Architectural Optimizations

Aaron Zhao, Imperial College London

Introduction

Learning Objectives

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

- Understand different strategies for optimizing neural network architectures for efficiency
- Explain how depthwise separable convolutions reduce computational complexity in MobileNet
- Describe how Longformer addresses the quadratic complexity of attention mechanisms
- Recognize how hybrid architectures like MobileViT combine strengths of different operators
- Understand the role of KV caching in accelerating autoregressive generation

Motivation

Efficiency is a key metric in evaluating model performance, especially for deployment on resource-constrained devices and real-time applications.

Three main approaches to architectural optimization:

- **Re-design the basic operands**: Use cheaper operators to approximate standard operators (e.g., depthwise separable convolutions)
- **Architecture-level re-engineering**: Combine different operators strategically (e.g., hybrid CNN-Transformer models)
- **System-level re-structuring**: Optimize computation patterns at inference time (e.g., KV caching)

Most of these modifications happen at the algorithmic level rather than hardware level!

Lecture Outline

We will explore four examples in detail:

- **MobileNet** – Depthwise Separable Convolutions for efficient CNNs
- **Longformer** – Local windowed attention for long sequences
- **MobileViT** – Hybrid CNN-Transformer architectures
- **LLaMA** – KV caching for efficient autoregressive generation

MobileNet

Background: Standard Convolution

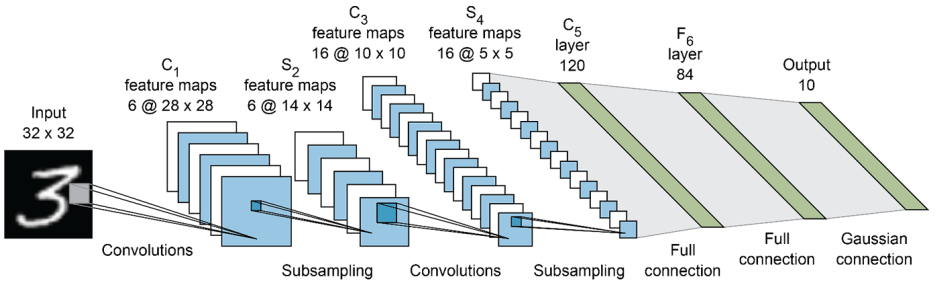
Recall from previous lectures that a standard convolution operation can be characterized by $(N, C_{in}, C_{out}, K, H, W)$:

- N: Batch size
- C_{in}: Input channels
- C_{out}: Output channels
- K: Kernel size
- H, W: Height and width of feature maps

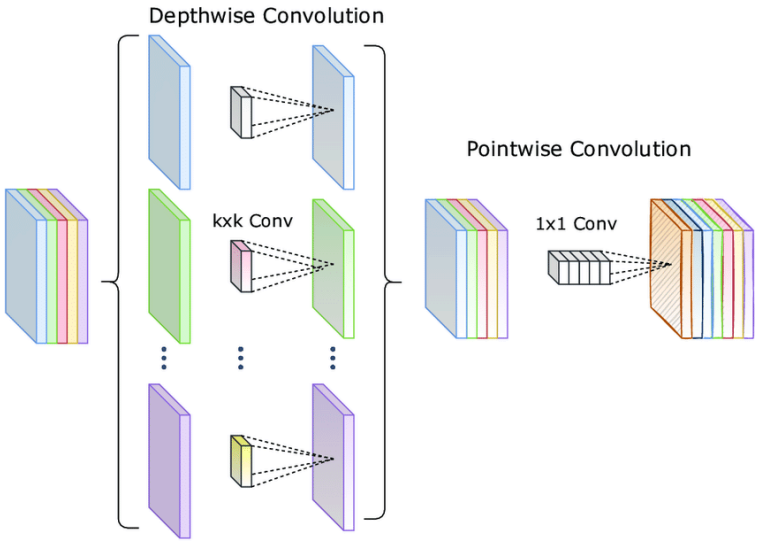
Computational cost:

- Parameters: $C_{in} \times C_{out} \times K \times K$
- FLOPs: $N \times C_{in} \times C_{out} \times K \times K \times H \times W$

Background: Standard Convolution (ii)



Depthwise Separable Convolution



Depthwise Separable Convolution (ii)

Core idea: Decompose standard convolution into two simpler operations.

