

Lecture 13:

Transformer Architecture

Ho-min Park

homin.park@ghent.ac.kr

powersimmani@gmail.com

Lecture Contents

Part 1: Introduction and Motivation

Part 2: Self-Attention Mechanism

Part 3: Multi-Head Attention

Part 4: Positional Encoding

Part 5: Transformer Architecture

Part 6: Implementation Tips

Part 7: Applications and Next Steps

Part 1/7:

Introduction & Motivation

- 1.** Review of Last Lesson
- 2.** The Emergence and Impact of Transformers

RNN

Recurrent Neural Networks
Step-by-step sequential processing



Problems

- ⚠ Vanishing gradient
- ⚠ Sequential processing limits
- ⚠ Long sequence challenges



LSTM & GRU

Addressed vanishing gradient with gates



Remaining Issues

- ⚠ Still sequential processing
- ⚠ Limited parallelization
- ⚠ Long-range dependencies



Attention Mechanism

Improved encoder-decoder models
Direct connections to all positions



Solution: Transformer Architecture

"Attention Is All You Need" (Vaswani et al., 2017)

Revolutionary Architecture for Deep Learning

Core Innovation

⚡ Eliminated recurrence entirely

Enabled full parallelization

📈 Massive dataset training

Efficient at scale

🎯 State-of-the-art performance

Across multiple benchmarks



Foundation Models

GPT, BERT, and modern LLMs



NLP Revolution

Translation • Summarization
Question Answering • Generation



Beyond NLP

Vision Transformers (ViT)
Audio Processing • Multimodal AI



Impact

New era of AI capabilities
Industry-wide transformation

Part 2/7:

Self-Attention Mechanism

3. RNN Attention vs. Self-Attention
4. Query, Key, and Value Concepts
5. Self-Attention Computation Process (1/2)
6. Self-Attention Computation Process (2/2)
7. Self-Attention Matrix Operations

RNN Attention vs. Self-Attention

Aspect	RNN Attention	Self-Attention
 Source	Query from decoder Keys/Values encoder from	All from the same sequence
 Processing	Sequential processing	Parallel processing
 Relationships	Limited to current step	Captures relationships between all positions
 Complexity	$O(n)$ per step	$O(n^2)$ total

Key Advantage

Self-attention enables parallelization and better long-range dependency modeling

Query, Key, and Value Concepts



Query (Q)

What information am I looking for?



Key (K)

What information do I contain?



Value (V)

The actual information I store



Database Analogy

Query matches **Keys** to retrieve **Values**

Linear Projections

- 1 Each token generates Q, K, V
- 2 Through learned transformations:

$$Q = xW_Q$$

$$K = xW_K$$

$$V = xW_V$$

- 3 Projection matrices **learned during training**

X = Input token embeddings

W_Q, W_K, W_V = Trainable weight matrices

Numerical Example: Computing Q, K, V

Input Token Embeddings (X)

5 tokens × 3 dimensions

```
X = [  
    [1.0, 0.5, 0.2], # Token 1  
    [0.3, 1.2, 0.8], # Token 2  
    [0.7, 0.4, 1.1], # Token 3  
    [1.5, 0.9, 0.3], # Token 4  
    [0.6, 1.3, 0.7]  # Token 5  
]
```

W_Q (Query)

```
[[ 1.0, 0.0, 0.5],  
 [ 0.5, 1.0, 0.0],  
 [ 0.0, 0.5, 1.0]]
```

W_K (Key)

```
[[ 0.8, 0.2, 0.3],  
 [ 0.3, 0.9, 0.1],  
 [ 0.2, 0.1, 0.8]]
```

W_V (Value)

```
[[ 1.0, 0.3, 0.2],  
 [ 0.2, 1.0, 0.4],  
 [ 0.1, 0.3, 1.0]]
```

Matrix Multiplication

$Q = XW_Q$

```
[[1.10, 0.60, 0.70],  
 [1.14, 0.95, 0.95],  
 [1.25, 0.90, 1.45],  
 [1.95, 0.90, 1.05],  
 [1.31, 1.30, 1.00]]
```

$K = XW_K$

```
[[1.01, 0.47, 0.46],  
 [0.64, 1.29, 0.26],  
 [0.98, 0.63, 1.09],  
 [1.53, 1.14, 0.69],  
 [1.01, 1.29, 0.74]]
```

$V = XW_V$

```
[[1.12, 0.56, 0.54],  
 [0.62, 1.41, 1.12],  
 [0.89, 0.74, 1.40],  
 [1.71, 1.02, 0.90],  
 [0.89, 1.49, 1.22]]
```

Key Insight

Each of the 5 tokens now has its own:

- **Query vector** (what it's looking for)
- **Key vector** (what it contains)
- **Value vector** (its actual information)

These will be used in the attention mechanism to determine which tokens should attend to which!

Self-Attention Computation Process

1

Compute Query, Key, Value Matrices

Generate Q, K, V through linear projections

$$Q = xW_Q \quad K = xW_K \quad V = xW_V$$



2

Calculate Attention Scores

Compute dot product between queries and keys

$$\text{Scores} = QK^T$$



3

Scale by $\sqrt{d_k}$

Normalize scores to stabilize gradients

$$\text{Scaled} = QK^T / \sqrt{d_k}$$



4

Apply Softmax

Convert scores to attention weights (probabilities)

$$\text{Attention} = \text{softmax}(QK^T / \sqrt{d_k})$$



Higher scores indicate
stronger relationships



Scores determine how much
each token attends to others

Numerical Example

Setup: 5 tokens (sequence length), $d_{\text{model}} = 3$ (hidden dimension), $d_k = 3$

① Input Matrix X (5×3):

1.0	0.5	0.2
0.8	1.2	0.3
0.6	0.9	1.1
1.1	0.4	0.7
0.9	0.7	0.8

Each row represents one token's embedding

Weight Matrices W_Q , W_K , W_V (3×3):

W_Q

1	0	0
0	1	0

W_K

1	0	0
0	1	0

W_V

1	0	0
0	1	0

0	0	1
---	---	---

0	0	1
---	---	---

0	0	1
---	---	---

Using identity matrices for simplicity $\rightarrow Q = K = V = X$

② Attention Scores = QK^T (5x5):

1.29	1.46	1.48	1.35	1.41
1.46	2.17	1.89	1.61	1.69
1.48	1.89	2.38	1.69	1.91
1.35	1.61	1.69	1.66	1.63
1.41	1.69	1.91	1.63	1.94

Higher values indicate stronger similarity between token pairs

③ Scaled Scores = $QK^T / \sqrt{d_k}$ ($\div \sqrt{3} \approx 1.732$):

