Deep Learning & Generative AI in Healthcare

Session 03

Gradient-Based Optimization & Error Surfaces

Optimization Challenge

Goal: Find network parameters (weights and biases) that minimize error function E(w) for good test set generalization

Why Gradients?

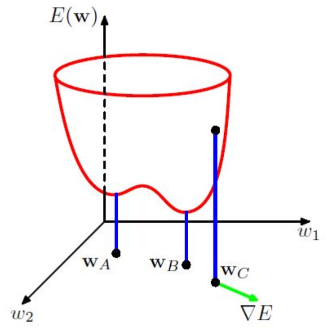
Direct error evaluation is very inefficient. Gradient information via backpropagation enables efficient optimization.

Computational Complexity

Method	Complexity
Without gradients	$O(W^3)$ function evaluations
With gradients	$O(W^2)$ steps
Each gradient eval.	W pieces of information

Backpropagation: Efficient technique to evaluate error function derivatives by flowing computations backward through the network

Error Surface Geometry



Stationary Points

Where ∇E(w) = 0:

- **Global minimum:** Smallest value across all w-space
- **Local minima:** Higher error values
- **Saddle points:** Mixed curvature

Symmetries: Two-layer network with M hidden units has M!2^M equivalent minima (permutations + sign flips)

Gradient Descent Algorithms

Iterative Weight Update

General Form:

$w^{(\tau)} = w^{(\tau-1)} + \Delta w^{(\tau-1)}$

Different algorithms use different choices for $\Delta w^{(\tau)}$

Batch Gradient Descent

Update Rule:

$$w^{(\tau)} = w^{(\tau-1)} - \eta\nabla E(w^{(\tau-1)})$$

- η: learning rate (controls step size)
- Move in direction of steepest descent
- Process entire training set each iteration
- Also called "batch methods"

Stochastic Gradient Descent (SGD)

Update Rule:

$$w^{(\tau)} = w^{(\tau-1)} - \eta\nabla E_n(w^{(\tau-1)})$$

where $E(w) = \Sigma E_n(w)$

- Update based on single data point
- One complete pass = training epoch
- Also called "online gradient descent"
- Handles data redundancy efficiently
- Can escape local minima

Mini-Batch Gradient Descent

Hybrid Approach:

Use small subset of data points per iteration

Algorithm 7.1: Stochastic Gradient Descent

```
Input: Training set {1,...,N}, E_n(w), η, initial w
Output: Final weight vector w

n ← 1
repeat
  w ← w - η∇E_n(w)
  n ← n + 1 (mod N)
until convergence
return w
```

Mini-Batch Considerations

Gradient Noise: Error estimate $\sigma/\sqrt{N}$ (diminishing returns)
100× batch increase → only 10× error reduction

Hardware Efficiency: Powers of 2 work well (64, 128, 256)

Backpropagation Algorithm

Forward Propagation

General Feed-Forward Network:

$a_j = \Sigma_i w_{ji} z_i$ (pre-activation)

$z_j = h(a_j)$ (activation)

- z_i: activation from previous layer or input
- w_{ji}: weight connecting unit i to j
- $h(\cdot)$: nonlinear activation function
- Biases included via fixed +1 input

Error Signal δ

Definition:

$\delta_j \equiv \partial E_n / \partial a_j$

Error signal for unit j (often called "errors")

Output Units:

$\delta_k = y_k - t_k$

(for canonical link with sum-of-squares)

Backward Propagation

Chain Rule Application:

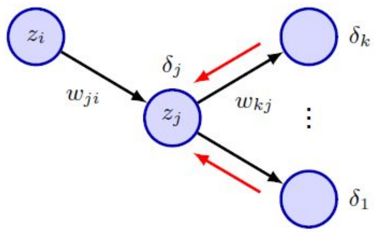
$\partial E_n / \partial w_{ji} = (\partial E_n / \partial a_j)(\partial a_j / \partial w_{ji})$

$= \delta_j z_i$

Backpropagation Formula:

$\delta_j = h'(a_j) \Sigma_k w_{kj} \delta_k$

Propagate δ's backwards from units k to unit j



Key Insight: Calculate δ for each unit, then apply $\partial E_n / \partial w_{ji} = \delta_j z_i$

Structured Data & Computer Vision Applications

Structured vs Unstructured Data

Unstructured	Structured
Elements independent Random permutation OK	Relationships between variables Permutation breaks structure
Standard neural networks	Specialized architectures (CNNs)

Image Data Properties

Key Characteristics:

- Rectangular array of pixels
- RGB channels (triplet per pixel)
- 8-bit precision: 0-255 range
- High dimensionality (megapixels)
- Strong local correlations
- Spatial 2D grid structure

3D Extensions: MRI scans (voxels), Videos (time-stacked frames)

