MIDDLE EAST TECHNICAL UNIVERSITY

SEMESTER I EXAMINATION 2024-2025

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

January 2025 TIME ALLOWED: 3 HOURS

INSTRUCTIONS

1. This is the ANSWERED version with detailed explanations.
2. Each answer includes step-by-step reasoning to help you understand the concepts.
3. Pay attention to the connections between different concepts.
4. Focus on understanding WHY things work the way they do, not just memorizing.

Question 1. CNN Motivation and Limitations of Multi-Layer Perceptrons (25 marks)

Based on the professor's discussion: "If you want to work with real life large scale problems let's say object recognition with high resolution...you want to recognize objects in such high resolution images what we do is we vectorize...then we use multiperceptron."

(a) The professor calculated that for a 1000×1000 input image with 10,000 neurons in the first hidden layer, "just this layer is going to introduce 10 billion parameters." Explain why this parameter explosion is problematic and analyze the professor's statement that "more parameters means more data" with the quadratic relationship. (8 marks)

Answer: Let me break down why 10 billion parameters is a massive problem:

The Calculation:

- Input size: 1000 × 1000 pixels = 1,000,000 input values
- Hidden layer: 10,000 neurons
- Each neuron connects to EVERY input pixel
- Parameters = 1,000,000 × 10,000 = 10,000,000,000 (10 billion)
- Plus 10,000 bias terms (negligible compared to weights)

Why This Is Problematic:

1. **Memory Requirements:** If we store each parameter as a 32-bit float (4 bytes), we need: 10 billion × 4 bytes = 40 GB just for ONE layer's parameters! During training, we also need to store gradients, activations, and optimizer states, easily exceeding 100GB for this single layer.

2. **Data Requirements - The Quadratic Relationship:** The professor's key insight: as parameters increase, data requirements grow even faster.

- With 10 billion parameters, we risk severe overfitting
- Rule of thumb: need at least 10× more training examples than parameters
- This means we'd need 100+ billion training examples!

- The relationship is quadratic because: doubling parameters typically requires 4× more data to maintain the same generalization performance

3. **Computational Cost:** Each forward pass requires 10 billion multiply-add operations just for this layer. Training involves millions of forward and backward passes. This makes training prohibitively expensive and slow.

4. **Overfitting Risk:** With so many parameters, the network can memorize the entire training set rather than learning general patterns. Each parameter is a ”degree of freedom” - more parameters mean the model can fit increasingly complex (and potentially spurious) patterns.

(b) Analyze the professor’s examples of equivariance and invariance problems. For image segmentation, explain why we need ”if you transform the input...we expect the output to be transformed by the same transformation.” For object recognition, explain why we need the prediction to ”be independent of that transformation.” (10 marks)

Answer: Let me explain these two crucial but different requirements:

Equivariance for Image Segmentation:

Image segmentation assigns a label to EACH PIXEL (e.g., ”car”, ”road”, ”sky”).

Example scenario:

- Original image: Car in the center
- Segmentation output: Pixels labeled as ”car” in the center
- Now shift the image 10 pixels right
- Expected: The ”car” labels should ALSO shift 10 pixels right

Why this matters:

- The spatial relationship between input and output must be preserved
- If a pixel at position (100, 200) is labeled ”car”, and we shift the image by (10, 20), that same car pixel should now be labeled at position (110, 220)

- This is called EQUIVARIANCE: the output transforms in the same way as the input

Mathematical notation: If T is a transformation, we want: f(T(x)) = T(f(x))

Invariance for Object Recognition:

Object recognition produces a SINGLE label for the entire image (e.g., ”this image contains a cat”).

Example scenario:

- Original image: Cat in top-left corner → Output: ”cat”
- Shifted image: Cat in bottom-right corner → Output: still ”cat”
- The location doesn’t matter!

Why this matters:

- We want to recognize objects regardless of their position
- A cat is a cat whether it’s in the center, corner, or anywhere else
- This is called INVARIANCE: the output stays the same despite input transformations

Mathematical notation: We want: f(T(x)) = f(x)

The Key Difference:

- Segmentation: Spatial correspondence matters (equivariance)
- Recognition: Only content matters, not location (invariance)
- CNNs naturally provide equivariance through convolution
- CNNs achieve invariance through pooling layers that aggregate spatial information

(c) The professor showed how shifting an image by one pixel in a multilayer perceptron causes ”all of the activations...they are going to change.” Explain why this is a ”very severe limitation” and how CNNs address this through architectural design. (7 marks)

Answer: The MLP Problem with Pixel Shifts:

In an MLP, each neuron has a unique weight for each pixel position:

- Neuron 1 might have weight $w_{1,1}$ for pixel (0,0), $w_{1,2}$ for pixel (0,1), etc.
- When we shift the image by one pixel, EVERY pixel moves to a new position
- Pixel that was at (0,0) is now at (0,1)
- But the weights don't shift! Weight $w_{1,1}$ still looks at position (0,0)
- Result: Every neuron sees completely different input values

Why This Is a Severe Limitation:

1. **No Spatial Understanding:** The network learns "there should be an edge at pixel (100,200)" instead of "edges are important features wherever they appear"

