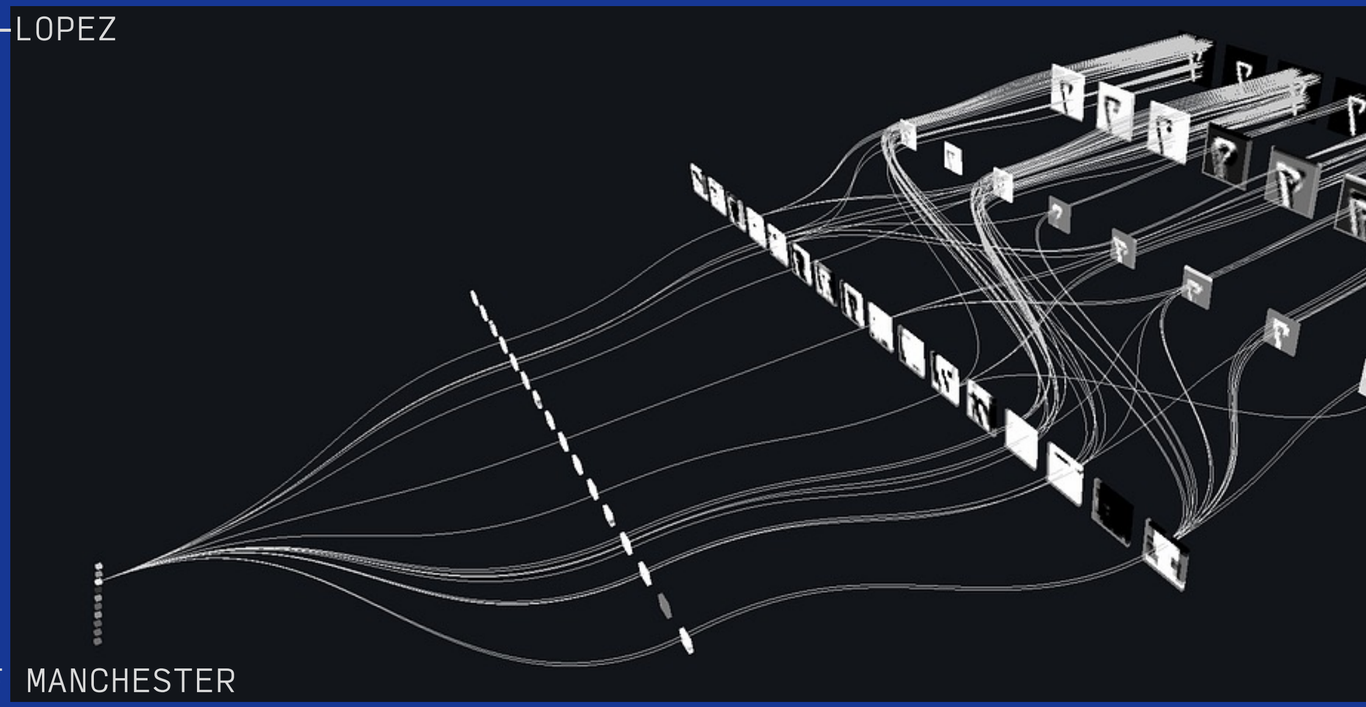
SESSION 04

WORKSHOP MARCH 2026 BY DR CORONA-LOPEZ

CONVOLUTIONAL NEURAL NETWORKS



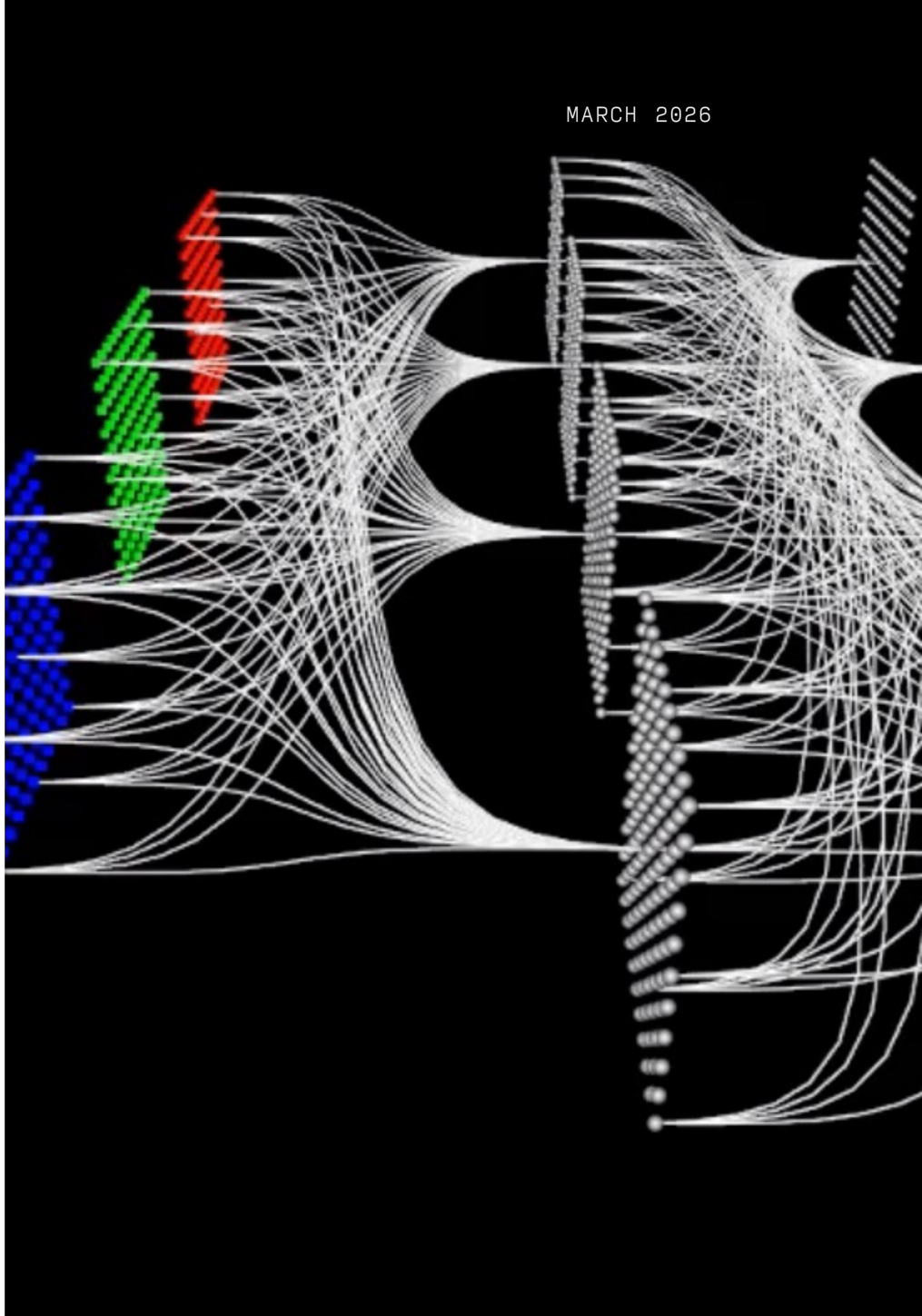
FACULTY OF SCIENCE AND ENGINEERING UNIVERSITY OF MANCHESTER

BY DR CORONA-LOPEZ

MARCH 2026

AGENDA

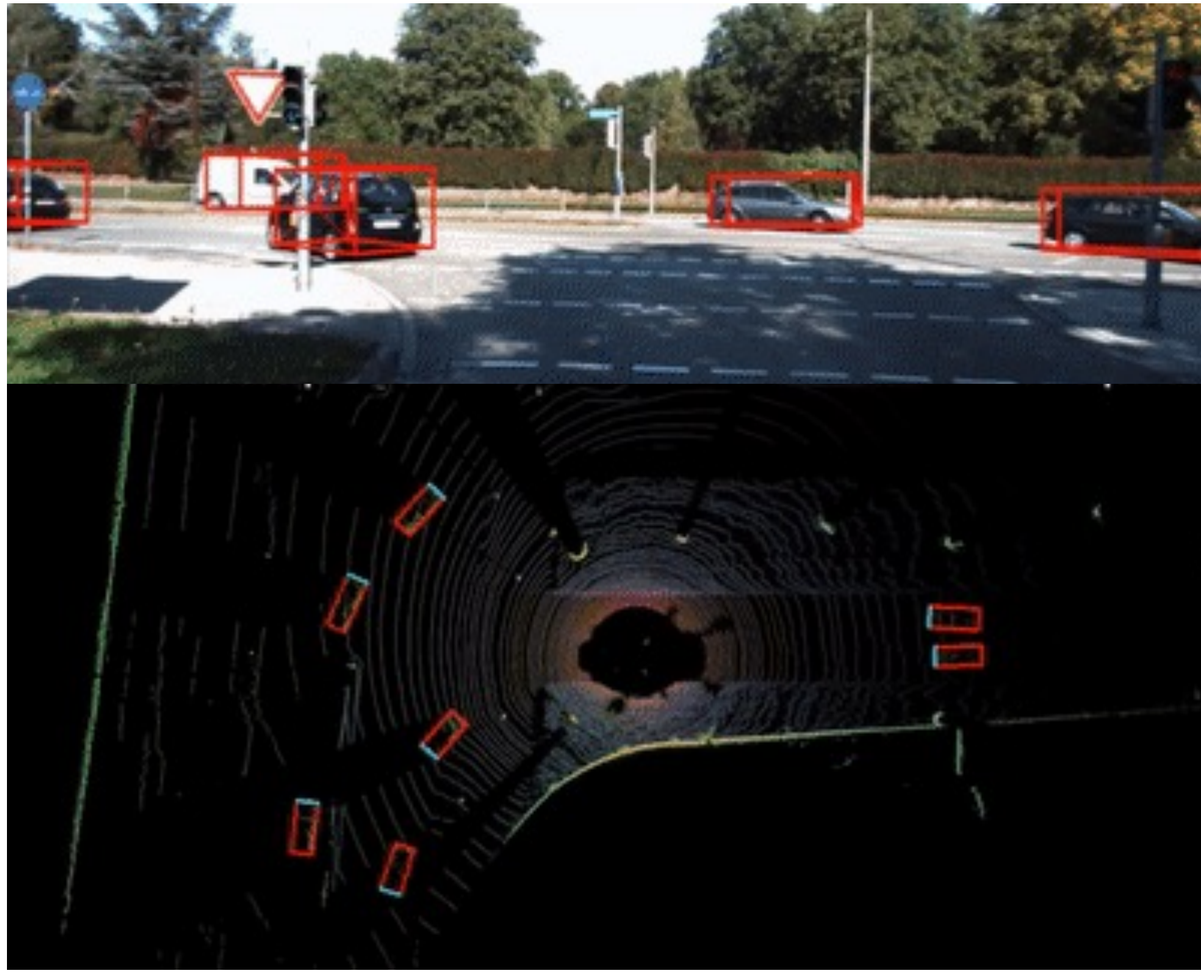
- **convolutional neural networks:** why standard networks struggle with images.
- **the convolution operation:** kernels, filters, and calculating output dimensions.
- **preparing image data:** torchvision transforms and the historical cracks dataset.
- **implementing cnns:** building a simple cnn and dataloaders.
- **evaluation:** accuracy and classification reports.
- **advanced architectures:** mobilenet.



CNNs represent a monumental shift in artificial intelligence, moving beyond simple spreadsheets and audio waves to actually "seeing" and understanding visual data.