Computer Vision Applications

1. **Classification:** Image recognition (e.g., skin lesion → benign/malignant)
2. **Detection:** Object locations (e.g., pedestrians in autonomous vehicles)
3. **Segmentation:** Pixel-wise classification (e.g., cancerous vs. normal tissue)
4. **Caption Generation:** Text description from image
5. **Synthesis:** Generate new images (e.g., human faces)
6. **Inpainting:** Replace image region with synthesized pixels
7. **Style Transfer:** Transform image style (photo → painting)
8. **Super-Resolution:** Increase image resolution
9. **Depth Prediction:** Predict scene distance from camera
10. **Scene Reconstruction:** 2D images → 3D representation

Convolutional Filters & Feature Detectors

Motivation for CNNs

Fully Connected Networks Problem:

- $10^3 \times 10^3$ color image = 3×10^6 pixels
- 1,000 hidden units → 3×10^9 weights in first layer alone!
- Must learn invariances from data (huge datasets needed)

CNN Solution:

Incorporate inductive biases about image structure → dramatically reduce parameters and improve generalization

Key Concepts

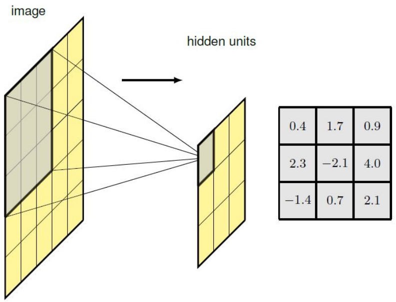
Hierarchy: Natural hierarchical structure (edges → shapes → objects)

Locality: Features detected from local patches

Equivariance: Translation equivariance (shift input → shift output)

Invariance: Small translations don't affect classification

Receptive Field & Feature Detection



Receptive Field: Small rectangular patch from image that a unit receives as input

Example: 3×3 patch visualized as kernel/filter

Feature Detector Output:

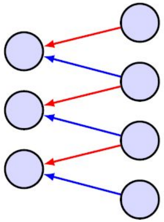
$z = \mathrm{ReLU}(w^T x + w_0)$

Maximum response when image patch matches kernel (up to scaling)

Convolution Operation & Properties

Translation Equivariance

Weight Sharing: Replicate same hidden unit weights at multiple locations across image



Feature Map: All units share same weights

- Sparse connections (most absent)
- Shared weights (parameter efficiency)
- Convolution transformation

2D Convolution Formula

$$C(j,k) = \Sigma_l \Sigma_m I(j+l, k+m)K(l,m)$$

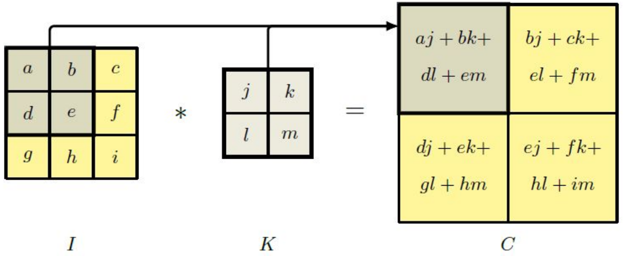
C - I ∗ K (cross-correlation)

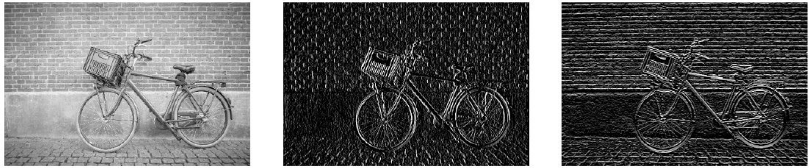
Edge Detection Example

Vertical Edge Filter (3×3):

-1	0	1
-1	0	1
-1	0	1

Detects local intensity changes moving horizontally





(a) (b) (c)

Padding, Strides & Multi-dimensional Convolutions

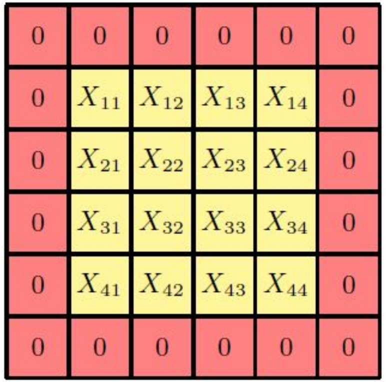
Padding

Problem: Convolution reduces feature map size

J×K image ∗ M×M kernel → (J-M+1)×(K-M+1) feature map

Padding Types:

- **Valid (P=0):** No padding, shrinks output
- **Same (P=(M-1)/2):** Output same size as input
- Typically: zero padding (after mean subtraction)



Strided Convolutions

Purpose: Create smaller feature maps for flexibility

Move filter in steps of size S (stride)