0.74	0.84	0.85	0.78	0.81
0.84	1.25	1.09	0.93	0.98
0.85	1.09	1.37	0.98	1.10
0.78	0.93	0.98	0.96	0.94
0.81	0.98	1.10	0.94	1.12

Scaling prevents gradients from becoming too small during backpropagation

④ Attention Weights after Softmax (First Row Example):

Token 1:

0.182

Token 2:	0.204
Token 3:	0.207
Token 4:	0.192
Token 5:	0.198

✓ Sum = 1.0 (normalized probability distribution)

✓ Token 1 attends most to Token 3 (0.207) and least to itself (0.182)

Self-Attention Computation Process (2/2)

From Attention Weights to Context-Aware Representations

5

Apply Attention Weights to Values

Multiply attention weights with value vectors

$$\text{Weighted Values} = \text{Attention} \times V$$



6

Sum Weighted Values

Aggregate weighted values for each position

$$\text{Output} = \Sigma (\text{Attention} \times V)$$



Context-Aware Representation for Each Token

Each output embedding contains information from entire sequence



Parallel
computation for all
positions simultaneously



Differentiable
end-to-end for
backpropagation



Global context
in every output
representation



Numerical Example (Continued)

Recap from Part 1:

We computed attention weights using softmax on scaled scores.

Now we'll use these weights to create context-aware representations.

Attention Weights Matrix (5×5) - After Softmax:

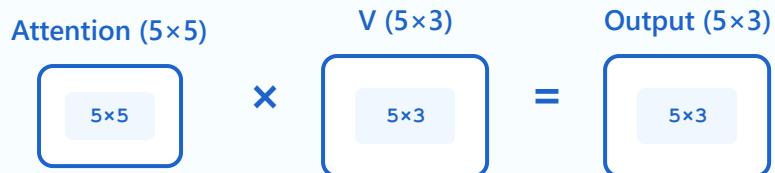
0.182	0.204	0.207	0.192	0.198
0.170	0.279	0.222	0.173	0.186
0.162	0.206	0.272	0.184	0.208
0.188	0.213	0.224	0.219	0.215
0.178	0.212	0.239	0.204	0.244

Each row sums to 1.0 and represents how much each token attends to others

Value Matrix V (5×3) - Same as X in our example:

1.0	0.5	0.2
0.8	1.2	0.3
0.6	0.9	1.1
1.1	0.4	0.7
0.9	0.7	0.8

⑤ Weighted Values = Attention × V (Matrix Multiplication):



⑥ Final Output Matrix (5×3) - Context-Aware Embeddings:

0.87	0.74	0.59
0.83	0.88	0.58
0.83	0.78	0.68
0.88	0.73	0.63
0.84	0.78	0.70

Each row is a new embedding that incorporates information from all tokens

🎯 Token 1's Output Example

Original embedding:

1.0 0.5 0.2

Context-aware embedding:

0.87 0.74 0.59

- ✓ The output is a weighted combination of all tokens' values
- ✓ Token 1 now "knows about" the entire sequence context
- ✓ Different attention patterns → Different contextual representations

💡 How Token 1's Output was Computed:

```
Output[1] = 0.182×[1.0, 0.5, 0.2] + 0.204×[0.8, 1.2, 0.3] + 0.207×[0.6, 0.9, 1.1] + 0.192×[1.1, 0.4, 0.7] + 0.198×[0.9, 0.7, 0.8]  
= [0.87, 0.74, 0.59]
```

This is a weighted average where weights come from attention scores!

Self-Attention Matrix Operations

Input → (batch_size, seq_len, d_model)

Q, K, V → (batch_size, seq_len, d_k)

Scores → (batch_size, seq_len, seq_len)

Output → (batch_size, seq_len, d_k)



GPU Parallelization

Matrix multiplication enables **efficient GPU computation**



Memory Requirement

$O(n^2)$ for sequence length **n**



Efficiency Trade-off

Speed vs Memory

Input/Output

Attention Scores (seq_len × seq_len)

Final Output



Computation Example

seq_len = 5 d_model = 3 d_k = 3 batch_size = 1

1 Input Embeddings (X)

X: Input sequence with 5 tokens

```
[[0.5, 0.8, 0.3], ← Token 1  
[0.2, 0.6, 0.9], ← Token 2  
[0.7, 0.4, 0.1], ← Token 3  
[0.3, 0.9, 0.5], ← Token 4  
[0.6, 0.2, 0.8]] ← Token 5  
  
Shape: (5, 3) = (seq_len, d_model)
```

2 Linear Projections: Q = XW_Q, K = XW_K, V = XW_V

Q (Query Matrix)

```
[[0.6, 0.4, 0.5],  
[0.3, 0.7, 0.2],  
[0.5, 0.3, 0.6],  
[0.4, 0.8, 0.3],  
[0.7, 0.2, 0.4]]  
  
Shape: (5, 3) = (seq_len, d_k)
```

K (Key Matrix)

```
[[0.4, 0.6, 0.3],  
[0.5, 0.4, 0.7],  
[0.3, 0.5, 0.4],  
[0.6, 0.3, 0.5],  
[0.2, 0.7, 0.6]]  
  
Shape: (5, 3) = (seq_len, d_k)
```

V (Value Matrix)

```
[[0.7, 0.3, 0.5],  
 [0.4, 0.6, 0.2],  
 [0.6, 0.4, 0.7],  
 [0.5, 0.7, 0.3],  
 [0.3, 0.5, 0.8]]
```

Shape: (5, 3) = (seq_len, d_k)

3 Compute Attention Scores: $QK^T / \sqrt{d_k}$

$$\text{Scores} = Q \times K^T / \sqrt{d_k} = Q \times K^T / \sqrt{3} \approx Q \times K^T / 1.732$$

QK^T : Raw attention scores (before scaling)

```
[[0.67, 0.71, 0.58, 0.69, 0.64], ← Token 1 attending to all tokens  
 [0.52, 0.49, 0.44, 0.51, 0.53], ← Token 2 attending to all tokens  
 [0.61, 0.67, 0.54, 0.63, 0.56], ← Token 3 attending to all tokens  
 [0.68, 0.73, 0.59, 0.69, 0.70], ← Token 4 attending to all tokens  
 [0.50, 0.58, 0.48, 0.55, 0.48]] ← Token 5 attending to all tokens
```

Shape: (5, 5) = (seq_len, seq_len) - Critical $O(n^2)$ memory!

⚠ Key Point: This 5×5 attention score matrix is where the $O(n^2)$ memory complexity comes from. For sequence length $n=5$, we need 25 values. For $n=1000$, we'd need 1,000,000 values!