Depthwise Separable Convolution = Depthwise Convolution + Pointwise Convolution

Depthwise Convolution:

- Apply a single filter per input channel (grouped convolution where groups = channels)
- Each $K \times K$ filter operates on only one channel
- Output has same number of channels as input

Pointwise Convolution:

- Use 1×1 convolutions to combine information across channels
- Creates linear combinations of the depthwise outputs
- Produces desired number of output channels

Complexity comparison:

Depthwise Separable Convolution (iii)

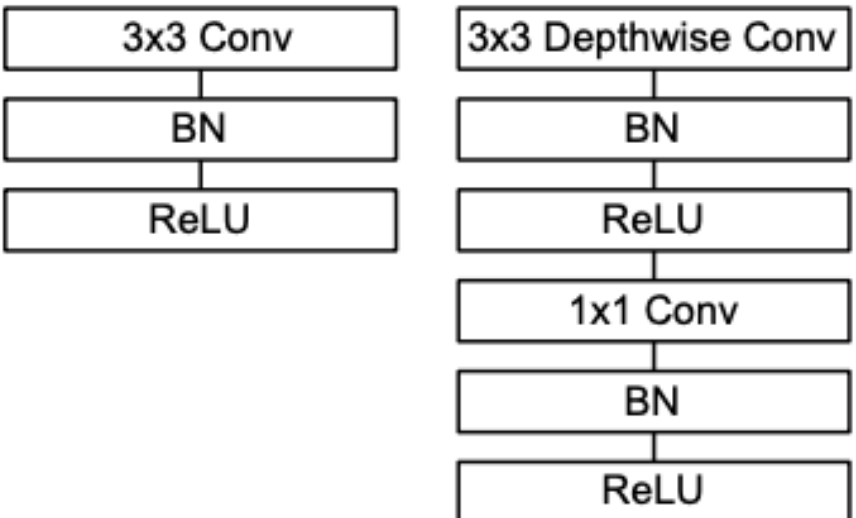
Standard Convolution Parameters: $C_{in} \times C_{out} \times K \times K$

Depthwise Separable Parameters: $C_{in} \times K \times K + C_{in} \times C_{out}$

Reduction ratio: $\frac{C_{in} \times K \times K + C_{in} \times C_{out}}{C_{in} \times C_{out} \times K \times K} = \frac{1}{C_{out}} + \frac{1}{K^2}$

For $C_{out} = 128$ and $K = 3$: approximately 8-9 $\times$ reduction!

MobileNet Block Design



MobileNet Block Design (ii)

The complete MobileNet block includes:

- Depthwise Convolution with BatchNorm and ReLU
- Pointwise Convolution with BatchNorm and ReLU

Key observations:

- BatchNorm (BN) normalizes activations and improves training stability
- ReLU activations are applied after each convolution
- This modular design can replace standard convolution blocks in any CNN

MobileNet Architecture

Table 1. MobileNet Body Architecture

Type / Stride	Filter Shape	Input Size
Conv / s2	$3 \times 3 \times 3 \times 32$	$224 \times 224 \times 3$
Conv dw / s1	$3 \times 3 \times 32$ dw	$112 \times 112 \times 32$
Conv / s1	$1 \times 1 \times 32 \times 64$	$112 \times 112 \times 32$
Conv dw / s2	$3 \times 3 \times 64$ dw	$112 \times 112 \times 64$
Conv / s1	$1 \times 1 \times 64 \times 128$	$56 \times 56 \times 64$
Conv dw / s1	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 128$	$56 \times 56 \times 128$
Conv dw / s2	$3 \times 3 \times 128$ dw	$56 \times 56 \times 128$
Conv / s1	$1 \times 1 \times 128 \times 256$	$28 \times 28 \times 128$
Conv dw / s1	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 256$	$28 \times 28 \times 256$
Conv dw / s2	$3 \times 3 \times 256$ dw	$28 \times 28 \times 256$
Conv / s1	$1 \times 1 \times 256 \times 512$	$14 \times 14 \times 256$
$5\times$ Conv dw / s1	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
$5\times$ Conv / s1	$1 \times 1 \times 512 \times 512$	$14 \times 14 \times 512$
Conv dw / s2	$3 \times 3 \times 512$ dw	$14 \times 14 \times 512$
Conv / s1	$1 \times 1 \times 512 \times 1024$	$7 \times 7 \times 512$
Conv dw / s2	$3 \times 3 \times 1024$ dw	$7 \times 7 \times 1024$
Conv / s1	$1 \times 1 \times 1024 \times 1024$	$7 \times 7 \times 1024$
Avg Pool / s1	Pool 7×7	$7 \times 7 \times 1024$
FC / s1	1024×1000	$1 \times 1 \times 1024$
Softmax / s1	Classifier	$1 \times 1 \times 1000$

