Building GPT Architecture: Implementing Core Model Components

Prof. Rongyu Lin

Lecture 5

Quinnipiac University School of Computing and Engineering

Reading: Raschka Ch. 4 (pp. 96-149) "Building GPT Architecture"

Learning Objectives

By the end of this lecture, you will be able to:

- **Implement Causal Attention:** Apply masking techniques to prevent future token access
- **Construct Multi-Head Attention:** Combine multiple attention heads for richer representations
- **Build Neural Network Components:** Layer normalization, GELU activation, feed-forward networks
- **Assemble Transformer Blocks:** Integrate attention and feed-forward with shortcut connections
- **Code Complete GPT Architecture:** Build a functional GPT model ready for training
- **Understand Text Generation:** Token-by-token generation in autoregressive models

From attention mechanisms to complete GPT implementation

Course Context: Where We Are

Building Towards Complete LLM Understanding

Completed Foundation:

- Lecture 1: AI Historical Perspective
- Lecture 2: LLM Foundations & Pre-training
- Lecture 3: Text Data Processing & Tokenization
- Lecture 4: Attention Mechanisms

Today - Lecture 5: GPT Architecture

- Causal attention implementation
- Neural network building blocks
- Complete transformer assembly

Coming Next:

- Lecture 6: Pre-training
- Lecture 7: Fine-tuning
- Lecture 8: Advanced Architecture

Today: Bridge from attention to complete architecture

Section 1

Advanced Attention Mechanisms

Causal Masking & Multi-Head Implementation

Why Causal Attention? Preventing Future Information Leakage

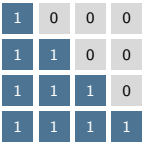
The Problem with Standard Attention

Standard Attention:

- All tokens attend to all tokens
- Bidirectional information flow
- Problem: Can't see the future!

Causal Attention Solution:

- Mask future token positions
- Maintain autoregressive property
- Each token attends only to previous positions



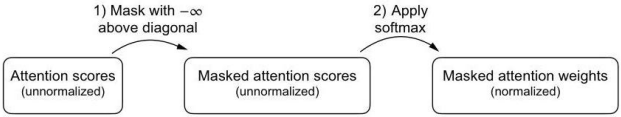
Causal Mask

Key Insight

Autoregressive generation requires preventing access to future information.

Causal Masking Matrix Visualization

Interactive Lower Triangular Matrix Example



Key Observation: Only lower triangular attention weights are computed

Before Masking:

- Blue: Raw attention scores (0.25)
- All positions accessible

After Masking + Softmax:

- Gray: Future positions become 0.0
- Gold: Renormalized valid weights

Key Insight

Each token can only attend to itself and previous tokens, preventing information leakage from future positions during training.

Mathematical Foundation of Causal Masking

```
# Create lower triangular mask
mask = torch.tril(torch.ones(seq_len, seq_len))

# Apply mask to attention scores
attention_scores = attention_scores.masked_fill(
    mask == 0, float('-inf'))
```

Attention Weight Modification
Standard Attention:

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

Causal Attention with Masking:

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

Mask Matrix M:

- $M_{i,j} = 0$ if $j \leq i$ (allowed)
- $M_{i,j} = -\infty$ if $j > i$ (masked)

Why $-\infty$?

After softmax, $\text{softmax}(-\infty) = 0$, eliminating attention to future positions.

Step-by-Step Causal Attention Implementation

```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class CausalAttention(nn.Module):
    def __init__(self, d_in, d_out,
                 context_length, dropout=0.0):
        super().__init__()
        self.d_out = d_out
        self.W_query = nn.Linear(d_in, d_out,
                                 bias=False)
        self.W_key = nn.Linear(d_in, d_out,
                               bias=False)
        self.W_value = nn.Linear(d_in, d_out,
                                 bias=False)
        self.dropout = nn.Dropout(dropout)

        # Register causal mask as buffer
        self.register_buffer(
            'mask',
            torch.tril(torch.ones(
                context_length, context_length))
        )
```

Building Causal Attention Layer Key

Design Decisions:

- `register_buffer`: Mask moves with model to GPU/CPU
- `bias=False`: Follows GPT-2 design
- `torch.tril`: Lower triangular matrix for masking

Architecture Notes:

- Creates query, key, and value projections
- Stores mask as non-trainable buffer
- Dropout for regularization
- Context length defines maximum sequence size

Efficient Masking Implementation

```python
def forward(self, x):
    b, num_tokens, d_in = x.shape  # batch, sequence, embedding

    # Generate queries, keys, values
    queries = self.W_query(x)  # (b, num_tokens, d_out)
    keys = self.W_key(x)       # (b, num_tokens, d_out)
    values = self.W_value(x)   # (b, num_tokens, d_out)

    # Compute attention scores
    attention_scores = queries @ keys.transpose(-2, -1)  # (b, num_tokens, num_tokens)
    attention_scores = attention_scores / (self.d_out ** 0.5)  # Scaling

    # Apply causal mask
    attention_scores = attention_scores.masked_fill(
        self.mask[:num_tokens, :num_tokens] == 0, float('-inf')
    )

    # Apply softmax and dropout
    attention_weights = F.softmax(attention_scores, dim=-1)
    attention_weights = self.dropout(attention_weights)

    # Generate context vectors
    context = attention_weights @ values  # (b, num_tokens, d_out)
    return context
```

Forward Pass with Masking Key

Implementation Steps:

- **Shape Extraction:** Get batch size, sequence length, and embedding dimensions
- **Linear Projections:** Transform input to queries, keys, and values
- **Attention Computation:** Calculate attention scores with scaling
- **Causal Masking:** Apply mask to prevent future information leakage
- **Normalization:** Softmax to get attention weights
- **Regularization:** Apply dropout for training stability
- **Context Generation:** Weight values to produce final output

Information Leakage Prevention: Why It Works

Mathematical Properties of Masked Softmax

Before Masking:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}} \quad (1)$$

After Masking:

$$\text{softmax}(z_i + M_i) = \frac{e^{z_i + M_i}}{\sum_j e^{z_j + M_j}} \quad (2)$$

Where $M_j = -\infty$ for $j > i$:

$$e^{z_j - \infty} = 0 \quad (3)$$

$$\text{Effectively: } \sum_j \rightarrow \sum_{j \leq i} \quad (4)$$

Weight Redistribution:

Before Masking:

0.25	0.25	0.25	0.25
T1	T2	T3	T4

After Masking:

1.0	0.0	0.0	0.0
T1	T2	T3	T4

Renormalization Effect

Masked positions get zero weight, remaining weights sum to 1.0. No information leakage.

♦ Hands-on Exercise: Implement Causal Masking

Interactive Coding Activity (5 minutes)

Your Task: Complete the causal masking implementation

```
1
2  import torch
3
4  def create_causal_mask(seq_length):
5      """Create a causal mask for attention."""
6      # TODO: Create lower triangular mask
7      mask = torch.tril(torch.ones(seq_length, seq_length))
8      return mask
9
10 def apply_causal_mask(attention_scores, mask):
11     """Apply causal mask to attention scores."""
12     # TODO: Apply mask using masked_fill
13     masked_scores = attention_scores.masked_fill(
14         mask == 0, float('-inf')
15     )
16     return masked_scores
17
18 # Test your implementation
19 seq_len = 4
20 mask = create_causal_mask(seq_len)
21 print("Causal mask:")
22 print(mask)
```

Key Concepts: **torch.tril():**

- Creates lower triangular matrix
- 1s below diagonal, 0s above

masked_fill():

- Replaces values where condition is True
- Uses $-\infty$ to zero out after softmax

Expected Output:

$$\text{mask} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \tag{5}$$

Tip: Test with seq_len=4 first!

Dropout for Regularization in Attention

Preventing Overfitting in Attention Weights

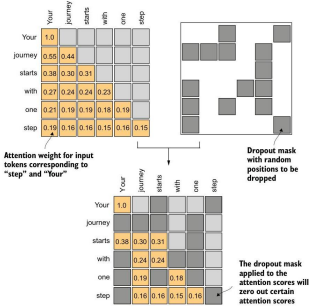
Why Dropout in Attention?

- Attention weights can become concentrated
- Model may over-rely on specific tokens
- Dropout forces distributed attention
- Improves generalization performance

Implementation Strategy:

- Apply dropout **after** softmax
- During training: randomly zero weights
- Remaining weights scaled up (compensation)
- During inference: use all weights

Dropout Visualization



Dropout Formula

$$\text{Dropout}(x) = \begin{cases} 0 & \text{with prob. } p \\ \frac{x}{1-p} & \text{with prob. } 1-p \end{cases}$$

Dropout Implementation in Attention

```python
class CausalAttention(nn.Module):
    def __init__(self, d_in, d_out, context_length,
      dropout=0.0):
        super().__init__()
        # ... (previous initialization code)
        self.dropout = nn.Dropout(dropout)  # Dropout layer

    def forward(self, x):
        # ... (compute attention_scores)

        # Apply softmax to get attention weights
        attention_weights = F.softmax(attention_scores, dim=-1)

        # Apply dropout AFTER softmax
        attention_weights = self.dropout(attention_weights)

        # Generate context vectors
        context = attention_weights @ values
        return context
```

PyTorch Implementation Details

Key Implementation Points:

- **Timing:** Dropout applied after softmax normalization
- **Training vs Inference:** Automatic handling with `model.train()/eval()`
- **Scaling:** PyTorch automatically handles $\frac{1}{1-p}$ scaling
- **Typical Values:** 0.0-0.1 for attention dropout

Dropout Benefits:

- Prevents overfitting to specific attention patterns
- Forces model to use diverse attention strategies
- Improves generalization performance
- Standard practice in transformer architectures

Complete Causal Attention Class

```python
class CausalAttention(nn.Module):
    def __init__(self, d_in, d_out, context_length,
      dropout=0.0, qkv_bias=False):
        super().__init__()
        self.d_out = d_out
        self.W_query = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_key = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_value = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.dropout = nn.Dropout(dropout)
        # Register buffer for causal mask
        self.register_buffer('mask',torch.tril(
            torch.ones(context_length, context_length)))
    def forward(self, x):
        b, num_tokens, d_in = x.shape
        queries = self.W_query(x)
        keys = self.W_key(x)
        values = self.W_value(x)
        attention_scores = queries @ keys.transpose(-2, -1)
        attention_scores = attention_scores / (self.d_out **
      0.5)
        attention_scores = attention_scores.masked_fill(
            self.mask[:num_tokens, :num_tokens] == 0,
      float('-inf'))
        attention_weights = F.softmax(attention_scores, dim=-1)
        attention_weights = self.dropout(attention_weights)
        context = attention_weights @ values
        return context
```

Full Implementation with Best Practices

Class Architecture:

- **Linear Layers:** Query, key, value projections
- **Causal Mask:** Lower triangular mask as buffer
- **Dropout:** Regularization after softmax
- **Scaling:** Temperature scaling with $\sqrt{d_{out}}$

Key Features:

- GPT-2 compatible (bias=False default)
- Efficient masking with register_buffer
- Flexible dropout and bias options
- Production-ready implementation

Usage:

- d_in=768, d_out=768 (GPT-2 Base)
- context_length=1024 or 2048
- dropout=0.1 (typical value)

Batch Processing in Causal Attention

Handling Multiple Sequences Efficiently

Key Tensor Transformations:

Operation	Input	Output
Input	(B, T, D_{in})	-
Linear (Q,K,V)	(B, T, D_{in})	(B, T, D_{out})
Attention Scores	(B, T, D_{out})	(B, T, T)
Masked Softmax	(B, T, T)	(B, T, T)
Context	(B, T, T)	(B, T, D_{out})

Batch Efficiency Considerations:

- **Mask Broadcasting:** Same mask for all batch elements
- **Parallel Processing:** All sequences processed simultaneously
- **Memory Scaling:** Usage scales as $O(B \times T^2)$

Memory Consideration

For long sequences, attention memory grows quadratically: 1000 tokens $\rightarrow$ 1M weights per head

Testing the Causal Attention Module

```python
# Test causal attention implementation
torch.manual_seed(123)

# Initialize causal attention
d_in, d_out = 768, 768
context_length = 1024
dropout = 0.1

causal_attn = CausalAttention(d_in, d_out, context_length,
      dropout)

# Test input
batch_size, seq_len = 2, 10
x = torch.randn(batch_size, seq_len, d_in)

# Forward pass
with torch.no_grad():
    context = causal_attn(x)

print(f"Input shape: {x.shape}")
print(f"Output shape: {context.shape}")
print(f"Shape preserved: {x.shape == context.shape}")

# Verify causal mask is working
print(f"Mask shape: {causal_attn.mask.shape}")
print(f"Mask (first 5x5):")
print(causal_attn.mask[:5, :5])
```

Validation and Shape Verification

Testing Strategy:

- **Reproducibility:** Fixed random seed for consistent results
- **Shape Testing:** Verify input/output dimension preservation
- **Mask Verification:** Check causal mask structure
- **Numerical Stability:** Ensure no NaN/Inf values

Expected Behavior:

- Input/output shapes should be identical
- Mask should be lower triangular
- Context vectors meaningful (not NaN/Inf)

Key Outputs:

- Input shape: torch.Size([2, 10, 768])
- Output shape: torch.Size([2, 10, 768])
- Shape preserved: True
- Mask: Lower triangular matrix

Multi-Head Attention: Parallel Representation Learning

Why Multiple Attention Heads?

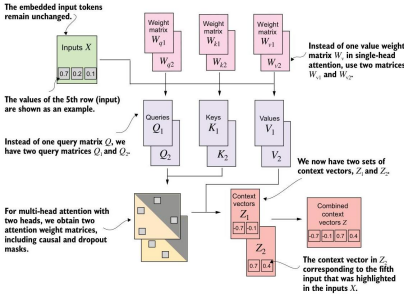
Single Head Limitations:

- One representation subspace
- Limited perspective on relationships
- Single attention pattern per layer

Multi-Head Advantages:

- Multiple subspaces: Different relationship aspects
- Diverse patterns: Syntactic + semantic attention
- Parallel processing: Computational efficiency
- Richer representations: Combined knowledge

Multi-Head Architecture



Mathematical Formulation:

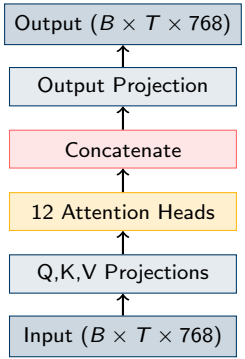
$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O$$

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

Detailed Multi-Head Attention Architecture

Complete 12-Head GPT Implementation

Simplified Architecture Flow



Key Components:

- **Input:** $B \times T \times 768$
- **Q,K,V:** 3 linear layers (768×768)
- **12 Heads:** Each processes 64 dims
- **Concat:** Merge all head outputs
- **Output:** Final projection layer

Head Dimensions:

- Total: 768 dimensions
- Per head: $768 \div 12 = 64$
- Parallel processing

Parameters:

- Q, K, V: 3×768^2
- Output: 768^2
- Total: 2.36M params

Simple Multi-Head Implementation: Wrapper Approach

```python
class MultiHeadAttentionWrapper(nn.Module):
    def __init__(self, d_in, d_out, context_length, dropout,
      num_heads):
        super().__init__()
        self.heads = nn.ModuleList([
            CausalAttention(d_in, d_out, context_length,
        dropout)
            for _ in range(num_heads)
        ])

    def forward(self, x):
        # Run each head independently
        head_outputs = [head(x) for head in self.heads]
        # Concatenate along the feature dimension
        return torch.cat(head_outputs, dim=-1)
```

Output Shape: If each head outputs (B, T, d_{out}), final is $(B, T, num\_heads \times d_{out})$

Usage Example: 12 heads $\times$ 64 dimensions $=$ 768 total (GPT-2 configuration)

Conceptually Clear Implementation

Architecture Design:

- **ModuleList:** Separate attention heads
- **Sequential Processing:** Each head runs independently
- **Concatenation:** Combine outputs along feature dimension

Pros and Cons:

- ✓ **Simple to understand**: Clear head separation
- ✓ **Easy debugging**: Individual head inspection
- × **Inefficient**: Multiple forward passes
- × **Memory overhead**: Separate parameters per head

Efficient Multi-Head Attention Implementation (Part 1)

Initialization and Setup

```python
class MultiHeadAttention(nn.Module):
    def __init__(self, d_in, d_out, context_length, dropout, num_heads):
        super().__init__()
        assert d_out % num_heads == 0, "d_out must be divisible by num_heads"
        self.d_out = d_out
        self.num_heads = num_heads
        self.head_dim = d_out // num_heads

        # Single large matrices instead of separate heads
        self.W_query = nn.Linear(d_in, d_out, bias=False)
        self.W_key = nn.Linear(d_in, d_out, bias=False)
        self.W_value = nn.Linear(d_in, d_out, bias=False)
        self.out_proj = nn.Linear(d_out, d_out)  # Output projection

        self.dropout = nn.Dropout(dropout)
        self.register_buffer('mask', torch.tril(torch.ones(
            context_length, context_length)))
```

Key Design Decisions:

- **Single Matrices:** One large matrix for all heads (efficient computation)
- **Head Dimension:** $d_{head} = d_{out}/num\_heads$ (typically 64)
- **Output Projection:** Final linear layer after concatenation
- **Causal Mask:** Registered as buffer for GPU compatibility

Efficient Multi-Head Attention Implementation (Part 2)

```python
def forward(self, x):
    b, num_tokens, d_in = x.shape
    # Generate Q, K, V matrices
    queries = self.W_query(x)  # (b, num_tokens, d_out)
    keys = self.W_key(x)       # (b, num_tokens, d_out)
    values = self.W_value(x)   # (b, num_tokens, d_out)
    # Reshape for multi-head processing
    queries = queries.view(b, num_tokens, self.num_heads, self.head_dim)
    queries = queries.transpose(1, 2)  # (b, num_heads, num_tokens, head_dim)
    keys = keys.view(b, num_tokens, self.num_heads, self.head_dim)
    keys = keys.transpose(1, 2)
    values = values.view(b, num_tokens, self.num_heads, self.head_dim)
    values = values.transpose(1, 2)
    # Compute attention scores and weights
    attention_scores = queries @ keys.transpose(-2, -1)
    attention_scores = attention_scores / (self.head_dim ** 0.5)
    attention_scores = attention_scores.masked_fill(
        self.mask[:num_tokens, :num_tokens] == 0, float('-inf'))
    attention_weights = F.softmax(attention_scores, dim=-1)
    attention_weights = self.dropout(attention_weights)
    # Apply attention and concatenate heads
    context = attention_weights @ values  # (b, num_heads, num_tokens, head_dim)
    context = context.transpose(1, 2).contiguous()
    context = context.view(b, num_tokens, self.d_out)
    # Final output projection
    context = self.out_proj(context)
    return context
```

Tensor Transformations:

- Input: (B, T, D_{in})
- After Linear: (B, T, D_{out})
- Reshape: (B, T, H, D_h)
- Transpose: (B, H, T, D_h)
- Final: (B, T, D_{out})

Efficiency Benefits:

- Single forward pass for all heads
- Better memory utilization
- GPU-friendly parallel computation
- Reduced parameter overhead

♦ Interactive Exercise: Multi-Head vs Single Head

```python
# Test both implementations
torch.manual_seed(123)
d_in, d_out = 768, 768
context_length = 1024
dropout = 0.0
num_heads = 12

# Create both versions
wrapper_mha = MultiHeadAttentionWrapper(d_in, d_out//num_heads,
    context_length, dropout, num_heads)
efficient_mha = MultiHeadAttention(d_in, d_out, context_length,
    dropout, num_heads)

# Test input
x = torch.randn(2, 10, d_in)

# TODO: Complete the comparison
with torch.no_grad():
    output_wrapper = wrapper_mha(x)
    output_efficient = efficient_mha(x)

print(f"Wrapper output shape:
      {output_wrapper.shape}")
print(f"Efficient output shape:
      {output_efficient.shape}")
```

```python
# TODO: Count parameters
wrapper_params = sum(p.numel()
    for p in wrapper_mha.parameters())
efficient_params = sum(p.numel()
    for p in efficient_mha.parameters())

print(f"Wrapper parameters:
      {wrapper_params}")
print(f"Efficient parameters:
      {efficient_params}")
print(f"Efficiency ratio:
      {efficient_params / wrapper_params:.2f}")
```

Discussion questions:

- Which implementation is more memory efficient?
- Which would be faster for large inputs?
- What are the trade-offs?

Section 2

Neural Network Building Blocks

GPT Architecture Components

GPT Model Overview: Three Stages of Development

From Components to Complete Language Model

Stage 1: Pre-training

- Next-token prediction on large corpus
- Learns general language patterns
- Unsupervised learning approach

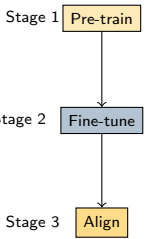
Stage 2: Supervised Fine-tuning

- Task-specific training data
- Classification, question answering, etc.
- Adapts knowledge to specific tasks

Stage 3: Alignment & RLHF

- Human feedback optimization
- Safety and helpfulness alignment
- Reinforcement learning from preferences

Development Pipeline



Current Status

Today: Building architecture foundation for pre-training
Next Lectures: Training procedures, fine-tuning, and alignment

GPT-2 Model Specifications: Scaling Parameters

Model Size Configurations

Model	Parameters	Layers	Heads	Embedding Dim
GPT-2 Small	124M	12	12	768
GPT-2 Medium	345M	24	16	1024
GPT-2 Large	762M	36	20	1280
GPT-2 XL	1.5B	48	25	1600

Our Focus: GPT-2 Small (124M) Configuration

- **Vocabulary Size:** 50,257 tokens (BPE encoding)
- **Context Length:** 1024 tokens maximum
- **Architecture:** 12 transformer blocks $\times$ 12 attention heads
- **Feed-forward:** 4$\times$ expansion (768 $\rightarrow$ 3072 $\rightarrow$ 768)
- **Training Data:** 40GB of internet text

Parameter Breakdown:

- Token Embeddings: ~39M
- Position Embeddings: ~0.8M
- Transformer Blocks: ~80M
- Output Layer: ~39M

Computational Requirements:

- Training: ~100 GPU-days
- Memory: ~500MB (inference)
- Speed: ~1000 tokens/sec

Model Configuration and Initialization

```python
# GPT-2 Small configuration
GPT_CONFIG_124M = {
    "vocab_size": 50257,      # Tokens
    "context_length": 1024,   # Seq length
    "emb_dim": 768,           # Embedding
    "n_heads": 12,            # Attention
    "n_layers": 12,           # Blocks
    "drop_rate": 0.1,         # Dropout
    "qkv_bias": False         # QKV bias
}

class GPTModel(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.tok_emb = nn.Embedding(
            cfg["vocab_size"], cfg["emb_dim"])
        self.pos_emb = nn.Embedding(
            cfg["context_length"], cfg["emb_dim"])
        self.drop_emb = nn.Dropout(
            cfg["drop_rate"])

        # Transformer blocks
        self.trf_blocks = nn.Sequential(*[
            TransformerBlock(cfg)
            for _ in range(cfg["n_layers"])
        ])
```

```python
        # Final layer norm and output
        self.final_norm = LayerNorm(cfg["emb_dim"])
        self.out_head = nn.Linear(
            cfg["emb_dim"], cfg["vocab_size"],
            bias=False)

    def forward(self, in_idx):
        batch_size, seq_len = in_idx.shape
        # Implementation details coming up...
```

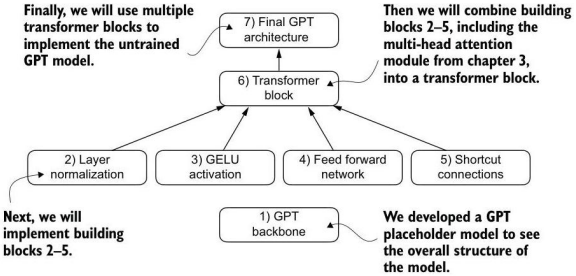
Configuration Benefits:
Easy experimentation with different model sizes.

