Attention Mechanism and Transformers

CSCI E-89B: Introduction to Natural Language Processing

Lecture 12

Lecture Information

Course: CSCI E-89B: Introduction to Natural Language Processing
Lecture: 12 – Attention Mechanism and Transformers
Institution: Harvard Extension School
Topics: Attention Mechanism, Query-Key-Value, Multi-Head Attention, Transformers, BERT, GPT

Contents

1 Introduction and Quiz Review

Overview

This lecture introduces two revolutionary concepts in modern NLP: the **attention mechanism** and the **transformer architecture**. These form the foundation of virtually all state-of-the-art language models including BERT, GPT, and ChatGPT.

1.1 Review of Previous Concepts

Before diving into attention mechanisms, let's review key concepts from previous lectures:

Important

Conditional Random Fields vs. Hidden Markov Models:

- CRFs can use more complex dependencies—they can look at joint distributions with **both past and future tokens**
- HMMs assume current state depends **only on the previous state**
- This is not about discrete vs. continuous variables; it's about the directionality of dependencies

Key Quiz Points:

1. **CRF Advantage:** Ability to capture dependencies from both past and future context
2. **LSTM + CRF Benefit:** Enhanced ability to capture both past and future context
3. **Regular Autoencoder Issue:** The encoding space is *not structured*—even slight shifts in encodings can completely destroy the output
4. **Variational Autoencoder:** The mean (μ) is a deterministic function of the input; adding noise makes the output non-deterministic

2 Limitations of Recurrent Neural Networks

2.1 The Problem with Sequential Processing

Definition

Recurrent Neural Network (RNN) Structure:
In a standard RNN, we have:

- Input sequence: $x_1, x_2, \ldots, x_T$
- Hidden states: $h_1, h_2, \ldots, h_T$
- Each hidden state depends on the previous: $h_t = f(h_{t-1}, x_t)$

This sequential nature creates several fundamental problems:

2.1.1 Problem 1: Vanishing Gradients

Warning

As sequences get longer, gradients that flow backward through time become increasingly small. This makes it extremely difficult to learn long-range dependencies.
Analogy: Imagine passing a message through a long chain of people, where each person slightly garbles the message. By the end, the original message is nearly unrecognizable.

Even with LSTM's long-term memory mechanism, the vanishing gradient problem is only *mitigated*, not eliminated.

2.1.2 Problem 2: No Parallel Computation

Warning

Because each hidden state h_t depends on h_{t-1}, we **cannot** compute states in parallel. This makes training extremely slow for long sequences.
Impact: A 1000-word document requires 1000 sequential computations, regardless of how many GPUs you have.

2.1.3 Problem 3: Fixed-Size Bottleneck

In encoder-decoder architectures (like sequence-to-sequence models for translation):

- The entire input sequence is compressed into a **single fixed-dimension vector**
- Whether your sentence is 5 words or 50 words, it must fit into the same size representation (e.g., 128 dimensions)
- This creates an information bottleneck

Example

The Compression Problem:
Consider trying to store different length sentences in a 128-dimensional vector:

- "Hello" $\rightarrow$ 128-dim vector (easy)
- "The quick brown fox jumps over the lazy dog" $\rightarrow$ same 128-dim vector
- A 500-word paragraph $\rightarrow$ still the same 128-dim vector!

Mathematically, there IS enough space (128 dimensions is huge), but practically, learning this mapping is extremely difficult.

2.1.4 Problem 4: Difficulty with Future Context

In a standard RNN:

$$y_2 = f(y_1, x_2) = f(f(y_0, x_1), x_2) \tag{1}$$

The prediction at position 2 depends only on positions 0, 1, and 2. But what if position 2's meaning depends on position 4?

Example

When Future Words Matter:
"I went to the bank to deposit my check."
When translating "bank," you need to see "deposit" (which comes later) to know it's a financial institution, not a river bank.

3 The Attention Mechanism

3.1 Core Idea

Definition

Attention: Instead of compressing the entire input into a single vector, attention allows the model to "look at" all input positions and dynamically decide which positions are most relevant for each output position.
Key Innovation: Create a **context vector** C_t for each time step that is a weighted combination of *all* hidden states.