Reading architecture tables:

- Each row represents a layer or block in the network
- “Conv dw” denotes depthwise convolution layers

MobileNet Architecture (ii)

- “s2” indicates stride 2 (spatial downsampling)
- Progressive channel expansion and spatial reduction
- Final layers: average pooling + fully connected + softmax

The architecture follows the standard CNN pattern: extract features at increasing abstraction levels while reducing spatial dimensions.

Longformer

The Context Length Problem

Transformers face a quadratic complexity bottleneck: full attention scales as $O(N^2)$ with sequence length N.

Attention mechanism recap:

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

Complexity analysis:

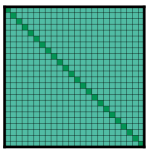
- $Q, K \in \mathbb{R}^{N \times d_k}$ (sequence length N, dimension d_k)
- Computing QK^T requires $O(N^2 \times d_k)$ operations
- Memory requirement: $O(N^2)$ to store attention matrix
- Problem: For long documents (e.g., $N = 4096$), this becomes prohibitively expensive

Motivation for Longformer:

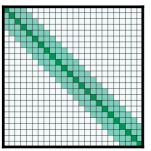
The Context Length Problem (ii)

- Many NLP tasks require long context (document classification, QA on long texts)
- Standard Transformers limited to 512-1024 tokens due to memory constraints
- Need efficient attention mechanism that scales linearly with sequence length

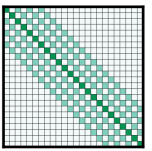
Longformer Attention Patterns



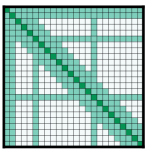
(a) Full n^2 attention



(b) Sliding window attention



(c) Dilated sliding window



(d) Global+sliding window

Longformer combines three attention patterns to achieve $O(N)$ complexity:

1. Sliding Window Attention:

- Each token attends only to w neighboring tokens ($w/2$ on each side)
- Complexity: $O(N \times w)$ where w is fixed window size
- Captures local context efficiently

2. Dilated Attention:

- Skip tokens with gaps (dilation) to increase receptive field

Longformer Attention Patterns (ii)

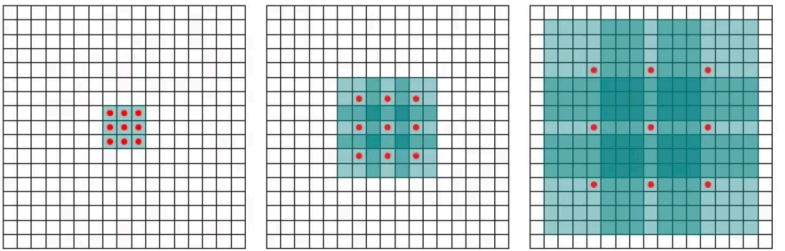
- Similar to dilated convolutions - larger effective receptive field without extra computation
- Helps capture longer-range dependencies

3. Global Attention:

- Select specific tokens (e.g., [CLS], task-specific tokens) to attend to all positions
- These tokens can aggregate information globally
- Task-dependent: classification uses [CLS], QA uses question tokens