Key Parameters:

- 50,257 vocabulary tokens
- 1024 maximum sequence length
- 768-dimensional embeddings
- 12 attention heads, 12 layers
- Configurable dropout rate

Data Flow Through GPT Model

Token Processing Pipeline



Key Transformations:

- **Tokenization:** Text → integer token IDs
- **Embedding:** Sparse IDs → dense vectors
- **Processing:** Self-attention and feed-forward transformations
- **Output:** Probability distribution over vocabulary

Need for Layer Normalization: Training Stability

Deep Network Training Challenges

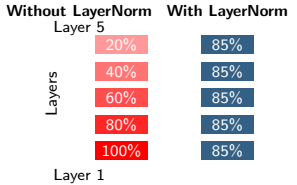
Problems in Deep Networks:

- **Vanishing Gradients:** Gradients become small
- **Exploding Gradients:** Gradients become large
- **Internal Covariate Shift:** Input distributions change
- **Slow Convergence:** Training takes very long

Layer Normalization Benefits:

- ✓ **Stable Training:** Consistent gradient flow
- ✓ **Faster Convergence:** Reduced training time
- ✓ **Less Sensitive:** To init and learning rate
- ✓ **Better Generalization:** Improved test performance

Gradient Flow Comparison



Core Insight

Layer norm ensures layers receive consistent input statistics (mean ≈ 0, std ≈ 1).

Layer Normalization Mathematics

Statistical Normalization Formula

Layer Normalization Formula:

$$\text{LayerNorm}(x) = \gamma \odot \frac{x - \mu}{\sigma + \epsilon} + \beta$$

Where:

- $\mu = \frac{1}{D}\sum_{i=1}^{D} x_i$ (mean across features)
- $\sigma = \sqrt{\frac{1}{D}\sum_{i=1}^{D}(x_i - \mu)^2}$ (standard deviation)
- γ = learnable scale parameter (init to 1)
- β = learnable shift parameter (init to 0)
- ϵ = small constant for stability (10^{-5})
- $\odot$ = element-wise multiplication

Normalization Process:

1. **Center:** Subtract mean $\rightarrow$ zero-centered
2. **Scale:** Divide by std $\rightarrow$ unit variance
3. **Transform:** Apply learnable γ, β $\rightarrow$ optimal scale/shift

Key Difference from Batch Norm

Layer norm: statistics **across features** per sample. Batch norm: **across samples** per feature.

Layer Normalization Implementation

```python
class LayerNorm(nn.Module):
    def __init__(self, emb_dim):
        super().__init__()
        self.eps = 1e-5
        self.scale = nn.Parameter(
            torch.ones(emb_dim))  # gamma
        self.shift = nn.Parameter(
            torch.zeros(emb_dim)) # beta

    def forward(self, x):
        # x: (batch_size, seq_len, emb_dim)

        # Compute mean and variance across
        # last dimension (features)
        mean = x.mean(dim=-1, keepdim=True)
        # Shape: (batch_size, seq_len, 1)
        var = x.var(dim=-1, keepdim=True,
                    unbiased=False)

        # Normalize
        norm_x = (x - mean) / torch.sqrt(
            var + self.eps)

        # Scale and shift
        return self.scale * norm_x + self.shift
```

Step-by-Step Implementation

```python
# Test implementation
torch.manual_seed(123)
layer_norm = LayerNorm(768)
x = torch.randn(2, 10, 768)
# (batch, seq_len, embedding_dim)

normalized = layer_norm(x)
print(f"Input mean: {x.mean():.4f}, std: {x.std():.4f}")
print(f"Output mean: {normalized.mean():.4f}, std:
      {normalized.std():.4f}")

# Check per-sample normalization
print(f"First sample mean:
      {normalized[0].mean(dim=-1)[:3]}")
# Should be ~0
print(f"First sample std:
      {normalized[0].std(dim=-1)[:3]}")
# Should be ~1
```

Key Points:

- Normalization across features (dim=-1)
- Learnable scale and shift parameters
- `unbiased=False` for GPT-2 compatibility

Layer Normalization Class with GPT-2 Compatibility

```python
class LayerNorm(nn.Module):
    def __init__(self, emb_dim):
        super().__init__()
        self.eps = 1e-5
        self.scale = nn.Parameter(
            torch.ones(emb_dim))
        self.shift = nn.Parameter(
            torch.zeros(emb_dim))

    def forward(self, x):
        mean = x.mean(dim=-1, keepdim=True)
        var = x.var(dim=-1, keepdim=True,
                    unbiased=False)
        norm_x = (x - mean) / torch.sqrt(
            var + self.eps)
        return self.scale * norm_x + self.shift
```

PyTorch Built-in Alternative:

```python
# Equivalent to our implementation
builtin_norm = nn.LayerNorm(emb_dim, eps=1e-5)
```

Production Implementation Details

Implementation Details:

- `unbiased=False`: Matches GPT-2 implementation exactly
- `keepdim=True`: Preserves dimensions for broadcasting
- `eps=1e-5`: Prevents division by zero, standard value
- `scale/shift`: Learnable parameters, identity init

Why from scratch?
Understanding math and ensuring GPT-2 compatibility.

Key Benefits:

- Exact reproducibility with published models
- Control over numerical precision
- Educational understanding

Biased vs. Unbiased Variance in Layer Normalization

Technical Detail: Variance Calculation

Two Variance Formulations:

Unbiased Variance (default in PyTorch):

$$\sigma^2_{\text{unbiased}} = \frac{1}{N-1}\sum_{i=1}^{N}(x_i - \mu)^2$$

Biased Variance (used in GPT-2):

$$\sigma^2_{\text{biased}} = \frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2$$

Why GPT-2 Uses Biased Variance:

- **Consistency:** Matches other deep learning frameworks (TensorFlow)
- **Numerical Stability:** Slightly more stable for small feature dimensions
- **Historical Reasons:** Original GPT implementation choice
- **Reproducibility:** Ensures exact match with published results

Practical Impact:

- Very small difference in practice (especially with large embedding dimensions)
- Important for exact reproducibility of published models
- Use `unbiased=False` in PyTorch to match GPT-2

Implementation Note

Always use `unbiased=False` when implementing GPT-2 to ensure exact compatibility with pre-trained weights.

♦ Hands-on Exercise: Layer Normalization

```python
import torch
import torch.nn as nn

# Complete the LayerNorm implementation
class CustomLayerNorm(nn.Module):
    def __init__(self, emb_dim, eps=1e-5):
        super().__init__()
        self.eps = eps
        # Create learnable parameters
        self.scale = nn.Parameter(
            torch.ones(emb_dim))
        self.shift = nn.Parameter(
            torch.zeros(emb_dim))

    def forward(self, x):
        # Compute mean across features
        mean = x.mean(dim=-1, keepdim=True)

        # Compute variance with unbiased=False
        var = x.var(dim=-1, keepdim=True,
                    unbiased=False)

        # Normalize and apply scale/shift
        norm_x = (x - mean) / torch.sqrt(
            var + self.eps)
        return self.scale * norm_x + self.shift
```

Interactive Implementation Activity (5 minutes)

```python
# Test your implementation
torch.manual_seed(123)
custom_norm = CustomLayerNorm(4)
pytorch_norm = nn.LayerNorm(4, eps=1e-5)

x = torch.randn(2, 3, 4)
# (batch, seq, features)
custom_output = custom_norm(x)
pytorch_output = pytorch_norm(x)
print("Custom implementation mean/std:")
print(f"Mean: {custom_output[0].mean(dim=-1)}")
print(f"Std: {custom_output[0].std(dim=-1,
      unbiased=False)}")
print("\nOutputs should be similar:")
print(f"Max difference: {torch.max(torch.abs(
      custom_output - pytorch_output))}")
```

Exercise Goals:

- Implement LayerNorm from scratch
- Compare with PyTorch built-in
- Verify numerical correctness
- Understand parameter roles

GELU Activation Function: Smooth Alternative to ReLU

Gaussian Error Linear Units

GELU Mathematical Definition:

$$\text{GELU}(x) = x \cdot \Phi(x)$$

where $\Phi(x)$ is the CDF of standard normal distribution.

