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

Week 4 Overview

Why Can't We Just Use RNNs?

Why This Matters

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

The Fundamental Problem:

What RNNs expect:

- Input: Sequence of length n
- Output: Sequence of length n
- One output per input!

Example that works:

- POS tagging: word $\rightarrow$ tag
- "The cat sat" $\rightarrow$ "DET NOUN VERB"
- 3 inputs $\rightarrow$ 3 outputs ✓

What translation needs:

- Input: Sequence of length n
- Output: Sequence of length m
- $n \neq m$ in general!

Example that fails:

- Translation: English $\rightarrow$ French
- "I love you" $\rightarrow$ "Je t'aime"
- 3 inputs $\rightarrow$ 2 outputs $\times$

Check Your Understanding

Can you think of other tasks where input and output lengths differ?

Real Translation Examples

Let's see why word-by-word translation fails:

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

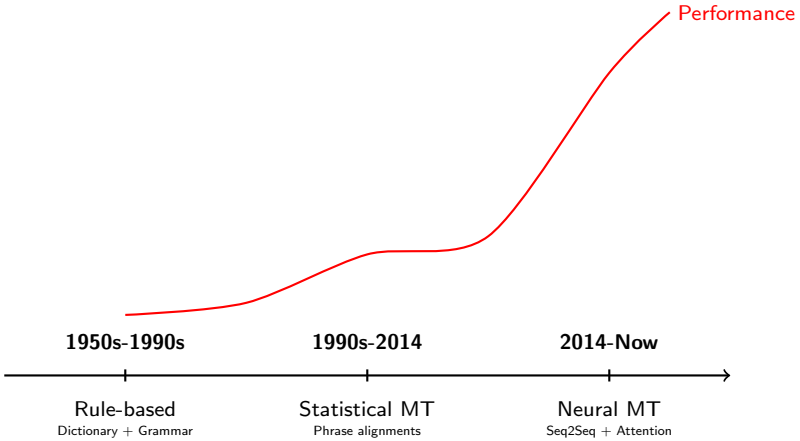
Even worse with longer sentences:

- EN: "The International Conference on Machine Learning" (6 words)
- FR: "La Conférence Internationale sur l'Apprentissage Automatique" (6 words)
- DE: "Die Internationale Konferenz für Maschinelles Lernen" (6 words)
- But word alignment is completely different!

⚠ Common Misconception

Wrong: "We can just pad shorter sequences with zeros!"
Problem: Where do you pad? Beginning? End? How many zeros?
The model has no way to know the target length beforehand!

Evolution of Translation Approaches



The 2014 Breakthrough:

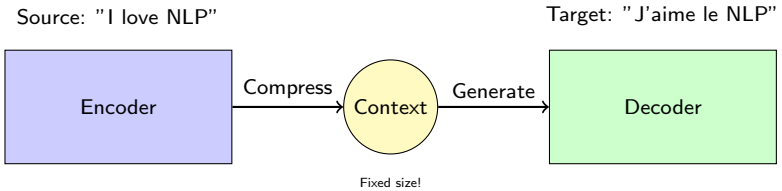
- Sutskever, Vinyals, and Le: "Sequence to Sequence Learning with Neural Networks"
- Key insight: **Separate the reading from the writing!**
- Use two different networks: Encoder and Decoder

The Brilliant Insight

How humans 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 approach (mimics humans):



Check Your Understanding

Why do we need TWO networks instead of one? Think about it!

Building Intuition: The Encoder

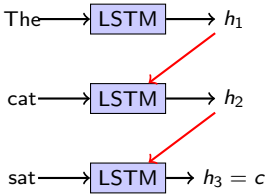
Why This Matters

The encoder's job: Read the input and create an "understanding" of it

Step-by-step encoding process:

What happens at each step:

1. Word → Embedding vector
2. Combine with previous understanding
3. Update the understanding
4. Final state = Complete understanding



Concrete example: "The cat sat"

- "The" → $h_1 = [0.1, -0.2, 0.3]$
- "cat" → $h_2 = [0.5, 0.1, -0.1]$
- "sat" → $h_3 = [0.3, 0.6, 0.2]$
- Context $c = h_3$ (final understanding)

The Mathematics of Encoding

Why This Matters

Now let's formalize what we just saw intuitively

Encoder equations (with explanations):

