Computer Vision
How Machines See

From Pixels to Object Detection

Nipun Batra | IIT Gandhinagar

The Story So Far

Week	We Learned
5	Neural networks: MLPs, backprop, PyTorch
6	How to apply neural networks to IMAGES

Today's Agenda

1. **Images as Data** - How computers "see"
2. **Convolutional Neural Networks** - The breakthrough
3. **CNN Architecture** - Building blocks
4. **Object Detection** - What + Where
5. **YOLO** - Real-time detection

Part 1: Images as Data

How Computers "See"

What Is an Image to a Computer?

To humans: A scene, objects, colors, emotions

To computers: A grid of numbers!

```
┌──────────────────────────────┐
│ 145  148  152  155  159  ... │  Row 0
│ 147  151  156  159  162  ... │  Row 1
│ 149  153  158  162  166  ... │  Row 2
│ ...  ...  ...  ...  ...      │
└──────────────────────────────┘

Each number = pixel brightness (0-255)
```

Grayscale vs Color

Type	Dimensions	Example
Grayscale	Height × Width	28 × 28 = 784 values
Color (RGB)	Height × Width × 3	224 × 224 × 3 = 150,528 values

RGB = Red, Green, Blue channels

```
Image shape: (224, 224, 3)
             height, width, channels
```

MNIST: The "Hello World" of Vision

```
┌──────────┐
│ 28 × 28  │  = 784 pixels
│ 0  0  0  7 │
│ 0  0 23 155│   Each pixel: 0-255
│ 0  0 89 254│   (grayscale)
│ ...      │
└──────────┘

Label: "5"
```

Why Not Use MLPs for Images?

Problem 1: Too many parameters

- Image: 224 × 224 × 3 = 150,528 inputs
- Hidden layer: 1,000 neurons
- Parameters: 150,528 × 1,000 = **150 million weights!**

Why Not Use MLPs for Images? (cont.)

Problem 2: No spatial awareness

The same object at different positions looks completely different to an MLP.

What We Need

Requirement	Solution
Fewer parameters	Weight sharing
Translation invariance	Local patterns
Spatial structure	Keep 2D arrangement

Enter: Convolutional Neural Networks (CNNs)

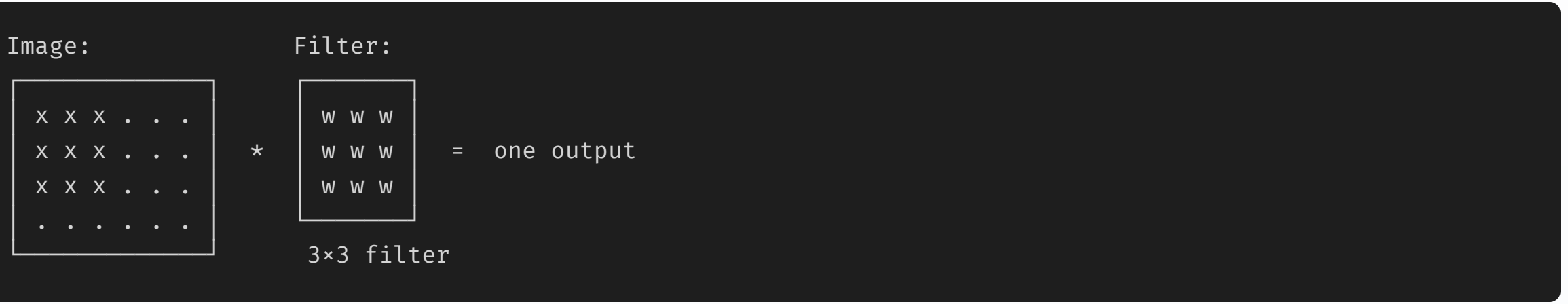
Part 2: Convolutional Neural Networks

The Breakthrough (2012)

The Key Insight

Instead of connecting every pixel to every neuron...

Connect small **local regions** to neurons!



Convolution Operation

Slide a small filter across the image:

```
Image              Filter          Output

┌─────────┐        ┌─────┐
│ 1 2 3 0 1 │      │ 1 0 1 │
│ 0 1 2 3 2 │  *   │ 0 1 0 │   =   [12, ...]
│ 1 2 1 0 1 │      │ 1 0 1 │
│ 0 1 2 3 2 │      └─────┘
│ 2 1 0 1 2 │
└─────────┘

1×1 + 2×0 + 3×1 + 0×0 + 1×1 + 2×0 + 1×1 + 2×0 + 1×1 = 12
```

Why Convolution Works

Property	Benefit
Local connectivity	Each neuron sees a small region
Weight sharing	Same filter applied everywhere
Translation equivariance	Object detected regardless of position

Filters Learn Features

Different filters detect different patterns:

Filter	What It Detects
Edge detector	Boundaries between regions
Corner detector	Sharp corners
Texture detector	Repeating patterns

The network LEARNS which filters are useful!

