Lecture 7:

Deep Neural Networks and Architecture Patterns

Ho-min Park

homin.park@ghent.ac.kr

powersimmani@gmail.com

Lecture Contents

Part 1/3:

The Need for Deep Neural Networks

1. Limitations of Shallow Networks
2. The Power of Depth - Hierarchical Representations
3. Feature Reuse and Composition
4. Parameter Efficiency
5. Challenges of Deep Networks
6. Vanishing Gradient Problem
7. Exploding Gradient Problem
8. Overview of Solutions

Limitations of Shallow Networks

UNIVERSAL APPROXIMATION THEOREM

Single hidden layer networks can theoretically approximate any function
BUT exponentially many hidden units may be required

↓

No Hierarchical Patterns 1

Cannot efficiently capture hierarchical structure in images and text

Compositional Struggle 2

Must learn all combinations from scratch without reusing patterns

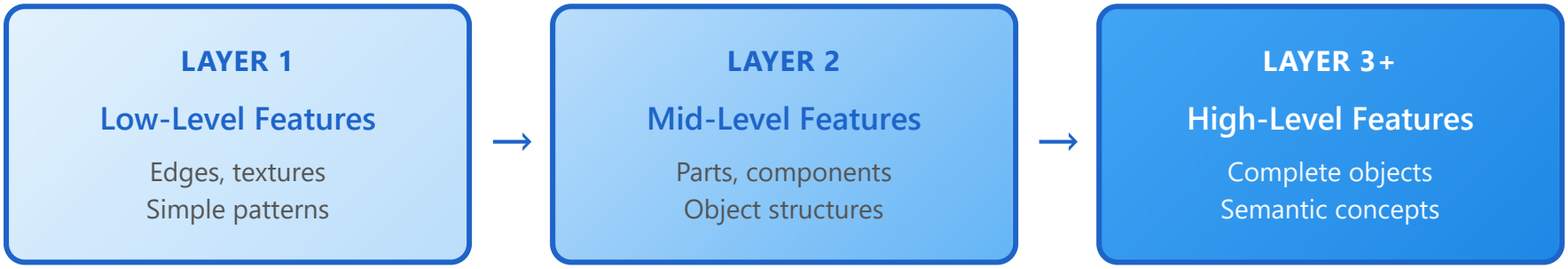
Poor Generalization 3

Weak performance on high-dimensional data with limited samples

Limited Abstraction 4

Manual feature engineering still required for complex tasks

The Power of Depth: Hierarchical Representations

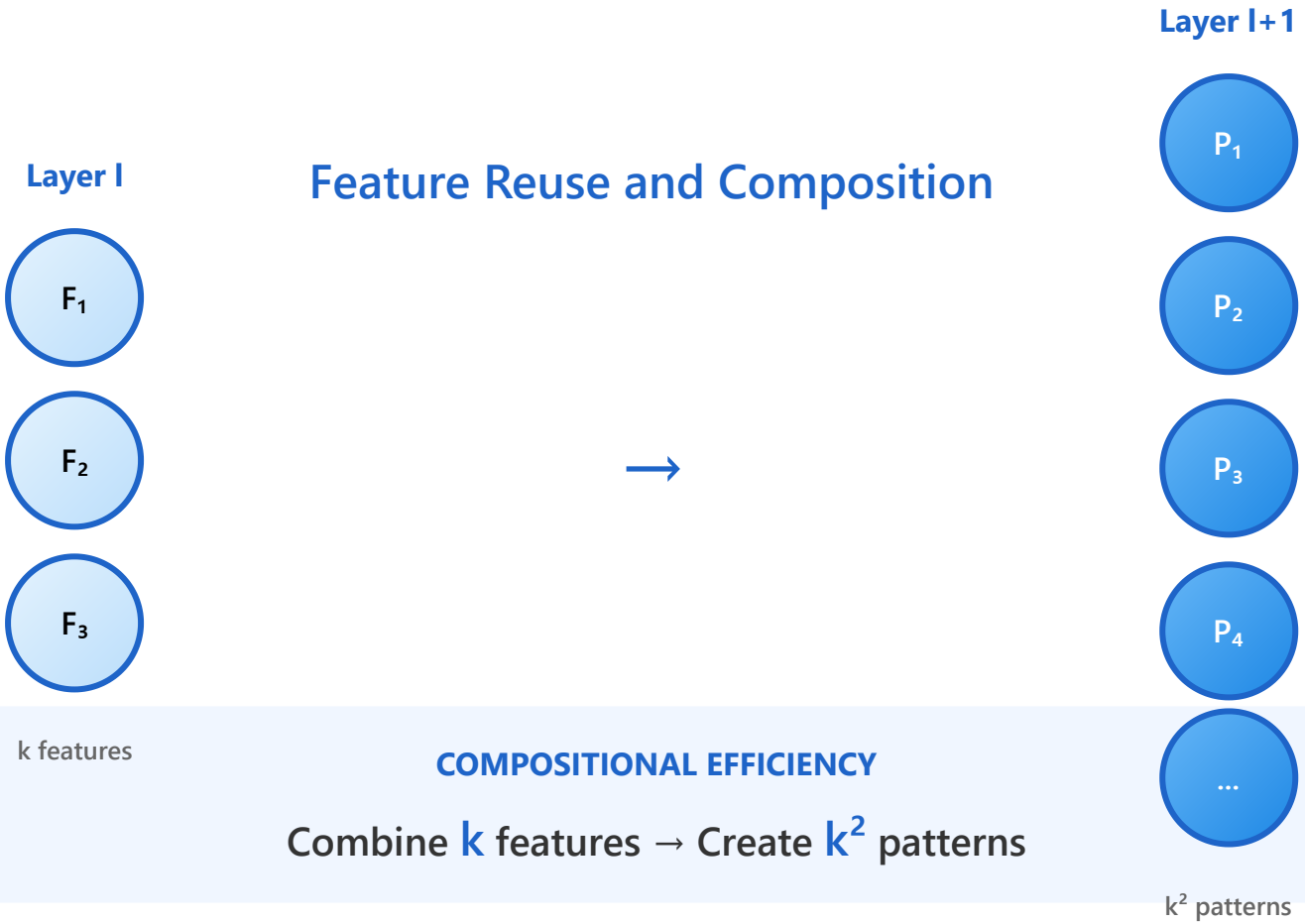


Human Visual Cortex Analogy

V1 → V2 → V4 → IT

- ✓ Transforms representation space to be more linearly separable
- ✓ Enables transfer learning: lower layers generalize across tasks

Feature Reuse and Composition



COMPOSITIONAL EFFICIENCY

Combine k features → Create k^2 patterns

Reusable Components

Exponential Expressiveness

