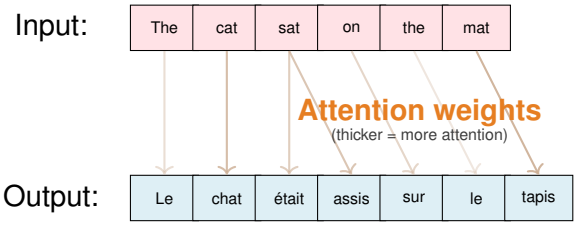
Deep Learning for Perception

Lecture 08: Attention Mechanisms & Transformers



Topics Covered in This Lecture:

- Seq2Seq & Machine Translation
- The Bottleneck Problem
- Attention Mechanism
- Attention Score Functions
- Context Vectors
- Encoder Architectures
- Self-Attention
- Query, Key, Value
- Multi-Head Attention
- Transformer Architecture
- Positional Encoding

Instructor: Deep Learning Faculty

Department of Computer Science
FAST-NUCES

Contents

Advance Organizer — What You'll Learn

The Big Picture: From RNNs to Transformers
You've learned RNNs/LSTMs process sequences step-by-step. But they have a critical flaw: **information gets compressed into a single fixed-size vector**. This lecture introduces the revolutionary **attention mechanism** that changed everything.
Learning Objectives: By the end of this lecture, you will be able to:

1. **Explain** why fixed-length encoding fails for long sequences
2. **Describe** how attention allows selective focus on relevant inputs
3. **Distinguish** between hard and soft attention
4. **Calculate** attention weights using different scoring functions
5. **Explain** self-attention and Query-Key-Value mechanism
6. **Describe** multi-head attention and its benefits
7. **Understand** the Transformer architecture components

Why This Matters for Your Future Learning:

- Attention is the foundation of **all modern NLP** (GPT, BERT, etc.)
- Understanding attention helps you grasp **encoder-decoder architectures**
- These concepts connect to **Variational Autoencoders** (next topic)
- Transformers power **ChatGPT, DALL-E, and modern AI**

Prior Knowledge Required:

- RNNs and LSTMs (Lecture 07)
- Softmax function
- Dot product and matrix multiplication

1 The Problem: Why RNNs Struggle with Long Sequences

Why It Matters

Before learning the solution (attention), you must deeply understand the problem. This section explains why the pioneering Seq2Seq models fail on long sentences—a limitation that attention brilliantly solves.

1.1 Sequence-to-Sequence (Seq2Seq) Models

Definition

Sequence-to-Sequence (Seq2Seq) models transform one sequence into another sequence. They consist of two parts:

1. **Encoder:** Reads input sequence and compresses it into a fixed-size vector
2. **Decoder:** Takes the fixed-size vector and generates output sequence

Key Application: Machine Translation (e.g., English → French)

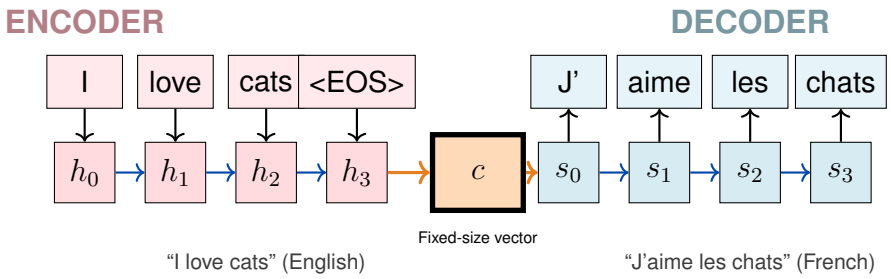


Figure 1: Basic Seq2Seq: Encoder compresses input to fixed vector c, Decoder generates output

1.2 The Bottleneck Problem

The Critical Flaw: Fixed-Size Bottleneck

The Problem: The entire input sequence must be compressed into a **single fixed-size vector** c (typically 256-1024 dimensions).
For short sentences: Works reasonably well

- "I love cats" → 256-dim vector → "J'aime les chats"

For long sentences: Fails badly!