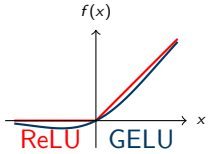
Intuitive Interpretation:

- Stochastic regularizer during training
- Smooth activation (differentiable everywhere)
- Combines sigmoid gating with input scaling
- Better gradient flow than ReLU

Advantages over ReLU:

- ✓ No "dead neurons" problem
- ✓ Smooth gradients everywhere
- ✓ Better performance in transformers
- ✓ Probabilistic interpretation

Activation Comparison



Approximate GELU (computationally efficient):

$$\text{GELU}(x) \approx 0.5x\left(1 + \tanh\left(\sqrt{\frac{2}{\pi}}(x + 0.044715x^3)\right)\right)$$

GELU Implementation: Exact vs. Approximate

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import math

class GELU(nn.Module):
    def __init__(self, approximate="none"):
        super().__init__()
        self.approximate = approximate

    def forward(self, x):
        if self.approximate == "none":
            # Exact GELU using error function
            return 0.5 * x * (1.0 + torch.erf(
                x / math.sqrt(2.0)))

        elif self.approximate == "tanh":
            # Tanh approximation
            # (faster, used in original GPT-2)
            return 0.5 * x * (
                1.0 + torch.tanh(
                    math.sqrt(2.0 / math.pi) *
                    (x + 0.044715 * torch.pow(x, 3.0))))
        else:
            raise ValueError(f"approximate must be 'none'
      or 'tanh'")
```

Different Implementation Strategies

```python
# PyTorch built-in GELU (recommended)
builtin_gelu = nn.GELU()
# Or functional version
def gelu_functional(x):
    return F.gelu(x)
# Performance comparison
x = torch.randn(1000, 768)
exact_gelu = GELU("none")
approx_gelu = GELU("tanh")
# Test outputs (should be very similar)
exact_out = exact_gelu(x)
approx_out = approx_gelu(x)
builtin_out = builtin_gelu(x)
print(f"Max difference (exact vs approx):
      {torch.max(torch.abs(exact_out - approx_out)):.6f}")
print(f"Max difference (exact vs builtin):
      {torch.max(torch.abs(exact_out - builtin_out)):.6f}")
```

Implementation Choices:

- **Exact:** Uses error function, slower
- **Tanh:** Approximation, faster
- **Built-in:** PyTorch optimized

Feed-Forward Networks: Position-wise Processing

Feed-Forward Network Structure:

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

```
class FeedForward(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(cfg["emb_dim"],
                      4 * cfg["emb_dim"]),
            GELU(),nn.Linear(4 * cfg["emb_dim"],
                      cfg["emb_dim"]),)

    def forward(self, x):
        return self.layers(x)
```

Neural Network Layer in Transformers

Key Properties:

- **Position-wise:** Applied to each position independently
- **4x Expansion:** Hidden dimension is 4× embedding dimension
- **GELU Activation:** Smooth, non-linear transformation
- **Parameter Heavy:** Most parameters in transformer block

Architecture Details:

- Input: $(B, T, 768)$
- Hidden: $(B, T, 3072)$ (4× expansion)
- Output: $(B, T, 768)$
- Parameters: ∼4.7M per layer

Feed-Forward 4× Expansion: Design Rationale

Why 4× Expansion in GPT Architectures

Mathematical Foundation:

- For embedding dimension d_{model}, feed-forward hidden dimension is $d_{ff} = 4 \times d_{model}$
- GPT-2 Small: $d_{model} = 768$, $d_{ff} = 3072$

Design Rationale:

- **Representational Capacity:** Larger hidden space allows complex transformations
- **Information Bottleneck:** Expansion → compression forces learning useful features
- **Non-linearity:** GELU activation creates rich feature interactions
- **Empirical Optimization:** 4× found optimal through experimentation

Parameter Analysis for GPT-2 Small:

Layer	Shape	Parameters
Linear 1	(768, 3072)	2,359,296
Linear 2	(3072, 768)	2,359,296
Per FF Block		**4,718,592**
12 FF Blocks		**56,623,104**

Alternative Ratios Explored:

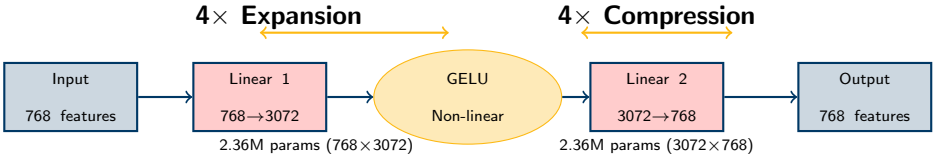
- 2× expansion: Too constraining, poor performance
- 8× expansion: Marginal gains, 2× parameter cost
- 4× expansion: Sweet spot for parameter efficiency vs. performance

Design Insight

Feed-forward networks contain 45.6% of total model parameters. The 4× expansion provides optimal balance between model capacity and computational efficiency.

Feed-Forward 4× Expansion: Visual Architecture

Visual Feed-Forward Expansion Process



Processing Flow:

- Position-wise transformation
- No inter-position communication
- Parallel computation across sequence

Computational Cost:

- 2× matrix multiplications per block
- 4.7M parameters per FF block
- Dominates model parameter count

Parameter Distribution Analysis: Feed-Forward Dominance

Understanding Parameter Allocation in GPT Models

GPT-2 Small Parameter Breakdown:

Component	% of Total
Token Embeddings	31.1%
Output Layer	31.1%
Feed-Forward	45.6%
Attention	22.8%
Position Embeddings	0.6%
Layer Norms	0.01%

Per-Layer Parameter Count:

Component	Parameters	% of Block
Multi-Head Attention	2,359,296	33.3%
Feed-Forward Network	4,718,592	66.7%
Layer Normalization ($2\times$)	1,536	0.02%
Total per Block	**7,079,424**	**100%**

Implications:

- **Memory Bottleneck:** Feed-forward layers determine GPU memory requirements
- **Computational Cost:** Matrix multiplication dominates training time
- **Optimization Focus:** Most gradient updates occur in FF parameters
- **Scaling Decisions:** Model size primarily determined by FF expansion

When scaling from GPT-2 Small to Large:

- Embedding dimension: $768 \rightarrow 1280$ ($1.67\times$ increase)
- FF parameters: $56.6M \rightarrow 157.3M$ ($2.78\times$ increase)
- FF dominates parameter growth

Position-wise Processing: Independent Feature Transformation

Position-wise Independence:

$$\text{FFN}(\mathbf{X})_{i,:} = \text{FFN}(\mathbf{X}_{i,:})$$

```python
def demonstrate_position_independence():
    """Show that FFN processes each
    position independently"""
    batch_size, seq_len, emb_dim = 2, 5, 768
    # Create feed-forward network
    ff = FeedForward({"emb_dim": emb_dim})
    # Input sequence
    x = torch.randn(batch_size, seq_len, emb_dim)
    # Method 1: Process entire sequence
    output_batch = ff(x)
    # Method 2: Process each position separately
    output_separate = torch.zeros_like(x)
    for i in range(seq_len):
        output_separate[:, i, :] = ff(
            x[:, i:i+1, :]).squeeze(1)
    # Should be identical
    max_diff = torch.max(torch.abs(output_batch -
      output_separate))
    print(f"Maximum difference: {max_diff.item():.10f}")
    print("Position-wise independence verified!"
          if max_diff < 1e-6 else "Error!")

demonstrate_position_independence()
```

Computational Advantages:

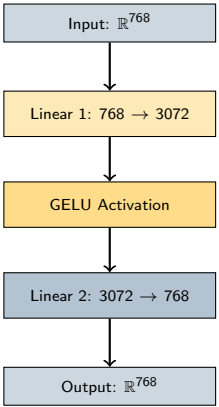
- **Parallelization:** All positions processed simultaneously
- **Memory Locality:** Efficient matrix operations
- **Hardware Utilization:** GPU-friendly computation pattern

Theoretical Implications:

- No cross-positional information mixing in FF layers
- Attention handles sequence relationships, FF handles feature transformation
- Enables efficient batched computation across all positions

Expansion/Contraction Visualization: Information Processing

Feed-Forward Information Flow



Information Processing Pattern:

- **Expansion:** Projects input to higher-dimensional space
- **Non-linearity:** GELU enables complex feature interactions
- **Contraction:** Compresses back to original dimensionality
- **Integration:** Residual connection preserves information flow

Think of feed-forward as a "feature factory":

- Expand raw features into intermediate representations
- Apply complex transformations in high-dimensional space
- Compress useful features back to standard size
- Add processed features to original input

Mathematical Interpretation:

- Expansion: $\mathbf{h}_1 = \text{GELU}(X\mathbf{W}_1 + \mathbf{b}_1)$
- Contraction: $\mathbf{h}_2 = \mathbf{h}_1\mathbf{W}_2 + \mathbf{b}_2$
- Information bottleneck forces learning of compressed,

Memory Usage and Computational Impact Analysis

Resource Requirements of Feed-Forward Networks

Memory Analysis (GPT-2 Small, Batch Size = 8, Sequence Length = 1024):

Component	Memory (MB)	% of Total
Parameters		
FF Weights (all layers)	217.4	43.7%
Attention Weights	108.7	21.9%
Embeddings	149.6	30.1%
Activations (Forward Pass)		
FF Intermediate (3072 dim)	96.0	19.3%
FF Output (768 dim)	24.0	4.8%
Attention Activations	48.0	9.7%
Training Overhead		
FF Gradients	217.4	43.7%
FF Optimizer States (Adam)	434.8	87.4%

Computational Cost (FLOPs per token):

- **FF Layer 1:** $768 \times 3072 = 2.36$M multiply-adds
- **GELU:** ≈ 3072 operations (negligible)
- **FF Layer 2:** $3072 \times 768 = 2.36$M multiply-adds
- **Total per FF:** $\approx 4.7M$ FLOPs
- **All 12 layers:** $\approx 56M$ FLOPs per token

Optimization Strategies:

- **Mixed Precision:** Use FP16 to halve memory requirements
- **Activation Checkpointing:** Trade compute for memory
- **Gradient Accumulation:** Simulate larger batches with less memory
- **Expert Layers:** Sparse activation patterns (MoE models)

Memory Bottleneck

Feed-forward layers are the primary memory and computational bottleneck in transformer training.

Complete FeedForward Class Implementation

```python
class FeedForward(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        # Expansion factor (typically 4 for GPT-2)
        self.expansion_factor = 4
        hidden_dim = cfg["emb_dim"] *
      self.expansion_factor
        # Two linear transformations with GELU
        self.w1 = nn.Linear(cfg["emb_dim"],
                            hidden_dim, bias=True)
        self.w2 = nn.Linear(hidden_dim,
                            cfg["emb_dim"], bias=True)
        # Activation function
        # (GELU for GPT-2 compatibility)
        self.activation = GELU()
        # Optional dropout for regularization
        self.dropout = nn.Dropout(
            cfg.get("ff_dropout", 0.0))
        # Initialize weights
        self._init_weights()
    def _init_weights(self):
        # Xavier initialization
        nn.init.xavier_uniform_(self.w1.weight)
        nn.init.xavier_uniform_(self.w2.weight)
        # Small bias initialization
        nn.init.constant_(self.w1.bias, 0.0)
        nn.init.constant_(self.w2.bias, 0.0)
```

```python
    def forward(self, x):
        """
        Args:
            x: Input tensor (batch_size, seq_len, emb_dim)
        Returns:
            Output tensor (batch_size, seq_len, emb_dim)
        """
        # Expand to higher dimension
        h = self.w1(x)
        # (batch_size, seq_len, 4*emb_dim)
        # Apply non-linear activation
        h = self.activation(h)
        # Optional dropout
        h = self.dropout(h)
        # Contract back to original dimension
        output = self.w2(h)
        # (batch_size, seq_len, emb_dim)
        return output
```

Best Practices:

- Xavier weight initialization
- Configurable dropout
- Clear documentation
- Proper bias handling

TransformerBlock: Combining Components

```python
class TransformerBlock(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.att = MultiHeadAttention(
            d_in=cfg["emb_dim"],
            d_out=cfg["emb_dim"],
            context_length=cfg["context_length"],
            num_heads=cfg["n_heads"],
            dropout=cfg["drop_rate"],
            qkv_bias=cfg["qkv_bias"]
        )
        self.ff = FeedForward(cfg)
        self.norm1 = LayerNorm(cfg["emb_dim"])
        self.norm2 = LayerNorm(cfg["emb_dim"])
        self.drop_shortcut = nn.Dropout(
            cfg["drop_rate"])

    def forward(self, x):
        # Attention block with shortcut connection
        shortcut = x
        x = self.norm1(x)
        x = self.att(x)
        x = self.drop_shortcut(x)
        x = x + shortcut  # Shortcut connection
```

