MIDDLE EAST TECHNICAL UNIVERSITY

SEMESTER I EXAMINATION 2024-2025

CENG 403 – Deep Learning - CNN Fundamentals & Convolution Types (University Sources - ANSWERED)

January 2025 TIME ALLOWED: 3 HOURS

INSTRUCTIONS

1. This is the ANSWERED version with detailed mathematical explanations.
2. Each answer includes step-by-step calculations and intuitive explanations.
3. Focus on understanding the mathematical foundations and practical implications.
4. All calculations are shown in detail to prevent confusion.

Question 1. CNN Motivation and MLP Computational Analysis (25 marks)
Based on Stanford CS231n and UC Berkeley CS280 deep learning course materials.

(a) Calculate the parameter explosion in fully connected networks. For a 200×200 RGB image classification task: (10 marks)

- Calculate parameters for single hidden layer with 1000 neurons
- Compare with equivalent CNN layer (64 filters, 3×3 kernels, same output size)
- Determine parameter reduction ratio and explain computational implications

Answer: Part 1: Fully Connected Layer Parameters

Let me calculate step by step:

Input Dimensions:

- Image size: 200×200 pixels
- Channels: 3 (RGB)
- Total input values: $200 \times 200 \times 3 = 120{,}000$

Fully Connected Layer:

- Input neurons: 120,000
- Hidden neurons: 1,000
- Weights: $120{,}000 \times 1{,}000 = 120{,}000{,}000$ (120 million)
- Biases: 1,000
- Total parameters: 120,001,000 120 million

Part 2: CNN Layer Parameters

For a CNN layer with 64 filters of size 3×3:

Convolution Layer:

- Filter size: 3×3
- Input channels: 3 (RGB)

- Output channels: 64
- Weights per filter: $3 \times 3 \times 3 = 27$
- Total weights: $27 \times 64 = 1{,}728$
- Biases: 64
- Total parameters: 1,792

Output Size Calculation:

- With padding=1, stride=1: Output = 200×200×64
- Same spatial dimensions, 64 channels

Part 3: Parameter Reduction Ratio

Comparison:

- MLP parameters: 120,001,000
- CNN parameters: 1,792
- Reduction ratio: 120,001,000 / 1,792 = 66,965
- That's approximately 67,000× fewer parameters!

Computational Implications:

1. **Memory Requirements:**
   - MLP: 120M × 4 bytes = 480 MB (just for weights)
   - CNN: 1,792 × 4 bytes = 7 KB
   - During training, need 3-4× more for gradients and optimizer states
   - MLP would need 2GB, CNN needs 28KB
2. **Training Speed:**
   - MLP forward pass: 120M multiply-adds
   - CNN forward pass: 200×200×64×27 = 69M multiply-adds
   - Similar computation but CNN is parallelizable on GPU
   - CNN trains 10-100× faster in practice
3. **Generalization:**

- MLP: 120M parameters need 1.2B training samples
- CNN: 1,792 parameters need 18K training samples
- CNN can generalize from much smaller datasets

4. **Gradient Flow:**

- MLP: Gradients through 120M parameters (vanishing gradient risk)
- CNN: Shared weights improve gradient flow
- CNN trains more stably

(b) Analyze the three fundamental limitations of MLPs for computer vision tasks: (10 marks)

- Translation variance: Mathematical demonstration of sensitivity to spatial shifts
- Full connectivity overfitting: Relationship between parameter count and generalization
- Curse of dimensionality: Data requirements scaling with parameter count

Answer: 1. Translation Variance - Mathematical Demonstration

Let me show mathematically why MLPs fail with shifted inputs:

Setup:

- Input image x as vector: $[x_1, x_2, ..., x_n]$
- MLP neuron j: $h_j = \sigma(\sum_i w_{ji} x_i + b_j)$
- Weight w_{ji} connects input position i to neuron j

Original Image: Say we have a vertical edge at pixels 100-102:

- $x_{99} = 0$, $x_{100} = 0$, $x_{101} = 128$, $x_{102} = 255$
- Neuron trained to detect this: $w_{j,99} = 0$, $w_{j,100} = -1$, $w_{j,101} = 0$, $w_{j,102} = 1$
- Activation: $h_j = \sigma(0 - 0 + 0 + 255) = \sigma(255)$ Strong response

Shifted Image (1 pixel right):

- Now: $x_{99} = 0$, $x_{100} = 0$, $x_{101} = 0$, $x_{102} = 128$
- Same neuron computes: $h_j = \sigma(0 + 0 + 0 + 128) = \sigma(128)$
- Different response! The neuron partially fails
- For 2-pixel shift: $h_j = \sigma(0) = 0.5$ - Complete failure!

Mathematical Proof: For shift by k pixels, if x' is shifted version of x:

$$h_j(x') = \sigma(\sum_i w_{ji} x'_i) = \sigma(\sum_i w_{ji} x_{i-k}) \neq h_j(x)$$

Unless weights have special structure (which they don't in MLPs).

2. Full Connectivity Overfitting

Statistical Learning Theory: The generalization error bound (simplified VC-dimension):

$$\text{Error} \leq \text{Training Error} + O\left(\sqrt{\frac{d \log(n/d) + \log(1/\delta)}{n}}\right)$$

Where:

- d = number of parameters (VC dimension proxy)
- n = number of training samples
- δ = confidence parameter

For our MLP with 120M parameters:

- $d = 120,000,000$
- Need $n \gg d$ for small generalization gap
- Typically need $n \approx 10d = 1.2$ billion samples!
- ImageNet only has 1.2M images (1000× too few)

Overfitting Demonstration:

- With 120M parameters, can memorize 15M images perfectly (8 bytes/image)
- Training accuracy: 100%

- Test accuracy: random guessing
- Each parameter can "memorize" specific training examples

3. Curse of Dimensionality

The Exponential Growth Problem:

As input dimension grows, data requirements grow exponentially:

Covering the Input Space:

- 1D input, 10 samples: Good coverage of [0,1]
- 2D input, 10 samples: Sparse coverage of $[0,1]^2$
- 120,000D input, 10 samples: Essentially no coverage!

Nearest Neighbor Intuition: In high dimensions, all points are far apart:

- Average distance between random points in d dimensions: $\propto \sqrt{d}$
- For $d = 120,000$: average distance ≈ 346
- All training examples are equally "far" from test examples
- No local generalization possible

Sample Complexity: To maintain same coverage quality:

- 1D: Need n samples
- 2D: Need n^2 samples
- dD: Need n^d samples
- For $d = 120,000$: Impossible!

CNN Solution: CNNs escape this by:

- Imposing structure (local connectivity)
- Sharing parameters (translation equivariance)
- Hierarchical features (compositional structure)
- Effective dimension much lower than input dimension

(c) Compare memory requirements during training for MLP vs CNN processing 224×224×3 images: (5 marks)

- Forward pass activation storage
- Backward pass gradient storage
- Total memory footprint analysis

Answer: Memory Requirements Analysis

Input: 224×224×3 = 150,528 values

Let's compare first layer: MLP (1000 hidden) vs CNN (64 filters, 3×3)

Forward Pass - Activation Storage:

MLP:

- Input: 150,528 × 4 bytes = 602 KB
- Hidden activations: 1,000 × 4 bytes = 4 KB
- Pre-activation values: 1,000 × 4 bytes = 4 KB
- Total: 610 KB

CNN:

- Input: 224×224×3 × 4 bytes = 602 KB
- Output feature maps: 224×224×64 × 4 bytes = 12.8 MB
- Total: 13.4 MB

Wait, CNN uses MORE memory for activations! But this is because CNN preserves spatial structure.