- A 50-word paragraph with complex grammar and multiple clauses...
- ...must STILL fit in the same 256-dim vector!
- Information is inevitably lost
- Translation quality degrades significantly

Research Evidence (Cho et al., 2014):

- Performance drops sharply as sentence length increases
- Beyond 20 words, quality deteriorates rapidly
- The fixed vector simply **cannot capture all information** needed for long sentences

Analogy — Think of It Like This

The Telephone Game Analogy
Imagine playing the telephone game:
Traditional Seq2Seq (Bottleneck):

- Person A reads a long story
- Person A must whisper ONE sentence to Person B
- Person B must recreate the entire story from that one sentence!
- Result: Most details are lost

With Attention (Solution):

- Person A reads the story and takes notes on every paragraph
- Person B can ASK about any specific part at any time
- "What happened in paragraph 3?" $\rightarrow$ Person A reads those notes
- Result: All details are accessible when needed!

Key Insight: Instead of one compressed summary, attention lets the decoder "look back" at the entire input whenever needed!

Memory Hook — Remember This!

Remember the Core Problem:

Bottleneck = Information Loss

Long sequence $\xrightarrow{\text{compress}}$ Fixed vector $\xrightarrow{\text{decode}}$ Poor translation

"You can't fit a novel into a tweet!"

2 The Solution: Attention Mechanism

Why It Matters

Attention is one of the most important innovations in deep learning. It solves the bottleneck problem elegantly: instead of compressing everything into one vector, **let the decoder decide which parts of the input to focus on at each step**.

2.1 The Key Idea

Definition

Attention Mechanism allows the decoder to **selectively focus** on different parts of the input sequence when generating each output word.
Three-Step Process:

1. **Encoder** produces a hidden state for **every** input word (not just the last one!)
2. **Attention weights** are computed to determine each input's relevance for the current output
3. **Context vector** is a weighted sum of all encoder hidden states

Key Insight: The decoder no longer relies on a single fixed vector—it can "attend to" any part of the input at any time!

2.2 Hard vs. Soft Attention

Before diving into the standard (soft) attention, let's understand two approaches:

Two Types of Attention

Hard Attention:

- Selects **exactly one** input word to focus on
- Like pointing at one specific word
- Problem: Not differentiable (can't use backpropagation!)
- Problem: One word may not be enough ("the" $\rightarrow$ "l' " depends on the noun!)

Soft Attention (Standard):

- Uses a **weighted combination** of all inputs
- Weights are continuous values between 0 and 1 (sum to 1)
- Fully differentiable—works with backpropagation!
- Can consider multiple relevant words simultaneously

We use Soft Attention because it's differentiable and more flexible!

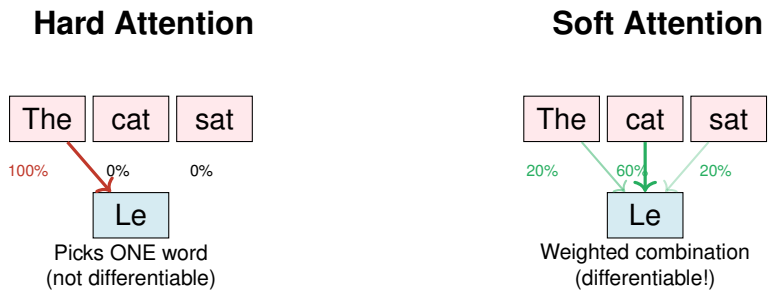


Figure 2: Hard attention picks one word; Soft attention uses weighted combination

2.3 How Attention Works: Step by Step

Step 1: Encoder Produces Hidden States for ALL Inputs

Unlike basic Seq2Seq (which only uses the final hidden state), attention **keeps all encoder hidden states**:

Input: "I love cats"

h_0 = hidden state after "I"
h_1 = hidden state after "love"
h_2 = hidden state after "cats"

All three are kept and passed to decoder!

Why? Each hidden state contains information about that word *and its context*. The decoder needs access to all of them.