2. **Poor Generalization:** A cat learned in the top-left won't be recognized in the bottom-right Each position needs to be learned separately, multiplying training data requirements

3. **Wasted Parameters:** We need different weights to detect the same pattern at each possible location

How CNNs Solve This:

1. **Shared Weights (Convolution Filters):**

- Same 3×3 filter slides across the entire image
- The filter that detects edges works the same way at EVERY position
- When image shifts, the same filter just detects the pattern at the new location

2. **Local Connectivity:**

- Each neuron only looks at a small region (e.g., 3×3 pixels)
- Shifting by one pixel means most of the receptive field content stays similar
- Only the edges of the receptive field change

3. **Translation Equivariance:**

- If input shifts right, the activation pattern also shifts right

- The network's response moves with the content
- This is exactly what we want for robust vision systems

Question 2. Neuroscience Inspiration and CNN Fundamentals (22 marks)

The professor discussed Hubel and Wiesel's Nobel Prize-winning findings: ”They did recordings real recordings from the brains of cats...they showed that...connectivity in biological neurons are not fully connected...neurons have receptive fields.”

(a) Explain the two key findings from Hubel and Wiesel that inspired CNN design: restricted connectivity and columnar structure. Describe how these translate to ”receptive fields” and specialized neurons for ”recognizing certain patterns at different scales.” (10 marks)

Answer: Hubel and Wiesel's Groundbreaking Discoveries:

They recorded electrical signals from individual neurons in cat visual cortex while showing various visual stimuli. This revealed how biological vision actually works.

Finding 1: Restricted Connectivity (Receptive Fields)

What they found:

- Each neuron in visual cortex responds only to stimuli in a SMALL region of the visual field
- This region is called the neuron's ”receptive field”
- A neuron might only ”see” a 5° × 5° patch of the visual world
- Different neurons have receptive fields in different locations

How this translates to CNNs:

- Convolutional neurons have small receptive fields (e.g., 3×3 or 5×5 pixels)
- Each neuron only connects to a local patch, not the entire image
- This dramatically reduces parameters: instead of connecting to all 1 million pixels, connect to just 9 (for 3×3)
- Different positions in the feature map correspond to different receptive field locations

Finding 2: Columnar Structure (Feature Specialization)

What they found:

- Neurons are organized in columns perpendicular to cortex surface
- Within a column, neurons respond to the SAME location but DIFFERENT features
- Simple cells: Respond to edges at specific orientations (0°, 45°, 90°, etc.)
- Complex cells: Respond to edges regardless of exact position within receptive field
- Hypercomplex cells: Respond to more complex patterns like corners or line endings

How this translates to CNNs:

- Multiple filters (channels) at each layer = different neurons in a column
- Each filter learns to detect different patterns
- Early layers: Simple patterns (edges, colors)
- Deeper layers: Complex patterns (textures, parts, objects)
- Hierarchical feature learning mimics simple → complex → hypercomplex progression

Pattern Recognition at Different Scales:

- In biology: Neurons deeper in visual pathway have larger receptive fields
- In CNNs: Deeper layers see larger image regions due to stacked convolutions
- Layer 1: 3×3 receptive field (detects edges)
- Layer 2: 5×5 effective receptive field (detects corners, simple textures)
- Layer 5: 51×51 effective receptive field (detects entire objects)
- This allows recognizing patterns from local edges to global objects

(b) Compare the evolution from Neocognitron (1979) to CNNs. Explain how Fukushima's ”simple cells” and ”complex cells” correspond to ”convolution” and ”pooling” in modern CNNs, and why gradient descent training was crucial for practical success. (8 marks)

Answer: Neocognitron (1979) - Fukushima's Architecture:

Fukushima directly implemented Hubel and Wiesel's findings in an artificial network:

S-cells (Simple Cells) → Modern Convolution Layers:

- **S-cells:** Detected specific patterns at specific locations
- Fixed, hand-designed templates (not learned)
- Responded strongly to exact pattern matches
- **Modern Convolution:** Learnable filters that detect patterns
- Filters are learned through backpropagation
- Can detect subtle variations of patterns

Key similarity: Both perform template matching with spatial specificity Key difference: Convolution filters are learned, not hand-designed

C-cells (Complex Cells) → Modern Pooling Layers:

- **C-cells:** Provided position tolerance within local regions
- Responded if pattern appeared anywhere in a small area
- Created invariance to small shifts
- **Modern Pooling:** Max or average over local regions
- Reduces spatial resolution while keeping important features
- Provides translation invariance and computational efficiency

Key similarity: Both aggregate local information to create position tolerance Key difference: Pooling is simpler (just max/average) vs complex cell computations

Why Gradient Descent Was Crucial:

1. **Learning vs Hand-Design:**

- Neocognitron required manually designing each S-cell template

- Impossible to hand-design thousands of filters for complex tasks
- Gradient descent automatically learns optimal filters from data

2. **End-to-End Optimization:**

   - Neocognitron trained layer-by-layer with unsupervised rules
   - No guarantee that features useful for final task
   - Backpropagation optimizes all layers jointly for the specific task

