with decoder state:

$$score_i = h_{\text{decoder}} \cdot h_i^{\text{encoder}}$$

Why? Dot product = alignment = relevance (from previous slide)

Step 2: Normalize (Make Probabilities)
Convert scores to weights that sum to 100%:

$$\alpha_i = \frac{\exp(score_i)}{\sum_j \exp(score_j)}$$

Why? Need weights for weighted average
Tool: Softmax function (ensures positive, sums to 1)

Step 3: Combine (Weighted Average)
Take weighted sum of encoder states:

$$context = \sum_i \alpha_i \cdot h_i^{\text{encoder}}$$

Why? Focus mostly on relevant, a bit on others

Example weights:

- α_1 (The) = 0.05
- α_2 (black) = 0.15
- α_3 (cat) = 0.70
- α_4 (sat) = 0.05
- α_5 (on) = 0.05

Context is mostly "cat"!

Key Property: Context is *dynamic* - recomputed for each output word with different weights!

Let's trace generating "chat" with actual numbers:

Given:

- Decoder state: $h_{\text{dec}} = [0.5, -0.2, 0.8]$
- Encoder states: $h_1 = [0.1, 0.2, 0.1]$ (The), $h_2 = [0.8, 0.1, 0.7]$ (cat), $h_3 = [0.2, 0.3, 0.2]$ (sat)

Step 1: Compute scores (dot products)

$$\begin{aligned} score_1 &= [0.5, -0.2, 0.8] \cdot [0.1, 0.2, 0.1] \\ &= (0.5)(0.1) + (-0.2)(0.2) + (0.8)(0.1) \\ &= 0.05 - 0.04 + 0.08 = 0.09 \end{aligned}$$

$$\begin{aligned} score_2 &= [0.5, -0.2, 0.8] \cdot [0.8, 0.1, 0.7] \\ &= (0.5)(0.8) + (-0.2)(0.1) + (0.8)(0.7) \\ &= 0.40 - 0.02 + 0.56 = 0.94 \leftarrow \text{Highest!} \end{aligned}$$

$$\begin{aligned} score_3 &= [0.5, -0.2, 0.8] \cdot [0.2, 0.3, 0.2] \\ &= (0.5)(0.2) + (-0.2)(0.3) + (0.8)(0.2) \\ &= 0.10 - 0.06 + 0.16 = 0.20 \end{aligned}$$

Step 2: Softmax to weights

$$\begin{aligned} \alpha_1 &= \frac{e^{0.09}}{e^{0.09} + e^{0.94} + e^{0.20}} \\ &= \frac{1.09}{4.02} = 0.27 \end{aligned}$$

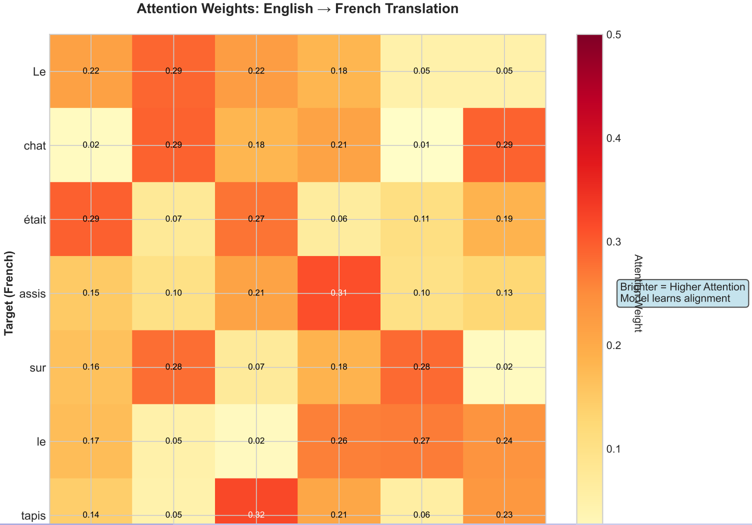
$$\begin{aligned} \alpha_2 &= \frac{e^{0.94}}{4.02} \\ &= \frac{2.56}{4.02} = 0.63 \end{aligned}$$

$$\alpha_3 = \frac{e^{0.20}}{4.02}$$

1.22

Visualizing Attention: The Alignment Matrix

Attention weights reveal what the model is "looking at":



Why Attention Solves the Bottleneck

Information capacity comparison:

Without Attention:

- 30 words compressed to 256d vector
- Capacity: 256 numbers (fixed)
- 30 words = 3000 numbers needed
- Overflow: 2744 numbers lost!
- Same context for all outputs

Information loss:

- 91% of information discarded
- Only high-level summary kept
- Details necessarily lost

With Attention:

- Keep all 30 encoder states
- Capacity: $30 \times 256 = 7680$ numbers
- All information preserved
- Select relevant subset per output
- Dynamic context each time

Information preserved:

- 100% of information available
- Focus on relevant parts
- No forced compression

The Key Insight: Dynamic selection beats static compression!
Instead of "compress everything to 256 numbers", use "keep everything, select as needed"

Attention Results: The Vindication

Performance comparison validates the hypothesis:

Sentence Length	No Attention	With Attention	Improvement
5-10 words	35.2 BLEU	36.1 BLEU	+2.6%
10-20 words	28.5 BLEU	32.7 BLEU	+14.7%
20-30 words	18.7 BLEU	28.9 BLEU	+54.5%
30-40 words	12.4 BLEU	24.8 BLEU	+100%
40+ words	8.1 BLEU	24.3 BLEU	+200%

The pattern:

- Short sentences: Small improvement (bottleneck wasn't the problem)
- Medium sentences: Moderate improvement (bottleneck starts to matter)
- Long sentences: **Massive improvement** (bottleneck was killing performance)

Validation: Attention solves exactly the problem we diagnosed!
Improvement is largest where bottleneck hurt most (long sentences).

Historical Impact: This 2015 paper (Bahdanau et al.) launched the attention revolution in NLP.

Implementing Attention (Surprisingly Simple)

The complete mechanism in code:

```
def attention(decoder_state, encoder_states):
    """
␣␣␣␣decoder_state:␣[256]␣-␣current␣decoder␣hidden␣state
␣␣␣␣encoder_states:␣[seq_len,␣256]␣-␣all␣encoder␣states
␣␣␣␣Returns:␣context␣[256],␣attention_weights␣[seq_len]
␣␣␣␣"""
    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
```

Three operations:

1. **Lines 9-11:** Dot products (relevance scores)
2. **Lines 15-16:** Softmax (probabilities)
3. **Lines 18-20:** Weighted sum (dynamic context)

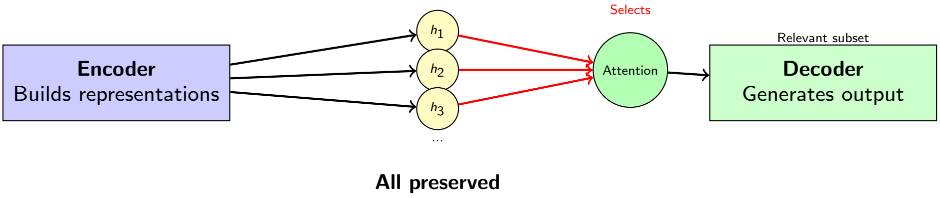
That's it!
Just 3 operations:

- Dot product (similarity)
- Softmax (normalize)
- Weighted sum (combine)

Key difference:
Context recomputed EVERY step with different weights!

Checkpoint: Can you trace what happens when decoder generates "chat" with input "The cat sat"?

Unified architecture diagram:



The three innovations:

1. **Two-stage architecture:** Separate reading (encoder) from writing (decoder)
   - Handles variable-length input and output
   - Mimics human translation process
2. **Sequence-to-sequence:** No fixed input/output size
   - 3 words in → 2 words out (possible!)
   - 100 words in → 50 words out (possible!)
3. **Attention mechanism:** Dynamic selection over static compression
   - Solves information bottleneck
   - Provides interpretability
   - Enables long-sentence translation

What We Learned (Conceptual Insights)

Beyond seq2seq - general lessons:

1. **Compression Trade-off:** Information capacity fundamentally limits performance
   - Can't fit arbitrary information into fixed size
   - Longer inputs → worse compression → lost details
   - Quantifiable: compression ratio predicts quality degradation
2. **Selection ¿ Compression:** For complex tasks, keep everything and select
   - Don't throw away information prematurely
   - Dynamic selection more flexible than static summary
   - "Soft" selection (weighted average) enables gradient flow
3. **Learned Alignment:** Network discovers correspondences without supervision
   - Attention weights show word alignments
   - Model learns which source words matter for each output
   - Interpretable - we can visualize reasoning
4. **Differentiable Operations:** All steps trainable via backpropagation
   - Score, softmax, weighted sum all have gradients
   - End-to-end learning of entire system
   - No hand-crafted alignment rules needed

Beyond Translation: 2024 Applications

The attention explosion across AI:

Language (Original):

- Machine translation (133 languages)
- Text summarization
- Question answering
- Dialogue systems

Vision:

- Image captioning (attend to regions)
- Visual question answering
- Object detection
- Image generation (DALL-E)

Historical timeline:

- 2014: Seq2seq (encoder-decoder)
- 2015: Attention mechanism (this lecture!)
- 2017: Transformers ("Attention is All You Need")
- 2018+: BERT, GPT, current AI revolution

Attention is the foundation of all modern AI systems!

