Lecture 12:

Advanced Sequence Models

Deep Learning for Natural Language Processing

Ho-min Park

homin.park@ghent.ac.kr

powersimmani@gmail.com

Lecture Contents

Part 7: Conclusion & Next Steps

Part 1/7:

Introduction & Review

1. Review of Last Lesson
2. Today's Learning Objectives
3. Exploring the Limitations of RNNs

Review of Last Lesson

Recurrent Neural Networks

RNNs process sequential data by maintaining **hidden states** that carry information across time

Hidden States

Hidden states act as **memory**, carrying contextual information from previous time steps to future

Gradient Challenges

Vanishing/exploding gradients make it difficult to learn long-term dependencies in sequences

LSTM & GRU Solutions

LSTM and GRU architectures use gates to mitigate gradient problems and capture long-term patterns

Gate Control Flow

Gates control information flow: **forget gate** (what to discard), **input gate** (what to add), **output gate**

Key Applications

RNNs excel at: **language modeling**, **time series prediction**, and **speech recognition**

1

Understand RNN Limitations

Explore vanishing gradients and long-term dependency issues

2

Master Bidirectional RNNs

Learn architecture and applications for context-aware processing

3

Implement Seq2Seq Models

Build encoder-decoder architectures with attention mechanism

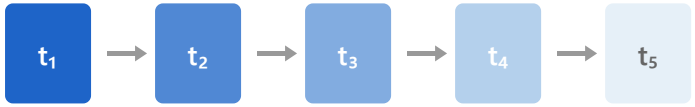
4

Apply Advanced Techniques

Use teacher forcing, batching, and masking in practice

Key Limitations of Standard RNNs

∇ Vanishing Gradient Problem



- Gradients diminish exponentially during backpropagation
- Early time steps receive negligible weight updates
- Hinders learning from distant past information

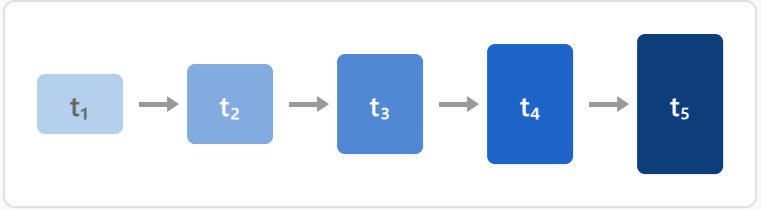
↔ Long-term Dependencies

Information decays over long sequences

- Difficulty capturing dependencies across long sequences
- Context from early tokens gets lost or diluted
- Limits performance on tasks requiring long-range context

↗ Exploding Gradient Problem

Sequential Processing Bottleneck



- Gradients grow exponentially during backpropagation
- Causes numerical instability and training divergence
- Requires gradient clipping to stabilize training

Cannot parallelize ⚠

- Must process sequence step-by-step sequentially
- Cannot leverage parallel computation effectively
- Slow training and inference on long sequences

Part 2/7:

Bidirectional RNNs

4. Why Bidirectional?
5. Bidirectional RNN Architecture
6. BiRNN Formulas and Operations
7. BiRNN Pros and Cons & Applications

Context Matters: Unidirectional vs Bidirectional Processing

Unidirectional RNN

→

The cat sat on the **bank** of the river

t_1 → t_2 → t_3 → ...

Limited Context

Only sees **past** words. At "bank", cannot see "river" yet.

✗ Cannot disambiguate without future context

✗ Incomplete understanding at each time step

Bidirectional RNN

⇄

The cat sat on the **bank** of the river

t_1 → t_2 → t_3

t_1 ← t_2 ← t_3

Full Context

Sees **both past and future** words. At "bank", also sees "river".

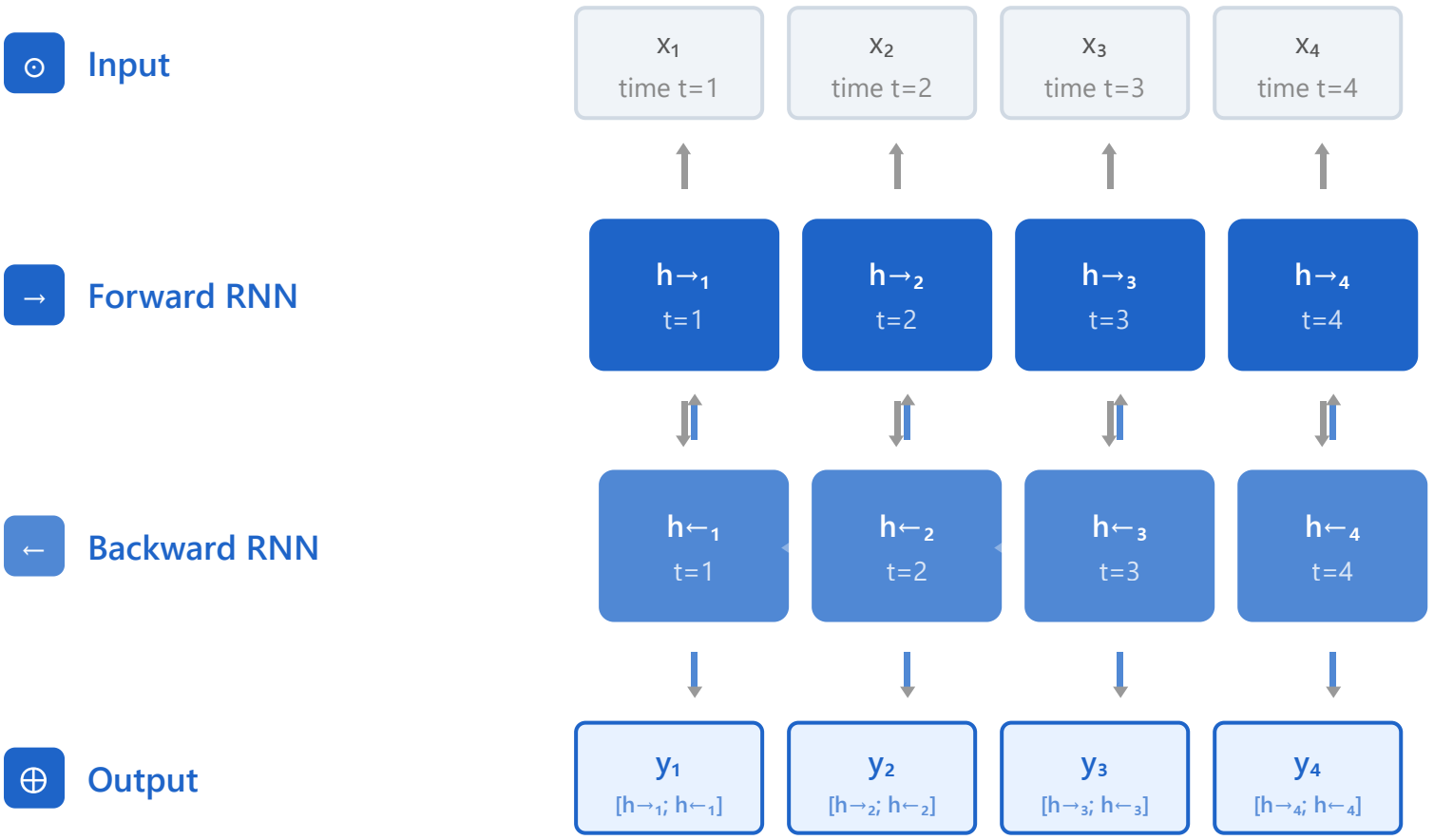
✓ Complete context for better disambiguation

✓ Richer representations at each time step

✗ Poor performance on tasks requiring full sentence context

✓ Superior for NER, POS tagging, sentiment analysis

Bidirectional RNN Architecture



Key Feature

BiRNN processes sequences in both directions, concatenating forward (h→) and backward (h←) hidden states to capture full context at each time step.

Training Process

1 Forward Pass

Both directional RNNs process the input sequence simultaneously. The Forward RNN computes hidden states from t=1 to t=T, while the Backward RNN computes from t=T to t=1.

$x_1, x_2, \ldots, x_t$ → Forward: $h\rightarrow_t = f(x_t, h\rightarrow_{t-1})$ + Backward: $h\leftarrow_t = f(x_t, h\leftarrow_{t+1})$

2 Output Computation

At each time step, the forward and backward hidden states are concatenated to produce the final output. This allows the model to utilize both past and future context at each position.

