HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG

KHOA CÔNG NGHỆ THÔNG TIN

BÁO CÁO BÀI TẬP LỚN

XỬ LÝ ẢNH

ĐỀ TÀI: XÂY DỰNG MÔ HÌNH CNN NHẬN DIỆN CHỮ SỐ VIẾT TAY VÀ HÌNH HỌC CƠ BẢN

Giảng viên hướng dẫn	**TS. Phạm Hoàng Việt**
Nhóm lớp	**02**
Nhóm bài tập lớn	**22**
Sinh viên thực hiện	**Đào Đức Duy - B22DCCN145** **Bùi Công Bắc - B22DCCN073**

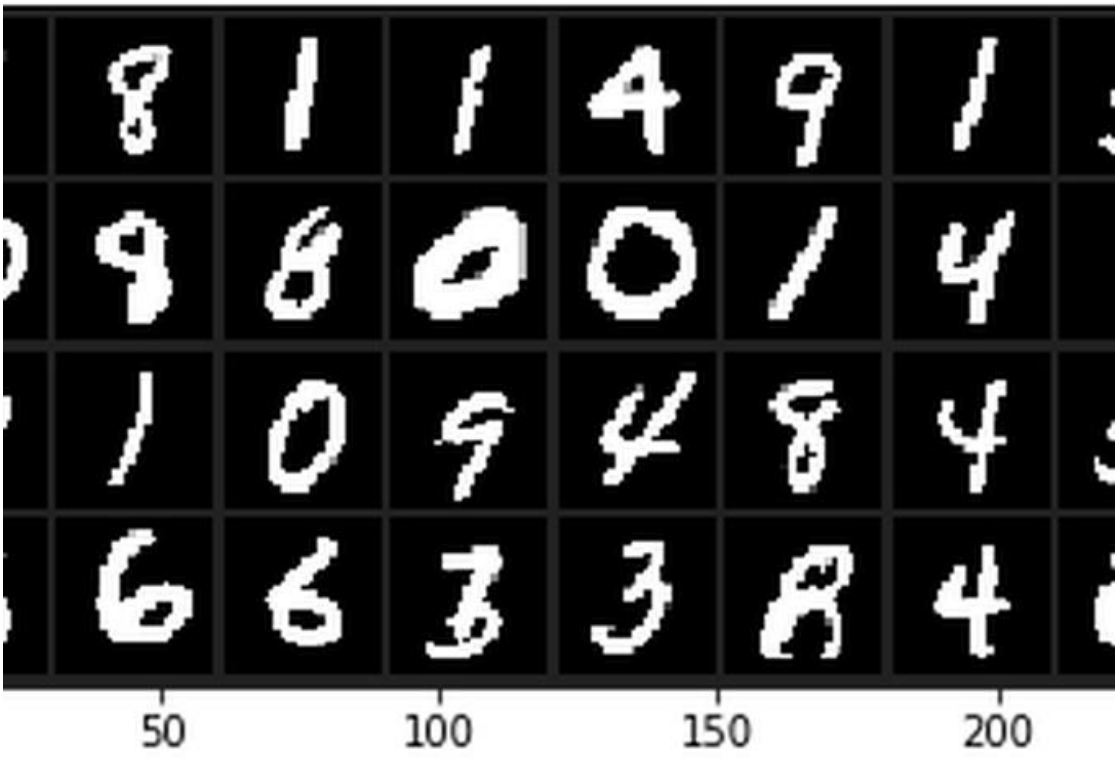
HÀ NỘI - 2025

MỤC LỤC

I. Tập dữ liệu

1. Tập dữ liệu MNIST:

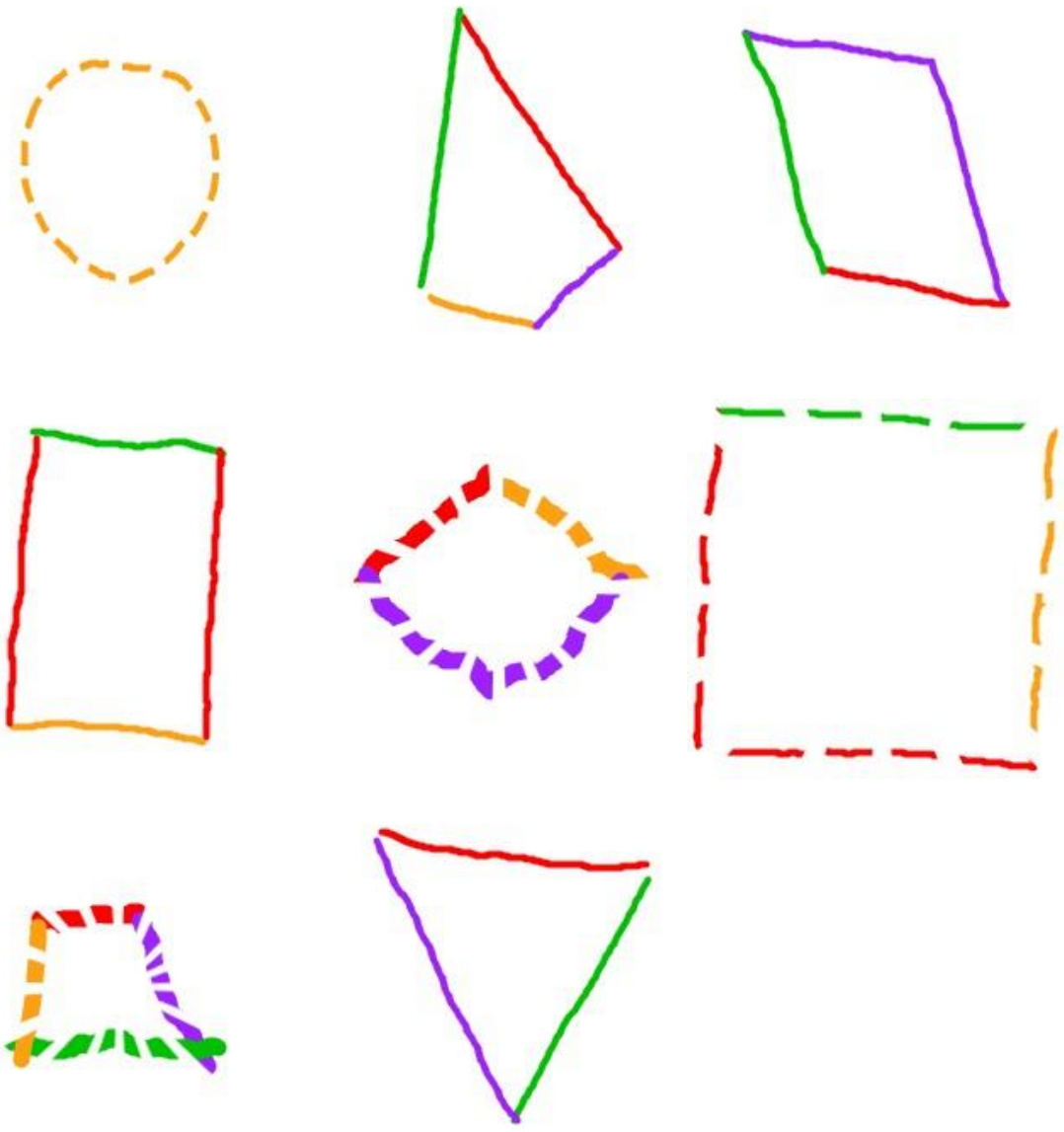
- **Nội dung:** Các chữ số viết tay từ 0 đến 9.
- **Số lượng:** Tổng cộng 70.000 ảnh.
- **Tập huấn luyện (Training set):** 60.000 ảnh.
- **Tập kiểm tra (Test set):** 10.000 ảnh.
- **Kích thước:** Mỗi ảnh có kích thước 28 x 28 pixels.
- **Màu sắc:** Ảnh xám (Grayscale)



2. Tập dữ liệu Shape

- **Không gian đầu vào (Input Shape):** 224 x 224 x 3
- **Số lượng lớp (Classes):** C = 8.
    - ***Danh sách:*** Circle, Kite, Parallelogram, Square, Rectangle, Rhombus, Trapezoid, Triangle.