3. **Scalability:**

   - Gradient descent scales to millions of parameters
   - Can learn hierarchical representations automatically
   - Enables training on massive datasets (ImageNet: 1.2M images)

4. **Performance Gap:**

   - Neocognitron: Limited to simple digit recognition
   - Modern CNNs: State-of-the-art on complex real-world vision tasks
   - The difference: learned features far exceed hand-designed ones

(c) The professor mentioned that in CNNs we use ”fixed receptive field size for every neuron” unlike the brain where ”in the central parts...we have smaller receptive field...in the peripheries...we have higher receptive field.” Discuss the implications of this design choice. (4 marks)

Answer: Biological Vision - Variable Receptive Fields:

The human visual system has:

- **Fovea (center):** Tiny receptive fields, high resolution
- Used for detailed tasks like reading
- **Periphery:** Large receptive fields, low resolution
- Used for motion detection, general awareness
- This matches how we use vision: focus on details at center, monitor surroundings

CNNs - Fixed Receptive Fields:

All neurons in a CNN layer have identical receptive field sizes (e.g., all use 3×3).

Implications of Fixed Size:

Advantages:

- **Computational Efficiency:** Uniform operations enable GPU optimization
- **Parameter Sharing:** Same weights used everywhere reduces memory
- **Simplicity:** Easier to implement and reason about
- **Translation Equivariance:** Consistent behavior across image

Disadvantages:

- **Inefficiency:** Same computation everywhere, even in ”boring” regions
- **No Attention:** Can’t focus more resources on important areas
- **Scale Limitations:** Fixed size may miss multi-scale patterns

Modern Solutions:

- **Attention Mechanisms:** Dynamically weight different regions
- **Deformable Convolutions:** Adaptive receptive fields
- **Multi-Scale Architectures:** Different branches for different scales
- These bring CNNs closer to biological flexibility while maintaining efficiency

Question 3. CNN Architecture and Parameter Sharing (20 marks)
The professor emphasized: ”The first critical bit is that connectivity is restricted...the second important change...is that the weights are shared...we have the same parameters shared and used by every receptive field.”

(a) Explain how restricted connectivity and parameter sharing address the dimensionality problem. The professor stated: ”if this is 1 million...the number of parameters here it doesn't depend on that actually...we just have three parameters W1 W2 W3.” (8 marks)

Answer: Understanding the Dimensionality Problem:

First, let's see what happens WITHOUT these two innovations:

- Input: 1 million pixels (1000×1000 image)
- Hidden layer: 10,000 neurons
- Full connectivity: 1M × 10K = 10 billion parameters

Solution 1: Restricted Connectivity

Instead of connecting to ALL pixels, each neuron connects to a SMALL LOCAL REGION:

- Example: 3×3 receptive field = 9 connections per neuron
- Even with 10,000 neurons: 9 × 10,000 = 90,000 parameters
- Reduction: From 10 billion to 90,000 (99.999% fewer!)

But wait, this alone isn't enough...

Solution 2: Parameter Sharing

The KEY INSIGHT: Use the SAME weights for every spatial location!

The professor's example with W1, W2, W3:

- Imagine a 1D convolution with 3 weights: [W1, W2, W3]
- Position 0: Uses W1 for pixel 0, W2 for pixel 1, W3 for pixel 2
- Position 1: Uses W1 for pixel 1, W2 for pixel 2, W3 for pixel 3
- Position 2: Uses W1 for pixel 2, W2 for pixel 3, W3 for pixel 4
- And so on...

The SAME three weights slide across the entire input!

The Magic: Parameters Don't Depend on Input Size!

For a 2D image with 3×3 filters:

- 100×100 image: 9 parameters per filter
- 1000×1000 image: STILL 9 parameters per filter
- 10,000×10,000 image: STILL JUST 9 parameters per filter!

This is what the professor means by ”doesn't depend on that actually”!

Total parameters = (filter size) × (input channels) × (output channels) NOT multiplied by image dimensions!

Complete Example:

- Input: 1000×1000×3 (RGB image)
- Layer 1: 64 filters of size 3×3
- Parameters: 3×3×3×64 = 1,728 parameters
- Compare to fully connected: 3,000,000×64 = 192 million parameters
- Reduction factor: 111,111× fewer parameters!

(b) Analyze the trade-off the professor discussed: ”if you restrict connectivity...a neuron receives information just from a restricted part of the input” versus the solution of increasing depth so ”neurons in the following layers...will have the chance to integrate information across the whole input.” (8 marks)

Answer: The Fundamental Trade-off:

When we restrict connectivity, we create a limitation: each neuron has ”tunnel vision” - it can only see a tiny part of the image.

The Problem with Restricted Connectivity:

Layer 1 Neuron Limitations:

- Can only see 3×3 pixels
- Cannot detect patterns larger than 3×3
- Cannot understand relationships between distant image regions

- Like looking at an elephant through a straw!

Example: To recognize a face, you need to see eyes, nose, mouth, and their spatial relationships. A 3×3 receptive field can't even see one complete eye!

The Solution: Depth (Stacking Layers)

The professor's insight: Use multiple layers to gradually increase receptive field!

