Wrocław University of Science and Technology

ARTIFICIAL INTELLIGENCE AND COMPUTER VISION

Faculty of Electronics, Photonics and Microsystems
AIR DRAWING CALCULATOR (AI AND CV)

Theme of class: Final Report

Students:
1. Stanislav Kustov 275512

Date of class: 2025-01-30

Submission Date: 2025-01-30

Lab assistant: Mateusz Cholewiski

Contents

1 Introduction

The **Air Drawing Calculator** is an interactive application that combines **Computer Vision (CV)** and **Artificial Intelligence (AI)** to recognize and process handwritten mathematical expressions in real-time. Using **hand tracking, gesture recognition, and deep learning**, this system allows users to draw numbers and mathematical operators in the air and instantly compute results.

1.1 Purpose and Applications

The purpose of this project is to develop an interactive application that combines hand tracking, gesture recognition, and optical character recognition (OCR) to create a virtual environment for drawing and recognizing handwritten symbols and equations.

1.2 How It Works

The system consists of two key components:

- **Hand Tracking & Gesture Recognition:**
  - Uses **Mediapipe** to track the user's hand.
  - The fingertip acts as a digital pen for writing numbers and operators.
  - A palm gesture allows erasing content effortlessly.
- **AI-Based Symbol Recognition:**
  - A **CNN model** classifies handwritten digits (0-9).
  - A separate model detects operators (+, -, *, /).
  - Small unintended marks are ignored, while deliberate dots are recognized as decimal points.

1.3 Technologies Used

This project is built using:

- **OpenCV** For image processing and video capture.
- **Mediapipe** For hand tracking and gesture detection.
- **TensorFlow/Keras** For deep learning-based handwriting recognition.
- **NumPy** For numerical operations and data processing.

1.4 Overview

This report covers:

- Implementation of hand tracking and gesture recognition.
- AI models for handwritten digit and operator recognition.
- Testing, accuracy improvements, and future enhancements.

By combining AI with gesture-based input, the Air Drawing Calculator offers a new, interactive way to solve mathematical problems without the need for a physical interface.

2 Assumptions

2.1 Functional Assumptions

- The user will write mathematical expressions clearly and legibly.
- The system assumes that only one equation or a set of numbers is written at a time.
- Small accidental marks will be ignored unless they resemble meaningful digits or operators.
- Hand gestures will be used for writing, erasing, and confirming calculations.
- The system will automatically detect and process digits, operators, and decimal points.

2.2 Design Assumptions

- The application is designed for real-time use with minimal processing delay.
- Users will interact with the system via hand tracking and gesture recognition.
- The drawing area is a virtual canvas displayed over the camera feed.
- The system prioritizes single-line input but supports multi-line numbers.
- Operators and digits are written separately to ensure accurate recognition.

2.3 Hardware Components

- A camera with a reasonable resolution (e.g., 720p or higher) is required for accurate hand tracking.
- A computer with a CPU or GPU capable of running deep learning models in real-time.
- Adequate lighting conditions are assumed to improve gesture recognition accuracy.
- Users should maintain a steady hand while writing to enhance recognition reliability.

2.4 Software Assumptions

- The system is built using Python and requires OpenCV, Mediapipe, and TensorFlow/Keras.
- The machine learning models are pre-trained and loaded during execution.
- The software runs on a local machine and does not require an internet connection.
- The program assumes that all libraries and dependencies are installed and configured properly.
- The application will run smoothly on modern operating systems (Windows, Linux, macOS).

3 Description of Software

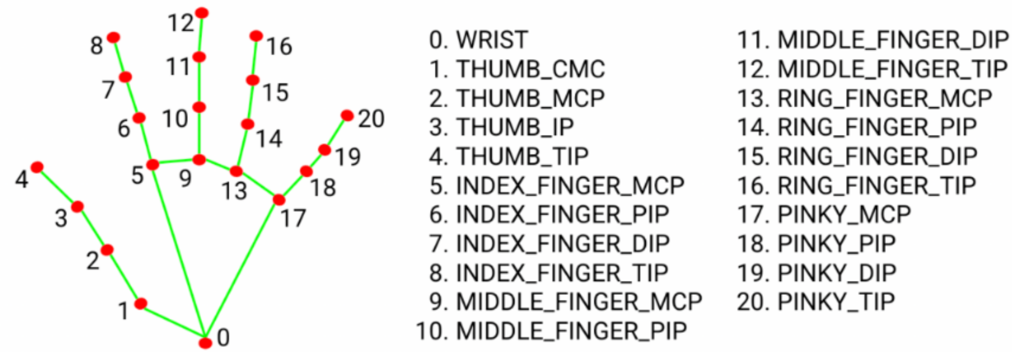
3.1 Hand Tracking

The development of the Air Calculator began with the implementation of real-time hand tracking and gesture recognition using **Mediapipe**, an advanced framework for detecting and analyzing hand landmarks.

The **HandLandmarker** module in Mediapipe accurately identifies the position of the user's hands and fingertips, establishing the foundation for gesture-based interactions such as drawing and erasing.

The implementation was based on the official Mediapipe documentation and GitHub repository. Below is a simplified example demonstrating how Mediapipe detects hand landmarks in an image:

The hand landmark model bundle detects the keypoint localization of 21 hand-knuckle coordinates within the detected hand regions. The model was trained on approximately 30K real-world images, as well as several rendered synthetic hand models imposed over various backgrounds.