- **Phân phối dữ liệu (Data Distribution):**
    - **Training set:** 12,000 mẫu (1,500 mẫu/lớp).
    - **Validation set:** 4,000 mẫu (500 mẫu/lớp).
    - **Test set:** 4,000 mẫu (500 mẫu/lớp).
    - **Tổng cộng:** 20,000 mẫu.



II. Các bước tiền xử lí dữ liệu đầu vào

1. Chuyển ảnh đầu vào thành Grayscale

- **Công thức chuẩn:** Hàm sử dụng công thức độ chói (Luminance) để phù hợp với cảm nhận của mắt người:

$$Gray = 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B$$

```python
def rgb_to_grayscale(image_array):
    """
    Chuyển ảnh RGB sang grayscale bằng công thức chuẩn
    Gray = 0.299*R + 0.587*G + 0.114*B

    Args:
        image_array: Ảnh RGB hoặc grayscale dạng numpy array

    Returns:
        Ảnh grayscale dạng numpy array (uint8)
    """
    # Handle already-grayscale
    if image_array is None:
        raise ValueError("Input image is None")

    if len(image_array.shape) == 2:
        gray = image_array
    else:
        # If image has alpha channel, drop it
        if image_array.shape[-1] == 4:
            image_array = image_array[..., :3]

        # If values are floats in [0,1], convert to 0-255
        if np.issubdtype(image_array.dtype, np.floating):
            if image_array.max() <= 1.0:
                image_array = (image_array * 255.0).astype(np.uint8)
            else:
                image_array = image_array.astype(np.uint8)

        gray = np.dot(image_array[..., :3], [0.299, 0.587, 0.114])

    # Ensure uint8 output
    gray = np.clip(gray, 0, 255)
    return gray.astype(np.uint8)
```

2. Thuật toán tìm ngưỡng tự động Otsu

Bước 1: Tính Histogram và xác suất

```
"""
hist, _ = np.histogram(gray_image.flatten(), bins=256, range=(0, 256))
hist = hist.astype(float)
hist /= hist.sum()
```

- **gray_image.flatten()**: Duỗi thẳng ma trận ảnh 2 chiều thành mảng 1 chiều. Otsu không quan tâm vị trí pixel (x, y), chỉ quan tâm giá trị của nó.

- **np.histogram**: Đếm số lượng pixel cho mỗi mức xám từ 0 đến 255.

  - Ví dụ: Có 50 pixel giá trị 0, 100 pixel giá trị 1...

- **hist /= hist.sum()**: **Chuẩn hóa (Normalization)**.

  - Thay vì đếm số lượng (Frequency), ta chuyển sang **Xác suất (Probability)** p_i
  - p_i là xác suất để một pixel ngẫu nhiên trong ảnh có giá trị bằng i.
  - Tổng của mảng hist sau bước này sẽ bằng 1.0.

Bước 2: Chuẩn bị các mảng cộng dồn

```
bins = np.arange(256)
weight1 = np.cumsum(hist)
weight2 = 1.0 - weight1
```

- Ở đây, ta giả sử đang thử **tất cả** các ngưỡng t từ 0 đến 255 cùng lúc.

  - **bins**: Mảng các giá trị pixel [0, 1, 2, ..., 255].
  - **np.cumsum(hist)**: Tính tổng tích lũy (Cumulative Sum).
    - Tại vị trí t, giá trị này chính là trọng số của lớp nền (Background Weight) $w_0(t)$.
    - $w_0(t)$ (Tổng xác suất các pixel từ 0 đến t).

- **weight2**: Trọng số của lớp vật thể (Foreground Weight) $w_1(t)$.
    - Vì tổng xác suất là 1, nên $w_1(t) = 1 - w_0(t)$.

Bước 3: Tính giá trị trung bình (Means)

```
mean1 = np.cumsum(hist * bins) / (weight1 + 1e-10)
global_mean = np.sum(hist * bins)
mean2 = (global_mean - np.cumsum(hist * bins)) / (weight2 + 1e-10)
```

- **Mục tiêu:** Tính giá trị trung bình độ sáng của 2 lớp tại mỗi ngưỡng t.

1. **hist * bins**: Tính $p_i * i$.
2. **np.cumsum(hist * bins)**: Tính tổng tích lũy $\sum_{i=0}^{t} pi * i$
    - Đây là tử số của công thức tính trung bình cộng dồn.
3. **mean1 (μ_0)**:
    - Công thức: $\mu_0(t) = \frac{\sum_{i=0}^{t} pi * i}{w_0(t)}$
    - Code: cumsum(...) / weight1.
    - + **1e-10**: Kỹ thuật **Numerical Stability**. Giúp tránh lỗi "Chia cho 0" (ZeroDivisionError) nếu tại ngưỡng t nào đó không có pixel nào (weight1 = 0).
4. **global_mean (μ_T)**: Trung bình độ sáng của toàn bộ bức ảnh.
5. **mean2 (μ_1)**:
    - Ta dùng công thức: $\mu_T = w_0\mu_0 + w_1\mu_1$
    - Suy ra: $\mu_1 = \frac{\mu_T - w_0\mu_0}{w_1}$
    - Trong code, tử số global_mean - np.cumsum(hist * bins) chính là tổng giá trị pixel của phần còn lại (từ t+1 đến 255).

Bước 4: Tính phương sai giữa các lớp

```
variance = weight1 * weight2 * (mean1 - mean2) ** 2
```

- Ta tính phương sai σ_B^2cho **tất cả 256 ngưỡng cùng lúc**.

- Công thức: $\sigma_B^2(t) = w_0(t) \cdot w_1(t) \cdot [\mu_0(t) - \mu_1(t)]^2$
- Biến variance lúc này là một mảng gồm 256 phần tử, mỗi phần tử đại diện cho độ tốt của việc chọn ngưỡng tại vị trí đó.
- Giá trị càng lớn → Sự tách biệt giữa Nền và Chữ càng rõ ràng.

Bước 5: Chọn ngưỡng tối ưu

```
threshold = np.argmax(variance)

return threshold
```

- np.argmax(variance): Tìm vị trí (index) mà tại đó giá trị trong mảng variance là lớn nhất.

- Vị trí đó chính là giá trị ngưỡng (Threshold) tối ưu mà chúng ta cần tìm.

3. Phân ngưỡng và đảo màu ảnh

```python
def apply_threshold_inverted(gray_image):
    """
    Áp dụng threshold và đảo ngược (nền đen, chữ trắng)

    Args:
        gray_image: Ảnh grayscale dạng numpy array

    Returns:
        Ảnh binary inverted (uint8)
    """
    # Compute Otsu threshold
    threshold = otsu_threshold(gray_image)

    # Candidate 1: non-inverted (foreground = bright regions)
    binary_noninv = np.zeros_like(gray_image, dtype=np.uint8)
    binary_noninv[gray_image >= threshold] = 255
    binary_noninv[gray_image < threshold] = 0

    # Candidate 2: inverted (foreground = dark regions)
    binary_inv = np.zeros_like(gray_image, dtype=np.uint8)
    binary_inv[gray_image < threshold] = 255
    binary_inv[gray_image >= threshold] = 0

    # Evaluate candidates by fraction of foreground pixels
    total_pixels = gray_image.size
    cnt_noninv = np.count_nonzero(binary_noninv)
    cnt_inv = np.count_nonzero(binary_inv)
    frac_noninv = cnt_noninv / total_pixels
    frac_inv = cnt_inv / total_pixels

    # Prefer a candidate whose foreground is small but non-zero (object occupies small a
    min_frac = 1e-4
    max_frac = 0.5

    valid_noninv = (cnt_noninv > 0) and (min_frac <= frac_noninv <= max_frac)
    valid_inv = (cnt_inv > 0) and (min_frac <= frac_inv <= max_frac)

    if valid_noninv and valid_inv:
        # Both look plausible: pick the smaller foreground (likely the object)
        chosen = binary_noninv if frac_noninv <= frac_inv else binary_inv
        return chosen
```

```
    if valid_noninv:
        return binary_noninv

    if valid_inv:
        return binary_inv

    # If neither candidate is in the desired fraction range, try mean-based threshold
    alt_thresh = int(np.mean(gray_image))
    alt_bin = np.zeros_like(gray_image, dtype=np.uint8)
    alt_bin[gray_image >= alt_thresh] = 255
    alt_bin[gray_image < alt_thresh] = 0
    if np.count_nonzero(alt_bin) > 0 and (np.count_nonzero(alt_bin) / total_pixels) <= max_frac:
        return alt_bin

    # Last resort: treat non-white pixels as foreground but try to avoid full-image masks
    mask = gray_image < 250
    fallback = np.zeros_like(gray_image, dtype=np.uint8)
    fallback[mask] = 255
    if np.count_nonzero(fallback) == 0 or (np.count_nonzero(fallback) / total_pixels) > 0.95:
        # If it's still bad, return the non-inverted candidate (safer default)
        return binary_noninv

    return fallback
```

- Chuyển đổi sang nhị phân (Binary): Biến ảnh xám thành ảnh chỉ có hai màu đen và trắng tuyệt đối (0 và 255), giúp tách biệt hoàn toàn chữ số ra khỏi nền.