3.2 Historical Context

The attention mechanism was introduced in the landmark paper:

Info

"Neural Machine Translation by Jointly Learning to Align and Translate"
Bahdanau, Cho, and Bengio (2015)
This paper showed that attention-based models significantly outperform traditional encoder-decoder models for machine translation.

3.3 How Attention Works

3.3.1 Step 1: Compute Hidden States

First, we run a bidirectional RNN (or LSTM) to get hidden states that capture both past and future context:

$$\overrightarrow{h_t} = \text{forward RNN}(x_1, \ldots, x_t) \tag{2}$$

$$\overleftarrow{h_t} = \text{backward RNN}(x_T, \ldots, x_t) \tag{3}$$

$$h_t = [\overrightarrow{h_t}; \overleftarrow{h_t}] \quad \text{(concatenation)} \tag{4}$$

3.3.2 Step 2: Compute Alignment Scores

For each output position t, we compute how much it should "attend" to each input position j:

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

where:

- s_{t-1} is the previous decoder state
- h_j is the encoder hidden state at position j
- a is an **alignment function** (a small neural network)

The alignment function is typically:

$$a(s_{t-1}, h_j) = \tanh(W_s \cdot s_{t-1} + W_h \cdot h_j) \tag{6}$$

3.3.3 Step 3: Convert to Probabilities (Attention Weights)

Apply softmax to get normalized attention weights:

$$\alpha_{tj} = \frac{\exp(e_{tj})}{\sum_{k=1}^{T} \exp(e_{tk})} \tag{7}$$

Important

Key Insight: The α values are **not learnable parameters**—they are computed dynamically based on the input! This allows the model to adapt to inputs of any length.

3.3.4 Step 4: Compute Context Vector

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

$$\boxed{C_t = \sum_{j=1}^{T} \alpha_{tj} \cdot h_j} \tag{8}$$

3.3.5 Step 5: Generate Output

The decoder uses both the previous state and the context vector:

$$s_t = f(s_{t-1}, y_{t-1}, C_t) \tag{9}$$

3.4 Visualizing Attention

Example

Translation Example: English to French
"The agreement on the European Economic Area was signed in August 1992."
↓ translates to ↓
"L'accord sur la zone économique européenne a été signé en août 1992."
The attention mechanism learns that:

- "zone" (French, position 5) attends strongly to "Area" (English, position 7)
- "1992" attends to "1992" (diagonal alignment)
- "signé" attends to "signed"

This is visualized as a matrix where bright squares indicate high attention weights.

3.5 Benefits of Attention

Summary

Advantages of Attention Mechanism:

1. **Dynamic Context:** No more rigid, fixed-size bottleneck
2. **Long-Range Dependencies:** Can directly connect distant positions
3. **Variable-Length Inputs:** Same architecture works for any sequence length
4. **Interpretability:** Attention weights show what the model focuses on
5. **Better Translation:** Handles word reordering across languages

4 From Attention to Transformers

4.1 The Key Question

With attention, we've solved the bottleneck problem. But we still have recurrent connections. A researcher might ask:

Important

"If attention is so powerful that we're computing relationships between ALL positions anyway, do we even need the recurrent network?"
Answer: **NO!** This insight led to the Transformer architecture.

4.2 The Transformer Paper

Info

"Attention Is All You Need"
Vaswani et al., Google Brain (2017)
This paper introduced the Transformer architecture, which completely removes recurrence and relies solely on attention. It revolutionized NLP and became the foundation for BERT, GPT, and virtually all modern language models.

4.3 Limitations of RNNs with Attention

Even with attention, RNN-based models still have issues:

1. **Sequential Computation:** Still can't parallelize the RNN part
2. **Vanishing Gradients:** Still present in the recurrent connections
3. **Knowledge Transfer:** Difficult to transfer learned representations to new tasks

4.4 Training Cost Perspective

Warning

The Cost of Training:
Training a large Transformer model (like GPT-3) is estimated to cost around **$4.6 million** in compute costs alone!
This is why transfer learning is so important—we want to train once and reuse the knowledge for many downstream tasks.