The hand landmarker model bundle contains a palm detection model and a hand landmarks detection model. The Palm detection model locates hands within the input image, and the hand landmarks detection model identifies specific hand landmarks on the cropped hand image defined by the palm detection model.

```python
# Step 1: Import necessary modules
import mediapipe as mp
from mediapipe.tasks import python
from mediapipe.tasks.python import vision

# Step 2: Create a HandLandmarker object
base_options = python.BaseOptions(model_asset_path='hand_landmarker.task')
options = vision.HandLandmarkerOptions(base_options=base_options, num_hands=2)
detector = vision.HandLandmarker.create_from_options(options)

# Step 3: Load an input image
image = mp.Image.create_from_file("image.jpg")

# Step 4: Detect hand landmarks
detection_result = detector.detect(image)

# Step 5: Visualize the detected hand landmarks
annotated_image = draw_landmarks_on_image(image.numpy_view(), detection_result)
cv2_imshow(cv2.cvtColor(annotated_image, cv2.COLOR_RGB2BGR))
```

This code initializes the Mediapipe hand tracking system, loads an image, and detects hand landmarks, which can then be used for further processing.

3.2 Fingertip Tracking

With hand tracking established, the next phase involved real-time tracking of the **fingertip position**, which is essential for the air drawing functionality. By focusing on the index fingertip (landmark 8), the system can precisely capture the users drawing movements. Additionally, a **thumb gesture** serves as a control mechanism, allowing users to switch between drawing and idle modes effortlessly.

```
fingertip_x = hand_landmarks.landmark[8].x * frame.shape[1]
fingertip_y = hand_landmarks.landmark[8].y * frame.shape[0]
fingertip_position = (int(fingertip_x), int(fingertip_y))

if is_thumb_extended(hand_landmarks):
    drawing_mode = True
else:
    drawing_mode = False

if drawing_mode:
    if previous_position is not None:
        cv2.line(canvas, previous_position, fingertip_position, (255, 255, 255), 5)
    previous_position = fingertip_position
else:
    previous_position = None
```

Step-by-Step Explanation:

- **Fingertip Position Tracking:** The index fingertip (landmark 8) is detected and converted into pixel coordinates:

```
fingertip_x = hand_landmarks.landmark[8].x * frame.shape[1]
fingertip_y = hand_landmarks.landmark[8].y * frame.shape[0]
fingertip_position = (int(fingertip_x), int(fingertip_y))
```

- **Thumb Gesture Detection:** The system determines whether the thumb is extended to activate drawing mode:

```
if is_thumb_extended(hand_landmarks):
    drawing_mode = True
else:
    drawing_mode = False
```

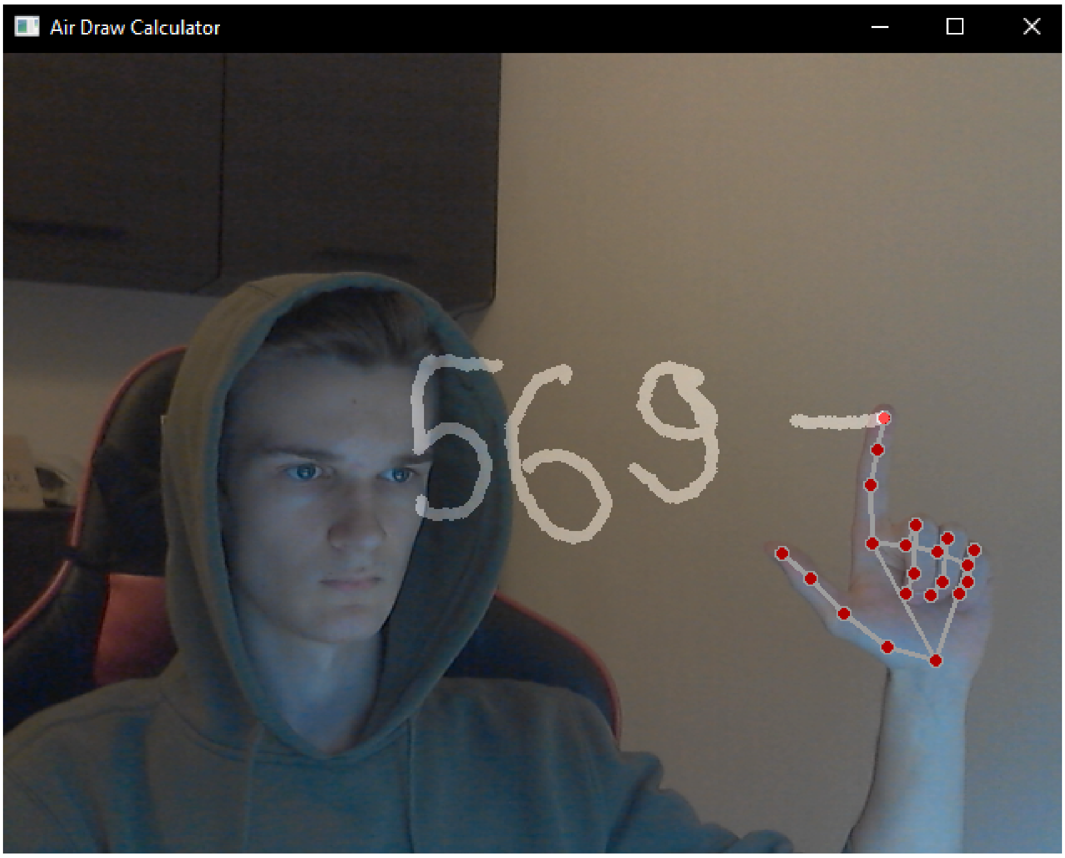
- **Drawing Mode:** When drawing is enabled, lines are drawn between consecutive fingertip positions:

```
if drawing_mode:
    if previous_position is not None:
        cv2.line(canvas, previous_position, fingertip_position, (255, 255, 255), 5)
    previous_position = fingertip_position
else:
    previous_position = None
```

- **Canvas Initialization:** A black canvas matching the frame dimensions is used as the drawing surface:

```
if canvas is None:
    canvas = np.zeros_like(frame)
```

This implementation allows users to **draw lines in real-time** by moving their fingertip while extending the thumb. When the thumb is not extended, drawing is disabled.