- Chuẩn hóa về "Chữ trắng - Nền đen":

- Dù bạn đưa vào ảnh **chữ đen trên giấy trắng** (ảnh chụp) hay **chữ trắng trên nền đen**, hàm này đều tự động phát hiện và đưa về định dạng **chữ trắng nền đen** (đúng chuẩn bộ dữ liệu MNIST).

- Cơ chế lọc nhiễu: Dựa vào nguyên lý "diện tích chữ số luôn nhỏ hơn diện tích nền" để chọn ra kết quả đúng nhất, loại bỏ trường hợp ảnh bị đảo ngược màu sai.

4. Hàm tìm vị trí chữ số/ shape

```python
def find_bounding_box(binary_image):
    """
    Tìm bounding box của vùng có pixel trắng (chữ số)

    Args:
        binary_image: Ảnh binary dạng numpy array

    Returns:
        Tuple (x_min, y_min, x_max, y_max)

    Raises:
        ValueError: Nếu không tìm thấy vùng chữ số
    """
    rows = np.any(binary_image, axis=1)
    cols = np.any(binary_image, axis=0)

    if not np.any(rows) or not np.any(cols):
        raise ValueError("Không tìm thấy vùng chữ số trong ảnh")

    y_min, y_max = np.where(rows)[0][[0, -1]]
    x_min, x_max = np.where(cols)[0][[0, -1]]

    return x_min, y_min, x_max, y_max
```

- Xác định vị trí: Tìm ra tọa độ 4 cạnh (trên, dưới, trái, phải) của hình chữ nhật **ôm sát nhất** lấy chữ số trong ảnh.

- Loại bỏ khoảng thừa: Cung cấp thông số để cắt bỏ toàn bộ phần nền đen vô nghĩa xung quanh, giúp AI chỉ tập trung phân tích hình dáng chữ số chứ không bị nhiễu bởi vị trí của nó (dù viết ở góc hay ở giữa).

5. Hàm thêm padding vào ảnh

```python
def add_padding_square(image):
    """
    Thêm padding để ảnh thành hình vuông

    Args:
        image: Ảnh dạng numpy array

    Returns:
        Ảnh vuông với padding
    """
    h, w = image.shape

    if h == w:
        return image

    size = max(h, w)
    padded = np.zeros((size, size), dtype=image.dtype)

    y_offset = (size - h) // 2
    x_offset = (size - w) // 2

    padded[y_offset:y_offset+h, x_offset:x_offset+w] = image

    return padded
```

- Biến ảnh thành hình vuông: Hàm này nhận diện cạnh dài nhất của ảnh gốc (chiều cao hoặc chiều rộng) và tạo ra một khung nền đen hình vuông có kích thước bằng cạnh đó.

- Giữ nguyên tỉ lệ (Chống méo): Thay vì kéo dãn ảnh (stretch) để làm vuông (khiến chữ số bị biến dạng, ví dụ số 1 bị mập ra), hàm này lấp đầy khoảng trống bằng màu đen (padding).

- Căn giữa: Nó đặt ảnh gốc vào **chính giữa** khung vuông mới.

6. Thay đổi kích thước ảnh

```python
def resize_image(image, new_size):
    """
    Resize ảnh bằng PIL

    Args:
        image: Ảnh dạng numpy array
        new_size: Kích thước mới (int)

    Returns:
        Ảnh đã resize (numpy array)
    """
    pil_image = Image.fromarray(image)
    resized = pil_image.resize((new_size, new_size), Image.LANCZOS)
    return np.array(resized)
```

- Thu phóng ảnh về kích thước cố định 28x28 pixel.

7. Thêm viền cho ảnh

```python
def add_border_padding(image, padding=4):
    """
    Thêm padding xung quanh ảnh

    Args:
        image: Ảnh dạng numpy array
        padding: Số pixel padding (mặc định 4)

    Returns:
        Ảnh với border padding
    """
    h, w = image.shape
    padded = np.zeros((h + 2*padding, w + 2*padding), dtype=image.dtype)
    padded[padding:padding+h, padding:padding+w] = image
    return padded
```

- Biến ảnh chữ nhật (sau khi crop) thành ảnh vuông bằng cách thêm nền đen vào hai bên (hoặc trên dưới).

8. Chuẩn hóa ảnh

```python
def normalize_image(image):
    """
    Normalize ảnh về khoảng [0.0, 1.0]

    Args:
        image: Ảnh dạng numpy array (uint8)

    Returns:
        Ảnh normalized (float32)
    """
    return image.astype(np.float32) / 255.0
```

- Chuẩn hóa giá trị pixel từ phạm vi số nguyên [0, 255] về phạm vi số thực [0.0, 1.0]
- Tăng tốc độ học (Hội tụ nhanh hơn): Các thuật toán tối ưu hóa (như Gradient Descent) hoạt động hiệu quả hơn nhiều với các con số nhỏ (quanh khoảng 0-1) so với các số lớn (0-255).

III. Xây dựng mô hình

```
model = models.Sequential(name='Combined_MNIST_Shapes_Optimized')

# Input layer
model.add(layers.Input(shape=input_shape))

# Block 1: Basic feature extraction (32 filters)
model.add(layers.Conv2D(32, kernel_size=(5, 5), activation='relu', padding='same', name='conv1'))
model.add(layers.BatchNormalization(name='bn1'))
model.add(layers.Conv2D(32, kernel_size=(3, 3), activation='relu', padding='same', name='conv2'))
model.add(layers.BatchNormalization(name='bn2'))
model.add(layers.MaxPooling2D(pool_size=(2, 2), name='pool1'))
model.add(layers.Dropout(0.25, name='dropout1'))

# Block 2: Intermediate features (64 filters)
model.add(layers.Conv2D(64, kernel_size=(3, 3), activation='relu', padding='same', name='conv3'))
model.add(layers.BatchNormalization(name='bn3'))
model.add(layers.Conv2D(64, kernel_size=(3, 3), activation='relu', padding='same', name='conv4'))
model.add(layers.BatchNormalization(name='bn4'))
model.add(layers.MaxPooling2D(pool_size=(2, 2), name='pool2'))
model.add(layers.Dropout(0.3, name='dropout2'))

# Block 3: Advanced features (128 filters)
model.add(layers.Conv2D(128, kernel_size=(3, 3), activation='relu', padding='same', name='conv5'))
model.add(layers.BatchNormalization(name='bn5'))
model.add(layers.Conv2D(128, kernel_size=(3, 3), activation='relu', padding='same', name='conv6'))
model.add(layers.BatchNormalization(name='bn6'))

# Global pooling for spatial invariance
model.add(layers.GlobalAveragePooling2D(name='global_pool'))

# Dense layers
model.add(layers.Dense(256, activation='relu', name='dense1'))
model.add(layers.BatchNormalization(name='bn7'))
model.add(layers.Dropout(0.4, name='dropout3'))

# Output layer
model.add(layers.Dense(num_classes, activation='softmax', name='output'))
```

1. Lớp Conv2D

- Chức năng: Trích xuất các đặc trưng hình ảnh (features) từ đơn giản đến phức tạp.

- Cơ chế hoạt động: Nó sử dụng các "cửa sổ trượt" (kernel/filter) quét qua toàn bộ bức ảnh.