Modular Learning

EXAMPLE

"Eye" feature reused for detecting different faces, animals → Reduces redundancy

Parameter Efficiency: Deep vs. Shallow Networks

Shallow Network

1000 units

Layers:	1
Total Units:	1,000
Parameters:	1M+

Deep Network

100 units

↓

100 units

↓

100 units

Layers:	3
Total Units:	300
Parameters:	30K

Computational Complexity (n inputs)

Shallow: $O(2^n)$

vs

Deep: $O(n^2)$

- ✓ Better generalization
- ✓ Faster training
- ✓ Reduced redundancy
- ✓ Faster inference

Challenges of Deep Networks

Training Complexity

- Non-convex optimization
- Longer training time
- High computational cost

Gradient Problems

- Vanishing gradients in early layers
- Exploding gradients
- Unstable parameter updates

Generalization Issues

- Overfitting risk
- Memorizing training data
- Poor test performance

Configuration Sensitivity

- Critical initialization requirements
- Hyperparameter sensitivity
- Architecture design choices

⚠ HISTORICAL BARRIER (Pre-2006)

Deep networks often performed worse than shallow ones due to these challenges

These challenges motivated research into better training techniques and architectures

Vanishing Gradient Problem

MATHEMATICAL CAUSE

$$\text{gradient} \propto \prod(\partial a_l / \partial z_l) \rightarrow \text{Product of many terms} < 1$$

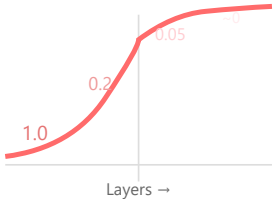
⚠ Consequences

- Early layers learn extremely slowly
- Weights barely change
- Training loss plateaus
- Network behaves like shallow