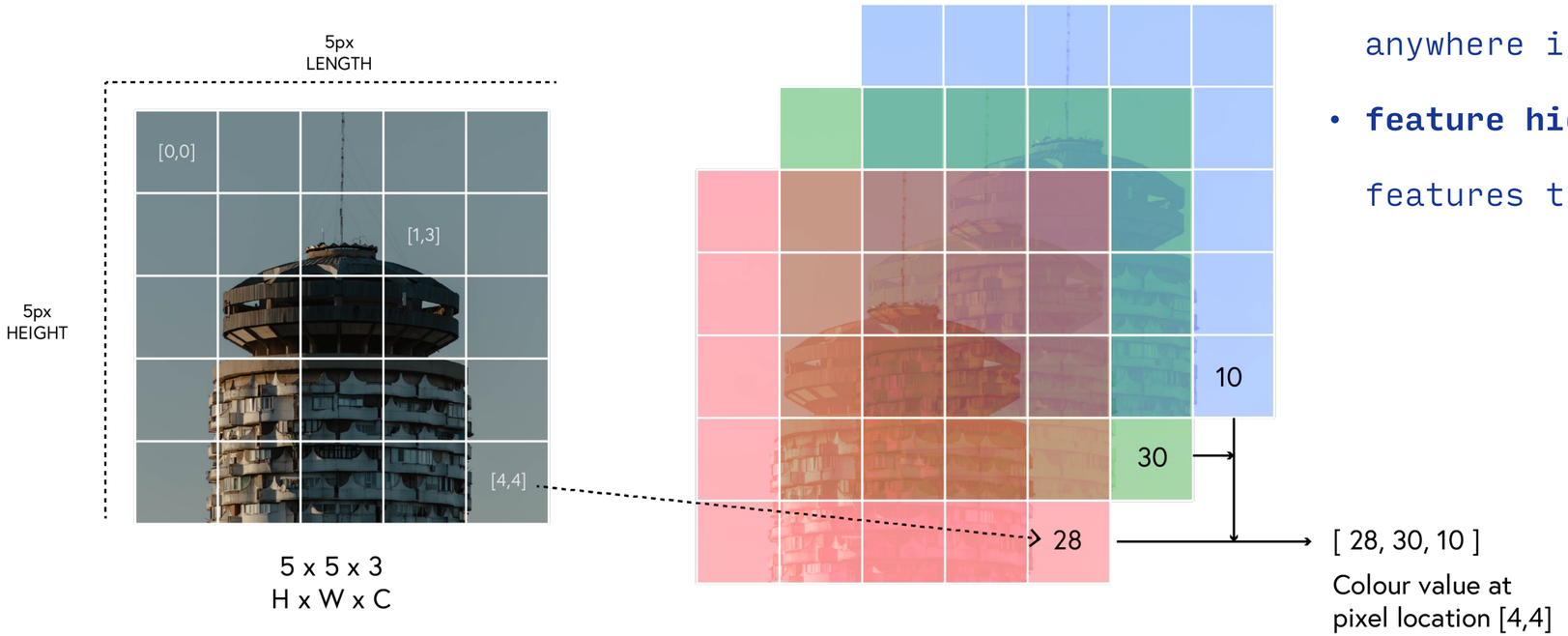
unlike standard networks that process flattened lists of numbers, cnns are specifically engineered to comprehend spatial relationships, geometry, shapes, and textures within images and video.

in practice, this allows cnns to perform highly complex, real-world visual tasks. they can seamlessly execute image classification to determine the main subject of a photo, perform object detection to draw precise bounding boxes around pedestrians for autonomous vehicles.



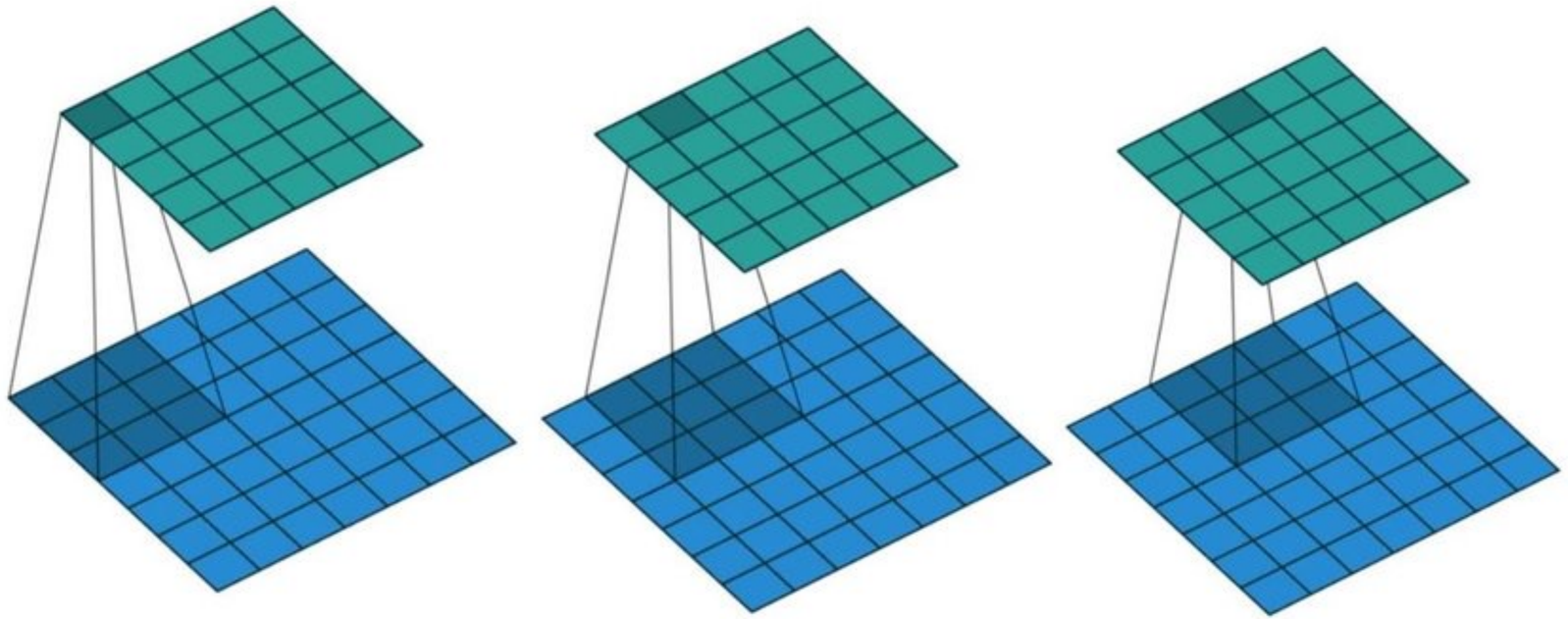
ANN STRUGGLE WITH IMAGES

- **spatial relationships:** standard networks don't account for spatial relationships between pixels
- **parameter explosion:** a 224×224×3 image would require over 150,000 weights per neuron
- **translation invariance:** objects can appear anywhere in an image but have the same meaning
- **feature hierarchy:** images contain low-level features that compose into higher-level features



CONVOLUTION

definition: sliding a small window (kernel/filter) over an image, performing element-wise multiplication, and summing the results to create a "feature map".



