# Minor in AI

## MLP => CNN

# 1 Introduction: The Birth of Digital Vision

Real-World Case Study

**The Post Office Problem (1980s):**
Imagine mountains of handwritten letters arriving daily at a post office. Workers had to manually read ZIP codes, a tedious and error-prone process. Computer scientist Yann LeCun faced this challenge: **"How can machines learn to read handwritten digits as accurately as humans?"**

- **Challenge:** Same digit (e.g., '5') varies in shape, size, and tilt
- **Solution Path:** Create neural networks that learn patterns from examples
- **Breakthrough:** MNIST dataset - 70,000 labeled digit images

# 2 Multi-Layer Perceptrons (MLPs): Digital Brain Cells

## 2.1 Anatomy of an MLP

An MLP works like a team of detectives:

- **Input Layer:** 784 "sensors" (28x28 pixel values)
- **Hidden Layers:** Detect patterns (edges → curves → shapes)
- **Output Layer:** 10 "jurors" voting for digit 0-9

Why Matrix Math?

**Neural networks = Fancy calculators:**
Layers communicate through matrix operations. For layer 1 with 4 neurons:

$$\begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1,784} \\ w_{21} & w_{22} & \cdots & w_{2,784} \\ w_{31} & w_{32} & \cdots & w_{3,784} \\ w_{41} & w_{42} & \cdots & w_{4,784} \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{784} \end{bmatrix} = \begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{bmatrix}$$

Each weight $w_{ij}$ is a **connection strength** between pixel $j$ and neuron $i$.

## 2.2 Code Walkthrough: Matrix Operations

Listing 1: Python Implementation of Layer Calculation

```
import numpy as np

# Flatten 28x28 image into 784 numbers (0-255)
pixel_values = np.random.rand(784)  # Simulated MNIST image

# First hidden layer weights (4 neurons)
```

```
weights_layer1 = np.random.randn(4, 784) * 0.1  # Small random values

# Calculate neuron activations
raw_output = np.dot(weights_layer1, pixel_values)  # Matrix multiplication
activated_output = 1 / (1 + np.exp(-raw_output))  # Sigmoid "squish"

print(f"Neuron activations: {activated_output}")
```

**Code Explanation**

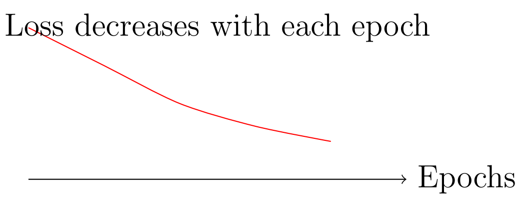
**What's happening here:**

- `np.random.rand(784)`: Simulates normalized pixel values (0=black, 1=white)
- `np.random.randn(4, 784)`: Creates 4×784 weight matrix (Gaussian distribution)
- `np.dot()`: Matrix multiplication (input pixels × weights)
- `1/(1+np.exp(-x))`: Sigmoid activation (squishes values to 0-1)

# 3 Training Dynamics: Teaching Neural Networks

## Three key ingredients:

1. **Batches: The Classroom Analogy**
   Like studying flashcards in small sets rather than entire textbooks
   - Process 64-256 images at once (mini-batches)
   - Why? *Memory efficiency* and *frequent updates*
   - Example: 60,000 MNIST images → 600 batches of 100 images
   - Code analogy: `for batch in dataset: train(batch)`
2. **Epochs: The Semester System**
   Complete passes through all training data
   - 1 epoch = 1 full cycle through all 60,000 images
   - Typical training: 10-100 epochs
   - Why multiple? *Gradual learning* like semester revisions




3. **Backpropagation: The Blame Game**
   "Which neuron made the biggest mistake?"

- Calculates error gradients using chain rule
- Flows backward: Output → Hidden layers → Input

**Weight Update Equation**

$$\underbrace{W_{new}}_{\text{Updated weights}} = \underbrace{W_{old}}_{\text{Current weights}} - \underbrace{\alpha}_{\text{Learning rate}} \times \underbrace{\frac{\partial L}{\partial W}}_{\text{Error gradient}}$$

**Key components:**

- Learning Rate ($\alpha$): Controls step size (0.001 typical)
  - Too small: Slow learning (turtle pace)
  - Too big: Overshooting (jumping valleys)
- Gradient ($\frac{\partial L}{\partial W}$): Direction to reduce error

# 4 The Flattening Problem: Why MLPs Struggle

## Spatial Relationships

**The Jigsaw Puzzle Analogy:** Imagine solving two different puzzles:

- **Intact Puzzle:** Edges match, colors flow naturally
- **Shuffled Pieces:** No spatial relationships, pure color matching

MLPs work with shuffled pieces! When we flatten a 28×28 image into 784 pixels:

- **Lost:**
  - Local patterns (horizontal edge at top, curve at bottom-right)
  - Relative positions (nose above mouth in face recognition)
  - Translation invariance (recognize '5' anywhere in image)
- **Kept:**
  - Global brightness (dark pixels vs light background)
  - Individual pixel intensities (0-255 values)

**Why This Matters for Digits:** A handwritten '8' requires:

- Top loop + bottom loop + middle cross
- Spatial arrangement crucial

MLP's Limited Perspective

**"I see pixels, not patterns!"**
An MLP neuron might learn:

- Pixel 123 is dark → maybe a '7'
- Pixel 456 is bright → maybe a '0'

But misses the **relationships** between pixels 123 and 456!

# 5 The Path to Convolutional Networks

## Key Takeaways

- **The Flattening Paradox:**
  - MLPs require 1D vectors → Destroys 2D structure
  - Like reading a book by scrambling all words
- **Training Machinery:**
  - Batches: 64-256 samples (memory-efficient learning)
  - Backpropagation: Error feedback through layers
  - Weight updates: $W = W - \alpha \nabla L$ (learning rate )
- **Probability Conversion:**
  - Softmax: $p_i = \frac{e^{q_i}}{\sum e^{q_j}}$
  - Converts scores → Probability distribution
- **The Spatial Crisis:**
  - MLP accuracy plateau: 97-98% on MNIST
  - Real-world images need **hierarchical pattern recognition**

**The CNN Revolution:** Convolutional Neural Networks solve spatial blindness through:

- Local receptive fields: Detect edges/textures
- Weight sharing: Same pattern detectors across image
- Pooling: Spatial hierarchy preservation

**Historical Insight:** Yann LeCun's 1998 LeNet-5 (early CNN) achieved 99.2% on MNIST - a breakthrough showing spatial processing matters!