How Receptive Fields Grow:

- Layer 1: Each neuron sees 3×3 pixels
- Layer 2: Each neuron sees 3×3 neurons from Layer 1
- But each Layer 1 neuron already sees 3×3 pixels
- So Layer 2 effectively sees 5×5 pixels
- Layer 3: Effectively sees 7×7 pixels
- Layer 4: Effectively sees 9×9 pixels
- And so on...

Formula: Effective receptive field = 1 + (k-1) × L Where k = kernel size, L = layer number

Information Integration Process:

Layer-by-Layer Integration:

- **Early layers:** Detect local features (edges, colors, textures)
- Small receptive fields are GOOD here - we want precise localization
- **Middle layers:** Combine local features into parts (eyes, wheels, corners)
- Medium receptive fields integrate local patterns
- **Deep layers:** Recognize complete objects (faces, cars, buildings)
- Large receptive fields see enough context for recognition

The Beautiful Balance:

This design achieves multiple goals simultaneously:

1. **Efficiency:** Few parameters per layer (due to small filters) 2. **Expressiveness:** Can learn complex hierarchical features 3. **Global Understanding:** Deep layers see the entire image 4. **Local Precision:** Early layers maintain spatial accuracy

It's like building understanding: you read letters → words → sentences → meaning!

Practical Implications:

- Shallow networks with large filters: Many parameters, less expressive
- Deep networks with small filters: Fewer parameters, more expressive
- This is why modern networks (ResNet, EfficientNet) are deep, not wide
- VGGNet proved this: only 3×3 filters, but 16-19 layers deep

(c) The professor showed that parameter sharing provides "equivariance to translation...if you shifted the input...the pattern will shift to the next receptive field...accordingly the activations will shift as well." Explain this mathematical property and why it doesn't extend to scale and rotation. (4 marks)

Answer: Translation Equivariance - Mathematical Definition:

A function f is equivariant to transformation T if:

$$f(T(x)) = T(f(x))$$

For translation by vector $\vec{v}$:

$$f(\text{translate}(x, \vec{v})) = \text{translate}(f(x), \vec{v})$$

In simple terms: "If you move the input, the output moves the same way"

Why Convolution IS Translation Equivariant:

Example with 1D convolution:

- Input: [0, 0, 1, 2, 3, 0, 0]

- Filter: [1, -1] (edge detector)
- Output: [0, 1, 1, 1, -3, 0]
- Now shift input right by 2: [0, 0, 0, 0, 1, 2, 3]
- New output: [0, 0, 0, 1, 1, 1, -3]
- The output pattern shifted right by 2 as well!

This works because the SAME filter slides across all positions.

Why Convolution is NOT Scale Equivariant:

- 3×3 filter detects edges at specific scale
- If you zoom in 2×, edges become 2× thicker
- Same 3×3 filter now sees different patterns
- A vertical edge might now look like a gradient to the small filter
- Output changes completely, not just scaled

Solution: Multi-scale architectures (pyramid networks) process multiple scales

Why Convolution is NOT Rotation Equivariant:

- Filter learned to detect vertical edges: [[1, 0, -1], [1, 0, -1], [1, 0, -1]]
- Rotate image 90°: vertical edges become horizontal
- Same filter now produces zero response!
- Would need different filter [[1, 1, 1], [0, 0, 0], [-1, -1, -1]] for horizontal

Solution: Data augmentation (train with rotated images) or specialized architectures

Question 4. Convolution Operation and Hyperparameters (25 marks)

The professor explained: ”We have stride...padding...receptive field size...we need to be careful when we are choosing filter size padding and stride because this needs to be an integer.”

(a) Derive and apply the formula the professor used: ”size of the next layer = (W - F + 2×padding)/stride + 1.” For AlexNet's first layer with input 227×227, filter size 11, stride 4, padding 0, verify the output size of 55×55. (8 marks)

Answer: Deriving the Output Size Formula:

Let me build this formula step by step so it makes complete sense:

Step 1: Without stride or padding

- Input width: W
- Filter width: F
- The filter starts at position 0 and slides right
- Last valid position: when right edge of filter reaches right edge of input
- This happens at position W - F
- Number of positions: (W - F) + 1 (the +1 includes the starting position)

Step 2: Adding padding

- Padding adds P pixels on each side
- Effective input width becomes: W + 2P
- Formula becomes: (W + 2P - F) + 1

Step 3: Adding stride

- Stride S means we skip S-1 positions between applications
- If we have N valid positions, with stride S we only use every Sth position
- Number of actual positions: N/S

- Formula becomes: $\frac{(W+2P-F)}{S}+1$

Final Formula:

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

The floor function $\lfloor\rfloor$ ensures we get an integer (can't have fractional neurons!)

AlexNet First Layer Calculation:

Given:

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

Calculation:

$$\begin{aligned}
\text{Output Size} &= \left\lfloor \frac{227-11+2(0)}{4} \right\rfloor + 1 && (1)\\
&= \left\lfloor \frac{227-11}{4} \right\rfloor + 1 && (2)\\
&= \left\lfloor \frac{216}{4} \right\rfloor + 1 && (3)\\
&= \lfloor 54 \rfloor + 1 && (4)\\
&= 54 + 1 && (5)\\
&= 55 && (6)
\end{aligned}$$

Verified: Output is 55×55 as expected!