CONVOLUTION

1. **kernel size:** the dimensions of the filter
2. **stride:** how many pixels the filter shifts
3. **padding:** adding extra pixels around the border

$$Output = \frac{Input - Kernel + (2 \times Padding)}{Stride} + 1$$

```
conv_layer = torch.nn.Conv2d(in_channels,
                             out_channels,
                             kernel_size,
                             bias,
                             padding,
                             groups)
```

FILTERS

filters are small matrices that detect specific patterns in images

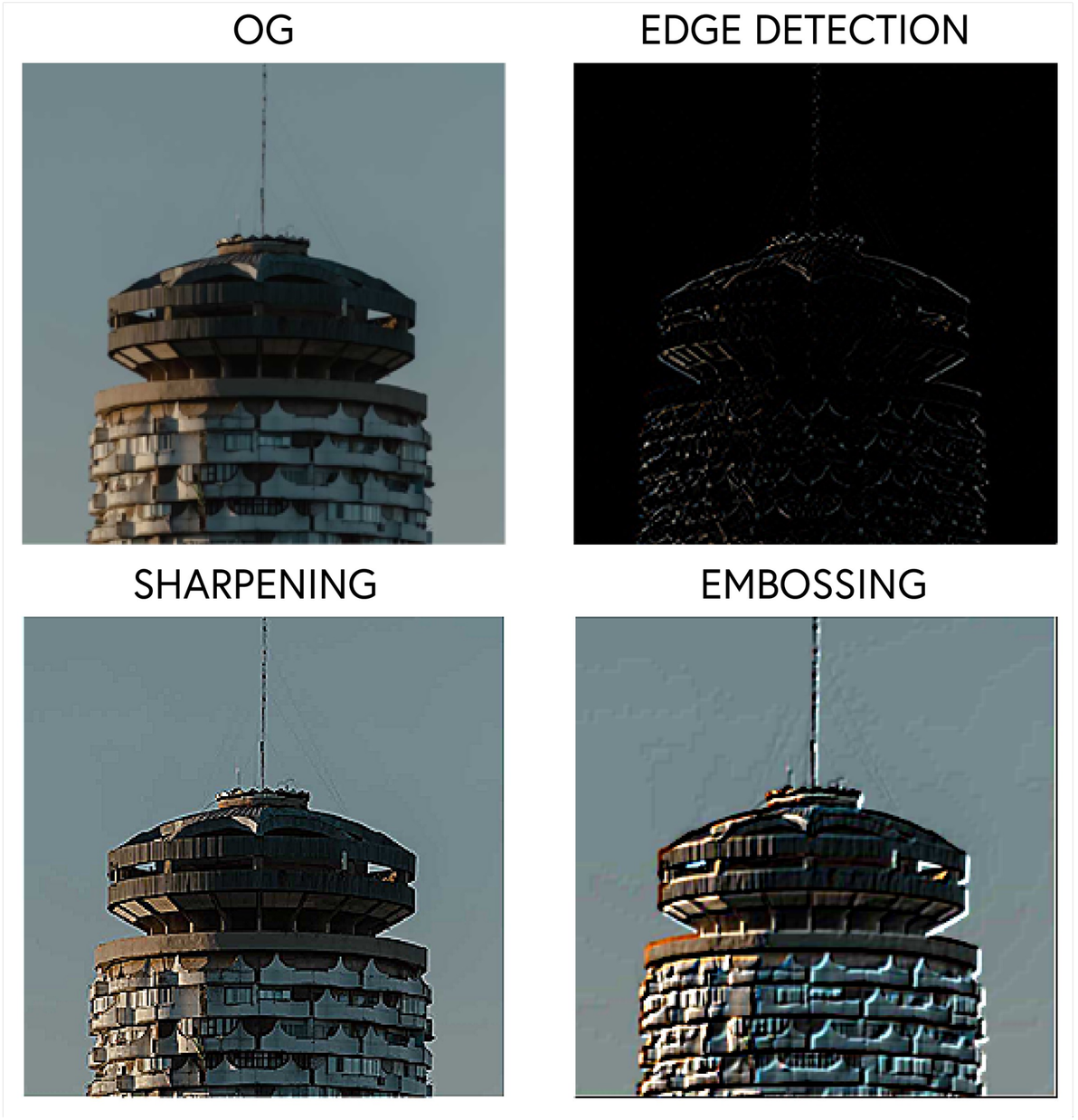
- different filters detect different features:
  - edge detection:

```
[[1, 0, -1],
 [1, 0, -1],
 [1, 0, -1]]
```

  - sharpening:

```
[[ 0, -1,  0],
 [-1,  5, -1],
 [ 0, -1,  0]]
```

- weights in filters are learned during training



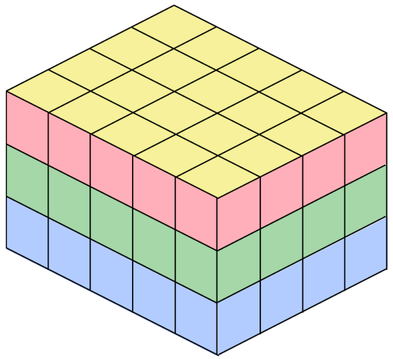
WORKING WITH IMAGES

we need to convert images into proper format for CNNs

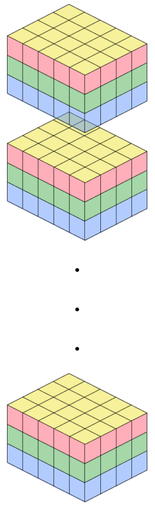
- pytorch expects 4d tensors:
  (batch_size, channels, height, width)

- data augmentation techniques increase training set diversity

[3 x 5 x 4]



BATCHES x CHANNELS x HEIGHT x WIDTH
[N x 3 x 5 x 4]



[B x C x H x W]

[B x H x W x C]

PRE-PROCESSING

```python
from torchvision import transforms

train_transforms = transforms.Compose([
    transforms.Resize((64, 64)),
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(15),
    transforms.ColorJitter(brightness=0.2, contrast=0.2),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406],
                         [0.229, 0.224, 0.225])
])
```

torchvision.transforms is a powerful library used to preprocess images on the fly before they enter the neural network.

the pipeline -> transforms.Compose

- **geometric:** flips, rotations, scaling, cropping
- **color:** brightness, contrast, saturation adjustments
- **noise:** adding random noise for robustness
- **occlusion:** random erasing to simulate partial obscuring

combining multiple augmentations:

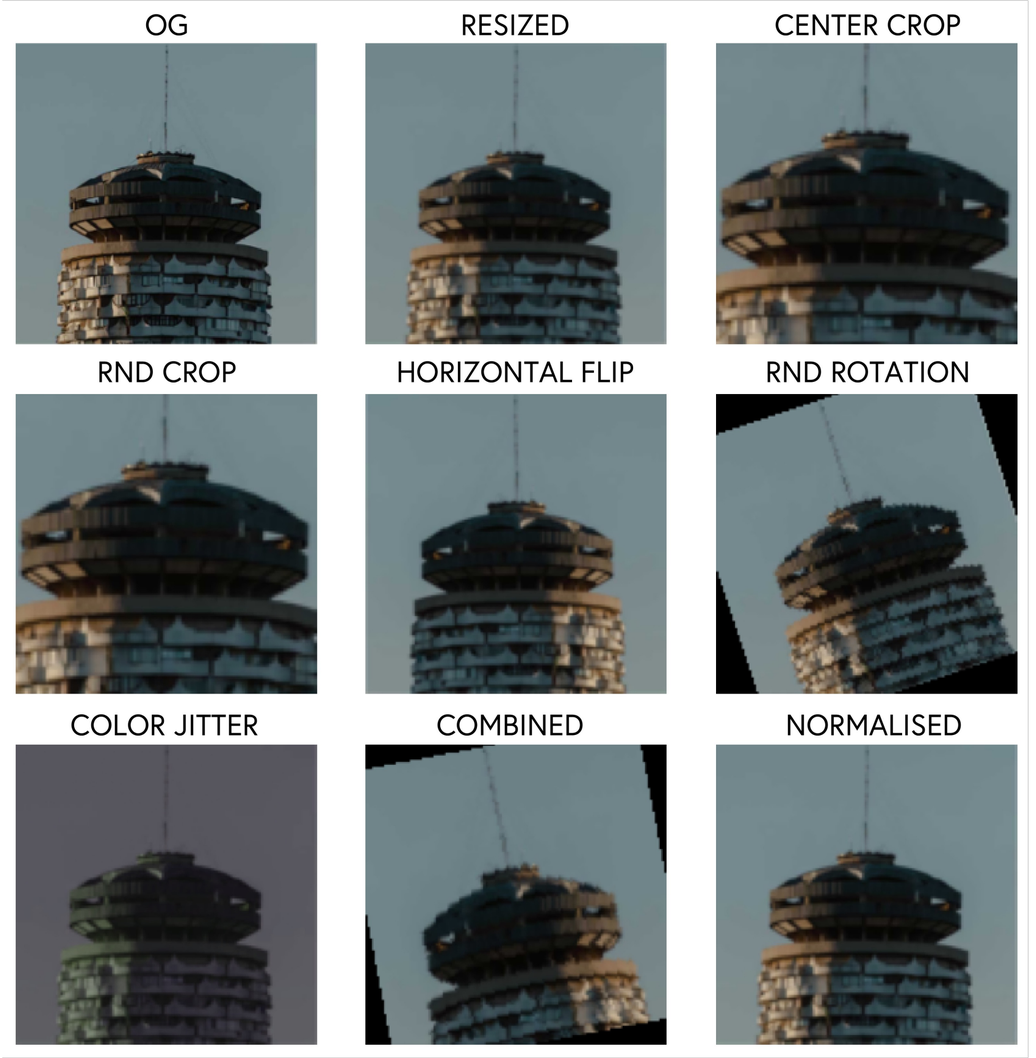
- domain-specific augmentations (e.g., for medical images)
- online vs. offline augmentation

AUGMENTATIONS

artificially expanding dataset by applying transformations

benefits:

- prevents overfitting
- improves model generalization
- handles varied real-world conditions
- addresses class imbalance



BY DR CORONA-LOPEZ

INDUSTRIAL SURFACE DEFECT DETECTION

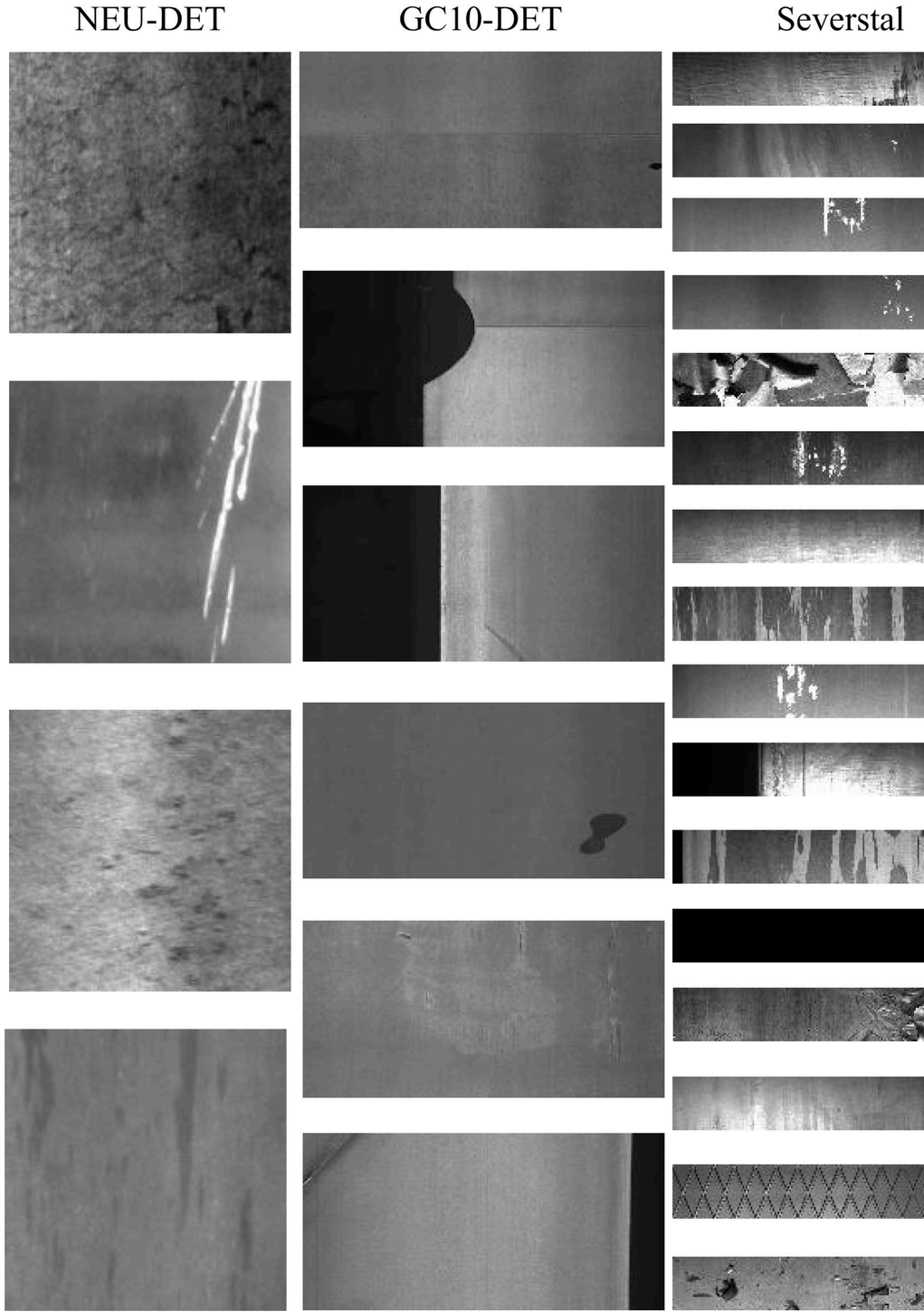
the problem: manual quality control in manufacturing is slow, expensive, and prone to human error.