5 The Transformer Architecture

5.1 High-Level Overview

The Transformer consists of two main components:

1. **Encoder:** Processes the input sequence
2. **Decoder:** Generates the output sequence

Definition

Transformer Architecture Components:
Encoder:

- Input Embeddings
- Positional Encoding
- Multi-Head Self-Attention
- Feed-Forward Network

- Residual Connections + Layer Normalization

Decoder:

- Output Embeddings (shifted right)
- Positional Encoding
- Masked Multi-Head Self-Attention
- Encoder-Decoder Attention
- Feed-Forward Network
- Linear + Softmax Output

5.2 Query, Key, and Value

Important

The heart of the Transformer is the **Query-Key-Value (QKV)** attention mechanism. This is fundamentally different from the earlier attention mechanism.

5.2.1 The Motivation: Context-Dependent Embeddings

Example

The Problem with Static Embeddings:
Consider the word "bank":

- "I went to the bank to deposit my check." → Financial institution
- "I sat by the river bank." → Edge of a river

With traditional embeddings (Word2Vec, GloVe), "bank" has the **same vector** in both cases!
We need **context-dependent representations**.

5.2.2 Defining Q, K, V

Given input embeddings H (a matrix where each row is a token embedding):

$$Q = H \cdot W_Q \quad \text{(Query)} \tag{10}$$
$$K = H \cdot W_K \quad \text{(Key)} \tag{11}$$
$$V = H \cdot W_V \quad \text{(Value)} \tag{12}$$

where W_Q, W_K, W_V are learnable weight matrices.

5.2.3 Intuition for Q, K, V

Definition

Understanding Query, Key, Value:
Think of it like a database lookup:

- **Query (Q):** "What am I looking for?" — Represents the current token asking for information
- **Key (K):** "What do I have to offer?" — Represents each token's identity for matching
- **Value (V):** "What information do I carry?" — The actual content to be retrieved

Analogy: Searching a library

- Query: Your search terms
- Key: Book titles/keywords (used for matching)
- Value: Actual book content (what you get back)

5.2.4 Why Three Different Representations?

Info

Flexibility Through Different Roles:
The same token plays different roles in attention:

1. When it's the **token being updated**: Use Q
2. When it's **helping to update another token**: Use K (for matching) and V (for content)

Having separate transformations gives the model more flexibility to learn different aspects for each role.

5.3 Scaled Dot-Product Attention

The attention scores are computed as:

$$\boxed{\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V} \tag{13}$$

5.3.1 Step-by-Step Computation

1. **Compute similarity scores:** QK^T gives a matrix of dot products between all query-key pairs
2. **Scale:** Divide by $\sqrt{d_k}$ (dimension of keys) to prevent extremely large values

3. **Softmax:** Convert to probabilities (rows sum to 1)
4. **Weighted sum:** Multiply by V to get the output

Example

Concrete Example: "Clouds drift across blue sky"
Let's trace attention for the word "sky":
Step 1: Compute Q for "sky"

$$Q_{\text{sky}} = H_{\text{sky}} \cdot W_Q = [q_1, q_2, \ldots, q_d] \tag{14}$$

Step 2: Compute scores with all keys

$$\text{score}_{\text{clouds}} = Q_{\text{sky}} \cdot K_{\text{clouds}}^T / \sqrt{d_k} \tag{15}$$
$$\text{score}_{\text{drift}} = Q_{\text{sky}} \cdot K_{\text{drift}}^T / \sqrt{d_k} \tag{16}$$
$$\vdots \tag{17}$$
$$\text{score}_{\text{sky}} = Q_{\text{sky}} \cdot K_{\text{sky}}^T / \sqrt{d_k} \tag{18}$$

Step 3: Apply softmax

$$[\alpha_{\text{clouds}}, \alpha_{\text{drift}}, \alpha_{\text{across}}, \alpha_{\text{blue}}, \alpha_{\text{sky}}] = \text{softmax}([\text{scores}]) \tag{19}$$

Example result: $[0.30, 0.10, 0.10, 0.30, 0.20]$
Step 4: Compute new representation

$$\text{New}_{\text{sky}} = 0.30 \cdot V_{\text{clouds}} + 0.10 \cdot V_{\text{drift}} + \ldots + 0.20 \cdot V_{\text{sky}} \tag{20}$$

The new representation of "sky" now incorporates context from related words like "blue" and "clouds"!