$h\rightarrow_t \oplus h\leftarrow_t \rightarrow y_t = g([h\rightarrow_t; h\leftarrow_t])$

$$\hat{y}_t = \text{softmax}(W_\gamma \cdot [h\rightarrow_t; h\leftarrow_t] + b_\gamma)$$

3 Loss Calculation

The loss is computed by comparing predictions with actual target values. The losses from all time steps are summed to obtain the total sequence loss.

$\hat{y}_t$ (prediction) vs y_t (target) → Loss = $\Sigma\, L(\hat{y}_t, y_t)$

$$L = -\Sigma_t \Sigma_i\, y_{t,i} \log(\hat{y}_{t,i}) \text{ (Cross-Entropy Loss)}$$

4 Backpropagation Through Time (BPTT)

Gradients are propagated backward through time. In BiRNN, BPTT is performed independently for the Forward RNN and Backward RNN.

Forward BPTT: $\partial L/\partial h_{\rightarrow 4}$ $\partial L/\partial h_{\rightarrow 3}$ $\partial L/\partial h_{\rightarrow 2}$ $\partial L/\partial h_{\rightarrow 1}$ ← backward propagation

Backward BPTT: $\partial L/\partial h_{\leftarrow 1}$ $\partial L/\partial h_{\leftarrow 2}$ $\partial L/\partial h_{\leftarrow 3}$ $\partial L/\partial h_{\leftarrow 4}$ → forward propagation

5 Parameter Update

All parameters are updated using the computed gradients. The Forward RNN, Backward RNN, and output layer parameters are updated independently.

$\nabla W_{\rightarrow}$ $\nabla W_{\leftarrow}$ ∇W_{γ} → $\theta \leftarrow \theta - \eta \nabla L$

$$W_{\rightarrow} \leftarrow W_{\rightarrow} - \eta \cdot \partial L/\partial W_{\rightarrow}\,,\; W_{\leftarrow} \leftarrow W_{\leftarrow} - \eta \cdot \partial L/\partial W_{\leftarrow}\,,\; W_{\gamma} \leftarrow W_{\gamma} - \eta \cdot \partial L/\partial W_{\gamma}$$

Training Summary

- ✓ Forward/Backward RNNs **do not share parameters** (independent learning)
- ✓ **LSTM/GRU recommended** to mitigate vanishing gradient problem
- ✓ BPTT is performed **independently for each direction**
- ✓ Requires full sequence → **not suitable for real-time** (offline only)

BiRNN Formulas and Operations

📝 Notation Guide

x_t Input at time t	$\vec{h}_t$ Forward hidden state	$\overleftarrow{h}_t$ Backward hidden state
W_f, U_f Forward weights	W_b, U_b Backward weights	y_t Output at time t

→ Forward RNN (Left to Right)

Hidden State:

$$\vec{h}_t = \tanh(W_f \cdot x_t + U_f \cdot \vec{h}_{t-1} + b_f)$$

Processes sequence from **start to end**, capturing past context. Each hidden state depends on current input and previous hidden state.

← Backward RNN (Right to Left)

Hidden State:

$$\overleftarrow{h}_t = \tanh(W_b \cdot x_t + U_b \cdot \overleftarrow{h}_{t+1} + b_b)$$

Processes sequence from **end to start**, capturing future context. Each hidden state depends on current input and next hidden state.

Output Combination

Concatenation:

$$h_t = [\vec{h}_t \; ; \; \overleftarrow{h}_t]$$

Output Layer:

$$y_t = W_o \cdot h_t + b_o = W_o \cdot [\vec{h}_t \; ; \; \overleftarrow{h}_t] + b_o$$

Combines both directions by **concatenating** forward and backward hidden states, creating a representation with **full context** (past + future).

Key Computational Points

Independent Processing

Forward and backward RNNs have separate parameters and process independently

Double Dimensionality

Output dimension is 2× hidden size due to concatenation: [$\vec{h}$; $\overleftarrow{h}$]

Parallel Computation

Full Sequence Required

Forward and backward passes can be computed in parallel for efficiency

Complete sequence must be available before backward RNN can start processing

Implementation Note

In practice, frameworks like PyTorch and TensorFlow handle BiRNN efficiently with built-in functions (e.g., `nn.LSTM(bidirectional=True)`). The framework automatically manages the forward and backward passes, parameter initialization, and output concatenation.

✓ Advantages

- ✓ **Full context** - captures both past and future information
- ✓ **Better representations** - richer hidden states at each time step
- ✓ **Improved accuracy** - superior performance on many NLP tasks
- ✓ **Disambiguation** - resolves ambiguity using surrounding context

✗ Disadvantages

- ✗ **Cannot stream** - requires full sequence before processing
- ✗ **Higher computation** - 2x parameters and memory compared to unidirectional
- ✗ **Not for generation** - unsuitable for autoregressive tasks
- ✗ **Slower training** - increased training time and inference latency

⚙ Key Applications

Named Entity Recognition

POS Tagging

Sentiment Analysis

Capture sentiment from entire text

Identify entities using full sentence context

Accurate part-of-speech labeling with context

Slot Filling

Extract structured info from utterances

Dependency Parsing

Understand syntactic relationships

Speech Recognition

Phoneme prediction with context

Part 3/7:

Sequence-to-Sequence

8. The Need for Seq2Seq
9. Encoder-Decoder Architecture
10. Encoder Details
11. Decoder Details
12. Overall Seq2Seq Process

Why Do We Need Sequence-to-Sequence Models?

⚠ Challenge: Variable-Length Input → Variable-Length Output

Machine Translation

Hello

→

안녕하세요

Text Summarization

Long article...

→

Short summary

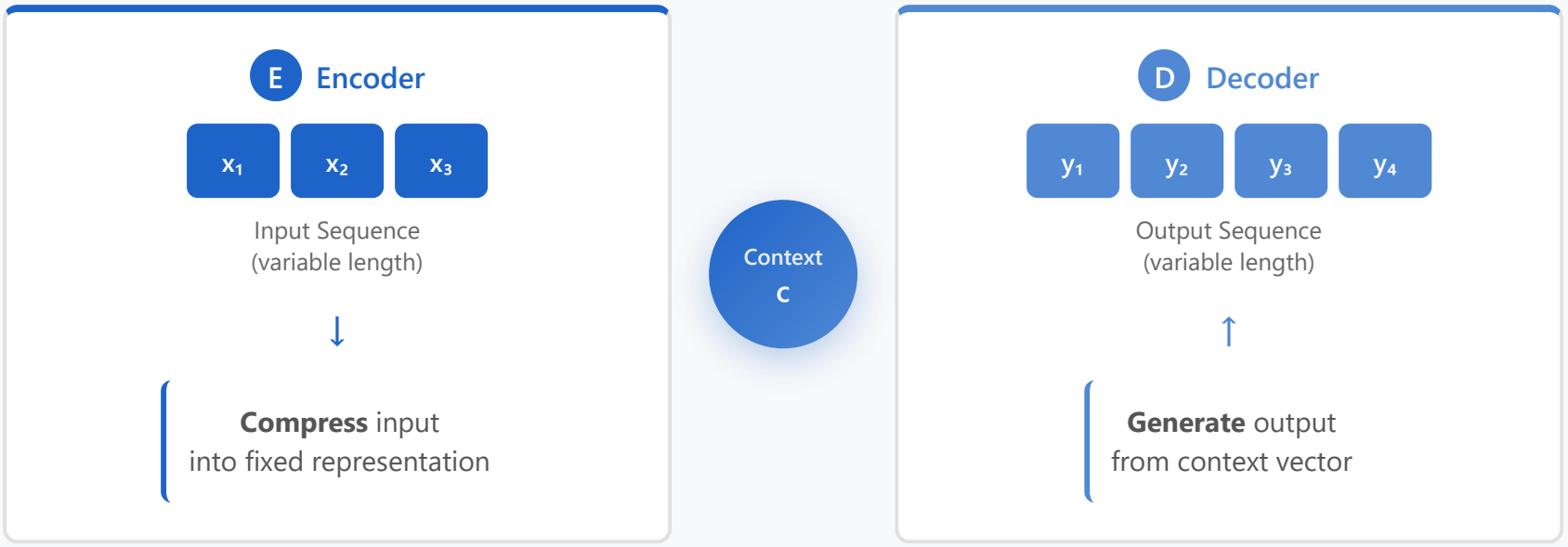
Question Answering

What is AI?

→

AI is...

