Image processing

🧠 1. Feature Engineering for Image Data

🔷 What It Means

Feature engineering in the context of image data involves converting raw pixel values into a form that machine learning algorithms can use effectively — typically vectors or tensors.

🔷 Why It's Needed

Machine learning models cannot process raw image files (like `.jpg`, `.png`) directly. Instead, these images must be **numerically encoded** to represent spatial and visual information (colors, shapes, textures, etc.).

🔷 Key Concepts

- **Image → Tensor**: An image is transformed into a multidimensional array (tensor) — e.g., a color image becomes a 3D array (height × width × channels).
- **Feature Vector**: You can flatten or reduce these tensors into 1D arrays for algorithms like k-means or SVM, though CNNs work directly with tensors.

🌈 2. Handling Colour Images

🔷 RGB Color Channels

Color images use the **RGB color model**, which includes:

- **Red Channel**
- **Green Channel**
- **Blue Channel**

Each channel is a **2D matrix** of pixel intensity values between 0 and 255 (for 8-bit images).

🔷 Image Shape

For example, an image of size 32×32 will have:

`Shape: (32, 32, 3)`

Where:

- `32` is the height
- `32` is the width
- `3` is the number of color channels (R, G, B)

🔷 Grayscale Conversion

Sometimes color images are converted to **grayscale** by averaging or using weighted sums of the RGB channels to simplify the feature extraction process.

🔍 3. Incorporating Local Context Using Convolutions

🔷 What is a Convolution?

A **convolution** is a mathematical operation where a small matrix called a **filter** or **kernel** slides across the image to compute **dot products** with local patches.

For a 3×3 filter applied to a 5×5 image patch:

```
[1 0 -1]      [img11 img12 img13]
[1 0 -1]   *   [img21 img22 img23]
[1 0 -1]      [img31 img32 img33]
```

🔷 Why Use Convolutions?

- Preserve **spatial locality** — neighboring pixels often share semantic meaning.
- Extract **meaningful patterns** — edges, corners, textures.
- Mimic **human visual perception** where localized features are crucial.

🔷 Filter Types

Filters (kernels) can be:

- **Smoothing** (e.g., mean filter: all ones divided by size)
- **Edge detection** (e.g., Sobel, Prewitt)

- **Sharpening** (emphasize borders)

🔷 Output

After applying a filter:

- You get a **new image** (convolved feature map).
- Each value represents a **local pattern match** between the filter and the input image patch.

🔷 Stacking Feature Maps

You can apply multiple filters to get a **stack of feature maps** — each highlighting a different type of pattern. These form the input to deeper layers in CNNs or can be flattened into feature vectors for classical ML.

🔧 1. Basic Methods of Image Feature Extraction

🟥 A. Raw Pixel Intensities

- **Approach**: Flatten the 2D (grayscale) or 3D (RGB) image into a 1D vector.
- **Example**:
  - Grayscale image 28×28 → vector of size 784
  - RGB image 32×32×3 → vector of size 3072

```
import numpy as np
image = ... # a 32×32×3 image
features = image.flatten()  # shape becomes (3072,)
```

- ✅ **Pros**: Simple and quick
- ❌ **Cons**: Ignores spatial relationships and patterns

🟧 B. Statistical Features

- **Histogram of pixel values**: Captures intensity distribution.
- **Mean, Variance, Skewness** of each channel.
- Often used in medical or texture-based image analysis.

```
mean = np.mean(image, axis=(0, 1))
std = np.std(image, axis=(0, 1))
```

🟨 C. Convolution-based Features (Manual Filters)

- Use 3×3, 5×5, etc., **kernels** to extract local features like edges, corners, textures.
- Apply convolution → flatten the result → use as features.

```
from scipy.signal import convolve2d

kernel = np.ones((3, 3)) / 9  # simple blur filter
convolved = convolve2d(gray_image, kernel, mode='valid')
features = convolved.flatten()
```

- This simulates part of what CNNs do automatically.