Step 2: Compute Attention Scores

At each decoder time step t, we compute how **relevant** each encoder hidden state is:
Attention Score $e_{t,i}$ = similarity between:

- Decoder's current hidden state s_{t-1} (what we're trying to predict)
- Encoder's hidden state h_i (information from input word i)

Higher score = more relevant input word for current prediction
How to measure similarity? Several options (next section)...

Step 3: Convert Scores to Weights (Softmax)

Raw scores can be any value. We convert them to **probabilities** using softmax:

$$\alpha_{t,i} = \frac{\exp(e_{t,i})}{\sum_{j=1}^{n} \exp(e_{t,j})}$$

Properties of attention weights α:

- All weights are between 0 and 1
- All weights sum to 1: $\sum_i \alpha_{t,i} = 1$
- Higher weight = more attention to that input word

Step 4: Compute Context Vector

The **context vector** c_t is a weighted sum of all encoder hidden states:

$$c_t = \sum_{i=1}^{n} \alpha_{t,i} \cdot h_i$$

Intuition:

- If $\alpha_{t,2} = 0.8$ (80% attention on word 2), then $c_t \approx h_2$
- If attention is spread evenly, c_t is an average of all hidden states
- Context vector "summarizes" the relevant parts of input for current prediction

Key Insight: Context vector is **different at each time step!** The decoder focuses on different input words when generating different output words.

Step 5: Make Prediction

The decoder uses **three inputs** to predict the next word:

1. Previous output word (what we just generated)
2. Previous hidden state s_{t-1} (decoder's memory)
3. **Context vector** c_t (relevant input information) — *NEW!*

This is much richer than basic Seq2Seq, which only used the first two!

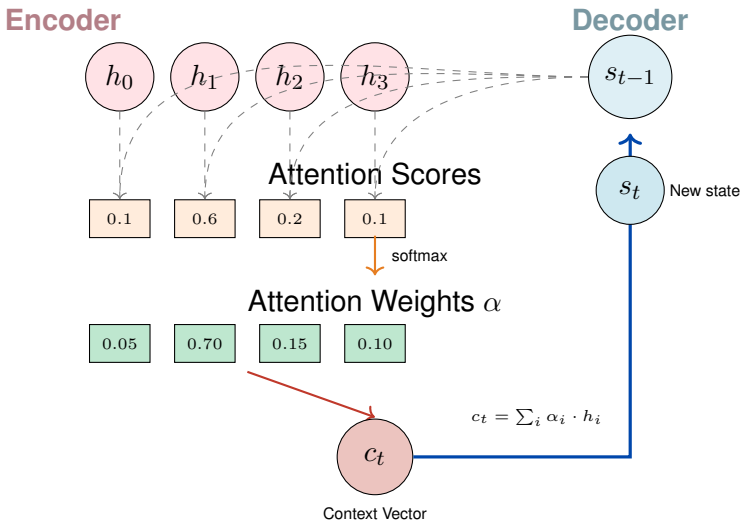


Figure 3: Complete attention mechanism: scores → weights → context vector → decoder

2.4 Attention Score Functions

Key Formula

How to compute attention scores?
The attention score measures **similarity** between decoder state s and encoder state h. Three common approaches:

Method	Formula	Parameters
Dot Product	$e = s^T h$	None
Bilinear (General)	$e = s^T W h$	Matrix W
MLP (Additive)	$e = v^T \tanh(W_1 s + W_2 h)$	W_1, W_2, v

Dot Product: Simplest, no parameters to learn. Works when s and h have same dimension.
Bilinear: Learns a transformation matrix W. More flexible than dot product.
MLP: Most expressive. Uses a small neural network to compute scores.

Concrete Example

Example: Translation with Attention
Task: Translate "The man ate the apple" to French
When generating "L'homme" (the man):

- High attention on "The" and "man" (directly relevant)
- Low attention on "ate", "the", "apple"
- Weights: [0.3, 0.6, 0.05, 0.03, 0.02]

When generating "mangea" (ate):

- High attention on "ate"
- Some attention on "man" (subject affects verb conjugation)
- Weights: [0.1, 0.2, 0.6, 0.05, 0.05]

When generating "la pomme" (the apple):

- High attention on "the" and "apple"
- Low attention on others
- Weights: [0.05, 0.05, 0.1, 0.3, 0.5]

Key Insight: The model learns to **align** source and target words automatically!