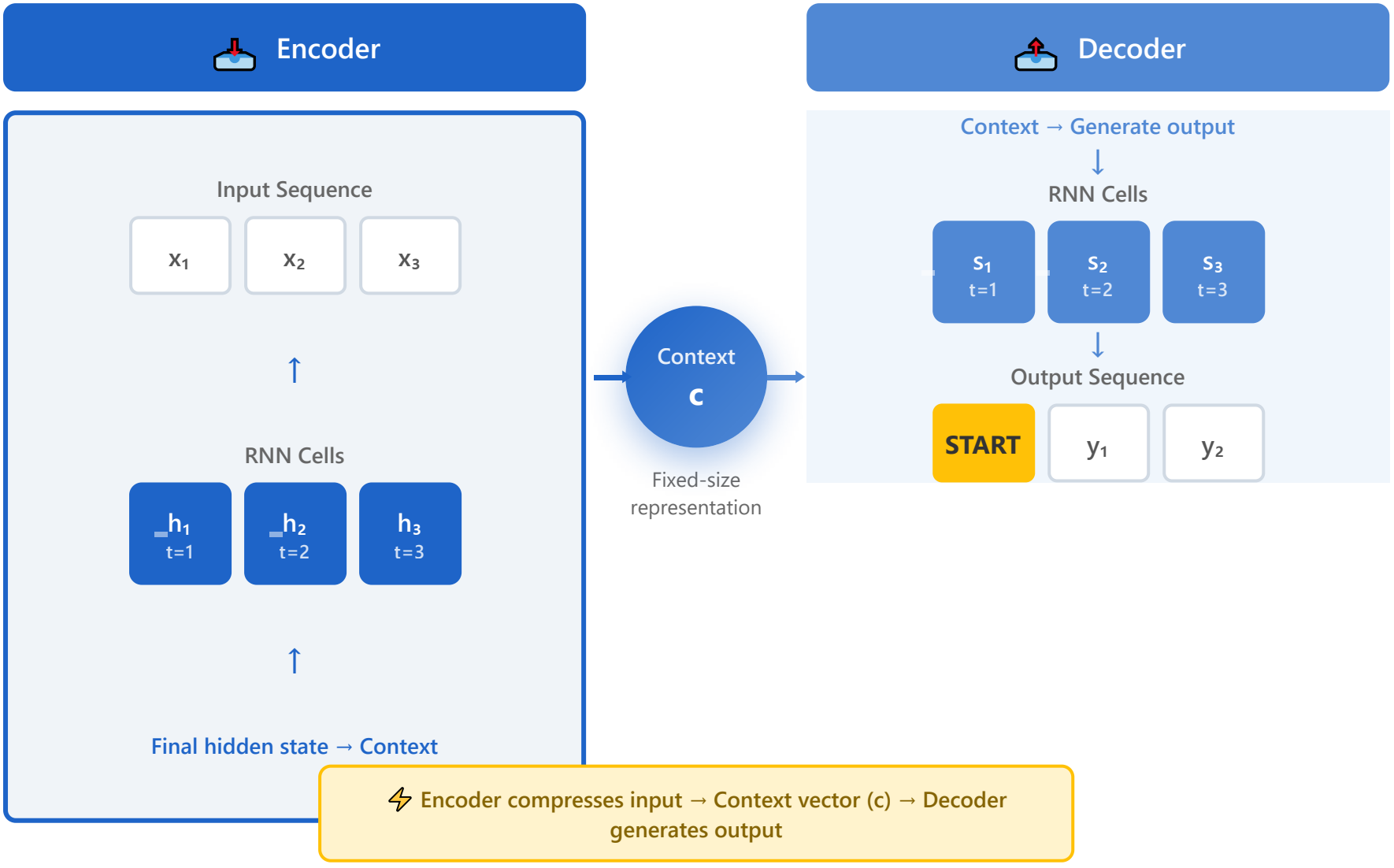
✓ Solution: Encoder-Decoder Architecture



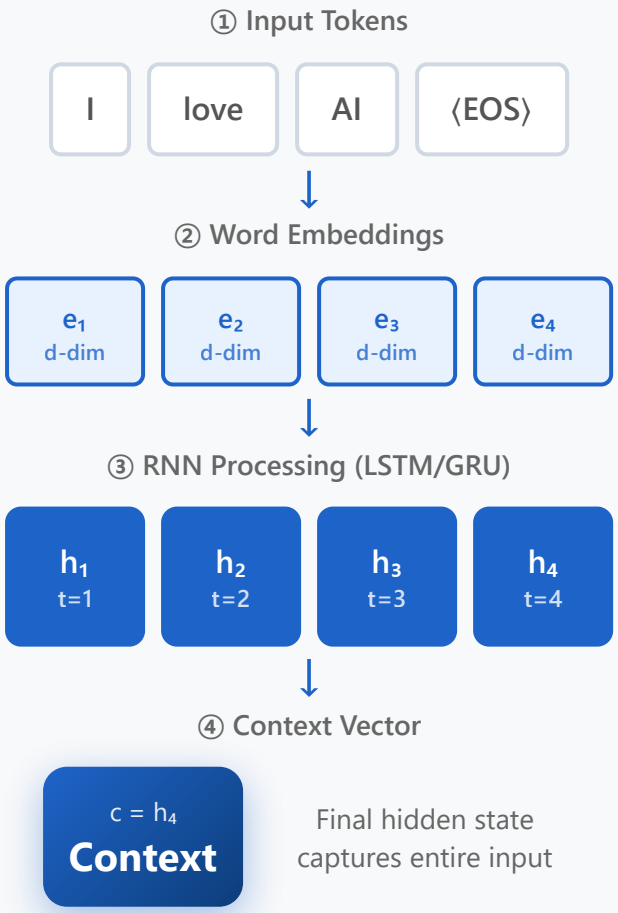
Handles different lengths

Fixed context vector

End-to-end learning


Encoder
Input Sequence
x1
x2
x3
RNN Cells
h1
t=1
h2
t=2
h3
t=3
Final hidden state → Context
Context
c
Fixed-size
representation
Decoder
Context → Generate output
RNN Cells
s1
t=1
s2
t=2
s3
t=3
Output Sequence
START
y1
y2
Encoder compresses input → Context vector (c) → Decoder generates output

Encoder Processing Flow



Key Components

- **Embedding Layer:** Convert tokens to vectors
- **RNN:** LSTM or GRU for sequential processing
- **Hidden States:** Maintain temporal information

Computation

- Process input sequentially left-to-right
- Each step updates hidden state

$$h_t = f(x_t, h_{t-1})$$

Output

- **Context vector (c):** Final hidden state h_n
- Fixed-size representation of variable-length input
- Passed to decoder as initial state

Decoder Processing Flow

① Context Vector (from Encoder)

Context **c**

Initializes decoder state: s_0 = c

↓

② Initial Input Token

⟨START⟩ Special token to begin generation

↓

③ RNN Processing (Autoregressive)

s_1 t=1	s_2 t=2	s_3 t=3	s_4 t=4

↓

④ Generated Output Tokens

나는 y_1	AI를 y_2	사랑해 y_3	⟨EOS⟩ y_4

Key Components

- **Initial State:** s_0 = context vector c
- **RNN:** LSTM or GRU for generation
- **Softmax:** Predicts next token