For each input word x_t at time t:

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

Where:

- x_t = embedding of word at position t (e.g., "cat" $\rightarrow$ [0.2, -0.1, ...])
- h_{t-1}^{enc} = previous hidden state (what we understood so far)
- h_t^{enc} = new hidden state (updated understanding)

After processing all T words:

$c = h_T^{enc}$ (context vector = final understanding)

Dimensions (crucial for implementation!):

- Input embedding: $x_t \in \mathbb{R}^{d_{embed}}$ (typically 100-300)

Encoder Implementation (Simplified)

Why This Matters

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

```python
class Encoder:
    def __init__(self, vocab_size, embed_dim, hidden_dim):
        # Initialize components
        self.embedding = Embedding(vocab_size, embed_dim)
        self.lstm = LSTM(embed_dim, hidden_dim)

    def forward(self, input_sequence):
        # input_sequence: list of word indices [5, 23, 67, ...]

        # Step 1: Convert words to embeddings
        embeddings = [self.embedding(word) for word in input_sequence]

        # Step 2: Process through LSTM
        hidden = zeros(hidden_dim)  # Initial state

        for embed in embeddings:
            hidden = self.lstm(embed, hidden)  # Update understanding

        # Step 3: Return final understanding
        context = hidden  # This is our context vector!
        return context
```

Check Your Understanding

Building Intuition: The Decoder

Why This Matters

The decoder's job: Use the understanding to generate the translation

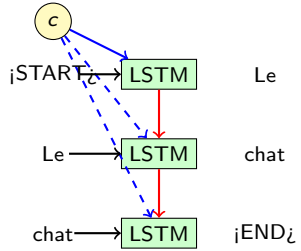
Generation process (step-by-step):

At each generation step:

1. Start with context + previous word
2. Compute new hidden state
3. Predict next word probabilities
4. Sample or take argmax
5. Repeat until ¡END¿ token

Concrete example: Generating "Le chat"

- Start: ¡START¿ + context → "Le"
- Step 1: "Le" + context → "chat"
- Step 2: "chat" + context → ¡END¿
- Output: ["Le", "chat"]



The Mathematics of Decoding

Why This Matters

Let's formalize the generation process

Decoder equations (with explanations):

For each output position t:

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

Where:

- y_{t-1} = previous generated word (or ¡START¿ if $t = 1$)
- h_{t-1}^{dec} = previous decoder hidden state
- c = context vector from encoder (always the same!)

To predict next word:

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

This gives probability for each word in vocabulary!

Common Misconception

Decoder Implementation (Simplified)

```
class Decoder:
    def __init__(self, vocab_size, embed_dim, hidden_dim):
        self.embedding = Embedding(vocab_size, embed_dim)
        self.lstm = LSTM(embed_dim + hidden_dim, hidden_dim)  # Note: larger input!
        self.output_projection = Linear(hidden_dim, vocab_size)

    def forward(self, context, max_length=50):
        outputs = []
        hidden = context  # Initialize with context
        word = START_TOKEN

        for _ in range(max_length):
            # Step 1: Embed previous word
            embed = self.embedding(word)

            # Step 2: Combine with context and process
            lstm_input = concatenate([embed, context])  # Key: use context always!
            hidden = self.lstm(lstm_input, hidden)

            # Step 3: Predict next word
            logits = self.output_projection(hidden)
            probs = softmax(logits)
            word = argmax(probs)  # Or sample from probs

            outputs.append(word)
            if word == END_TOKEN:
                break

        return outputs
```

Training: Teacher Forcing

Why This Matters

How do we train this model? We can't wait for it to generate everything!

