Natural Language Processing Course

Week 4: Sequence-to-Sequence Models

Breaking the Fixed-Length Barrier

NLP Course 2025

Week 4: What You'll Master Today

By the end of this week, you will:

- **Understand** why ChatGPT's predecessors needed variable-length processing
- **Build** intuition for encoder-decoder architecture (used in Google Translate)
- **Discover** the attention mechanism that powers modern AI
- **Implement** a complete seq2seq model from scratch
- **Connect** these concepts to today's transformer-based systems

Core Breakthrough: Input and output sequences can have different lengths!

This innovation (2014-2016) laid the foundation for everything from Google Translate to GitHub Copilot

Why This Matters: Your Daily AI Interactions (2024)

Translation & Communication:

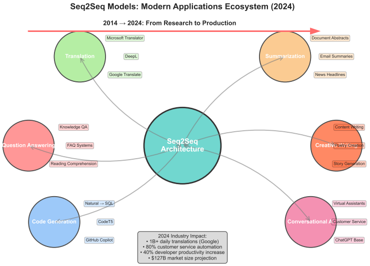
- Google Translate: 1+ billion translations daily
- DeepL: 1 billion people served monthly
- Real-time conversation translation

Code & Development:

- GitHub Copilot: 40% faster development
- Natural language → SQL queries
- Comment → complete function generation

Content & Communication:

- Email summarization (Outlook, Gmail)
- Meeting transcript → action items
- Customer service automation (80% first-line)



The Variable-Length Challenge

The Core Problem:

Different languages express ideas with different lengths:

- English: "I love you" (3 words)
- French: "Je t'aime" (2 words)
- German: "Ich liebe dich" (3 words)
- Japanese: "aishiteru" (1 compound word)

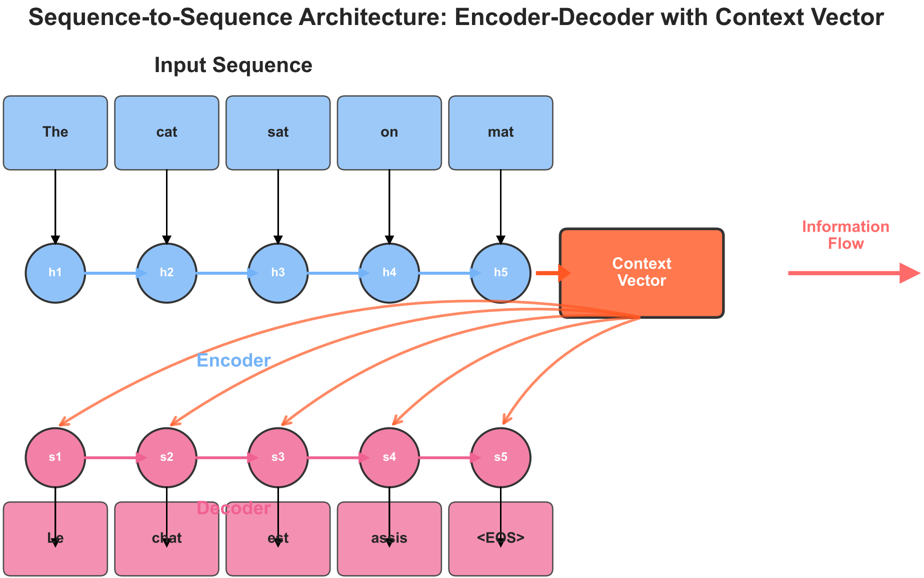
Traditional RNN Limitation:

- Each input produces exactly one output
- Fixed-length constraint breaks translation
- Can't handle summarization (long $\rightarrow$ short)
- Fails at code generation (comment $\rightarrow$ function)

Failed Approaches:

1. Pad to maximum length (wasteful)
2. Truncate long sequences (loses information)
3. Force 1:1 word mapping (doesn't work)

We need to decouple input and output lengths!


Sequence-to-Sequence Architecture: Encoder-Decoder with Context Vector
Input Sequence
The
cat
sat
on
mat
h1
h2
h3
h4
h5
Context Vector
Information Flow
Encoder
s1
s2
s3
s4
s5
Decoder
Le
chat
est
assis
<EOS>

Seq2Seq Mathematics: The Core Equations

Encoder Phase:

$$h_t = \text{LSTM}(h_{t-1}, x_t) \quad \text{where} \quad h_T = \text{context vector}$$

Decoder Phase:

$$s_t = \text{LSTM}(s_{t-1}, y_{t-1}) \quad \text{conditioned on context}$$

Output Probability:

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

Training Objective:

$$\max \textstyle\sum_{t=1}^{T'} \log P(y_t^* \mid y_{<t}^*, x)$$

Key Insights:

- Context vector = compressed representation
- Decoder generates one word at a time
- Training uses teacher forcing
- Inference uses beam search

Dimensions:

- Input length: T (variable)
- Output length: T' (variable)
- Context size: d (fixed)

Building Seq2Seq: PyTorch Implementation

```python
class Seq2SeqEncoder(nn.Module):
    def __init__(self, vocab_size, embed_size, hidden_size):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.lstm = nn.LSTM(embed_size, hidden_size, batch_first=True)

    def forward(self, x):
        # x: [batch, seq_len]
        embedded = self.embedding(x)  # [batch, seq_len, embed]
        output, (h_n, c_n) = self.lstm(embedded)
        return h_n, c_n  # Final hidden and cell states

class Seq2SeqDecoder(nn.Module):
    def __init__(self, vocab_size, embed_size, hidden_size):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embed_size)
        self.lstm = nn.LSTM(embed_size, hidden_size, batch_first=True)
        self.output_projection = nn.Linear(hidden_size, vocab_size)

    def forward(self, x, hidden, cell):
        # x: [batch, 1] (one word at a time)
        embedded = self.embedding(x)
        output, (h_n, c_n) = self.lstm(embedded, (hidden, cell))
        logits = self.output_projection(output)
        return logits, h_n, c_n
```

Key Components:

- **Encoder**: Processes entire input sequence
- **Context**: Hidden state transfer
- **Decoder**: Generates output step-by-step

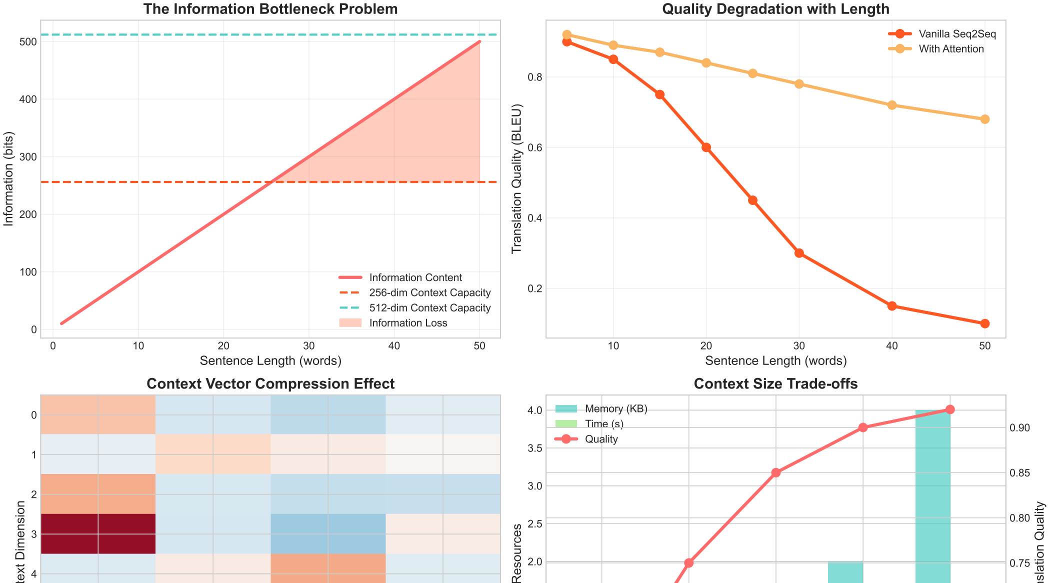
Training Process:

1. Encode input sentence
2. Initialize decoder with context
3. Teacher forcing during training
4. Beam search during inference

Modern Usage:

- Foundation of Transformer encoder-decoder
- Still used in specialized applications
- Basis for understanding attention

The Information Bottleneck Problem



The Key Insight:

- Don't compress everything into one vector
- Keep *all* encoder hidden states
- Let decoder *choose* what to focus on
- Different output words attend to different input words

Attention Mechanism:

$$c_t = \sum_{i=1}^{T} \alpha_{t,i} h_i \quad \text{where} \quad \alpha_{t,i} = \frac{\exp(e_{t,i})}{\sum_{j=1}^{T} \exp(e_{t,j})}$$

Alignment Score:

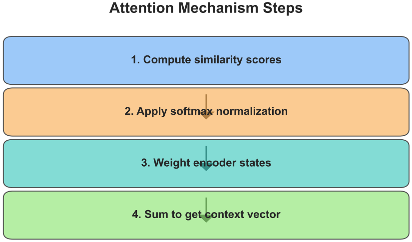
$$e_{t,i} = \text{align}(s_{t-1}, h_i) \quad \text{(similarity function)}$$

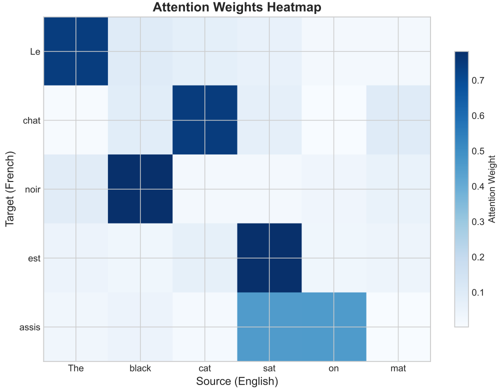
Breakthrough Results:

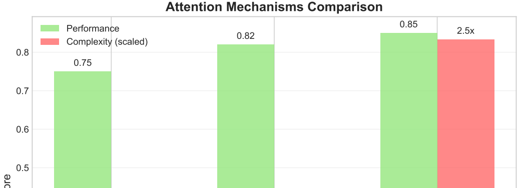
- Google NMT (2016): 60% improvement
- Handles sentences up to 80+ words
- Foundation for Transformers
- Powers all modern LLMs

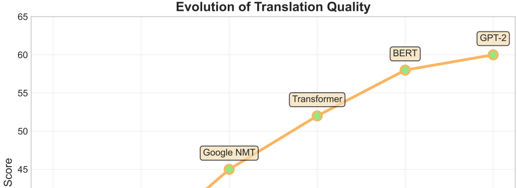
"Attention is All You Need"
(Transformer paper, 2017)

Attention in Action: What the Model Looks At









Implementing Attention: The Complete Mechanism

```python
class AttentionMechanism(nn.Module):
    def __init__(self, hidden_size):
        super().__init__()
        self.hidden_size = hidden_size
        self.W_a = nn.Linear(hidden_size, hidden_size)
        self.U_a = nn.Linear(hidden_size, hidden_size)
        self.v_a = nn.Linear(hidden_size, 1)

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

        batch_size, seq_len, _ = encoder_outputs.size()

        # Expand decoder hidden for all time steps
        decoder_hidden = decoder_hidden.unsqueeze(1).repeat(1,
            seq_len, 1)

        # Compute alignment scores
        energy = torch.tanh(
            self.W_a(decoder_hidden) + self.U_a(encoder_outputs)
        )
        attention_scores = self.v_a(energy).squeeze(2)

        # Apply softmax to get attention weights
        attention_weights = F.softmax(attention_scores, dim=1)

        # Compute context vector
        context = torch.bmm(
            attention_weights.unsqueeze(1),
            encoder_outputs
        ).squeeze(1)

        return context, attention_weights
```