- Các lớp đầu (như Conv2D(32, 5x5)): Tìm các nét cơ bản như đường thẳng, đường cong, góc cạnh.
- Các lớp sâu hơn (như Conv2D(128, 3x3)): Kết hợp các nét cơ bản để nhận ra các hình dạng phức tạp hơn (ví dụ: vòng tròn của số 8, nét gạch ngang của số 7).

2. Lớp BatchNormalization

- Vấn đề: Khi dữ liệu đi qua nhiều lớp, phân phối giá trị của nó bị thay đổi liên tục (gọi là *Internal Covariate Shift*), làm mạng khó học và dễ bị mất ổn định (gradient quá nhỏ hoặc quá lớn).

- Giải pháp: BN chuẩn hóa dữ liệu đầu ra của lớp trước đó về dạng phân phối chuẩn (mean ~ 0, variance ~ 1).

- Tác dụng thực tế:

- **Tăng tốc độ huấn luyện:** Giúp mạng hội tụ nhanh hơn gấp nhiều lần.
- **Cho phép Learning Rate lớn:** Bạn có thể train "bạo" hơn mà không sợ vỡ model.
- **Regularization nhẹ:** Giúp giảm thiểu Overfitting một chút.

3. Lớp MaxPooling2D

- Chức năng: Giảm kích thước không gian (chiều rộng, chiều cao) của ảnh nhưng giữ lại đặc trưng quan trọng nhất.

- Cơ chế: Với pool_size=(2, 2), nó chia ảnh thành các ô 2x2 pixel và chỉ giữ lại **giá trị lớn nhất** trong ô đó.

- **Giảm tải tính toán:** Biến ảnh 28x28 thành 14x14 giúp giảm 75% số lượng phép tính cho lớp sau.
- **Bất biến dịch chuyển (Translation Invariance):** Nếu bạn viết số 5 hơi lệch sang trái một chút, giá trị Max trong ô vẫn không đổi. Điều này giúp mô hình nhận diện tốt dù vật thể không nằm chính giữa.

4. Lớp Dropout

- Chức năng: Ngăn chặn hiện tượng Overfitting (Mô hình học thuộc lòng dữ liệu train nhưng kém trên dữ liệu thực tế).

- Cơ chế: Trong quá trình huấn luyện, Dropout sẽ **tắt ngẫu nhiên** một tỷ lệ nơ-ron (ví dụ 0.25 tức là 25%).

5. Dense (Fully Connected Layer)

- Vị trí: Thường nằm ở cuối cùng của mạng.

- Chức năng: Tổng hợp tất cả các đặc trưng mà các lớp Conv2D đã trích xuất để đưa ra quyết định cuối cùng.

- Cơ chế: Mỗi nơ-ron trong lớp Dense kết nối với **tất cả** nơ-ron của lớp trước đó. Nó học các mối quan hệ phi tuyến tính phức tạp.

- Lớp Dense(256): Học cách kết hợp các đặc trưng bậc cao.
- Lớp Dense(num_classes) (Output): Tính toán xác suất cho từng lớp (Ví dụ: 80% là số 5, 10% là số 3, ...).

6. Lớp GlobalAveragePooling2D

- Chức năng: Biến đổi khối dữ liệu 3 chiều (Feature Maps) thành vector 1 chiều bằng cách tính trung bình cộng các giá trị trong từng bản đồ.

- Cơ chế (Ví dụ minh họa):

- Giả sử lớp Conv cuối cùng trả về 128 bản đồ đặc trưng, mỗi bản đồ kích thước 7 * 7.

- **Input:** Tensor 7 * 7 *128).
- **Xử lý:** Với mỗi bản đồ 7x7 (49 số), cộng lại rồi chia trung bình → ra 1 con số.
- **Output:** Vector (128).

- Tác dụng

- **Chống Overfitting cực mạnh (Quan trọng nhất):**
    - Nó loại bỏ hàng triệu liên kết thừa so với cách dùng Flatten truyền thống.
    - Ít tham số hơn → Mô hình khó "học vẹt" hơn.
- **Tăng tính bất biến không gian (Translation Invariance):**

- Giúp mô hình nhận diện được vật thể (ví dụ: số 5) bất kể nó nằm ở góc trái, góc phải hay chính giữa. GAP chỉ quan tâm "có đặc điểm đó hay không", không quan tâm "nó nằm ở đâu".

- **Linh hoạt kích thước đầu vào:**
  - Cho phép mô hình nhận ảnh đầu vào có kích thước bất kỳ (không bắt buộc cố định 28x28), vì đầu ra của GAP luôn cố định theo số lượng filters (ví dụ: luôn là 128 số).

IV. Huấn luyện model

```
# Compile model
model.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)
```

1. Thuật toán tối ưu Adam

- Thuật toán dùng để cập nhật trọng số (weights) của mạng neural sau mỗi lần học.

- Cơ chế: Kết hợp giữa **Đà (Momentum)** (lao nhanh xuống dốc) và **Tự thích nghi (Adaptive)** (phanh lại khi gặp đường gập ghềnh).

- Thuật toán Adam tạo ra 2 biến momentum và RMS
- Thuật toán Adam không điều chỉnh trọng số của mô hình dựa trên giá trị gradient thô mà dựa vào momentum. Momentum là giá trị trung bình của gradient, giúp cho vượt quá được các cực tiểu địa phương của hàm loss.
- Biến RMS là căn bậc 2 của trung bình bình phương gradient. Thuật toán Adam sẽ chia learning rate cho giá trị này để tùy chỉnh learning rate cho từng trọng số, giúp quá trình học của mô hình ổn định và nhanh hơn. Khi mà gradient lớn thì learning rate sẽ giảm đi, và ngược lại.

- Nó là thuật toán **nhanh nhất và ổn định nhất** hiện nay cho hầu hết các bài toán.

2. Các hàm callbacks

```
callbacks = [
    EarlyStopping(
        monitor='val_accuracy',
        patience=15,
        restore_best_weights=True,
        verbose=1
    ),
    ModelCheckpoint(
        model_path,
        monitor='val_accuracy',
        save_best_only=True,
        verbose=1
    ),
    ReduceLROnPlateau(
        monitor='val_loss',
        factor=0.5,
        patience=7,
        min_lr=1e-7,
        verbose=1
    )
]
```

- EarlyStopping: Dừng việc huấn luyện lại ngay khi thấy mô hình không còn tiến bộ nữa để tránh lãng phí thời gian và chống học vẹt (Overfitting).

- ModelCheckpoint: Tự động lưu lại mô hình (file .h5 hoặc .keras).

- ReduceLROnPlateau: Giúp mô hình thoát khỏi điểm tắc nghẽn (local minima) hoặc hội tụ sâu hơn bằng cách giảm tốc độ học (Learning Rate).

3. Data Augmentation

Các tham số trong data augmentation:

- **rotation_range (Xoay):** Để nhận diện được vật thể khi nó bị **nghiêng**.
- **width_shift_range & height_shift_range (Dịch chuyển):** Để nhận diện được vật thể khi nó **không nằm chính giữa** ảnh.

- **zoom_range (Thu/phóng):** Để nhận diện được vật thể ở các **kích thước to/nhỏ** khác nhau (chụp xa/gần).
- **fill_mode (Lấp đầy):** Để **điền vào các khoảng trống** bị tạo ra khi xoay hoặc dịch chuyển ảnh.

