Natural Language Processing

Week 4: Sequence-to-Sequence Models

Breaking the Fixed-Length Barrier

Week 4 Overview

The Translation Revolution

A Brief History:

- **1950s-1990s**: Rule-based translation
  - Dictionary lookups + grammar rules
  - "The spirit is willing but the flesh is weak"
  - → Russian → English: "The vodka is good but the meat is rotten"
- **1990s-2010s**: Statistical machine translation
  - Phrase-based models
  - Required parallel corpora
- **2014**: **Sequence-to-Sequence revolution**
  - Neural networks learn to translate
  - No rules, just examples!
- **2017-now**: Attention and Transformers
  - Near-human quality
  - Powers Google Translate, DeepL

The Fundamental Problem

Different languages = Different lengths!

English	Translation	Words
I love you	Je t'aime (French)	$3 \rightarrow 2$
I love you	Ich liebe dich (German)	$3 \rightarrow 3$
I love you	Aishiteru (Japanese)	$3 \rightarrow 1$
I love you	Eu te amo (Portuguese)	$3 \rightarrow 3$

The challenge:

- Input length $\neq$ Output length
- Word order changes between languages
- One word can become many (and vice versa)

RNNs produce one output per input - this won't work!

Where Fixed-Length Models Fail

Naive Approach 1: Padding

- Pad all sequences to maximum length
- Problem: Wastes computation, learns to ignore padding

Naive Approach 2: Truncation

- Cut sequences to fixed length
- Problem: Loses critical information!

Naive Approach 3: Sliding Windows

- Process in fixed-size chunks
- Problem: Breaks semantic units

$$\boxed{\text{Fixed-length constraint}} \xrightarrow{\text{Seq2Seq}} \boxed{\text{Variable-length I/O}}$$

Real-World Variable-Length Tasks

Where we need flexible input/output:

Translation

- Any language pair
- Technical documents
- Real-time conversation

Summarization

- Article → headline
- Book → abstract
- Meeting → minutes

Dialog Systems

- Question → answer
- Chat → response
- Command → action

Code Generation

- Comment → code
- Spec → implementation
- Bug description → fix

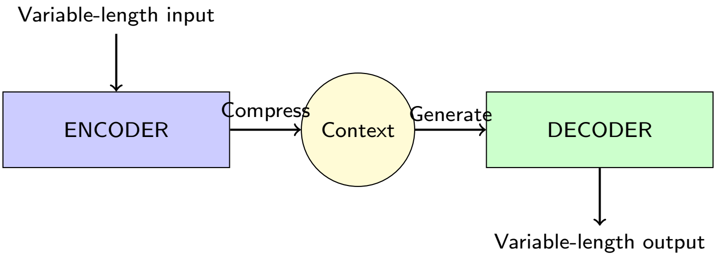
All these tasks have variable-length inputs AND outputs!

The Key Insight: Two-Stage Processing

How humans translate:

1. **Read and understand** the entire source sentence
2. **Generate** the translation based on understanding

The Seq2Seq insight:



Key: Fixed-size context vector bridges variable-length sequences!

Evolution: Predicting the Next Word

How each approach handles "next word prediction":

Method	Context	Method	Output
N-gram	Fixed n words	Count	Single word
RNN	All previous	Hidden state	Single word
Seq2Seq	**Entire input**	**Encode-decode**	**Full sequence**

Example: "How are you?" $\rightarrow$ "Comment allez-vous?"

- **N-gram**: Would need to see exact phrase before
- **RNN**: Produces "How" $\rightarrow$ "Comment", "are" $\rightarrow$?, stuck!
- **Seq2Seq**: Reads full input, generates full output

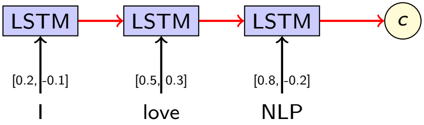
Seq2Seq doesn't just predict next word - it generates entire sequences!

