Week 4: Sequence-to-Sequence Models

From Variable Lengths to Attention Mechanisms

BSc-Level Enhanced Version with Layout Templates

Today's Journey

Where We Are:

- Week 3: RNNs handle sequences
- Week 3: Fixed input → fixed output
- **Today: Variable input → variable output**

Today's Learning Objectives:

1. Understand why translation needs special architecture
2. Master encoder-decoder framework
3. Discover attention mechanism
4. Apply seq2seq in practice

Prerequisite

You should know:

- RNN basics
- Hidden states
- Backpropagation

Why Can't We Just Use RNNs?

Build Your Intuition

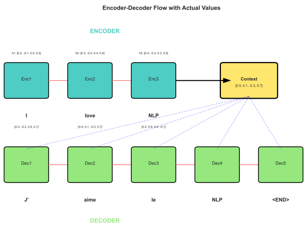
RNNs expect: one input → one output
Translation needs: any length → any length

The Fundamental Problem:

- English: "I love you" (3 words)
- French: "Je t'aime" (2 words)
- Japanese: "Aishiteru" (1 word)

RNN Limitation:

- Fixed mapping between positions
- Can't handle length mismatch
- No way to "wait" or "generate multiple"



Common Misconception

"Just pad with zeros!"
No - that changes meaning and wastes computation.

Concrete Example: "Hello World" Translation

Let's trace through actual numbers:

Input: "Hello world" (English)

- Word 1: "Hello" $\rightarrow$ embed = [0.2, -0.1, 0.5]
- Word 2: "world" $\rightarrow$ embed = [0.3, 0.4, -0.2]

Target: "Bonjour monde" (French)

- Word 1: "Bonjour"
- Word 2: "monde"

Try It

With standard RNN:

- Position 1: Hello $\rightarrow$ Bonjour ✓
- Position 2: world $\rightarrow$ monde ✓
- Position 3: ??? $\rightarrow$ Nothing needed

But what if French had 3 words?
The system breaks!

Checkpoint

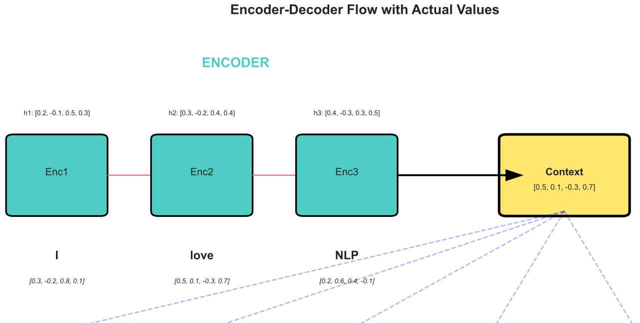
Can you explain why padding doesn't solve this?
Answer: Because we don't know output length in advance!

The Brilliant Insight: Two-Stage Process

Build Your Intuition

Think like a human translator:

1. READ and understand the whole sentence
2. THINK about the meaning
3. WRITE the translation



Encoder: Step-by-Step with Numbers

Processing "The cat sat":

Step 1: Embedding

- "The" $\rightarrow x_1 = [0.1, -0.2, 0.3]$
- "cat" $\rightarrow x_2 = [0.5, 0.1, -0.1]$
- "sat" $\rightarrow x_3 = [0.3, 0.6, 0.2]$

Step 2: LSTM Processing

$$h_0 = [0, 0, 0] \text{ (initial)}$$
$$h_1 = \text{LSTM}(x_1, h_0) = [0.2, -0.1, 0.1]$$
$$h_2 = \text{LSTM}(x_2, h_1) = [0.4, 0.3, -0.2]$$
$$h_3 = \text{LSTM}(x_3, h_2) = [0.6, 0.5, 0.3]$$

Step 3: Context Vector

$$c = h_3 = [0.6, 0.5, 0.3]$$

Try It

Notice how each hidden state builds on the previous:

- h_1: knows "The"
- h_2: knows "The cat"
- h_3: knows "The cat sat"

Final h_3 contains the complete meaning!

Encoder Implementation

```python
class Encoder(nn.Module):
    def __init__(self, vocab_size,
                 embed_dim=100,
                 hidden_dim=256):
        super().__init__()
        self.embedding = nn.Embedding(
            vocab_size, embed_dim)
        self.lstm = nn.LSTM(
            embed_dim, hidden_dim)

    def forward(self, source):
        # source: [seq_len, batch]
        embedded = self.embedding(source)
        # embedded: [seq_len, batch, embed]

        outputs, (hidden, cell) = \
            self.lstm(embedded)
        # hidden: [1, batch, hidden_dim]

        return hidden, cell  # Context!
```

Key Implementation Points:

- `vocab_size`: How many words we know
- `embed_dim`: Word vector size (100)
- `hidden_dim`: Context size (256)

Dimensions Example:

- Input: "Hello world" = [2, 1]
- After embedding: [2, 1, 100]
- Context output: [1, 1, 256]

Checkpoint

The context vector is always the same size (256) regardless of input length!

Decoder: Generating Step by Step

Generating "Le chat" from context $c = [0.6, 0.5, 0.3]$:

Step 1: Initialize with context

- $h_0^{dec} = c = [0.6, 0.5, 0.3]$
- Input: ¡START¿ token

Step 2: Generate first word

- $h_1^{dec} = \text{LSTM}(< START >, h_0^{dec})$
- Output probabilities: {Le: 0.7, La: 0.2, Un: 0.1}
- Select: "Le"

Step 3: Generate second word

- $h_2^{dec} = \text{LSTM}(``Le'', h_1^{dec})$
- Output probabilities: {chat: 0.8, chien: 0.2}
- Select: "chat"

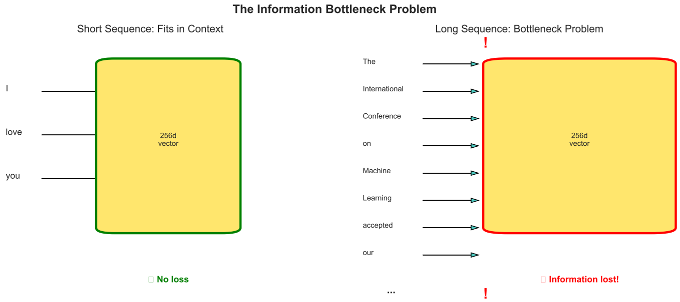
```python
# Decoder forward pass
def forward(self, context):
    hidden = context
    outputs = []

    word = "<START>"
    for _ in range(max_len):
        embed = embedding(word)
        hidden = lstm(embed, hidden)
        probs = softmax(hidden)
        word = sample(probs)

        if word == "<END>":
            break
        outputs.append(word)

    return outputs
```

The Information Bottleneck



The Problem:

- 10-word sentence: ≈ 100 bits of info
- 256-dim vector: 256 numbers capacity
- 50-word sentence: ≈ 500 bits of info
- Same 256-dim vector! Overflow!

Common Misconception

"Just make the vector bigger!"
Problems with that:

- More parameters to learn
- Slower training
- Still fixed size

Attention: The Solution

Key Insight: Look back at ALL encoder states, not just the last one!

Generating "chat" (cat in French):

Step 1: Score each encoder state

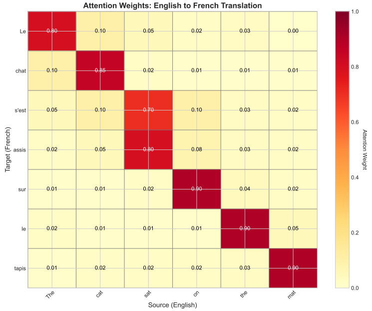
- $e_1 = \text{score}(h_{chat}^{dec}, h_{The}^{enc}) = 0.1$
- $e_2 = \text{score}(h_{chat}^{dec}, h_{cat}^{enc}) = 0.9$
- $e_3 = \text{score}(h_{chat}^{dec}, h_{sat}^{enc}) = 0.3$

Step 2: Normalize to probabilities

- $\alpha_1 = \text{softmax}(0.1) = 0.15$
- $\alpha_2 = \text{softmax}(0.9) = 0.70$
- $\alpha_3 = \text{softmax}(0.3) = 0.15$

Step 3: Weighted combination

$$c_{chat} = 0.15 \cdot h_1 + 0.70 \cdot h_2 + 0.15 \cdot h_3$$



Try It

Attention weights tell us:

- 70% focus on "cat"
- 15% on "The"
- 15% on "sat"

Attention Implementation

```python
class BahdanauAttention(nn.Module):
    def __init__(self, hidden_dim):
        super().__init__()
        self.W1 = nn.Linear(hidden_dim, hidden_dim)
        self.W2 = nn.Linear(hidden_dim, hidden_dim)
        self.V = nn.Linear(hidden_dim, 1)

    def forward(self, decoder_hidden,
                encoder_outputs):
        # decoder_hidden: [batch, hidden]
        # encoder_outputs: [seq_len, batch, hidden]

        # Score each encoder output
        scores = self.V(torch.tanh(
            self.W1(decoder_hidden) +
            self.W2(encoder_outputs)))
        # scores: [seq_len, batch, 1]

        # Convert to probabilities
        weights = F.softmax(scores, dim=0)

        # Weighted sum
        context = torch.sum(
            weights * encoder_outputs, dim=0)

        return context, weights
```

Numerical Example:

Given:

- Decoder hidden: [0.5, 0.3]
- Encoder outputs:
  - h_1: [0.1, 0.2]
  - h_2: [0.7, 0.4]
  - h_3: [0.3, 0.6]

Computation:

- Scores: [0.2, 0.8, 0.3]
- Weights: [0.15, 0.70, 0.15]
- Context: $0.15 \times [0.1, 0.2] + 0.70 \times [0.7, 0.4] + 0.15 \times [0.3, 0.6] = [0.55, 0.46]$

Checkpoint

Attention gives different context for each output word!

Training: Teacher Forcing

Problem: Early in training, model makes mistakes that compound.