Why Integer Constraint Matters:

If we had used input size 225:

$$\frac{225-11+0}{4}+1=\frac{214}{4}+1=53.5+1=54.5 \qquad (7)$$

Can't have 54.5 neurons! This is why the professor emphasized being careful with hyperparameters. Frameworks typically floor the result, but it's better to design for exact integers.

(b) Analyze the professor's AlexNet example where ”stride is four...we are reducing dimensionality by a factor of four that is a huge reduction.” Explain why this large stride works in early layers but would be problematic in later layers, citing his explanation about information redundancy. (10 marks)

Answer: Understanding Stride 4 in AlexNet's First Layer:

The numbers tell the story:

- Input: 227×227 = 51,529 spatial positions
- Output: 55×55 = 3,025 spatial positions
- Reduction: 51,529 / 3,025 = 17× fewer positions!
- This is approximately $4^2 = 16$ (stride affects both dimensions)

Why Large Stride Works in Early Layers:

1. High Spatial Redundancy in Natural Images:

- Adjacent pixels often have similar values (smooth surfaces, gradients)
- A 4×4 pixel region might be almost uniform (sky, wall, skin)
- Sampling every 4th pixel captures essential information without much loss
- Think of it like downsampling audio - works if signal is smooth

2. Computational Efficiency:

- Reduces spatial dimensions early → fewer computations in all subsequent layers
- AlexNet was designed when GPUs had limited memory
- 17× reduction in first layer saves enormous computation throughout network

3. Large Receptive Fields Help:

- 11×11 filter sees substantial context
- Can detect meaningful patterns even when skipping pixels
- Like reading by scanning words, not individual letters

Why Large Stride Fails in Later Layers:

1. Information is Already Compressed:

- Early layers: pixels → edges (high redundancy)
- Later layers: object parts → objects (low redundancy)
- Each spatial position in deep layers represents complex, unique features
- Skipping positions = losing critical information

2. Spatial Resolution Already Low:

- Layer 5 might be 13×13
- Stride 4 would give 3×3 output
- Too coarse to maintain spatial relationships
- Can't tell where objects are located

3. Breaking Feature Hierarchies:

- Deep features have precise spatial arrangements
- ”Eye above nose above mouth” for faces
- Large strides destroy these relationships
- Network loses ability to recognize structured patterns

Modern Best Practices:

- Early layers: Stride 2 is common (more conservative than AlexNet)
- Middle layers: Stride 2 occasionally for dimension reduction
- Late layers: Stride 1 almost always (preserve spatial information)
- Alternative: Use pooling for gradual reduction instead of large strides

The professor's key insight: Information density increases with depth, so aggressive subsampling must happen early or not at all.

(c) The professor discussed the channel dimension: ”when we say a two-dimensional filter actually it might have a third dimension...that spans the channels of its input layer.” For 96 filters of size 11×11×3, calculate the total parameters and explain how ”different filters...learned to extract different types of information.” (7 marks)

Answer: Understanding 3D Filters in CNNs:

Despite being called ”2D convolution,” filters are actually 3D:

- Spatial dimensions: 11×11 (height × width)
- Channel dimension: 3 (must match input channels - RGB)
- Each filter produces ONE output channel
- Need multiple filters for multiple output channels

Parameter Calculation:

For 96 filters of size 11×11×3:

Per Filter:

- Weights: $11 \times 11 \times 3 = 363$ parameters
- Bias: 1 parameter
- Total per filter: 364 parameters

All Filters:

- 96 filters × 364 parameters = 34,944 parameters
- Or breaking it down:
- Weights: $96 \times 11 \times 11 \times 3 = 34{,}848$
- Biases: 96
- Total: 34,944 parameters

How Different Filters Extract Different Information:

Each of the 96 filters learns to detect different patterns:

Color-Specific Detectors:

- Filter 1: Strong response to red channel → detects reddish regions
- Filter 2: Green-blue difference → detects sky vs grass boundaries

- Filter 3: All channels equal → detects grayscale edges

Edge Orientation Detectors:

- Filters 10-20: Vertical edges at different scales
- Filters 21-30: Horizontal edges
- Filters 31-40: Diagonal edges (45°, 135°)

Texture Detectors:

- Filter 50: Checkerboard patterns
- Filter 51: Dots/circles
- Filter 52: Stripes

Complex Pattern Detectors:

- Filter 90: Color blobs (face-like regions)
- Filter 91: Corner intersections
- Filter 92: Center-surround patterns

Visualization Insight:

When researchers visualize learned filters:

- Each filter shows what pattern maximally activates it
- They discover Gabor-like filters (similar to biological vision)
- Filters automatically specialize without explicit programming
- This emergent specialization is what makes deep learning powerful

The professor's point: We don't manually design these 96 filters. Through training, they automatically learn to decompose visual information into useful components, just like the visual cortex!