Computation

- Each step takes previous output as input
- Updates hidden state and predicts next token

$$s_t = f(y_{t-1}, s_{t-1})$$
$$y_t = \text{softmax}(W \cdot s_t)$$

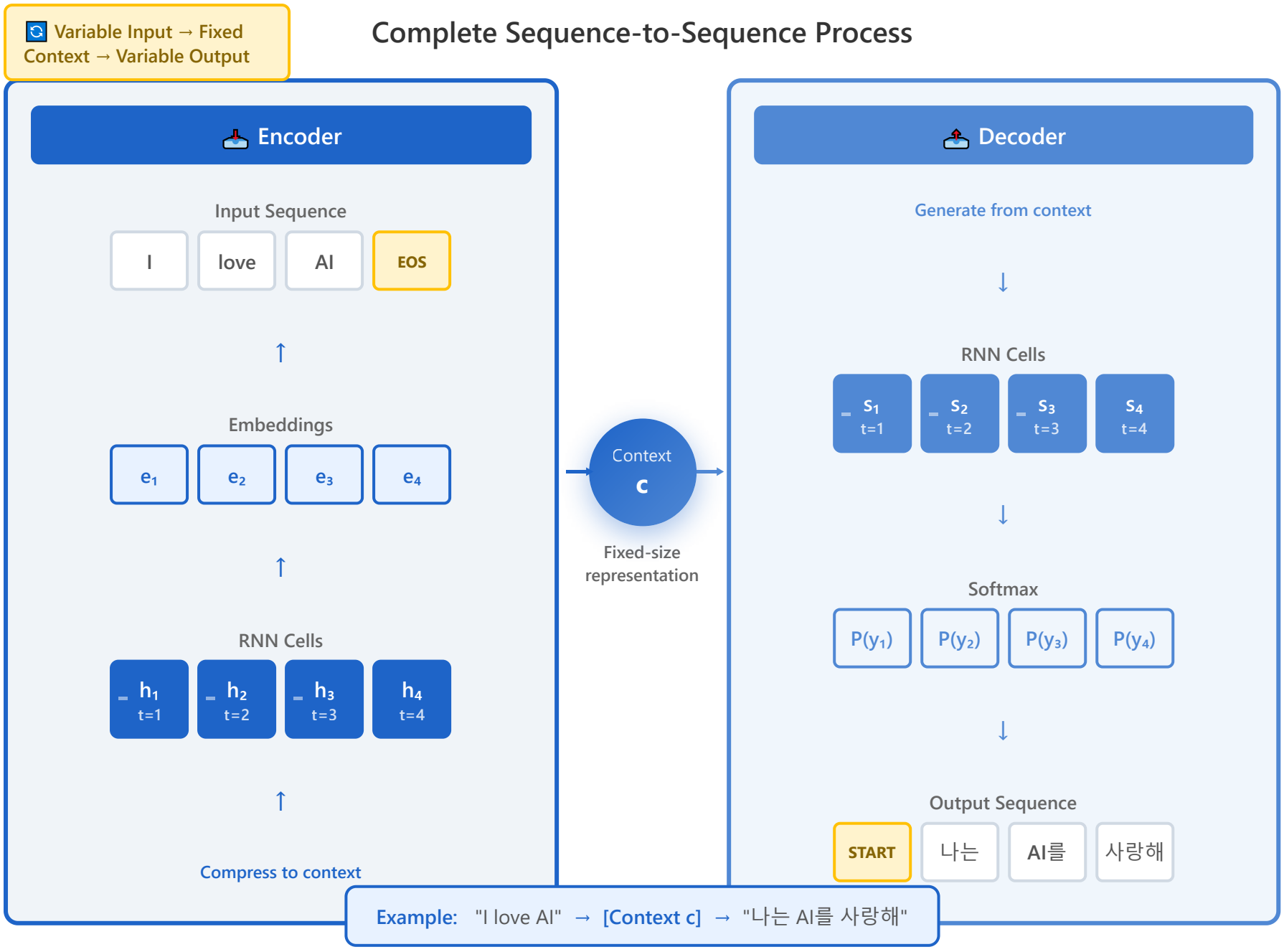
Generation Process

- **Autoregressive:** Uses own predictions
- Stops when ⟨EOS⟩ token is generated
- Can use greedy or beam search decoding

⚠ Key Difference

Decoder uses its own previous predictions as input (autoregressive), unlike encoder which

processes fixed input.


Variable Input → Fixed Context → Variable Output
Complete Sequence-to-Sequence Process
Encoder
Input Sequence
I
love
AI
EOS
Embeddings
e1
e2
e3
e4
RNN Cells
h1 t=1
h2 t=2
h3 t=3
h4 t=4
Compress to context
Context c
Fixed-size representation
Decoder
Generate from context
RNN Cells
s1 t=1
s2 t=2
s3 t=3
s4 t=4
Softmax
P(y1)
P(y2)
P(y3)
P(y4)
Output Sequence
START
나는
AI를
사랑해
Example: "I love AI" → [Context c] → "나는 AI를 사랑해"

Part 4/7:

Teacher Forcing

13. What is Teacher Forcing?
14. Teacher Forcing vs. Autoregressive
15. Teacher Forcing Problems and Solutions

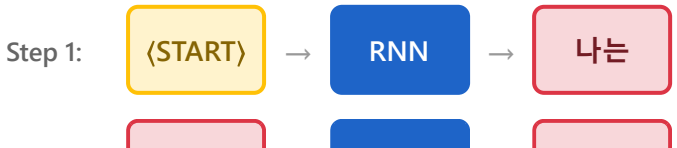
What is Teacher Forcing?

Definition

Teacher Forcing is a training strategy where the **ground-truth (target) tokens** from the training data are used as inputs to the decoder at each time step, instead of the model's own predictions.

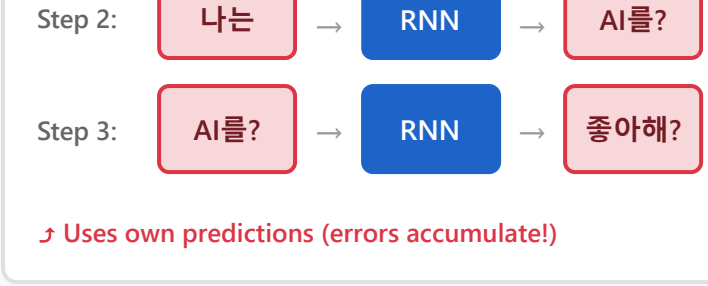
Without Teacher Forcing

(Autoregressive)



With Teacher Forcing

(Training Mode)

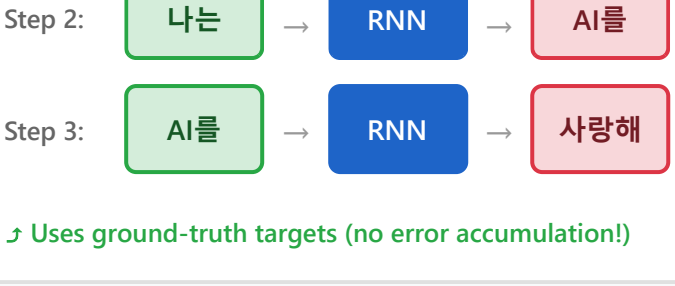


↻ Uses own predictions (errors accumulate!)

✗ Errors compound over time

✗ Slower training convergence

✗ Exposure bias problem



↻ Uses ground-truth targets (no error accumulation!)

✓ Faster training convergence

✓ Prevents error accumulation

✓ More stable gradient flow

Teacher Forcing is used during **training** to speed up learning. During **inference**, the model uses its own predictions (autoregressive).

Teacher Forcing vs. Autoregressive Decoding

Teacher Forcing

Training

How it Works

- Uses **ground-truth** target tokens as input
- Model receives correct previous token
- No error accumulation during training

✓ Advantages

+ Faster convergence
+ Stable gradients

Autoregressive

Inference

How it Works

- Uses model's **own predictions** as input
- Each prediction feeds into next step
- Errors can compound over sequence

✓ Advantages

+ Matches inference mode
+ More realistic training

+ Easier to train
+ Parallel processing

✗ Disadvantages

- Exposure bias problem
- Train/test mismatch
- Less robust at inference
- Can't handle own errors

+ Better generalization
+ No exposure bias

✗ Disadvantages

- Slower training
- Unstable gradients
- Error accumulation
- Sequential processing

📊 Quick Comparison

Aspect	Teacher Forcing	Autoregressive
Input Source	Ground truth	Model predictions
Usage Phase	Training	Inference
Training Speed	Fast ⚡	Slow 🐌
Robustness	Lower 📉	Higher 📈

Teacher Forcing: Problems and Solutions

⚠ Key Problems

✗ Exposure Bias
Model never sees its own mistakes during training, struggles at inference