2.5 Encoder Architectures for Attention

The encoder processes the input sequence and produces hidden states. Two popular architectures:

Bidirectional RNN

Idea: Process input in **both directions** (forward and backward)
Why? Understanding a word often requires context from *both* past and future:

- "Teddy Roosevelt was a president" $\rightarrow$ "Teddy" is a person
- "Teddy bears are soft" $\rightarrow$ "Teddy" is a toy

How it works:

- Forward RNN: Reads left-to-right, produces $\overrightarrow{h_i}$
- Backward RNN: Reads right-to-left, produces $\overleftarrow{h_i}$
- Combined: $h_i = [\overrightarrow{h_i}; \overleftarrow{h_i}]$ (concatenation)

Used in: Bahdanau's NMT (2015) — the original attention paper!

Stacked (Deep) RNN

Idea: Multiple RNN layers stacked on top of each other
Why? Deeper networks learn more complex patterns:

- Layer 1: Basic word features
- Layer 2: Phrase-level patterns
- Layer 3+: High-level semantic meaning

Used in: Google's NMT (8 layers, first layer bidirectional)

Memory Hook — Remember This!

Attention Summary — The Key Points:

1. **Problem:** Fixed-size bottleneck loses information for long sequences
2. **Solution:** Let decoder "look at" all encoder hidden states
3. **How:** Compute attention weights (softmax of similarity scores)
4. **Result:** Context vector = weighted sum of encoder states
5. **Benefit:** Different context for each output word!

Performance Impact: Attention maintains quality even for long sentences!

3 Self-Attention: The Foundation of Transformers

Why It Matters

Attention (previous section) relates **two different sequences** (encoder → decoder). But what if we want to relate **words within the same sequence**? This is **self-attention**—the revolutionary idea behind Transformers that power GPT, BERT, and modern AI.

3.1 From RNN to Self-Attention

The Problem with RNNs

RNNs have fundamental limitations:
1. Sequential Processing:

- Must process word-by-word, left to right
- Cannot parallelize—slow training!
- GPUs are designed for parallel computation

2. Long-Range Dependencies:

- Information must "flow" through all intermediate states
- Even LSTMs struggle with very long sequences
- Vanishing gradients still a problem

3. Fixed Path Length:

- Word 1 and word 100 are 99 steps apart in RNN
- Hard to learn direct relationships between distant words

Definition

Self-Attention creates a new representation of each word by considering its relationship to **all other words in the same sequence**.
Key Difference from General Attention:

- **General Attention:** Relates words from *different* sequences (encoder ↔ decoder)
- **Self-Attention:** Relates words within the *same* sequence

Result: Every word gets a new representation that encodes its relationship to all other words—**without any recurrence!**

Concrete Example

Why Self-Attention Helps: Word Disambiguation
Consider: "I arrived at the **bank** after crossing the river."
What does "bank" mean?

- Financial institution? (money, account, loan...)
- River bank? (water, shore, edge...)

Self-attention solves this:

- Compute attention between “bank” and all other words
- High attention score with “river” and “crossing”
- New representation of “bank” encodes: “this is a river bank”

Another example: “She gave her brother his present because **she** loves him.”

- Who is “she”? Self-attention links “she” to “She” (same person)
- High attention weight between these pronouns

3.2 Query, Key, Value: The QKV Mechanism

Definition

Self-attention uses three learned projections for each word:
Query (Q): “What am I looking for?”

- Represents the word’s “question” about other words
- Used to compute attention scores

Key (K): “What do I contain?”

- Represents the word’s “label” or “tag”
- Compared with queries to determine relevance

Value (V): “What information do I provide?”

- The actual content to be aggregated
- Weighted by attention scores

Each is computed: $Q = XW^Q$, $K = XW^K$, $V = XW^V$
where W^Q, W^K, W^V are learned weight matrices.