Convolution in Code

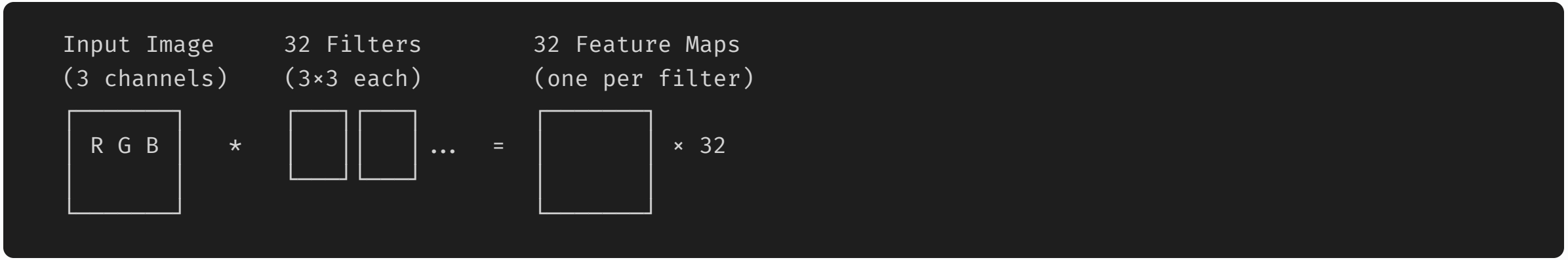
```
import torch.nn as nn

# Convolutional layer
conv = nn.Conv2d(
    in_channels=3,    # RGB input
    out_channels=32,  # 32 filters
    kernel_size=3,    # 3×3 filters
    padding=1         # Keep same size
)

# Apply to image
x = torch.randn(1, 3, 224, 224)  # [batch, channels, H, W]
out = conv(x)  # [1, 32, 224, 224]
```

Feature Maps

Each filter produces a **feature map**:



Part 3: CNN Architecture

The Building Blocks

CNN Building Blocks

Layer	Purpose	Output Change
Conv	Detect features	Different channels
ReLU	Non-linearity	Same size
Pooling	Reduce size	Smaller spatial
Flatten	2D → 1D	Vector
Linear	Classification	Class scores

Max Pooling

Take the maximum value in each region:

Input (4×4):

1	3	2	1
4	2	6	5
7	8	1	0
3	5	9	2

→

Max Pool (2×2):

4	6
8	9

Reduces spatial size by 2×

Why Pooling?

Benefit	Explanation
Reduces parameters	Smaller feature maps
Provides invariance	Small shifts don't matter
Increases receptive field	Later layers "see" more

A Simple CNN

```
class SimpleCNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, padding=1)  # 28→28
        self.pool1 = nn.MaxPool2d(2)                  # 28→14
        self.conv2 = nn.Conv2d(32, 64, 3, padding=1) # 14→14
        self.pool2 = nn.MaxPool2d(2)                  # 14→7
        self.fc = nn.Linear(64 * 7 * 7, 10)          # Classify

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.pool1(x)
        x = F.relu(self.conv2(x))
        x = self.pool2(x)
        x = x.view(-1, 64 * 7 * 7)  # Flatten
        x = self.fc(x)
        return x
```

CNN for MNIST

```
Input: 28×28×1
    ↓ Conv1 (32 filters)
28×28×32
    ↓ MaxPool (2×2)
14×14×32
    ↓ Conv2 (64 filters)
14×14×64
    ↓ MaxPool (2×2)
7×7×64 = 3136
    ↓ Flatten + Linear
10 class scores
```

Famous CNN Architectures

Year	Model	Key Innovation
1998	LeNet	First CNN
2012	AlexNet	Deep CNN, GPU training
2014	VGG	Very deep (16-19 layers)
2015	ResNet	Skip connections (152 layers!)
2017	EfficientNet	Neural architecture search

AlexNet: The Game Changer

```
ImageNet 2012:

┌──────────────────────────────────────┐
│ Previous best (hand-crafted): 25.8%  │
│ AlexNet (CNN):                16.4%  │  ← 9% improvement!
│ Human performance:            ~5%    │
└──────────────────────────────────────┘
```

This started the deep learning revolution!

Transfer Learning

Don't train from scratch! Use pre-trained models:

```
import torchvision.models as models

# Load pre-trained ResNet (trained on ImageNet)
model = models.resnet50(pretrained=True)

# Replace last layer for your task
model.fc = nn.Linear(2048, num_classes)

# Fine-tune on your data
```

Part 4: Object Detection

What + Where

Classification vs Detection

Task	Question	Output
Classification	"What is this?"	One label
Detection	"What + Where?"	Boxes + labels

Why Detection Matters

Self-driving car needs to know:

- What is there? (pedestrian, car, sign)
- Where is it? (position, distance)
- How many? (one child or five?)

Classification can't answer "where"!