Backward Pass - Gradient Storage:

MLP:

- Weight gradients: 150,528 × 1,000 × 4 bytes = 602 MB
- Bias gradients: 1,000 × 4 bytes = 4 KB
- Input gradients: 150,528 × 4 bytes = 602 KB
- Total: 603 MB

CNN:

- Weight gradients: 3×3×3×64 × 4 bytes = 7 KB
- Bias gradients: 64 × 4 bytes = 256 bytes
- Input gradients: 224×224×3 × 4 bytes = 602 KB
- Total: 609 KB

Total Memory Footprint (Single Layer):

MLP Total:

- Parameters: 602 MB
- Activations: 610 KB
- Gradients: 603 MB
- Optimizer states (Adam): 2 × 602 MB = 1.2 GB
- Total: 2.4 GB

CNN Total:

- Parameters: 7 KB
- Activations: 13.4 MB
- Gradients: 609 KB
- Optimizer states: 14 KB
- Total: 14 MB

Key Insights:

- MLP: 170× more memory than CNN
- MLP memory dominated by parameters/gradients
- CNN memory dominated by activations
- For deep networks, CNN advantage grows (MLP becomes impossible)
- Batch processing: CNN can use 170× larger batches!

Question 2. Mathematical Foundation of Convolution Operations (28 marks)
Based on MIT 6.036 and CMU 10-301 mathematical treatments of convolution.

(a) Derive the convolution output size formula and apply to practical examples: (12 marks)

- Prove: Output size $= \frac{W-F+2P}{S} + 1$
- Apply to AlexNet Conv1: Input 227×227, Filter 11×11, Stride 4, Padding 0
- Calculate feature map sizes through first 3 layers of AlexNet
- Verify mathematical consistency with integer constraints

Answer: Deriving the Output Size Formula from First Principles

I'll build this formula step by step so every part makes sense.

Step 1: Basic Case (No Padding, Stride=1)

Consider 1D for simplicity:

- Input size: W (positions 0 to $W-1$)
- Filter size: F (needs F consecutive positions)
- Valid positions for filter center: where filter fits entirely

First valid position: Filter starts at 0, ends at $F-1$ Last valid position: Filter starts at $W-F$, ends at $W-1$

Number of positions: $(W-F)-0+1 = W-F+1$

Step 2: Adding Padding

Padding P adds P positions on each side:

- New effective input size: $W+2P$
- Substitute into formula: $(W+2P)-F+1$
- Simplified: $W-F+2P+1$

Step 3: Adding Stride

Stride S means we skip positions:

- Positions used: 0, S, 2S, 3S, ..., last feasible
- Last feasible: Largest $kS \leq W - F + 2P$
- Number of positions: $k + 1$ where $k = \lfloor \frac{W-F+2P}{S} \rfloor$

Final Formula:

$$\boxed{\text{Output Size} = \left\lfloor \frac{W - F + 2P}{S} \right\rfloor + 1}$$

AlexNet Conv1 Calculation:

Given:

- $W = 227$ (width and height)
- $F = 11$ (filter size)
- $S = 4$ (stride)
- $P = 0$ (no padding)

Calculation:

$$\begin{aligned} \text{Output} &= \left\lfloor \frac{227 - 11 + 2(0)}{4} \right\rfloor + 1 && (1) \\ &= \left\lfloor \frac{216}{4} \right\rfloor + 1 && (2) \\ &= \lfloor 54 \rfloor + 1 && (3) \\ &= 55 && (4) \end{aligned}$$

Output: $55 \times 55 \times 96$ (96 filters)

AlexNet First 3 Layers:

Layer 1: Already calculated

- Input: $227 \times 227 \times 3$
- Output: $55 \times 55 \times 96$

Layer 2: Pooling

- Input: $55 \times 55 \times 96$
- Pool size: 3, Stride: 2
- Output: $\lfloor \frac{55-3}{2} \rfloor + 1 = 27$
- Output: $27 \times 27 \times 96$

Layer 3: Conv2

- Input: $27 \times 27 \times 96$
- Filter: 5×5, Stride: 1, Padding: 2
- Output: $\lfloor \frac{27-5+4}{1} \rfloor + 1 = 27$
- Output: $27 \times 27 \times 256$

Integer Constraint Verification:

All calculations yielded integers - this is crucial!

What if we used 225×225 input?

$$\text{Conv1 Output} = \left\lfloor \frac{225 - 11 + 0}{4} \right\rfloor + 1 \tag{5}$$

$$= \left\lfloor \frac{214}{4} \right\rfloor + 1 \tag{6}$$

$$= \lfloor 53.5 \rfloor + 1 = 54 \tag{7}$$

The 0.5 is lost! This creates alignment issues in later layers. AlexNet's 227×227 was carefully chosen to ensure integer sizes throughout.

(b) Analyze computational complexity of convolution operations: (10 marks)

- Derive FLOPs formula: $H' \times W' \times C_{out} \times F \times F \times C_{in}$
- Compare with equivalent fully connected layer complexity
- Calculate speedup ratio for typical CNN layer dimensions

Answer: Deriving the FLOPs Formula for Convolution

FLOPs = Floating Point Operations (multiply-adds)

Breaking Down One Output Value: To compute one value in the output feature map:

- Apply one $F \times F$ filter
- Across all C_{in} input channels
- Operations: $F \times F \times C_{in}$ multiplications
- Plus: $(F \times F \times C_{in} - 1)$ additions
- Plus: 1 bias addition
- Total: $2 \times F \times F \times C_{in}$ FLOPs (counting multiply-add as 2)

For Entire Output:

- Output size: $H' \times W' \times C_{out}$
- Each output value: $2 \times F \times F \times C_{in}$ FLOPs
- Total FLOPs: $2 \times H' \times W' \times C_{out} \times F \times F \times C_{in}$

Often simplified (counting multiply-add as 1 FLOP):

$$\boxed{\text{FLOPs} = H' \times W' \times C_{out} \times F \times F \times C_{in}}$$

Example Calculation:

Conv layer: Input 56×56×64 → Output 56×56×128, Filter 3×3

$$\begin{aligned} \text{FLOPs} &= 56 \times 56 \times 128 \times 3 \times 3 \times 64 && (8)\\ &= 3,136 \times 128 \times 9 \times 64 && (9)\\ &= 231,211,008 \text{ FLOPs} && (10)\\ &\approx 231 \text{ MFLOPs} && (11) \end{aligned}$$

Comparison with Fully Connected Layer:

Same input/output dimensions as FC layer:

- Input: 56×56×64 = 200,704 neurons
- Output: 56×56×128 = 401,408 neurons
- FC FLOPs: 200,704 × 401,408 = 80,530,866,432

- 80.5 GFLOPs

Speedup Ratio:

$$\text{Speedup} = \frac{\text{FC FLOPs}}{\text{Conv FLOPs}} \tag{12}$$

$$= \frac{80,530,866,432}{231,211,008} \tag{13}$$

$$= 348.4\times \tag{14}$$

The convolution is 348× more efficient!

Why Convolution is Faster:

1. Parameter Sharing:

- Conv: Same weights used at every position
- FC: Different weights for every connection
- Conv exploits spatial structure

2. Local Connectivity:

- Conv: Each output connects to $F \times F \times C_{in}$ inputs
- FC: Each output connects to ALL inputs
- Massive reduction in connections

3. Practical Benefits:

- Better cache utilization (local memory access)
- Highly parallel (each output position independent)
- Optimized implementations (cuDNN, etc.)
- Can use specialized hardware (tensor cores)

(c) Implement convolution using matrix multiplication (im2col): (6 marks)

- Explain im2col transformation for GPU optimization
- Show how convolution becomes GEMM operation
- Analyze memory vs computation trade-offs

Answer: The im2col (Image to Column) Transformation

im2col converts convolution into matrix multiplication - GPUs love matrix multiplication!

