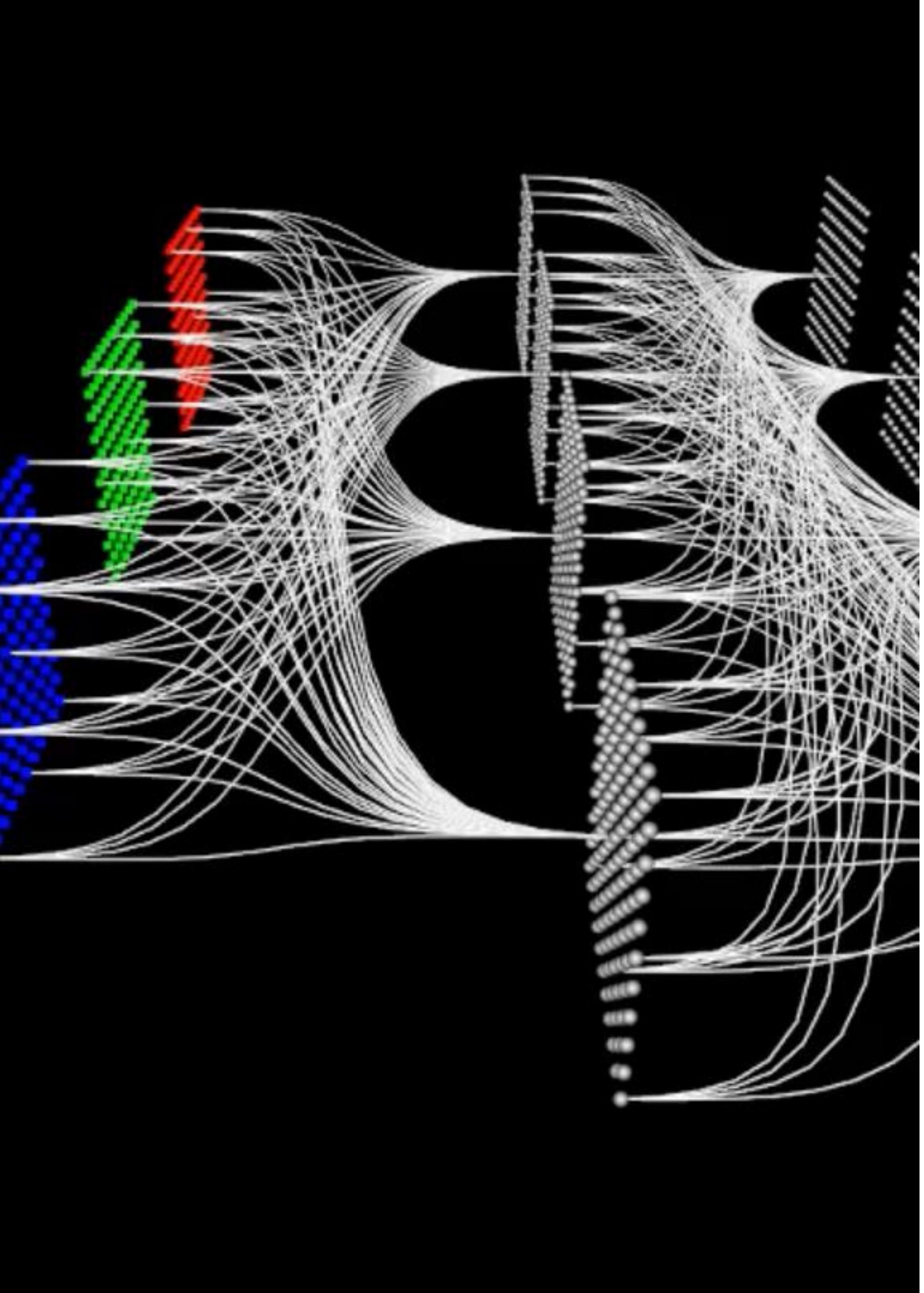
SESSION // 03
TRAINING NEURAL NETWORKS

FACULTY OF
SCIENCE AND ENGINEERING

+++

MANCHESTER 1824
The University of Manchester

Diego Corona Lopez – AI Technical Specialist



AGENDA

Introduction to Convolutional Neural Networks (CNNs)

- Understanding what makes CNNs unique for image processing

The Convolution Operation

- Mathematical foundations of convolution
- Key parameters: kernel size, stride, padding, dilation

Filters in CNNs

- Understanding what filters detect in images
- Different types of filters for edge detection, sharpening, etc.