⚙️ 2. Intermediate Techniques

🟩 A. PCA (Principal Component Analysis)

- Reduces dimensionality while retaining variance.
- Converts flattened image vectors into a lower-dimensional space.

```
from sklearn.decomposition import PCA

pca = PCA(n_components=50)
features = pca.fit_transform(flat_images)
```

🟦 B. Histogram of Oriented Gradients (HOG)

- Captures edge orientations.
- Common in object detection (e.g., pedestrian detection).

```
from skimage.feature import hog
```

```
features = hog(image, pixels_per_cell=(8, 8), cells_per_block=(2, 2), multichannel=True)
```

🤖 3. Advanced Methods

🟪 A. Pre-trained CNN Feature Extractors

- Use models like **ResNet**, **VGG**, or **MobileNet** pretrained on ImageNet.
- Remove the classification layer and extract from a hidden layer.

```
from tensorflow.keras.applications import VGG16
from tensorflow.keras.applications.vgg16 import preprocess_input
from tensorflow.keras.models import Model

model = VGG16(weights='imagenet', include_top=False, input_shape=(224,224,3))
feature_extractor = Model(inputs=model.input, outputs=model.output)

features = feature_extractor.predict(preprocess_input(image_batch))
flattened_features = features.reshape(features.shape[0], -1)
```

- ✅ Captures very rich hierarchical features
- ✅ Highly effective for transfer learning and clustering

🟫 B. Autoencoders

- Train an encoder-decoder network to reconstruct the image.
- Use the bottleneck layer as the feature vector.

```
# encoder_output = encoder.predict(image_batch)
```

✅ Best Practices

- **Normalize pixel values**: Scale between 0 and 1 or use standardization (mean=0, std=1)
- **Resize images**: Ensure consistent shape across dataset

- **Augmentation**: For better generalization, augment with flips, rotations, etc.

📌 Summary Table

Method	Feature Type	Complexity	Captures Spatial Info
Raw pixels	Flat vector	Low	❌ No
Manual convolutions	Local patterns	Low-Medium	✅ Some
PCA	Reduced linear components	Medium	❌ Mostly global
HOG	Edge orientations	Medium	✅ Yes
Pre-trained CNNs	Deep hierarchical features	High	✅ Strong
Autoencoders	Latent representations	High	✅ Strong

📉 1. Downsizing Images

🔷 What It Means

Downsizing an image refers to **reducing its resolution** — i.e., making it smaller in terms of width and height (fewer pixels).

🔷 Why It's Done

- **Reduce computational cost**: Smaller images require less memory and processing.
- **Speed up training**: Especially in deep learning where input size significantly affects performance.
- **Standardize input**: Most ML models and CNN architectures require fixed-size input (e.g., 224×224 for VGG).

🔷 Example

Original size: 256×256

Downsized to: 64×64

Each image goes from 65,536 pixels → 4,096 pixels.

🔷 How It's Done

Using `cv2` (OpenCV) or `PIL`:

```
import cv2
resized = cv2.resize(image, (64, 64))
```

🔷 Trade-offs

- ❗ **Information loss**: Important details like edges or fine textures may be lost.
- ✅ Best to keep enough resolution to preserve features relevant to the task.

🌈 2. Downsizing Color (Reducing Color Depth or Channels)

🔷 What It Means

There are **two main interpretations**:

A. Reducing Color Depth (Bit Depth)

- Reducing the number of bits used to represent color intensity.
- From 8 bits/channel (256 values) → 4 bits/channel (16 values), etc.

This reduces file size and can simplify features but may lead to visible color banding.

B. Converting Color Images to Grayscale

- Reducing from **3 channels (RGB)** to **1 channel (Grayscale)**.
- Common when color information isn't essential.
  `gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)`
- RGB image shape: (32, 32, 3)
- Grayscale image shape: (32, 32)

🔷 Why It's Done

- **Simplifies models**: Less input data.
- **Speeds up processing**.

- **Avoids unnecessary complexity** if color isn't needed (e.g., digit recognition).

🔷 Trade-offs

- ❗ Color-based features are lost, which might be crucial in tasks like traffic light detection or object classification in natural images.

✅ Summary

Aspect	Downsizing Images	Downsizing Color
What it reduces	Spatial resolution	Color information
Input dimension	From H×W to smaller H×W	From 3 channels → 1 channel
Use case	Efficiency, standardization	When color is non-essential
Trade-off	Blurry/loss of detail	Loss of color cues