Analogy — Think of It Like This

Library Analogy for Query-Key-Value
Imagine searching in a library:
Query (Q): Your search request

- “I want books about machine learning”
- What you’re looking for

Key (K): Book labels/tags in the catalog

- “AI”, “Programming”, “Statistics”, “History”...
- How books are indexed

Value (V): The actual book content

- The information you retrieve
- What you actually read

Search Process:

1. Compare your query to all keys (labels)

2. Find matching books (high similarity scores)
3. Retrieve content from matching books (weighted values)

Self-attention works the same way!

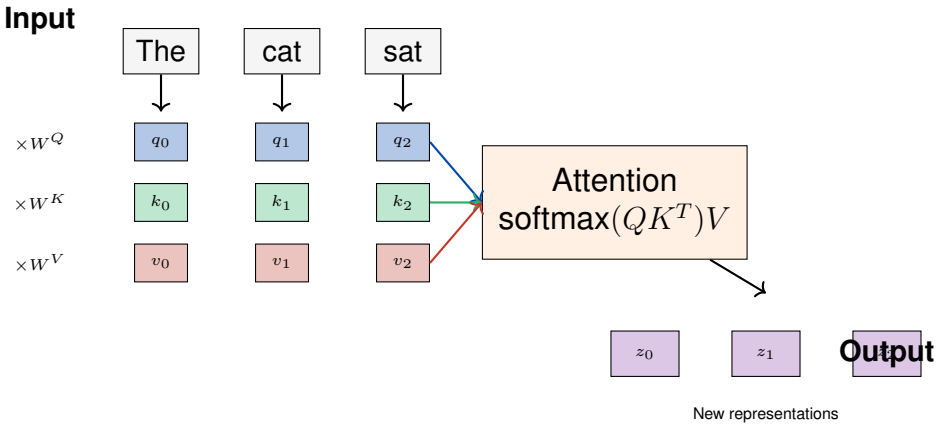


Figure 4: Self-attention: Input → Q, K, V projections → Attention computation → Output

3.3 Computing Self-Attention Step by Step

Step 1: Create Q, K, V Vectors

For each input word embedding x_i, compute three vectors:

$$q_i = x_i W^Q \quad \text{(Query)}$$
$$k_i = x_i W^K \quad \text{(Key)}$$
$$v_i = x_i W^V \quad \text{(Value)}$$

W^Q, W^K, W^V are learned during training.
Example: If input has 3 words, we get:

- 3 query vectors: q_0, q_1, q_2
- 3 key vectors: k_0, k_1, k_2
- 3 value vectors: v_0, v_1, v_2

Step 2: Compute Attention Scores

For each word, compute similarity with **all words** (including itself):

$$e_{ij} = q_i \cdot k_j \quad \text{(dot product)}$$

Example for word 0:

- $e_{00} = q_0 \cdot k_0$ (how relevant is word 0 to word 0?)
- $e_{01} = q_0 \cdot k_1$ (how relevant is word 1 to word 0?)
- $e_{02} = q_0 \cdot k_2$ (how relevant is word 2 to word 0?)

Step 3: Scale and Softmax

Scale scores by $\sqrt{d_k}$ (dimension of key vectors) for stable gradients:

$$\alpha_{ij} = \text{softmax}\left(\frac{e_{ij}}{\sqrt{d_k}}\right)$$

Why scale? Large dot products cause softmax to have very small gradients. Scaling helps training.

Step 4: Compute Weighted Sum of Values

New representation for word i:

$$z_i = \sum_j \alpha_{ij} \cdot v_j$$

Intuition: The new representation aggregates information from all words, weighted by relevance!

Key Formula

Scaled Dot-Product Attention (Complete Formula)

$$\boxed{\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V}$$

Where:

- Q = matrix of all query vectors (each row is one query)
- K = matrix of all key vectors
- V = matrix of all value vectors
- d_k = dimension of key vectors
- QK^T = attention scores matrix
- Softmax applied row-wise

This entire operation can be computed in ONE matrix multiplication—highly parallelizable!