Image Data Preparation

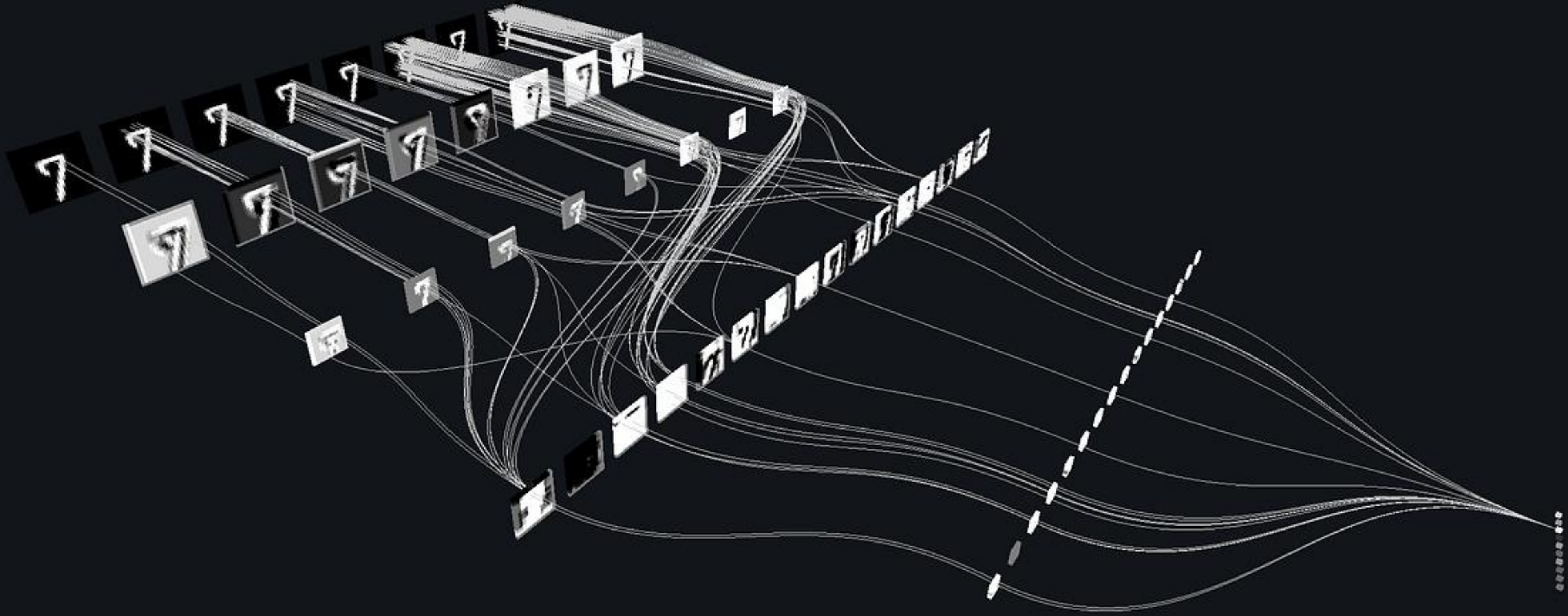
Data Loading for Deep Learning

Simple CNN Implementation

Advanced CNN Components

Case Study: Crack Detection in Historical Buildings

Popular CNN Architectures



CNN

Convolutional Neural Networks (CNNs) are specialized neural networks designed for processing structured grid-like data, such as images