Implementation note: Requires custom CUDA kernels to realize speedup in practice (standard attention implementations don't exploit sparsity efficiently)

Understanding Dilation



l=1 (left), l=2 (Middle), l=4 (Right)

Dilation is a technique originally developed for convolutions:

Concept:

- Skip intermediate positions with regular gaps
- Sample the input at intervals rather than consecutively
- Dilation rate r: sample every r-th position

Understanding Dilation (ii)

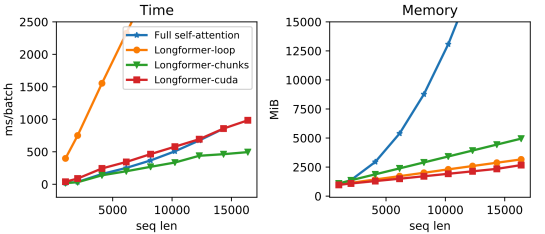
Visual explanation:

- Images show dilation factors $r = 1, 2, 4$
- $r = 1$: standard operation (no skipping)
- $r = 2$: skip every other position
- $r = 4$: sample every 4th position

Benefits:

- Larger receptive field without additional parameters
- Captures multi-scale information efficiently
- In attention: fewer computations while maintaining long-range connections

Longformer Performance



Empirical results confirm theoretical analysis:

- Memory usage scales linearly with sequence length: $O(N)$ vs. $O(N^2)$ for standard attention
- Enables processing of sequences up to 4096+ tokens on standard GPUs
- Time complexity also linear, though wall-clock speedup depends on implementation

Implementation considerations:

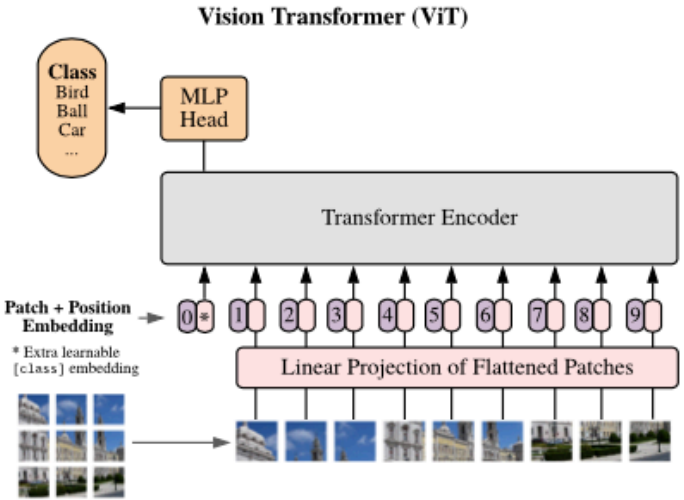
Longformer Performance (ii)

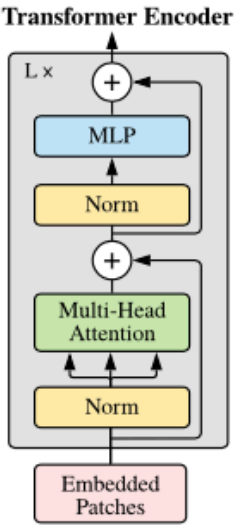
- CUDA kernels implemented using TVM (tensor virtual machine)
- Native hand-optimized CUDA could be even faster
- Sparsity pattern must be exploited at low level to see real speedups

Key takeaway: Operator redesign can fundamentally change complexity class from quadratic to linear!

MobileViT

Background: Vision Transformers vs CNNs





Background: Vision Transformers vs CNNs (ii)

Vision Transformers (ViTs) achieve excellent accuracy but are computationally expensive:

Parameter comparison:

- ViT-B/16: 86 million parameters
- MobileNetv3: 7.5 million parameters

Key observation: High-parameter performance doesn't guarantee low-parameter efficiency

- At 5-6M parameter budget: DeiT is 3% **less** accurate than MobileNetv3
- ViTs struggle in the mobile/edge deployment regime

Why CNNs are more efficient:

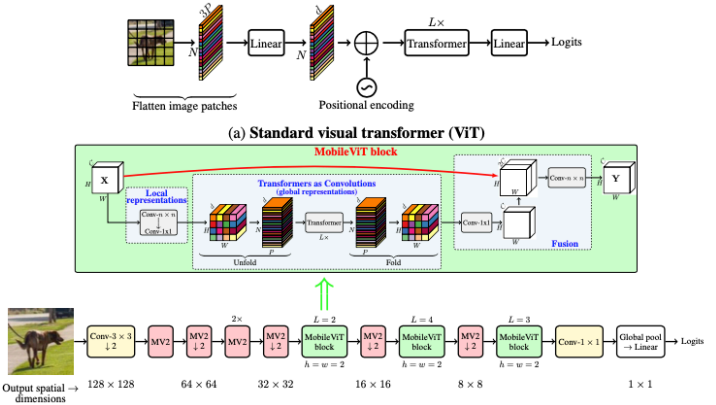
- Inductive biases: translation equivariance, locality
- Efficient operators: depthwise separable convolutions
- Mature optimization: years of architecture engineering

MobileViT's insight: Combine the strengths of both paradigms

Background: Vision Transformers vs CNNs (iii)

- Use CNNs for efficient local feature extraction
- Use Transformers for global context modeling

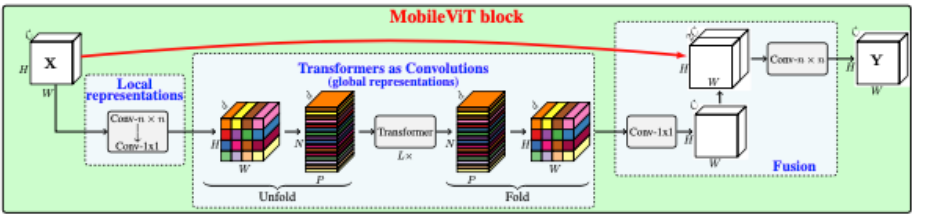
MobileViT Architecture Overview



Design principle: Interleave convolutional layers with ViT blocks

- Early layers: Standard and depthwise convolutions for local features
- Middle/late layers: MobileViT blocks combining CNN and Transformer
- Progressive fusion of local and global representations

MobileViT Block Details



Input: Feature map $\boldsymbol{X} \in \mathbb{R}^{H \times W \times C}$

Processing pipeline:

1. **Local representation** (Convolution):
   - Project to lower dimension: $\boldsymbol{X} \to \boldsymbol{X}_P \in \mathbb{R}^{H \times W \times d}$
   - Reduces channels from C to d (typically $d < C$)
2. **Unfold to patches**:
   - Reshape to sequence: $\boldsymbol{X}_P \to \boldsymbol{X}_U \in \mathbb{R}^{N \times P \times d}$
   - N patches, each of size P pixels, d channels

MobileViT Block Details (ii)

- Each patch becomes a "token" for Transformer

3. **Global modeling** (Transformer):
   - Apply multi-headed self-attention and FFN
   - Captures long-range dependencies between patches
   - Output: $\boldsymbol{X}_T \in \mathbb{R}^{N \times P \times d}$
4. **Fold back to spatial**:
   - Reshape: $\boldsymbol{X}_T \to \boldsymbol{X}_F \in \mathbb{R}^{H \times W \times d}$
5. **Fusion** (Convolution):
   - Project back: $\boldsymbol{X}_F \to \boldsymbol{X}_C \in \mathbb{R}^{H \times W \times C}$

MobileViT Fusion and Skip Connection

Residual connection: Concatenate input with processed output

$$[\boldsymbol{X}, \boldsymbol{X}_C] \to \boldsymbol{X}_{\text{out}}$$

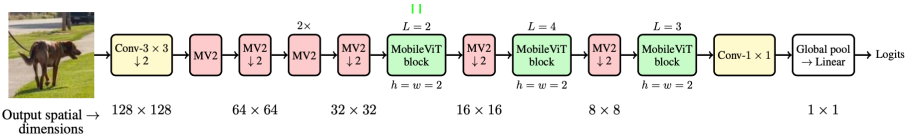
where:

- $\boldsymbol{X} \in \mathbb{R}^{H \times W \times C}$: Original input
- $\boldsymbol{X}_C \in \mathbb{R}^{H \times W \times C}$: Processed through MobileViT block
- $\boldsymbol{X}_{\text{out}} \in \mathbb{R}^{H \times W \times C}$: Final output

Key design choices:

- Skip connection preserves gradient flow (like ResNet)
- Concatenation allows network to select between local (CNN) and global (Transformer) features
- Final projection combines both pathways

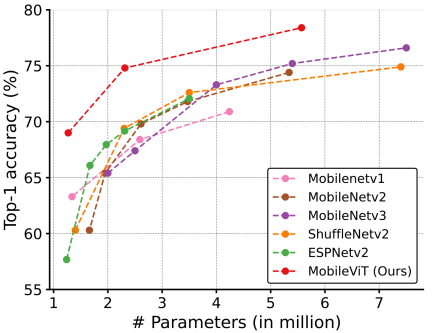
MobileViT Fusion and Skip Connection (ii)



Full architecture:

- Fusion of CNNs (local, parameter-efficient) and ViTs (global, expressive)
- Best of both worlds: efficiency and performance

MobileViT: Pareto Optimality



Evaluating architecture choices: Use the Pareto frontier

- X-axis: Computational cost (FLOPs or parameters)
- Y-axis: Performance (accuracy)
- Pareto optimal: No other model is better in both dimensions

Key insights:

MobileViT: Pareto Optimality (ii)

- MobileViT achieves better accuracy-efficiency trade-off
- Systematically combining operators outperforms pure CNN or pure ViT
- Architecture-level re-engineering is a principled design approach
- Related to Neural Architecture Search (covered in later lectures)

KV Caching

Autoregressive Generation in LLMs

Modern LLMs (like LLaMA, GPT) use decoder-only architectures for text generation. **Generation process is iterative**: Generate one token at a time, append to input, repeat.

```
i = 0
while out_token != token_eos:
  logits, _ = model(in_tokens)  # Forward pass on entire sequence
  out_token = torch.argmax(
    logits[-1, :], dim=0, keepdim=True)  # Take last token
prediction
  in_tokens = torch.cat((in_tokens, out_token), 0)  # Append to
input
  text = tokenizer.decode(in_tokens)
  print(f'step {i} input: {text}', flush=True)
  i += 1
```

Problem: Each iteration reprocesses the entire growing sequence!

Generation Example

Input prompt: "Lionel Messi is a"

Step-by-step generation:

```
Step 0: Lionel Messi is a player
Step 1: Lionel Messi is a player who
Step 2: Lionel Messi is a player who has
...
```

Final output: "Lionel Messi is a player who has been a key part of the team."

Computational cost analysis:

- Step 0: Process 5 tokens
- Step 1: Process 6 tokens (recompute all 5 + new 1)
- Step 2: Process 7 tokens (recompute all 6 + new 1)
- ...
- Step n: Process $(5 + n)$ tokens

Total operations: $O(n^2)$ where n is output length - very expensive!

The Key Insight: Causal Attention

Observation: When generating token i, we've already computed intermediate values for all previous tokens $(0, 1, ..., i-1)$.

Causal Language Modeling (CLM):

- Decoder uses causal attention: token i can only attend to tokens $\leq i$
- Implemented via causal mask on attention matrix QK^T
- Upper triangular part of attention matrix is masked out

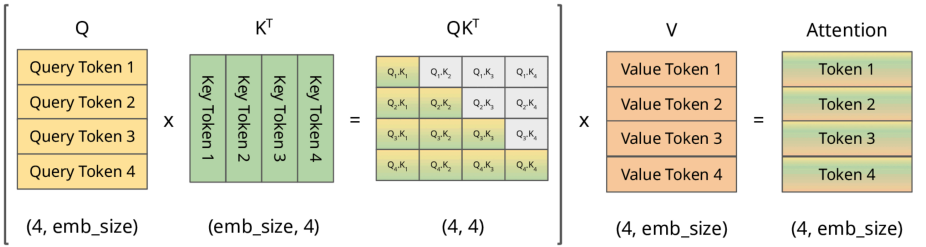
Redundant computation:

- At step i, we recompute attention for tokens 0 to $i-1$
- But these computations are identical to previous steps!
- We only need the attention for the new token i

Key-Value Caching solution:

- Cache the Key and Value matrices from previous steps
- Only compute Query for the new token
- Reuse cached K and V for attention with new Q

Without KV Cache: Naive Approach



Step 1: Compute $Q_1 K_1^T$

- Process first token

Step 2: Compute $Q_1 K_1^T, Q_2 K_1^T, Q_2 K_2^T$

- Recompute $Q_1 K_1^T$ (redundant!)
- Compute new interactions with token 2

Step 3: Compute $Q_1 K_1^T, Q_2 K_1^T, Q_2 K_2^T, Q_3 K_1^T, Q_3 K_2^T, Q_3 K_3^T$

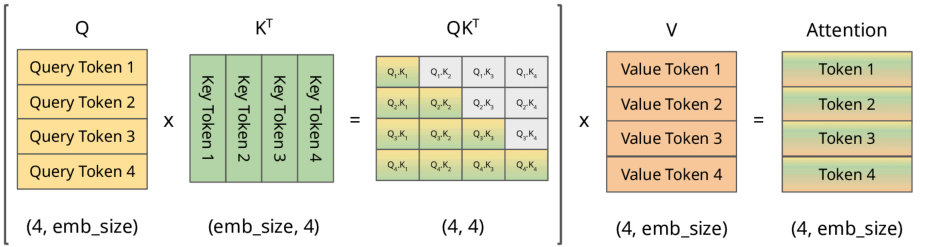
- Recompute everything from steps 1-2 (very redundant!)

Without KV Cache: Naive Approach (ii)

- Only $Q_3K_1^T, Q_3K_2^T, Q_3K_3^T$ are new

Total complexity: $O(n^2)$ for generating n tokens

With KV Cache: Efficient Approach



Step 1:

- Compute Q_1, K_1, V_1 and attention $Q_1 K_1^T$
- **Cache** K_1, V_1

Step 2:

- **Retrieve** cached K_1, V_1
- Compute only Q_2, K_2, V_2
- Compute new attention: $Q_2[K_1, K_2]^T$ (using cached K_1)

With KV Cache: Efficient Approach (ii)

- **Update cache**: Add K_2, V_2 to cache

Step 3:

- **Retrieve** cached K_1, K_2, V_1, V_2
- Compute only Q_3, K_3, V_3
- Compute new attention: $Q_3[K_1, K_2, K_3]^T$ (using cached keys)
- **Update cache**: Add K_3, V_3 to cache

Total complexity: $O(n)$ for generating n tokens - much better!

KV Cache Trade-offs

Benefits:

- Reduces time complexity from $O(n^2)$ to $O(n)$ for generation
- Essential for real-time inference in production LLMs
- Enables longer generation sequences

Costs:

- Memory overhead: Must store K, V for all previous tokens
- For L layers, H heads, d dimensions, n tokens: $O(2 \times L \times n \times d)$ memory
- Batch size limited by GPU memory (larger cache = fewer parallel requests)

Implementation:

- Used in all modern LLMs: GPT, LLaMA, Claude, etc.
- Trade memory for computation - usually worthwhile
- Critical for acceptable latency in chat applications

Summary

Three Approaches to Architectural Optimization

1. Re-design the basic operands

- Replace expensive operations with efficient approximations
- **MobileNet**: Depthwise separable convolutions (8-9 $\times$ parameter reduction)
- **Longformer**: Sparse attention patterns ($O(N^2) \rightarrow O(N)$ complexity)

2. Architecture-level re-engineering

- Strategically combine different operators
- **MobileViT**: Hybrid CNN-Transformer architecture
- Use CNNs for local features, Transformers for global context
- Pareto-optimal accuracy-efficiency trade-offs

3. System-level re-structuring

- Optimize computation patterns at inference time
- **KV Caching**: Eliminate redundant attention computations
- Essential for practical LLM deployment

Key principle: Optimization can happen at multiple levels of the stack!