3.4 Multi-Head Attention

Definition

Multi-Head Attention runs self-attention **multiple times in parallel**, each with different learned projections.
Why? One attention head might focus on:

- Syntactic relationships (subject-verb agreement)
- Semantic relationships (word meanings)
- Positional relationships (nearby words)

Multiple heads capture different types of relationships!

Formula:

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$
$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O$$

Concrete Example

Multi-Head Attention Example
Sentence: "The animal didn't cross the street because it was too tired."
Question: What does "it" refer to?
Head 1 (syntactic):

- Focuses on grammatical structure
- "it" $\rightarrow$ high attention on "animal" (both nouns, subject position)

Head 2 (semantic):

- Focuses on meaning
- "tired" $\rightarrow$ high attention on "animal" (animals get tired)

Combined: Multiple heads agree that "it" = "animal"!
Key Insight: Different heads learn to focus on different linguistic phenomena.

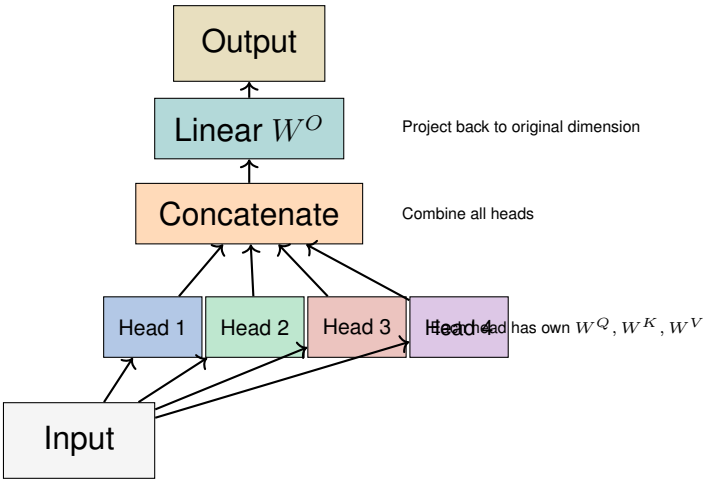


Figure 5: Multi-Head Attention: Multiple parallel attention operations combined

4 The Transformer Architecture

Why It Matters

The Transformer ("Attention Is All You Need", 2017) revolutionized NLP by eliminating recurrence entirely. It's the foundation of GPT, BERT, and virtually all modern language models. Understanding its components is essential for modern deep learning.

Historical Context

2017: "Attention Is All You Need" (Vaswani et al.)
This paper introduced the Transformer architecture:

- Replaced RNNs entirely with self-attention
- Achieved state-of-the-art on machine translation
- Enabled massive parallelization $\rightarrow$ faster training
- Became the foundation for GPT-1 (2018), BERT (2018), GPT-2 (2019), GPT-3 (2020), ChatGPT (2022)

The paper's bold claim: You don't need recurrence or convolution—attention is all you need!

4.1 Transformer Block Components

A Transformer consists of stacked **Transformer blocks**. Each block has:

Component 1: Multi-Head Self-Attention

Purpose: Learn relationships between all positions in the sequence
What it does:

- Each word attends to all words (including itself)
- Multiple attention heads capture different patterns
- Output: New representations encoding word relationships

Component 2: Feed-Forward Network (FFN)

Purpose: Add non-linear transformations
Structure: Two linear layers with ReLU activation

$$\text{FFN}(x) = \text{ReLU}(xW_1 + b_1)W_2 + b_2$$

Key Point: Applied to each position **independently** (no interaction between positions here—that's what attention does!)

Component 3: Residual Connections

Purpose: Help gradient flow in deep networks
How: Add input to output of each sublayer

$$\text{Output} = \text{LayerNorm}(x + \text{Sublayer}(x))$$

Why it helps:

- Gradients can flow directly through skip connections
- Makes training deep networks (6+ layers) much easier
- Same idea as ResNet (from CNNs!)