Bounding Boxes

A bounding box = 4 numbers describing a rectangle:

Detection Output

For each object:

- **Class label**: "dog", "car", "person"
- **Bounding box**: (x1, y1, x2, y2)
- **Confidence**: 0.95 (95% sure)

```
detections = [
    {"class": "dog", "box": [100, 50, 300, 200], "conf": 0.95},
    {"class": "cat", "box": [400, 100, 550, 250], "conf": 0.87},
]
```

IoU: Measuring Box Overlap

Intersection over Union:

$$\text{IoU} = \frac{\text{Area of Overlap}}{\text{Area of Union}}$$

IoU	Interpretation
1.0	Perfect match
0.5	Good detection
0.0	No overlap

Non-Maximum Suppression (NMS)

Problem: Model often detects the same object multiple times.

Solution: Keep only the best detection, remove overlapping ones.

```
# Many boxes for same dog
boxes = [[100,50,300,200], [105,55,305,205], [110,48,295,198]]
confs = [0.95, 0.92, 0.88]

# After NMS: keep only [100,50,300,200] with conf 0.95
```

Part 5: YOLO

You Only Look Once

YOLO: The Revolution

Before YOLO (2015): Detection was SLOW (seconds per image)

After YOLO: **Real-time detection** (30-60 FPS!)

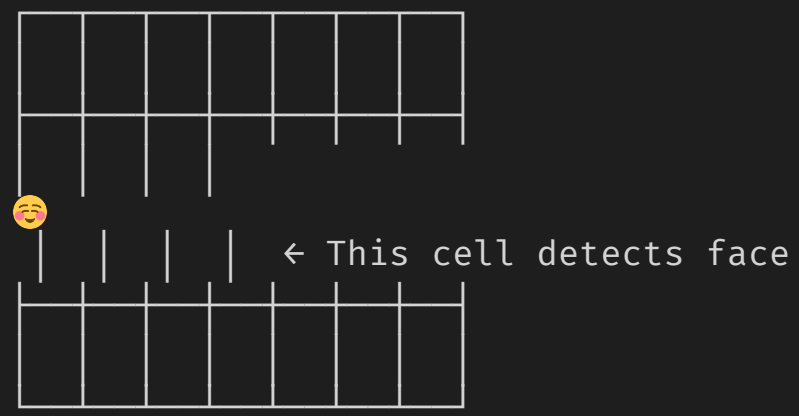
Approach	Speed	How
R-CNN	~50 sec	Propose regions, classify each
Fast R-CNN	~2 sec	Share computation
YOLO	**~0.02 sec**	One network, one pass

YOLO Key Idea

Process the entire image in one forward pass:

1. Divide image into grid (e.g., 7×7)
2. Each cell predicts:
   - Bounding boxes
   - Confidence scores
   - Class probabilities
3. Single neural network does everything!

YOLO Grid



Cell predicts: box coords + confidence + class

Using YOLO in Practice

```python
from ultralytics import YOLO

# Load pre-trained model
model = YOLO('yolov8n.pt')

# Detect objects
results = model('image.jpg')

# Process results
for result in results:
    for box in result.boxes:
        x1, y1, x2, y2 = box.xyxy[0].tolist()
        conf = box.conf[0].item()
        cls = int(box.cls[0].item())
        print(f"Class {cls}: ({x1:.0f},{y1:.0f}) to ({x2:.0f},{y2:.0f})")
```

YOLO Model Sizes

Model	Speed	Accuracy	Use Case
YOLOv8n	Fastest	Lower	Mobile, edge
YOLOv8s	Fast	Medium	General use
YOLOv8m	Medium	Good	Better accuracy
YOLOv8l	Slower	Better	High accuracy
YOLOv8x	Slowest	Best	Maximum accuracy

Training Your Own Detector

```python
from ultralytics import YOLO

# Start with pre-trained model
model = YOLO('yolov8n.pt')

# Train on custom data
model.train(
    data='my_dataset.yaml',
    epochs=100,
    imgsz=640
)

# Evaluate
metrics = model.val()
print(f"mAP@0.5: {metrics.box.map50:.3f}")
```

Detection Metrics

Metric	What It Measures
Precision	Of detections, how many correct?
Recall	Of actual objects, how many found?
mAP@0.5	Average precision at IoU=0.5
mAP@0.5:0.95	Average over IoU thresholds

Key Takeaways

1. **Images = grids of numbers** (pixels)
2. **CNNs use convolutions** for efficient image processing
   - Weight sharing
   - Translation equivariance
3. **CNN building blocks**: Conv → ReLU → Pool → Flatten → Linear
4. **Detection = Classification + Localization**
5. **YOLO** enables real-time object detection

You Can Now Make Computers See!

Next: Language Models - How Machines Understand Text

Lab: Build a CNN classifier and run YOLO detection

"The question is not whether intelligent machines can have any emotions, but whether machines can be intelligent without any emotions."

— Marvin Minsky

Questions?