The Problem: Convolution involves sliding windows - irregular memory access patterns that GPUs handle poorly.

The Solution: Reshape the data so convolution becomes a single matrix multiplication.

How im2col Works:

Example: 4×4 input, 3×3 filter, stride=1

Step 1: Extract All Patches Original input:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

Extract all 3×3 patches:

- Patch (0,0): [1,2,3,5,6,7,9,10,11]
- Patch (0,1): [2,3,4,6,7,8,10,11,12]
- Patch (1,0): [5,6,7,9,10,11,13,14,15]
- Patch (1,1): [6,7,8,10,11,12,14,15,16]

Step 2: Create Matrix Stack patches as columns:

$$X_{col} = \begin{bmatrix} 1 & 2 & 5 & 6 \\ 2 & 3 & 6 & 7 \\ 3 & 4 & 7 & 8 \\ 5 & 6 & 9 & 10 \\ 6 & 7 & 10 & 11 \\ 7 & 8 & 11 & 12 \\ 9 & 10 & 13 & 14 \\ 10 & 11 & 14 & 15 \\ 11 & 12 & 15 & 16 \end{bmatrix}_{9\times 4}$$

Step 3: Convolution as Matrix Multiply Filter as vector: $W = [w_1, w_2, ..., w_9]^T$

Convolution result: $Y = W^T \times X_{col}$

This computes all output positions in one matrix operation!

GEMM (General Matrix Multiply):

For multiple filters and channels:

- Input: $H \times W \times C_{in}$
- Filters: C_{out} filters of size $F \times F \times C_{in}$
- im2col matrix: $(F^2 \cdot C_{in}) \times (H' \cdot W')$
- Filter matrix: $C_{out} \times (F^2 \cdot C_{in})$
- Output: $C_{out} \times (H' \cdot W')$
- Reshape to: $H' \times W' \times C_{out}$

Now it's a standard GEMM operation - highly optimized on all hardware!

Memory vs Computation Trade-offs:

Memory Overhead:

- Original input: $H \times W \times C_{in}$
- im2col matrix: $F^2 \times C_{in} \times H' \times W'$
- Expansion factor: F^2 (9× for 3×3 filters!)
- Can be gigabytes for large feature maps

Benefits:

- Single optimized GEMM call
- 10-50× speedup on GPUs
- Better cache utilization
- Enables batching efficiently

Modern Optimizations:

- Implicit im2col: Generate patches on-the-fly
- Winograd: Reduce multiplications for small filters
- Direct convolution: Optimized kernels for common sizes

- Memory-computation trade-off depends on hardware

The key insight: Transform the problem to match what hardware does best!

Question 3. Receptive Field Theory and Parameter Sharing Analysis (22 marks)

Based on Stanford CS231n and UC Berkeley theoretical frameworks.

(a) Calculate effective receptive fields in deep CNN architectures: (10 marks)

- Derive recursive formula: $RF_l = RF_{l-1} + (k_l - 1) \times \prod_{i=1}^{l-1} s_i$
- Apply to architecture: Conv(3×3,s=1) → Conv(3×3,s=1) → Conv(3×3,s=2) → Conv(3×3,s=1)
- Calculate receptive field at each layer
- Explain relationship between depth and spatial coverage

Answer: Deriving the Receptive Field Formula

The receptive field (RF) is the region in the input that affects a particular output unit.

Key Insights:

- Each layer expands the receptive field
- Stride affects how much the RF grows
- We need to track cumulative effect through layers

Base Case:

- First layer: $RF_1 = k_1$ (kernel size)
- One pixel sees $k_1 \times k_1$ input region

Recursive Case: For layer l with kernel size k_l:

- Each unit in layer l sees k_l units from layer $l-1$
- Each unit in layer $l-1$ already sees RF_{l-1} pixels
- But they overlap! The expansion is $(k_l - 1)$ units
- Must account for all previous strides: $\prod_{i=1}^{l-1} s_i$

Formula:

$$\boxed{RF_l = RF_{l-1} + (k_l - 1) \times \prod_{i=1}^{l-1} s_i}$$

Applying to Given Architecture:

Architecture: Conv(3×3,s=1) → Conv(3×3,s=1) → Conv(3×3,s=2) → Conv(3×3,s=1)

Let me calculate layer by layer:

Layer 1: Conv(3×3, s=1)

- $RF_1 = k_1 = 3$
- Cumulative stride: $S_1 = 1$

Layer 2: Conv(3×3, s=1)

- $RF_2 = RF_1 + (k_2 - 1) \times S_1$
- $RF_2 = 3 + (3 - 1) \times 1 = 3 + 2 = 5$
- Cumulative stride: $S_2 = S_1 \times s_2 = 1 \times 1 = 1$

Layer 3: Conv(3×3, s=2)

- $RF_3 = RF_2 + (k_3 - 1) \times S_2$
- $RF_3 = 5 + (3 - 1) \times 1 = 5 + 2 = 7$
- Cumulative stride: $S_3 = S_2 \times s_3 = 1 \times 2 = 2$

Layer 4: Conv(3×3, s=1)

- $RF_4 = RF_3 + (k_4 - 1) \times S_3$
- $RF_4 = 7 + (3 - 1) \times 2 = 7 + 4 = 11$
- Cumulative stride: $S_4 = S_3 \times s_4 = 2 \times 1 = 2$

Summary:

- Layer 1: RF = 3×3
- Layer 2: RF = 5×5
- Layer 3: RF = 7×7

- Layer 4: RF = 11×11

Depth-Coverage Relationship:

Linear Growth (when stride=1):

- Each layer adds $k - 1$ to RF
- For 3×3 kernels: adds 2 per layer
- n layers: $RF = 1 + 2n$
- Very deep networks see large regions

Effect of Stride:

- Stride amplifies RF growth in later layers
- Layer 3's stride=2 caused Layer 4 to grow by 4 instead of 2
- Strategic stride placement affects coverage

Practical Implications:

- Early layers: Small RF, detect local features (edges)
- Middle layers: Medium RF, detect parts (eyes, wheels)
- Deep layers: Large RF, detect whole objects
- Need sufficient depth for object recognition
- Too shallow: can't see enough context

(b) Analyze parameter sharing efficiency: (8 marks)

- Compare: Single 7×7 conv ($49C^2$ parameters) vs Three 3×3 convs ($27C^2$ parameters)
- Calculate parameter reduction percentage: $\frac{49C-27C}{49C} = 44.9\%$
- Explain why stacked small filters outperform large filters

Answer: Parameter Comparison:

Let's compare two architectures with same receptive field (7×7):

Option A: Single 7×7 Convolution

- Input channels: C

- Output channels: C
- Parameters: $7 \times 7 \times C \times C = 49C^2$
- Receptive field: 7×7

Option B: Three Stacked 3×3 Convolutions

- Layer 1: $3 \times 3 \times C \times C = 9C^2$ parameters
- Layer 2: $3 \times 3 \times C \times C = 9C^2$ parameters
- Layer 3: $3 \times 3 \times C \times C = 9C^2$ parameters
- Total: $3 \times 9C^2 = 27C^2$ parameters
- Receptive field: $3 \to 5 \to 7$ (same as 7×7)

Parameter Reduction:

$$\text{Reduction} = \frac{49C^2 - 27C^2}{49C^2} \tag{15}$$

$$= \frac{22C^2}{49C^2} \tag{16}$$

$$= \frac{22}{49} \tag{17}$$

$$= 0.449 \tag{18}$$

$$= 44.9\% \tag{19}$$

Nearly half the parameters for same receptive field!