- Mô hình nhận diện chữ số MNIST

Với mô hình nhận diện MNIST rotation_range = 30 độ để có thể nhận diện được chữ số 1 và 8 nằm ngang

```
]

# Data Augmentation
print("\n" + "="*70)
print("DATA AUGMENTATION CONFIGURATION")
print("="*70)
datagen = ImageDataGenerator(
    rotation_range=30,           # Xoay ±30 độ
    width_shift_range=0.1,       # Dịch ngang 10%
    height_shift_range=0.1,      # Dịch dọc 10%
    zoom_range=0.1,              # Zoom ±10%
    fill_mode='nearest'          # Điền pixel khi transform
)
```

- Mô hình nhận diện Shape

```
]

# Data Augmentation
print("\n" + "="*70)
print("DATA AUGMENTATION CONFIGURATION")
print("="*70)
datagen = ImageDataGenerator(
    rotation_range=10,           # Xoay ±10 độ
    width_shift_range=0.1,       # Dịch ngang 10%
    height_shift_range=0.1,      # Dịch dọc 10%
    zoom_range=0.1,              # Zoom ±10%
    shear_range=0.1,             # Shear transformation
)
```

V. Đánh giá mô hình

```python
def evaluate_model(model, test_data):
    """
    Đánh giá model trên test set

    Args:
        model: Trained keras model
        test_data: tuple (test_images, test_labels)

    Returns:
        ndarray: Predicted classes
    """
    test_images, test_labels = test_data

    print("\n" + "="*70)
    print("EVALUATING MODEL ON TEST SET")
    print("="*70)

    # Evaluate
    test_loss, test_acc = model.evaluate(test_images, test_labels, verbose=0)

    print(f"\nTest Accuracy: {test_acc*100:.2f}%")
    print(f"Test Loss: {test_loss:.4f}")

    # Predictions
    predictions = model.predict(test_images, verbose=0)
    predicted_classes = np.argmax(predictions, axis=1)

    # Classification report
    print("\n" + "="*70)
    print("CLASSIFICATION REPORT")
    print("="*70)

    class_names = [get_class_name(i) for i in range(18)]
    print(classification_report(test_labels,
          predicted_classes, target_names=class_names))

    # Per-category accuracy
    print("\n" + "="*70)
    print("PER-CATEGORY ACCURACY")
    print("="*70)

    for i in range(18):
        mask = test_labels == i
        if mask.sum() > 0:
            acc = (predicted_classes[mask] == test_labels[mask]).mean()
            class_name = get_class_name(i)
            print(
                f"Class {i:2d} ({class_name:15s}): {acc*100:.2f}% ({mask.sum()} images)")

    print("="*70)

    return predicted_classes
```

- Dùng hàm **classification_report** của **scikit learn** để tạo ra 1 báo cáo các metrics đánh giá mô hình:

- Precision: độ chuẩn xác (ví dụ trong tất cả các trường hợp mô hình dự đoán là hình tròn thì có bao nhiêu trường hợp dự đoán chính xác)

- Recall: độ phủ (trong tất cả trường hợp ảnh là hình tròn thì mô hình dự đoán đúng được bao nhiêu trong đó)

- F1-score: hệ số cân bằng giữa precision và recall

```
================================================================
CLASSIFICATION REPORT
================================================================
              precision    recall  f1-score   support

      circle       1.00      0.98      0.99       500
        kite       0.99      0.97      0.98       500
parallelogram      0.99      0.99      0.99       500
   rectangle       0.98      0.98      0.98       500
     rhombus       0.97      0.99      0.98       500
      square       0.97      0.98      0.98       500
   trapezoid       0.99      0.97      0.98       500
    triangle       0.95      0.96      0.96       500

    accuracy                           0.98      4000
   macro avg       0.98      0.98      0.98      4000
weighted avg       0.98      0.98      0.98      4000
```

VI. AIR WRITING

1. Xử lý ảnh

1.1. Chuyển đổi không gian màu

- Trong xử lý ảnh, thứ tự các kênh màu rất quan trọng. Thư viện PIL và chuẩn hiển thị thường dùng RGB, trong khi OpenCV thường dùng BGR

- bgr_to_rgb

```python
def bgr_to_rgb(image):
    """
    Chuyển đổi ảnh từ BGR sang RGB
    Args:
        image: Ảnh BGR (H, W, 3)
    Returns:
        Ảnh RGB (H, W, 3)
    """
    if len(image.shape) != 3 or image.shape[2] != 3:
        return image
    # BGR -> RGB: đảo ngược channel cuối cùng
    return image[:, :, ::-1].copy()
```

- Mục đích: Đổi chỗ 2 kênh màu xanh dương và đỏ

- Cơ chế: Sử dụng kỹ thuật slicing của Numpy trên trục cuối cùng (trục channel): image[:, :, ::-1]

+ Giả sử mảng ảnh 3D có kích thước (H, W, C), với C = 3

+ [::-1] trên trục C sẽ đảo ngược các thứ tự phần tử

+ (B, G, R) -> (R, G, B) và ngược lại

+ Hàm .copy() được sử dụng để đảm bảo kết quả là một mảng mới, độc lập với mảng gốc

- grayscale_to_bgr

```python
def grayscale_to_bgr(image):
    """
    Chuyển đổi ảnh grayscale sang BGR (3 channels)
    Args:
        image: Ảnh grayscale (H, W)
    Returns:
        Ảnh BGR (H, W, 3)
    """
    if len(image.shape) == 2:
        # Grayscale -> BGR: copy channel 3 lần
        return np.stack([image, image, image], axis=2)
    elif len(image.shape) == 3 and image.shape[2] == 1:
        # (H, W, 1) -> (H, W, 3)
        return np.repeat(image, 3, axis=2)
    return image.copy()
```

- Mục đích: Chuyển ảnh xám 1 kênh thành ảnh màu 3 kênh, nhưng các kênh đều giống nhau (ảnh vẫn hiển thị đen trắng), nếu muốn hiển thị ảnh grayscale trên giao diện hoặc sử dụng các hàm xử lý ảnh khác (draw_circle hoặc put_text) vốn được thiết kế để hoạt động dựa trên ảnh màu BGR

- Cơ chế:

+ Sử dụng np.stack([image, image, image], axis=2) để xếp chồng ảnh xám lên nhau 3 lần theo trục channel (trục 2)

+ Kết quả là 1 ảnh (H, W, 3) với B = G = R

1.2. Xử lý và biến đổi ảnh

- flip_horizontal

```python
def flip_horizontal(image):
    """
    Lật ảnh theo chiều ngang (trái -> phải)
    Args:
        image: Ảnh đầu vào
    Returns:
        Ảnh đã lật
    """
    return np.fliplr(image).copy()
```

- **Mục đích:** Lật ảnh theo chiều ngang (trục Y). Rất quan trọng trong các hệ thống xử lý ảnh camera thời gian thực để tạo ra ảnh phản chiếu như gương. Trong bài toán Air-Writing, camera thường là webcam quay mặt người dùng, ảnh thu được thường bị lật ngang (mirror) so với thực tế, khiến người dùng thấy hành động của mình tự nhiên hơn. Tuy nhiên để nhận diện chính xác hình vẽ (ví dụ, số '3' bị lật thành hình gần giống 'ε', cần lật lại hình ảnh để mô hình có thể nhận diện đúng

- **Cơ chế:** Sử dụng hàm np.fliplr(image) của Numpy, nó đơn giản đảo ngược thứ tự các cột(trục W) của ảnh

1.3. Các hàm vẽ

- Dùng để hiển thị thông tin như đường viết, điểm ngón tay và kết quả nhận dạng

- draw_circle

```python
def draw_circle(image, center, radius, color, thickness=-1):
    img = image.copy()
    x, y = center

    h, w = img.shape[:2]

    # Tạo mask cho hình tròn
    yy, xx = np.ogrid[:h, :w]
    dist_sq = (xx - x) ** 2 + (yy - y) ** 2

    if thickness == -1:
        # Fill circle
        mask = dist_sq <= radius ** 2
    else:
        # Circle outline
        outer_mask = dist_sq <= radius ** 2
        inner_mask = dist_sq <= (radius - thickness) ** 2
        mask = outer_mask & ~inner_mask

    # Vẽ màu
    if len(img.shape) == 2:
        # Grayscale
        if isinstance(color, (int, np.integer)):
            img[mask] = color
        else:
            # Nếu color là tuple, lấy giá trị đầu tiên
            img[mask] = color[0] if len(color) > 0 else 255
    else:
        # Color image
        if isinstance(color, (int, np.integer)):
            img[mask] = color
        else:
            # BGR format
            for c in range(min(len(color), img.shape[2])):
                img[:, :, c][mask] = color[c]

    return img
```

- Mục đích: Vẽ hình tròn hoặc hình tròn được tô đầy, sử dụng để đánh dấu ví trị tức thời của "cây bút" (thường là đầu ngón tay), khi phát triển vị trí tay/đầu bút, vẽ 1 chấm tròn (draw_circle) tại vị trí đó