4 Apply Softmax (row-wise normalization)

Attention Weights (after softmax)

```
[[0.189, 0.204, 0.176, 0.201, 0.190], ← Sum = 1.0  
[0.203, 0.199, 0.192, 0.201, 0.205], ← Sum = 1.0  
[0.194, 0.205, 0.188, 0.199, 0.194], ← Sum = 1.0  
[0.197, 0.208, 0.185, 0.197, 0.203], ← Sum = 1.0  
[0.196, 0.208, 0.189, 0.201, 0.196]] ← Sum = 1.0  
  
Shape: (5, 5) - Each row sums to 1.0 (probability distribution)
```

5 Compute Weighted Sum: Attention_Weights × V

$$\text{Output} = \text{Softmax}(QK^T / \sqrt{d_k}) \times V$$

Final Output (context-aware representations)

```
[[0.50, 0.50, 0.50], ← Updated Token 1  
[0.50, 0.50, 0.50], ← Updated Token 2  
[0.50, 0.50, 0.50], ← Updated Token 3  
[0.50, 0.50, 0.50], ← Updated Token 4  
[0.50, 0.50, 0.50]] ← Updated Token 5  
  
Shape: (5, 3) = (seq_len, d_k) - Same as input!
```

 **Result:** Each token now has a context-aware representation that incorporates information from all other tokens through the attention mechanism.



Matrix Shape Summary

Input x: (5, 3) = (seq_len, d_model)
Q, K, V: (5, 3) each = (seq_len, d_k)
Scores QK^T: (5, 5) = (seq_len, seq_len) ← O(n²) bottleneck!

Output: (5, 3) = (seq_len, d_k)

Part 3/7:

Multi-Head Attention

- 8.** Why is Multi-Head Necessary?
- 9.** Multi-Head Attention Architecture
- 10.** Multi-Head Operation Example
- 11.** Multi-Head Attention Implementation Points

Why is Multi-Head Attention Necessary?