the objective: build a convolutional neural network (CNN) to automatically classify defects on hot-rolled steel strips.

dataset contents & scale:

- 1,800 total images (grayscale)
- uniform resolution: all images are 200×200 pixels
- perfectly balanced: exactly 300 images per class
- the 6 defect classes: crazing (cr), inclusion (in), patches (pa), pitted surface (ps), rolled-in scale (rs), scratches (sc)

the challenge: high intra-class variation (defects of the same type look different) and low inter-class variation (different defects can look visually similar).



DATASET ORGANISATION

```
neu_cls_dataset/
│
├── train/
│   ├── crazing/          <-- Folder name becomes the class label (0)
│   │   ├── cr_1.jpg
│   │   └── cr_2.jpg
│   ├── inclusion/        <-- Folder name becomes the class label (1)
│   │   ├── in_1.jpg
│   │   └── in_2.jpg
│   └── ... (other classes)
│
└── val/
    ├── crazing/
    ├── inclusion/
    └── ... (other classes)
```

a well-structured dataset allows for:

- data splitting strategies
- stratified splitting: ensures class distribution is maintained across splits
- cross-validation: for smaller datasets or when maximum data usage is needed

DATASET ORGANISATION

```
neu_cls_dataset/
│
├── train/
│   ├── crazing/          <-- Folder name becomes the class label (0)
│   │   ├── cr_1.jpg
│   │   └── cr_2.jpg
│   ├── inclusion/        <-- Folder name becomes the class label (1)
│   │   ├── in_1.jpg
│   │   └── in_2.jpg
│   └── ... (other classes)
│
└── val/
    ├── crazing/
    ├── inclusion/
    └── ... (other classes)
```

```python
from torchvision.datasets import ImageFolder

# Training transforms — include data augmentation for better
generalisation.

ts_train = transforms.Compose([
    transforms.Grayscale(num_output_channels=1),  # ensure 1 channel
    transforms.Resize((128, 128)),
    transforms.ToTensor(),            # scales pixel values to [0, 1]
])


# Load datasets using ImageFolder
train_data = ImageFolder(root=train_folder, transform=ts_train)

print(f"Classes ({len(train_data.classes)}): {train_data.classes}")
print(f"Train samples : {len(train_data)}")
```

DATALOADERS

we cannot load all 1,800 images into the GPU memory at once. we also need to feed the data in a randomized order so the network doesn't memorize patterns.

torch.utils.data.DataLoader wraps an iterable around your torch.utils.data.Dataset to handle all the heavy lifting automatically.

- **batching:** groups images together (e.g., 32 at a time) to speed up training and fit into memory.
- **shuffling:** randomizes the order of images every epoch so the model generalizes better.
- **multiprocessing:** uses multiple cpu cores (num_workers) to load the next batch of images while the GPU is busy training the current batch.

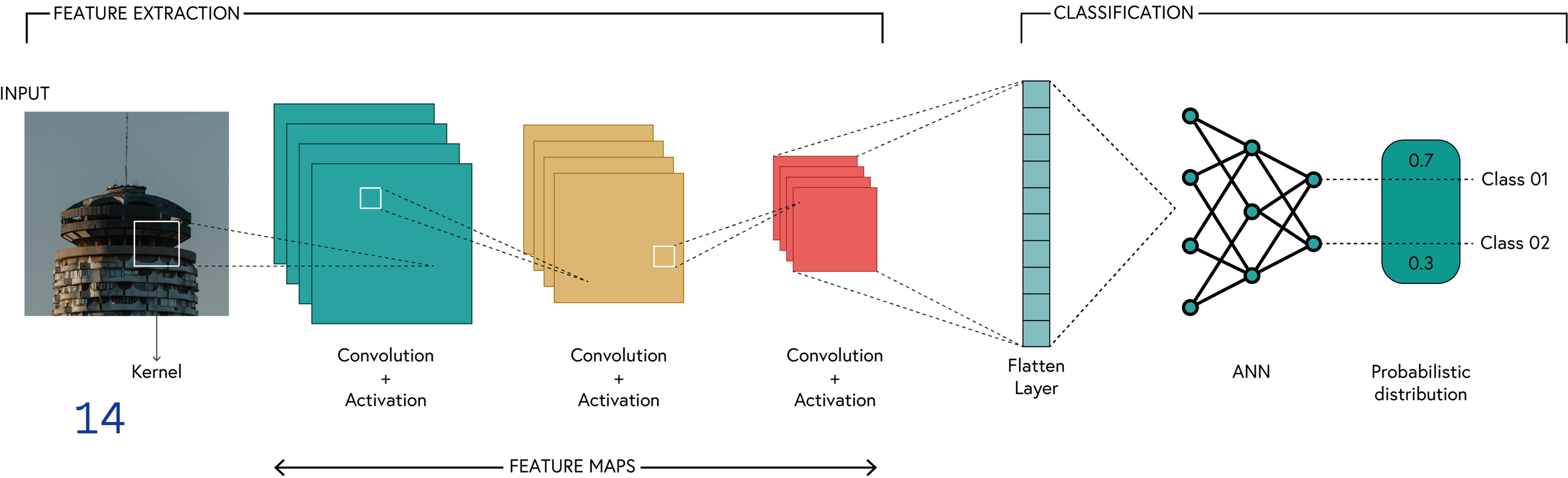
```python
from torch.utils.data import DataLoader

# Create efficient data loading pipeline
train_loader = DataLoader(
    train_dataset,
    batch_size=32,          # Number of samples per batch
    shuffle=True,           # Shuffle data at each epoch
    num_workers=4,          # Parallel data loading processes
    pin_memory=True,        # Speed up data transfer to GPU
    drop_last=False         # Keep incomplete final batch
)

# Inspect a batch
for images, labels in train_loader:
    print(f"Batch shape: {images.shape}")  # torch.Size([32, 3, 224, 224])
    print(f"Labels: {labels}")             # tensor([0, 1, 0, 1, ...])
    break
```