5.3.2 Why Scale by $\sqrt{d_k}$?

Warning

Numerical Stability:
For high-dimensional vectors, dot products can become very large. If $d_k = 512$:

- Expected magnitude of dot product: $\approx \sqrt{512} \approx 22.6$
- Softmax of large values $\rightarrow$ extremely peaked distribution
- Extremely peaked $\rightarrow$ near-zero gradients $\rightarrow$ no learning!

Dividing by $\sqrt{d_k}$ normalizes the variance back to 1.

5.4 Multi-Head Attention

Definition

Multi-Head Attention:
Instead of computing attention once, we compute it multiple times in parallel with different projections:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O \tag{21}$$

$$\text{where head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \tag{22}$$

Typical configurations use $h = 8$ or $h = 16$ heads.

5.4.1 Why Multiple Heads?

Info

Benefits of Multi-Head Attention:

1. **Different Types of Relationships:** One head might learn syntactic relationships, another semantic
2. **Different Positions:** Different heads can focus on nearby vs. distant tokens
3. **Redundancy:** Multiple chances to capture important patterns
4. **Parallel Computation:** All heads can be computed simultaneously

Analogy: Like having multiple experts look at the same problem from different angles.

5.5 Positional Encoding

Warning

The Order Problem:
Unlike RNNs, Transformers process all positions simultaneously. Without any modification:
"The cat sat on the mat" and "mat the on sat cat The"
would produce **identical representations!**

5.5.1 The Solution: Add Position Information

We add a **positional encoding** to each embedding:

$$\text{Input}_{\text{final}} = \text{Embedding} + \text{PositionalEncoding} \tag{23}$$

The original Transformer uses sinusoidal functions:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right) \tag{24}$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right) \tag{25}$$

where:

- pos = position in the sequence (0, 1, 2, ...)
- i = dimension index
- d_{model} = embedding dimension

Example

Example: Computing Positional Encoding

For position 0 with $d_{model} = 4$:

$$PE_{(0,0)} = \sin(0) = 0 \tag{26}$$

$$PE_{(0,1)} = \cos(0) = 1 \tag{27}$$

$$PE_{(0,2)} = \sin(0) = 0 \tag{28}$$

$$PE_{(0,3)} = \cos(0) = 1 \tag{29}$$

For position 1:

$$PE_{(1,0)} = \sin(1/10000^0) = \sin(1) \approx 0.84 \tag{30}$$

$$PE_{(1,1)} = \cos(1) \approx 0.54 \tag{31}$$

$$\vdots \tag{32}$$

Each position gets a unique encoding!

5.5.2 Why Sinusoidal?

- **Unique patterns:** Each position has a distinct encoding
- **Relative positions:** The model can learn to attend to relative positions
- **Generalization:** Can extrapolate to longer sequences than seen during training

5.6 Residual Connections (Skip Connections)

Definition

Residual Connection:

$$\text{Output} = \text{LayerNorm}(x + \text{Sublayer}(x)) \tag{33}$$

We add the input directly to the output of each sub-layer.

5.6.1 Why Residual Connections?

- **Gradient Flow:** Gradients can flow directly through the addition operation
- **Training Stability:** Prevents weights from diverging to extreme values
- **Learning Residuals:** Network learns the *difference* from the identity function

5.7 Masked Attention in the Decoder

Important

The Autoregressive Constraint:
During generation, we can only use information from tokens that have already been generated. We **cannot** look at future tokens!
Solution: Mask out (set to $-\infty$) attention scores for future positions before softmax.