Component 4: Layer Normalization

Purpose: Stabilize training, faster convergence
How: Normalize activations within each layer
Applied after: Each sublayer (attention and FFN)

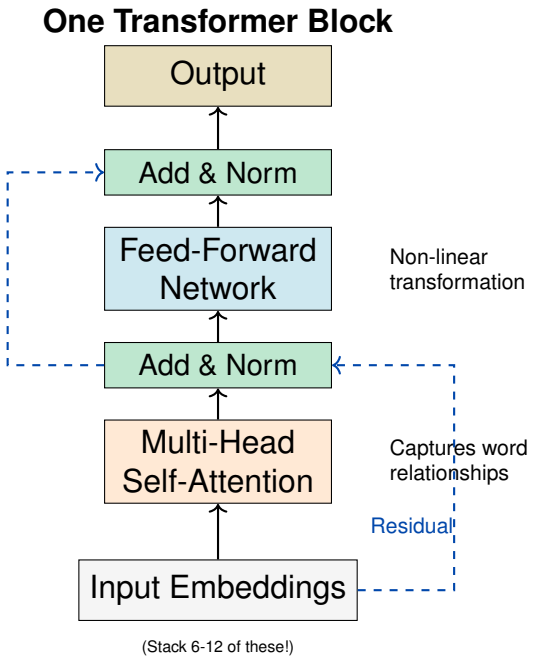


Figure 6: Transformer block: Self-Attention + FFN + Residuals + LayerNorm

4.2 Positional Encoding: Handling Word Order

The Position Problem

RNNs: Process words sequentially, so position is implicit
Self-Attention: Processes all words in parallel—**no notion of order!**

- "The cat sat on the mat" and "mat the on sat cat The"
- Would produce the SAME attention scores!
- This is clearly wrong—word order matters!

Solution: Add **positional information** to embeddings

Definition

Positional Encoding adds position information to word embeddings.
Two approaches:

1. Learned Position Embeddings:

- Learn a vector for each position (1st, 2nd, 3rd, ...)
- Simple but limited to maximum sequence length seen in training

2. Sinusoidal Position Encodings (original Transformer):

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

Added to input: $\text{input}_i = \text{embedding}_i + \text{positional\_encoding}_i$

Analogy — Think of It Like This

Positional Encoding Analogy
Think of it like **timestamps on a recording**:
Without timestamps:

- Audio clips with no time information
- "Hello" "World" "!" — but in what order?
- Could be "Hello World !" or "World ! Hello"

With timestamps:

- "Hello" (t=0), "World" (t=1), "!" (t=2)
- Now order is clear!

Positional encoding adds the "timestamp" to each word embedding.

4.3 Full Transformer for Machine Translation

The original Transformer has an **encoder-decoder structure**:

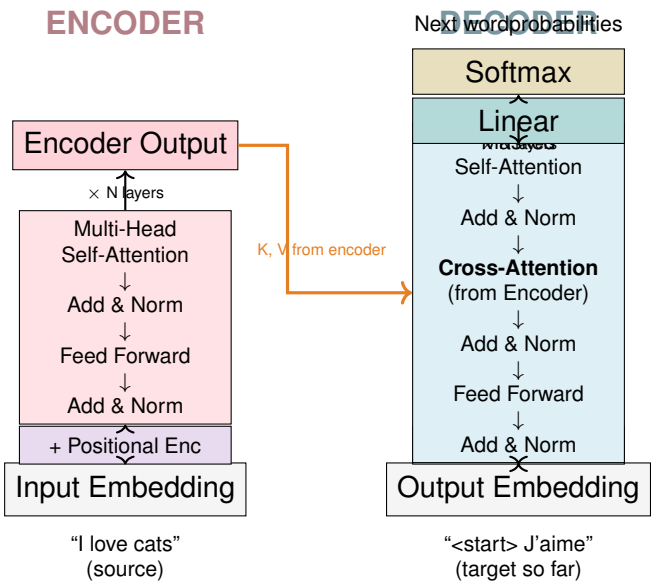


Figure 7: Full Transformer: Encoder (left) processes source, Decoder (right) generates target

Key Components of Full Transformer

Encoder (processes source sentence):

- Self-attention over all source words
- N = 6 identical layers in original paper
- Output: Rich representations of source

Decoder (generates target sentence):

- **Masked** self-attention: Can only attend to previous words (prevents "cheating")
- **Cross-attention**: Attends to encoder output (like original attention mechanism!)
- N = 6 identical layers

Cross-Attention:

- Query (Q): from decoder
- Key (K) and Value (V): from encoder
- This is where encoder-decoder communication happens!