Without Teacher Forcing:

- Target: "Le chat dort"
- Step 1: Generate "La" (wrong!)
- Step 2: Based on "La", generate "maison"
- Step 3: Based on "maison", generate "est"
- Result: "La maison est" (completely wrong!)

With Teacher Forcing:

- Target: "Le chat dort"
- Step 1: Generate "La", but feed "Le" to next
- Step 2: Based on "Le", generate "chat" ✓
- Step 3: Based on "chat", generate "dort" ✓
- Learning signal much stronger!

Try It

Teacher forcing schedule:

- Epoch 1-10: 100% forcing
- Epoch 11-20: 50% forcing
- Epoch 21+: 0% forcing

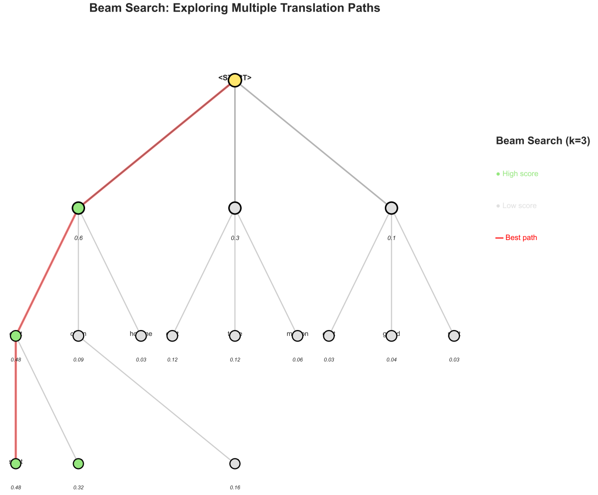
Gradually let the model learn to trust itself!

Common Misconception

"Always use teacher forcing!" No - creates mismatch between training and inference.

Inference: Beam Search

Finding the best translation with beam size = 2:



Evaluation: BLEU Score Calculation

Evaluating: "The cat sits on mat"

Reference: "Le chat est sur le tapis"
Generated: "Le chat sur tapis"

Step 1: Count n-grams

- 1-grams: Le(1/1), chat(1/1), sur(1/1), tapis(1/1)
- 2-grams: "Le chat"(1/1), "chat sur"(0/1), "sur tapis"(0/1)
- 3-grams: None match

Step 2: Calculate precision

- $P_1 = 4/4 = 1.00$
- $P_2 = 1/3 = 0.33$
- $P_3 = 0/2 = 0.00$

Step 3: Brevity penalty

- Generated length: 4
- Reference length: 6
- $BP = e^{1-6/4} = e^{-0.5} = 0.61$

Step 4: Final BLEU

$$BLEU = BP \times \sqrt[3]{P_1 \times P_2 \times P_3}$$

$$= 0.61 \times \sqrt[3]{1.0 \times 0.33 \times 0.01}$$

$$= 0.61 \times 0.22 = 0.13$$

Try It

BLEU scores:

- < 0.1: Useless
- $0.1 - 0.3$: Understandable
- $0.3 - 0.5$: Good
- > 0.5: Very good

Summary: Key Takeaways

What We Learned:

1. **Problem:** Variable length sequences
2. **Solution:** Encoder-Decoder architecture
3. **Enhancement:** Attention mechanism
4. **Training:** Teacher forcing
5. **Inference:** Beam search
6. **Evaluation:** BLEU score

Key Insights:

- Context vector = compressed understanding
- Attention = dynamic focus on relevant parts
- Trade-off between exploration and exploitation

Next Week:

Transformers - Attention is all you need!

- No more recurrence
- Parallel processing
- Self-attention
- Much faster training

Checkpoint

Final check: Can you explain why we need TWO networks?
Answer: Input length $\neq$ output length!

Complete Mathematical Formulation

Encoder:

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

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

Decoder (without attention):

$$h_t^{dec} = f_{dec}(y_{t-1}, h_{t-1}^{dec}, c) \tag{3}$$

$$p(y_t|y_{<t}, x) = \text{softmax}(W_o h_t^{dec}) \tag{4}$$

Decoder (with attention):

$$e_{ti} = a(h_{t-1}^{dec}, h_i^{enc}) \tag{5}$$

$$\alpha_{ti} = \frac{\exp(e_{ti})}{\sum_j \exp(e_{tj})} \tag{6}$$

$$c_t = \sum_i \alpha_{ti} h_i^{enc} \tag{7}$$

$$h_t^{dec} = f_{dec}(y_{t-1}, h_{t-1}^{dec}, c_t) \tag{8}$$

Where $a(\cdot)$ is the attention scoring function.

Seq2Seq in 2024

Still Using Seq2Seq:

- Machine Translation (Google Translate)
- Speech Recognition (Whisper)
- Summarization
- Code Generation (GitHub Copilot)

Evolution to Transformers:

- BERT: Encoder-only
- GPT: Decoder-only
- T5: Full encoder-decoder

Key Improvements:

- Self-attention (next week!)
- Multi-head attention
- Positional encoding
- Layer normalization

Performance Gains:

- 2014 Seq2Seq: BLEU 20
- 2017 Transformer: BLEU 28
- 2024 GPT-4: BLEU 45+