```python
        # Feed-forward block with shortcut connection
        shortcut = x
        x = self.norm2(x)
        x = self.ff(x)
        x = self.drop_shortcut(x)
        x = x + shortcut  # Shortcut connection

        return x
```

Key Components:

- **Multi-Head Attention:** Global context
- **Feed-Forward:** Position-wise transformation
- **Layer Normalization:** Stable training
- **Residual Connections:** Gradient flow
- **Dropout:** Regularization

Processing Order:

1. LayerNorm → Attention → Dropout → Add
2. LayerNorm → FeedForward → Dropout → Add

Complete Transformer Block Data Flow Architecture

Visual Architecture Overview

Component Legend:

- **LN**: LayerNorm
- **MHA**: Multi-Head Attention
- **FFN**: Feed-Forward Network
- **Drop**: Dropout
- **+**: Residual Add
- **Red**: Skip connections

Parameters (GPT-2 Small):

- Attention: 2.36M per block
- Feed-Forward: 4.72M per block
- Total per block: 7.1M

Memory Usage:

- Input/Output: $B \times T \times 768$
- FF Hidden: $B \times T \times 3072$ (peak)
- Attention: $B \times 12 \times T \times T$

Complete GPT Model Implementation

```python
class GPTModel(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.tok_emb = nn.Embedding(
            cfg["vocab_size"], cfg["emb_dim"])
        self.pos_emb = nn.Embedding(
            cfg["context_length"], cfg["emb_dim"])
        self.drop_emb = nn.Dropout(
            cfg["drop_rate"])

        self.trf_blocks = nn.Sequential(*[
            TransformerBlock(cfg)
            for _ in range(cfg["n_layers"])
        ])

        self.final_norm = LayerNorm(
            cfg["emb_dim"])
        self.out_head = nn.Linear(
            cfg["emb_dim"], cfg["vocab_size"],
            bias=False)
```

```python
    def forward(self, in_idx):
        batch_size, seq_len = in_idx.shape
        tok_embeds = self.tok_emb(in_idx)
        pos_embeds = self.pos_emb(
            torch.arange(seq_len,
                         device=in_idx.device))
        x = tok_embeds + pos_embeds
        x = self.drop_emb(x)
        x = self.trf_blocks(x)
        x = self.final_norm(x)
        logits = self.out_head(x)
        return logits
```

Architecture Summary:

- Token + Position embeddings
- 12 Transformer blocks
- Final LayerNorm
- Output projection to vocabulary
- Total: 124M parameters

GPT Model: Token and Position Embedding Integration

```python
class GPTModel(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        # Token embeddings: map token IDs to vectors
        self.tok_emb = nn.Embedding(
            cfg["vocab_size"], cfg["emb_dim"])
        # Position embeddings: add positional info
        self.pos_emb = nn.Embedding(
            cfg["context_length"], cfg["emb_dim"])
        # Dropout for regularization
        self.drop_emb = nn.Dropout(cfg["drop_rate"])
        # Transformer blocks (defined separately)
        self.trf_blocks = nn.Sequential(*[
            TransformerBlock(cfg)
            for _ in range(cfg["n_layers"])])
        # Final layer norm and output projection
        self.final_norm = LayerNorm(cfg["emb_dim"])
        self.out_head = nn.Linear(cfg["emb_dim"],
      cfg["vocab_size"], bias=False)
```

```python
    def forward(self, in_idx):
        batch_size, seq_len = in_idx.shape
        # Get token embeddings
        tok_embeds = self.tok_emb(in_idx)
        # (batch, seq_len, emb_dim)
        # Get position embeddings
        pos_indices = torch.arange(
            seq_len, device=in_idx.device)
        pos_embeds = self.pos_emb(pos_indices)
        # (seq_len, emb_dim)
        # Combine embeddings
        x = tok_embeds + pos_embeds
        # Apply dropout
        # Process through transformer blocks
        x = self.trf_blocks(x)
        # Final normalization and projection
        x = self.final_norm(x)
        logits = self.out_head(x)
        return logits
```

Key Implementation Details:

- **Position Broadcasting:** Position embeddings broadcast across batch
- **Embedding Addition:** Token + position embeddings combined element-wise
- **Dropout Regularization:** Applied to combined embeddings

Parameter Counting and Memory Analysis

Detailed GPT-2 Small Parameter Breakdown

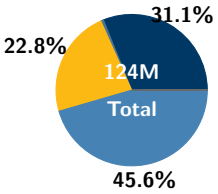
Component	Computation	Parameters	% of Total
	Embedding Layers		
Token Embedding	$50,257 \times 768$	38,597,376	31.1%
Position Embedding	$1,024 \times 768$	786,432	0.6%
	Transformer Blocks (12 layers)		
Multi-Head Attention	$768 \times 768 \times 4 \times 12$	28,311,552	22.8%
Feed-Forward Networks	$(768 \times 3072 + 3072 \times 768) \times 12$	56,623,104	45.6%
Layer Normalizations	$768 \times 2 \times 12$	18,432	0.01%
	Output Layer		
Final Layer Norm	768	768	0.001%
Output Projection	$768 \times 50,257$	38,597,376	31.1%
Total Parameters		**124,439,808**	**100%**

Memory Requirements (Float32):

- **Parameters:** $124M \times 4$ bytes $= 498$ MB
- **Gradients:** Additional 498 MB during training
- **Optimizer States:** $2\times$ gradients for Adam $= 996$ MB
- **Activations:** Varies by batch size and sequence length
- **Total Training:** 2-4 GB GPU memory minimum

GPT Parameter Distribution Visualization

GPT-2 Small (124M) Parameter Breakdown



Component Breakdown:

- **Token Embeddings:** 38.6M (31.1%)
- **Position Embeddings:** 0.8M (0.6%)
- **Multi-Head Attention:** 28.3M (22.8%)
- **Feed-Forward Networks:** 56.6M (45.6%)
- **Layer Normalizations:** 0.02M (0.01%)

Key Insights:

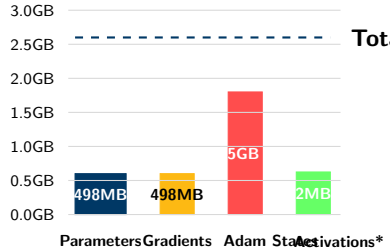
- Feed-forward layers dominate parameter count
- Attention mechanisms use 22.8% of parameters
- Embeddings require significant memory
- Layer norms contribute minimally
- Weight tying can eliminate output layer

Engineering Insight

FFN layers scale quadratically with hidden dimension ($d_{model} \times 4d_{model}$), making them the primary bottleneck in larger models.

Memory Usage Breakdown for GPT Training

Detailed Memory Analysis for GPT-2 124M



Memory Formulas:

$$\text{Parameters} = N_p \times 4 \text{ bytes} \quad (6)$$

$$\text{Gradients} = N_p \times 4 \text{ bytes} \quad (7)$$

$$\text{Adam States} = N_p \times 12 \text{ bytes} \quad (8)$$

$$\text{Activations} = f(B, L, d) \quad (9)$$

Where: N_p = parameter count, B = batch size, L = sequence length, d = hidden dimension

Scaling Insights:

- Adam optimizer dominates memory usage (3× parameters)
- Activations scale with batch size and sequence length
- Mixed precision (FP16) can halve memory requirements
- Gradient checkpointing trades compute for memory

Weight Tying: Sharing Embeddings and Output Parameters

Concept: Share weights between token embedding and output projection layers.

```python
class GPTModel(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.tok_emb = nn.Embedding(
            cfg["vocab_size"], cfg["emb_dim"])
        self.pos_emb = nn.Embedding(
            cfg["context_length"], cfg["emb_dim"])
        self.drop_emb = nn.Dropout(cfg["drop_rate"])
        self.trf_blocks =
      nn.Sequential(*[TransformerBlock(cfg)
        for _ in range(cfg["n_layers"])])
        self.final_norm = LayerNorm(cfg["emb_dim"])

        # No separate output layer - use embedding weights
        # self.out_head = nn.Linear(
        #     cfg["emb_dim"], cfg["vocab_size"],
        #     bias=False)
```

```python
    def forward(self, in_idx):
        batch_size, seq_len = in_idx.shape
        # Standard forward pass
        tok_embeds = self.tok_emb(in_idx)
        pos_indices = torch.arange(seq_len, device=in_idx.device)
        pos_embeds = self.pos_emb(pos_indices)
        x = tok_embeds + pos_embeds
        x = self.drop_emb(x)
        x = self.trf_blocks(x)
        x = self.final_norm(x)
        # Weight tying: use embedding weights transposed
        logits = F.linear(x, self.tok_emb.weight)
        # Shape: (batch, seq_len, vocab_size)
        return logits
```

Benefits of Weight Tying:

- **Parameter Reduction:** Saves 38.6M parameters (31.1% reduction)
- **Improved Generalization:** Forces consistent representations
- **Memory Efficiency:** Reduces GPU memory requirements

Mathematical Justification

If embedding maps token i to vector $\mathbf{e}_i$, then logits for predicting token j are $\mathbf{h} \cdot \mathbf{e}_j$ - same vector space.

Model Configuration Scaling: GPT-2 Family

Systematic Architecture Scaling

Configuration	Small	Medium	Large	XL
Parameters	124M	345M	762M	1.5B
Layers (n_layers)	12	24	36	48
Attention Heads	12	16	20	25
Embedding Dim	768	1024	1280	1600
Feed-Forward Dim	3072	4096	5120	6400
Context Length	1024	1024	1024	1024

Scaling Relationships:

- **FF Dimension:** Always $4\times$ embedding dimension
- **Head Dimension:** $d_{head} = d_{emb}/n_{heads}$ (typically 64-80)
- **Parameter Growth:** Mostly from feed-forward expansion
- **Constant Context:** All models use 1024 token limit

Performance Implications:

- Larger models show better language understanding
- Exponentially more compute required for training
- Memory requirements scale with model size
- Inference time increases with depth and width

Scaling Law

Model performance improves as a power law with respect to model size, dataset size, and compute budget.

Model Configuration Management in Code

```python
# Configuration dictionary for easy model switching
GPT_CONFIGS = {
    "gpt2-small": {
        "vocab_size": 50257,"context_length": 1024,
        "emb_dim": 768,"n_heads": 12,
        "n_layers": 12, "drop_rate": 0.1,
        "qkv_bias": False},

    "gpt2-medium": {
        "vocab_size": 50257,"context_length": 1024,
        "emb_dim": 1024,"n_heads": 16,
        "n_layers": 24, "drop_rate": 0.1,
        "qkv_bias": False},

    "gpt2-large": {
        "vocab_size": 50257,"context_length": 1024,
        "emb_dim": 1280,"n_heads": 20,
        "n_layers": 36,"drop_rate": 0.1,
        "qkv_bias": False}}

# Initialize any model size
config = GPT_CONFIGS.get("gpt2-medium")
model = GPTModel(config)
# Count parameters
total_params = sum(p.numel() for p in model.parameters())
print(f"Total parameters: {total_params:,}")
```

Benefits of Configuration-Based Design:

- Easy experimentation with model sizes
- Consistent architecture across scales
- Simplified hyperparameter management
- Reproducible model instantiation

Complete GPT Model Code Implementation

```python
class GPTModel(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.tok_emb = nn.Embedding(
            cfg["vocab_size"], cfg["emb_dim"])
        self.pos_emb = nn.Embedding(
            cfg["context_length"], cfg["emb_dim"])
        self.drop_emb = nn.Dropout(cfg["drop_rate"])