3.3 Air Drawing and Erasing

Once the fingertip position is tracked, the system allows users to draw in the air. The path of the fingertip is visualized as white strokes on a black canvas.

Drawing Mode:

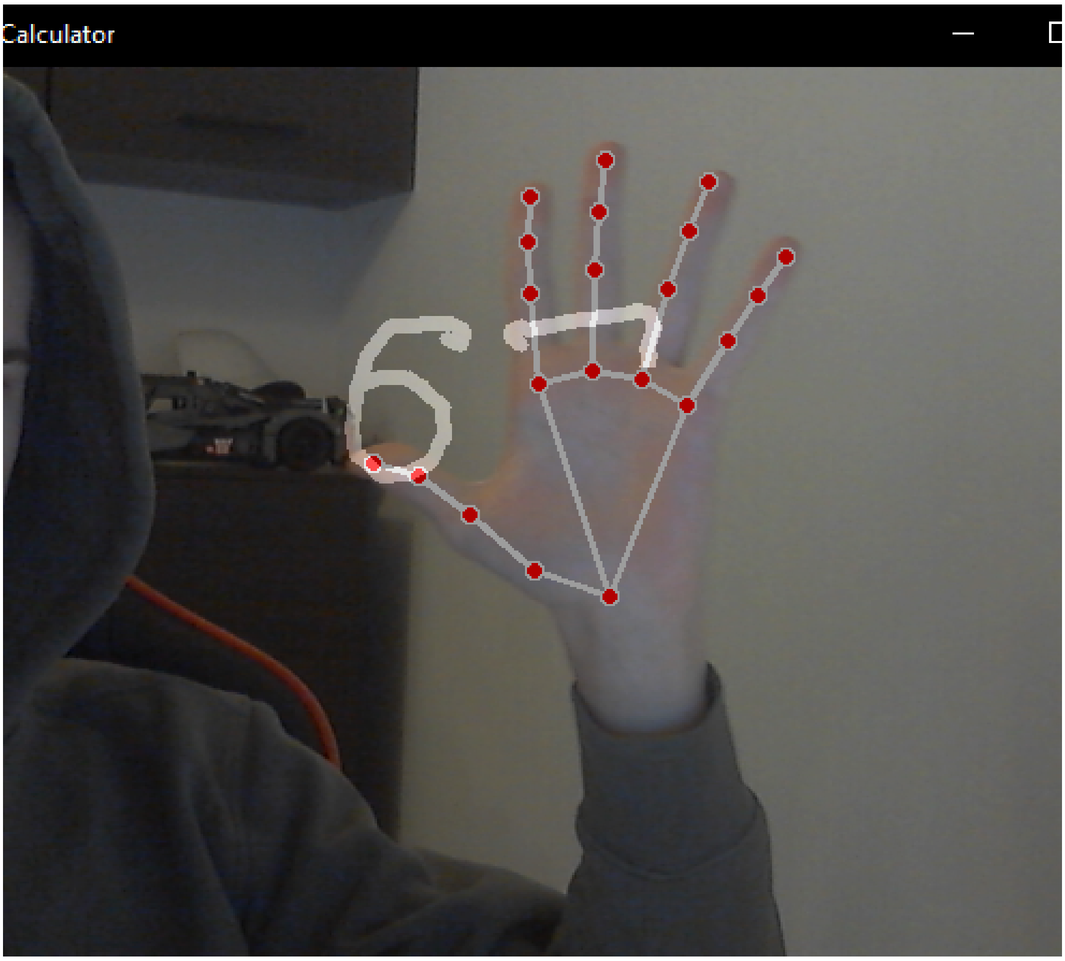
- The user extends their thumb to activate drawing mode.
- Movement of the index fingertip is recorded and displayed on the screen.

Erasing Mode:

- If the user's palm is detected, an erasing operation is triggered.
- A black rectangle removes drawn elements.
- If the palm detection fails, the system erases based on finger position.

Erasing Implementation:

```
if is_hand_open(hand_landmarks):
    erase_area(hand_landmarks, canvas, frame.shape)
```



3.4 Digit and Operator Recognition

When the user finishes writing an expression, the system segments the written input into separate components:

- **Digits (0-9)**: Recognized using a CNN trained on handwritten digits.
- **Operators** (+, -, *, /): Recognized using a separate CNN for mathematical symbols.

Digit Recognition:

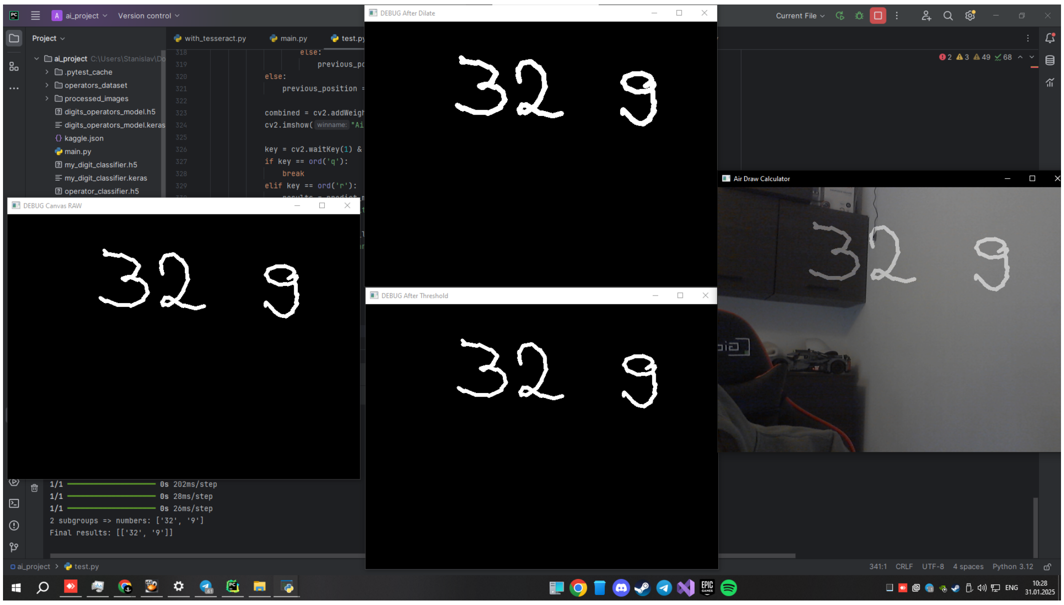
- The digit is extracted and resized to **28x28** pixels.
- The image is normalized and passed into the trained CNN model.
- The output is a classification result in the range **0-9**.