CNNs

the feature extractor: we group our convolutional layers, activations (ReLu), and pooling layers into a sequence.

the classifier: after extracting features using convolutions, we use standard linear (dense) layers to make the final classification.

the flattening problem: convolutional outputs are 3d tensors (channels × height × width). linear layers expect a 1d vector.

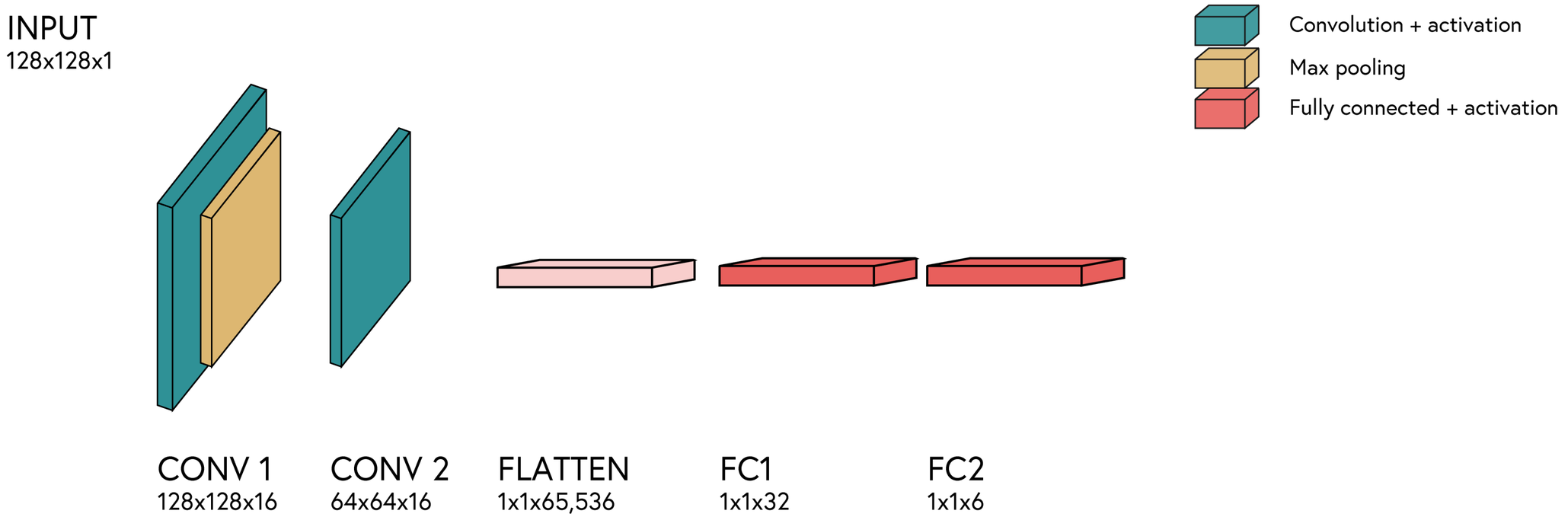


CNNs

the feature extractor: we group our convolutional layers, activations (ReLu), and pooling layers into a sequence.

the classifier: after extracting features using convolutions, we use standard linear (dense) layers to make the final classification.

the flattening problem: convolutional outputs are 3d tensors (channels × height × width). linear layers expect a 1d vector.



CNNs

```
import torch.nn as nn

class SimpleDefectCNN(nn.Module):
    def __init__(self):
        super().__init__()
        # Block 1: Conv -> ReLU -> MaxPool
        self.conv1 = nn.Conv2d(in_channels=1,
                               out_channels=16,
                               kernel_size=3, padding=1)
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)

        # Block 2: Conv -> ReLU -> MaxPool
        self.conv2 = nn.Conv2d(in_channels=16,
                               out_channels=32, kernel_size=3,
                               padding=1)
        self.relu2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
```

```
# Continuing the __init__ method...

        # Calculate flattened size: 32 channels * 50 width * 50 height
        self.flattened_size = 32 * 50 * 50

        # Fully Connected (Linear) Layers
        self.fc1 = nn.Linear(in_features=self.flattened_size,
                             out_features=128)
        self.relu3 = nn.ReLU()

        # Output layer: 6 classes for the NEU-CLS dataset
        self.fc2 = nn.Linear(in_features=128, out_features=6)
```

CNNs

```
def forward(self, x):
        # Pass through Block 1
        x = self.conv1(x)
        x = self.relu1(x)
        x = self.pool1(x)

        # Pass through Block 2
        x = self.conv2(x)
        x = self.relu2(x)
        x = self.pool2(x)

        # FLATTEN the tensor
        x = torch.flatten(x, 1) # Flattens all dimensions except batch

        # Pass through Classifier
        x = self.fc1(x)
        x = self.relu3(x)
        x = self.fc2(x) # Outputs raw logits for 6 classes

        return x
```

the data flow: the forward method defines the exact sequence in which the input tensor (x) passes through the layers we defined in `__init__`.

the flatten operation: we use `torch.flatten(x, 1)` or `x.view()` to physically reshape the tensor before passing it to the linear layers. the 1 means we flatten everything except the batch dimension.

no softmax (yet): in pytorch, we usually output raw logits (unnormalized scores) because the standard loss function (`nn.crossentropyloss`) applies softmax internally for numerical stability.

