CSC 4315: INTRODUCTION TO ARTIFICIAL INTELLIGENCE

Week 13, Lecture 26

Transformer Architecture Overview

Lecture Overview

This lecture covers the complete Transformer architecture including positional encoding, feed-forward networks, and the full encoder/decoder structure.

Topics Covered:

- Transformer overall structure
- Positional encoding
- Feed-forward network
- Layer normalization and residuals
- Complete encoder/decoder blocks

1. Transformer Architecture

1.1 Overall Structure

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

print("Transformer Architecture (Vaswani et al., 2017)")
print("="*70)

print('"Attention Is All You Need"')

print("\nOriginal Transformer has encoder-decoder structure:")
print("  Encoder: Process input sequence")
print("  Decoder: Generate output sequence")

print("\nKey components:")
print("  1. Multi-Head Self-Attention")
print("  2. Feed-Forward Network")
print("  3. Layer Normalization")
print("  4. Residual Connections")
print("  5. Positional Encoding")

print("\nEncoder block:")
print("  x -&gt; MultiHead(x) + x -&gt; LayerNorm")
print("  -&gt; FFN(x) + x -&gt; LayerNorm")

print("\nDecoder block:")
print("  x -&gt; Masked-MultiHead(x) + x -&gt; LayerNorm")
print("  -&gt; Cross-Attention(x, encoder_out) + x -&gt; LayerNorm")
print("  -&gt; FFN(x) + x -&gt; LayerNorm")
```

```
Transformer Architecture (Vaswani et al., 2017)
======================================================================
"Attention Is All You Need"

Original Transformer has encoder-decoder structure:
  Encoder: Process input sequence
  Decoder: Generate output sequence

Key components:
  1. Multi-Head Self-Attention
  2. Feed-Forward Network
  3. Layer Normalization
  4. Residual Connections
  5. Positional Encoding

Encoder block:
  x -&gt; MultiHead(x) + x -&gt; LayerNorm
  -&gt; FFN(x) + x -&gt; LayerNorm

Decoder block:
  x -&gt; Masked-MultiHead(x) + x -&gt; LayerNorm
  -&gt; Cross-Attention(x, encoder_out) + x -&gt; LayerNorm
  -&gt; FFN(x) + x -&gt; LayerNorm
```

2. Transformer Components

2.1 Positional Encoding

```
print("Positional Encoding")
print("="*70)

print("Problem: Attention has no sense of position!")
print("  'The cat sat on the mat' vs 'mat the on sat cat The'")
print("  Would produce same attention without position info")

print("\nSolution: Add positional encoding to embeddings")
print("  PE(pos, 2i) = sin(pos / 10000^(2i/d))")
print("  PE(pos, 2i+1) = cos(pos / 10000^(2i/d))")

class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_len=5000):
        super().__init__()
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len).unsqueeze(1).float()
        div_term = torch.exp(torch.arange(0, d_model, 2).float() *
                             (-np.log(10000.0) / d_model))

        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        pe = pe.unsqueeze(0)  # (1, max_len, d_model)
        self.register_buffer('pe', pe)

    def forward(self, x):
        return x + self.pe[:, :x.size(1)]

pe = PositionalEncoding(d_model=64)
x = torch.randn(2, 10, 64)  # batch=2, seq=10
out = pe(x)
print(f"Input: {x.shape}")
print(f"Output: {out.shape}")

print("\nWhy sinusoidal?")
print("  - Unique encoding for each position")
print("  - Can extrapolate to longer sequences")
print("  - Relative positions are learnable")
```

```
Positional Encoding
======================================================================
Problem: Attention has no sense of position!
  'The cat sat on the mat' vs 'mat the on sat cat The'
  Would produce same attention without position info

Solution: Add positional encoding to embeddings
  PE(pos, 2i) = sin(pos / 10000^(2i/d))
  PE(pos, 2i+1) = cos(pos / 10000^(2i/d))
Input: torch.Size([2, 10, 64])
Output: torch.Size([2, 10, 64])

Why sinusoidal?
  - Unique encoding for each position
  - Can extrapolate to longer sequences
  - Relative positions are learnable
```

2.2 Feed-Forward Network

```
print("Feed-Forward Network")
print("="*70)

print("Applied position-wise (same at each position)")
print("  FFN(x) = ReLU(xW1 + b1)W2 + b2")
print("  Inner dimension typically 4x embedding dim")

class FeedForward(nn.Module):
    def __init__(self, d_model, d_ff):
        super().__init__()
        self.linear1 = nn.Linear(d_model, d_ff)
        self.linear2 = nn.Linear(d_ff, d_model)
```