✗ Train-Test Mismatch
Training uses ground truth, but inference uses predictions

✗ Error Recovery
Model doesn't learn to recover from its own errors

✗ Overconfidence
Model becomes overconfident on ground-truth inputs

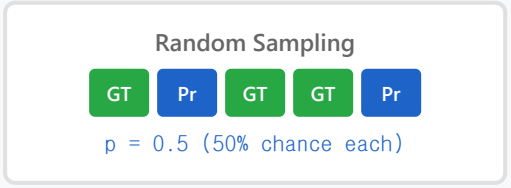
✓ Solutions & Techniques

1 Scheduled Sampling

Gradually mix ground-truth and model predictions during training

2 Mixed Training

Randomly alternate between teacher forcing and autoregressive modes



3 Curriculum Learning

Start with easier tasks and progressively increase difficulty

Learning Stages

1. Short sequences
2. Medium sequences
3. Long sequences
4. Full complexity

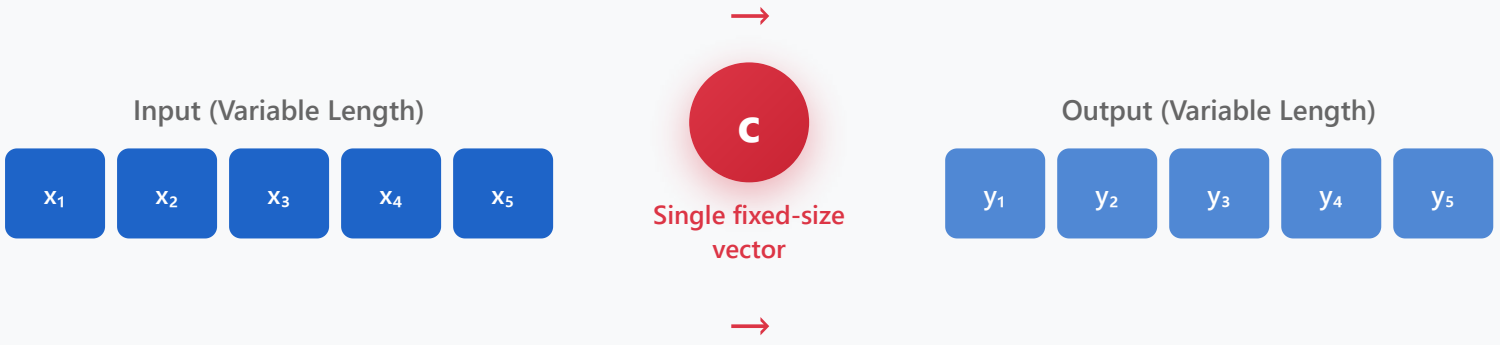
Part 5/7:

Attention Mechanism Basics

16. Limitations of Seq2Seq
17. Core Ideas of Attention
18. Attention Mechanism Structure
19. Attention Formula
20. Effects of Attention

Limitations of Basic Seq2Seq Models

The Information Bottleneck Problem



1 Information Bottleneck

All input information must be compressed into a **single fixed-size context vector**, regardless of input length.

Impact:

Long sequences lose critical information. Model struggles with lengthy inputs.

2 Gradient Vanishing

Earlier tokens in long sequences have **vanishing gradients** as information must flow through many time steps.

Impact:

Poor performance on long-range dependencies and long documents.

3 Equal Weighting

All input tokens treated equally when creating context vector - **no selective focus** on important parts.

Impact:
Cannot emphasize relevant information for each output step.

4 No Alignment

Decoder has **no mechanism to look back** at specific input positions when generating each output token.

Impact:
Difficult tasks requiring input-output alignment (e.g., translation).

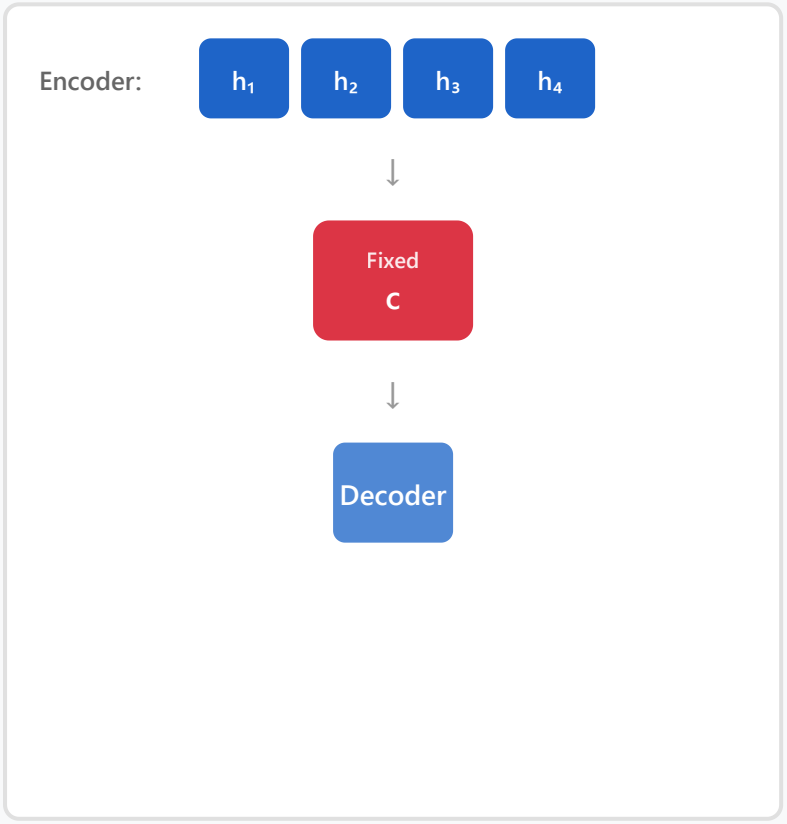
Solution needed: Attention Mechanism allows decoder to selectively focus on different parts of input for each output step!

Core Ideas of Attention Mechanism

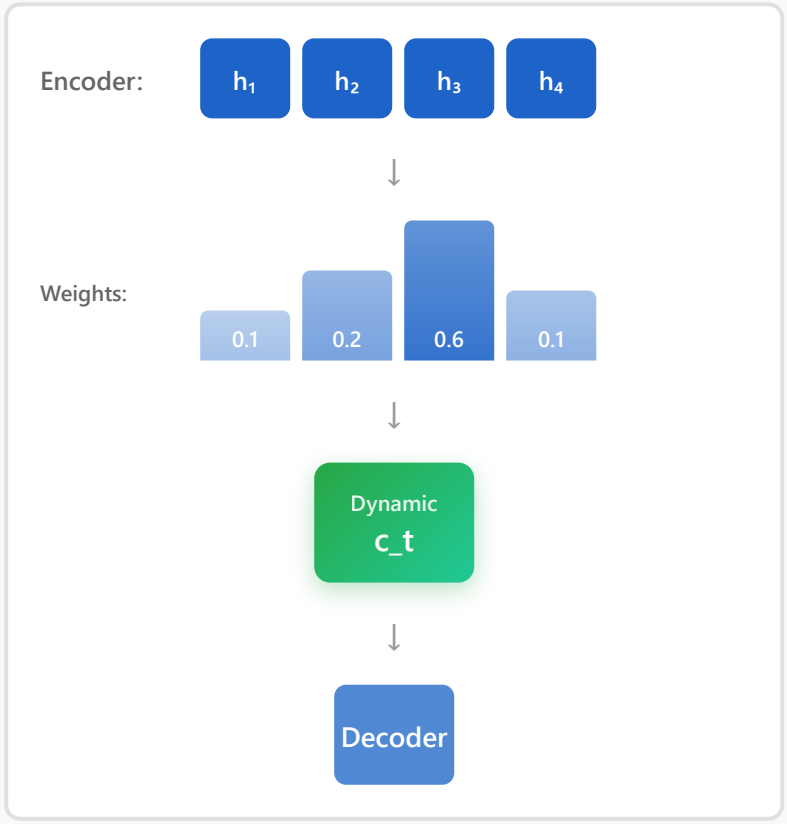
Central Concept

Instead of compressing everything into a single context vector, **let the decoder selectively focus on different parts of the input** at each time step based on what's most relevant.