ReLU Activation	Skip Connections	Careful Init	Batch Norm

Activation Function Comparison

Sigmoid (σ) ❌ Gradient Vanishing



$\sigma'(x) = \sigma(x)(1-\sigma(x)) \leq 0.25$

ReLU ✅ Gradient Preserved

$ReLU'(x) = 1 \ (if \ x > 0)$

📊 Activation Functions & Gradient Vanishing Risk

⚠️ High Risk

Sigmoid (σ)

σ'(z) = σ(z)(1-σ(z))

Max gradient: 0.25

$0.25^5 = 0.00098$ ❌

⚠️ Medium Risk

Tanh

$\tanh'(z) = 1 - \tanh^2(z)$

Max gradient: 1.0

Saturates at extremes → vanish

✅ Lower Risk

ReLU

ReLU'(z) = 1 if z>0 else 0

Gradient: 0 or 1

No saturation for z>0 ✓

🔗 Chain Rule Multiplication: How Gradients Vanish

Backpropagation through layers:

$$\partial L/\partial W_1 = \partial L/\partial a_4 \times \mathbf{\partial a_4 /\partial z_4} \times \partial z_4 /\partial a_3 \times \mathbf{\partial a_3 /\partial z_3} \times \partial z_3 /\partial a_2 \times \mathbf{\partial a_2 /\partial z_2} \times \partial z_2 /\partial a_1 \times \mathbf{\partial a_1 /\partial z_1} \times \partial z_1 /\partial W_1$$

Blue terms: activation derivatives (σ', tanh', ReLU')

Example with Sigmoid activation:

❌ Bad Case (Sigmoid)

```
Layer 1: grad × W₁  × σ'
= 1.0 × 1.0 × 0.25 = 0.25
Layer 2: 0.25 × 1.0 × 0.25 = 0.0625
Layer 3: 0.0625 × 1.0 × 0.25 = 0.0156
Layer 4: 0.0156 × 1.0 × 0.25 ≈ 0.004
```

✅ Good Case (ReLU)

```
Layer 1: 1.0 × 1.0 × 1.0 = 1.0
Layer 2: 1.0 × 1.0 × 1.0 = 1.0
Layer 3: 1.0 × 1.0 × 1.0 = 1.0
Layer 4: 1.0 × 1.0 × 1.0 = 1.0
```

10 layers: $0.25^{10} \approx 0.00000095$
Practically zero! 📉

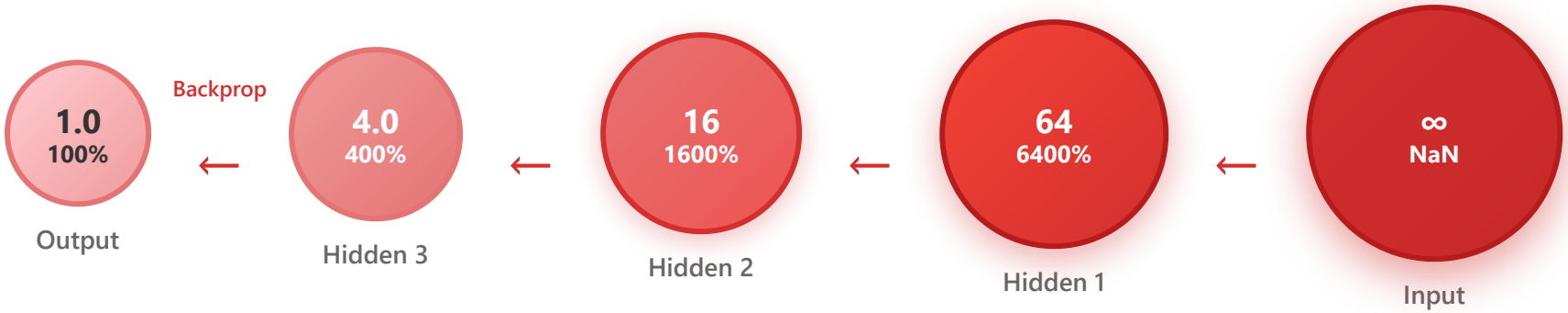
Gradient preserved!
Can train very deep networks ✓

📉 Real Vanishing Scenario

Sigmoid Max Gradient		10 Layers		Gradient Scale
$\sigma' = 0.25$	×	$n = 10$	→	$0.25^{10} \approx 0.000001$

With 20 layers: $0.25^{20} \approx 9.09 \times 10^{-13}$ → Effectively zero! ❄️

Exploding Gradient Problem



MATHEMATICAL CAUSE

$$\text{gradient} \propto \prod(\partial a_l / \partial z_l) \rightarrow \text{Product of many terms} > 1$$