Operator Recognition:

- The extracted operator is inverted (black to white).
- It is resized and normalized before being classified by the CNN model.
- The output is mapped to one of the four basic operators.

Digit Prediction Code:

```
final_28 = cropped.astype('float32') / 255.0
input_data = final_28.reshape(1, 28, 28, 1)
preds = digits_model.predict(input_data, verbose=1)
digit = np.argmax(preds)
```



3.5 Equation Processing and Solving

When the user presses the **R** key, the program runs a function called `predict_all_in_one_line(canvas)` that does the following:

1. **Preprocessing the Canvas:**
   - Converts the black-and-white "drawing" (the canvas) to grayscale.
   - Applies a binary threshold (e.g., value of 127) so that only solid white contours remain.
   - Performs a small dilation to ensure the contours are connected clearly.
   - Uses `cv2.findContours` to detect all white patches (each patch is a contour).
2. **Extracting Bounding Boxes:**
   - For each contour, the program computes a bounding box (x, y, w, h).
   - These boxes are stored in a list.
3. **Grouping Bounding Boxes (Sublines):**
   - The list of boxes is sorted from left to right based on the `x`-coordinate.
   - The function `split_line_into_groups` checks the *horizontal gap* between consecutive boxes. For two boxes with coordinates (x, y, w, h) and (px, py, pw, ph), the gap is
     $$\text{gap} = x - (px + pw).$$
   - If gap $\leq$ `space_x_thresh` (default = 15 pixels), the boxes are merged into the same group, which usually indicates consecutive parts of a single number (like "1" next to "5" forming "15").
   - If gap > 15, the program starts a new group, indicating a larger separation (for example, a gap between a digit and an operator).
4. **Determining if it is an Equation:**
   - The program then counts how many **groups** were formed.
   - If there are exactly 3 groups, the code assumes the user wrote:
     $$\text{(Group 1: Number)} \quad \text{(Group 2: Operator)} \quad \text{(Group 3: Number)}.$$
   - If there are more or fewer than 3 groups, the program simply treats them as a *list of recognized items* (most likely just digits or decimal points) and prints them separated by semicolons without trying to do any computation.

Why three groups means "Equation": By design, a basic arithmetic expression like `"3 + 5"` or `"15 * 24"` naturally splits into **three** groups:

- *Left operand* (Group 1) – e.g., `"3"` or `"15"`.
- *Operator* (Group 2) – e.g., `"+"` or `"*"`.
- *Right operand* (Group 3) – e.g., `"5"` or `"24"`.

If the user drew more symbols (e.g., `"3"`, `"2"`, `"15"`, `"46"`) or fewer symbols (e.g., `"15"`, `"24"`), the system does not recognize a standard "number – operator – number" pattern.

3.5.1 Performing the Calculation

Once the program determines that there are exactly three groups, it takes the following steps to compute the result. Below is the relevant portion of the code:

```python
if len(sublines) == 3:
    # Convert recognized strings to float
    left_val = float(left_str)
    right_val = float(right_str)

    # Operator check
    if op_str == '+':
        res = left_val + right_val
    elif op_str == '-':
        res = left_val - right_val
    elif op_str == '*':
        res = left_val * right_val
    elif op_str == '/':
        res = left_val / right_val if right_val != 0 else "ERR(div0)"
```

- **String to Number Conversion:** The left and right groups (e.g., `"3"` or `"15.2"`) are transformed into floating-point values.
- **Recognizing the Operator:** The middle group is passed to the operator model, which returns `'+'`, `'-'`, `'*'`, or `'/'`.
- **Applying Arithmetic:** Depending on the operator, the program uses a basic `if-elif` block to do addition, subtraction, multiplication, or division.
- **Division by Zero:** If `right_val` is zero and the operator is `'/'`, the program returns `"ERR(div0)"`.
- **Rounding to an Integer (if needed):** After the calculation, if the result is very close to an integer (for example, 8.000000001), the code converts it to an `int` to avoid printing `8.0`.

3.5.2 Examples of Different Group Counts

Example 1: Three Groups, Equation ("3 + 5")

- First group is recognized as `"3"` → converts to `3.0`.
- Second group is recognized as `"+"`.
- Third group is recognized as `"5"` → converts to `5.0`.
- The system calculates `3.0 + 5.0 = 8.0` and then rounds it to `8`.
- It prints: `"3 + 5 = 8"`.

Example 2: Four Groups ("3", "2", "15", "46")

- The user draws four distinct symbols far enough apart that each becomes its own group.
- There is no `operator` recognized in a separate middle group.
- The system outputs them as: `"3; 2; 15; 46"` with no attempt at arithmetic, because `len(sublines)` = 4.

Example 3: Two Groups ("15", "24")

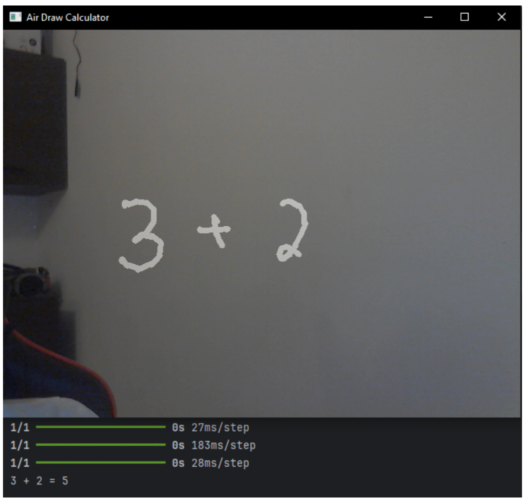
- Again, no operator group exists.
- The program will simply output: `"15; 24"`.