- Cơ chế: đây là 1 ví dụ điển hình cho việc sử dụng vector hoá của Numpy để vẽ hình học:

+ Tạo lưới toạ độ (Y, X) bằng np.ogrid

+ Tính toán bình phương khoảng cách từ tâm hình tròn (x, y) đến mọi pixel (xx, yy) trên ảnh:

$$dist\_sq = (xx - x)^2 + (yy - y)^2$$

+ Tạo 1 Boolean mask (True/False) cho tất cả các pixel nằm trong bán kính r: mask = dist_sq <= radius ** 2

+ Nếu vẽ viền (thickness > 0), tạo thêm inner_mask và dùng phép toán AND (&) và NOT (~) để lấy ra các pixel nằm trong viền mask = outer_mask & ~inner_mask

+ Gán giá trị màu color cho các pixel mà mask là True

- draw_line

```python
def draw_line(image, pt1, pt2, color, thickness=1):
    img = image.copy()
    x1, y1 = pt1
    x2, y2 = pt2

    h, w = img.shape[:2]

    # Tính toán các điểm trên đường thẳng bằng Bresenham's line algorithm
    dx = abs(x2 - x1)
    dy = abs(y2 - y1)
    sx = 1 if x1 < x2 else -1
    sy = 1 if y1 < y2 else -1
    err = dx - dy

    x, y = x1, y1

    # Tập hợp tất cả các điểm trên đường thẳng
    points = []
    while True:
        points.append((x, y))
        if x == x2 and y == y2:
            break

        e2 = 2 * err
        if e2 > -dy:
            err -= dy
            x += sx
        if e2 < dx:
            err += dx
            y += sy

    # Vẽ các điểm với thickness
    if thickness == 1:
        # Vẽ từng điểm
        for px, py in points:
            if 0 <= px < w and 0 <= py < h:
                if len(img.shape) == 2:
                    # Grayscale
                    if isinstance(color, (int, np.integer)):
                        img[py, px] = color
                    else:
                        img[py, px] = color[0] if len(color) > 0 else 255
                else:
                    # Color
                    if isinstance(color, (int, np.integer)):
                        img[py, px] = color
                    else:
                        for c in range(min(len(color), img.shape[2])):
                            img[py, px, c] = color[c]
    else:
        # Vẽ với thickness > 1: vẽ nhiều điểm xung quanh mỗi điểm trên đường thẳng
        radius = thickness // 2
        for px, py in points:
            # Vẽ hình tròn nhỏ tại mỗi điểm
            yy, xx = np.ogrid[:h, :w]
            dist_sq = (xx - px) ** 2 + (yy - py) ** 2
            mask = dist_sq <= radius ** 2

            if len(img.shape) == 2:
                if isinstance(color, (int, np.integer)):
                    img[mask] = color
                else:
                    img[mask] = color[0] if len(color) > 0 else 255
            else:
                if isinstance(color, (int, np.integer)):
                    img[mask] = color
                else:
                    for c in range(min(len(color), img.shape[2])):
                        img[:, :, c][mask] = color[c]

    return img
```

- Mục đích: Vẽ đường thẳng, dùng để theo dõi quỹ đạo ngón tay. Khi tay di chuyển, cần vẽ 1 đường thẳng (draw_line) để nối các chấm, tạo ra hình vẽ mà người dùng đang thực hiện.

- Cơ chế: sử dụng Bresenham's Line Algorithm (Giải thuật Bresenham)

+ Đây là 1 thuật toán hiệu quả để xác định chính xác các toạ độ pixel phải được bật sáng để tạo thành 1 đường thẳng mượt mà giữa 2 điểm, chỉ sử dụng phép toán số học nguyên

+ Thuật toán này tính toán sự thay đổi tích luỹ (err) để quyết định bước đi tiếp theo là đi ngang, đi dọc hay đi chéo

+ Nếu thickness > 1, hàm sẽ vẽ nhiều hình tròn nhỏ (dùng bán kính radius = thickness // 2) tại mỗi điểm được tính bằng thuật toán Bresenham. Điều này tạo ra 1 đường thẳng có độ dày

- put_text

```python
def put_text(image, text, org, font_scale=1.0, color=(255, 255, 255), thickness=1, font=None):
    img = image.copy()
    x, y = org

    # Chuyển sang PIL để vẽ text dễ hơn
    if len(img.shape) == 2:
        # Grayscale -> RGB
        pil_image = Image.fromarray(img, mode='L').convert('RGB')
        is_grayscale = True
    else:
        # BGR -> RGB
        pil_image = Image.fromarray(bgr_to_rgb(img), mode='RGB')
        is_grayscale = False

    # Tạo ImageDraw
    draw = ImageDraw.Draw(pil_image)

    # Tính kích thước font
    try:
        # Thử dùng font hệ thống
        if font and os.path.exists(font):
            base_font = ImageFont.truetype(font, int(20 * font_scale))
        else:
            # Font mặc định
            try:
                # Thử dùng font mặc định của hệ thống
                if os.name == 'nt':  # Windows
                    base_font = ImageFont.truetype("arial.ttf", int(20 * font_scale))
                else:  # Linux/Mac
                    base_font = ImageFont.truetype("/usr/share/fonts/truetype/dejavu/DejaVuSans.ttf", int(20 * font_scale))
            except:
                # Fallback về font mặc định
                base_font = ImageFont.load_default()
    except:
        base_font = ImageFont.load_default()
```

```python
    # Chuyển màu BGR -> RGB
    if isinstance(color, (int, np.integer)):
        text_color = (color, color, color)
    else:
        if len(color) >= 3:
            # BGR -> RGB
            text_color = (color[2], color[1], color[0])
        else:
            text_color = (255, 255, 255)

    # Vẽ text
    # PIL dùng (x, y) là góc trên bên trái, cần điều chỉnh
    try:
        bbox = draw.textbbox((x, y), text, font=base_font)
        text_height = bbox[3] - bbox[1]
        # Điều chỉnh y để text bắt đầu từ org (góc dưới bên trái)
        text_y = max(0, y - text_height)  # Đảm bảo không âm
    except:
        # Fallback nếu không đo được
        text_y = max(0, y - 20)

    # Vẽ text với outline nếu thickness > 1
    if thickness > 1:
        # Vẽ outline
        for adj in range(-thickness, thickness + 1):
            for adj2 in range(-thickness, thickness + 1):
                if adj != 0 or adj2 != 0:
                    draw.text((x + adj, text_y + adj2), text, font=base_font, fill=(0, 0, 0))

    # Vẽ text chính
    draw.text((x, text_y), text, font=base_font, fill=text_color)

    # Chuyển về numpy array
    result_array = np.array(pil_image)

    # Chuyển về BGR nếu cần
    if len(img.shape) == 3:
        result_array = result_array[:, :, ::-1]  # RGB -> BGR

    # Chuyển về grayscale nếu ảnh gốc là grayscale
    if is_grayscale:
        result_array = rgb_to_grayscale(result_array)

    return result_array
```

- Mục đích: Vẽ chuỗi ký tự (text) lên ảnh, dùng để hiển thị kết quả nhận diện (ví dụ: “Đã nhận diện 5” hoặc “Hình học: Tam giác) trực tiếp lên khung hình video để người dùng theo dõi

- Cơ chế:

+ Chuyển sang PIL để tận dụng khả năng vẽ text chất lượng cao với TrueType Font (ImageFont.truetype)

+ Sử dụng ImageDraw để vẽ

+ Điều chỉnh toạ độ (offset) do PIL dùng góc trên bên trái của text làm gốc trong khi OpenCV dùng góc dưới bên phải

+ Hỗ trợ vẽ outline (viền) cho text bằng cách vẽ text chính nhiều lần, dịch chuyển 1 chút xung quanh vị trí gốc bằng màu đen hoặc màu nền, sau đó vẽ text chính bằng màu mong muốn