- Inspired by visual cortex organization
- Revolutionized computer vision

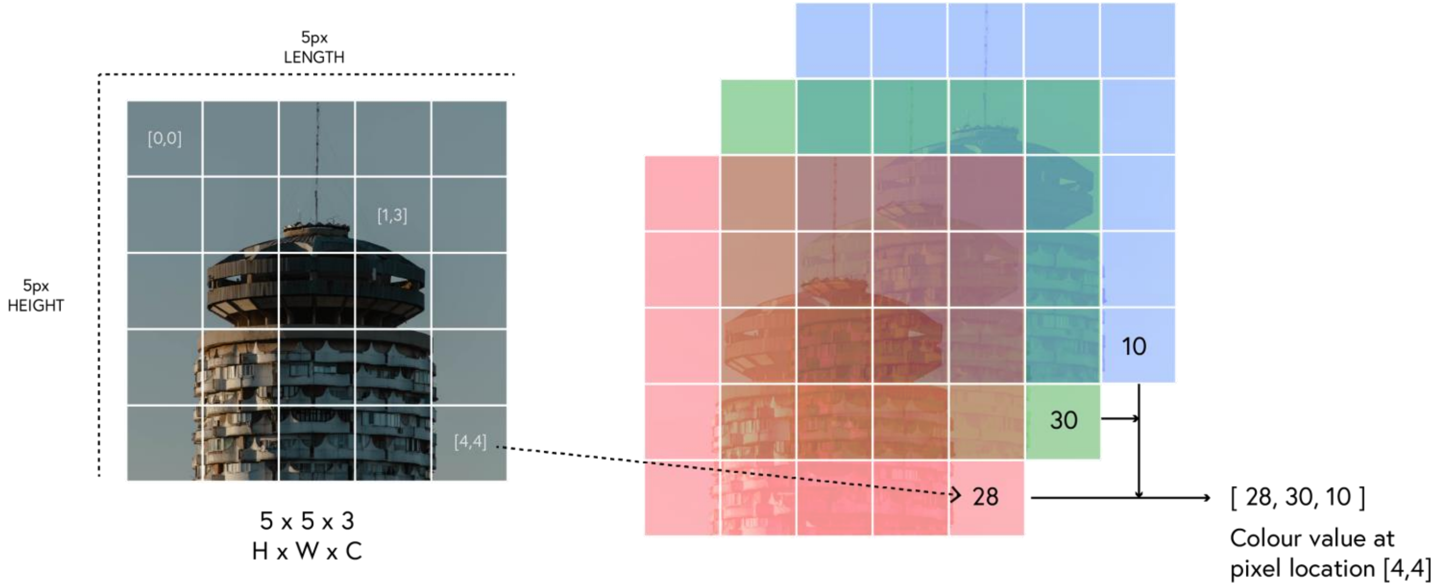
WHY STANDARD NEURAL NETWORKS STRUGGLE WITH IMAGES

Spatial Relationships: Standard networks don't account for spatial relationships between pixels

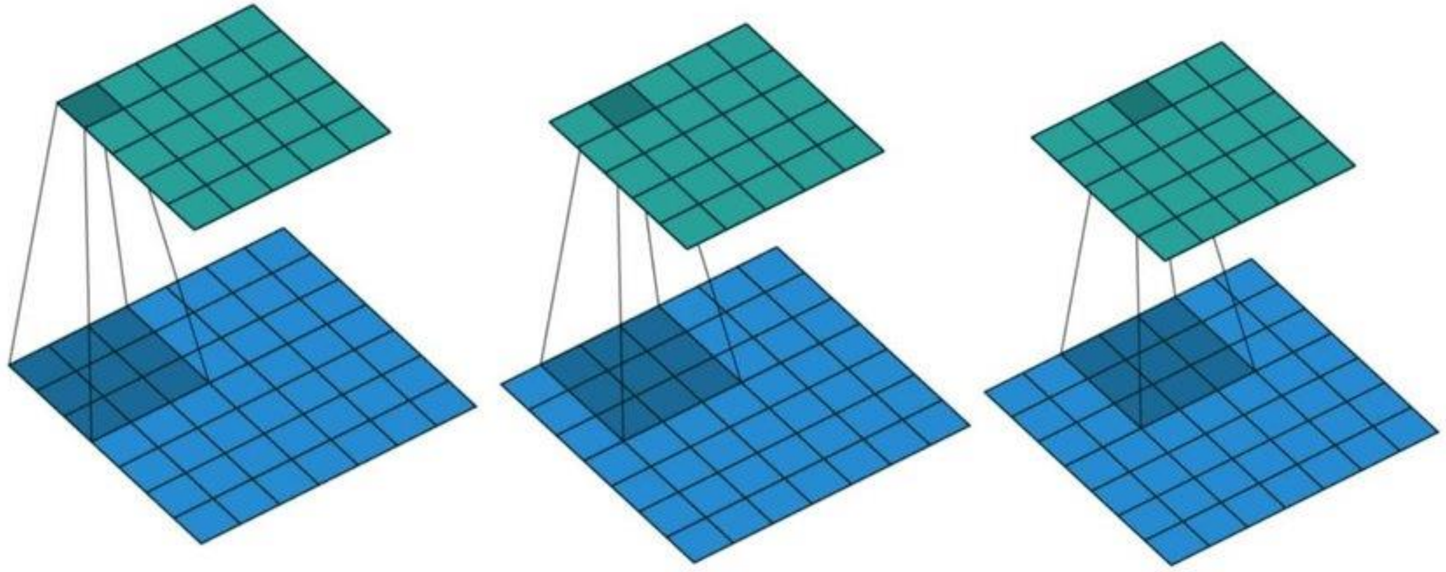
Parameter Explosion: A 224×224×3 image would require over 150,000 weights per neuron

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: A mathematical operation that slides a filter over an input, performing element-wise multiplication and summation