Example

Masking Example:
For "clouds drift across blue sky":
If generating position 3 ("across"):

- Can attend to: clouds, drift, across (✓)
- Cannot attend to: blue, sky (×)

The attention scores for "blue" and "sky" are set to $-\infty$, which becomes 0 after softmax.

6 BERT and GPT: Transformer Applications

6.1 Splitting the Transformer

The full Transformer was designed for sequence-to-sequence tasks like translation. But we can use its parts for different purposes:

Definition

BERT (Bidirectional Encoder Representations from Transformers):

- Uses only the **Encoder** part
- Bidirectional: Can see both past and future context
- Great for: Classification, NER, Question Answering
- Pre-training: Masked Language Model + Next Sentence Prediction

Definition

GPT (Generative Pre-trained Transformer):

- Uses only the **Decoder** part

- Unidirectional: Can only see past context (uses masking)
- Great for: Text generation, completion
- Pre-training: Next token prediction

6.2 The Power of Transfer Learning

Important

Pre-training + Fine-tuning:

1. **Pre-train** on massive text corpora (Wikipedia, books, internet)
2. **Fine-tune** on specific downstream tasks

The expensive pre-training is done once. Fine-tuning is fast and cheap.
Why this works: The model learns general language understanding during pre-training, which transfers to specific tasks.

7 Implementation Details

7.1 Transformer in Code

Listing 1: Multi-Head Attention Layer

```python
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

class MultiHeadAttention(layers.Layer):
    def __init__(self, d_model, num_heads):
        super().__init__()
        self.num_heads = num_heads
        self.d_model = d_model
        self.depth = d_model // num_heads

        # Learnable weight matrices
        self.wq = layers.Dense(d_model)
        self.wk = layers.Dense(d_model)
        self.wv = layers.Dense(d_model)
        self.dense = layers.Dense(d_model)

    def split_heads(self, x, batch_size):
        # Split last dimension into (num_heads, depth)
        x = tf.reshape(x, (batch_size, -1, self.num_heads, self.depth))
        return tf.transpose(x, perm=[0, 2, 1, 3])

    def call(self, q, k, v, mask=None):
        batch_size = tf.shape(q)[0]
```

```
25
26          # Linear projections
27          q = self.wq(q)  # (batch, seq_len, d_model)
28          k = self.wk(k)
29          v = self.wv(v)
30
31          # Split into heads
32          q = self.split_heads(q, batch_size)  # (batch, heads, seq, depth
                )
33          k = self.split_heads(k, batch_size)
34          v = self.split_heads(v, batch_size)
35
36          # Scaled dot-product attention
37          matmul_qk = tf.matmul(q, k, transpose_b=True)
38          dk = tf.cast(tf.shape(k)[-1], tf.float32)
39          scaled_attention = matmul_qk / tf.math.sqrt(dk)
40
41          if mask is not None:
42              scaled_attention += (mask * -1e9)
43
44          attention_weights = tf.nn.softmax(scaled_attention, axis=-1)
45          output = tf.matmul(attention_weights, v)
46
47          # Concatenate heads
48          output = tf.transpose(output, perm=[0, 2, 1, 3])
49          output = tf.reshape(output, (batch_size, -1, self.d_model))
50
51          return self.dense(output)
```

7.2 Positional Encoding Implementation

Listing 2: Positional Encoding

```
1  import numpy as np
2
3  def get_positional_encoding(max_seq_len, d_model):
4      """Generate␣sinusoidal␣positional␣encodings."""
5      position = np.arange(max_seq_len)[:, np.newaxis]
6      div_term = np.exp(np.arange(0, d_model, 2) * -(np.log(10000.0) /
           d_model))
7
8      pe = np.zeros((max_seq_len, d_model))
9      pe[:, 0::2] = np.sin(position * div_term)
10     pe[:, 1::2] = np.cos(position * div_term)
11
12     return tf.constant(pe, dtype=tf.float32)
13
14 # Example usage
15 max_len = 100
16 d_model = 512
```

```
pos_encoding = get_positional_encoding(max_len, d_model)
# Add to embeddings: embedded + pos_encoding[:seq_len, :]
```