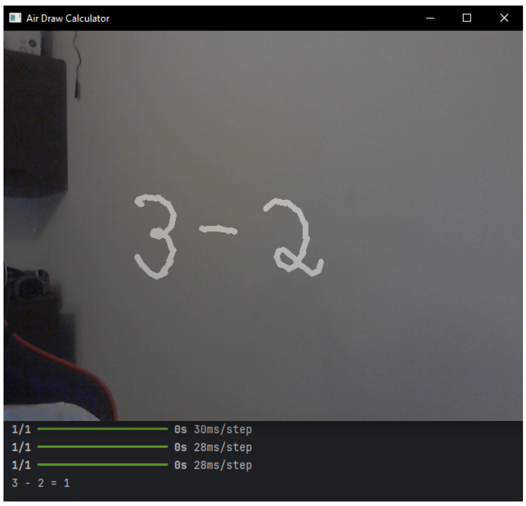
Example 4: Three Groups with an Operator ("15 * 24")

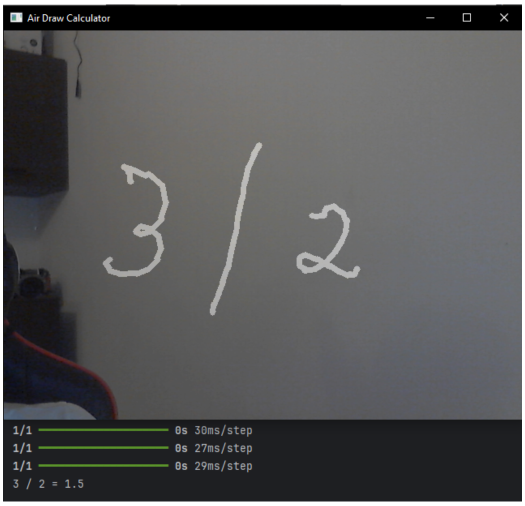
- Group 1: `"15"` → float(15.0)
- Group 2: `"*"`
- Group 3: `"24"` → float(24.0)
- The result is `15 * 24 = 360`, so the console prints `"15 * 24 = 360"`.

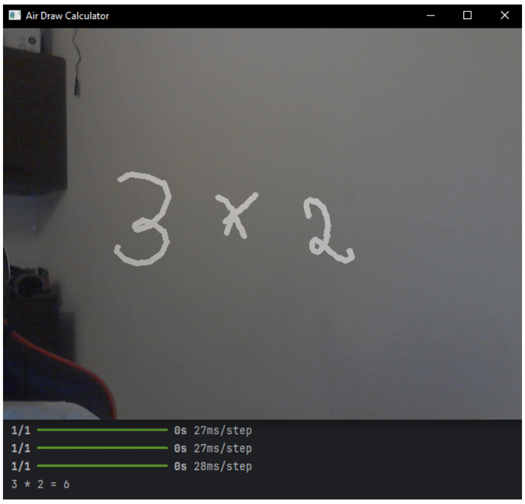
3.5.3 Visual Indicators for Operators

Below are example images (plus, minus, divide, times) that the model recognizes:









3.5.4 Summary

In summary, the program classifies written symbols into digits (possibly forming multi-digit numbers) and exactly one operator if three groups are formed. It then converts the digit strings to floats, applies the recognized operator, and handles errors such as division by zero. If there are fewer or more groups, the program does not perform any calculation, instead simply reporting the recognized digits as separate items.

3.6 Training the Digit Recognition Model: Code, Methods, Results, and Discussion

Overview: In this section, I describe how I trained a Convolutional Neural Network (CNN) on the MNIST dataset. I explain the key methods I used to boost performance, including data augmentation, specific CNN blocks, and additional layers that improved generalization. I also show the training and validation accuracy over 20 epochs and interpret the final results.

3.6.1 Model Architecture and Training Code

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.datasets import mnist
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# 1. Load MNIST
(x_train, y_train), (x_test, y_test) = mnist.load_data()

x_train = x_train.astype('float32') / 255.0
x_test  = x_test.astype('float32')  / 255.0

# Add a channel dimension: from (28,28) to (28,28,1)
x_train = np.expand_dims(x_train, axis=-1)
x_test  = np.expand_dims(x_test, axis=-1)

num_classes = 10
y_train = tf.keras.utils.to_categorical(y_train, num_classes)
y_test  = tf.keras.utils.to_categorical(y_test,  num_classes)

# 2. Data Augmentation
datagen = ImageDataGenerator(
    rotation_range=25,
    width_shift_range=0.2,
    height_shift_range=0.2,
    zoom_range=0.2,
    shear_range=10.0,
    fill_mode='constant',
    cval=0.0
)
datagen.fit(x_train)

# 3. Define the CNN Model
model = models.Sequential()

# Block 1
model.add(layers.Conv2D(32, (3,3), padding='same', input_shape=(28,28,1)))
model.add(layers.BatchNormalization())
model.add(layers.Activation('relu'))
```

```

model.add(layers.Conv2D(32, (3,3), padding='same'))
model.add(layers.BatchNormalization())
model.add(layers.Activation('relu'))
model.add(layers.MaxPooling2D((2,2)))  # Down to 14x14

# Block 2
model.add(layers.Conv2D(64, (3,3), padding='same'))
model.add(layers.BatchNormalization())
model.add(layers.Activation('relu'))

model.add(layers.Conv2D(64, (3,3), padding='same'))
model.add(layers.BatchNormalization())
model.add(layers.Activation('relu'))
model.add(layers.MaxPooling2D((2,2)))  # Down to 7x7