KEY PARAMETERS IN CONVOLUTION

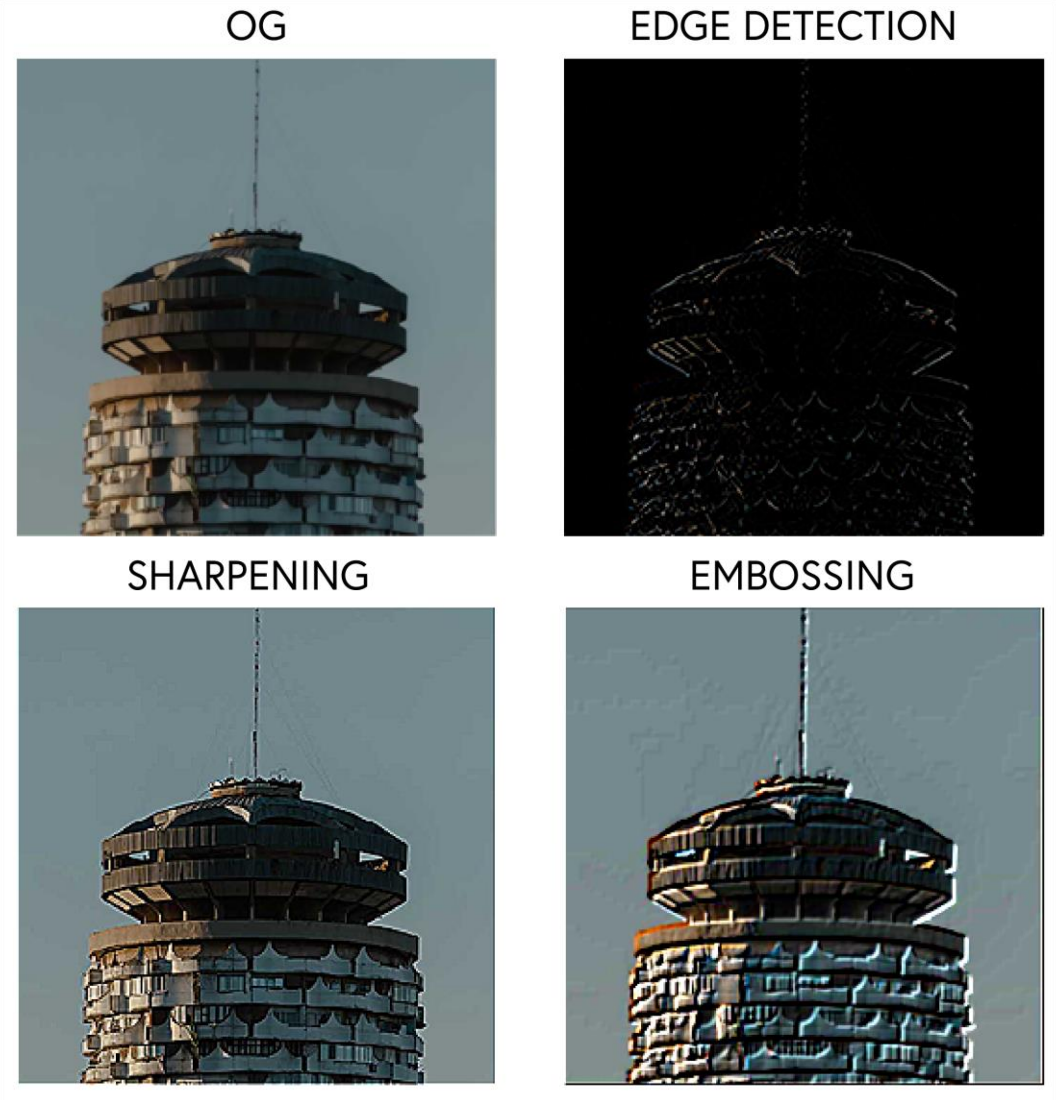
- **Kernel Size:** The dimensions of the filter
- **Stride:** How many pixels the filter shifts
- **Padding:** Adding extra pixels around the border
- **Dilation:** Spacing between kernel elements

```python
def calculate_output_size(input_height, input_width,
                          kernel_size, stride=1, padding=0):
    output_height = int((input_height - kernel_size + 2 * padding) / stride + 1)
    output_width = int((input_width - kernel_size + 2 * padding) / stride + 1)
    return output_height, output_width
```

FILTERS IN CNNS

Filters are small matrices that detect specific patterns in images

- Different filters detect different features (edges, textures, etc.)
- Weights in filters are learned during training

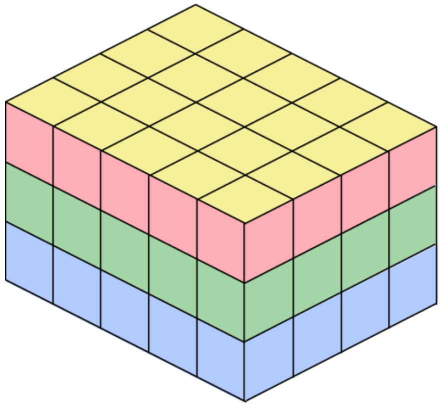


PREPARING IMAGES

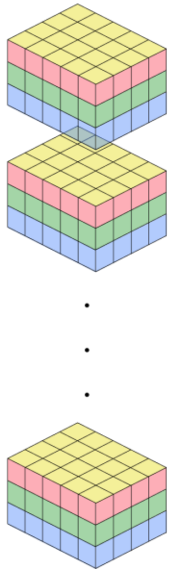
Need to convert images into proper format for CNNs

- **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]

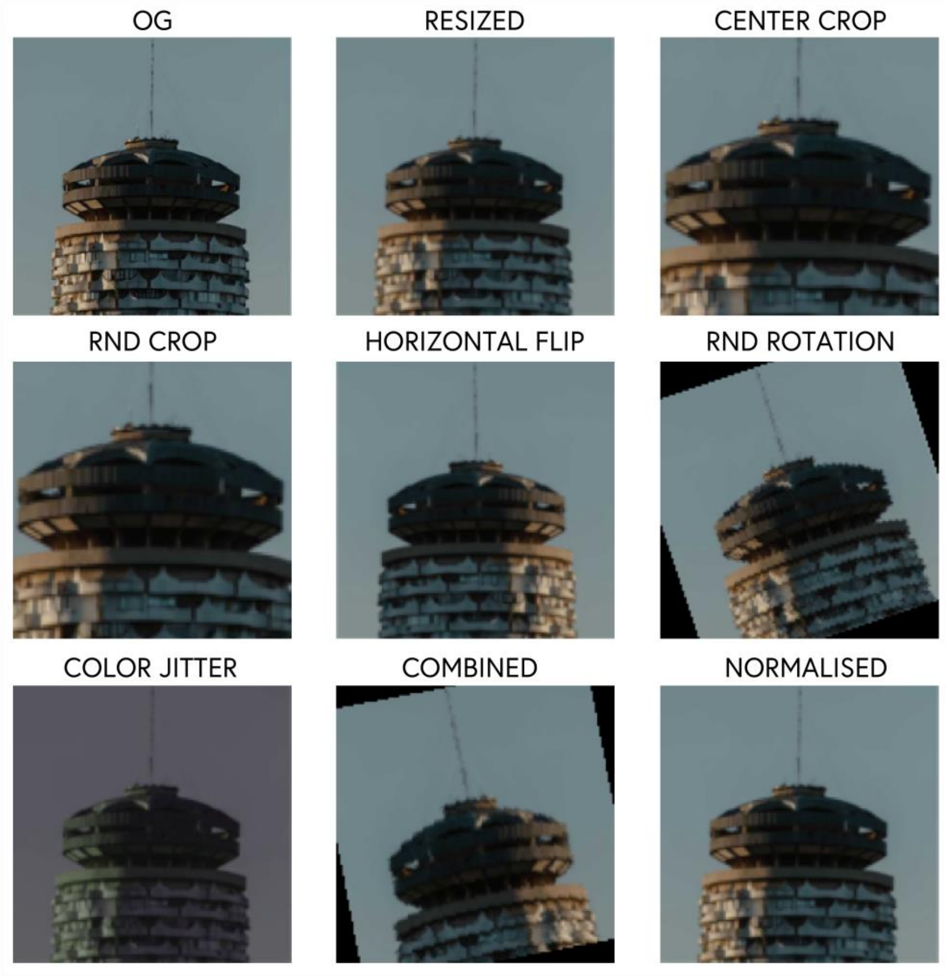
[B x H x W x C]

IMAGE TRANSFORMATIONS AND AUGMENTATION

Artificially expanding dataset by applying transformations

Benefits:

- Prevents overfitting
- Improves model generalization
- Handles varied real-world conditions
- Addresses class imbalance



AUGMENTATION TECHNIQUES

- 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

```
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])
])
```

DATASET ORGANISATION

```
datasets/
├── raw/                   # Original unprocessed data
├── processed/             # Preprocessed data ready for training
└── project_datasets/      # Project-specific organized datasets
    ├── crack_detection/   # Our current project
    │   ├── train/
    │   │   ├── crack/
    │   │   └── no_crack/
    │   ├── validation/
    │   │   ├── crack/
    │   │   └── no_crack/
    │   └── test/
    │       ├── crack/
    │       └── no_crack/
    └── future_project/
        └── ...
```

For CNN projects, proper dataset organization is crucial. A well-structured dataset allows for:

- Data Splitting Strategies
- Train/Validation/Test Split
- Stratified Splitting: Ensures class distribution is maintained across splits
- Cross-Validation: For smaller datasets or when maximum data usage is needed

IMAGEFOLDER: SMART DATASET MANAGEMENT

```python
from torchvision.datasets import ImageFolder
from torchvision import transforms

# Define transformations
transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

# Automatically load images and assign labels based on folder structure
train_dataset = ImageFolder(root="datasets/crack_detection/train",
transform=transform)

# Access class information
print(f"Classes: {train_dataset.classes}")        # ['crack', 'no_crack']
print(f"Class to index: {train_dataset.class_to_idx}")  # {'crack': 0, 'no_crack':
1}
```

EFFICIENT BATCH PROCESSING

```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
```

THE HISTORICAL CRACK DATASET: PRESERVING OUR HERITAGE

Dataset Overview

- 757 crack images / 3,139 non-crack images
- First dataset specifically for historical building monitoring
- Captures unique patterns in traditional materials

Why It Matters

- Manual inspection is time-consuming, costly, and error-prone
- Historical buildings require specialized monitoring approaches
- Early crack detection can prevent catastrophic structural failure
- AI solutions can scale inspection across multiple heritage sites

Technical Applications

- Automated drone surveys for continuous monitoring
- Mobile applications for conservation specialists
- Detection of early-stage deterioration before visible to human eye

SIMPLE CNN

CNN Components:

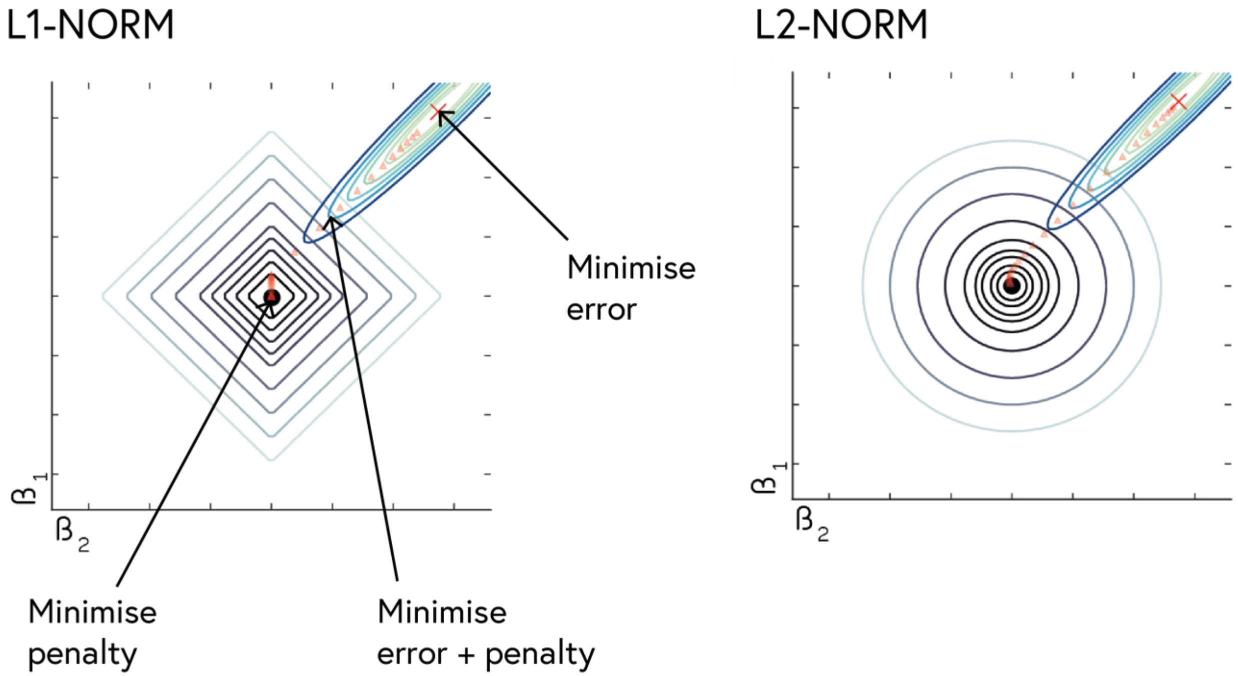
- Conv2D
- ReLU
- Fully connected layers
- Information flow through the network
- Parameter sharing and local connectivity

```python
class simpleCNN(torch.nn.Module):
    def __init__(self, n_classes):
        super(simpleCNN, self).__init__()
        self.conv1 = torch.nn.Conv2d(3, 16, kernel_size=3,