The Encoder: Compressing Meaning

What the encoder does:
Process input sequence $\rightarrow$ Create fixed-size representation

$$h_t^{enc} = \text{LSTM}(x_t, h_{t-1}^{enc}) \quad \text{Process each word}$$
$$c = h_T^{enc} \quad \text{Final hidden state = context}$$

Step-by-step example: "I love NLP"



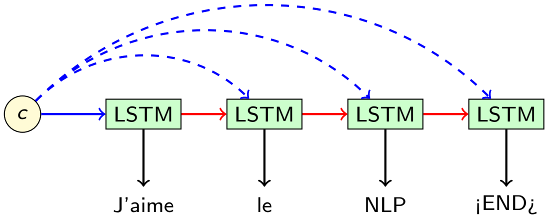
Context vector c captures the meaning of entire input!

The Decoder: Generating from Context

What the decoder does:
Use context vector $\rightarrow$ Generate output sequence

$$h_t^{dec} = \text{LSTM}(y_{t-1}, h_{t-1}^{dec}, c) \quad \text{Generate each word}$$
$$y_t = \text{softmax}(W \cdot h_t^{dec}) \quad \text{Predict next word}$$

Generation process: Context $\rightarrow$ "J'aime le NLP"



Complete Seq2Seq Implementation

Minimal working seq2seq in 20 lines:

```
class Seq2Seq:
    def __init__(self, vocab_size, hidden_size):
        self.encoder = LSTM(vocab_size, hidden_size)
        self.decoder = LSTM(vocab_size, hidden_size)
        self.output_proj = Linear(hidden_size, vocab_size)

    def encode(self, source_sequence):
        h = zeros(hidden_size)
        for word in source_sequence:
            h, _ = self.encoder(embed(word), h)
        return h  # This is our context vector

    def decode(self, context, max_length=50):
        h = context
        word = START_TOKEN
        output = []
        for _ in range(max_length):
            h, _ = self.decoder(embed(word), h)
            word = softmax(self.output_proj(h))
            output.append(word)
            if word == END_TOKEN: break
        return output
```

Teacher Forcing: Training Trick

Problem: Early in training, decoder makes mistakes → compounds errors

Solution: During training, feed correct previous word (not predicted)

Without Teacher Forcing:

- Target: "J'aime le NLP"
- Predicts: "Je"
- Next input: "Je" (wrong!)
- Predicts: "suis" (more wrong!)
- Cascade of errors...

With Teacher Forcing:

- Target: "J'aime le NLP"
- Predicts: "Je" (wrong)
- Next input: "J'aime" (correct!)
- Predicts: "le" (learning!)
- Faster convergence

At test time: Use predicted words (no teacher forcing)

Numerical Example: Translation Step-by-Step

Translating: "cat" → "chat"

Encoding:

- Input embedding: "cat" → [0.3, -0.2, 0.8, 0.1]
- Encoder LSTM: [0.3, -0.2, 0.8, 0.1] → context [0.5, 0.1, -0.3, 0.7]

Decoding:

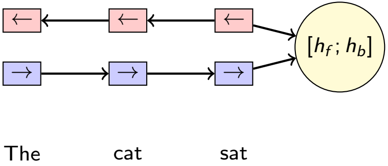
1. Start with context: [0.5, 0.1, -0.3, 0.7]
2. Decoder LSTM step 1:
   - Input: ¡START¿ + context
   - Hidden: [0.4, 0.2, -0.1, 0.6]
   - Output scores: {chat: 2.3, chien: 0.8, maison: 0.2, ...}
   - Softmax: {chat: 0.73, chien: 0.15, maison: 0.03, ...}
   - Select: "chat"
3. Decoder LSTM step 2:
   - Input: "chat" + hidden
   - Output: ¡END¿ token

Result: "cat" successfully translated to "chat"!

Bidirectional Encoders: Looking Both Ways

Problem: Forward LSTM only sees past context

Solution: Process sequence in both directions!