Speech:

- Speech recognition (attend to audio frames)
- Speech synthesis
- Real-time translation

Modern AI:

- **Transformers:** Pure attention (Week 5!)
- GPT-4, Claude, Gemini
- Multimodal models (CLIP)
- All modern LLMs use attention

Summary: The Complete Compression Journey

What you now understand from first principles:

1. **Why embeddings:** Computers need numbers, embeddings give numerical meaning
   - Similar words $\rightarrow$ similar vectors
   - From bytes to meaning
2. **Why hidden states:** Capture evolving understanding as we read
   - Accumulates meaning word-by-word
   - Final state = complete sentence understanding
3. **Why encoder-decoder:** Separate reading from writing
   - Handles variable lengths
   - Mimics human translation
4. **Why context vectors:** Compress meaning, but creates bottleneck
   - Fixed size for any input
   - Information overflow gets lost
5. **Why attention:** Solve bottleneck by keeping all states and selecting
   - Dynamic selection beats compression
   - 200% improvement on long sentences
6. **Why dot product:** Geometric measure of relevance (vector alignment)
   - Similar directions $\rightarrow$ high value
   - Differentiable for training

Next week: Remove encoder/decoder RNNs, use ONLY attention $\rightarrow$ Transformers!

Encoder (RNN Processing):

At each time step $t = 1, 2, \ldots, T_x$:

1. Embedding lookup:

$$x_t = \text{Embed}(w_t) \in \mathbb{R}^{d_{\text{emb}}}$$

2. Encoder hidden state:

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

Explicitly:

$$h_t^{\text{enc}} = \tanh(W_x x_t + W_h h_{t-1}^{\text{enc}} + b_h)$$

where $h_t^{\text{enc}} \in \mathbb{R}^{d_h}$ (typically 256-512d)

3. Context vector:

$$c = h_{T_x}^{\text{enc}}$$

(Final encoder state = compressed meaning)

Encoder Dimensions:

- Vocabulary: $|V| = 10,000$ to $50,000$
- Embedding: $d_{\text{emb}} = 128$ to 512

Decoder (Generation):

Initialize: $s_0 = c$ (context from encoder)

At each generation step $t = 1, 2, \ldots, T_y$:

1. Decoder hidden state:

$$s_t = \text{RNN}_{\text{dec}}(y_{t-1}, s_{t-1}, c)$$

Explicitly:

$$s_t = \tanh(W_y y_{t-1} + W_s s_{t-1} + W_c c + b_s)$$

2. Output distribution:

$$P(y_t \mid y_{<t}, x) = \text{softmax}(W_o s_t + b_o)$$

3. Teacher forcing (training):

$$y_{t-1} = y_{t-1}^* \quad \text{(use true previous word)}$$

4. Loss function:

$$L = -\sum^{T_y} \log P(y_t^* \mid y_{<t}, x)$$

The Attention Computation:

Given encoder hidden states $\{h_1^{\text{enc}}, \ldots, h_{T_x}^{\text{enc}}\}$ and decoder state s_t:

Step 1: Alignment Scores
Compute relevance of each encoder state:

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

Common scoring functions:

Dot product (Luong attention):

$$e_{t,i} = s_t^T h_i^{\text{enc}}$$

Additive (Bahdanau attention):

$$e_{t,i} = v^T \tanh(W_s s_t + W_h h_i^{\text{enc}})$$

Step 2: Attention Weights
Normalize scores to probabilities:

$$\alpha_{t,i} = \frac{\exp(e_{t,i})}{\sum_{j=1}^{T_x} \exp(e_{t,j})}$$

Step 3: Context Vector
Weighted sum of encoder states:

$$c_t = \sum_{i=1}^{T_x} \alpha_{t,i} h_i^{\text{enc}}$$

Key property: Context c_t is DYNAMIC - recomputed at each decoder step with different weights!

Modified Decoder with Attention:

$$s_t = \text{RNN}_{\text{dec}}(y_{t-1}, s_{t-1}, c_t)$$

Why Softmax?

1. **Normalization:** Ensures weights sum to 1
2. **Differentiability:** Smooth function for backprop
3. **Sparsity:** Exponentiation amplifies differences:

$$e_1 = 0.9, e_2 = 0.1 \rightarrow \alpha_1 = 0.64, \alpha_2 = 0.36$$

$$e_1 = 5.0, e_2 = 1.0 \rightarrow \alpha_1 = 0.98, \alpha_2 = 0.02$$

Computational Complexity: