Sequence-to-Sequence Models

Encoder-Decoder Architecture & Attention Mechanisms

Natural Language Processing - Week 4

2025

Overview

The Variable-Length Challenge

Why Sequence-to-Sequence?

Previous models' limitations:

- **Fixed-size** input/output
- Cannot handle variable lengths
- Loss of **sequential information**

Key Innovation

Map variable-length input to variable-length output

Sutskever et al., 2014 - Revolutionary approach to sequence modeling

Applications

- Machine Translation
  - English $\rightarrow$ French
  - Variable sentence lengths
- Text Summarization
  - Long article $\rightarrow$ Brief summary
- Dialogue Systems
  - Question $\rightarrow$ Answer

The Core Architecture

Encoder

- Processes **input sequence**
- Creates context vector c
- Hidden states: $h_1, h_2, \ldots, h_T$

Decoder

- Generates **output sequence**
- Uses context vector c
- Autoregressive generation

Encoder Equations

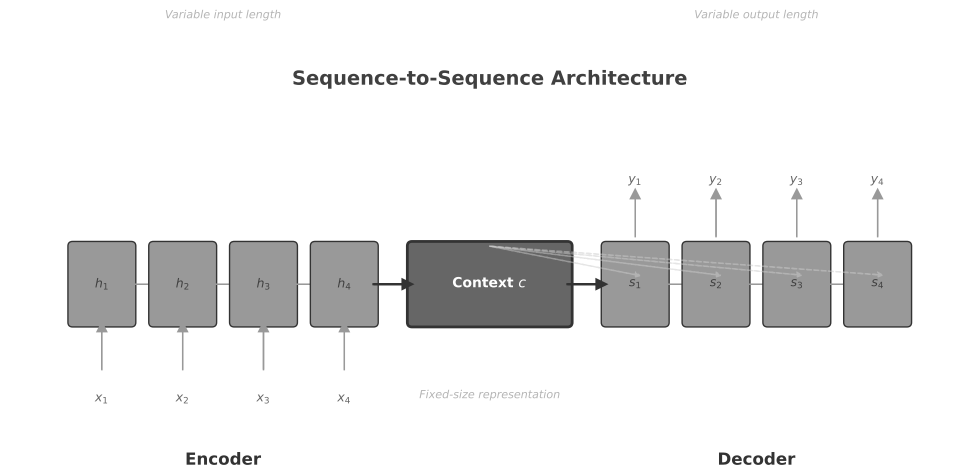
$$h_t = \mathrm{RNN}_{\mathrm{enc}}(x_t, h_{t-1}) \tag{1}$$
$$c = h_T \tag{2}$$

Decoder Equations

$$s_t = \mathrm{RNN}_{\mathrm{dec}}(y_{t-1}, s_{t-1}, c) \tag{3}$$
$$p(y_t \mid y_{<t}, x) = \mathrm{softmax}(W_s s_t) \tag{4}$$

The context vector c is the information bottleneck

Visual Architecture



Teacher Forcing vs Inference

Training: Teacher Forcing

- Use **ground truth** as input
- Faster convergence
- Exposure bias problem

Inference: Autoregressive

- Use **own predictions**
- Error accumulation
- Beam search helps

Training Process

At each step t:

1. Input: true y_{t-1}
2. Predict: $\hat{y}_t$
3. Loss: $\mathcal{L}(y_t, \hat{y}_t)$

Inference Process

At each step t:

1. Input: predicted $\hat{y}_{t-1}$
2. Predict: $\hat{y}_t$
3. Continue until ¡EOS¿

Teacher forcing creates a mismatch between training and inference

The Attention Solution

Motivation

- **Bottleneck** in fixed-size c
- Long sequences lose information
- Need dynamic context

Key Idea
Different decoder steps attend to different encoder positions

Attention Benefits

- Solves long-range dependencies
- Provides alignment
- Improves gradient flow

Attention Computation

Mathematical Formulation

$$e_{tj} = a(s_{t-1}, h_j) \tag{5}$$

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

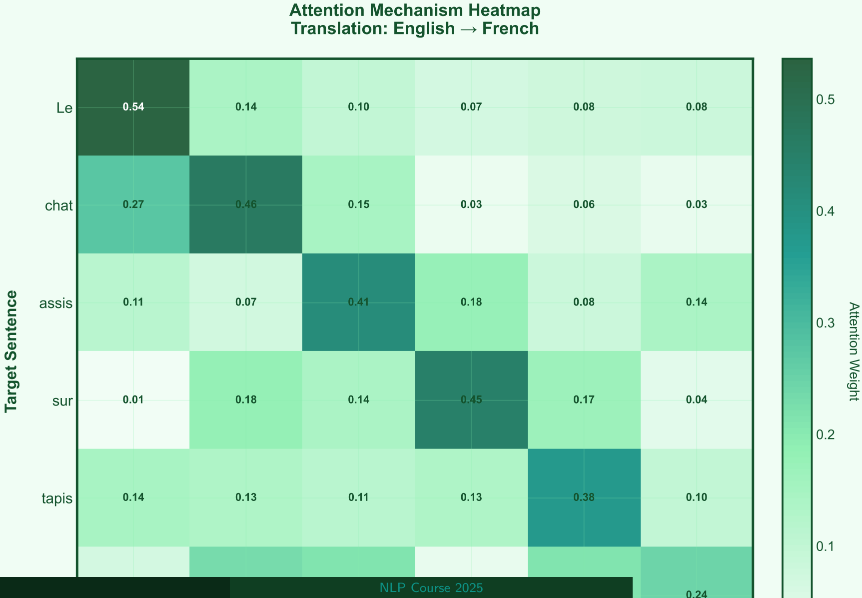
$$c_t = \sum_{j=1}^{T} \alpha_{tj} h_j \tag{7}$$

Where:

- e_{tj}: alignment score
- α_{tj}: **attention weight**
- c_t: dynamic context

Bahdanau et al., 2015 - Neural Machine Translation by Jointly Learning to Align and Translate

Attention Visualization



Beam Search Decoding

Why Not Greedy?

- Greedy: **locally optimal**
- May miss better sequences
- No backtracking possible

Beam Search Solution

- Keep *k* **best** hypotheses
- Explore multiple paths
- Balance quality vs speed

Beam Width Trade-off

- $k = 1$: Greedy (fast, lower quality)
- $k = 5 - 10$: Typical (balanced)
- $k = \infty$: Exhaustive (slow, optimal)

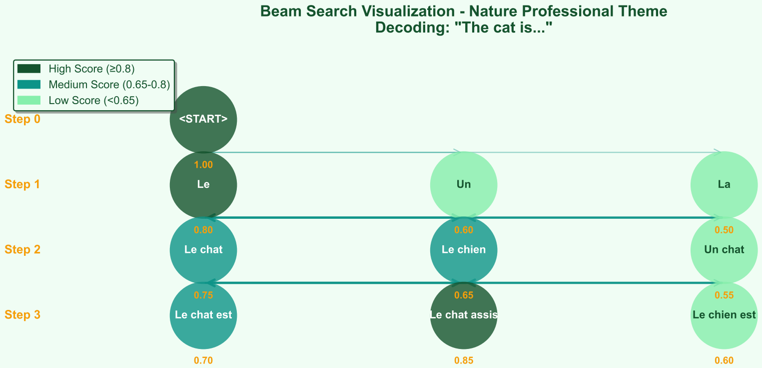
Algorithm

Beam Search Steps

1. Initialize beam with ¡START¿
2. For each time step:
   - Expand all hypotheses
   - Score all candidates
   - Keep top-k sequences
3. Stop when:
   - All beams end with ¡EOS¿
   - Maximum length reached
4. Return highest scoring sequence

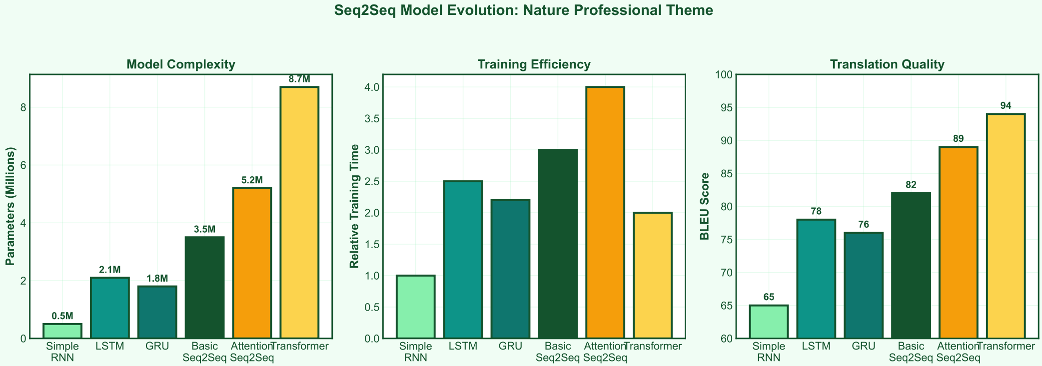
Beam search significantly improves translation quality

Beam Search Tree



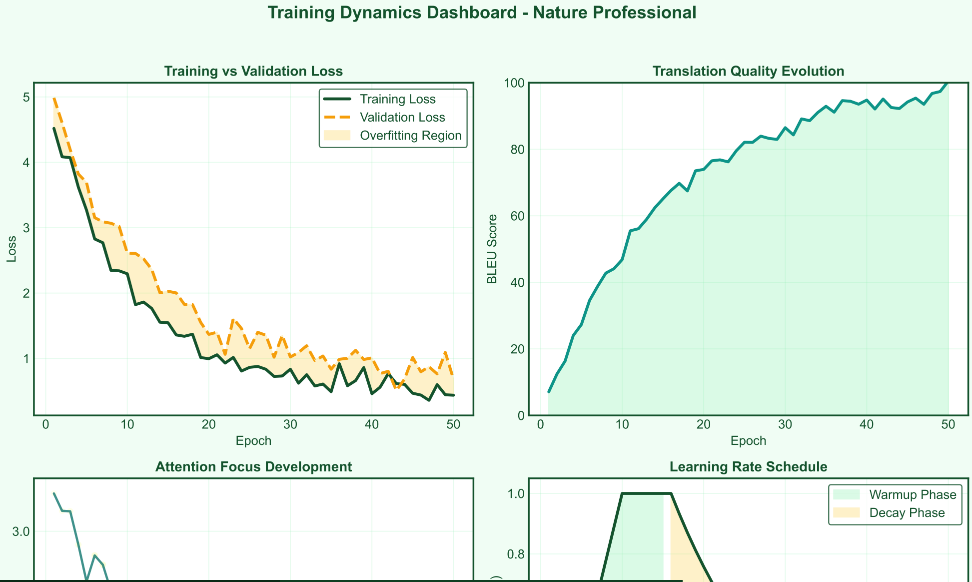
Beam width $k = 3$: Exploring multiple translation hypotheses

Model Comparison



Evolution from RNN to Transformer: complexity vs performance trade-offs

Training Dynamics



PyTorch Implementation

Encoder

```
class Encoder(nn.Module):
    def __init__(self, input_dim,
                 emb_dim, hid_dim,
                 n_layers, dropout):
        super().__init__()
        self.embedding = nn.Embedding(
            input_dim, emb_dim)
        self.rnn = nn.LSTM(emb_dim,
            hid_dim, n_layers,
            dropout=dropout)
        self.dropout = nn.Dropout(dropout)

    def forward(self, src):
        embedded = self.dropout(
            self.embedding(src))
        outputs, (hidden, cell) = \
            self.rnn(embedded)
        return hidden, cell
```

Decoder with Attention

```
class Decoder(nn.Module):
    def __init__(self, output_dim,
                 emb_dim, hid_dim,
                 n_layers, dropout):
        super().__init__()
        self.embedding = nn.Embedding(
            output_dim, emb_dim)
        self.rnn = nn.LSTM(emb_dim + hid_dim,
            hid_dim, n_layers,
            dropout=dropout)
        self.attention = Attention(hid_dim)
        self.fc = nn.Linear(
            hid_dim * 2, output_dim)

    def forward(self, input, hidden,
                cell, encoder_outputs):
        embedded = self.embedding(input)
        a = self.attention(hidden[-1],
            encoder_outputs)
        weighted = torch.bmm(a.unsqueeze(1),
            encoder_outputs).squeeze(1)
        rnn_input = torch.cat(
            (embedded, weighted), dim=1)
        output, (hidden, cell) = \
            self.rnn(rnn_input.unsqueeze(0),
                     (hidden, cell))
        prediction = self.fc(torch.cat(
            (output.squeeze(0), weighted),
            dim=1))
        return prediction, hidden, cell
```

Complete implementation available in course repository

Training Tips & Tricks

Optimization

- **Gradient clipping** essential
- Learning rate scheduling
- Dropout in RNN layers

Hyperparameters

- Hidden size: 256-512
- Layers: 2-4
- Dropout: 0.2-0.3
- Beam width: 5-10

Data Preprocessing

- **Tokenization** consistency
- Vocabulary size limits
- Handle rare words

Common Issues

Exposure Bias

- Problem: Train/test mismatch
- Solution: Scheduled sampling

Long Sequences

- Problem: Gradient vanishing
- Solution: Attention mechanism

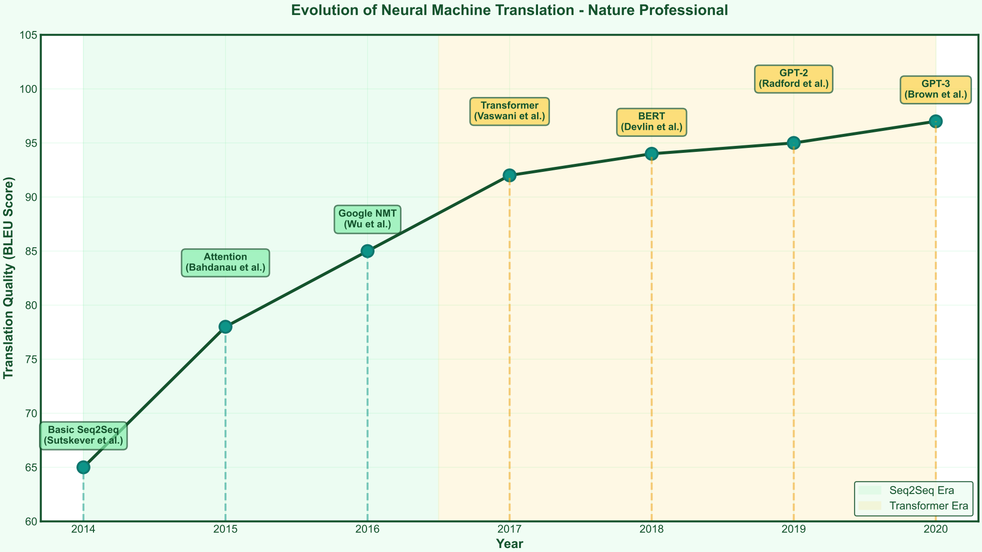
Rare Words

- Problem: UNK tokens
- Solution: Copy mechanism

Evaluation Metrics

- **BLEU**: n-gram precision
- ROUGE: recall-oriented
- METEOR: semantic similarity

From Seq2Seq to Transformers



The path from basic seq2seq to modern transformer architectures

Exercise: Simple Translation

Task

Implement a basic seq2seq model for number translation:

- Input: "one two three"
- Output: "1 2 3"

Starter Code

```python
# Define vocabularies
source_vocab = {
    'one': 1, 'two': 2,
    'three': 3, 'four': 4,
    'five': 5, '<sos>': 0,
    '<eos>': 6, '<pad>': 7
}
target_vocab = {
    '1': 1, '2': 2, '3': 3,
    '4': 4, '5': 5,
    '<sos>': 0, '<eos>': 6,
    '<pad>': 7
}

# Training data
train_data = [
    ('one-two', '1-2'),
    ('three-four-five', '3-4-5'),
    # Add more examples
]
```

Implementation Steps

Step 1: Data Processing

- Tokenize sequences
- Convert to indices
- Pad to same length

Step 2: Model Building

- Create Encoder class
- Create Decoder class
- Combine into Seq2Seq

Step 3: Training Loop

- Teacher forcing
- Calculate loss
- Backpropagation

Key Takeaways

What We Learned

1. **Encoder-Decoder** architecture
   - Variable length handling
   - Context vector bottleneck
2. **Attention Mechanism**
   - Dynamic context
   - Alignment visualization
3. **Training Techniques**
   - Teacher forcing
   - Beam search decoding

Core Insight

Attention solves the information bottleneck problem

Practical Applications

- Machine Translation
- Text Summarization
- Question Answering
- Image Captioning
- Speech Recognition

Next Week Preview

Week 5: Transformers

- Self-attention mechanism
- Parallel processing
- Positional encoding
- Multi-head attention

Seq2seq models laid the foundation for modern NLP architectures

References

Essential 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

Additional Resources

- Course Repository
  github.com/course/week4-seq2seq
- Lab Notebook
  week04_seq2seq_lab.ipynb
- PyTorch Tutorial
  pytorch.org/tutorials/seq2seq

Office Hours

Tuesday 2-4 PM
Thursday 10-12 AM

Questions? Please post on the course forum or attend office hours

Thank You!

See you next week for Transformers

Remember to complete the lab exercise