+ Chuyển ảnh kết quả của PIL từ RGB về lại BGR hoặc Grayscale Numpy array

1.4. Trộn ảnh

- add_weighted

```python
def add_weighted(src1, alpha, src2, beta, gamma=0.0):
    # Đảm bảo 2 ảnh cùng kích thước
    if src1.shape != src2.shape:
        # Resize src2 về kích thước src1
        h, w = src1.shape[:2]
        src2_resized = resize_image(src2, max(h, w))
        # Crop nếu cần
        if src2_resized.sha  (variable) src2_resized: NDArray[Any]
            src2_resized = src2_resized[:h, :w]
        src2 = src2_resized

    # Trộn ảnh
    result = alpha * src1.astype(np.float32) + beta * src2.astype(np.float32) + gamma

    # Clip về [0, 255] và chuyển về uint8
    result = np.clip(result, 0, 255).astype(np.uint8)

    return result
```

- Mục đích: Trộn 2 ảnh lại với nhau theo 1 tỷ lệ nhất định

+ Trộn hình ảnh khung video hiện tại với hình ảnh chỉ chứa đường vẽ (là kết quả của việc theo dõi chuyển động ngón tay), làm cho đường vẽ hiển thị nổi bật trên khung hình gốc

+ Làm mờ/ổn định: trộn nhiều khung hình lại với nhau để tạo hiệu ứng tích luỹ hoặc làm mượt

- Cơ chế: Thực hiện phép toán tuyến tính theo công thức

+ src1, src2 là 2 ảnh đầu vào

+ α và β là trọng số (thường là + = 1 để giữ độ sáng ổn định)

+ γ là 1 giá trị cộng thêm (bi-as) để tăng/giảm độ sáng tổng thể

+ Ví dụ ứng dụng: trộn khung hình camera (src1) với đường viết đã vẽ (src2) để thấy cả 2

+ Giới hạn (clipping): kết quả được giới hạn trong khoảng [0, 255] bằng np.clip và chuyển về kiểu np.uint8 để đảm bảo là 1 ảnh hợp lệ

2. Tiền xử lý

- Áp dụng phần tiền xử lý tương tự như của nhận diện chữ số MNIST và nhận diện các hình học cơ bản. Bao gồm

+ Chuyển ảnh từ rgb sang grayscale

+ Phân ngưỡng và đảo ngược nếu cần

+ Tìm bounding box

+ Crop ảnh

+ Padding ảnh thành ảnh vuông

+ Resize ảnh

+ Thêm border để chuyển ảnh về 28x28 pixel

+ Normalize về [0, 1]

3. Luồng xử lý chính

3.1. Lớp SmoothingFilter

```
class SmoothingFilter:
    """Lớp lọc smoothing cho tọa độ ngón tay với adaptive smoothing"""
    def __init__(self, alpha=0.75):
        """
        alpha: hệ số smoothing cơ bản (0-1)
        - 0: không smoothing (phản hồi tức thì)
        - 1: smoothing tối đa (rất mượt nhưng chậm)
        """
        self.base_alpha = alpha
        self.x = None
        self.y = None
        self.prev_x = None
        self.prev_y = None
        self.velocity = 0.0  # Tốc độ di chuyển
```

- Chịu trách nhiệm duy nhất là làm mượt toạ độ của ngón tay trỏ, loại bỏ hiện tượng rung và nhiễu (jitter) thường thấy khi nhận diện chuyển động thực

- Bộ lọc được khởi tạo với hệ số làm mượt alpha

- Update

```python
def update(self, x, y):
    """Cập nhật tọa độ với adaptive smoothing dựa trên tốc độ"""
    if self.x is None or self.y is None:
        # Lần đầu tiên, lấy giá trị trực tiếp
        self.x = float(x)
        self.y = float(y)
        self.prev_x = float(x)
        self.prev_y = float(y)
        self.velocity = 0.0
    else:
        # Tính tốc độ di chuyển
        dx = float(x) - self.prev_x
        dy = float(y) - self.prev_y
        current_velocity = np.sqrt(dx**2 + dy**2)

        # Smooth velocity để tránh thay đổi đột ngột
        self.velocity = 0.7 * self.velocity + 0.3 * current_velocity

        # Adaptive smoothing: di chuyển nhanh thì smoothing ít hơn (phản hồi nhanh)
        # di chuyển chậm thì smoothing nhiều hơn (mượt mà)
        if self.velocity > FAST_MOVEMENT_THRESHOLD:
            # Di chuyển nhanh: giảm smoothing để phản hồi nhanh
            alpha = self.base_alpha * 0.6  # Smoothing ít hơn
        elif self.velocity > FAST_MOVEMENT_THRESHOLD * 0.5:
            # Di chuyển trung bình: smoothing vừa phải
            alpha = self.base_alpha * 0.8
        else:
            # Di chuyển chậm: smoothing nhiều để mượt
            alpha = self.base_alpha

        # Exponential smoothing với alpha động
        self.x = alpha * self.x + (1 - alpha) * float(x)
        self.y = alpha * self.y + (1 - alpha) * float(y)

        # Lưu tọa độ gốc để tính velocity cho lần sau
        self.prev_x = float(x)
        self.prev_y = float(y)

    return int(self.x), int(self.y)
```

- Mục đích: Tính toán toạ độ làm mượt mới dựa trên toạ độ thô (x, y) hiện tại, đây là cơ chế chính để chống rung (anti-jitter), nó giúp bù đắp cho các dao động nhỏ tay rung hoặc sai số nhận diện tạm thời của MediaPipe, làm cho đường vẽ cuối cùng trở nên liên tục và dễ nhận dạng hơn, giúp đường vẽ **mượt mà** (giảm rung) nhưng vẫn đảm bảo **phản hồi nhanh** (giảm độ trễ) khi người dùng di chuyển tay nhanh, tối ưu hóa trải nghiệm viết.

- Cơ chế:

+ **Tính vận tốc:** Tính khoảng cách di chuyển giữa tọa độ gốc hiện tại (x, y) và tọa độ gốc của frame trước (self.prev_x, self.prev_y) để có current_velocity. Sau đó, smooth vận tốc này.

$$\text{current\_velocity} = \sqrt{(x - \text{prev\_x})^2 + (y - \text{prev\_y})^2}$$

+ **Adaptive Logic:** Dựa trên self.velocity so với FAST_MOVEMENT_THRESHOLD, điều chỉnh hệ số alpha (VD: nếu tốc độ nhanh, giảm alpha xuống 60% base_alpha).

+ **Exponential Smoothing:** Áp dụng công thức smoothing với alpha động để tính toán tọa độ mới:

$$\text{x\_smooth} = \alpha \times \text{x\_smooth\_cũ} + (1 - \alpha) \times \text{x\_raw}$$

+ Lưu tọa độ gốc hiện tại vào self.prev_x, self.prev_y.

- reset(self)

```
def reset(self):
    """Reset filter"""
    self.x = None
    self.y = None
    self.prev_x = None
    self.prev_y = None
    self.velocity = 0.0
```

- Mục đích: Đặt lại trạng thái của bộ lọc. Được gọi khi bắt đầu hoặc xóa canvas để đảm bảo quá trình smoothing bắt đầu lại từ đầu, không bị ảnh hưởng bởi tọa độ cuối cùng của lần vẽ trước.

- Cơ chế: Đặt lại tất cả các biến tọa độ (self.x, self.y, self.prev_x, self.prev_y) về None và tốc độ về 0.0.