The Teacher Forcing trick:

During Training:

- Feed the TRUE previous word
- Not the model's prediction
- This speeds up training dramatically
- Prevents error accumulation

Example: Teaching "Le chat noir"

- Step 1: ¡START¿ → predict "Le"
- Step 2: "Le" (true) → predict "chat"
- Step 3: "chat" (true) → predict "noir"

During Inference:

- Feed the MODEL's previous prediction
- No true words available!
- Errors can accumulate
- Need beam search to help

Example: Generating translation

- Step 1: ¡START¿ → generates "Le"
- Step 2: "Le" (generated) → generates "chat"
- Step 3: "chat" (generated) → generates "noir"

Check Your Understanding

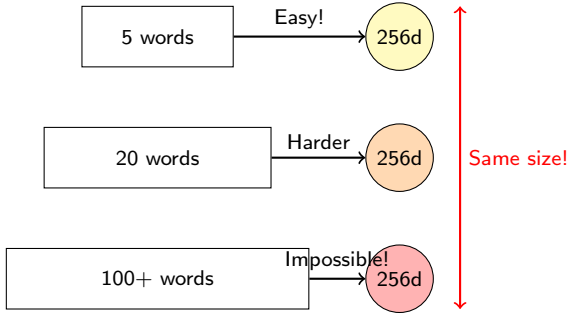
Why might teacher forcing cause problems during deployment?

When Seq2Seq Fails

Why This Matters

Seq2Seq works great for short sentences. But what about long ones?

The Compression Problem:



Information Theory Perspective:

- Each word: $\approx$ 10 bits of information
- 100 words: 1000 bits of information

Visualizing Information Loss

What gets lost in long sequences?

Original paragraph: "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 working on various aspects of learning algorithms, accepted our paper about using neural networks for natural language understanding."

What the context captures:

- ✓General topic (ML conference)
- ✓Sentiment (positive - accepted)
- ×Specific conference name
- ×"researchers around the world"
- ×"neural networks" detail
- ×Word order/grammar structure

⚠ Common Misconception

Wrong: "Just use a bigger context vector!"
Problem: Even 1024d isn't enough for books!

Experimental Evidence (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

The Bottleneck Visualization

[Bottleneck visualization showing information flow]

Key Insights:

1. Early words get "overwritten" by later ones
2. Middle information gets averaged out
3. Final words dominate the context vector
4. Decoder forgets the beginning by the end!

Check Your Understanding

If you were designing a solution, what would you do?
Think: How do humans handle long texts when translating?

The Key Insight: Look Back!

Why This Matters

What if the decoder could look back at ALL encoder states, not just the final one?

Human Translation Process:
When translating "The black cat sat on the mat" $\rightarrow$ "Le chat noir..."

Generating	Looking at
"Le"	Mainly "The"
"chat"	Mainly "cat"
"noir"	Mainly "black"
"s'est assis"	Mainly "sat"
"sur"	Mainly "on"
"le"	Mainly "the"
"tapis"	Mainly "mat"

The Attention Solution:

- Keep ALL encoder hidden states $h_1^{enc}, h_2^{enc}, ..., h_T^{enc}$
- At each decoder step, decide which ones are relevant
- Create a weighted combination based on relevance
- Use this custom context for each word!

How Attention Works (Intuition)

The 3-step attention process:

1. **Score:** How relevant is each encoder state?
   - Current decoder state: h_t^{dec} (what we're generating)
   - Each encoder state: h_i^{enc} (source word i)
   - Score: $e_{ti} = \text{score}(h_t^{dec}, h_i^{enc})$
2. **Normalize:** Convert scores to probabilities
   - Apply softmax: $\alpha_{ti} = \frac{\exp(e_{ti})}{\sum_j \exp(e_{tj})}$
   - Now $\sum_i \alpha_{ti} = 1$ (probability distribution!)
3. **Combine:** Weighted sum of encoder states
   - Context: $c_t = \sum_i \alpha_{ti} \cdot h_i^{enc}$
   - Different context for each decoder step t!

Check Your Understanding

Why do we use softmax instead of just normalizing by sum?
Hint: Softmax makes differences more pronounced!

Attention Mathematics (Step by Step)

Why This Matters

Let's see the exact equations with concrete numbers

Example: Generating "chat" when translating "The cat sat"

Step 1: Score each source word

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

$$e_{2,1} = h_2^{dec} \cdot h_1^{enc} = [0.5,-0.2,0.8] \cdot [0.1,0.2,0.1] = 0.09$$
$$e_{2,2} = h_2^{dec} \cdot h_2^{enc} = [0.5,-0.2,0.8] \cdot [0.8,0.1,0.7] = 0.94$$
$$e_{2,3} = h_2^{dec} \cdot h_3^{enc} = [0.5,-0.2,0.8] \cdot [0.2,0.3,0.2] = 0.20$$