# Block 3
model.add(layers.Conv2D(128, (3,3), padding='same'))
model.add(layers.BatchNormalization())
model.add(layers.Activation('relu'))

model.add(layers.Conv2D(128, (3,3), padding='same'))
model.add(layers.BatchNormalization())
model.add(layers.Activation('relu'))
model.add(layers.MaxPooling2D((2,2)))  # Down to 3x3

# Dense layers
model.add(layers.Flatten())
model.add(layers.Dense(256))
model.add(layers.BatchNormalization())
model.add(layers.Activation('relu'))
model.add(layers.Dropout(0.4))

model.add(layers.Dense(128))
model.add(layers.BatchNormalization())
model.add(layers.Activation('relu'))
model.add(layers.Dropout(0.4))

model.add(layers.Dense(num_classes, activation='softmax'))

model.compile(
    optimizer='adam',
    loss='categorical_crossentropy',
    metrics=['accuracy']
)

model.summary()

# 4. Training
batch_size = 64
epochs = 20
history = model.fit(
```

```
    datagen.flow(x_train, y_train, batch_size=batch_size),
    steps_per_epoch=len(x_train)//batch_size,
    epochs=epochs,
    validation_data=(x_test, y_test)
)

# 5. Plot Accuracy
plt.plot(history.history['accuracy'], label='Train acc')
plt.plot(history.history['val_accuracy'], label='Val acc')
plt.legend()
plt.title("Training vs. Validation Accuracy")
plt.show()

# 6. Evaluate on Test Set
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=0)
print("Accuracy on test:", test_acc)

# 7. Save the Model
model.save("super_digit_classifier.keras", save_format="keras")
```

Key Points:

- **Data Augmentation:** I use rotations, shifts, zooming, and shearing to mimic a wide range of handwriting styles and random distortions.
- **CNN Blocks:** Each "block" contains two `Conv2D` layers and a `MaxPooling2D` layer. `BatchNormalization` follows each convolution to stabilize gradients, while `ReLU` provides non-linearity.
- **Layer Separation:** After three convolution blocks, I transition from the learned feature maps to fully connected layers (Dense layers). This separation is key: the lower blocks focus on extracting spatial patterns, while the dense layers specialize in the final classification.
- **Dropout & BatchNormalization in Dense Layers:** Including `Dropout(0.4)` in two successive dense layers helps prevent overfitting by randomly "dropping out" neurons. The BatchNormalization layers help reduce internal covariate shift, accelerating convergence.
- **Final Layer:** A softmax layer with 10 outputs corresponds to the ten MNIST digit classes.

3.6.2 Training Methods and Improvements

1. **Data Augmentation with `ImageDataGenerator`:**
   - *Rotation (up to 25 degrees):* Creates tilted versions of digits, forcing the network to learn rotation invariance.
   - *Width/Height Shifts (up to 20%):* Moves digits around the canvas, helping the model detect digits in varied positions.
   - *Zoom and Shear:* Simulates zoomed-in or slanted digits, improving robustness against shape distortions.
2. **Multiple Convolution Blocks:**
   - *Block 1 (32 filters), Block 2 (64 filters), Block 3 (128 filters):* Each block increases the network depth, allowing it to learn progressively abstract features (edges, corners, strokes, etc.).
   - *MaxPooling2D:* Reduces the spatial dimension after each pair of `Conv2D` layers, speeding up training and preventing oversized feature maps.
3. **Separated Dense Layers:**
   - After flattening, I feed the features into two Dense layers (256 and 128 units). This separation lets me apply dropout and further normalization specifically in the classification part of the network.
   - A dropout rate of `0.4` is relatively high but proves effective for managing overfitting on MNIST.
4. **Adam Optimizer:** I train using the `Adam` optimizer, which adapts the learning rate automatically. This optimizer usually converges quickly on MNIST.

3.6.3 Results and Interpretation

```
Epoch 1/20
937/937 47s 46ms/step - accuracy: 0.7315 - val_accuracy: 0.9515
Epoch 2/20
937/937 43s 47ms/step - accuracy: 0.9375 - val_accuracy: 0.9555
...
Epoch 15/20
937/937 accuracy: 0.9811 - val_accuracy: 0.9931
Epoch 20/20
937/937 accuracy: 0.9844 - val_accuracy: 0.9872