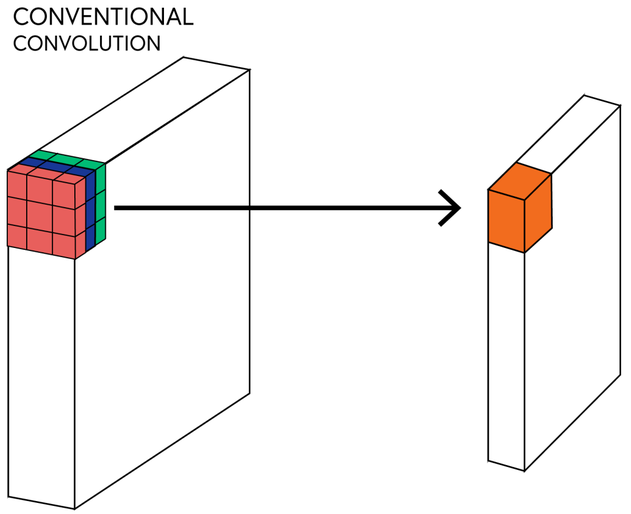
BOTTLENECK

the dual task: standard convolutions do two things simultaneously:

- extract spatial features (looking at neighboring pixels).
- combine channels (mixing rgb or feature map channels).

the cost: because every filter interacts with every input channel across a spatial grid, the number of mathematical operations (multiply-accumulate operations) explodes as networks get deeper.

the industrial reality: in a real factory setting, computing power is often limited. we need models that can run in real-time on edge devices (like a camera over a conveyor belt) without needing a massive, expensive gpu server.



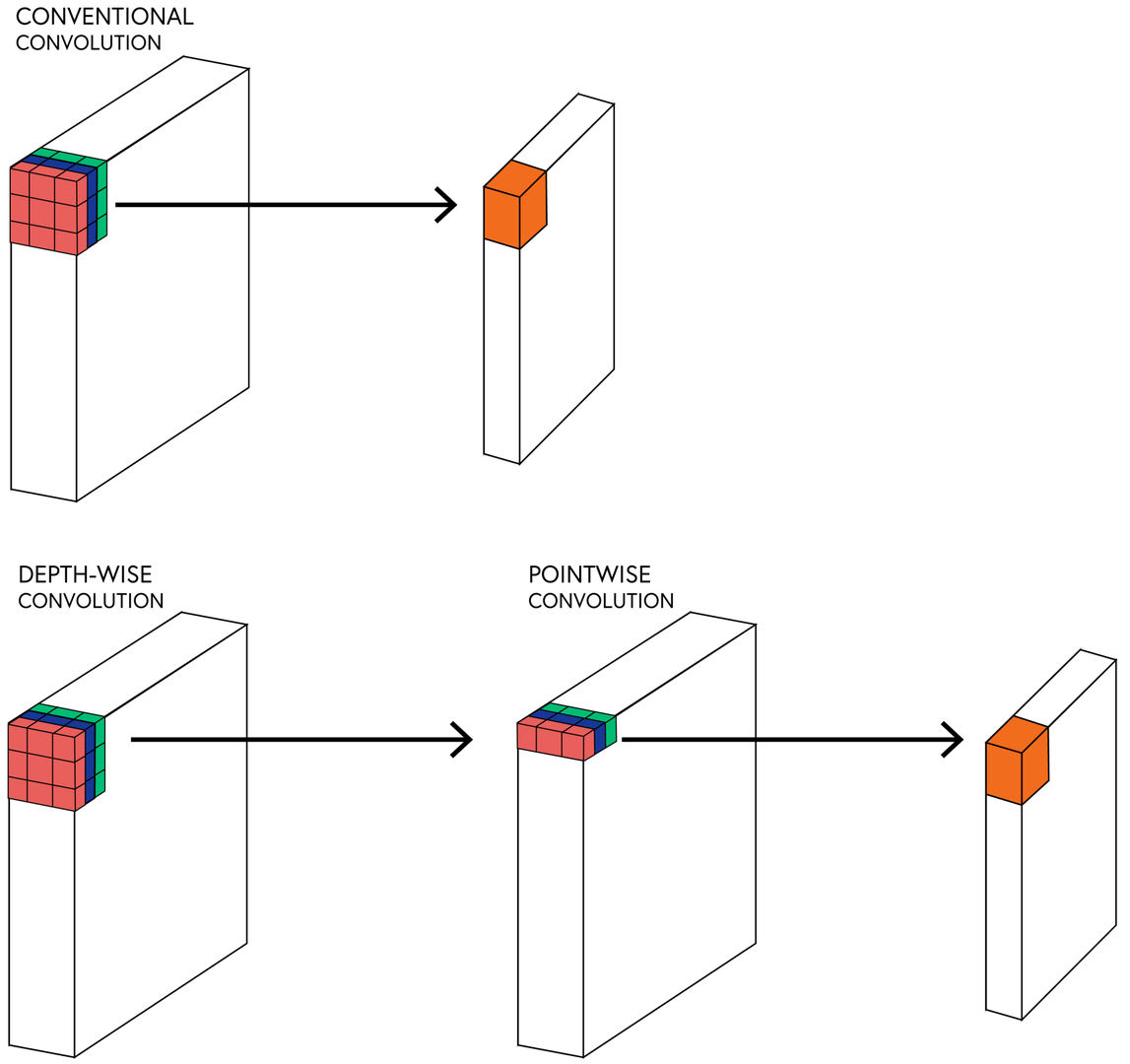
DEPTHWISE CONVOLUTION

we break the standard convolution into two separate, highly efficient steps.

- **step 1:** depthwise convolution
applies a single spatial filter to each input channel separately. it only looks at spatial patterns, not across channels.

pytorch trick: achieved by setting groups = in_channels.

- **step 2:** pointwise convolution
a standard 1 x 1 convolution. it only looks at a single pixel but mixes the data across all channels.



DEPTHWISE CONVOLUTION

```python
import torch.nn as nn

class DepthwiseSeparableConv(nn.Module):
    def __init__(self, in_channels, out_channels):
        super().__init__()

        # 1. Depthwise: groups=in_channels means 1 filter per channel
        self.depthwise = nn.Conv2d(in_channels,
                                   in_channels, kernel_size=3,
                                   padding=1, groups=in_channels)

        # 2. Pointwise: 1x1 kernel to combine channels
        self.pointwise = nn.Conv2d(in_channels,
                                   out_channels, kernel_size=1)

    def forward(self, x):
        x = self.depthwise(x)
        x = self.pointwise(x)
        return x
```

MOBILENET

mobilenet is a famous architecture built almost entirely out of these depthwise separable convolution blocks.

the impact: it drastically reduces the number of parameters compared to standard architectures (like vgg or resnet). it runs significantly faster while maintaining highly competitive accuracy.