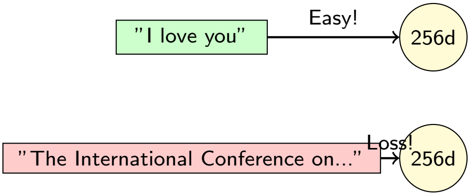


Benefits:

- "cat" knows about "sat" (future context)
- Better representation of each word
- Standard in modern seq2seq

When Context Vectors Fail

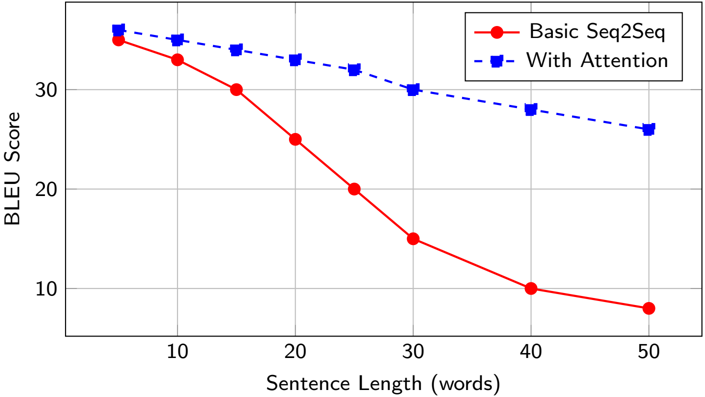
The bottleneck: Compressing everything into fixed-size vector



Information theory perspective:

- 256-dimensional vector = 1KB of information
- Long document = 100KB of information
- **Cannot compress 100KB $\rightarrow$ 1KB without loss!**

Performance vs. Sentence Length:



Performance degrades drastically after 15-20 words!

Real Examples of Bottleneck Failures

Legal Document Translation:

Input (50 words): "The party of the first part hereby agrees to indemnify and hold harmless the party of the second part from any and all claims, damages, losses, and expenses, including but not limited to reasonable attorney's fees, arising from..."
Output: "La partie accepte de payer." (The party agrees to pay.)
Lost: Who indemnifies whom, what claims, legal details!

Technical Manual Translation:

Input (35 words): "To replace the filter, first disconnect power, then remove the four screws on the top panel, lift carefully to avoid damaging the sensor wire, and locate the filter housing behind the main unit."
Output: "Remplacer le filtre." (Replace the filter.)
Lost: All safety steps and detailed instructions!

Why Fixed Context Fails: Memory Analogy

Human memory analogy:

Imagine memorizing a book by reading it once and storing it in your mind as a single "feeling"

Short story (5 pages):

- Can remember plot
- Can recall characters
- Can retell accurately

Novel (500 pages):

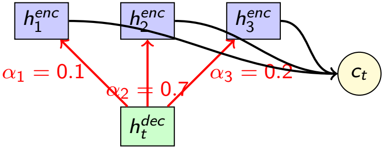
- Forget early chapters
- Mix up characters
- Only remember gist

Humans don't compress books into single thoughts -
we refer back to specific parts when needed!

This insight leads to the attention mechanism...

The Attention Revolution (Bahdanau et al., 2015)

The key insight: Don't compress everything - look back at what you need!



How attention works:

1. Keep ALL encoder hidden states (not just last)
2. At each decoder step, compute relevance scores
3. Create weighted sum of encoder states
4. Use this dynamic context for generation

Attention Mathematics Made Simple

Three steps to attention:

1. Score: How relevant is each encoder state?

$$score(h_t^{dec}, h_i^{enc}) = h_t^{dec} \cdot h_i^{enc} \quad \text{(dot product)}$$

2. Normalize: Convert scores to probabilities

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

3. Combine: Weighted sum of encoder states

$$c_t = \sum_i \alpha_i \cdot h_i^{enc}$$

Intuition: The decoder "asks" each encoder state:
"How much should I pay attention to you right now?"

Attention in Action: Translation Example

Translating: "The cat sat" → "Le chat s'est assis"

Generating	Attention Weights		
	The	cat	sat
Le	0.7	0.2	0.1
chat	0.1	0.8	0.1
s'est	0.1	0.1	0.8
assis	0.1	0.2	0.7

Observations:

- "Le" attends to "The" (articles align)
- "chat" attends to "cat" (nouns align)
- "s'est assis" attends to "sat" (verbs align)
- Model learns alignment without explicit rules!