Accuracy on test: 0.9872000217437744
```

Training Log Snippet

- The model starts around 73% training accuracy during the first epoch, surging to over 95% on the validation set by its end.
- By later epochs, the training accuracy occasionally hits 100% on some passes, while validation remains mostly in the 95–99% range, sometimes exceeding 99%.
- The final reported test accuracy is **98.72**%, a strong result for MNIST, especially considering the extra distortions introduced by augmentation.

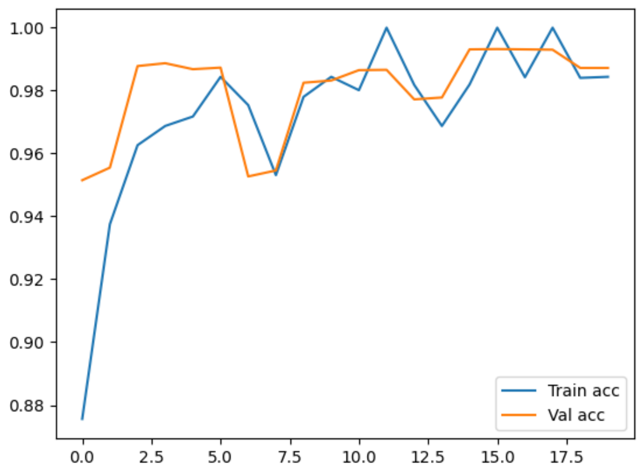


Figure 3: Training vs. Validation Accuracy over 20 epochs

Accuracy Plot Analysis

- **Rapid Early Improvement:** The model quickly learns essential digit shapes, rising from $\sim 88\%$ to $\sim 96\%$ in just a few epochs.
- **Fluctuations from Augmentation:** I observe occasional dips (e.g., around Epochs 7–8), reflecting the models adaptation to heavily augmented batches. It recovers swiftly, illustrating robustness.
- **High Final Accuracy:** After approximately 15–20 epochs, both curves stabilize above 97–98%. The training curve occasionally reaches 100%, while validation accuracy hovers around 98–99%.
- **Modest Overfitting:** Some minor gap appears between training and validation, but dropout and batch normalization help keep that gap in check.

3.6.4 Conclusion

- **Data Augmentation + CNN Blocks:** Combining augmentation with progressively deeper CNN blocks has proven effective, providing strong invariance to common handwriting distortions.
- **Separated Dense Layers with Dropout:** Including two dense layers (256 and 128 neurons) with a 40% dropout significantly reduces overfitting and improves generalization.
- **High Accuracy on MNIST (98.72%):** This confirms that my model is well-tuned for digit recognition tasks, thanks to the chosen methods (augmentation, block architecture, batch normalization, and dropout).
- **Practical Implications:** I saved the final network as `super_digit_classifier.keras`. This model can be easily loaded for inference or integrated into other applications (e.g., a digit recognition pipeline or an air-writing calculator).

3.7 Training the Operator Recognition Model: Code, Methods, Results, and Discussion

Overview: In this section, I explain how I trained a Convolutional Neural Network (CNN) to recognize four mathematical operators: addition (`add`), division (`divide`), multiplication (`multiply`), and subtraction (`subtract`). I used a folder-based dataset with separate directories for train, validation, and test splits. Below is the full code and a discussion of the training process, data augmentation, and final accuracy results.

3.7.1 Model Architecture and Training Code

```python
import os
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.preprocessing.image import ImageDataGenerator

# Paths to train, val, test folders
base_dir = r"C:\Users\Stanislav\Downloads\archive (5)\final_symbols_split_ttv"
train_dir = os.path.join(base_dir, "train")
val_dir   = os.path.join(base_dir, "val")
test_dir  = os.path.join(base_dir, "test")

# 1) Generators
train_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=25,
    width_shift_range=0.2,
    height_shift_range=0.2,
    zoom_range=0.2,
    shear_range=10.0,
    fill_mode='constant',
    cval=0.0
)
val_datagen = ImageDataGenerator(rescale=1./255)
test_datagen = ImageDataGenerator(rescale=1./255)

batch_size = 32
img_size = (28, 28)  # 28x28, grayscale

train_generator = train_datagen.flow_from_directory(
    directory=train_dir,
    target_size=img_size,
    color_mode='grayscale',
    class_mode='categorical',
    batch_size=batch_size,
    shuffle=True
)

val_generator = val_datagen.flow_from_directory(
```

```
    directory=val_dir,
    target_size=img_size,
    color_mode='grayscale',
    class_mode='categorical',
    batch_size=batch_size,
    shuffle=False
)

test_generator = test_datagen.flow_from_directory(
    directory=test_dir,
    target_size=img_size,
    color_mode='grayscale',
    class_mode='categorical',
    batch_size=batch_size,
    shuffle=False
)

num_classes = train_generator.num_classes
print("Number of classes:", num_classes)
print("class_indices (train):", train_generator.class_indices)

# 2) CNN Model
model = models.Sequential()

# Block 1
model.add(layers.Conv2D(32, (3,3), padding='same', input_shape=(28,28,1)))
model.add(layers.BatchNormalization())
model.add(layers.Activation('relu'))

model.add(layers.Conv2D(32, (3,3), padding='same'))
model.add(layers.BatchNormalization())
model.add(layers.Activation('relu'))
model.add(layers.MaxPooling2D((2,2)))  # -> 14x14

# Block 2
model.add(layers.Conv2D(64, (3,3), padding='same'))
model.add(layers.BatchNormalization())
model.add(layers.Activation('relu'))

model.add(layers.Conv2D(64, (3,3), padding='same'))
model.add(layers.BatchNormalization())
model.add(layers.Activation('relu'))
model.add(layers.MaxPooling2D((2,2)))  # -> 7x7

# Block 3
model.add(layers.Conv2D(128, (3,3), padding='same'))
model.add(layers.BatchNormalization())
model.add(layers.Activation('relu'))

model.add(layers.Conv2D(128, (3,3), padding='same'))
model.add(layers.BatchNormalization())
model.add(layers.Activation('relu'))
```

```
model.add(layers.MaxPooling2D((2,2)))  # -> 3x3

# Dense part
model.add(layers.Flatten())
model.add(layers.Dense(256))
model.add(layers.BatchNormalization())
model.add(layers.Activation('relu'))
model.add(layers.Dropout(0.4))

model.add(layers.Dense(128))
model.add(layers.BatchNormalization())
model.add(layers.Activation('relu'))
model.add(layers.Dropout(0.4))

model.add(layers.Dense(num_classes, activation='softmax'))

model.compile(
    optimizer='adam',
    loss='categorical_crossentropy',
    metrics=['accuracy']
)

model.summary()

# 3) Training
epochs = 20
history = model.fit(
    train_generator,
    epochs=epochs,
    validation_data=val_generator
)

# 4) Plot Accuracy
plt.plot(history.history['accuracy'], label='Train acc')
plt.plot(history.history['val_accuracy'], label='Val acc')
plt.legend()
plt.title("Operator Model Accuracy")
plt.show()