Consequences

- Unstable parameter updates
- Loss becomes NaN or infinity
- Training loss spikes dramatically
- Weights become very large

✂ Gradient Clipping	🎯 Xavier/He Init	📊 Batch Normalization

📈 Activation Functions & Gradient Explosion Risk

⚠ High Risk

Sigmoid (σ)

$\sigma'(z) = \sigma(z)(1-\sigma(z))$

Max gradient: 0.25

⚠ Medium Risk

Tanh

$\tanh'(z) = 1 - \tanh^2(z)$

Max gradient: 1.0

✅ Lower Risk

ReLU

$\text{ReLU}'(z) = 1 \text{ if } z>0 \text{ else } 0$

Gradient: 0 or 1

$0.25^5 = 0.00098$ ❌

Poor weight init → explode

But large W → still explode

🔢 Chain Rule Multiplication: How Gradients Explode

Backpropagation through layers:

$$\partial L/\partial W_1 = \partial L/\partial a_4 \times \mathbf{\partial a_4/\partial z_4} \times \partial z_4/\partial a_3 \times \mathbf{\partial a_3/\partial z_3} \times \partial z_3/\partial a_2 \times \mathbf{\partial a_2/\partial z_2} \times \partial z_2/\partial a_1 \times \mathbf{\partial a_1/\partial z_1} \times \partial z_1/\partial W_1$$

Red terms: activation derivatives (σ', tanh', ReLU')

Example with poorly initialized weights:

❌ Bad Case (W ~ 2.0)

```
Layer 1: grad × W₁  × σ'
= 1.0 × 2.0 × 0.25 = 0.5
Layer 2: 0.5 × 2.0 × 0.25 = 0.25
Layer 3: 0.25 × 2.0 × 0.25 = 0.125
Layer 4: 0.125 × 2.0 × 0.25 = 0.0625
```

If W=3: 3×0.25=0.75 → vanish!
If W=5: 5×0.25=1.25 → explode! 💥

✅ Good Case (Xavier Init)

```
W ~ N(0, 1/√n)
Keeps gradient variance stable

E[∂L/∂W] ≈ constant
No explosion or vanishing
```

💥 Real Explosion Scenario

Initial Weight		5 Layers		Gradient Scale
W = 2.5	×	**n = 5**	→	**$2.5^5 = 97.7$**

With 20 layers: $2.5^{20} = 9.09 \times 10^8$ → NaN! 🔥

Solutions to Deep Learning Challenges

Better Activation Functions

Prevents vanishing gradients during backpropagation

ReLU, Leaky ReLU, ELU, GELU

→ Vanishing Gradient

Smart Initialization

Maintains gradient scale across layers

Xavier, He Initialization

→ Gradient Issues

Normalization Techniques

Stabilizes training and speeds up convergence

Batch Norm, Layer Norm

→ Training Stability

Skip Connections

Creates gradient highways through the network

ResNet, Highway Networks

→ Vanishing Gradient

Careful Optimization

Adaptive learning rates for stable updates

Adam, RMSprop, AdaGrad

→ Training Efficiency

Regularization & Clipping

Prevents overfitting and gradient explosion

Dropout, Weight Decay, Grad Clipping

→ Overfitting & Explosion

Modern Deep Learning

Robust training is achieved by combining multiple techniques together

Part 2/3:

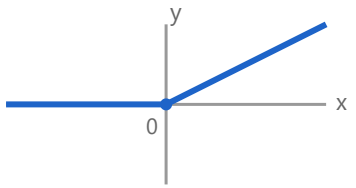
Modern Activation Functions

9. The ReLU Revolution
10. Leaky ReLU, PReLU
11. ELU, SELU
12. Swish, GELU
13. Activation Function Selection Guide
14. Dead ReLU Problem
15. Gradient Flow Analysis
16. Layer-wise Activation Patterns

The ReLU Revolution

DEFINITION

$$\mathrm{ReLU}(x) = \max(0, x)$$



HISTORICAL IMPACT

Enabled training of **AlexNet (2012)**

First deep CNN to win **ImageNet**

→ Sparked the deep learning revolution

✓ Prevents vanishing gradients (derivative = 1 for x > 0)

✓ Computational efficiency (no exponential calculations)