Implementing Attention

```python
def attention(decoder_hidden, encoder_outputs):
    """
    decoder_hidden: current decoder state [hidden_size]
    encoder_outputs: all encoder states [seq_len, hidden_size]
    """
    # Step 1: Calculate scores
    scores = torch.dot(decoder_hidden, encoder_outputs.T)
    # scores shape: [seq_len]

    # Step 2: Normalize with softmax
    attention_weights = torch.softmax(scores, dim=0)
    # attention_weights shape: [seq_len], sum to 1

    # Step 3: Weighted sum of encoder outputs
    context = torch.sum(
        attention_weights.unsqueeze(1) * encoder_outputs,
        dim=0
    )
    # context shape: [hidden_size]

    return context, attention_weights

# Example usage:
# context, weights = attention(decoder_h, all_encoder_h)
# decoder_h_new = LSTM(input, decoder_h, context)
```

Types of Attention

Different ways to compute attention scores:

1. **Dot Product (Luong):**
    - $score = h_t^{dec} \cdot h_i^{enc}$
    - Fast, no parameters
    - Requires same dimensionality
2. **Additive (Bahdanau):**
    - $score = v^T \tanh(W_1 h_t^{dec} + W_2 h_i^{enc})$
    - More flexible
    - Can handle different dimensions
3. **Multiplicative:**
    - $score = h_t^{dec} W h_i^{enc}$
    - Learnable weight matrix
    - Balance of flexibility and speed

All types learn to align source and target automatically!

Attention Visualization: What Model Sees

Real attention heatmap from translation:

[Attention heatmap visualization will be generated]

Key insights:

- Diagonal pattern = word alignment
- Off-diagonal = reordering between languages
- Brightness = attention strength
- Model learns linguistic structure!

Performance Impact of Attention

BLEU scores on WMT'14 English-French:

Model	BLEU Score	Improvement
Basic Seq2Seq	25.3	baseline
+ Bidirectional encoder	27.1	+1.8
+ Attention	31.7	+6.4
+ Beam search	33.2	+7.9
Google Translate (2016)	38.9	+13.6
Transformer (2017)	41.8	+16.5

Attention gives 25% relative improvement!
This breakthrough enabled near-human translation quality.

Summary: Breaking the Fixed-Length Barrier

What we learned:

1. **Variable-length challenge:**
    - Different languages have different lengths
    - Fixed-size models fail
2. **Encoder-Decoder architecture:**
    - Separate compression from generation
    - Enables variable input/output
3. **Information bottleneck:**
    - Fixed context loses information
    - Performance degrades with length
4. **Attention mechanism:**
    - Look back at all encoder states
    - Dynamic, focused context
    - Dramatic performance improvement

Seq2Seq + Attention = Foundation for modern NLP!
(Transformers are attention taken to the extreme)

Complete Encoder Equations

Bidirectional LSTM Encoder:

Forward pass:

$$\vec{h}_t = \text{LSTM}_{\text{forward}}(x_t, \vec{h}_{t-1}) \tag{1}$$

$$\vec{h}_t \in \mathbb{R}^d \tag{2}$$

Backward pass:

$$\overleftarrow{h}_t = \text{LSTM}_{\text{backward}}(x_t, \overleftarrow{h}_{t+1}) \tag{3}$$

$$\overleftarrow{h}_t \in \mathbb{R}^d \tag{4}$$

Concatenation:

$$h_t^{enc} = [\vec{h}_t; \overleftarrow{h}_t] \tag{5}$$

$$h_t^{enc} \in \mathbb{R}^{2d} \tag{6}$$

Dimensions:

- Input embedding: $x_t \in \mathbb{R}^e$ (e = embedding size)
- Hidden states: $h_t \in \mathbb{R}^d$ (d = hidden size)
- Final encoder states: $H^{enc} \in \mathbb{R}^{T \times 2d}$ (T = sequence length)

Decoder with Attention: Full Equations

At each decoder timestep t:

1. Attention scores:

$$\begin{aligned} e_{ti} &= \text{score}(s_{t-1}, h_i^{enc}) && (7) \\ &= v^T \tanh(W_a s_{t-1} + U_a h_i^{enc}) && (8) \end{aligned}$$

2. Attention weights:

$$\alpha_{ti} = \frac{\exp(e_{ti})}{\sum_{j=1}^{T} \exp(e_{tj})} \tag{9}$$

3. Context vector:

$$c_t = \sum_{i=1}^{T} \alpha_{ti} h_i^{enc} \tag{10}$$

4. Decoder update:

$$\begin{aligned} s_t &= \text{LSTM}([y_{t-1}; c_t], s_{t-1}) && (11) \\ p(y_t | y_{<t}, x) &= \text{softmax}(W_o [s_t; c_t]) && (12) \end{aligned}$$

Beam Search Algorithm

Problem: Greedy decoding may not find best sequence

Solution: Keep top-K hypotheses at each step

Input: Context c, Beam size K
Output: Best sequence

```
beams = [([], 0)] // (sequence, score)
for t = 1 to T_max do
    new_beams = []
    foreach (seq, score) in beams do
        foreach word in vocabulary do
            new_score = score + log P(word—seq, c)
            new_beams.append((seq + [word], new_score))
        end
    end
    beams = top_K(new_beams, K)
    if all beams end with ¡END¿ then
        break
    end
end
return best beam
```

BLEU Score Calculation

BLEU: Bilingual Evaluation Understudy

Measures n-gram overlap between prediction and reference:

$$\text{BLEU} = BP \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right) \tag{13}$$

Where:

- p_n = precision of n-grams
- w_n = weights (usually $1/N$)
- BP = brevity penalty (penalizes short translations)

Example:

- Reference: ”The cat sat on the mat”
- Prediction: ”The cat sat on a mat”
- Unigram matches: $6/6 = 1.0$
- Bigram matches: $4/5 = 0.8$
- BLEU-2 ≈ 0.89

Interpretation:

- BLEU ¡ 10: Almost useless
- BLEU 10-20: Understandable
- BLEU 20-30: Good quality
- BLEU ¿ 30: High quality

Seq2Seq in Production Systems

1. **Machine Translation:**
    - **Google Translate:** 100+ languages
    - **DeepL:** Higher quality for European languages
    - **Facebook:** Real-time translation in comments
2. **Conversational AI:**
    - **Customer service:** Automated support chatbots
    - **Virtual assistants:** Siri, Alexa understanding
    - **Therapy bots:** Mental health support
3. **Code Generation:**
    - **GitHub Copilot:** Comment $\rightarrow$ code
    - **Tabnine:** Autocomplete entire functions
    - **CodeT5:** Bug description $\rightarrow$ fix
4. **Content Creation:**
    - **Summarization:** Article $\rightarrow$ headline
    - **Paraphrasing:** Rewrite for clarity
    - **Style transfer:** Formal $\leftrightarrow$ casual

Multimodal Seq2Seq Applications

Beyond text-to-text:

Image Captioning:

- CNN encoder (image)
- LSTM decoder (text)
- "A cat sitting on a mat"

Video Description:

- 3D CNN encoder
- Attention over frames
- Action recognition

Speech Recognition:

- Audio encoder
- Text decoder
- Whisper, wav2vec2

Music Generation:

- Text description $\rightarrow$ audio
- Style transfer
- MuseNet, Jukebox

Seq2Seq is the foundation for any sequence transformation task!

From Seq2Seq to Transformers

Evolution timeline:

- **2014:** Basic Seq2Seq (Sutskever et al.)
  - First neural translation
  - Fixed context bottleneck
- **2015:** Attention mechanism (Bahdanau et al.)
  - Dynamic context
  - Alignment learning
- **2017:** Transformer (Vaswani et al.)
  - "Attention is all you need"
  - No recurrence, pure attention
  - Parallel processing
- **2018-now:** Pre-trained models
  - BERT, GPT, T5
  - Transfer learning
  - Few-shot learning

Next week: We'll explore Transformers - seq2seq on steroids!