Why Stacked Small Filters Outperform:

1. More Non-linearity:

- Single 7×7: Input → Conv → ReLU → Output (1 non-linearity)
- Three 3×3: Input → Conv → ReLU → Conv → ReLU → Conv → ReLU → Output (3 non-linearities)
- More non-linearities = more expressive power
- Can learn more complex functions

2. Implicit Regularization:

- Fewer parameters = less overfitting
- Forces network to learn hierarchical features
- Each layer must produce useful intermediate representations
- Can't "cheat" with one large transformation

3. Computational Efficiency:

- FLOPs for 7×7: $H \times W \times C \times 49 \times C$
- FLOPs for 3×3×3: $3 \times H \times W \times C \times 9 \times C$
- Ratio: $\frac{3\times9}{49} = \frac{27}{49} = 55\%$ of computation

4. Gradient Flow:

- Deeper architecture with skip connections works better
- Gradients have multiple paths
- Less vanishing gradient issues

5. Feature Reuse:

- Intermediate features from layer 1,2 can be useful
- Modern architectures (DenseNet) exploit this
- Single 7×7 has no intermediate features

This principle led to VGGNet's design: all 3×3 filters, very deep. It's now standard practice!

(c) Prove translation equivariance property mathematically: (4 marks)

- Formal proof: If $f(T_\delta(x)) = T_\delta(f(x))$ for translation T_δ
- Show why this property doesn't extend to rotation or scaling

Answer: Mathematical Proof of Translation Equivariance:

Definitions:

- x: input image
- f: convolution operation
- T_δ: translation by vector $\delta = (\delta_x, \delta_y)$

- $T_\delta(x)[i,j] = x[i-\delta_x, j-\delta_y]$

To Prove: $f(T_\delta(x)) = T_\delta(f(x))$

Proof:

Convolution is defined as:

$$f(x)[i,j] = \sum_{m,n} w[m,n] \cdot x[i+m, j+n]$$

Left side: $f(T_\delta(x))[i,j]$

$$f(T_\delta(x))[i,j] = \sum_{m,n} w[m,n] \cdot T_\delta(x)[i+m, j+n] \tag{20}$$

$$= \sum_{m,n} w[m,n] \cdot x[(i+m)-\delta_x, (j+n)-\delta_y] \tag{21}$$

$$= \sum_{m,n} w[m,n] \cdot x[(i-\delta_x)+m, (j-\delta_y)+n] \tag{22}$$

Right side: $T_\delta(f(x))[i,j]$

$$T_\delta(f(x))[i,j] = f(x)[i-\delta_x, j-\delta_y] \tag{23}$$

$$= \sum_{m,n} w[m,n] \cdot x[(i-\delta_x)+m, (j-\delta_y)+n] \tag{24}$$

Both sides are equal! Translation equivariance is proven.

Why Not Rotation Equivariant:

Consider 90° rotation R_{90}:

Convolution filter:

$$w = \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

(Vertical edge detector)

On original image: Detects vertical edges **On rotated image:** Original vertical edges are now horizontal **But filter still looks for vertical patterns:** Fails to detect!

For rotation equivariance, we'd need:

$$f(R_{90}(x)) = R_{90}(f(x))$$

But the filter itself needs to rotate too! Standard convolution uses fixed filters.

Why Not Scale Equivariant:

Consider 2× scaling S_2:

Problem:

- 3×3 filter on original: Sees complete edge
- 3×3 filter on 2× scaled: Sees only part of (now thicker) edge
- Different pattern → different response
- Output doesn't simply scale by 2×

Mathematical issue: Scaling changes the frequency content. A fixed-size filter is tuned to specific frequencies and can't adapt.

Solutions:

- Data augmentation: Train with multiple scales/rotations
- Special architectures: Spatial Transformer Networks
- Multi-scale processing: Feature pyramids

Question 4. Advanced Convolution Types and Efficiency Analysis (30 marks)
Based on comprehensive analysis from multiple university deep learning courses.

(a) Analyze depthwise separable convolution efficiency: (12 marks)

- Standard convolution FLOPs: $M \times D_k^2 \times D_p^2 \times N$
- Depthwise separable FLOPs: $M \times D_k^2 \times D_p^2 + M \times D_p^2 \times N$
- Calculate reduction ratio: $\frac{1}{N} + \frac{1}{D_k^2}$ for large N
- Apply to MobileNet example: 3×3 depthwise + 1×1 pointwise vs 3×3 standard

Answer: Understanding Depthwise Separable Convolution

Depthwise separable convolution splits a standard convolution into two steps: 1. Depthwise convolution: Spatial filtering per channel 2. Pointwise convolution: 1×1 conv to mix channels

Let's define our variables clearly:

- M: Number of input channels
- N: Number of output channels
- D_k: Kernel size (e.g., 3 for 3×3)
- D_p: Output spatial dimension (height/width)

Standard Convolution FLOPs:

For each output position and channel:

- Apply one $D_k \times D_k$ kernel
- Across all M input channels
- FLOPs per output: $D_k^2 \times M$
- Total positions: $D_p^2 \times N$

$$\boxed{\text{Standard FLOPs} = D_p^2 \times N \times D_k^2 \times M = M \times D_k^2 \times D_p^2 \times N}$$

Depthwise Separable FLOPs:

Step 1 - Depthwise:

- One $D_k \times D_k$ kernel per input channel
- No cross-channel mixing
- FLOPs: $M \times D_k^2 \times D_p^2$

Step 2 - Pointwise (1×1):

- Mix M channels to produce N channels
- FLOPs: $M \times N \times D_p^2$

$$\boxed{\text{Depthwise Separable FLOPs} = M \times D_k^2 \times D_p^2 + M \times D_p^2 \times N}$$

Reduction Ratio:

$$\text{Ratio} = \frac{\text{Depthwise Separable FLOPs}}{\text{Standard FLOPs}} \tag{25}$$

$$= \frac{M \times D_k^2 \times D_p^2 + M \times D_p^2 \times N}{M \times D_k^2 \times D_p^2 \times N} \tag{26}$$

$$= \frac{M \times D_p^2(D_k^2 + N)}{M \times D_p^2 \times D_k^2 \times N} \tag{27}$$

$$= \frac{D_k^2 + N}{D_k^2 \times N} \tag{28}$$

$$= \frac{1}{N} + \frac{1}{D_k^2} \tag{29}$$

For large N: Ratio $\approx \frac{1}{D_k^2}$

MobileNet Example:

Compare 3×3 standard vs depthwise separable:

- Input: 112×112×32 (M=32)
- Output: 112×112×64 (N=64)
- Kernel: 3×3 ($D_k = 3$)

- Spatial: 112×112 ($D_p = 112$)

Standard 3×3 Conv:

$$\text{FLOPs} = 32 \times 3^2 \times 112^2 \times 64 \tag{30}$$
$$= 32 \times 9 \times 12,544 \times 64 \tag{31}$$
$$= 231,211,008 \tag{32}$$

Depthwise Separable:

$$\text{Depthwise} = 32 \times 9 \times 12,544 = 3,612,672 \tag{33}$$
$$\text{Pointwise} = 32 \times 12,544 \times 64 = 25,690,112 \tag{34}$$
$$\text{Total} = 29,302,784 \tag{35}$$

Reduction:

$$\text{Speedup} = \frac{231,211,008}{29,302,784} = 7.89\times \tag{36}$$
$$\text{Theory} = \frac{1}{64} + \frac{1}{9} = 0.0156 + 0.111 = 0.127 \tag{37}$$
$$\text{Speedup} = \frac{1}{0.127} = 7.87 \times \checkmark \tag{38}$$

Nearly 8× fewer operations! This enables real-time inference on mobile devices.