3.2. Lớp AirWritingCore

```python
class AirWritingCore:
    """Lớp xử lý core logic cho Air Writing"""
    def __init__(self):
        self.cap = None
        self.active = False
        self.canvas = None
        self.drawing = False
        self.prev_point = None

        # MediaPipe
        self.mp_hands = mp.solutions.hands
        self.hands = None

        # Smoothing filter
        self.smoother = SmoothingFilter(alpha=SMOOTHING_FACTOR)
```

- Quản lý toàn bộ vòng lặp xử lý video, nhận diện và vẽ

- start

```python
def start(self, camera_index=0):
    """Bắt đầu Air Writing"""
    self.cap = cv2.VideoCapture(camera_index)
    if not self.cap.isOpened():
        return False

    frame_width = int(self.cap.get(cv2.CAP_PROP_FRAME_WIDTH) or 640)
    frame_height = int(self.cap.get(cv2.CAP_PROP_FRAME_HEIGHT) or 480)
    self.canvas = np.zeros((frame_height, frame_width), dtype=np.uint8)

    self.active = True
    self.drawing = False
    self.prev_point = None
    self.smoother.reset()

    self.hands = self.mp_hands.Hands(
        max_num_hands=1,
        min_detection_confidence=0.5,  # Giảm để tránh mất detection
        min_tracking_confidence=0.5,    # Giảm để tracking ổn định hơn
        static_image_mode=False,
        model_complexity=0  # Giảm complexity để tăng tốc độ xử lý
    )

    return True
```

- Mục đích: Khởi động camera, canvas và mô hình MediaPipe Hands, thu thập nguồn dữ liệu (camera) và bộ công cụ nhận diện (MediaPipe) trước khi bắt đầu vòng lặp xử lý, đảm bảo ứng dụng sẵn sàng hoạt động

- Cơ chế: Mở cv2.VideoCapture, tạo self.canvas (1 ma trận Numpy màu đen, np.zeros) cùng kích thước với mô hình camera, sau đó khởi tạo đối tượng self.hands từ MediaPipe

- stop

```
def stop(self):
    """Dừng Air Writing"""
    self.active = False
    if self.cap:
        self.cap.release()
        self.cap = None
    if self.hands:
        self.hands.close()
        self.hands = None
    self.smoother.reset()
```

- Mục đích: Dừng và giải phóng tài nguyên hệ thống, đặc biệt là camera, để tránh xung đột với các ứng dụng khác và ngăn rò rỉ bộ nhớ

- Cơ chế: Đặt trạng thái active = False, gọi self.cap.release() để tắt camera, và self.hands.close() để giải phóng mô hình MediaPipe

- process_frame

```python
def process_frame(self):
    """Xử lý một frame và trả về frame đã xử lý"""
    if not self.active or not self.cap:
        return None

    ret, frame = self.cap.read()
    if not ret:
        return None

    frame = flip_horizontal(frame)
    rgb = bgr_to_rgb(frame)
    results = self.hands.process(rgb)

    index_point = None

    if results.multi_hand_landmarks:
        for hand_landmarks in results.multi_hand_landmarks:
            h, w, _ = frame.shape
            x_raw = int(hand_landmarks.landmark[self.mp_hands.HandLandmark.INDEX_FINGER_TIP].x * w)
            y_raw = int(hand_landmarks.landmark[self.mp_hands.HandLandmark.INDEX_FINGER_TIP].y * h)

            # Áp dụng smoothing
            x_smooth, y_smooth = self.smoother.update(x_raw, y_raw)
            index_point = (x_smooth, y_smooth)

            # Vẽ chấm đỏ tại ngón tay trỏ (dùng tọa độ đã smooth)
            frame = draw_circle(frame, index_point, 6, (0, 0, 255), -1)

    # Drawing Logic với kiểm tra khoảng cách và interpolation
    if self.drawing and index_point is not None:
        if self.prev_point is not None:
            # Tính khoảng cách giữa 2 điểm
            dx = index_point[0] - self.prev_point[0]
            dy = index_point[1] - self.prev_point[1]
            distance = np.sqrt(dx**2 + dy**2)

            # Chỉ vẽ nếu khoảng cách đủ lớn (giảm rung)
            if distance >= MIN_DISTANCE:
                # Nếu khoảng cách quá lớn, thêm interpolation để vẽ mượt
                if distance > MAX_DISTANCE_FOR_INTERPOLATION:
                    # Tính số điểm cần interpolate - tối ưu dựa trên khoảng cách
                    # Di chuyển nhanh: ít điểm hơn để vẽ nhanh, di chuyển chậm: nhiều điểm hơn để mượt
                    segment_size = max(6, min(10, int(distance / 4)))  # Tối ưu segment size
                    num_segments = max(2, int(distance / segment_size))
                    prev_interp_point = self.prev_point

                    for i in range(1, num_segments + 1):
                        t = i / num_segments
                        interp_x = int(self.prev_point[0] + dx * t)
                        interp_y = int(self.prev_point[1] + dy * t)
                        interp_point = (interp_x, interp_y)

                        # Vẽ từng đoạn nhỏ liên tiếp
                        self.canvas = draw_line(self.canvas, prev_interp_point,
                                                interp_point, 255, thickness=8)
                        prev_interp_point = interp_point
                else:
                    # Khoảng cách bình thường, vẽ trực tiếp
                    self.canvas = draw_line(self.canvas, self.prev_point,
                                            index_point, 255, thickness=8)
                self.prev_point = index_point
        else:
            # Lần đầu tiên, lưu điểm
            self.prev_point = index_point
    elif not self.drawing:
        # Khi không vẽ, reset prev_point
        self.prev_point = None
    # Nếu drawing=True nhưng không detect được tay, giữ nguyên prev_point để tiếp tục khi detect lại

    # Overlay canvas
    color_canvas = grayscale_to_bgr(self.canvas)
    display = add_weighted(frame, 0.7, color_canvas, 0.3, 0)

    # Hiển thị trạng thái - font lớn hơn để dễ nhìn
    status_color = (0, 255, 0) if self.drawing else (0, 0, 255)
    display = put_text(display, f"Drawing: {self.drawing}", (10, 50),
                       font_scale=1.5, color=status_color, thickness=3)

    return display
```

- Là hàm được gọi lặp đi lặp lại để xử lý luồng video

- Đọc và lật frame: Đọc frame từ camera, sau đó flip_horizontal(frame) để lật ngang, tạo cảm giác phản chiếu gương, giúp người dùng dễ dàng điều khiển ngón tay theo hình ảnh trên màn hình

- Xử lý MediaPipe: Chuyển đổi frame sang RGB và gọi self.hands.process(rgb) để nhận diện tay, MediaPipe trả về một landmarks (toạ độ các khớp tay) đã được chuẩn hoá (từ 0 đến 1)

- Trích xuất và làm mượt toạ độ: Lấy toạ độ của đầu ngón trỏ (INDEX_FINGER_TIP), chuyển thành toạ độ pixel thô, và áp dụng self.smoother.update() để có được index_point đã làm mượt, đảm bảo điểm dùng để vẽ là điểm đã được khử nhiễu, đồng thời vẽ 1 chấm đỏ nhỏ (draw_circle) tại vị trí này để người dùng có phản hồi trực quan

- Logic vẽ:

+ Nếu self.drawing là True và có điểm ngón tay mới

1. Tính toán khoảng cách Euclidean giữa điểm hiển tại và slef.prev_point
2. Chỉ gọi draw_line() lên self.canvas nếu khoảng cách này lớn hơn MIN_DISTANCE
3. Nếu distance > MAX_DISTANCE_FOR_INTERPOLATION, tính toán số lượng phân đoạn và vẽ nhiều đường nhỏ (linear interpolation) để lấp đầy khoảng trống (gap-filling)

+ MIN_DISTANCE hoạt động như một ngưỡng chống rung thứ 2, nếu hai điểm quá gần nhau (do rung nhẹ), hệ thống sẽ không vẽ, ngăn tạo ra các đường nét quá ngắn, hoặc các chấm chấm không mong muốn. Điều này giúp đường nét mượt và sắc nét hơn

- Overlay Canvas: Chuyển canvas (grayscale) thành màu, sau đó dùng add_weighted() để pha trộn frame camera và canvas màu, pha trộn với trọng số để nét vẽ hiện lên rõ ràng nhưng vẫn giữ được nền video trong suốt, tạo hiệu ứng "vẽ trên không khí"

- clear_canvas: Đặt canvas về 0 (màu đen) và reset bộ lọc, cần thiết để xoá bảng vẽ và bắt đầu nét vẽ mới một cách sạch sẽ

```python
def clear_canvas(self):
    """Xóa canvas"""
    if self.canvas is not None:
        self.canvas[:] = 0
    self.prev_point = None
    self.smoother.reset()
```

- get_canvas: Trả về bản sao của canvas, cần thiết để module nhận dạng có thể lấy ảnh đã vẽ mà không làm hỏng dữ liệu gốc đang được hiển thị

```python
def get_canvas(self):
    """Lấy canvas hiện tại"""
    return self.canvas.copy() if self.canvas is not None else None
```