✓ Sparse activation (~50% neurons output zero)

✓ Unchanged gradient flow through active neurons

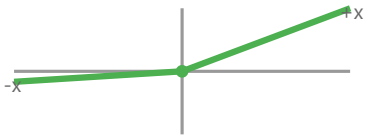
LIMITATIONS: Dead ReLU problem · Non-zero centered · Unbounded output

Leaky ReLU & PReLU: Addressing Dead Neurons

Leaky ReLU

$$f(x) = \max(\alpha x, x)$$

α = 0.01 (fixed)

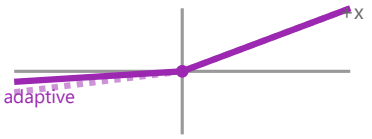


- ✓ Prevents dead neurons
- ✓ Small negative slope (0.01)
- ✓ Minimal computational cost

PReLU

$$f(x) = \max(\alpha x, x)$$

α learned during training



- ✓ Learnable parameter α
- ✓ Adapts per channel/layer
- ✓ Better in deep networks

Performance Comparison

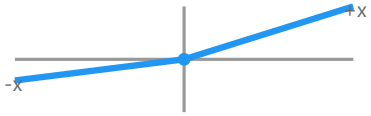
Aspect	Leaky ReLU	PReLU
Flexibility	Fixed α	Adaptive α
Complexity	Minimal	Slight increase
Use Case	General purpose	Deep CNNs

ELU & SELU: Advanced Activation Functions

ELU

Exponential Linear Unit

$$f(x) = x \text{ if } x > 0$$
$$f(x) = \alpha(e^x - 1) \text{ if } x \leq 0$$

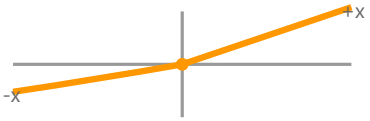


- ✓ Smooth exponential curve
- ✓ Negative values → zero mean
- ✓ Reduces bias shift
- ✓ More robust to noise

SELU

Scaled Exponential Linear Unit

$$f(x) = \lambda \times ELU(x)$$
$$\lambda \approx 1.05, \alpha \approx 1.67$$



- ✓ Self-normalizing properties
- ✓ Maintains mean/variance
- ✓ No Batch Norm needed

SPECIAL PROPERTY

Auto-normalization without Batch Normalization

Usage Comparison

Aspect	ELU	SELU
Computation	Exponential (moderate)	Exponential (moderate)
Best Use	General improvements	Fully-connected networks
Key Benefit	Faster learning vs ReLU	Self-normalization

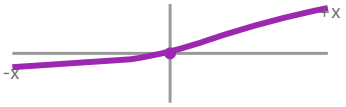
Swish & GELU: Modern State-of-the-Art Activations

Swish

Self-Gating Activation

$$f(x) = x \times \sigma(\beta x)$$

$$\beta = 1 \text{ (Swish-1)}$$



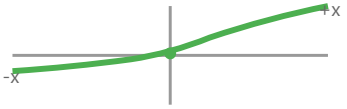
- ✓ Self-gating mechanism
- ✓ Smooth, non-monotonic
- ✓ Neural architecture search

GELU

Gaussian Error Linear Unit

$$f(x) = x \times \Phi(x)$$

$$\Phi = \text{Gaussian CDF}$$



- ✓ Probabilistic interpretation
- ✓ Smooth, non-monotonic
- ✓ State-of-the-art in NLP

ADOPTION

BERT, GPT (Default)

Key Properties

- Smooth everywhere
- Outperform ReLU in Transformers
- Higher computational cost

Best Use Cases

- 🎯 Transformer models
- 🎯 NLP tasks (BERT, GPT)
- 🎯 State-of-the-art performance

Activation Function Selection Guide

DEFAULT CHOICE

Start with ReLU

Fast, works well for most cases

Deep CNNs →

Leaky ReLU / PReLU

Prevents dead neurons in deep networks

Faster Convergence →

ELU

Slightly slower but better accuracy

Fully-Connected Deep Networks →

SELU

Eliminates need for Batch Normalization

Transformers / NLP →

GELU

Standard in BERT, GPT models

Limited Compute →

ReLU / Leaky ReLU

Fastest computational speed

Experimental / Research →

Swish / GELU

Potential accuracy gains

General rule: Modern smooth activations **(GELU, Swish)** often worth the extra computational cost