                                     stride=1, padding=1)
        self.fc1 = torch.nn.Linear(16 * 64 * 64, 16)
        self.fc2 = torch.nn.Linear(16, n_classes)

    def forward(self, x):
        x = torch.nn.functional.relu(self.conv1(x))
        x = x.view(-1, 16 * 64 * 64)
        x = torch.nn.functional.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

REGULARISATION TECHNIQUES

- **Dropout:** Prevents co-adaptation of neurons
- **Batch normalization:** Stabilises and accelerates training
- **Weight decay:** Penalises large weights
- **Early stopping:** Prevents overfitting

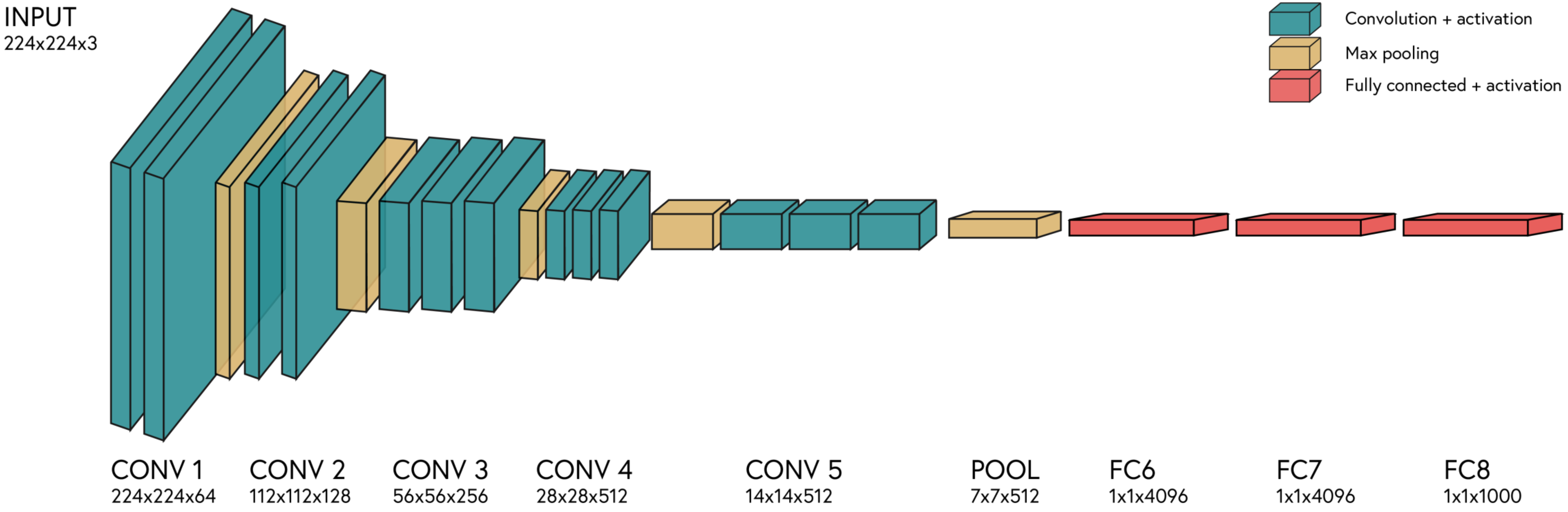


VGG16

Developed by Visual Geometry Group (Oxford, 2014)

Revolutionary simple yet effective design principles:

- Very deep network (16 weight layers)
- Small 3×3 convolution filters throughout
