Minor in AI

Image Processing Fundamentals

Original RGB Image

1 The Magic Behind Computer Vision

Imagine a self-driving car detecting pedestrians, or your phone unlocking with face recognition. These marvels begin with understanding **how computers see images**. Unlike humans who perceive colors and shapes naturally, computers need special instructions to interpret pixel patterns.

When security cameras detect suspicious activity, they don't "see" like humans. Instead, they analyze pixel intensity changes through edge detection.

2 Understanding Digital Images: From Pixels to Intelligence

2.1 The Building Blocks of Vision

Every digital image is composed of **pixels** - tiny color dots arranged in a grid. Think of them as microscopic mosaic tiles.

Key Properties:

- **Resolution:** Total pixels (e.g., 1920×1080 = 2M pixels)
- **Color Depth:** 8-bit = 256 shades per channel (0-255)
- **Channels:** RGB (Red, Green, Blue) layers combine to create full color

Real-World Analogy

A pixel grid is like a stadium LED display: each light (pixel) has red, green, and blue bulbs (channels) that combine to create any color.

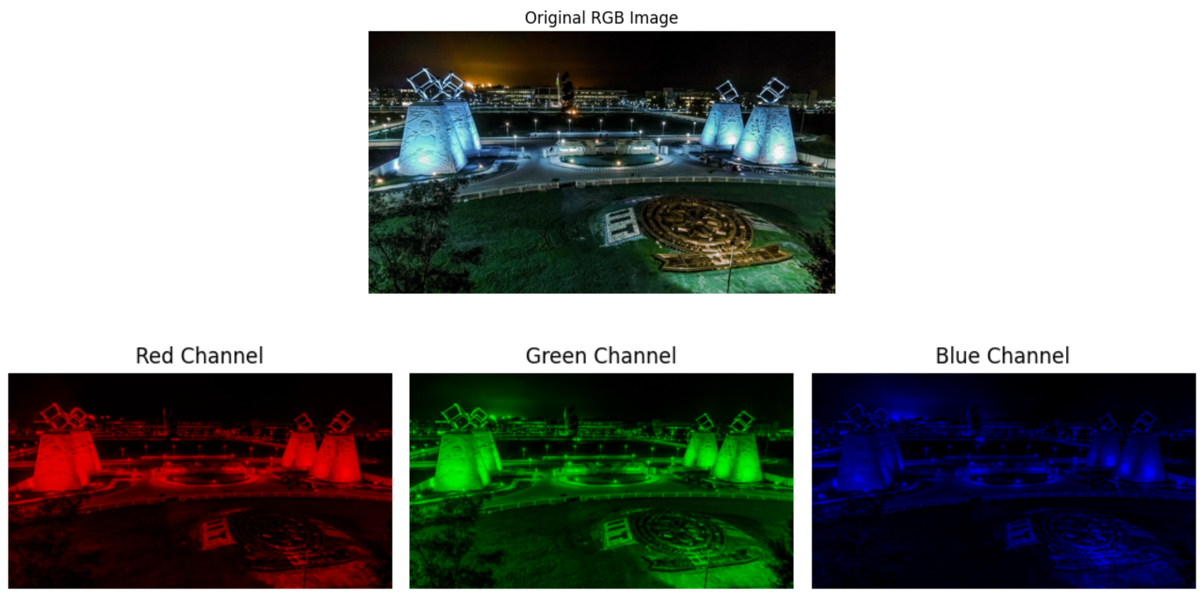
2.2 The RGB/BGR Conundrum

Digital cameras and displays use different color order conventions:

```python
import cv2

# Most cameras capture in BGR format
bgr_image = cv2.imread('photo.jpg')

# Convert for proper display
```



```python
rgb_image = cv2.cvtColor(bgr_image, cv2.COLOR_BGR2RGB)

# Split into individual color components
red_component = rgb_image[:, :, 0]  # Red channel matrix
green_component = rgb_image[:, :, 1] # Green channel matrix
blue_component = rgb_image[:, :, 2] # Blue channel matrix
```

Listing 1: Color Space Conversion

Why This Matters:

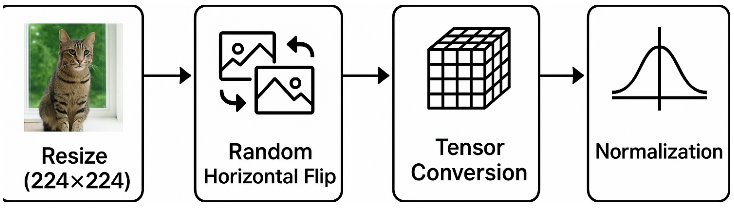
- Red Channel: Enhances text recognition (blood vessels in retinal scans)
- Green Channel: Carries luminance details (key for edge detection)
- Blue Channel: Useful in underwater imaging (water absorbs red/green)

3 Image Preprocessing: Preparing Data for AI

3.1 The Standardization Pipeline

Neural networks require consistent input formats. Our 4-step process:

1. **Resizing (224×224):**
   - Common size for modern CNNs (e.g., ResNet, VGG)
   - Aspect ratio maintained through intelligent cropping
   - Reduces computational complexity exponentially
2. **Random Horizontal Flip:**
   - Data augmentation technique
   - Teaches orientation invariance (crucial for object detection)



- Probability (p=0.5) balances variation and authenticity

3. **Tensor Conversion:**
   - Converts image to 3D matrix: (Channels $\times$ Height $\times$ Width)
   - Enables GPU acceleration and matrix operations

4. **Normalization:**
   - Scales pixel values from [0,255] to [-1,1] range
   - Formula: Normalized $= \frac{\text{pixel}-127.5}{127.5}$
   - Stabilizes neural network training

```python
from torchvision import transforms

transform = transforms.Compose([
    transforms.Resize((224, 224)),  # Standard input size
    transforms.RandomHorizontalFlip(p=0.5),  # 50% flip chance
    transforms.ToTensor(),  # Convert to (C,H,W) format
    transforms.Normalize(mean=[0.485, 0.456, 0.406],  # ImageNet stats
                         std=[0.229, 0.224, 0.225])
])
```

Listing 2: Complete Preprocessing Pipeline

4 Convolution: The Heart of Computer Vision

4.1 Edge Detection Fundamentals

The first step in visual understanding is identifying edges - boundaries between objects or regions.

Convolution Process:

1. Define a 3×3 **kernel** (feature detector)
2. Slide it across the image matrix
3. Calculate dot product at each position
4. Generate **feature map** highlighting patterns

```python
import torch.nn as nn

# Create convolutional layer
conv_layer = nn.Conv2d(1, 1, kernel_size=3, bias=False)

# Manual kernel for vertical edges
vertical_kernel = torch.tensor([
    [[[1, 0, -1],  # Left column positive
      [1, 0, -1],  | Right column negative
      [1, 0, -1]]] # Vertical edge detector
], dtype=torch.float32)

conv_layer.weight.data = vertical_kernel

# Sample input: Bright left, dark right
input_image = torch.tensor([[[
    [0.9, 0.9, 0.1, 0.1],
    [0.9, 0.9, 0.1, 0.1],
    [0.9, 0.9, 0.1, 0.1]
]]], dtype=torch.float32)

output = conv_layer(input_image)
```

Listing 3: Vertical Edge Detector

Interpreting Output:

- **Positive Values:** Bright-to-dark transitions (left edges)
- **Negative Values:** Dark-to-bright transitions (right edges)
- **Zero Values:** No vertical edges detected

5 Key Concepts in Practice

5.1 Case Study: Medical Imaging Analysis

Modern MRI scanners use similar preprocessing and convolution techniques:

- **Challenge:** Tumor boundaries often subtle
- **Solution:** Edge detection kernels highlight cell density changes
- **Process:**
  1. Convert 3D scans to 2D slices
  2. Apply Gaussian normalization
  3. Use multi-scale convolution kernels
  4. Aggregate edge maps for 3D reconstruction

Why This Matters

In 2023, Stanford researchers improved tumor detection accuracy by 37% using optimized convolution kernels in MRI analysis.

Learning Checklist

- Understand pixel composition and color channels
- Explain each preprocessing step's purpose
- Describe convolution operation mathematically
- Interpret feature map outputs
- Identify real-world applications