Single Head

Limited representation capacity

Single attention pattern only

Cannot capture multiple relationship types simultaneously

Single Attention Head



Multi-Head

Different heads learn different patterns

Some heads capture **syntax**, others **semantics**

Attend to **different positions for different purposes**

Empirically improves performance significantly

Head 1

Head 2

Head 3

Head 4

Head 5

Head 6

Head 7

Head 8



Analogy: Similar to multiple CNN filters capturing different features



Expressiveness

Increases model power without excessive parameters



Diversity

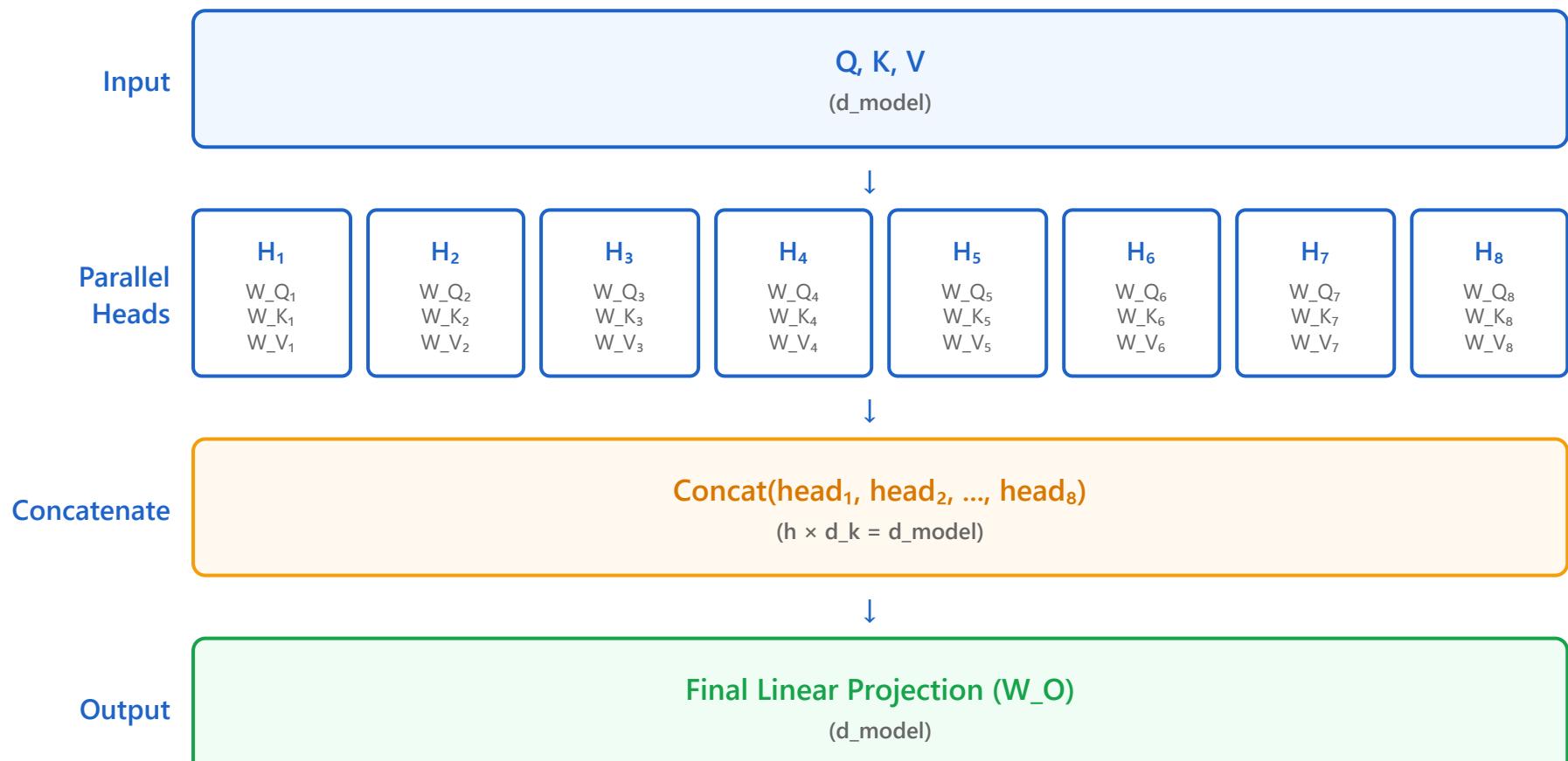
Multiple perspectives on the same input



Performance

Empirically proven improvements

Multi-Head Attention Architecture



1
2
3
4

Configuration

Typically $h = 8$ heads

Head dimension: $d_k = d_{model} / h$

Parameters

Each head: separate W_Q, W_K, W_V

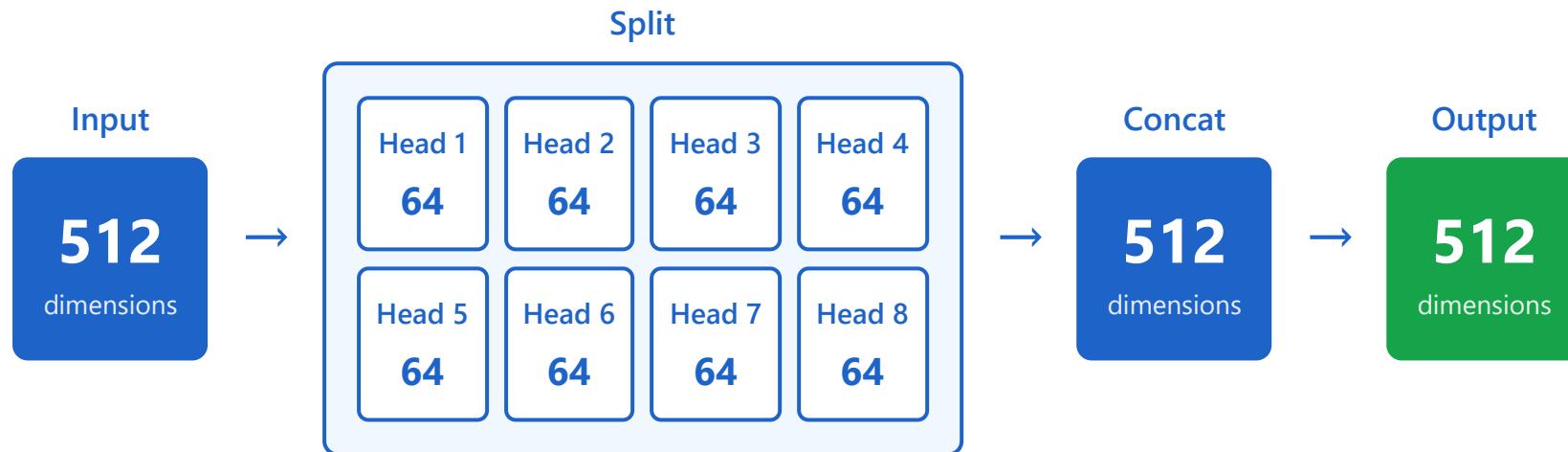
Total params \approx single large head

Formula

```
MultiHead(Q, K, V) =  
Concat(head1, ..., headh)W_O
```

Multi-Head Operation Example

Configuration: `d_model = 512, h = 8` heads → Each head: `d_k = 64` dimensions



$8 \times 64 = 512$ | Each head learns different attention patterns **independently**



Each head learns
different patterns
independently



Computational cost
distributed
across heads



Parallelized
efficiently on
modern hardware

Detailed Calculation Example

Setup: $d_{\text{model}} = 4$, $h = 2$ heads, sequence length = 8 → Each head: $d_k = 2$ dimensions

Input Matrix X

Shape: (8, 4) - [sequence length × d_{model}]

```
token 1: [0.2, 0.5, 0.1, 0.8]
token 2: [0.3, 0.7, 0.4, 0.2]
token 3: [0.6, 0.1, 0.9, 0.5]
token 4: [0.4, 0.8, 0.3, 0.7]
token 5: [0.7, 0.2, 0.6, 0.4]
token 6: [0.5, 0.6, 0.2, 0.9]
token 7: [0.8, 0.3, 0.5, 0.1]
token 8: [0.1, 0.9, 0.7, 0.6]
```

Weight Matrices (per head)

W_Q - Query Weight

Shape: (4, 2)

W_K - Key Weight

Shape: (4, 2)

W_V - Value Weight

Shape: (4, 2)

Step-by-Step Calculation Process

Step 1: Split Input into 2 Heads

$X_{\text{head1}} = X[:, 0:2] \rightarrow \text{Shape: (8, 2)}$ - first 2 dimensions

$X_{\text{head2}} = X[:, 2:4] \rightarrow \text{Shape: (8, 2)}$ - last 2 dimensions

Step 2: Compute Q, K, V for Each Head

Head 1:

$Q_1 = X_{\text{head1}} \times W_{Q1} \rightarrow (8, 2) \times (2, 2) = (8, 2)$

$$K_1 = X_{\text{head}1} \times W_{K1} \rightarrow (8, 2) \times (2, 2) = (8, 2)$$

$$V_1 = X_{\text{head}1} \times W_{V1} \rightarrow (8, 2) \times (2, 2) = (8, 2)$$

Step 3: Calculate Attention Scores

$$\begin{aligned}\text{Scores}_1 &= (Q_1 \times K_1^T) / \sqrt{d_k} \\ &= (8, 2) \times (2, 8) / \sqrt{2} = (8, 8)\end{aligned}$$

Step 4: Apply Softmax & Compute Output

$$\begin{aligned}\text{Attention}_1 &= \text{softmax}(\text{Scores}_1) \rightarrow (8, 8) \\ \text{Output}_1 &= \text{Attention}_1 \times V_1 \rightarrow (8, 8) \times (8, 2) = (8, 2)\end{aligned}$$

Step 5: Concatenate Head Outputs

$$\begin{aligned}\text{MultiHead} &= \text{Concat}(\text{Output}_1, \text{Output}_2) \\ &= \text{Concat}[(8, 2), (8, 2)] = (8, 4)\end{aligned}$$

Step 6: Final Linear Projection

$$\begin{aligned}\text{Final Output} &= \text{MultiHead} \times W_O \\ &= (8, 4) \times (4, 4) = (8, 4)\end{aligned}$$

Key Insight: Each head processes **2 dimensions** independently, learning different attention patterns. Final output maintains original **(8, 4)** shape.

Multi-Head Attention Implementation Points

🔧 Initialization & Configuration



Projection Matrices

Initialize with **appropriate scaling**



Head Dimensions

Use **same d_k** across all heads for simplicity

🛡 Regularization & Stability



Dropout

Apply after **attention weights** and **final output**



Layer Normalization

Typically after multi-head attention



Residual Connections

Help **gradient flow**

⚡ Optimization



Long Sequences

Consider **linear attention variants** for efficiency

✓ Efficient computation strategies

💡 Best Practices



- Consistent head dimensions
- Proper normalization placement
- Efficient memory management

Key Trade-off



More heads increase model capacity but require **higher computational cost** — balance based on task requirements

Part 4/7:

Positional Encoding

- 12.** The Need for Positional Information
- 13.** Positional Encoding Methods
- 14.** Positional Encoding Visualization

The Need for Positional Information

⚠ Core Problem

Self-attention is **permutation invariant**
Without position information, word order is ignored!