Connection to Future Topics

Connection to Your Next Topics
Encoder-Decoder Architecture:

- Transformers use encoder-decoder structure
- Same pattern appears in **Autoencoders** and **VAEs**!
- Encoder: Compress input to representation
- Decoder: Generate output from representation

What You'll Learn Next:

- **Autoencoders:** Encoder compresses → Decoder reconstructs

- **VAEs:** Add probability to latent space → Can generate new data!
- The attention you learned today is used in advanced VAEs too

The Big Picture:

Seq2Seq (RNN-based) → **Attention** (solve bottleneck) → **Transformer** (no RNN!)

Autoencoder (compress) → **VAE** (generate) → **Modern Generative AI**

5 Summary: The Complete Picture

Key Takeaways from This Lecture

1. The Problem: Fixed-Size Bottleneck

- Seq2Seq compresses entire input to single vector
- Information lost for long sequences
- Performance degrades with sentence length

2. The Solution: Attention Mechanism

- Decoder can "look at" all encoder hidden states
- Attention weights determine relevance of each input
- Context vector = weighted sum of encoder states
- Different context for each output word!

3. Attention Score Functions

- Dot product: $s^T h$ (simple, no parameters)
- Bilinear: $s^T W h$ (learnable)
- MLP: $v^T \tanh(W_1 s + W_2 h)$ (most expressive)

4. Self-Attention

- Relates words *within* same sequence
- Query-Key-Value mechanism
- Scaled dot-product: softmax$(QK^T/\sqrt{d_k})V$
- Enables parallel processing (no recurrence!)

5. Multi-Head Attention

- Multiple attention heads in parallel
- Different heads capture different patterns
- Concatenate and project outputs

6. Transformer Architecture

- Self-Attention + Feed-Forward + Residuals + LayerNorm
- Positional encoding for word order
- Encoder-decoder structure for translation
- Foundation of GPT, BERT, ChatGPT!

Self-Test — Check Your Understanding

Check Your Understanding:
Q1: Why does the basic Seq2Seq model fail on long sentences?
Answer: All information must be compressed into a single fixed-size vector, causing information loss for long sequences.
Q2: What are the three steps of the attention mechanism?
Answer: (1) Compute attention scores, (2) Apply softmax to get weights, (3) Compute context vector as weighted sum.

Q3: What is the difference between general attention and self-attention?
Answer: General attention relates two different sequences (encoder-decoder). Self-attention relates words within the same sequence.
Q4: In Query-Key-Value, what does each component represent?
Answer: Query: What am I looking for? Key: What do I contain? Value: What information do I provide?
Q5: Why does the Transformer need positional encoding?
Answer: Self-attention processes all words in parallel with no inherent notion of order. Positional encoding adds position information.
Q6: What is multi-head attention and why is it useful?
Answer: Multiple parallel attention operations with different learned projections. Different heads capture different types of relationships.

6 Glossary

Term	Definition
Seq2Seq	Sequence-to-sequence model with encoder-decoder architecture
Bottleneck	Fixed-size vector that compresses entire input sequence
Attention	Mechanism to selectively focus on relevant input parts
Hard Attention	Selects one input (not differentiable)
Soft Attention	Weighted combination of all inputs (differentiable)
Context Vector	Weighted sum of encoder hidden states
Self-Attention	Attention within the same sequence
Query (Q)	Vector representing what we're looking for
Key (K)	Vector representing what each word contains
Value (V)	Vector containing information to aggregate
Multi-Head	Multiple parallel attention operations
Transformer	Architecture using only attention (no RNN)
Positional Encoding	Added signal to represent word position
Cross-Attention	Decoder attends to encoder output
Masked Attention	Can only attend to previous positions