Dead ReLU Problem

DEFINITION

Neurons that always output zero for all inputs (permanently inactive)

CAUSE

Large negative gradients push weights into negative region

→

CONSEQUENCE

Neuron stops learning, effectively removed

→

IMPACT

Reduced capacity, wasted parameters

⚠ Symptoms & Detection

Significant portion of neurons inactive **(sometimes >40%)** · Monitor percentage of always-zero activations during training

✓ Solutions

- ⚡ Leaky ReLU / PReLU
- 🎯 Lower Learning Rate
- 🔧 Better Initialization
- ✂ Gradient Clipping

Gradient Flow Analysis

GOAL

Understand how gradients propagate through different activations

ReLU

⚠ Risky

Binary gradient (0 or 1)

Can block gradients completely when inactive

Sigmoid / Tanh

✗ Poor

Max derivative: 0.25

Gradients shrink exponentially

Leaky ReLU

✓ Good

Always allows gradient

Small slope in negative region

Smooth (ELU, GELU, Swish)

★ Best

Continuous gradients

Better for optimization

🎯 **Ideal Property**

Gradients neither vanish nor explode across depth

Modern Approach

Good Activations + **Normalization** + **Skip Connections**

Layer-wise Activation Patterns

KEY OBSERVATION

Different layers may benefit from different activations

Early Layers

ReLU

Low-level features

→

Middle Layers

ELU / GELU

Complex patterns

→

Output Layer

Task-Specific

Sigmoid / Softmax

Input → Features

Feature Processing

Predictions

Empirical Finding

Uniform activation often works well, mixed can be better

Research Direction

Neural Architecture Search explores per-layer choices

Practical Tip

If experimenting, start from output and work backward

Implementation

Start simple (uniform), optimize if needed

Transformer Pattern Example

GELU in feedforward layers · **Linear (no activation)** in attention mechanisms

Part 3/3:

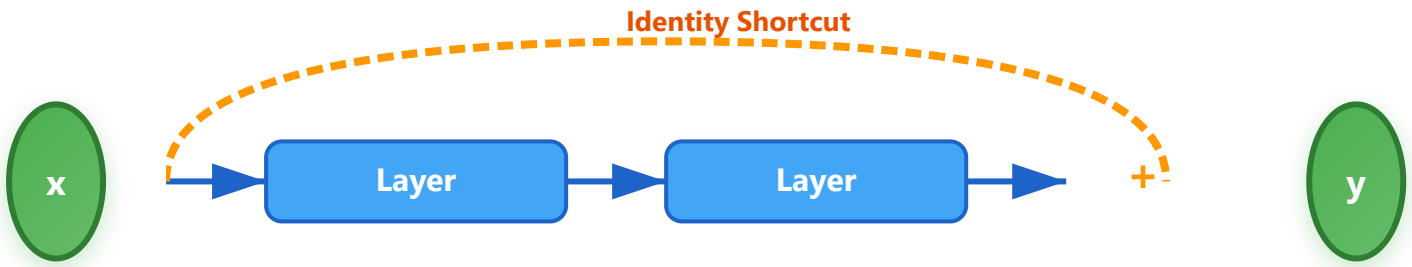
Advanced Architecture Patterns

17. Skip Connection (ResNet)
18. Dense Connection (DenseNet)
19. Bottleneck Architecture
20. The Role of 1x1 Convolution
21. Inception Module
22. Depthwise Separable Convolution
23. Neural Architecture Search
24. Model Compression Techniques
25. Practical Design Guidelines

Skip Connection (ResNet): Breakthrough Architecture

CORE FORMULA

$$y = F(x) + x$$



Extreme Depth
Enabled training 152+ layer networks

Gradient Highway
Direct path for gradients to early layers

Residual Learning
Learns F(x) instead of full H(x)

Math Benefit
Gradient always has component of 1

Dense Connection (DenseNet): Maximum Connectivity

CORE IDEA

Each layer connects to ALL subsequent layers

$$x_l = H_l([x_0, x_1, \ldots, x_{l-1}])$$

Dense Connectivity Pattern

✓ Advantages

Maximum information flow · Feature reuse · Parameter efficiency

📊 Efficiency

Fewer parameters than ResNet for same accuracy

⚠ Memory Consideration

Requires storing all intermediate features (higher memory)

🎯 Use Cases

Image classification · Feature extraction · Limited parameters

GROWTH RATE

Each layer adds **k feature maps** (typically k=12 or k=32)

Bottleneck Architecture: Efficient Design

PURPOSE

Reduce computational cost while maintaining expressiveness

Bottleneck Structure

DIMENSIONAL FLOW EXAMPLE

256 channels → 64 channels → 64 channels → 256 channels

Computational Savings

~70% fewer FLOPs

Trade-off

More layers, fewer ops

Used In

ResNet, Inception, etc.