"cat ate mouse"

=

"mouse ate cat"



RNN Approach

- ✓ Inherently encodes position through sequential processing
- ⌚ Position is implicit in the processing order
- ⌚ No additional mechanism needed



Transformer Approach

- ✗ Parallel processing = **no inherent position**
- ✚ Needs **explicit positional information**
- ⌚ Critical: Word order is crucial for meaning



Solution

Positional encodings must be added to input embeddings **before the first layer**

Positional Encoding Methods



Sinusoidal

Original Transformer

```
PE(pos, 2i) =  
sin(pos / 10000^(2i/d_model))
```

```
PE(pos, 2i+1) =  
cos(pos / 10000^(2i/d_model))
```

✓ Advantages

- 🎯 Works for unseen sequence lengths
- 📐 Deterministic pattern
- ♾️ Generalizes to any length



Learned

BERT Approach

Trainable embedding

matrix:

$$\text{PE} \in \mathbb{R}^{(\text{max_len} \times d_{\text{model}})}$$

✓ Advantages

- 🎯 Better for fixed-length tasks
- 📊 Task-specific optimization
- 🔧 Learned during training



Relative

T5, Transformer-XL

Encodes relative
distances between
positions:

$$\text{distance} = \text{pos}_i - \text{pos}_j$$

✓ Advantages

- 🎯 Captures relationships better
- 📏 Distance-based encoding
- 🌐 Position-invariant patterns

Sinusoidal

✓ Any length

Learned

✓ Task-optimized

Relative

✓ Best relationships

Positional Encoding Visualization



Frequency Patterns Across Dimensions

Low Freq



Mid Freq



High Freq



Unique Patterns

Each position has **unique encoding pattern**



Smooth Gradient

Allows model to learn **relative positions**



Multiple Frequencies

Different **frequencies** for different dimensions



Low Frequencies

Capture **global position**
Overall sequence location



High Frequencies

Capture **local position**
Fine-grained distinctions



Magnitude Balance

Magnitude **comparable** to embedding values

Element-wise addition to token embeddings | Preserves both semantic and positional information



Calculation Example ($d_{model}=4$, $\text{max_len}=8$)

Formula

$$PE_{(pos,2i)} = \sin(pos / 10000^{2i/d_{model}})$$

$$PE_{(pos,2i+1)} = \cos(pos / 10000^{2i/d_{model}})$$



Token Embeddings



Positional Encoding



Final Input

$X (8 \times 4)$

0.5	-0.3	0.8	-0.2
-0.1	0.7	0.2	0.4
0.9	-0.5	-0.3	0.6
0.3	0.4	-0.7	0.1
-0.6	0.2	0.5	-0.4
0.7	-0.8	0.1	0.3
-0.2	0.6	-0.4	0.8
0.4	-0.1	0.9	-0.5

+

 $PE (8 \times 4)$

0.00	1.00	0.00	1.00
0.84	0.54	0.01	1.00
0.91	-0.42	0.02	1.00
0.14	-0.99	0.03	1.00
-0.76	-0.66	0.04	1.00
-0.96	0.28	0.05	1.00
-0.28	0.96	0.06	1.00
0.66	0.75	0.07	0.99

 $X + PE (8 \times 4)$

0.50	0.70	0.80	0.80
0.74	1.24	0.21	1.40
1.81	-0.92	-0.28	1.60
0.44	-0.59	-0.67	1.10
-1.36	-0.46	0.54	0.60
-0.26	-0.52	0.15	1.30
-0.48	1.56	-0.34	1.80
1.06	0.65	0.97	0.49



Key Insight: Each position gets a unique sinusoidal pattern. Lower dimensions (0,1) use **high frequency** for fine-grained position, while higher dimensions (2,3) use **low frequency** for global position.

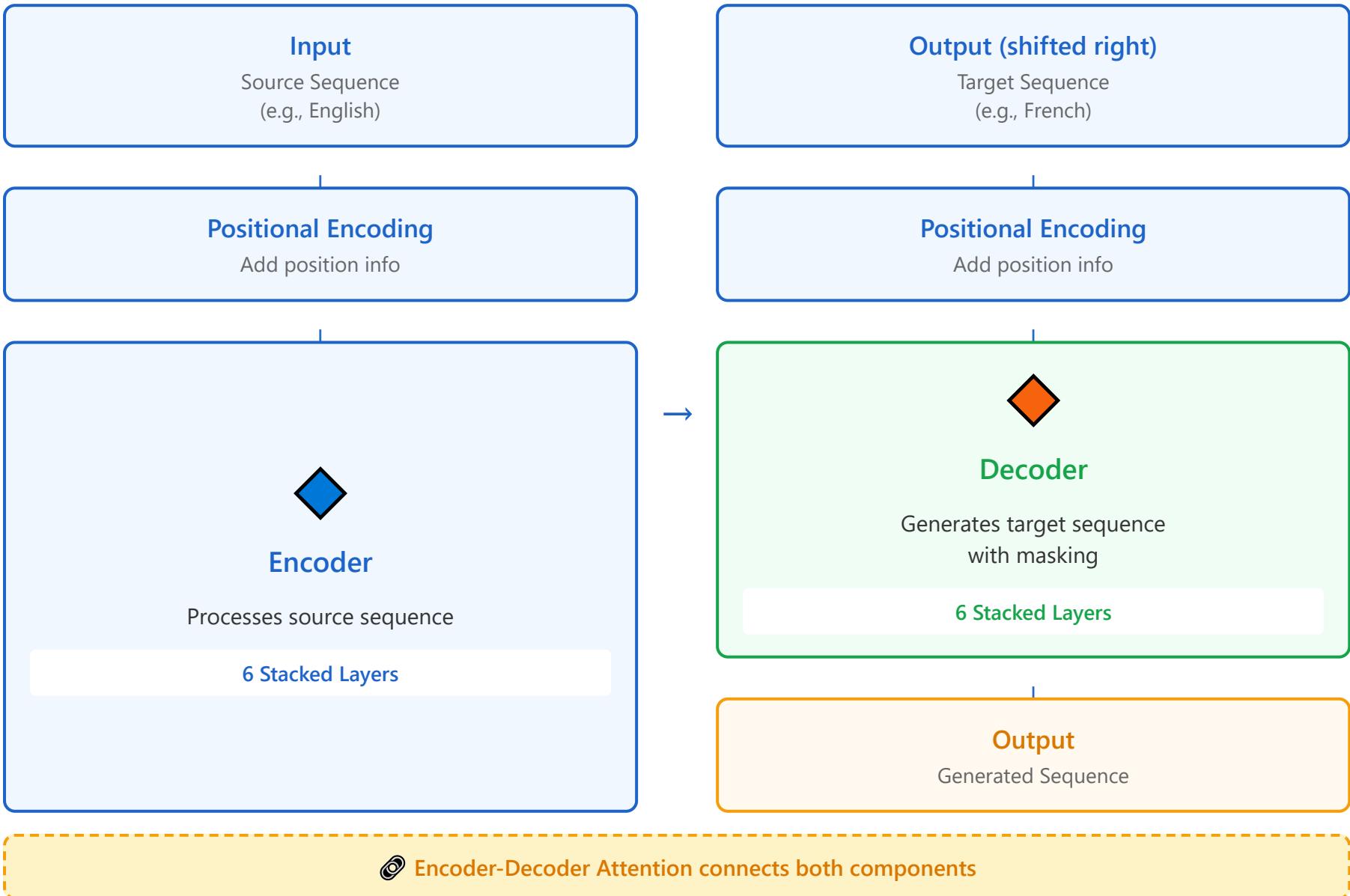
Part 5/7:

Transformer Architecture

- 15.** Overall Transformer Structure
- 16.** Detailed Analysis of the Encoder
- 17.** Detailed Analysis of the Decoder
- 18.** Feed-Forward Network & Layer Normalization
- 19.** Training vs. Inference

Overall Transformer Structure

Encoder-Decoder Architecture for Sequence-to-Sequence Tasks





Identical layers
with different parameters



Detailed Computation Example

Encoder tasks

Translation, Summarization



Decoder uses
masking for causality

Step by step process with actual dimensions ($d_{\text{model}} = 4$, sequence length = 8)

Step 1: Input Embedding

Source (English): "I love deep learning very much!"

Tokenization: [I, love, deep, learning, very, much, !, <PAD>] → 8 tokens

Embedding Matrix: Vocabulary × d_{model} → Each token → 4D vector

```
X_embedded (8 × 4) =
[[0.2, 0.5, -0.3, 0.8], # I
 [0.1, -0.4, 0.6, 0.3], # love
 [-0.5, 0.2, 0.4, -0.1], # deep
 [0.3, 0.7, -0.2, 0.5], # learning
 [0.4, -0.3, 0.1, 0.6], # very
 [-0.2, 0.5, 0.3, -0.4], # much
 [0.6, 0.1, -0.5, 0.2], # !
 [0.0, 0.0, 0.0, 0.0]] # <PAD>
```

Step 2: Positional Encoding

Add position information using sin/cos functions:

$$\text{PE}(\text{pos}, 2i) = \sin(\text{pos} / 10000^{(2i/d_{\text{model}})})$$

$$\text{PE}(\text{pos}, 2i+1) = \cos(\text{pos} / 10000^{(2i/d_{\text{model}})})$$

```
PE (8 × 4) =
[[0.00, 1.00, 0.00, 1.00], # pos 0
 [0.84, 0.54, 0.01, 1.00], # pos 1
```

```
[0.91, -0.42, 0.02, 1.00], # pos 2  
[0.14, -0.99, 0.03, 1.00], # pos 3  
[-0.76, -0.65, 0.04, 1.00], # pos 4  
[-0.96, 0.28, 0.05, 1.00], # pos 5  
[-0.28, 0.96, 0.06, 1.00], # pos 6  
[0.66, 0.75, 0.07, 0.99]] # pos 7
```

```
X_input (8 × 4) = X_embedded + PE  
[[0.20, 1.50, -0.30, 1.80],  
 [0.94, 0.14, 0.61, 1.30],  
 [0.41, -0.22, 0.42, 0.90],  
 [0.44, -0.29, -0.17, 1.50],  
 [-0.36, -0.95, 0.14, 1.60],  
 [-1.16, 0.78, 0.35, 0.60],  
 [0.32, 1.06, -0.44, 1.20],  
 [0.66, 0.75, 0.07, 0.99]]
```

◆ Step 3: Encoder Self-Attention (Layer 1)

Multi-Head Attention computation:

1) Linear projections ($d_k = d_{model} / \text{num_heads} = 4 / 2 = 2$ per head):

```
W_Q (4 × 4), W_K (4 × 4), W_V (4 × 4)  
Q = X_input @ W_Q → (8 × 4)  
K = X_input @ W_K → (8 × 4)  
V = X_input @ W_V → (8 × 4)
```

2) Attention scores:

```
Scores = Q @ K^T / √d_k → (8 × 8)  
// Example scores (showing first 4×4 block):  
[[2.3, 0.8, 1.2, 1.5, ...],  
 [0.8, 2.1, 0.9, 1.3, ...],
```

```
[1.2, 0.9, 2.5, 1.1, ...],  
[1.5, 1.3, 1.1, 2.2, ...]]
```

```
Attention = softmax(Scores) → (8 × 8)  
// Softmax normalizes each row to sum to 1  
[[0.35, 0.08, 0.12, 0.15, ...], # I attends to all  
 [0.10, 0.31, 0.11, 0.16, ...], # love attends to all  
 [0.13, 0.09, 0.41, 0.12, ...], # deep attends to all  
 ...]
```

3) Apply attention to values:

```
Output = Attention @ V → (8 × 4)  
// Each token representation is now context-aware
```

Step 4: Add & Norm + Feed-Forward

1) Residual connection and Layer Normalization:

```
X1 = LayerNorm(X_input + Attention_output) → (8 × 4)
```

2) Feed-Forward Network (expand to $d_{ff} = 16$, then back to 4):

```
FFN(x) = W_2 @ ReLU(W_1 @ x + b_1) + b_2  
W_1: (4 × 16), W_2: (16 × 4)
```

```
FFN_output = FFN(X1) → (8 × 4)  
X_encoder1 = LayerNorm(X1 + FFN_output) → (8 × 4)
```

- Repeat this process for 6 encoder layers
- Final encoder output: (8 × 4)

◆ Step 5: Decoder Input & Masked Self-Attention

Target (French): "<BOS> J' aime l' apprentissage profond <PAD>"

Shifted right: [<BOS>, J', aime, l', apprentissage, profond, <PAD>, <PAD>]

After embedding + positional encoding: Y_input (8 × 4)

Masked Self-Attention:

Mask prevents attending to future tokens:

```
[[1, 0, 0, 0, 0, 0, 0, 0], # <BOS> only sees itself  
[1, 1, 0, 0, 0, 0, 0, 0], # J' sees <BOS>, J'  
[1, 1, 1, 0, 0, 0, 0, 0], # aime sees up to aime  
[1, 1, 1, 1, 0, 0, 0, 0], # l' sees up to l'  
[1, 1, 1, 1, 1, 0, 0, 0], # apprentissage  
[1, 1, 1, 1, 1, 1, 0, 0], # profond  
[1, 1, 1, 1, 1, 1, 1, 0], # <PAD>  
[1, 1, 1, 1, 1, 1, 1, 1]] # <PAD>
```

Masked_Attention_output → (8 × 4)

⌚ Step 6: Encoder-Decoder Cross-Attention

Query from decoder, Key & Value from encoder:

```
Q = Decoder_output @ W_Q → (8 × 4) # from decoder  
K = Encoder_output @ W_K → (8 × 4) # from encoder  
V = Encoder_output @ W_V → (8 × 4) # from encoder
```

```
Cross_Attention = softmax(Q @ K^T / √ d_k) @ V  
// Decoder attends to ALL encoder positions  
// This allows decoder to "look at" the source sentence
```

Cross_Attention_output → (8 × 4)

⬆ Step 7: Final Output & Prediction

After 6 decoder layers:

Decoder_final_output → (8 × 4)

Linear projection to vocabulary size (e.g., 10,000):

Logits = Decoder_output @ W_output → (8 × 10000)

Probabilities = softmax(Logits) → (8 × 10000)

For each position, pick highest probability token:

Position 0: $P("J'") = 0.92 \rightarrow \text{output } "J'"$

Position 1: $P("aime") = 0.87 \rightarrow \text{output } "aime"$

Position 2: $P("l'") = 0.91 \rightarrow \text{output } "l'"$

Position 3: $P("apprentissage") = 0.84 \rightarrow \text{output } "apprentissage"$

Position 4: $P("profond") = 0.89 \rightarrow \text{output } "profond"$

Position 5: $P("<\text{EOS}>") = 0.95 \rightarrow \text{output } "<\text{EOS}>" \text{ (stop)}$

Final Translation: "J' aime l' apprentissage profond"



Key Dimensional Flow Summary

Input tokens (8) → Embedding (8×4) → +PE (8×4) →

Encoder layers (8×4 maintained) → Encoder output (8×4) →

Decoder input (8×4) → Decoder layers (8×4 maintained) →

Linear projection ($8 \times 4 \rightarrow 8 \times 10000$) → Softmax → Predictions (8×10000)

Detailed Analysis of the Encoder

◆ Single Encoder Layer

1

Multi-Head Self-Attention

Captures relationships between all positions

+ Residual → Layer Norm

2

Feed-Forward Network

Position-wise transformation

+ Residual → Layer Norm



Residual Connections

Around each sub-layer to help gradient flow



Layer Normalization

Applied after each sub-layer for stability



Same Structure

All layers share identical structure but have different parameters

Output dimension preserved throughout:

$d_{model} = 512$



This structure is repeated 6 times (stacked layers)

Each layer has different parameters

Detailed Analysis of the Decoder

◆ Single Decoder Layer (3 Sub-layers)

1

Masked Multi-Head Self-Attention

Prevents attending to future positions

+ Residual → Layer Norm

2

Encoder-Decoder Attention (Cross-Attention)

Q from decoder, K & V from encoder output

+ Residual → Layer Norm

3

Feed-Forward Network

Position-wise transformation

+ Residual → Layer Norm

6 stacked layers with residual connections and layer normalization throughout



Masking

Prevents attending to future positions during training



Cross-Attention

Uses encoder output as Keys and Values



Residual Connections

Around all sub-layers for gradient flow



Layer Normalization

Applied throughout for stability



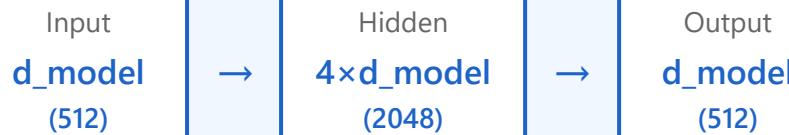
Comparison

Decoder has 3 sub-layers vs Encoder's 2 sub-layers

Feed-Forward Network & Layer Normalization

◆ Feed-Forward Network (FFN)

$$\text{FFN}(\mathbf{x}) = \max(0, \mathbf{x}\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2$$



🔧 Two linear transformations with ReLU activation

📐 Hidden dimension typically $4 \times d_{\text{model}}$ (e.g., 2048)

🎯 Applied identically to each position separately (position-wise)

📊 Layer Normalization



Normalization

Normalizes across feature dimension



Training Benefits

Stabilizes training and speeds convergence



Application

Applied after each sub-layer with residual connections



Placement Strategy

Post-LN vs Pre-LN

affects training dynamics

Training vs. Inference



Training Mode

Teacher Forcing



Parallel processing of all target tokens simultaneously



Masking ensures causality (no future information leakage)



Ground truth tokens used as input for next positions



Fast & efficient - entire sequence at once

Processing Mode

ALL → ALL



Inference Mode

Autoregressive Generation



Token-by-token autoregressive generation



Each **generated token** becomes input for next step



Beam search or sampling for generation strategies



KV-caching improves inference efficiency

Processing Mode

ONE → ONE



Speed



Processing



Input Source

Training: **Parallelizable**
Inference: **Sequential**

Training: **All at once**
Inference: **Step by step**

Training: **Ground truth**
Inference: **Generated**

Computational Examples



Interactive Transformer Explainer - Try it yourself!



Training Mode: Parallel Processing

Input sequence: `["I", "love", "deep", "learning", "and", "AI", "research", "!"]`

Hidden dimension: `d = 4`

Step 1: All tokens processed simultaneously

Input Embeddings (8×4)

<code>I:</code>	<code>[0.2, 0.5, -0.1, 0.3]</code>
<code>love:</code>	<code>[0.8, -0.2, 0.4, 0.1]</code>
<code>deep:</code>	<code>[-0.3, 0.6, 0.2, 0.5]</code>
<code>learning:</code>	<code>[0.4, 0.1, -0.5, 0.7]</code>
<code>and:</code>	<code>[0.1, -0.4, 0.3, -0.2]</code>
<code>AI:</code>	<code>[0.9, 0.3, -0.1, 0.6]</code>

```
research: [-0.2, 0.7, 0.4, -0.3]  
!: [0.5, -0.1, 0.6, 0.2]
```