Without Attention



With Attention



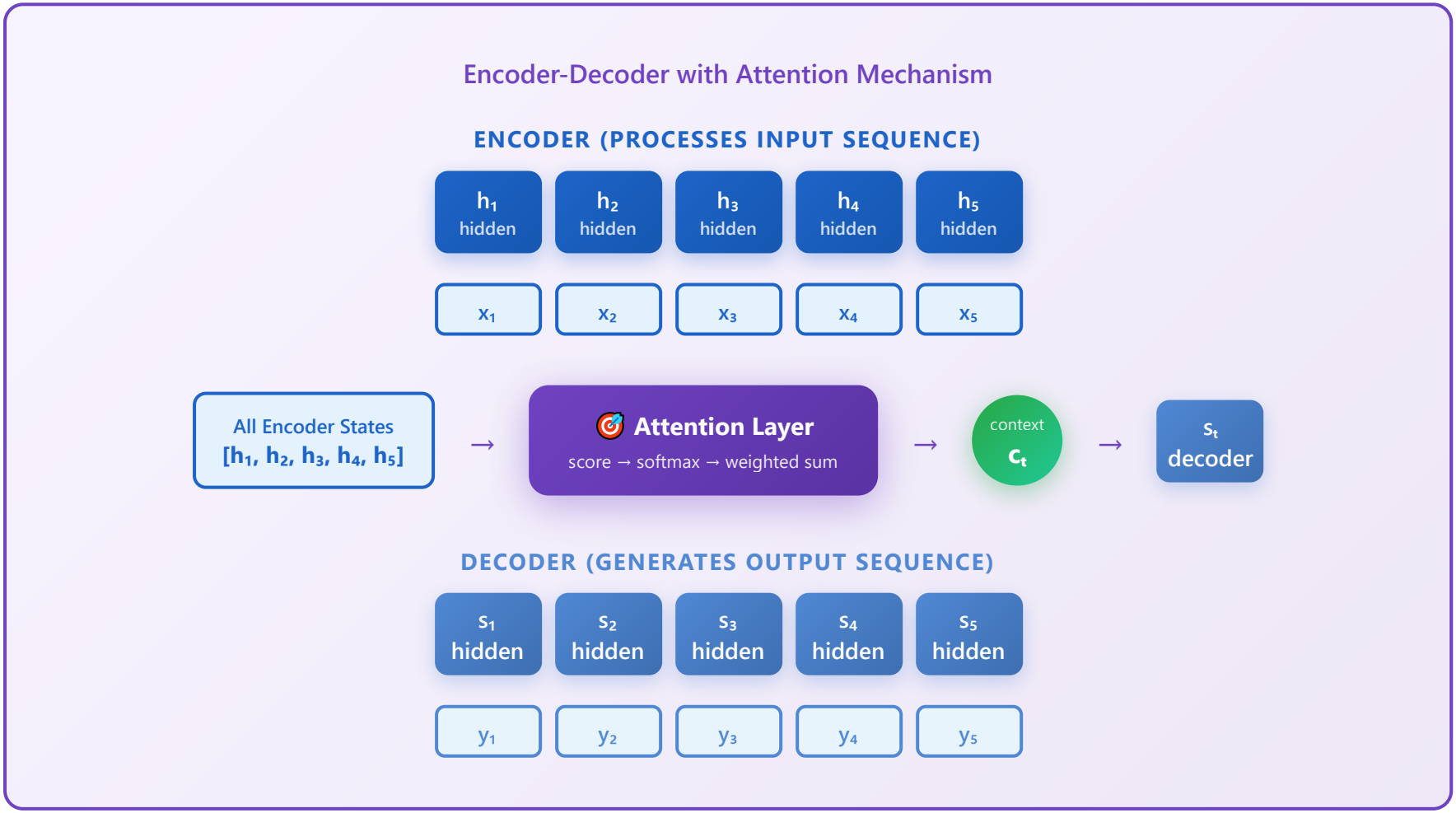
Same static context for all decoder steps. Information

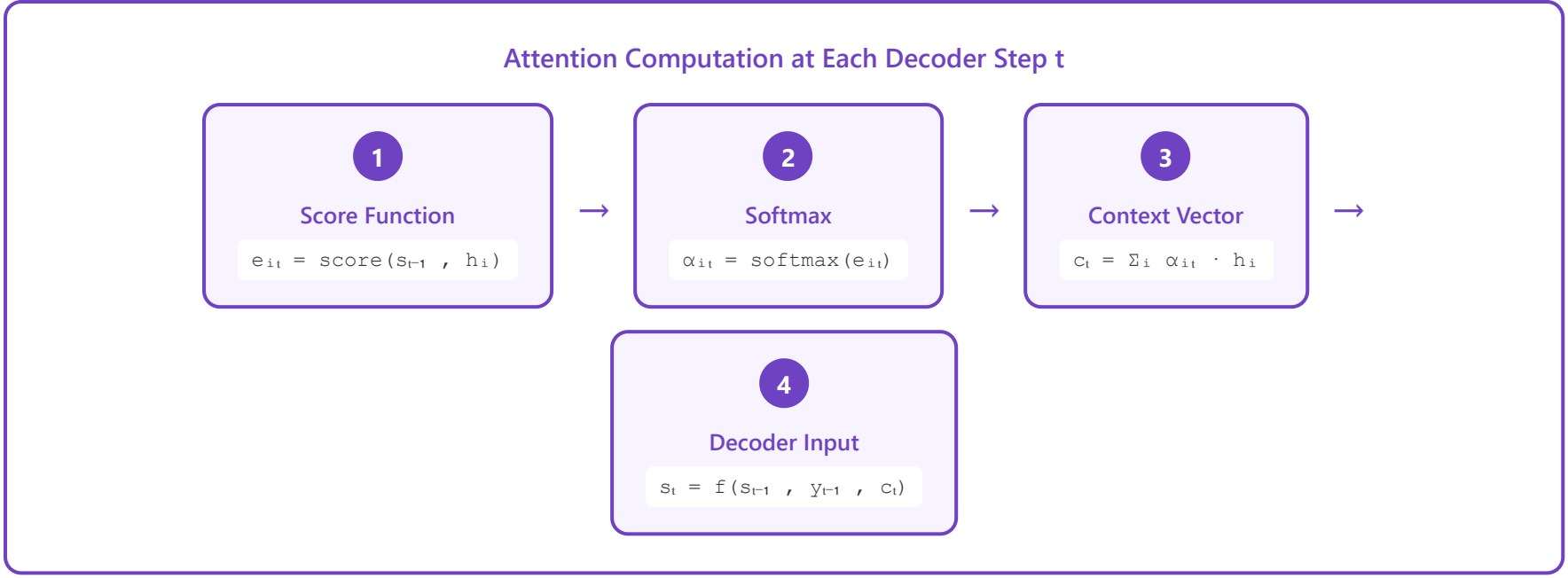
Dynamic context computed at each step. Weighted focus

🎯 Key Benefits of Attention

- ✓ No information bottleneck
- ✓ Handles long sequences
- ✓ Selective focus
- ✓ Better alignment
- ✓ Interpretable weights
- ✓ Improved performance

🔍 Where Attention is Applied in Seq2Seq Architecture



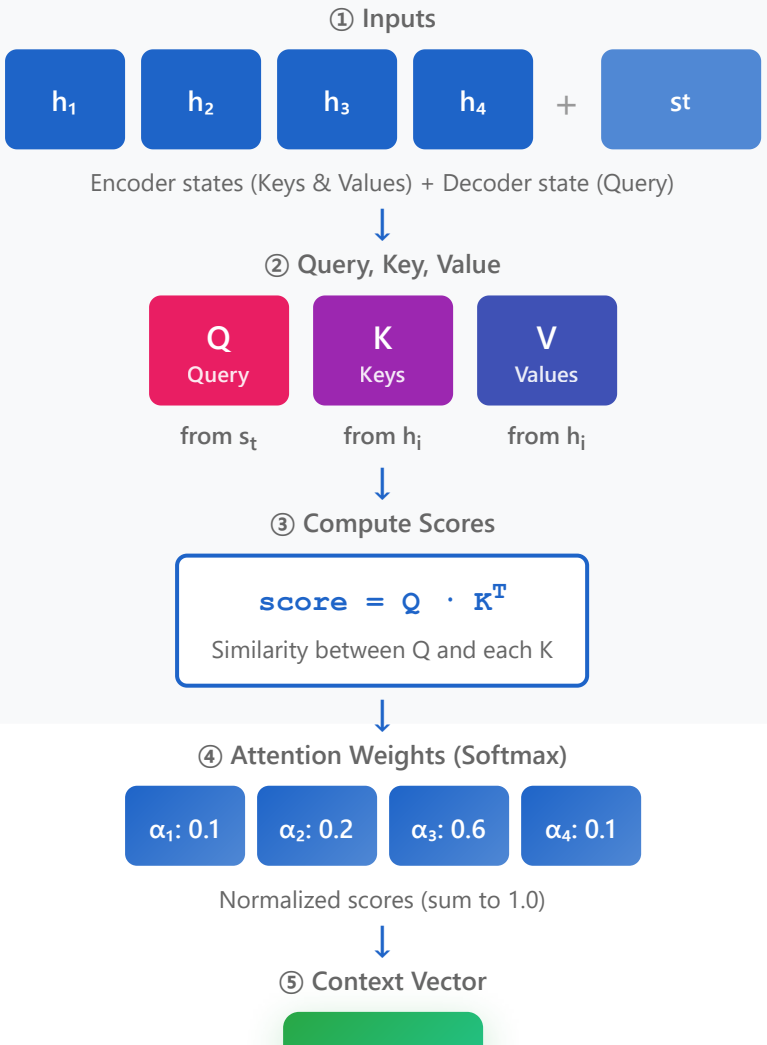


Key Insight: Attention Bridges Encoder and Decoder

Attention sits **between the encoder and decoder**, creating a dynamic connection. At each decoder time step, it computes a **new context vector** by looking at all encoder hidden states, allowing the model to focus on the most relevant input parts for generating each output token.