(b) Derive mathematical formulation for dilated convolution: (10 marks)

- Standard convolution: $y[m,n] = \sum_{i,j} x[m+i, n+j] \cdot h[i,j]$
- Dilated convolution: $y[m,n] = \sum_{i,j} x[m+d \cdot i, n+d \cdot j] \cdot h[i,j]$
- Calculate effective receptive field: $(k-1) \times d + 1$ for kernel size k, dilation d
- Analyze multi-scale feature extraction capabilities

Answer: Mathematical Formulation of Dilated Convolution

Dilated (atrous) convolution introduces spacing between kernel elements.

Standard Convolution: For kernel h of size $k \times k$ centered at origin:

$$y[m,n] = \sum_{i=-\lfloor k/2 \rfloor}^{\lfloor k/2 \rfloor} \sum_{j=-\lfloor k/2 \rfloor}^{\lfloor k/2 \rfloor} x[m+i, n+j] \cdot h[i,j]$$

Samples at consecutive positions: ..., $m-1$, m, $m+1$, ...

Dilated Convolution:

With dilation rate d:

$$y[m,n] = \sum_{i=-\lfloor k/2 \rfloor}^{\lfloor k/2 \rfloor} \sum_{j=-\lfloor k/2 \rfloor}^{\lfloor k/2 \rfloor} x[m+d \cdot i, n+d \cdot j] \cdot h[i,j]$$

Samples at positions: ..., $m-d$, m, $m+d$, ...

Visualization for 3×3 kernel:

- Standard (d=1): Sample at offsets (-1,-1), (-1,0), ..., (1,1)
- Dilated (d=2): Sample at offsets (-2,-2), (-2,0), ..., (2,2)
- Dilated (d=3): Sample at offsets (-3,-3), (-3,0), ..., (3,3)

Effective Receptive Field:

Derivation: For kernel size k and dilation d:

- Leftmost sample: $-\lfloor k/2 \rfloor \times d$
- Rightmost sample: $\lfloor k/2 \rfloor \times d$
- Coverage: From $-\lfloor k/2 \rfloor \times d$ to $\lfloor k/2 \rfloor \times d$
- Total span: $2 \times \lfloor k/2 \rfloor \times d + 1$

For odd k: $\lfloor k/2 \rfloor = (k-1)/2$

$$\boxed{\text{Effective RF} = (k-1) \times d + 1}$$

Examples:

- 3×3, d=1: $(3-1) \times 1 + 1 = 3$
- 3×3, d=2: $(3-1) \times 2 + 1 = 5$

- 3×3, d=4: $(3-1) \times 4 + 1 = 9$
- 5×5, d=3: $(5-1) \times 3 + 1 = 13$

Multi-Scale Feature Extraction:

1. Exponentially Increasing Receptive Fields: Stack dilated convolutions with rates 1, 2, 4, 8:

- Layer 1 (d=1): RF = 3×3
- Layer 2 (d=2): RF = 7×7 (cumulative)
- Layer 3 (d=4): RF = 15×15
- Layer 4 (d=8): RF = 31×31

Exponential growth without losing resolution!

2. Parallel Multi-Scale Processing: Use multiple dilations in parallel (ASPP - Atrous Spatial Pyramid Pooling):

- Branch 1: d=1 (fine details)
- Branch 2: d=6 (medium context)
- Branch 3: d=12 (large context)
- Branch 4: d=18 (global context)
- Concatenate all outputs

3. Advantages for Dense Prediction:

- Maintains spatial resolution (no pooling)
- Captures multi-scale context
- Fewer parameters than large kernels
- Critical for segmentation, depth estimation

4. Gridding Artifacts: Caution: Consecutive dilated convs can create "gridding" patterns. Solution: Use different dilation rates (1,2,5 instead of 1,2,4).

(c) Design and analyze transposed convolution for upsampling: (8 marks)

- Mathematical relationship: If conv maps $\mathbb{R}^n \to \mathbb{R}^m$, transposed conv maps $\mathbb{R}^m \to \mathbb{R}^n$

- Matrix formulation: $y = Cx$ vs $x = C^T y$
- Calculate output size: $O = (I-1) \times S - 2P + K$ for input I, stride S, padding P, kernel K

Answer: Understanding Transposed Convolution

Transposed convolution (deconvolution) reverses the forward/backward passes of standard convolution.

Key Insight: It's NOT the inverse operation! It's the transpose of the convolution operation when viewed as matrix multiplication.

Matrix Formulation:

Standard Convolution as Matrix: Flatten input $x \in \mathbb{R}^n$ and output $y \in \mathbb{R}^m$:

$$y = Cx$$

Where $C \in \mathbb{R}^{m \times n}$ encodes the convolution operation.

Example: 1D conv, input size 4, kernel size 3, stride 1:

$$C = \begin{bmatrix} w_1 & w_2 & w_3 & 0 \\ 0 & w_1 & w_2 & w_3 \end{bmatrix}_{2\times 4}$$

Maps $\mathbb{R}^4 \to \mathbb{R}^2$

Transposed Convolution:

$$x' = C^T y'$$

Where $C^T \in \mathbb{R}^{n \times m}$:

$$C^T = \begin{bmatrix} w_1 & 0 \\ w_2 & w_1 \\ w_3 & w_2 \\ 0 & w_3 \end{bmatrix}_{4\times 2}$$

Maps $\mathbb{R}^2 \to \mathbb{R}^4$ (reverses dimensions!)

Output Size Formula Derivation:

For transposed convolution, we reverse the roles:

Standard conv formula: $O = \lfloor \frac{I+2P-K}{S} \rfloor + 1$

Transposed conv reverses this relationship: Given input I' (which was output of forward conv), find output O':

Start with: $I' = \lfloor \frac{O'+2P-K}{S} \rfloor + 1$

Solve for O':

$$I' - 1 = \lfloor \frac{O' + 2P - K}{S} \rfloor \tag{39}$$

$$(I' - 1) \times S = O' + 2P - K \tag{40}$$

$$O' = (I' - 1) \times S - 2P + K \tag{41}$$

$$\boxed{O = (I - 1) \times S - 2P + K}$$

Note: This uses "inverse padding" - padding in transposed conv actually crops!

Practical Example:

Upsample 7×7 to 14×14 using transposed conv:

- Input: $I = 7$
- Desired output: $O = 14$
- Choose: $K = 4$, $S = 2$, $P = 1$

Verify:

$$O = (7 - 1) \times 2 - 2(1) + 4 \tag{42}$$

$$= 6 \times 2 - 2 + 4 \tag{43}$$

$$= 12 - 2 + 4 \tag{44}$$

$$= 14\checkmark \tag{45}$$

Common Issue - Checkerboard Artifacts:

- Overlapping regions get added multiple times
- Creates uneven brightness patterns
- Solution: Use resize + conv instead
- Or carefully choose kernel size = multiple of stride

Question 5. Neuroscience Foundation and Architectural Evolution (20 marks)
Based on MIT 6.034 and CMU historical perspectives on neural computation.

(a) Trace the evolution from biological vision to artificial CNNs: (12 marks)

- Hubel & Wiesel (1959): Simple cells (orientation & location specific) vs Complex cells (translation invariant)
- Fukushima's Neocognitron (1980): S-cells (feature detection) vs C-cells (spatial pooling)
- Modern CNNs: Convolution layers vs Pooling layers
- Analyze preserved principles and key innovations at each stage

Answer: The Journey from Biology to Artificial Intelligence

This is a fascinating story of how neuroscience discoveries led to one of AI's greatest breakthroughs.