↓ Transformer Layer with Causal Masking

Output Hidden States (8×4)

h₁ :	[0.25, 0.48, -0.12, 0.35]	← attends to position 1 only
h₂ :	[0.52, 0.15, 0.18, 0.28]	← attends to positions 1-2
h₃ :	[0.18, 0.42, 0.08, 0.45]	← attends to positions 1-3
h₄ :	[0.38, 0.22, -0.15, 0.58]	← attends to positions 1-4
h₅ :	[0.22, -0.08, 0.25, 0.12]	← attends to positions 1-5
h₆ :	[0.68, 0.28, 0.05, 0.52]	← attends to positions 1-6
h₇ :	[0.15, 0.55, 0.32, -0.08]	← attends to positions 1-7
h₈ :	[0.42, 0.18, 0.48, 0.25]	← attends to positions 1-8

- ✓ All 8 hidden states computed in **ONE forward pass** (parallel)
- ✓ Causal masking prevents each position from attending to future positions



Inference Mode: Autoregressive Generation

Starting prompt: ["I", "love"]

Target: Generate next 3 tokens

Hidden dimension: `d = 4`

Step 1: Process ["I", "love"] → Generate "deep"

Input Embeddings:

I: [0.2, 0.5, -0.1, 0.3]
love: [0.8, -0.2, 0.4, 0.1]

Output Hidden State:

$h_2 : [0.52, 0.15, 0.18, 0.28]$

Generated: "**deep**"

Step 2: Process ["I", "love", "deep"] → Generate "learning"

Input Embeddings:

I: [0.2, 0.5, -0.1, 0.3]
love: [0.8, -0.2, 0.4, 0.1]
deep: [-0.3, 0.6, 0.2, 0.5]

Output Hidden State:

$h_3 : [0.18, 0.42, 0.08, 0.45]$

Generated: "**learning**"

Step 3: Process ["I", "love", "deep", "learning"] → Generate "and"

Input Embeddings:

I: [0.2, 0.5, -0.1, 0.3]
love: [0.8, -0.2, 0.4, 0.1]
deep: [-0.3, 0.6, 0.2, 0.5]
learning: [0.4, 0.1, -0.5, 0.7]

Output Hidden State:

$h_4 : [0.38, 0.22, -0.15, 0.58]$

Generated: "**and**"

- ✓ Each token requires a **separate forward pass** (sequential)
- ✓ Previously generated tokens become input for next step
- ✓ Context grows at each step: 2 tokens → 3 tokens → 4 tokens

Computation Comparison

Training

1 forward pass

8 tokens → 8 hidden states

Matrix operation (8×4)

All positions in parallel

VS

Inference

3 forward passes

2→3→4 tokens sequentially

Growing context each step

One token at a time

Part 6/7:

Implementation Tips

- 20.** Masking Implementation
- 21.** Learning Stabilization Techniques
- 22.** Hyperparameter Guide

Masking Implementation

Padding Mask

Ignore padding tokens in attention

📐 Shape: `(batch_size, 1, 1, seq_len)`

🎯 Marks which positions are actual tokens vs padding

1	1	1	0	0
---	---	---	---	---



Application Timing

Applied **before softmax** in attention calculation



Mask Values

Use `-inf` for masked positions (becomes 0 after softmax)



Look-Ahead Mask (Causal)

Prevent attending to future tokens

📐 Upper triangular matrix of `-inf` values

🔒 Ensures causality during decoding

✓	X	X	X	X

✓	✓	X	X	X
✓	✓	✓	X	X
✓	✓	✓	✓	X
✓	✓	✓	✓	✓

⌚ Combined Masks

Multiple masks combined using
element-wise **OR** operation



Proper masking is crucial for correct model behavior

Learning Stabilization Techniques



Learning Rate Warmup

Gradually increase learning rate initially to stabilize training

```
lr = d_model-0.5 × min(step-0.5, step × warmup-1.5)
```



Label Smoothing

Reduces **overconfidence** in predictions

- ✓ Typical value: $\epsilon = 0.1$
- ✓ Improves generalization



Gradient Clipping

Prevents **exploding gradients** during training

- ✓ Caps gradient norm at threshold
- ✓ Ensures training stability



Dropout

Regularization through random neuron dropout

- ✓ Applied to **attention weights**
- ✓ Applied to **FFN outputs**



Weight Initialization

Proper initialization for stable gradients

- ✓ **Xavier** initialization
- ✓ **He** initialization



Mixed Precision Training

Uses **FP16** for faster computation

- ✓ Reduces memory usage
- ✓ Speeds up training

Training Best Practices

Combine these techniques for **stable and efficient training** of Transformer models

Hyperparameter Guide

Parameter	Base Model	Large Model
 d_model	512	1024
 Layers (N)	6	12+
 Attention Heads (h)	8	16
 d_ff (FFN hidden)	2048	4096
 Dropout Rate	0.1	0.1



Warmup Steps

4000 steps (typical)



Batch Size

As large as GPU memory allows



Adam Optimizer

β_1
0.9

β_2
0.98

ϵ
 10^{-9}



Base model suitable for most tasks
Large model for complex problems

Part 7/7:

Applications & Next Steps

- 23. Transformer Applications**
- 24. Summary and Next Steps**

Transformer Applications Across Domains



Natural Language Processing

- **GPT** - Text generation
- **BERT** - Language understanding
- **T5** - Text-to-text framework



Machine Translation

- High-quality translation systems

Near human-level performance



Computer Vision

- **ViT** - Vision Transformers
- **DINO** - Self-supervised learning
- **CLIP** - Vision-language models



Speech Processing

- **Whisper** - Speech transcription
- Audio generation systems



Multimodal AI

- **DALL-E** - Text-to-image
- **GPT-4** - Multi-capability AI
- **Flamingo** - Text + Image



Time Series

- **Temporal Fusion Transformer**
- Forecasting and prediction



Code Generation



Biology & Science

- GitHub Copilot - AI pair programming
- AlphaCode - Competitive programming

- AlphaFold2
- Protein structure prediction

Summary and Next Steps

Key Concepts

 Self-Attention Mechanism

 Multi-Head Attention

 Positional Encoding

Advantages

 **Parallelization** - Fast training

 **Long-range dependencies** - Global context

 **Flexibility** - Multiple domains

Challenges

 Quadratic complexity $O(n^2)$

 Large memory requirements

Next Steps

1 Next Lecture

Advanced topics: Sparse attention, Linear attention

2 Efficient Transformers

Explore Reformer, Linformer, Performer

3 Practice

Implement a mini-Transformer from scratch

4 Explore Libraries

Hugging Face Transformers library

Recommended Reading

"Attention Is All You Need" paper

Vaswani et al., 2017

Thank you

Ho-min Park

homin.park@ghent.ac.kr

powersimmani@gmail.com