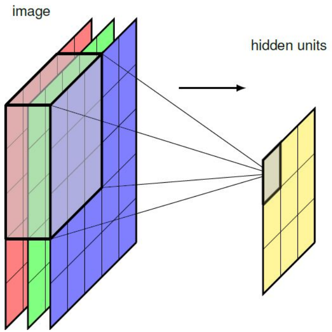
Output size: ⌊(J+2P-M)/S⌋ × ⌊(K+2P-M)/S⌋

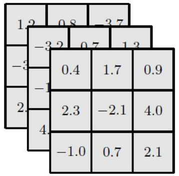
Large images + small filters → roughly 1/S smaller

Multi-dimensional Convolutions

Color Images: 3 channels (R, G, B)

- Image I: J×K×C tensor
- Filter K: M×M×C tensor
- Separate M×M filter per channel
- Output: single feature map per filter





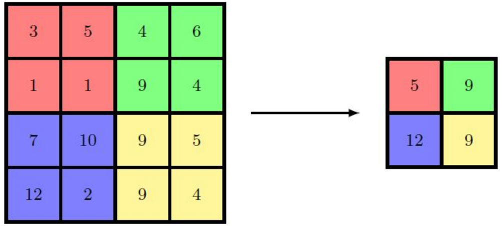
Pooling & Multilayer Convolutional Networks

Pooling Operations

Purpose: Build translation invariance and reduce dimensionality

Max-Pooling:

- No learnable parameters
- Fixed function of receptive field
- Preserves feature presence, discards position
- Example: 2×2 receptive field, stride 2

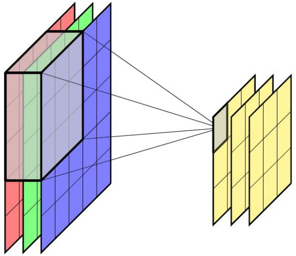


Variants: Average pooling, other pooling functions

Multilayer Architecture

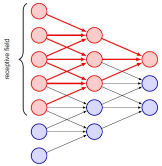
Layer Structure:

- Filter tensor: M×M×C_IN×C_OUT
- Parameters: (M^2C+1)C_OUT per layer
- Multiple filters → multiple feature maps (channels)
- C_OUT channels become C_IN for next layer



Growing Receptive Fields

Depth Effect: Effective receptive field grows with network depth



CNN Architectures: ImageNet & VGG-16

ImageNet Challenge

Dataset:

- 14 million natural images
- ~22,000 categories (hand-labeled)
- Challenge: 1,000 non-overlapping categories
- 1.28M training, 50K validation, 100K test

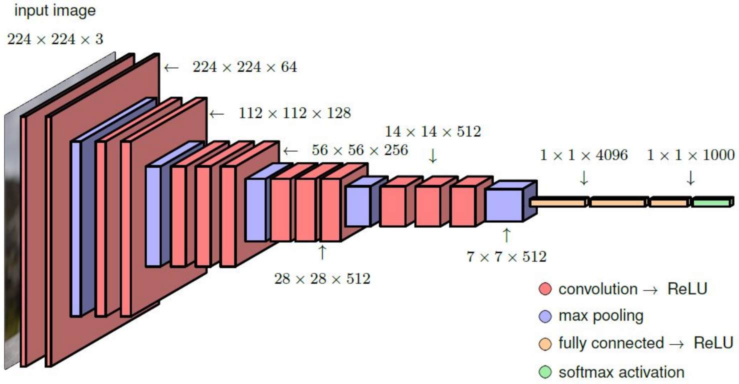
Evaluation Metrics:

- Top-1 error: true class at rank 1
- Top-5 error: true class in top 5 predictions
- Random guessing: 99.9% error

Historical Performance

Model	Year	Top-5 Error
Early results	~2010	~25.5%
AlexNet	2012	15.3%
Later advances	~2017	~3%
Human-level	-	~5%

VGG-16 Architecture



Design Principles (Simonyan & Zisserman, 2014):

- Input: 224×224×3 pixels
- All conv filters: 3×3, stride 1, same padding, ReLU
- All pooling: 2×2, stride 2 (down-samples by 4×)
- Channels: 64→64→128→256→512 (doubling with down-sampling)
- Final: 3 fully connected (4096, 4096, 1000 units)
- ~138M parameters (103M in first FC layer!)

Normalization in Neural Networks

Why Normalize?