        # Stack of transformer blocks
        self.trf_blocks = nn.Sequential(*[
            TransformerBlock(cfg)
            for _ in range(cfg["n_layers"])])

        # Final processing
        self.final_norm = LayerNorm(cfg["emb_dim"])
        self.out_head = nn.Linear(
            cfg["emb_dim"], cfg["vocab_size"],
            bias=False)
```

```python
    def forward(self, in_idx):
        batch_size, seq_len = in_idx.shape

        # Embedding layer
        tok_embeds = self.tok_emb(in_idx)
        pos_indices = torch.arange(
            seq_len, device=in_idx.device)
        pos_embeds = self.pos_emb(pos_indices)
        x = tok_embeds + pos_embeds
        x = self.drop_emb(x)

        # Transformer processing
        x = self.trf_blocks(x)

        # Output projection
        x = self.final_norm(x)
        logits = self.out_head(x)

        return logits
```

Key Features:

- **Modular Design:** Easy to modify and extend
- **Device Agnostic:** Handles GPU/CPU automatically
- **Configurable:** All hyperparameters externalized
- **Standard PyTorch:** Compatible with all PyTorch tools

Model Testing and Shape Verification

```python
def test_gpt_model():
    torch.manual_seed(123)
    cfg = GPT_CONFIG_124M
    model = GPTModel(cfg)
    # Test 1: Shape verification
    batch_size, seq_len = 2, 10
    input_ids = torch.randint(0, cfg["vocab_size"],
      (batch_size, seq_len))

    with torch.no_grad():
        logits = model(input_ids)

    print("Shape Tests:")
    print(f"Input: {input_ids.shape}")
    print(f"Output: {logits.shape}")
    print(f"Expected: ({batch_size}, {seq_len},
      {cfg['vocab_size']})")
    assert logits.shape == (batch_size, seq_len,
      cfg["vocab_size"])
```

```python
    # Test 2: Parameter count
    total_params = sum(p.numel()
        for p in model.parameters())
    print(f"\nParameter Count: {total_params:,}")
    print(f"Expected: ~124M parameters")

    # Test 3: Gradient flow (requires_grad check)
    trainable_params = sum(p.numel() for p in
      model.parameters() if p.requires_grad)
    print(f"Trainable parameters:  {trainable_params:,}")

    # Test 4: Memory usage
    model_memory = sum(p.numel() * 4
        for p in model.parameters()) / 1024**2  # MB
    print(f"Model memory: {model_memory:.1f} MB")
    print("\nAll tests passed! Model ready for training.")

# Run comprehensive tests
test_gpt_model()
```

Validation Checklist:

- ✓ Input/output shape consistency
- ✓ Parameter count matches specification
- ✓ All parameters require gradients
- ✓ Memory usage within expected range
- ✓ No runtime errors or warnings

♦ Hands-on Exercise: Build Complete GPT Model

```python
# Complete the missing parts of the GPT model
class GPTModel(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        # TODO: Initialize token embedding layer
        self.tok_emb = nn.Embedding(
            cfg["vocab_size"], cfg["emb_dim"])

        # TODO: Initialize position embedding layer
        self.pos_emb = nn.Embedding(
            cfg["context_length"], cfg["emb_dim"])

        # TODO: Add embedding dropout
        self.drop_emb = nn.Dropout(cfg["drop_rate"])

        # TODO: Create transformer blocks stack
        self.trf_blocks = nn.Sequential(*[
            TransformerBlock(cfg)
            for _ in range(cfg["n_layers"])])

        # TODO: Add final layer norm and output head
        self.final_norm = LayerNorm(cfg["emb_dim"])
        self.out_head = nn.Linear(
            cfg["emb_dim"], cfg["vocab_size"],
            bias=False)
```

```python
    def forward(self, in_idx):
        batch_size, seq_len = in_idx.shape
        # TODO: Get token embeddings
        tok_embeds = self.tok_emb(in_idx)

        # TODO: Get position embeddings
        pos_indices = torch.arange(seq_len,
      device=in_idx.device)
        pos_embeds = self.pos_emb(pos_indices)

        # TODO: Combine embeddings and apply dropout
        x = tok_embeds + pos_embeds
        x = self.drop_emb(x)

        # TODO: Process through transformer blocks
        x = self.trf_blocks(x)

        # TODO: Apply final norm and output projection
        x = self.final_norm(x)
        logits = self.out_head(x)

        return logits
# Test your implementation
model = GPTModel(GPT_CONFIG_124M)
test_input = torch.randint(0, 1000, (2, 5))
output = model(test_input)
print(f"Success! Output shape: {output.shape}")
```

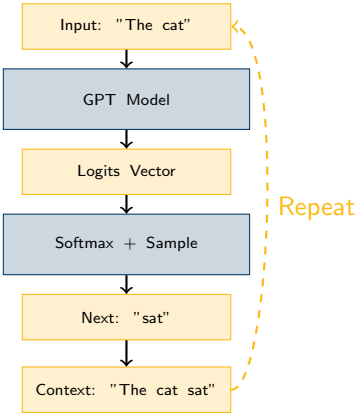
Section 4

Text Generation Detailed Process

Autoregressive Generation Mechanics

Autoregressive Text Generation Overview

Generation Pipeline:



Mathematical Formulation:

$$P(\text{seq}) = \prod_{t=1}^{T} P(x_t | x_{1:t-1})$$

Generation Steps:

1. **Context Input:** Feed current sequence
2. **Forward Pass:** Compute logits for next token
3. **Apply Temperature:** Scale logits by τ
4. **Softmax:** Convert to probabilities
5. **Sample:** Choose next token
6. **Append:** Add to sequence
7. **Repeat:** Until max length or EOS

Sampling Strategies:

- **Greedy:** arg max (deterministic)
- **Temperature:** $p_i \propto e^{z_i/\tau}$
- **Top-k:** Sample from top k tokens
- **Top-p:** Nucleus sampling

Context Window Management and Cropping

Context Cropping Strategy:

```python
def generate_with_cropping(model, idx, max_new_tokens,
      context_size):
    """Generate text with context window management"""
    model.eval()

    for _ in range(max_new_tokens):
        # Crop context if it exceeds maximum length
        idx_cropped = idx[:, -context_size:]  # Keep last
      context_size tokens

        with torch.no_grad():
            # Forward pass with cropped context
            logits = model(idx_cropped)

            # Get logits for the last position only
            logits = logits[:, -1, :]  # (batch_size,
      vocab_size)

            # Apply softmax and sample
            probs = torch.softmax(logits, dim=-1)
            idx_next = torch.multinomial(probs, num_samples=1)

            # Append new token (full sequence continues to
      grow)
            idx = torch.cat((idx, idx_next), dim=1)

    return idx
```

Handling Long Sequences The Challenge: GPT models have fixed maximum context length (1024 tokens for GPT-2).

Cropping Visualization:

Full Sequence	Cropped Context (Last 1024)
[token1, token2, ..., token1500]	[..., token477, ..., token1500]

Trade-offs:

- ✓ **Memory Efficient:** Constant memory usage
- ✓ **Consistent Performance:** No slowdown with length
- × **Limited Memory:** Loses old context information

Step-by-Step Generation Process

```python
def generate_text_detailed(model, tokenizer, prompt,
    max_new_tokens=50):
    model.eval()
    # Step 1: Tokenize initial prompt
    input_ids = tokenizer.encode(prompt)
    input_tensor = torch.tensor([input_ids])

    generated_tokens = input_tensor.clone()
    for step in range(max_new_tokens):
        with torch.no_grad():
            logits = model(generated_tokens)
            next_token_logits = logits[0, -1, :]
            probabilities = torch.softmax(next_token_logits,
  dim=-1)
            next_token_id = torch.multinomial(probabilities,
  num_samples=1).item()
            next_token_tensor = torch.tensor(
  [[next_token_id]])
            generated_tokens = torch.cat([generated_tokens,
  next_token_tensor], dim=1)
            next_token_text = tokenizer.decode([next_token_id])
            print(f"Step {step+1}: '{next_token_text}'")
            if next_token_id == tokenizer.eos_token_id:
                print("Reached EOS token.")
                break
    # Final decoding
    final_text = tokenizer.decode(generated_tokens[0].tolist())
    return final_text
```

Code Explanation:

Initialization (Lines 1-11):

- Set model to evaluation mode
- Tokenize input prompt
- Create tensor with batch dimension
- Initialize generation buffer

Generation Loop (Lines 13-40):

- **Forward Pass:** Compute logits for all positions
- **Extract Last:** Get predictions for next token
- **Softmax:** Convert logits to probabilities
- **Sampling:** Use multinomial to select token
- **Concatenate:** Add new token to sequence
- **Decode:** Convert token ID to text
- **Stop Check:** Exit if EOS encountered

Output Processing (Lines 42-44):

- Decode entire token sequence
- Return complete generated text

Generation Complexity

Time: $O(n \times L)$ where n = new tokens, L = sequence length

Sampling Strategies Beyond Greedy Decoding

```python
def sample_with_strategies(logits,
                           strategy="greedy",
                           temperature=1.0,
                           top_k=50):

    """Various sampling strategies for text generation"""
    if strategy == "greedy":
        # Always pick highest probability token
        next_token = torch.argmax(logits, dim=-1)

    elif strategy == "random":
        # Sample from full probability distribution
        probs = torch.softmax(
            logits / temperature, dim=-1)
        next_token = torch.multinomial(
            probs, num_samples=1).squeeze()

    elif strategy == "top_k":
        # Only sample from top-k tokens
        top_k_logits, top_k_indices =
            torch.topk(logits, top_k)
        top_k_probs = torch.softmax(
            top_k_logits / temperature, dim=-1)
        sampled_idx = torch.multinomial(
            top_k_probs, num_samples=1)
        next_token = top_k_indices[sampled_idx]
```