1. Hubel & Wiesel (1959-1962) - The Biological Foundation

The Experiments:

- Inserted electrodes into cat visual cortex
- Showed various visual stimuli (dots, bars, edges)
- Recorded individual neuron responses
- Nobel Prize in 1981 for this work

Key Discoveries:

Simple Cells:

- Respond to bars/edges at SPECIFIC orientations (0°, 45°, 90°, etc.)
- AND at SPECIFIC locations in visual field
- Very precise requirements: wrong angle = no response
- Like having specialized "edge detectors" at each location

Complex Cells:

- Respond to correct orientation ANYWHERE in receptive field

- Translation invariance within local region
- Seem to pool responses from multiple simple cells
- First evidence of hierarchical processing

Hypercomplex Cells:

- Respond to specific lengths, corners, or end-stopped lines
- Even more sophisticated pattern detection
- Build on complex cell inputs

2. Fukushima's Neocognitron (1980) - First Implementation

Direct Biological Inspiration: Fukushima explicitly modeled Hubel & Wiesel's findings:

S-cells (Simple) → Convolution:

- Detect specific patterns at specific locations
- Hand-designed templates (Gabor filters, edge detectors)
- Arranged in planes (feature maps)
- Each S-cell plane detects one feature type

C-cells (Complex) → Pooling:

- Pool S-cell responses over local regions
- Provide position tolerance (translation invariance)
- Blur operation preserving strongest responses
- Reduce spatial resolution progressively

Architecture: S1 → C1 → S2 → C2 → S3 → C3 → S4 → C4 (Alternating feature extraction and pooling)

Limitations:

- No backpropagation (trained layer by layer)
- Hand-designed features
- Limited to simple tasks (digit recognition)
- No end-to-end optimization

3. Modern CNNs (1989-present) - The Revolution

LeCun's Key Innovation (1989): Backpropagation through the entire network!

Convolution Layers (Modern S-cells):

- LEARNED filters, not hand-designed
- Shared weights (parameter efficiency)
- Multiple channels (feature maps)
- ReLU activation (better than sigmoid)

Pooling Layers (Modern C-cells):

- Max pooling (keep strongest activation)
- Average pooling (smooth aggregation)
- Stride ¿ 1 for dimension reduction
- Sometimes replaced by strided convolution

Preserved Principles:

1. **Hierarchical Processing:**
   - Biology: Simple → Complex → Hypercomplex
   - CNNs: Edges → Textures → Parts → Objects
   - Deep layers see progressively larger regions
2. **Local Connectivity:**
   - Biology: Neurons have limited receptive fields
   - CNNs: Convolution kernels are small (3×3, 5×5)
   - Efficient use of parameters
3. **Feature Maps:**
   - Biology: Columnar organization, similar features grouped
   - CNNs: Multiple filters create feature map channels
   - Parallel processing of different features

4. **Translation Invariance:**
   - Biology: Complex cells provide local invariance
   - CNNs: Combination of convolution + pooling
   - Robust to small shifts

Key Innovations Beyond Biology:

1. **End-to-End Learning:**
   - Biology: Partially genetic, partially learned
   - CNNs: Everything learned from data
   - Task-specific optimization
2. **Depth:**
   - Biology: 6-layer cortex typical
   - CNNs: 100+ layers possible (ResNet)
   - Skip connections prevent vanishing gradients
3. **Global Pooling:**
   - Biology: Complex feedback mechanisms
   - CNNs: Global average pooling for classification
   - Direct mapping to class probabilities
4. **Batch Normalization:**
   - No clear biological equivalent
   - Stabilizes training dramatically
   - Enables much deeper networks

The beautiful insight: Nature solved vision first, and we successfully reverse-engineered key principles!

(b) Compare biological vs artificial receptive field organizations: (8 marks)

- Biological: Variable RF sizes (foveal vs peripheral vision)
- Artificial: Fixed RF sizes across spatial locations

- Attention mechanisms as bridge to biological variability
- Trade-offs between biological realism and computational efficiency

Answer: Biological Receptive Field Organization

The human visual system is remarkably non-uniform:

Foveal Vision (Center):

- Tiny receptive fields (0.01° visual angle)
- High density of photoreceptors (cones)
- Used for detailed tasks: reading, face recognition
- Only 2° of visual field (size of thumbnail at arm's length)
- Processes fine spatial frequencies

Peripheral Vision:

- Large receptive fields (up to 10° visual angle)
- Low density, mostly rods
- Detects motion, general shapes
- Poor detail but wide coverage
- Processes low spatial frequencies

The Gradient: RF size increases smoothly from center to periphery This creates a log-polar sampling pattern Evolution optimized for: detail where we look, awareness elsewhere

Artificial CNN Organization

Standard CNNs use uniform sampling:

Fixed Receptive Fields:

- Every position uses same kernel size (e.g., 3×3)
- No distinction between "important" and "background" regions
- Treats top-left corner same as image center
- Computational democracy: every pixel gets equal processing

Why This Design:

- Simplicity: One kernel design, reused everywhere
- Hardware efficiency: Regular patterns optimize well
- Translation equivariance: Same features detected anywhere
- No prior assumptions about importance

Attention Mechanisms - Bridging the Gap

Modern attention mechanisms add biological-style focus:

Spatial Attention:

- Learn WHERE to look (like eye movements)
- Weight different regions by importance
- Dynamically allocate computation
- Example: Focus on faces in a crowd

Channel Attention:

- Learn WHAT features matter
- Weight different feature maps
- Task-dependent feature selection
- Example: Texture for materials, edges for sketches

Deformable Convolution:

- Adaptive receptive fields
- Learn offsets for each position
- Most biological-like mechanism
- RFs adapt to image content

Trade-offs Analysis

Biological Advantages:

- Efficiency: More processing where it matters
- Natural saliency: Built-in importance sampling
- Motion detection: Peripheral vision specialized for this

- Energy efficient: Brain uses 20W total

Biological Disadvantages:

- Requires eye movements to see details everywhere
- Can miss peripheral details
- Complex to implement/simulate
- Not translation equivariant

CNN Advantages:

- Uniformity enables parallelization
- No blind spots or preferred locations
- Simpler to train and understand
- Better for tasks requiring global analysis

CNN Disadvantages:

- Wasteful: Same computation on "boring" regions
- Can't focus on details without processing everything
- Larger memory footprint
- Less robust to scale variations

Future Directions:

- Foveated rendering in VR (biological inspiration)
- Adaptive computation time (process until confident)
- Learned routing (conditional computation)
- Neuromorphic hardware (event-driven processing)

The trend: As computation becomes cheaper, we're adding back biological complexity where it provides clear benefits!

Question 6. Deformable Convolution Mathematical Framework (25 marks)
Based on advanced computer vision research and graduate-level analysis.

(a) Derive the mathematical formulation of deformable convolution: (15 marks)

- Standard convolution: $y(p_0) = \sum_{p_n \in \mathcal{R}} w(p_n) \cdot x(p_0 + p_n)$
- Deformable convolution: $y(p_0) = \sum_{p_n \in \mathcal{R}} w(p_n) \cdot x(p_0 + p_n + \Delta p_n)$
- Explain learnable offset Δp_n estimation network
- Derive bilinear interpolation for non-integer sampling locations

Answer: Mathematical Framework for Deformable Convolution

Let's build the complete mathematical formulation step by step.

Standard 2D Convolution:

For position $p_0 = (x_0, y_0)$ in the output feature map:

$$y(p_0) = \sum_{p_n \in \mathcal{R}} w(p_n) \cdot x(p_0 + p_n)$$

Where:

- $\mathcal{R}$ is the regular grid of sampling locations
- For 3×3 kernel: $\mathcal{R} = \{(-1,-1), (-1,0), ..., (1,1)\}$
- $w(p_n)$ is the weight at offset p_n
- $x(p_0 + p_n)$ is the input value at position $p_0 + p_n$

This samples on a rigid grid - always the same pattern.

