CSC 4315: INTRODUCTION TO ARTIFICIAL INTELLIGENCE

Week 13, Lecture 25

Attention Mechanism and Self-Attention

Lecture Overview

This lecture introduces the attention mechanism that revolutionized NLP, including scaled dot-product attention, self-attention, and multi-head attention.

Topics Covered:

- Motivation for attention
- Scaled dot-product attention
- Self-attention mechanism
- Multi-head attention
- Causal masking for generation

1. Attention Mechanism

1.1 Motivation for Attention

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np

print("Attention Mechanism")
print("="*70)

print("Problem with RNN/LSTM:")
print("  - Sequential processing (slow)")
print("  - Information bottleneck through hidden state")
print("  - Struggle with very long sequences")

print("\nAttention solution:")
print("  - Direct connections to all positions")
print("  - Learn which positions are relevant")
print("  - Parallel computation")

print("\nCore idea:")
print("  'The cat sat on the mat because it was tired'")
print("  To understand 'it', look back at 'cat'")
print("  Attention learns these connections!")

print("\nAttention formula:")
print("  Attention(Q, K, V) = softmax(QK^T / sqrt(d_k)) V")
print("  - Q: Query (what I'm looking for)")
print("  - K: Key (what I have to offer)")
print("  - V: Value (what I actually give)")
```

```
Attention Mechanism
======================================================================
Problem with RNN/LSTM:
  - Sequential processing (slow)
  - Information bottleneck through hidden state
  - Struggle with very long sequences

Attention solution:
  - Direct connections to all positions
  - Learn which positions are relevant
  - Parallel computation

Core idea:
  'The cat sat on the mat because it was tired'
  To understand 'it', look back at 'cat'
  Attention learns these connections!

Attention formula:
  Attention(Q, K, V) = softmax(QK^T / sqrt(d_k)) V
  - Q: Query (what I'm looking for)
  - K: Key (what I have to offer)
  - V: Value (what I actually give)
```

1.2 Scaled Dot-Product Attention

```
print("Scaled Dot-Product Attention")
print("="*70)

def scaled_dot_product_attention(Q, K, V, mask=None):
    """
    Q, K, V: (batch, seq_len, d_k)
    """
    d_k = Q.size(-1)

    # Compute attention scores
    scores = torch.matmul(Q, K.transpose(-2, -1)) / np.sqrt(d_k)
    # scores: (batch, seq_len, seq_len)

    if mask is not None:
        scores = scores.masked_fill(mask == 0, -1e9)
    # Softmax over keys
    attention_weights = F.softmax(scores, dim=-1)

    # Apply to values
    output = torch.matmul(attention_weights, V)

    return output, attention_weights

# Example
batch_size, seq_len, d_k = 2, 4, 8
Q = torch.randn(batch_size, seq_len, d_k)
K = torch.randn(batch_size, seq_len, d_k)
V = torch.randn(batch_size, seq_len, d_k)

output, weights = scaled_dot_product_attention(Q, K, V)
print(f"Q, K, V shape: {Q.shape}")
print(f"Output shape: {output.shape}")
print(f"Weights shape: {weights.shape}")

print(f"\nAttention weights (sample):")
print(f"{weights[0].detach().numpy().round(3)}")
print("  Each row sums to 1 (softmax)")
print(f"  Row sums: {weights[0].sum(dim=-1).tolist()}")
```

```
Scaled Dot-Product Attention
======================================================================
Q, K, V shape: torch.Size([2, 4, 8])
Output shape: torch.Size([2, 4, 8])
Weights shape: torch.Size([2, 4, 4])

Attention weights (sample):
[[0.243 0.287 0.221 0.249]
 [0.198 0.312 0.241 0.249]
 [0.274 0.201 0.298 0.227]
 [0.231 0.256 0.247 0.266]]
  Each row sums to 1 (softmax)
  Row sums: [1.0, 1.0, 1.0, 1.0]
```

2. Self-Attention

2.1 Self-Attention Mechanism

```
print("Self-Attention")
print("="*70)

print("In self-attention: Q, K, V all come from same sequence")

class SelfAttention(nn.Module):
    def __init__(self, embed_dim):
        super().__init__()
        self.W_q = nn.Linear(embed_dim, embed_dim)
        self.W_k = nn.Linear(embed_dim, embed_dim)
        self.W_v = nn.Linear(embed_dim, embed_dim)
        self.scale = np.sqrt(embed_dim)

    def forward(self, x, mask=None):
        # x: (batch, seq_len, embed_dim)
        Q = self.W_q(x)
        K = self.W_k(x)
        V = self.W_v(x)

        scores = torch.matmul(Q, K.transpose(-2, -1)) / self.scale
        if mask is not None:
            scores = scores.masked_fill(mask == 0, -1e9)

        attn = F.softmax(scores, dim=-1)
        return torch.matmul(attn, V), attn

embed_dim = 64
self_attn = SelfAttention(embed_dim)
x = torch.randn(2, 5, 64)  # 2 sentences, 5 tokens

out, attn = self_attn(x)
print(f"Input: {x.shape}")
print(f"Output: {out.shape}")

print("\nSelf-attention learns:")
print("  - Which words relate to which")
print("  - 'it' attends to 'cat'")
print("  - 'bank' attends to 'river' or 'money' (context)")
```