Question 5. Alternative Convolution Types (28 marks)

The professor introduced multiple convolution variants: ”We have different ways to actually restrict connectivity and share parameters in a layer.”

(a) Compare unshared convolution, dilated convolution, and transposed convolution. For each, explain the professor’s rationale: unshared for problems without equivariance needs, dilated for increasing ”effective coverage...without increasing number of parameters,” and transposed for ”upsampling.” (12 marks)

Answer: 1. Unshared Convolution (Locally Connected Layers):

How it works:

- Like standard convolution: restricted connectivity (small receptive fields)
- Unlike standard convolution: NO parameter sharing
- Each spatial position has its OWN set of weights
- Position (0,0) uses weights $W_{0,0}$, position (0,1) uses different weights $W_{0,1}$

Professor’s Rationale - No Equivariance Needed:

- Use when different positions should be treated differently
- Example: Face recognition with aligned faces
- Eyes always at top, mouth always at bottom
- Want specialized detectors for each facial region
- Eye detector at mouth position would be useless!

Trade-offs:

- Pro: Can learn position-specific patterns
- Pro: More expressive for structured inputs
- Con: Many more parameters (loses sharing benefit)
- Con: Requires aligned/normalized inputs

2. Dilated Convolution (Atrous Convolution):

How it works:

- Standard 3×3 filter: samples at positions (0,0), (0,1), (0,2), (1,0), etc.
- Dilated 3×3 filter (dilation=2): samples at (0,0), (0,2), (0,4), (2,0), etc.
- ”Spreads out” the filter with gaps between samples
- Like poking holes (à trous = ”with holes” in French)

Professor’s Rationale - Increase Coverage Without More Parameters:

- 3×3 filter with dilation 2 → covers 5×5 area
- 3×3 filter with dilation 4 → covers 9×9 area
- Still only 9 parameters regardless of dilation!
- Effective receptive field = (k-1) × dilation + 1

Use cases:

- Semantic segmentation: need wide context while maintaining resolution
- Replace pooling layers: increase receptive field without losing resolution
- Multi-scale processing: stack different dilations for multi-scale features

3. Transposed Convolution (Deconvolution):

How it works:

- Standard convolution: large input → small output (downsampling)
- Transposed convolution: small input → large output (upsampling)
- Not the mathematical inverse! Just transposes the connectivity pattern
- Each input pixel influences multiple output pixels

Professor's Rationale - Upsampling:

- Need to increase spatial resolution (opposite of pooling)
- Example: 2×2 input → 4×4 output
- Learnable upsampling (better than bilinear interpolation)
- Used in generative models, segmentation decoders

Mathematical relationship:

- If conv: $nnmm$ with kernel k, stride s
- Then transposed conv: $mmnn$ with same k, s
- Output size = (input-1) × stride + kernel

Common issue:

- Can create ”checkerboard artifacts” due to uneven overlap
- Solution: Use resize + conv instead of transposed conv

(b) Analyze separable convolution as the professor explained: ”we can write a 3×3 matrix as a multiplication of two vectors...we can reduce the number of parameters...from nine parameters...to six parameters.” Explain depthwise separable convolution and its efficiency benefits. (10 marks)

Answer: Spatial Separable Convolution (Professor's Example):

The professor's insight: Some filters can be decomposed into simpler components.

Example - Separable 3×3 Filter:

- Original filter: $\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$
- Can be written as: $\begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \times \begin{bmatrix} 1 & 2 & 1 \end{bmatrix}$
- Vertical vector (3 params) × Horizontal vector (3 params) = 6 params total
- Original had 9 parameters → 33% reduction

How to apply:

- First: Convolve with vertical vector [1, 2, 1] (1D convolution)
- Then: Convolve with horizontal vector [1, 2, 1] (1D convolution)
- Result identical to original 2D convolution
- Computation: 6 multiplications instead of 9 per position

Limitation: Not all filters are separable! Only works for specific patterns.

Depthwise Separable Convolution (Modern Approach):

Different concept: Separate spatial and channel-wise operations.

Standard Convolution Problem:

- Input: H×W×C$_i n(e.g., 3232128) Filter : KKC_i n C_o ut (e.g., 33128256)$

- Parameters: 3×3×128×256 = 294,912
- Mixes spatial and channel operations together

Depthwise Separable Solution:

Step 1 - Depthwise Convolution:

- Apply one K×K filter per input channel
- 128 input channels → 128 filters of size 3×3
- Parameters: 3×3×128 = 1,152
- Output: H×W×128 (same channels)

Step 2 - Pointwise Convolution (1×1):

- Mix channels using 1×1 convolution
- 128 input → 256 output channels
- Parameters: 1×1×128×256 = 32,768
- Output: H×W×256

Total: 1,152 + 32,768 = 33,920 parameters Reduction: 294,912 / 33,920 = 8.7× fewer parameters!