Deformable Convolution:

Add learnable offsets to each sampling position:

$$y(p_0) = \sum_{p_n \in \mathcal{R}} w(p_n) \cdot x(p_0 + p_n + \Delta p_n)$$

Where:

- $\Delta p_n = (\Delta x_n, \Delta y_n)$ is the learned offset for position p_n
- Offsets are different for each output position p_0
- Now sampling at: $p_0 + p_n + \Delta p_n$ (fractional coordinates)

Offset Learning Network:

The offsets are predicted by a parallel convolutional branch:

Architecture:

- Input feature map: $\mathbf{F} \in \mathbb{R}^{H \times W \times C}$
- Offset predictor: Conv layer with $2|\mathcal{R}|$ output channels
- For 3×3 kernel: $2 \times 9 = 18$ channels (x,y offset for each position)
- Output: $\Delta \in \mathbb{R}^{H \times W \times 18}$

Offset Generation:

$$\Delta = \text{Conv}_{offset}(\mathbf{F})$$

The conv typically uses same kernel size as main convolution to maintain spatial correspondence.

Per-position Offsets: For each output position (x_0, y_0):

- Extract 18 values from $\Delta[x_0, y_0, :]$
- Reshape to 9 offset pairs: $\{(\Delta x_1, \Delta y_1), ..., (\Delta x_9, \Delta y_9)\}$
- These offsets deform the 3×3 sampling grid at this position

Bilinear Interpolation:

Since $p_0 + p_n + \Delta p_n$ is fractional, we need interpolation:

Let $p = (x, y) = p_0 + p_n + \Delta p_n$ be the fractional position.

Four Nearest Integer Positions:

- Top-left: $q_{11} = (\lfloor x \rfloor, \lfloor y \rfloor)$
- Top-right: $q_{21} = (\lceil x \rceil, \lfloor y \rfloor)$
- Bottom-left: $q_{12} = (\lfloor x \rfloor, \lceil y \rceil)$
- Bottom-right: $q_{22} = (\lceil x \rceil, \lceil y \rceil)$

Bilinear Weights: Let fractional parts be: $a = x - \lfloor x \rfloor$, $b = y - \lfloor y \rfloor$

$$w_{11} = (1-a)(1-b) \tag{46}$$
$$w_{21} = a(1-b) \tag{47}$$
$$w_{12} = (1-a)b \tag{48}$$
$$w_{22} = ab \tag{49}$$

Interpolated Value:

$$x(p) = \sum_{q \in \{q_{11}, q_{21}, q_{12}, q_{22}\}} w_q \cdot x(q)$$

Or expanded:

$$x(p) = (1-a)(1-b)x(q_{11}) + a(1-b)x(q_{21}) + (1-a)bx(q_{12}) + abx(q_{22})$$

Complete Forward Pass:

1. Input: Feature map $\mathbf{F}$ 2. Predict offsets: $\Delta = \text{Conv}_{offset}(\mathbf{F})$ 3. For each output position (x_0, y_0):

- Extract offsets for this position
- For each kernel position p_n:
  - Compute sampling location: $p = p_0 + p_n + \Delta p_n$
  - Bilinear interpolation at p
  - Multiply by kernel weight $w(p_n)$
- Sum all contributions

4. Output: Deformed convolution result

Gradient Computation: The beauty is that everything is differentiable!

- Gradients flow through bilinear interpolation weights
- Offset network learns to position samples optimally
- End-to-end training with standard backprop

(b) Analyze computational complexity and training considerations: (10 marks)

- Parameter overhead: $2n$ additional offset parameters for n-point kernel
- Forward pass complexity: Standard conv + offset prediction + bilinear interpolation
- Backward pass: Gradient flow through both content and spatial transformations
- Memory requirements and training stability analysis

Answer: Computational Complexity Analysis

Let's analyze each component systematically:

Given:

- Input: $H \times W \times C_{in}$
- Output: $H \times W \times C_{out}$
- Kernel: $k \times k$ (e.g., 3×3)
- Number of kernel points: $n = k^2$

Parameter Overhead:

Standard Convolution:

- Weights: $k \times k \times C_{in} \times C_{out}$
- Biases: C_{out}
- Total: $k^2 C_{in} C_{out} + C_{out}$

Deformable Convolution:

- Main conv weights: $k^2 C_{in} C_{out} + C_{out}$ (same)
- Offset conv weights: $k^2 C_{in} \times 2k^2 + 2k^2$
- For 3×3: $9 \times C_{in} \times 18 + 18 = 162 C_{in} + 18$
- Overhead ratio: $\frac{162 C_{in}}{9 C_{in} C_{out}} \approx \frac{18}{C_{out}}$

For typical $C_{out} = 256$: Only 7% parameter increase!

Forward Pass Complexity:

Breaking down operations per output position:

1. Offset Prediction:

- Conv operation: $k^2 \times C_{in} \times 2k^2$ FLOPs
- For 3×3: $9 \times C_{in} \times 18 = 162C_{in}$ FLOPs
- Per position in $H \times W$ output

2. Bilinear Interpolation: For each of k^2 kernel positions:

- Compute fractional coordinates: 2 ops
- Calculate 4 weights: 8 ops
- Sample 4 pixels: 4 memory reads
- Interpolate: 7 ops (4 muls + 3 adds)
- Total: 21 ops per kernel position
- For 3×3: $9 \times 21 = 189$ ops

3. Main Convolution:

- Standard: $k^2 \times C_{in}$ multiply-adds
- But with irregular memory access (slower)

Total Overhead:

- Computation: 3-4× standard convolution
- Memory bandwidth: 4× (bilinear sampling)
- Actual runtime: 2-3× (memory bound)

Backward Pass Complexity:

More complex due to spatial transformation gradients:

1. Gradient w.r.t Input:

- Standard path: Through convolution weights
- Additional path: Through offset network

- Must accumulate gradients from all positions that sampled each pixel
- Irregular scatter operation (not gather)

2. Gradient w.r.t Offsets:

- Flows through bilinear interpolation
- $\frac{\partial \text{output}}{\partial \Delta x} = \frac{\partial \text{output}}{\partial \text{interp}} \cdot \frac{\partial \text{interp}}{\partial \Delta x}$
- Involves gradients of sampling positions

3. Gradient w.r.t Weights:

- Similar to standard convolution
- But uses interpolated values

Backward pass: 4-5× standard convolution

Memory Requirements:

Additional Storage Needs:

- Offset maps: $H \times W \times 2k^2 \times 4$ bytes
- For 256×256×3×3: 256×256×18×4 = 4.5MB
- Offset gradients: Same size
- Interpolation coordinates: Temporary, can be recomputed

Memory Access Patterns:

- Standard conv: Predictable, cache-friendly
- Deformable: Random access based on learned offsets
- Poor cache utilization
- GPU memory bandwidth becomes bottleneck

Training Stability Considerations:

1. Offset Initialization:

- Initialize offset conv with zeros
- Starts as standard convolution

- Gradually learns deformations
- Large initial offsets cause instability

2. Offset Regularization:

- Unconstrained offsets can grow large
- May sample outside image boundaries
- Common: Add L2 penalty on offset magnitudes
- Or clip offsets to reasonable range

3. Gradient Scaling:

- Offset gradients can be larger than weight gradients
- May need different learning rates
- Common: Use 0.1× learning rate for offset network

4. Boundary Handling:

- Sampling outside image needs special care
- Options: Zero padding, edge replication, or reflection
- Must be differentiable

Best Practices:

- Start with pre-trained standard CNN
- Add deformable convs to later layers only
- Monitor offset magnitudes during training
- Use gradient clipping if needed
- Expect 2-3× slower training

Question 7. GPU Optimization and Implementation Efficiency (20 marks)
Based on NVIDIA Deep Learning Institute and high-performance computing courses.