Step 2: Convert to probabilities

$$\alpha_{2,1} = \frac{e^{0.09}}{e^{0.09}+e^{0.94}+e^{0.20}} = \frac{1.09}{4.04} = 0.27$$
$$\alpha_{2,2} = \frac{e^{0.94}}{4.04} = 0.63$$ **Highest attention on "cat"!**
$$\alpha_{2,3} = \frac{e^{0.20}}{4.04} = 0.10$$

Different Attention Mechanisms

Three main ways to compute attention scores:

1. **Dot Product (Luong):**

   $e_{ti} = h_t^{dec} \cdot h_i^{enc}$
   Fast, simple, works well

2. **Scaled Dot Product (Transformer):**

   $e_{ti} = \frac{h_t^{dec} \cdot h_i^{enc}}{\sqrt{d}}$
   Prevents values from getting too large

3. **Additive/Concat (Bahdanau):**

   $e_{ti} = v^T \cdot \tanh(W_1 h_t^{dec} + W_2 h_i^{enc})$
   More parameters, more flexible

Common Misconception

Common confusion: "Attention looks at future words"

Attention Implementation

```python
def attention(decoder_hidden, encoder_outputs):
    """
    decoder_hidden: current decoder state [hidden_dim]
    encoder_outputs: all encoder states [seq_len, hidden_dim]
    """
    # Step 1: Compute scores
    scores = []
    for enc_output in encoder_outputs:
        score = dot_product(decoder_hidden, enc_output)
        scores.append(score)

    # Step 2: Normalize with softmax
    scores = np.array(scores)
    exp_scores = np.exp(scores - np.max(scores))  # Stability trick
    attention_weights = exp_scores / np.sum(exp_scores)

    # Step 3: Compute weighted sum
    context = np.zeros_like(decoder_hidden)
    for weight, enc_output in zip(attention_weights, encoder_outputs):
        context += weight * enc_output

    return context, attention_weights

# Usage in decoder:
context, weights = attention(current_hidden, all_encoder_states)
# Now use context instead of fixed context vector!
```

Visualizing Attention

[Attention heatmap visualization]

What the visualization tells us:

- Diagonal pattern = word-to-word alignment
- Off-diagonal = reordering (common in translation)
- Distributed attention = phrase translation
- Sharp attention = direct word correspondence

Check Your Understanding

Look at "s'est assis" attending to "sat" - why two words for one?

The Impact of Attention

Performance Improvements (Bahdanau et al., 2015):

Model	Short (¡10)	Medium (10-20)	Long (¿20)
Seq2Seq	35.2	28.5	9.7
Seq2Seq + Attention	36.1	34.8	28.3
Improvement	+2.6%	+22.1%	**+192%**

Why attention helps so much:

1. No information bottleneck - access all source information
2. Handles long sequences - doesn't forget the beginning
3. Provides interpretability - see what model focuses on
4. Enables better gradient flow - shorter paths

Why This Matters

This single innovation led directly to Transformers ("Attention is All You Need") and ultimately to GPT, BERT, and modern LLMs!

Complete Seq2Seq Mathematics

Full mathematical formulation:

Encoder:

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

$$c = h_T^{enc} \tag{2}$$

Decoder without Attention:

$$h_t^{dec} = \text{LSTM}^{dec}([E^{dec}(y_{t-1}); c], h_{t-1}^{dec}) \tag{3}$$

$$P(y_t|y_{<t}, X) = \text{softmax}(W_o h_t^{dec} + b_o) \tag{4}$$

Decoder with Attention:

$$e_{ti} = \text{score}(h_t^{dec}, h_i^{enc}) \tag{5}$$

$$\alpha_{ti} = \frac{\exp(e_{ti})}{\sum_{j=1}^{T} \exp(e_{tj})} \tag{6}$$

$$c_t = \sum_{i=1}^{T} \alpha_{ti} h_i^{enc} \tag{7}$$

$$h_t^{dec} = \text{LSTM}^{dec}([E^{dec}(y_{t-1}); c_t], h_{t-1}^{dec}) \tag{8}$$

$$P(y_t|y_{<t}, X) = \text{softmax}(W_o[h_t^{dec}; c_t] + b_o) \tag{9}$$

Training Objectives

Loss Function:

Cross-entropy loss over entire sequence:
$\mathcal{L} = -\sum_{t=1}^{T'} \log P(y_t^* | y_{<t}^*, X)$
Where y^* is the true target sequence

Training Process:

1. Forward pass: Encode source, decode with teacher forcing
2. Compute loss at each time step
3. Backpropagate through entire network
4. Update parameters with optimizer (Adam, SGD)

Key Hyperparameters:

- Hidden dimension: 256-512 (trade-off: capacity vs speed)
- Embedding dimension: 100-300 (semantic richness)
- Vocabulary size: 10K-50K (coverage vs efficiency)
- Dropout rate: 0.2-0.3 (prevent overfitting)
- Learning rate: 0.001 (with decay)

Beam Search Algorithm

Why This Matters

Greedy decoding often produces suboptimal translations. Beam search explores multiple paths!

Algorithm (Pseudocode):

```
# Input: Encoder outputs, beam size k
beams = [(<START>, 0.0)]

while not all_beams_ended:
    new_beams = []
    for (sequence, score) in beams:
        if last_token(sequence) == <END>:
            add_to_completed(sequence, score)
        else:
            probs = decode_step(sequence)
            top_k = get_top_k_tokens(probs, k)
            for token, prob in top_k:
                new_seq = sequence + [token]
                new_score = score + log(prob)
                new_beams.append((new_seq, new_score))

    beams = select_top_k_beams(new_beams, k)

return best_completed_sequence()
```

BLEU Score Computation

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)

Example:

- Reference: "The cat sat on the mat"
- Hypothesis: "The cat is on the mat"
- 1-gram precision: $5/6 = 0.83$ (words)
- 2-gram precision: $2/5 = 0.40$ (bigrams)
- BLEU-2 ≈ 0.57

Check Your Understanding

Why do we use geometric mean (exp of log sum) instead of arithmetic mean?

Seq2Seq in 2024: Where It's Used

[Modern applications grid visualization]

Current Applications:

Text-to-Text:

- Machine Translation (Google Translate)
- Text Summarization
- Question Answering
- Dialogue Systems

Speech:

- Speech Recognition (Whisper)
- Speech Synthesis
- Voice Translation

Code:

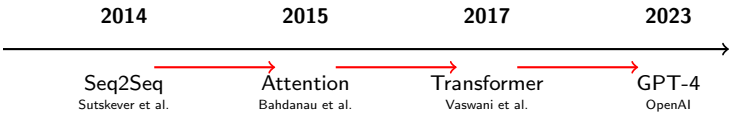
- Code Generation (GitHub Copilot)
- Code Translation
- Bug Fixing
- Documentation Generation

Multimodal:

- Image Captioning
- Video Description
- Visual Question Answering

From Seq2Seq to Transformers

The Evolution Path:



Key Innovations:

- Seq2Seq: Separate encoding and decoding
- +Attention: Solve the bottleneck problem
- Transformer: Attention is ALL you need (no RNNs!)
- GPT/BERT: Pre-training on massive data

Why This Matters

The seq2seq architecture you learned today is the foundation of ALL modern LLMs!

Limitations and Future Directions

Current Limitations of Seq2Seq:

1. **Sequential processing:** Can't parallelize
2. **Fixed vocabulary:** Can't handle new words
3. **No pre-training:** Trains from scratch
4. **Single modality:** Text only (originally)

How Modern Models Address These:

- Transformers: Full parallelization
- Subword tokenization: Handle any word
- Pre-training: Transfer learning
- Multimodal: Vision + Language models

Check Your Understanding

Based on what you learned, what would YOU improve about seq2seq?

Week 4 Summary: Key Takeaways

What We Learned:

1. Variable-length problem
2. Encoder-Decoder solution
3. Information bottleneck
4. Attention mechanism
5. Implementation details

Key Equations:

- Encoding: $h_t = \text{LSTM}(x_t, h_{t-1})$
- Attention: $\alpha_t = \text{softmax}(e_t)$
- Context: $c_t = \sum \alpha_{ti} h_i$

Practical Skills:

- Implement encoder-decoder
- Add attention mechanism
- Use teacher forcing
- Apply beam search
- Evaluate with BLEU

Next Week:

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

Why This Matters

You now understand the foundation of modern NLP! Everything from Google Translate to ChatGPT builds on these concepts.