Efficiency Analysis:

Parameter Reduction:

  - Standard: $K^2 \times C_{in} \times C_{out}$
  - Depthwise Separable: $K^2 \times C_{in} + C_{in} \times C_{out}$
  - Ratio: $\frac{1}{C_{out}} + \frac{1}{K^2}$
- For K=3, C$_{o}ut = 256 :~9reduction$

  Computational Reduction:

  - Similar ratio for FLOPs
  - Enables efficient networks: MobileNet, Xception
  - Critical for mobile/edge deployment

  Key Insight: The professor showed that we can decompose expensive operations into cheaper components without significant accuracy loss. This principle drives modern efficient architectures.

(c) The professor described 1×1 convolution as controlling ”the number of channels...without using any...without combining any information from a neighborhood.” Explain how this operation works and its role in efficient architectures like GoogleNet that will be discussed later. (6 marks)

Answer: Understanding 1×1 Convolution:

At first, ”1×1 convolution” sounds pointless - what can a 1×1 filter do? The key: It operates on the CHANNEL dimension, not spatial dimensions.

How 1×1 Convolution Works:

Operation:

  - Input: H×W×C$_{i}n(e.g., 3232128)11ConvwithC_{o}utfilters : HWC_{o}ut(e.g., 323264)$
- Each output channel is a weighted combination of ALL input channels
- At each spatial position independently

Mathematical View: For each spatial position (i,j):

  - Input vector: [c1, c2, ..., c128] (channel values at that position)
  - Output: Linear transformation (matrix multiply)
  - Like a fully connected layer applied at each position!

No Neighborhood Information: As the professor emphasized:

- Only looks at one spatial position at a time
- Cannot detect edges, textures, or spatial patterns
- Purely channel-wise operation

Three Critical Roles in Efficient Architectures:

1. Channel Dimension Reduction (Compression):

- Before expensive operations: 256 channels → 64 channels
- Apply expensive 5×5 convolution on 64 channels (not 256)
- Reduces computation by 4×
- GoogleNet's "bottleneck" design

2. Channel Dimension Expansion:

- After feature extraction: 64 channels → 256 channels
- Increases network capacity without spatial computation
- Cheap way to add expressiveness

3. Cross-Channel Information Mixing:

- Combines information across channels
- Learns correlations: "red + circular = apple"
- Essential after depthwise convolution (which doesn't mix channels)

GoogleNet (Inception) Example:

Inception module uses 1×1 convolutions brilliantly:

- Input: 256 channels
- Branch 1: 1×1 conv → 64 channels → 3×3 conv
- Branch 2: 1×1 conv → 64 channels → 5×5 conv
- Branch 3: 1×1 conv → 128 channels (direct)
- Without 1×1: 256×3×3 + 256×5×5 = 8,704 params
- With 1×1: 256×64 + 64×3×3 + 256×64 + 64×5×5 = 35,392 params

- But processes 4× fewer channels in expensive convolutions!

The professor's insight: 1×1 convolutions provide a cheap way to manipulate the channel dimension, enabling more efficient and deeper networks.

Question 6. Deformable Convolution and Advanced Concepts (30 marks)

The professor introduced deformable convolution: ”What if we learn directly from the data...a better positioning of the filter...for each position what if I estimate an offset along X and Y that would be more meaningful.”

(a) Explain the concept of deformable convolution as ”dynamic receptive field.” Describe how ”for each parameter...we need to have two offsets for X and Y delta X and delta Y” and why this makes the operation ”very expensive.” (12 marks)

Answer: The Limitation of Standard Convolution:

Standard convolution uses a rigid grid:

- 3×3 filter always samples at: (-1,-1), (-1,0), (-1,1), (0,-1), (0,0), (0,1), (1,-1), (1,0), (1,1)
- This fixed pattern applied everywhere, regardless of image content
- Like using a rigid stamp - can’t adapt to the actual shapes in the image

Deformable Convolution - The Core Idea:

Let the network learn WHERE to look, not just WHAT to look for!

Dynamic Receptive Fields: Instead of fixed positions, use learned offsets:

- Original position: (x, y)
- Learned offset: (Δx, Δy)
- Sample at: (x + Δx, y + Δy)
- Different offsets for each spatial location!
- Receptive field adapts to image content

The Offset Learning Mechanism:

For a 3×3 deformable convolution:

Standard Conv:

- 9 weight parameters (one per position)

- Fixed sampling positions

Deformable Conv:

- 9 weight parameters (unchanged)
- PLUS 18 offset parameters:
- 2 offsets (Δx, Δy) for each of 9 positions
- Offsets are LEARNED from data
- Different offsets at each spatial location

Architecture:

- Input → Offset prediction branch (conv layer)
- Outputs 18 channels (2 × 9 offsets)
- These offsets deform the sampling grid
- Main convolution uses deformed positions

Why It's "Very Expensive":

1. Additional Network Branch:

- Need separate convolution to predict offsets
- Adds parameters and computation
- Must run before main convolution

2. Irregular Memory Access:

- Standard conv: predictable memory pattern (GPU-friendly)
- Deformable: random access based on learned offsets
- Breaks cache locality
- Can't use optimized convolution implementations

3. Bilinear Interpolation:

- Offsets are real numbers, not integers
- (x + 0.7, y + 1.3) doesn't land on pixel grid
- Must interpolate between 4 nearest pixels

- Adds computation for every sample

4. Gradient Computation:

- Backprop through both weights AND offsets
- Gradients flow through bilinear interpolation
- More complex computational graph
- 3-4× slower than standard convolution

Concrete Example: Standard 3×3 conv on 256 channels:

- Operations: $9 \times 256 = 2{,}304$ per position

Deformable 3×3 conv:

- Offset prediction: 2,304 operations
- Bilinear interpolation: $4 \times 9 \times 256 = 9{,}216$ operations
- Main convolution: 2,304 operations
- Total: 13,824 operations (6× more!)