```python
    elif strategy == "temperature":
        # Use temperature scaling for
        # controlled randomness
        scaled_logits = logits / temperature
        probs = torch.softmax(scaled_logits, dim=-1)
        next_token = torch.multinomial(
            probs, num_samples=1).squeeze()

    return next_token
```

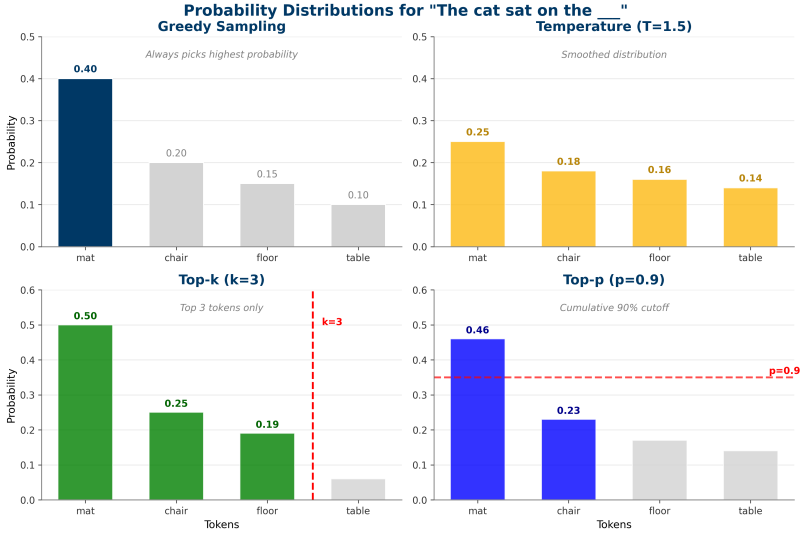
Temperature Effects:

- $T = 0.1$: Very focused, deterministic-like
- $T = 1.0$: Standard probability distribution
- $T = 2.0$: More random, creative output

Strategy Comparison:

- **Greedy:** Deterministic, coherent but repetitive
- **Random:** Creative but potentially incoherent
- **Top-k:** Balanced diversity and quality
- **Temperature:** Fine-grained randomness control

Sampling Strategies: Probability Distributions



Sampling Strategies: Formulations and Trade-offs

Mathematical Formulations:

Greedy: $w_t = \arg\max_i P(w_i|w_{<t})$
Always selects most probable token

Temperature: $P'(w_i) = \frac{\exp(z_i/T)}{\sum_j \exp(z_j/T)}$
$T<1$: deterministic, $T>1$: random

Top-k: Sample from k highest probabilities
Fixed vocabulary diversity

Top-p: Sample from tokens with $\sum P(w_i) \leq p$
Adaptive vocabulary based on confidence

Trade-offs and Characteristics:

- **Greedy:** ✓ Deterministic, coherent × Repetitive
- **Temperature:** ✓ Smooth randomness control × May pick unlikely tokens
- **Top-k:** ✓ Fixed vocabulary diversity × Ignores confidence
- **Top-p:** ✓ Adaptive vocabulary ✓ Confidence-based

Typical Parameter Values:

- Temperature: T = 0.7–1.2
- Top-k: k = 40–80
- Top-p: p = 0.9–0.95

Production Insight

Modern systems often combine strategies: temperature scaling with top-p filtering for optimal quality-diversity balance.

Complete Text Generation Implementation (Part 1)

```python
def generate_text_simple(model, idx, max_new_tokens,
                         context_size, temperature=1.0,
                         top_k=None, eos_token_id=None):
    """
    Generate text using the GPT model with various
      sampling strategies.
    Args:
        model: Trained GPT model
        idx: Starting token indices
             (batch_size, seq_len)
        max_new_tokens: Maximum number of tokens to
      generate
        context_size: Maximum context window
        temperature: Sampling temperature
        top_k: Sample from top-k tokens only
        eos_token_id: End-of-sequence token ID
    Returns:
        Generated token sequence
    """
    model.eval()

    for _ in range(max_new_tokens):
        # Crop context if exceeds window
        if idx.shape[1] > context_size:
            idx_cond = idx[:, -context_size:]
        else:
            idx_cond = idx
```

Function Parameters Explained:

Essential Parameters:

- **model**: The trained GPT model in eval mode
- **idx**: Initial token tensor (batch $\times$ seq)
- **max_new_tokens**: Generation limit
- **context_size**: Model's max context (1024 for GPT-2)

Sampling Controls:

- **temperature**: Controls randomness
  - < 1.0: More focused/deterministic
  - $= 1.0$: Standard sampling
  - > 1.0: More random/creative
- **top_k**: Limits vocabulary for sampling
- **eos_token_id**: Early stopping trigger

Design Pattern

Function provides flexible generation with multiple sampling strategies in a single interface.

Complete Text Generation Implementation (Part 2)

```
1       with torch.no_grad():
2           logits = model(idx_cond)
3           # Extract last token's predictions
4           logits = logits[:, -1, :] / temperature
5           # Apply top-k filtering if specified
6           if top_k is not None:
7               # Get top-k logits and indices
8               top_k_logits, top_k_indices = \
9                   torch.topk(logits, top_k, dim=-1)
10              # Create filtered logits tensor
11              logits_filtered = torch.full_like(
12                  logits, float('-inf'))
13              # Scatter top-k values back
14              logits_filtered.scatter_(1, top_k_indices,
        top_k_logits)
15              logits = logits_filtered
16          # Convert to probabilities
17          probs = torch.softmax(logits, dim=-1)
18          # Sample next token
19          idx_next = torch.multinomial(
20              probs, num_samples=1)
21          # Append to sequence
22          idx = torch.cat((idx, idx_next), dim=1)
23          # Check for early stopping
24          if eos_token_id and (idx_next ==
        eos_token_id).any():
25              break
26  return idx
```

Generation Process Breakdown:

1. **Forward Pass (Lines 1-6):**
   - Disable gradients with `torch.no_grad()`
   - Get logits for all positions
   - Extract last position predictions
   - Apply temperature scaling
2. **Top-k Filtering (Lines 8-20):**
   - Select k highest probability tokens
   - Set all other logits to $-\infty$
   - Ensures sampling from limited vocabulary
   - Improves generation quality
3. **Sampling (Lines 22-27):**
   - Apply softmax for probabilities
   - Use multinomial sampling
   - Concatenate new token to sequence
4. **Termination (Lines 29-33):**
   - Check for EOS token
   - Early exit if found

Performance Considerations and Optimization

Efficient Text Generation

Generation Performance Bottlenecks:

- **Sequential Nature:** Cannot parallelize generation steps
- **Memory Growth:** Context grows with each token
- **Repeated Computation:** Recalculating attention for same tokens
- **Model Size:** Large models slow on single GPU

Optimization Strategies:

Technique	Benefit	Trade-off
KV-Cache	10-50× faster	Higher memory usage
Batch Generation	2-4× throughput	Padding overhead
Model Quantization	2-4× memory reduction	Slight quality loss
Speculative Decoding	2-3× faster	Complex implementation
Context Cropping	Constant memory	Limited context

KV-Cache Optimization (Advanced):

- **Benefit:** Avoid recomputing attention for existing tokens
- **Implementation:** Modify attention to accept cached KV
- **Memory Cost:** $O(\text{layers} \times \text{seq\_len} \times d_{model})$

Typical Generation Speeds:

- GPT-2 Small (124M): 50-100 tokens/sec (GPU)
- GPT-3 (175B): 1-5 tokens/sec (GPU)
- With optimization: 2-10× improvement possible

♦ Hands-on Exercise: Implement Text Generation

```python
def generate_text_exercise(model, idx, max_new_tokens,
                           context_size):

    for step in range(max_new_tokens):
        # TODO: Apply context cropping
        idx_cond = idx[:, -context_size:] \
            if idx.shape[1] > context_size else idx

        with torch.no_grad():
            # TODO: Forward pass to get logits
            logits = model(idx_cond)
            # TODO: Get logits for last position
            logits = logits[:, -1, :]
            # TODO: Apply temperature scaling
            # (use temperature=0.8)
            temperature = 0.8
            logits = logits / temperature
            # TODO: Convert to probabilities
            probs = torch.softmax(logits, dim=-1)
            # TODO: Sample next token
            idx_next = torch.multinomial(
                probs, num_samples=1)
            # TODO: Append to sequence
            idx = torch.cat((idx, idx_next), dim=1)
            print(f"Step {step+1}: Generated token \
                  {idx_next.item()}")
    return idx
```

Interactive Generation Implementation (8 minutes)

```python
# Test your implementation
torch.manual_seed(42)
model = GPTModel(GPT_CONFIG_124M)  # Untrained model
start_tokens = torch.randint(1000, 5000, (1, 3))
# Random starting tokens
generated = generate_text_exercise(
    model, start_tokens, max_new_tokens=5,
    context_size=1024)
print(f"Generated sequence: \
      {generated[0].tolist()}")
print("Note: Untrained model produces \
      random tokens!")
```

Extension Challenges:

1. Implement top-k sampling
2. Add early stopping for special tokens
3. Compare different temperature values
4. Measure generation speed

Section 5

Transformer Architecture Assembly

Connecting Components into Complete Blocks

Shortcut Connections: The Key to Deep Training

ResNet-Style Skip Connections

Mathematical Formulation:

$$\mathbf{h}_{l+1} = \mathbf{h}_l + F(\mathbf{h}_l, \mathbf{W}_l)$$

Where:

- $\mathbf{h}_l$ = input to layer l
- $F(\mathbf{h}_l, \mathbf{W}_l)$ = transformation function
- Addition creates direct path for gradients

In Transformer Blocks:

- **Attention Shortcut:** $x + \text{Attention}(x)$
- **Feed-Forward Shortcut:** $x + \text{FeedForward}(x)$
- **Gradient Flow:** Direct path to early layers
- **Identity Mapping:** $F(x) = 0 \Rightarrow$ identity function

Critical Insight

Without shortcut connections, training 12+ layer transformers would be nearly impossible due to vanishing gradients.

Mathematical Proof of Gradient Vanishing

Chain Rule in Deep Networks:
For a network with L layers, the gradient at layer l is:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(l)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(L)}} \prod_{i=l+1}^{L} \frac{\partial \mathbf{z}^{(i)}}{\partial \mathbf{z}^{(i-1)}}$$

Problem Analysis: If each layer transformation has Jacobian with spectral norm $\rho < 1$:

$$\left\| \prod_{i=l+1}^{L} \frac{\partial \mathbf{z}^{(i)}}{\partial \mathbf{z}^{(i-1)}} \right\| \leq \rho^{L-l}$$

Gradient Decay Example:

- Layer 12: $\rho^0 = 1.0$ (full gradient)
- Layer 6: $\rho^6 \approx 0.5^6 = 0.016$ (98.4% reduction)
- Layer 1: $\rho^{11} \approx 0.5^{11} = 0.0005$ (99.95% reduction)

Shortcut Connection Solution:
With residual connection $\mathbf{h}_{l+1} = \mathbf{h}_l + F(\mathbf{h}_l)$:

$$\frac{\partial \mathbf{h}_{l+1}}{\partial \mathbf{h}_l} = \mathbf{I} + \frac{\partial F(\mathbf{h}_l)}{\partial \mathbf{h}_l}$$

The identity matrix $\mathbf{I}$ ensures gradient flow even if $\frac{\partial F}{\partial \mathbf{h}_l} \to 0$.

Gradient Flow Analysis: Empirical Demonstration

```python
def analyze_gradient_flow(model, loss):
    """Analyze gradient magnitudes across layers"""
    # Compute gradients
    loss.backward()
    gradient_norms = {}
    for name, param in model.named_parameters():
        if param.grad is not None:
            grad_norm = param.grad.norm().item()
            gradient_norms[name] = grad_norm
    return gradient_norms
# Simulate training step
model_without_shortcuts =
    DeepNetworkNoShortcuts(layers=12)
model_with_shortcuts = GPTModel(GPT_CONFIG_124M)
# Forward pass and loss
x = torch.randn(2, 10, 768)
target = torch.randint(0, 50257, (2, 10))

loss_no_shortcuts = F.cross_entropy(
    model_without_shortcuts(x).view(-1, 768),
    target.view(-1))
loss_with_shortcuts = F.cross_entropy(
    model_with_shortcuts(
        torch.randint(0, 50257, (2, 10))
    ).view(-1, 50257),
    target.view(-1)
)
```

```python
# Analyze gradients
grads_no_shortcuts = analyze_gradient_flow(
    model_without_shortcuts, loss_no_shortcuts)
grads_with_shortcuts = analyze_gradient_flow(
    model_with_shortcuts, loss_with_shortcuts)