Attention Mechanism Structure

Step-by-Step Attention Computation



Query

"What am I looking for?" Derived from current decoder state to find relevant encoder states.

$$Q = W_Q \times s_t$$

Key

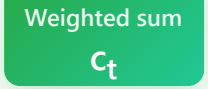
"What do I contain?" Encoder states transformed to be compared with query.

$$K = W_K \times h_i$$

Value

"What information to extract?" Actual content to be aggregated based on attention weights.

$$V = W_V \times h_i$$



$$c_t = \Sigma(\alpha_i \times V_i)$$

Attention Mechanism: Mathematical Formulas

1 Compute Attention Scores

Similarity

$$e_{t,i} = \text{score}(s_t, h_i) = s_t^T W_a h_i$$

Calculate similarity between decoder state s_t and each encoder hidden state h_i using learnable weight matrix W_a

2 Compute Attention Weights

Softmax

$$\alpha_{t,i} = \exp(e_{t,i}) / \Sigma_j \exp(e_{t,j})$$

Normalize scores using softmax to obtain attention weights that sum to 1. Higher scores → higher attention weights

3 Compute Context Vector

Weighted Sum

$$c_t = \Sigma_i \alpha_{t,i} \times h_i$$

Weighted sum of encoder hidden states using attention weights. More relevant states contribute more to the context

4 Generate Output

Prediction

$$\hat{y}_t = \text{softmax}(W_o [s_t; c_t] + b_o)$$

Concatenate decoder state and context vector, then pass through output layer to predict next token

🔑 Key Components Summary

Input Dimensions

$s_t \in \mathbb{R}^d$

$h_i \in \mathbb{R}^d$

$W_a \in \mathbb{R}^{d \times d}$

Attention Properties

$\Sigma_i \alpha_{t,i} = 1$

$\alpha_{t,i} \in [0, 1]$

$0 \leq \alpha_{t,i} \leq 1$

Output

$c_t \in \mathbb{R}^d$

Dynamic context

Per time step

s_t: Decoder hidden state h_i: Encoder hidden state $e_{t,i}$: Attention score $\alpha_{t,i}$: Attention weight c_t: Context vector

W_a: Alignment weight matrix

🧮 Concrete Example with 3D Vectors

📝 **Setup:** d = 3 (vector dimension), using 2 encoder hidden states (h_1, h_2).

1 Given Input Data

Decoder hidden state (s_t):

$$s_t = [1, 2, 1]^T$$

Encoder hidden states:

$$h_1 = [2, 0, 1]^T$$

$h_2 = [1, 1, 2]^T$

Weight matrix (W_a) $\in \mathbb{R}^{3\times3}$:

$W_a = [1\ 0\ 0]\ [0\ 1\ 0]\ [0\ 0\ 1]$

(Using Identity Matrix for simplicity)

2 Compute Attention Scores

📐 Formula: $e_{t,i} = s_t^T W_a h_i$

- **Score for h_1 :**

$e_{t,1} = s_t^T W_a h_1 = [1, 2, 1] \cdot [2, 0, 1]^T$

$= (1\times2) + (2\times0) + (1\times1)$

$= 2 + 0 + 1$

$= 3$

- **Score for h_2 :**

$e_{t,2} = s_t^T W_a h_2 = [1, 2, 1] \cdot [1, 1, 2]^T$

$= (1\times1) + (2\times1) + (1\times2)$

$= 1 + 2 + 2$

$= 5$

✓ Attention Scores:

$e_{t,1} = 3,\ e_{t,2} = 5$

3 Compute Attention Weights (Softmax)

📐 Formula: $\alpha_{t,i} = \exp(e_{t,i}) / \Sigma_j \exp(e_{t,j})$

- **Exponential calculation:**

$\exp(e_{t,1}) = \exp(3) \approx 20.09$

$\exp(e_{t,2}) = \exp(5) \approx 148.41$

- **Sum:**

$\Sigma \exp(e_{t,j}) = 20.09 + 148.41 = 168.50$

- **Attention Weights:**

$\alpha_{t,1} = 20.09 / 168.50 \approx$ **0.119**

$\alpha_{t,2} = 148.41 / 168.50 \approx$ **0.881**

- **Verification:**

$\alpha_{t,1} + \alpha_{t,2} = 0.119 + 0.881 = 1.000$ ✓

✓ Attention Weights:

$\alpha_{t,1} \approx 0.119\ (11.9\%),\ \alpha_{t,2} \approx 0.881\ (88.1\%)$

💡 **Interpretation:** h_2 receives about 88% attention because it's more similar to s_t!

4 Compute Context Vector

📐 Formula: $c_t = \Sigma_i \alpha_{t,i} \times h_i$

- **Weighted sum calculation:**

$$c_t = \alpha_{t,1} \times h_1 + \alpha_{t,2} \times h_2$$

- **First term:**

$$\alpha_{t,1} \times h_1 = 0.119 \times [2, 0, 1]^T$$

$$= [0.238, 0.000, 0.119]^T$$

- **Second term:**

$$\alpha_{t,2} \times h_2 = 0.881 \times [1, 1, 2]^T$$

$$= [0.881, 0.881, 1.762]^T$$

- **Final Context Vector:**

$$c_t = [0.238, 0.000, 0.119]^T + [0.881, 0.881, 1.762]^T$$

$$= \mathbf{[1.119, 0.881, 1.881]^T}$$

🎯 Final Context Vector:

$$c_t = [1.119, 0.881, 1.881]^T$$

✨ **Result Interpretation:** The context vector is formed closer to h_2 (88.1% weight). This means the current decoder state is more relevant to h_2.

🎓 Key Takeaways

1. Attention Score measures similarity via dot product between two vectors

2. Softmax normalizes scores into probabilities between 0 and 1

3. Context Vector is a weighted average, focusing on important information

4. High similarity → high attention weight → greater influence

The Effects of Attention Mechanism

Before Attention

All inputs treated equally

	x_1	x_2	x_3	x_4
y_1	0.25	0.25	0.25	0.25
y_2	0.25	0.25	0.25	0.25
y_3	0.25	0.25	0.25	0.25

x: Input tokens (source) **y:** Output tokens (target)

Static context: Same fixed representation used for all outputs. No selective focus on relevant information.

With Attention

Dynamic attention weights

	x_1	x_2	x_3	x_4
y_1	0.7	0.2	0.05	0.05
y_2	0.1	0.6	0.25	0.05
y_3	0.05	0.15	0.3	0.5

x: Input tokens (source) **y:** Output tokens (target)

Dynamic context: Different attention distribution for each output. Focuses on most relevant inputs at each step.

🎯 Better Alignment

Learns input-output correspondence automatically. Essential for translation tasks.

📏 Long Sequences

Handles long inputs effectively without information bottleneck. No gradient vanishing.

🔍 Interpretability

Attention weights visualize what model focuses on. Debugging and understanding easier.