(b) The professor showed the benefit: ”receptive fields are adjusted automatically...in such a way that receptive field actually pays attention to the most relevant content.” Analyze the sheep example and explain why this improves upon vanilla convolution’s ”rigid...regular” placement. (8 marks)

Answer: The Sheep Example - Visualizing Adaptive Receptive Fields:

The professor’s sheep example perfectly illustrates the power of deformable convolution:

Standard Convolution on a Sheep:

- 3×3 grid samples fixed positions
- Near sheep’s body: some points hit the sheep, others hit background
- Near legs: grid might sample ground, leg, and sky simultaneously
- The rigid grid doesn’t respect object boundaries
- Mixes relevant (sheep) and irrelevant (background) information

Deformable Convolution on a Sheep:

- Sampling points ADAPT to follow sheep's contour
- Near body: points cluster on the woolly texture
- Near legs: points align along the thin leg structure
- Near head: points focus on facial features
- Avoids sampling background pixels

Why This Improves Recognition:

1. Feature Purity:

- Standard conv: ”50% sheep, 50% grass” → confused features
- Deformable: ”100% sheep” → clean features
- Better signal-to-noise ratio
- More discriminative representations

2. Shape Adaptation:

- Objects have irregular shapes
- Rectangular grids don't match natural boundaries
- Deformable conv learns object-aware sampling
- Can follow curves, handle thin structures

3. Scale Adaptation:

- Small objects: offsets bring samples closer
- Large objects: offsets spread samples wider
- Automatic scale handling without multiple filter sizes

4. Context-Aware Processing:

- Different patterns need different sampling strategies
- Texture: dense, regular sampling
- Edges: aligned sampling along boundaries
- Network learns optimal strategy per location

Concrete Benefits in Practice:

Object Detection:

- Better bounding box predictions
- Handles occlusion better (focuses on visible parts)
- Improves small object detection

Semantic Segmentation:

- More accurate boundaries
- Better handling of thin structures (poles, legs)
- Reduced bleeding across object boundaries

The professor's key insight: By making receptive fields adaptive, we let the network focus on what matters, dramatically improving feature quality.

(c) Solve the backpropagation challenge the professor discussed: ”indices don't enter the calculations...we cannot do that because it is not part of the calculation.” Explain how bilinear interpolation makes ”everything differentiable” by incorporating offsets into the computation. (10 marks)

Answer: The Fundamental Problem - Non-Differentiable Sampling:

Integer Indexing Problem: Imagine trying to sample at learned positions:

- Offset prediction: (Δx, Δy) = (1.7, 2.3)
- Round to integers: (2, 2)
- Sample: value = image[x+2, y+2]

Why This Breaks Backpropagation:

- The rounding operation has zero gradient everywhere
- $\frac{\partial \text{round}(1.7)}{\partial \text{input}} = 0$
- Offset (1.7, 2.3) and (1.9, 2.4) both round to (2, 2)

- No gradient signal to improve offset predictions!
- The professor's point: ”indices don't enter calculations”

The Solution - Bilinear Interpolation:

Instead of rounding, use the fractional parts to blend nearby pixels!

How Bilinear Interpolation Works: For offset (Δx, Δy) = (1.7, 2.3):

- Don't round! Keep fractional parts
- Find 4 nearest pixels: (1,2), (2,2), (1,3), (2,3)
- Calculate weights based on distance
- Blend all 4 values

The Mathematics: Let p = (x + Δx, y + Δy) be the target position

- Top-left pixel: (p_x, p_y), weight $= (1 - \{p_x\})(1 - \{p_y\})$
- Top-right pixel: $(p_x + 1, p_y)$, weight $= \{p_x\}(1 - \{p_y\})$
- Bottom-left pixel: $(p_x, p_y + 1)$, weight $= (1 - \{p_x\})\{p_y\}$
- Bottom-right pixel: $(p_x + 1, p_y + 1)$, weight $= \{p_x\}\{p_y\}$

Where $\{a\}$ means fractional part of a

Output = $\Sigma(\text{pixel}_v alueweight)$

Why This Is Differentiable:

Key Insight: Offsets appear in the weight calculations!

Example with (1.7, 2.3):

- Fractional parts: fx = 0.7, fy = 0.3
- Weight for pixel (2,2) = fx × (1-fy) = 0.7 × 0.7 = 0.49
- If Δx increases by : fx becomes 0.7+
- New weight = (0.7+) × 0.7 = 0.49 + 0.7
- Gradient: $\frac{\partial \text{weight}}{\partial \Delta x} = 0.7$

Complete Gradient Flow:

- Loss → Output activations