```
Self-Attention
======================================================================
In self-attention: Q, K, V all come from same sequence
Input: torch.Size([2, 5, 64])
Output: torch.Size([2, 5, 64])

Self-attention learns:
  - Which words relate to which
  - 'it' attends to 'cat'
  - 'bank' attends to 'river' or 'money' (context)
```

2.2 Multi-Head Attention

```
print("Multi-Head Attention")
print("="*70)

print("Why multiple heads?")
print("  - Different heads learn different relationships")
print("  - Head 1: syntactic (subject-verb)")
print("  - Head 2: semantic (pronoun-noun)")
print("  - Head 3: positional (nearby words)")

class MultiHeadAttention(nn.Module):
    def __init__(self, embed_dim, num_heads):
        super().__init__()
        assert embed_dim % num_heads == 0

        self.embed_dim = embed_dim
        self.num_heads = num_heads
        self.head_dim = embed_dim // num_heads
        self.W_q = nn.Linear(embed_dim, embed_dim)
        self.W_k = nn.Linear(embed_dim, embed_dim)
        self.W_v = nn.Linear(embed_dim, embed_dim)
        self.W_o = nn.Linear(embed_dim, embed_dim)

    def forward(self, x, mask=None):
        batch, seq_len, _ = x.shape

        Q = self.W_q(x).view(batch, seq_len, self.num_heads, self.head_dim)
        K = self.W_k(x).view(batch, seq_len, self.num_heads, self.head_dim)
        V = self.W_v(x).view(batch, seq_len, self.num_heads, self.head_dim)

        # Transpose for attention: (batch, heads, seq, head_dim)
        Q, K, V = [t.transpose(1, 2) for t in [Q, K, V]]

        scores = torch.matmul(Q, K.transpose(-2, -1)) / np.sqrt(self.head_dim)
        if mask is not None:
            scores = scores.masked_fill(mask == 0, -1e9)

        attn = F.softmax(scores, dim=-1)
        out = torch.matmul(attn, V)

        # Concatenate heads
        out = out.transpose(1, 2).contiguous().view(batch, seq_len, self.embed_dim)
        return self.W_o(out)

mha = MultiHeadAttention(embed_dim=64, num_heads=8)
x = torch.randn(2, 10, 64)
out = mha(x)
print(f"Input: {x.shape}")
print(f"Output: {out.shape}")
print(f"\nNum heads: 8, Head dim: {64//8}")
```

```
Multi-Head Attention
======================================================================
Why multiple heads?
  - Different heads learn different relationships
  - Head 1: syntactic (subject-verb)
  - Head 2: semantic (pronoun-noun)
  - Head 3: positional (nearby words)
Input: torch.Size([2, 10, 64])
Output: torch.Size([2, 10, 64])

Num heads: 8, Head dim: 8
```

3. Causal Masking

3.1 Masked Self-Attention

```
print("Causal (Masked) Self-Attention")
print("="*70)

print("For language generation, can only attend to past tokens")

def create_causal_mask(seq_len):
    """Create lower triangular mask"""
    mask = torch.tril(torch.ones(seq_len, seq_len))
    return mask

mask = create_causal_mask(5)
print("Causal mask (5 tokens):")
print(mask.int())

print("\nToken 0: Can only see token 0")
print("Token 1: Can see tokens 0, 1")
print("Token 2: Can see tokens 0, 1, 2")
print("...")

print("\nWith masking:")
scores = torch.randn(5, 5)
masked_scores = scores.masked_fill(mask == 0, -1e9)
print(f"\nScores after masking:")
print(F.softmax(masked_scores, dim=-1).round(decimals=2))

print("\nUsed in:")
print("  - GPT-style decoder models")
print("  - Language generation")
print("  - Any autoregressive task")
```

```
Causal (Masked) Self-Attention
======================================================================
For language generation, can only attend to past tokens
Causal mask (5 tokens):
tensor([[1, 0, 0, 0, 0],
        [1, 1, 0, 0, 0],
        [1, 1, 1, 0, 0],
        [1, 1, 1, 1, 0],
        [1, 1, 1, 1, 1]])

Token 0: Can only see token 0
Token 1: Can see tokens 0, 1
Token 2: Can see tokens 0, 1, 2
...

With masking:

Scores after masking:
tensor([[1.00, 0.00, 0.00, 0.00, 0.00],
        [0.42, 0.58, 0.00, 0.00, 0.00],
        [0.31, 0.28, 0.41, 0.00, 0.00],
        [0.18, 0.32, 0.24, 0.26, 0.00],
        [0.15, 0.22, 0.19, 0.21, 0.23]])

Used in:
  - GPT-style decoder models
  - Language generation
  - Any autoregressive task
```

Summary

Key Takeaways:

- Attention allows direct connections between positions
- Attention computes weighted sum based on relevance

• Self-attention: Q, K, V from same sequence
• Multi-head learns different relationships
• Causal mask prevents attending to future
• Attention enables parallelization unlike RNNs

Practice Exercises:

1. Implement scaled dot-product attention
2. Visualize attention weights
3. Implement multi-head attention
4. Create causal mask
5. Compare attention vs LSTM on long sequences