(a) Analyze GPU-friendly convolution implementation strategies: (10 marks)

- Im2col transformation: Convert convolution to GEMM operations
- Winograd algorithm: Reduce multiplication count for small kernels
- Direct convolution: Optimized for specific kernel sizes
- Memory access patterns and cache utilization

Answer: GPU Architecture Considerations

Modern GPUs (e.g., NVIDIA V100, A100) have:

- Thousands of CUDA cores for parallel computation
- Tensor cores for matrix multiplication
- Limited memory bandwidth (900 GB/s)
- Small caches per SM (Streaming Multiprocessor)
- Optimized for regular memory access patterns

Convolution must be implemented to match these constraints.

1. Im2col + GEMM Strategy

Why GEMM is King:

- GPUs have highly optimized GEMM (General Matrix Multiply)
- cuBLAS achieves ¿95% of theoretical peak performance
- Tensor cores specifically designed for matrix ops
- Regular memory access patterns

Implementation: “‘ 1. Im2col: Unfold input patches into matrix columns 2. Reshape filters into matrix rows 3. Y = Filter$_{m}atrixInput_{m}atrix(GEMM)4.Resha$

Memory Analysis:

- Pro: Extremely fast GEMM computation

- Con: Im2col expansion uses k^2 more memory
- Memory bandwidth: Often the bottleneck
- Best for: Large batches, standard kernel sizes

2. Winograd Algorithm

Core Idea: Reduce multiplications using algebraic transformations.

For $F(2,3)$ (2×2 output, 3×3 kernel):

- Standard: 36 multiplications
- Winograd: 16 multiplications
- 2.25× fewer multiplications!

Algorithm: "' 1. Transform input tile: $\tilde{\mathrm{U}} = \mathrm{B}^T UB 2. Transform filter : G = GgG^T 3. Element - wise multiply : M = G\tilde{U} 4. Inverse transform : Y = A^T MA$"'

Where B, G, A are fixed transformation matrices.

GPU Considerations:

- Pro: Fewer arithmetic operations
- Pro: Transformations are small matrix ops (GPU-friendly)
- Con: Numerical stability issues with larger tiles
- Con: Extra memory for transformed data
- Best for: 3×3 convolutions, FP16 computation

3. Direct Convolution

Approach: Implement convolution directly with optimized CUDA kernels.

Optimization Techniques:

- Tiling: Each thread block processes a tile
- Shared memory: Cache filter weights and input tile
- Register blocking: Compute multiple outputs per thread
- Texture memory: For spatially local access

Example Kernel Structure: “‘cuda $_{global}{}_{voidconv}{}_{kernel(...)}{}_{shared}{}_{floattile[TILE}{}_{SIZE];}{}_{shared}{}_{floatfilter}{}_{shared[K}{}_{S}$

// Load filter to shared memory // Load input tile with halo $_{syncthreads}();$

// Compute convolution for this thread's output float sum = 0; for(int i = 0; i ¡ K$_{S}IZE; i++)sum+ = tile[...] * filter_shared[i]$; “‘

Best for:

- Unusual kernel sizes
- Depthwise convolution
- When memory is critical

4. Memory Access Optimization

Coalesced Access:

- Threads in warp access consecutive addresses
- 32 threads → 128-byte transaction (optimal)
- Im2col naturally provides this
- Direct conv needs careful indexing

Cache Utilization:

- L1 cache: 128KB per SM (Ampere)
- Tile computations to fit in L1
- Reuse data across thread block
- Texture cache for 2D spatial locality

Memory Bandwidth Calculation: For 3×3 conv, 256→256 channels:

- Read: Input (HW×256) + Weights (9×256×256)
- Write: Output (HW×256)
- Arithmetic Intensity = FLOPs/Bytes
- Need high AI for compute-bound (not memory-bound)

Optimization Decision Tree:

- Large batch + 3×3/5×5 → Winograd

- Any batch + standard sizes → Im2col + GEMM
- Depthwise/grouped → Direct convolution
- Memory limited → Direct or implicit GEMM

(b) Compare computational efficiency across convolution types: (10 marks)

- Standard vs Depthwise separable: FLOPs and memory bandwidth
- Group convolution: Parallelization benefits and limitations
- 1×1 convolution: Throughput optimization for channel mixing
- Batch processing effects on computational efficiency

Answer: Computational Efficiency Comparison

Let's analyze each convolution type for GPU execution efficiency.

1. Standard vs Depthwise Separable

Setup: Input 56×56×128 → Output 56×56×256, kernel 3×3

Standard Convolution:

- FLOPs: $56^2 \times 256 \times 3^2 \times 128 = 924M$
- Memory read: $(56^2 \times 128 + 3^2 \times 128 \times 256) \times 4B = 1.7MB$
- Memory write: $56^2 \times 256 \times 4B = 3.2MB$
- Arithmetic Intensity: $924M/4.9M = 189$ FLOPs/byte
- GPU utilization: Excellent (compute-bound)

Depthwise Separable:

Depthwise (3×3×128):

- FLOPs: $56^2 \times 128 \times 3^2 = 3.6M$
- Memory: $(56^2 \times 128 + 3^2 \times 128) \times 4B = 1.6MB$
- AI: $3.6M/1.6M = 2.25$ FLOPs/byte
- GPU utilization: Poor (memory-bound)

Pointwise (1×1):

- FLOPs: $56^2 \times 128 \times 256 = 103M$

- Memory: $(56^2 \times 128 + 128 \times 256) \times 4B = 1.7MB$
- AI: $103M/1.7M = 61$ FLOPs/byte
- GPU utilization: Good

Key Insight: Depthwise is memory-bound on GPUs! Despite fewer FLOPs, may run slower than standard conv on high-end GPUs.

2. Group Convolution

Splits channels into groups, processes independently.

Example: 128→256 channels, groups=4

- Each group: 32→64 channels
- 4 independent 32→64 convolutions
- Perfect parallelization opportunity

GPU Execution:

- Different groups → different thread blocks
- No inter-group communication needed
- Linear speedup with groups (ideal case)
- Memory access still coalesced within groups

Limitations:

- No cross-group information flow
- Need channel shuffle or 1×1 conv after
- Small groups may underutilize GPU
- Best with groups = 2,4,8 (power of 2)

Efficiency:

- FLOPs: Reduced by factor of groups
- Memory: Similar reduction
- AI: Stays similar (both scale equally)
- Practical speedup: 0.8-0.95× of theoretical

3. 1×1 Convolution Optimization

1×1 conv is just matrix multiplication at each spatial position!

Why It's Fast:

- No spatial operations (no im2col needed)
- Direct GEMM: $(C_out, C_in)(C_in, HW) Perfect for Tensor Cores$
- Highest arithmetic intensity

Optimization Strategies:

- Batch spatial dimensions: Process multiple positions together
- Use NHWC format: Better memory layout for 1×1
- Mixed precision: FP16 compute with FP32 accumulation
- Can achieve ¿90% of peak TFLOPS

Example Performance: V100 GPU: 125 TFLOPS (FP16)

- 1×1 conv: 110 TFLOPS (88% efficiency)
- 3×3 conv: 80 TFLOPS (64% efficiency)
- Depthwise: 20 TFLOPS (16% efficiency)

4. Batch Processing Effects

Batching dramatically improves GPU efficiency:

Small Batch (B=1):

- Limited parallelism
- Can't fill all SMs
- Kernel launch overhead significant
- Memory access less efficient

Large Batch (B=128):

- Full GPU utilization
- Amortize weight loading across batch
- Better cache utilization