Translation Quality	Long Sequences	Gradient Flow

↑ 15-20%	↑ 30-40%	✓ Stable
BLEU Score Improvement	Performance on 50+ tokens	Direct path to each input

Part 6/7:

Practical Implementation Tips

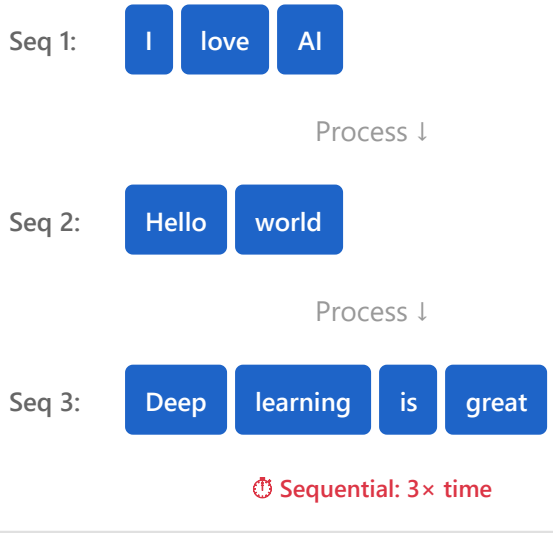
21. Batching
22. Masking
23. Practical Checklist

Batching in Sequence Models

⚠ Challenge: Variable-Length Sequences

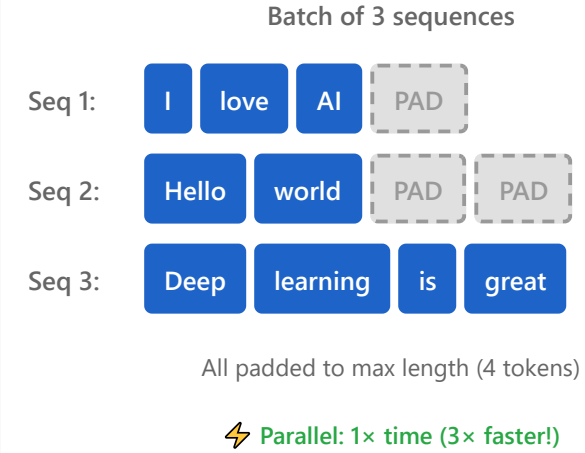
Sequences in a batch often have different lengths. We need to pad them to the same length for efficient parallel processing on GPUs.

Without Batching



Sequential processing: Process one sequence at a time. Slow and inefficient. Cannot leverage GPU parallelism.

With Batching (Padded)



Parallel processing: Process all sequences simultaneously. Fast and efficient. Full GPU utilization. Need masking!

Padding Strategy

Add special PAD tokens to shorter sequences to match the longest sequence in batch.

```
pad_sequence(sequences,
  batch_first=True,
  padding_value=0)
```

Use Masking

Create masks to ignore PAD tokens during attention computation and loss calculation.

```
mask = (input != PAD)
scores.masked_fill(
  ~mask, -inf)
```

Batch Size

Balance between memory and speed. Larger batches = faster but more memory. Typical: 32-128.

```
# Common sizes
batch_size = 32
batch_size = 64
batch_size = 128
```

Masking in Sequence Models

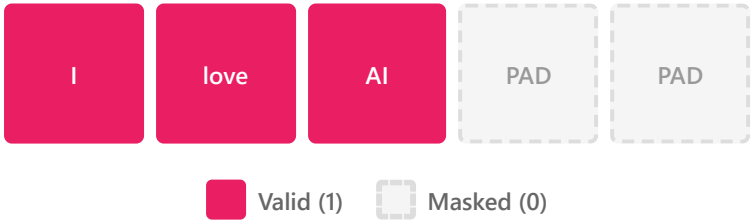
Why Do We Need Masking?

Masks prevent the model from attending to **invalid positions** like padding tokens or future tokens, ensuring correct computation and preventing information leakage.

Padding Mask

Ignore PAD tokens in attention computation

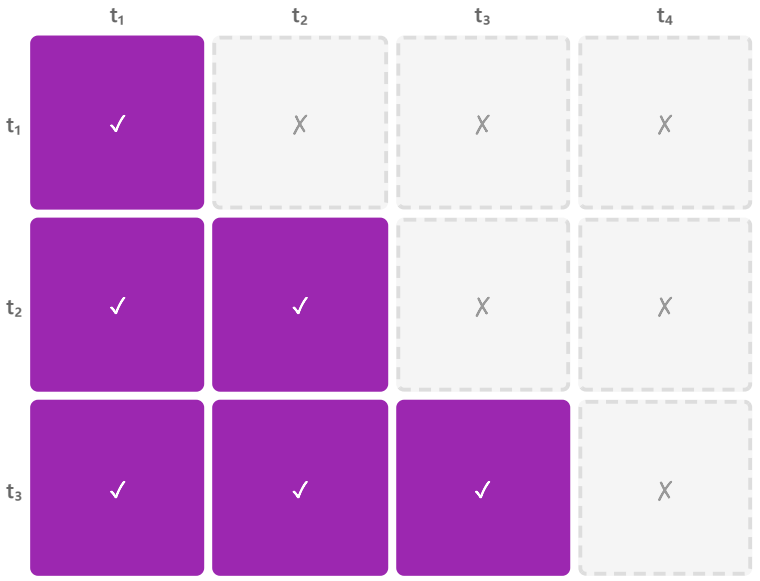
Example: Sequence with padding

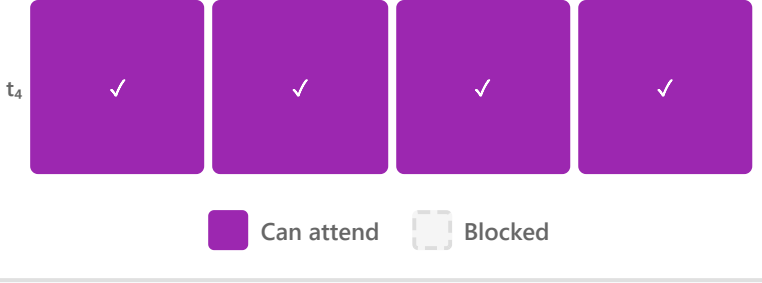


Look-ahead Mask

Prevent attending to future tokens (causal)

Attention mask matrix (lower triangular)






```
mask = (input != PAD_TOKEN)
scores.masked_fill(~mask, -∞)
```

```
mask = torch.tril(
torch.ones(seq_len, seq_len))
```

① Create Mask

```
# Padding mask
pad_mask = (x != PAD)

# Shape: (batch, seq_len)
```

② Apply to Attention

```
# Before softmax
scores = scores.masked_fill(
  ~mask, float('-inf'))
attn = softmax(scores)
```

③ Combine Masks

```
# Both padding + lookahead
mask = pad_mask &
       lookahead_mask
# Logical AND
```

Sequence Models: Practical Implementation Checklist

Essential considerations before deploying your model

Data Preparation

- [x] **Tokenization**
  Choose appropriate tokenizer (BPE, WordPiece, SentencePiece)
- [x] **Vocabulary Size**
  Balance between coverage and efficiency (typically 10K-50K)
- [x] **Special Tokens**
  Define PAD, UNK, BOS, EOS tokens correctly
- [x] **Sequence Length**
  Set max length based on data distribution (e.g., 128, 256, 512)

Model Architecture

- [x] **Hidden Dimensions**
  Choose appropriate size (256, 512, 1024) based on task
- [x] **Number of Layers**
  Balance depth and training time (2-12 layers typical)
- [x] **Attention Mechanism**
  Implement proper attention with correct masking
- [x] **Dropout Rates**
  Add dropout for regularization (0.1-0.3 typical)

Training Strategy

- [x] **Teacher Forcing**
  Use during training, consider scheduled sampling
- [x] **Batch Size**
  Optimize for GPU memory (32-128 typical)

Implementation Details

- [x] **Padding & Masking**
  Properly handle variable-length sequences
- [x] **Loss Calculation**
  Ignore padding tokens in loss computation

Learning Rate

Use warmup + decay schedule (e.g., 1e-4 to 1e-5)

Gradient Clipping

Prevent exploding gradients (clip norm 1.0-5.0)

Inference Strategy

Choose beam search, greedy, or sampling decoding

Evaluation Metrics

Track BLEU, perplexity, or task-specific metrics

Quick Reference: Typical Hyperparameters

Hidden Size	Batch Size	Vocab Size	Learning Rate
256-1024	32-128	10K-50K	1e-4 ~ 1e-3

Part 7/7:

Conclusion & Next Steps

24. Summary and Preview of Next Lecture

Advanced Sequence Models: Summary & Preview

What We Learned Today

1

Seq2Seq Architecture

- Encoder-Decoder
- Context vector
- Variable length I/O

2

RNN Variants

- LSTM cells
- GRU cells
- Gradient flow

3

Bidirectional RNN

- Forward + Backward
- Context from both
- Better encoding

4

Teacher Forcing

- Training strategy
- Exposure bias
- Scheduled sampling

5

Attention Mechanism

- Query, Key, Value
- Dynamic context
- Alignment weights

6

Implementation

- Batching & Padding
- Masking strategies
- Practical tips

Next Lecture: Transformers & Modern Architectures

- Self-Attention
- Transformer Architecture
- Multi-Head Attention
- BERT & GPT

Key Takeaway: Attention mechanism revolutionized sequence modeling by allowing models to selectively focus on relevant parts of input, solving the information bottleneck problem!

Thank you

Ho-min Park

homin.park@ghent.ac.kr

powersimmani@gmail.com