7.3 Creating the Attention Mask

Listing 3: Creating Look-Ahead Mask for Decoder

```
def create_look_ahead_mask(size):
    """Create mask to prevent attending to future positions."""
    # Upper triangular matrix with ones above diagonal
    mask = 1 - tf.linalg.band_part(tf.ones((size, size)), -1, 0)
    return mask  # (size, size)

# Example: for sequence length 5
# [[0, 1, 1, 1, 1],
#  [0, 0, 1, 1, 1],
#  [0, 0, 0, 1, 1],
#  [0, 0, 0, 0, 1],
#  [0, 0, 0, 0, 0]]
# 1s become -inf after scaling, 0 after softmax
```

8 Comparing Architectures

Table 1: Comparison of Sequence Modeling Architectures

Feature	RNN/LSTM	RNN + Attention	Transformer
Parallel computation	No	Partial	Yes
Long-range dependencies	Difficult	Better	Excellent
Variable-length inputs	Yes	Yes	Yes
Vanishing gradients	Problem	Problem	Solved
Position awareness	Implicit	Implicit	Explicit (PE)
Training speed	Slow	Slow	Fast
Interpretability	Low	Medium	High

9 Practical Considerations

9.1 Computational Complexity

Warning

Attention is Quadratic:
For sequence length n, self-attention requires $O(n^2)$ computation and memory.

- $n = 100$: 10,000 attention scores
- $n = 1000$: 1,000,000 attention scores
- $n = 10000$: 100,000,000 attention scores!

This is why models like GPT have maximum context lengths (e.g., 4096 tokens).

9.2 Model Dimensions

Info

Typical Transformer Sizes:

- **Original Transformer:** $d_{model} = 512$, 6 layers, 8 heads
- **BERT-base:** $d_{model} = 768$, 12 layers, 12 heads
- **GPT-3:** $d_{model} = 12288$, 96 layers, 96 heads, 175B parameters

10 One-Page Summary

Summary

Key Concepts from Lecture 12:

1. Problems with RNNs:

- Vanishing gradients make learning long-range dependencies difficult
- Sequential processing prevents parallelization
- Fixed-size bottleneck loses information

2. Attention Mechanism (2015):

- Dynamic context vector: $C_t = \sum_j \alpha_{tj} h_j$
- Attention weights α are computed (not learned parameters)
- Allows direct connections between distant positions

3. Transformer Architecture (2017):

- "Attention Is All You Need" — removes recurrence entirely
- Query, Key, Value: $Q = HW_Q$, $K = HW_K$, $V = HW_V$
- Scaled dot-product: Attention $= \text{softmax}(QK^T/\sqrt{d_k})V$
- Multi-head attention for richer representations
- Positional encoding for order information

4. Key Formulas:

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

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

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

5. **BERT vs GPT:**
   - BERT = Encoder only (bidirectional, good for understanding)
   - GPT = Decoder only (unidirectional, good for generation)
6. **Why Transformers Dominate:**
   - Fully parallelizable training
   - No vanishing gradient problem
   - Excellent transfer learning capabilities
   - State-of-the-art on virtually all NLP tasks

11 Glossary

Attention Mechanism
A technique that allows models to focus on relevant parts of the input when producing each part of the output

Query (Q)
The representation of the current token that is "asking" for relevant information

Key (K)
The representation used to determine relevance/matching with the query

Value (V)
The actual content retrieved based on attention weights

Multi-Head Attention
Running multiple attention operations in parallel with different projections

Positional Encoding
Added signal that provides position information to the model

Transformer
Neural architecture using only attention (no recurrence)

Self-Attention
Attention where Q, K, V all come from the same sequence

Masked Attention
Attention that prevents looking at future positions

BERT
Bidirectional Encoder Representations from Transformers

GPT
Generative Pre-trained Transformer

Residual Connection
: Adding the input directly to the output of a layer

Layer Normalization
: Normalizing activations across features for each sample

Context Vector
: Weighted combination of hidden states based on attention

Alignment Score
: Raw similarity between query and key before softmax