Modern Standard: Almost all efficient architectures use bottlenecks

The Role of 1×1 Convolution

PRIMARY FUNCTIONS

Dimensionality control · Channel mixing · Adding non-linearity

Channel Reduction

Projects high-dimensional features to lower dimensions

Channel Expansion

Increases feature map channels without spatial processing

Cross-Channel Mixing

Learns relationships between different channels

Example: Dimensionality Reduction

256

Input Channels

→

1×1 Conv

64

Output Channels

✓

Efficient

Much cheaper than 3×3 or 5×5

Computational efficiency vs larger kernels

Network-in-Network: Adds depth without overhead

Universal Use: Inception · ResNet · MobileNet · Transformers (feedforward)

Inception Module: Multi-Scale Architecture

CORE IDEA

Multiple parallel conv paths with different receptive fields

Inception Structure

Input Feature Map

1×1 reduce	1×1 reduce	1×1 reduce	Max pool
1×1 conv	3×3 conv	5×5 conv	1×1 conv

Concatenation (Filter Concat)

Multi-Scale

Captures features at different scales

Efficiency

1×1 convs reduce dimensions first

Diversity

Rich feature diversity

GoogLeNet (v1)	Inception v2/v3	Inception-ResNet	Modern Influence
ImageNet 2014	**Factorized Convs**	**Skip Connections**	**EfficientNet, NAS**

Depthwise Separable Convolution

KEY INSIGHT

Spatial and channel-wise processing can be separated

Standard Convolution

Single Operation
All Channels Together

`Params: D² × M × N`
`(expensive)`

Depthwise Separable

Depthwise Conv
Each Channel Separate

↓

Pointwise (1×1)
Combine Channels

`Params: D² × M + M × N`
`(efficient)`

Efficiency Gains

Computational Savings

8-9× fewer ops

Parameter Reduction

Massive reduction

Used In

MobileNet, EfficientNet, Xception

Trade-off

Slightly lower accuracy, massive efficiency

Neural Architecture Search (NAS)

GOAL

Automatically discover optimal network architectures using ML

NAS Process

Search Space

Operations · Connections · Layer counts · Kernel sizes · Channel counts

Search Strategies

Reinforcement learning · Evolutionary algorithms · Gradient-based

Success Stories

NASNet · EfficientNet · AmoebaNet - State-of-the-art results

Computational Cost

Originally thousands of GPU-hours · Now more efficient methods

Future Direction: Architecture search becoming standard practice for deployment optimization

Model Compression Techniques

Pruning

Remove unimportant weights/neurons

Structured or unstructured

Quantization

Reduce precision (FP32 → INT8)

~4× memory/speed improvement

Knowledge Distillation

Train small student to mimic large teacher

Transfer knowledge efficiently

Low-Rank Factorization

Decompose weight matrices into smaller components

Reduce parameter count

Weight Sharing

Group weights to reduce unique parameters

Efficient representation

Typical Compression Results

Compression Ratio	Accuracy Loss
5-10×	**<1%**

Mobile Deployment

Essential for phones, IoT devices

Tools

TensorFlow Lite · PyTorch Mobile · ONNX Runtime

Practical Design Guidelines

START SIMPLE

Begin with proven architectures (ResNet, EfficientNet), modify as needed

Architecture Choices

- Depth vs. width: Deeper generally better
- Residual connections for >10 layers

Key Components

- Batch norm: After conv, before activation
- Activation: ReLU/GELU (avoid sigmoid/tanh)

Regularization Stack

- Data augmentation
- Dropout
- Weight decay

Architecture Principles

- Gradually reduce spatial size
- Increase channels through network

Validate on Real Hardware

Consider **inference speed** and **memory**, not just accuracy

Thank you

Ho-min Park

homin.park@ghent.ac.kr

powersimmani@gmail.com