Removes the need for networks to deal with extremely large or small values — crucial for effective training

Three Types of Normalization

Type	Normalizes Across	When Applied
Data Normalization	Input data (all samples)	Once, before training
Batch Normalization	Mini-batches	Each mini-batch
Layer Normalization	Hidden units per sample	Each layer, each sample

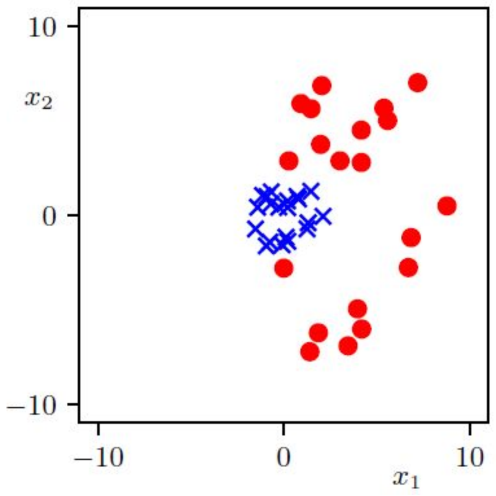
Although weights/biases can theoretically adapt to any input range, normalization makes gradient descent much more stable

Healthcare Example

Patient height: ~1.8 m

Blood platelet count: ~300,000 per μL

Such scale variations create error surfaces with very different curvatures along different axes → challenging for gradient descent



Data Normalization

Goal:
Re-scale continuous inputs so they span similar ranges (zero mean, unit variance)

Procedure

Step 1: Compute statistics (once, before training)

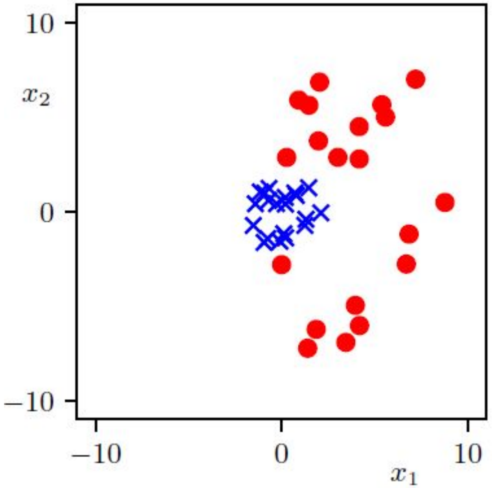
$$\mu_i = (1/N)\ \Sigma_n\ x_{ni}$$

$$\sigma_i^2 = (1/N)\ \Sigma_n\ (x_{ni} - \mu_i)^2$$

Step 2: Standardize

$$\tilde{x}_{ni} = (x_{ni} - \mu_i)\ /\ \sigma_i$$

Important:
Use the same μ_i and σ_i values for validation and test data to ensure consistent scaling



Batch Normalization: Algorithm

Step 1: Normalize

For mini-batch of size K:

$$\mu_i = (1/K)\ \Sigma_n\ a_{ni}$$

$$\sigma_i^2 = (1/K)\ \Sigma_n\ (a_{ni} - \mu_i)^2$$

$$\hat{a}_{ni} = (a_{ni} - \mu_i)\ /\ \sqrt{(\sigma_i^2 + \delta)}$$

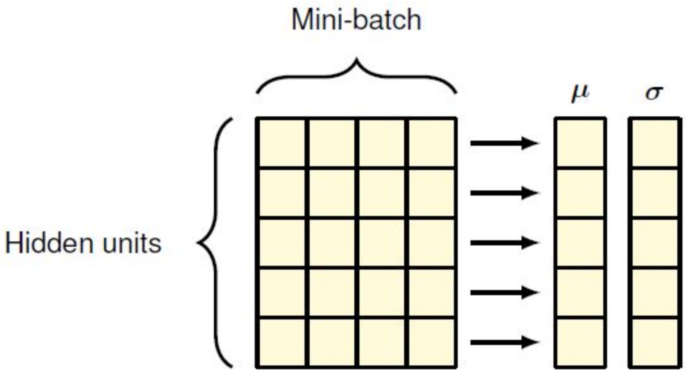
δ = small constant for numerical stability

Step 2: Scale and Shift

$$\tilde{a}_{ni} = \gamma_i\ \hat{a}_{ni} + \beta_i$$

γ_i, β_i are **learnable parameters** trained via gradient descent

Why learnable params?
Normalization reduces representational capacity. γ, β let network learn optimal mean/variance — easier to optimize than implicit weight dependence.



Layer Normalization

Motivation (Ba, Kiros & Hinton, 2016)

- Small batch sizes → noisy mean/variance estimates
- Large training sets across GPUs → global batch norm inefficient
- RNNs: distributions change each timestep

Key Difference

Batch Norm	Layer Norm
Across mini-batch per hidden unit	Across hidden units per data point

Equations (M hidden units):

$$\mu_n = (1/M)\ \Sigma_i\ a_{ni}$$

$$\hat{a}_{ni} = (a_{ni} - \mu_n) / \sqrt{(\sigma_n^2 + \delta)}$$

Advantages over Batch Norm:

- No dependence on batch size
- Same computation for training and inference
- No need to store moving averages
- Works well for transformers & RNNs