```
        self.dropout = nn.Dropout(0.1)

    def forward(self, x):
        return self.linear2(self.dropout(torch.relu(self.linear1(x))))

d_model, d_ff = 512, 2048
ffn = FeedForward(d_model, d_ff)
params = sum(p.numel() for p in ffn.parameters())
print(f"\nFFN(d_model={d_model}, d_ff={d_ff})")
print(f"Parameters: {params:,}")

x = torch.randn(2, 10, d_model)
out = ffn(x)
print(f"Input: {x.shape}")
print(f"Output: {out.shape}")
```

```
Feed-Forward Network
======================================================================
Applied position-wise (same at each position)
  FFN(x) = ReLU(xW1 + b1)W2 + b2
  Inner dimension typically 4x embedding dim

FFN(d_model=512, d_ff=2048)
Parameters: 2,099,712
Input: torch.Size([2, 10, 512])
Output: torch.Size([2, 10, 512])
```

3. Transformer Encoder Block

3.1 Complete Encoder Block

```
print("Complete Transformer Encoder Block")
print("="*70)

class TransformerEncoderBlock(nn.Module):
    def __init__(self, d_model, num_heads, d_ff, dropout=0.1):
        super().__init__()
        self.self_attn = nn.MultiheadAttention(d_model, num_heads, batch_first=True)
        self.ffn = FeedForward(d_model, d_ff)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, mask=None):
        # Self-attention with residual
        attn_out, _ = self.self_attn(x, x, x, attn_mask=mask)
        x = self.norm1(x + self.dropout(attn_out))

        # FFN with residual
        ffn_out = self.ffn(x)
        x = self.norm2(x + self.dropout(ffn_out))

        return x

block = TransformerEncoderBlock(d_model=512, num_heads=8, d_ff=2048)
x = torch.randn(2, 10, 512)
out = block(x)
print(f"Input: {x.shape}")
print(f"Output: {out.shape}")

params = sum(p.numel() for p in block.parameters())
print(f"Parameters: {params:,}")

print("\nTransformer Base (BERT/GPT):")
print("  d_model=768, num_heads=12, num_layers=12")
print("\nTransformer Large:")
print("  d_model=1024, num_heads=16, num_layers=24")
```

```
Complete Transformer Encoder Block
======================================================================
Input: torch.Size([2, 10, 512])
Output: torch.Size([2, 10, 512])
Parameters: 3,152,384

Transformer Base (BERT/GPT):
  d_model=768, num_heads=12, num_layers=12

Transformer Large:
  d_model=1024, num_heads=16, num_layers=24
```

3.2 PyTorch Transformer

```
print("PyTorch nn.Transformer")
print("="*70)

# Full encoder
encoder_layer = nn.TransformerEncoderLayer(
    d_model=512,
    nhead=8,
    dim_feedforward=2048,
    batch_first=True
)
encoder = nn.TransformerEncoder(encoder_layer, num_layers=6)

x = torch.randn(2, 10, 512)
out = encoder(x)
print(f"Encoder input: {x.shape}")
print(f"Encoder output: {out.shape}")

params = sum(p.numel() for p in encoder.parameters())
```

```
print(f"6-layer encoder params: {params:,}")

print("\nTransformer Variants:")
print("  Encoder-only: BERT (bidirectional)")
print("  Decoder-only: GPT (autoregressive)")
print("  Encoder-Decoder: T5, BART (seq2seq)")
```

```
PyTorch nn.Transformer
======================================================================
Encoder input: torch.Size([2, 10, 512])
Encoder output: torch.Size([2, 10, 512])
6-layer encoder params: 18,914,304

Transformer Variants:
  Encoder-only: BERT (bidirectional)
  Decoder-only: GPT (autoregressive)
  Encoder-Decoder: T5, BART (seq2seq)
```

Summary

Key Takeaways:

- Transformer uses attention instead of recurrence
- Positional encoding adds position information
- FFN processes each position independently
- Residual connections help gradient flow
- Layer norm stabilizes training
- Encoder-only, decoder-only, or both

Practice Exercises:

1. Implement positional encoding
2. Build complete encoder block
3. Compare pre-norm vs post-norm
4. Implement decoder with cross-attention
5. Train small transformer on toy task