- Consistent doubling of filter count (64→128→256→512)
- Max pooling for dimension reduction



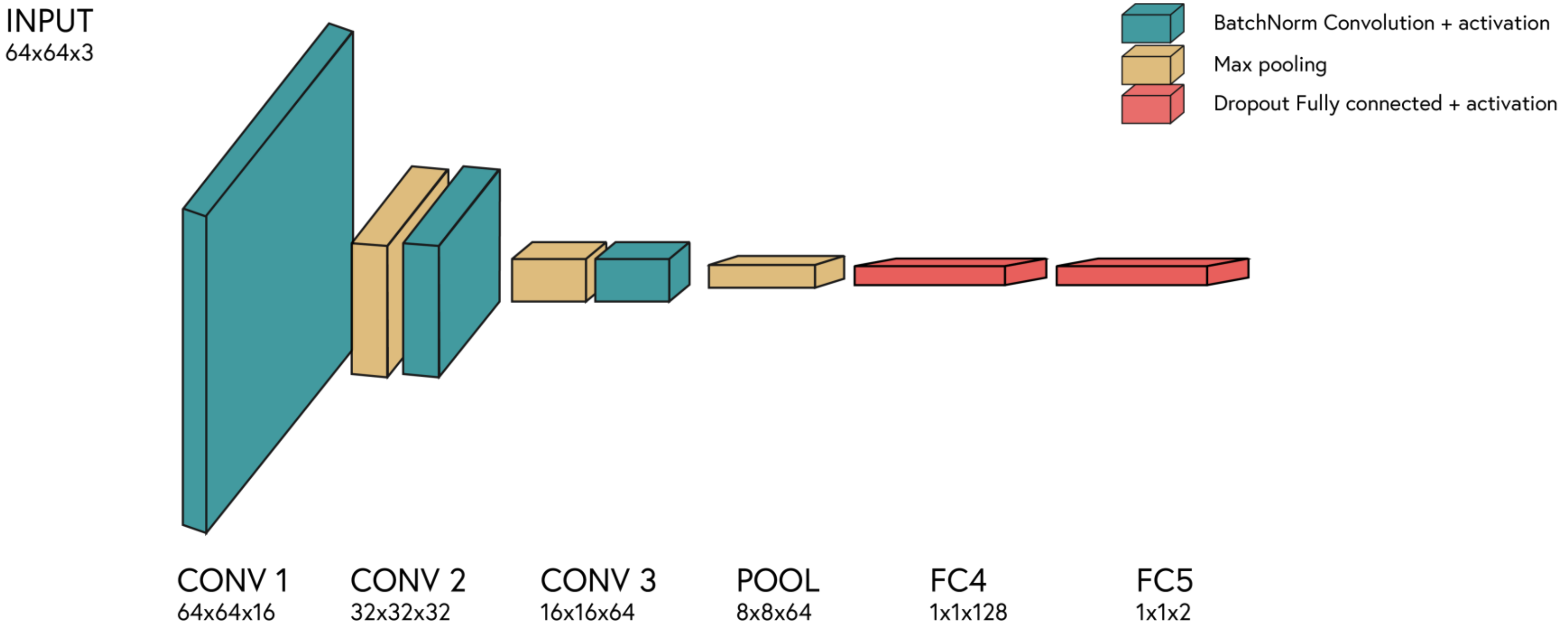
TINY VGG

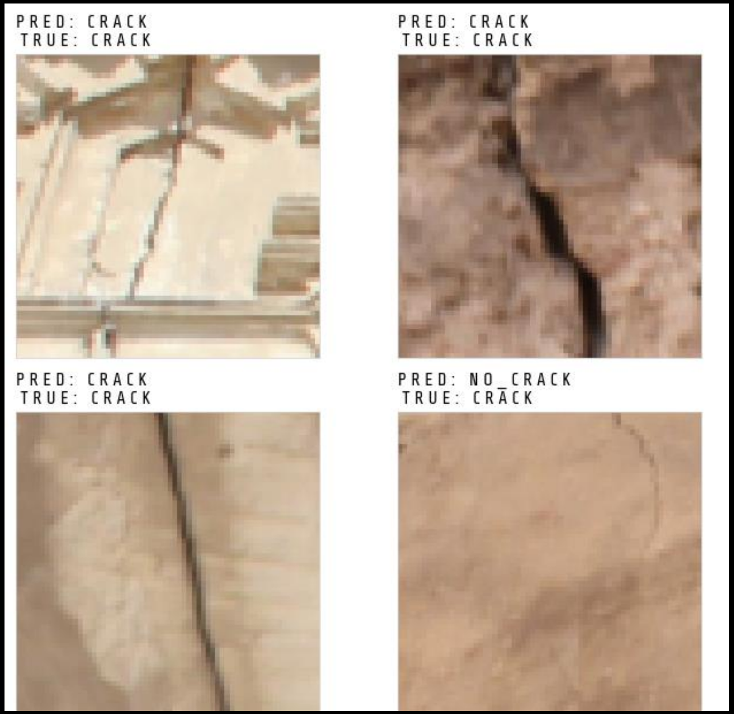
Simplified version of VGG with 3 convolutional blocks

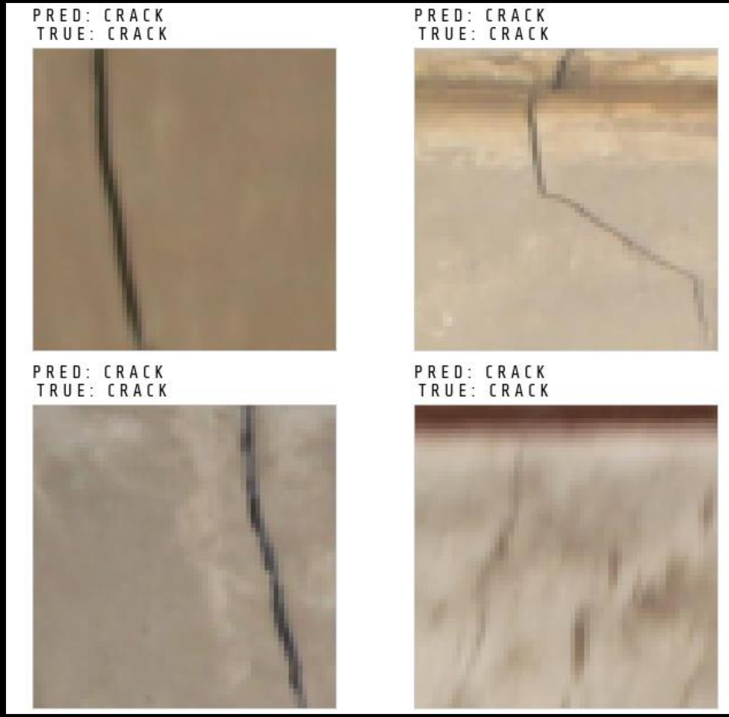
- Input → Conv → BatchNorm → ReLU → Pool → (repeat 3x) → FC → Output

Regularization:

- BatchNorm + Dropout (p=0.1) to prevent overfitting
- Progressive Feature Maps: 3→16→32→64 channels
- Dimensionality Reduction: Using pooling to reduce spatial dimensions
- Consistent Pattern: Conv → BatchNorm → ReLU → Pool at each level







BEST PRACTICES FOR CNNS

- Start with simple architectures and gradually increase complexity
- Use appropriate regularization techniques
- Apply proper data augmentation
- Monitor training with validation metrics
- Practice transfer learning when possible