# Compare gradient magnitudes by layer depth
print("Layer-wise Gradient Norms:")
for layer in range(12):
    print(f"Layer {layer+1}:
          No shortcuts={
          grads_no_shortcuts[f'layer{layer}']:.6f},
          With shortcuts={

      grads_with_shortcuts[f'transformer{layer}']:.6f}")
```

Expected Results:

- **Without shortcuts:** Exponential decay from layer 12 $\rightarrow$ 1
- **With shortcuts:** Relatively stable gradients across all layers

Training Stability: Quantitative Analysis

Empirical Benefits of Shortcut Connections

Training Metrics Comparison:

Metric	Without Shortcuts	With Shortcuts
Convergence Time	5-10× slower	Baseline
Final Loss	Higher (worse)	Lower (better)
Gradient Variance	High instability	Stable
Learning Rate Sensitivity	Very sensitive	Robust
Parameter Utilization	30-50% layers	90-95% layers
Training Success Rate	20-40% runs	95-98% runs

ResNet Historical Impact:

- **Pre-ResNet (2015):** Networks deeper than 8-10 layers very difficult to train
- **Post-ResNet:** Successfully trained 50, 101, 152+ layer networks
- **Transformer Adoption:** All transformer architectures use residual connections
- **Modern Scale:** Enables training of 12B+ parameter models

Mathematical Properties:

- **Identity Mapping:** If $F(\mathbf{x}) = 0$, residual block becomes identity
- **Gradient Preservation:** $\nabla_{\mathbf{x}}(\mathbf{x} + F(\mathbf{x})) = \mathbf{I} + \nabla_{\mathbf{x}} F(\mathbf{x})$
- **Error Surface:** Smoother optimization landscape
- **Depth Scalability:** Linear scaling with network depth possible

Training Insight

Residual connections transform very difficult optimization problem into tractable one.

♦ Hands-on Exercise: Compare Gradient Flows

```python
def compare_gradient_flows():
    """Compare gradient flows with and without
    residual connections"""
    torch.manual_seed(123)

    # Define simple networks for comparison
    class NetworkNoSkip(nn.Module):
        def __init__(self):
            super().__init__()
            self.layers = nn.Sequential(*[
                nn.Linear(768, 768)
                for _ in range(6)])
        def forward(self, x):
            return self.layers(x)
    class NetworkWithSkip(nn.Module):
        def __init__(self):
            super().__init__()
            self.layers = nn.ModuleList([
                nn.Linear(768, 768)
                for _ in range(6)])
        def forward(self, x):
            for layer in self.layers:
                x = x + layer(x)  # Residual
            return x
    # TODO: Initialize both networks
    net_no_skip = NetworkNoSkip()
    net_with_skip = NetworkWithSkip()
```

```python
    # TODO: Create dummy input and target
    x = torch.randn(4, 10, 768)
    target = torch.randn(4, 10, 768)
    # TODO: Forward pass and compute losses
    out_no_skip = net_no_skip(x)
    out_with_skip = net_with_skip(x)
    loss_no_skip = F.mse_loss(out_no_skip, target)
    loss_with_skip = F.mse_loss(out_with_skip, target)
    # TODO: Compute gradients
    loss_no_skip.backward()
    loss_with_skip.backward()
    # TODO: Compare gradient norms
    print("Gradient norms by layer:")
    print("No Skip | With Skip")
    for i, (p1, p2) in enumerate(
        zip(net_no_skip.parameters(),
            net_with_skip.parameters())):
        if p1.grad is not None and
           p2.grad is not None:
            norm1 = p1.grad.norm().item()
            norm2 = p2.grad.norm().item()
            print(f"Layer {i}: {norm1:.6f} |
                  {norm2:.6f}")
compare_gradient_flows()
```

Vanishing Gradient Problem in Deep Networks

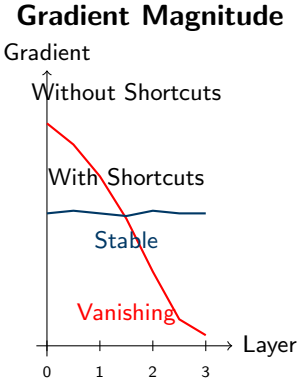
Why We Need Shortcut Connections

The Problem:

- Gradients become exponentially smaller in deeper layers
- Chain rule: $\frac{\partial L}{\partial x_1} = \frac{\partial L}{\partial x_n} \prod_{i=1}^{n-1} \frac{\partial x_{i+1}}{\partial x_i}$
- If derivatives ¡ 1, product approaches zero
- Early layers learn slowly or not at all

Impact on Training:

- Slow convergence or no learning in early layers
- Network effectively becomes shallow
- Poor utilization of model capacity
- Difficulty training very deep networks



Mathematical Example: If each layer has gradient $\frac{\partial x_{i+1}}{\partial x_i} = 0.5$, after 10 layers:

Model Parameter Counting: Understanding Scale

GPT-2 Small Parameter Breakdown

Component-wise Parameter Count:

Component	Shape	Parameters
Token Embedding	(50257, 768)	38.6M
Position Embedding	(1024, 768)	0.8M
12 × Transformer Blocks	-	85.1M
Multi-Head Attention	(768, 768) × 4	2.3M each
Feed-Forward	(768, 3072, 768)	4.7M each
Layer Norms	(768) × 2	1.5K each
Final Layer Norm	(768)	768
Output Head	(768, 50257)	38.6M
Total		**124.4M**

Key Insights:

- **Embeddings:** 62% of parameters (input + output)
- **Feed-Forward:** Dominates transformer block parameters
- **Attention:** Relatively parameter-efficient

Model Initialization and Testing

```python
import torch
torch.manual_seed(123)

# Initialize model with GPT-2 Small config
model = GPTModel(GPT_CONFIG_124M)

# Test forward pass
batch_size, seq_len = 2, 4
token_ids = torch.randint(
    0, GPT_CONFIG_124M["vocab_size"],
    (batch_size, seq_len))

print("Input shape:", token_ids.shape)
print("Input tokens:", token_ids)

# Forward pass
with torch.no_grad():
    logits = model(token_ids)

print("Output logits shape:",
      logits.shape)
```

```python
print("Expected shape:
      (batch_size, seq_len, vocab_size)")
print(f"Actual: ({batch_size}, {seq_len},
        {GPT_CONFIG_124M['vocab_size']})")

# Verify parameter count
total_params = sum(p.numel()
                   for p in model.parameters())
print(f"Total parameters: {total_params:,}")
print(f"Expected: ~124M parameters")
```

Validation Checks:

- Input: (batch, sequence) of token IDs
- Output: (batch, sequence, vocab) logits
- Parameters: ~124 million
- Forward pass executes successfully

Note: Untrained model produces random outputs!

Text Generation: From Logits to Tokens

Generation Pipeline:

1. **Forward Pass:** Model produces logits for next token
2. **Probability Conversion:** Apply softmax to get probabilities
3. **Sampling:** Select next token from probability distribution
4. **Context Update:** Add selected token to sequence
5. **Repeat:** Continue until stopping condition

Mathematical Formulation:

$$P(\text{token}_t|\text{context}_{1:t-1}) = \text{softmax}(\text{logits}_t)$$

Sampling Strategies:

- **Greedy:** Always pick highest probability token
- **Random:** Sample from full probability distribution
- **Top-k:** Sample from k highest probability tokens
- **Temperature:** Control randomness with scaling

♦ Interactive Exercise: Text Generation

```python
def generate_text_simple(model, idx,
                         max_new_tokens,
                         context_size):
    """
    Generate text using the trained GPT model.

    Args:
        model: The GPT model
        idx: Starting token indices
             (batch_size, seq_len)
        max_new_tokens: Number of tokens to generate
        context_size: Maximum context length
    """
    model.eval()  # Set to evaluation mode

    for _ in range(max_new_tokens):
        # TODO: Crop context if it gets too long
        idx_cond = idx[:, -context_size:]
        # Keep last context_size tokens

        with torch.no_grad():
            # TODO: Get model predictions
            logits = model(idx_cond)

            # TODO: Get logits for the last token
            logits = logits[:, -1, :]
            # (batch_size, vocab_size)
```

```python
            # TODO: Apply softmax and sample
            probs = torch.softmax(logits, dim=-1)
            idx_next = torch.multinomial(
                probs, num_samples=1)
            # (batch_size, 1)

            # TODO: Append to sequence
            idx = torch.cat((idx, idx_next), dim=1)

    return idx

# Test the generation function
torch.manual_seed(42)
model = GPTModel(GPT_CONFIG_124M)
model.eval()

# Start with "Hello" token (example)
start_context = torch.tensor([[15496]])
# Example token for "Hello"
generated = generate_text_simple(
    model, start_context, max_new_tokens=10,
    context_size=GPT_CONFIG_124M["context_length"])

print(f"Generated token sequence:
      {generated[0].tolist()}")
print("Note: Untrained model will produce
      random/meaningless output!")
```

Architecture Summary: What We've Built

Complete GPT Architecture Components

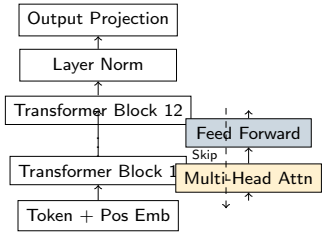
Core Components:

- ✓ **Causal Attention:** Masked self-attention
- ✓ **Multi-Head Attention:** Parallel processing
- ✓ **Layer Normalization:** Training stability
- ✓ **GELU Activation:** Smooth non-linearity
- ✓ **Feed-Forward Networks:** Feature transformation
- ✓ **Shortcut Connections:** Gradient flow

Integration Patterns:

- Transformer blocks with pre-normalization
- Residual connections around attention and FF
- Causal masking for autoregressive generation
- Parameter sharing and scaling strategies

Architecture Flow:



Current Capabilities:

- ✓ Forward pass through complete architecture
- ✓ Text generation (token-by-token)
- ✓ Proper gradient flow for training
- × No learned representations (untrained weights)

Achievement

We've built a **complete, functional GPT architecture** ready for training!

Current Model: Capabilities and Limitations

What We Have vs. What We Need

✓ **Current Capabilities:**

- Complete GPT-2 architecture (124M params)
- Proper causal attention masking
- Forward pass computation
- Text generation mechanism
- Gradient computation ready
- All components integrated correctly

♦ **Technical Verification:**

- Input/output shape consistency
- Causal masking working correctly
- Parameter count matches GPT-2 small
- No computational errors

What's Missing:

1. **Training Data:** Large corpus of text for learning
2. **Training Loop:** Optimization procedure
3. **Loss Function:** Next-token prediction loss
4. **Computation:** GPU training for days/weeks

× **Current Limitations:**

- **Random weights:** No learned patterns
- **No language understanding:** Gibberish output
- **No training:** Weights are random
- **No optimization:** No loss minimization

■ **Example Output:**

```
Input: "Hello, I am"
Untrained Output:
"xJtok4mK9nQ8vR"
(Complete nonsense!)
```

Next Step

Lecture 6: Pre-training pipeline to transform our architecture into a functioning language model!

Bridge to Lecture 6: From Architecture to Intelligence

The Journey Ahead

Today's Achievement:

Complete GPT Architecture
(Ready for Learning)

Lecture 6 Preview: Pre-training Large Language Models

- **Training Data:** Preparing massive text corpora
- **Loss Functions:** Cross-entropy for next-token prediction
- **Optimization:** Adam optimizer, learning rate scheduling
- **Training Loop:** Mini-batch processing and gradient updates
- **Evaluation:** Perplexity and generation quality metrics
- **Scaling:** Multi-GPU training and efficiency