Step-by-Step Process:

1. Compute similarity between decoder state and all encoder states
2. Apply softmax to get probability distribution
3. Weight encoder states by attention
4. Sum to get context vector

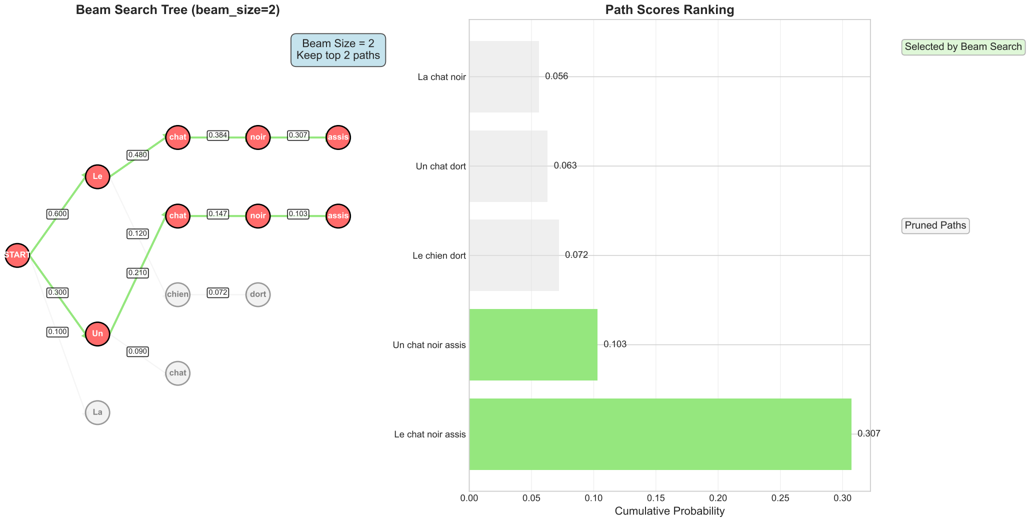
Three Attention Types:

- **Dot Product**: $h_i \cdot s_t$
- **Additive**: $v^T \tanh(W_h h_i + W_s s_t)$
- **Scaled Dot**: $\frac{h_i \cdot s_t}{\sqrt{d}}$

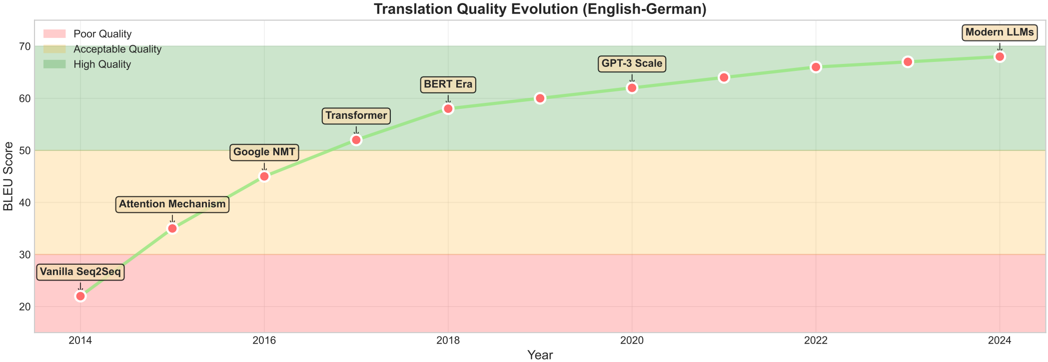
Modern Impact:

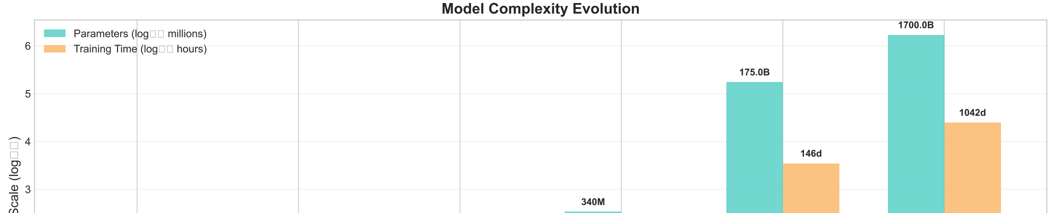
- Transformer uses scaled dot-product
- Multi-head attention in GPT/BERT
- Self-attention in modern LLMs

Beam Search: Finding the Best Translation



From 2014 to 2024: The Performance Revolution





Connecting to Modern AI: GPT, BERT, and Beyond

Seq2Seq Lives On in Modern Models:

- **ChatGPT:** Uses encoder-decoder principles internally
- **GitHub Copilot:** Comment $\rightarrow$ code is seq2seq
- **BERT:** Encoder-only architecture
- **T5:** "Text-to-Text Transfer Transformer" - pure seq2seq

Key Evolution Points:

- RNN $\rightarrow$ Transformer (parallelization)
- Single attention $\rightarrow$ Multi-head attention
- Fixed context $\rightarrow$ Self-attention
- Supervised $\rightarrow$ Pre-training + fine-tuning

Core Principle Unchanged: Variable-length input $\rightarrow$ Variable-length output

2024 Statistics:

- 127B+ model parameters (vs 10M in 2014)
- Trillions of tokens training data
- Sub-second response times
- Multi-modal capabilities

Industry Impact:

- $127B+ market size
- 40% developer productivity gains
- 1B+ daily translations
- 80% customer service automation

Week 4 Lab: Build Your Own Translator

In today's lab, you will:

1. Implement a complete seq2seq model from scratch
2. Train it on English-French translation
3. Add attention mechanism and see the improvement
4. Visualize attention weights on real examples
5. Compare different beam search strategies
6. Connect your implementation to modern transformers

Dataset: 10K English-French sentence pairs
Tools: PyTorch, Jupyter notebook with interactive visualizations
Outcome: Working translator that you can test with your own sentences!

Ready to build the technology behind Google Translate?

Key Takeaways: Breaking Free from Fixed Length

Core Concepts Mastered:

1. **Variable-length problem:** Why 1:1 mapping fails
2. **Encoder-decoder solution:** Separate encoding from decoding
3. **Information bottleneck:** Single context vector limitation
4. **Attention mechanism:** Dynamic focus on relevant inputs
5. **Beam search:** Efficient search through output space

Modern Relevance:

- Foundation of all current language models
- Core principle in ChatGPT, Copilot, and translation systems
- Attention evolved into self-attention (Transformers)
- Encoder-decoder architectures still dominant

Next Week: Transformers - "Attention is All You Need"

You now understand the foundation that made modern AI possible!

References and Further Reading

Foundational Papers:

- Sutskever et al. (2014). "Sequence to Sequence Learning with Neural Networks"
- Bahdanau et al. (2015). "Neural Machine Translation by Jointly Learning to Align and Translate"
- Luong et al. (2015). "Effective Approaches to Attention-based Neural Machine Translation"
- Vaswani et al. (2017). "Attention Is All You Need"

Modern Applications:

- Wu et al. (2016). "Google's Neural Machine Translation System"
- Radford et al. (2019). "Language Models are Unsupervised Multitask Learners" (GPT-2)
- Brown et al. (2020). "Language Models are Few-Shot Learners" (GPT-3)

Interactive Resources:

- The Illustrated Transformer (Jay Alammar)
- OpenAI's GPT papers and blog posts
- Hugging Face Transformers documentation
- Today's lab notebook with complete implementations