# 5) Test Evaluation
test_loss, test_acc = model.evaluate(test_generator, verbose=0)
print("Test accuracy:", test_acc)

# 6) Save the Model
model.save("operators_model_aiaiai.keras")
print("Model saved: operators_model.keras")

# 7) Inverse Class Dictionary
class_indices = train_generator.class_indices
inv_map = {v: k for k,v in class_indices.items()}
print("Inverse map:", inv_map)
```

Key Points:

- **Dataset:** The dataset is split into `train`, `val`, and `test` folders, each containing four subdirectories: `add`, `divide`, `multiply`, and `subtract`.
- **Data Augmentation for Training:**
  - *rotation_range=25, width_shift_range=0.2, height_shift_range=0.2, zoom_range=0.2, shear_range=10.0*
  - This helps the network handle variations in operator drawings, scale, and position.
- **Validation and Test Generators:** Only rescaling is applied (no augmentation). This ensures the validation and test sets remain consistent for accurate performance checks.
- **CNN Blocks:** Like before, I employ three convolution blocks, each containing two `Conv2D` + `BatchNormalization` layers and a `MaxPooling2D` layer.
- **Dense Layers:** I flatten the feature maps, then use two fully connected layers (256, 128) with `Dropout(0.4)`, `BatchNormalization`, and `ReLU`.
- **Final Layer:** A softmax output for the 4 classes (`add`, `divide`, `multiply`, `subtract`).

3.7.2 Results and Interpretation

```
Epoch 1/20
364/364 ... accuracy: 0.7753 - val_accuracy: 0.2045
Epoch 2/20
364/364 ... accuracy: 0.9501 - val_accuracy: 0.9193
Epoch 7/20
364/364 ... accuracy: 0.9833 - val_accuracy: 0.9767
Epoch 14/20
364/364 ... accuracy: 0.9872 - val_accuracy: 0.2604
Epoch 17/20
364/364 ... accuracy: 0.9906 - val_accuracy: 0.9974
Epoch 20/20
364/364 ... accuracy: 0.9933 - val_accuracy: 0.9979

Test accuracy: 0.9994066953659058
```

Training Log Snippet

- The training accuracy quickly escalates above 95% by epoch 2, showing that the model is capable of distinguishing most operator images with few errors.
- Validation accuracy fluctuates (e.g., it drops to around 20% at epoch 1, 26% at epoch 14, then jumps to above 99%). These strong oscillations may be due to differences in how augmented data aligns with the small-ish validation set, or potentially class imbalance in certain subsets.
- By the end of training (epoch 20), validation accuracy sits near 99.79%, and **test accuracy is about 99.94%**.

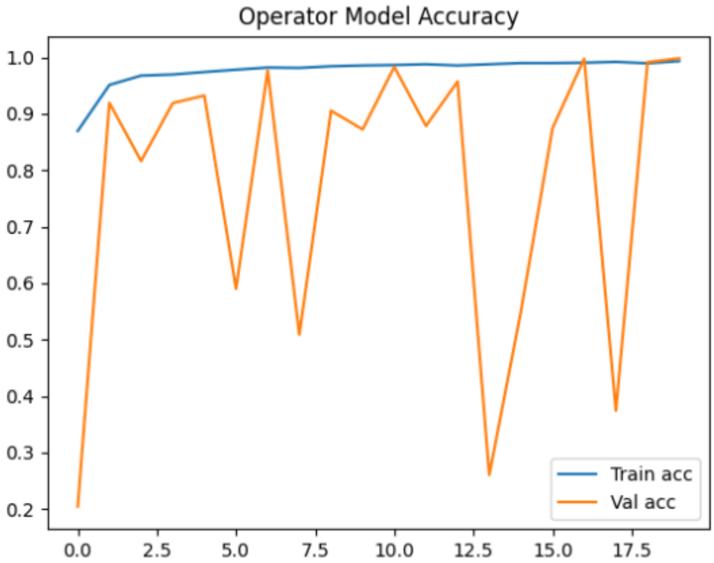


Figure 4: Train vs. Validation Accuracy for the Operator Model

Accuracy Plot Analysis

- **High and Steady Training Accuracy:** The blue line (train accuracy) remains above 97–98% from early in the training, reaching nearly 99–100% by later epochs.
- **Significant Validation Variations:** The orange line (val accuracy) shows big swings (from around 20% to over 90–95% or higher). These fluctuations might occur if certain batches in the validation set are more challenging or if there's a class imbalance that occasionally confuses the model.
- **Final Convergence:** Despite the dips, the validation accuracy ends extremely high (around 99–100%), indicating the network eventually generalizes well.

3.7.3 Conclusion

- **Data Augmentation for Operators:** The strategy of random rotations, shifts, zooming, and shearing helps the model handle a diverse set of handwritten operator images.
- **Robust CNN Blocks + Dense Layers:** Repeating convolution blocks and adding two dense layers with dropout effectively reduce overfitting, even though occasional validation drops appear
- **Excellent Final Performance:** A test accuracy of **99.94%** confirms that the model distinguishes `add`, `divide`, `multiply`, and `subtract` with near-perfect accuracy on the test set.

```latex

# 4 Conclusion

In this report, I presented the development of the "Air Calculator" system, which enables the recognition of handwritten mathematical expressions through real-time hand gesture tracking and convolutional neural networks (CNNs). I detailed every aspect of the project-from the foundational hardware and functional assumptions to the comprehensive software design and the training of both digit and operator recognition models. My key conclusions are summarized below:

- **Innovative User Interaction:** I developed a system that allows users to write mathematical expressions in mid-air using hand gestures, leveraging technologies such as Mediapipe for hand tracking and OpenCV for image processing. This approach results in an intuitive and interactive user experience.
- **Accurate Hand and Finger Tracking:** By focusing on precise tracking of the index fingertip and incorporating a thumb gesture for mode switching, I ensured that the system reliably captures drawing inputs and manages actions like drawing and erasing on a virtual canvas.
- **Effective Input Segmentation and Processing:** I implemented an efficient segmentation process that divides the drawn symbols into distinct groups, identifying operators and operands. For standard arithmetic expressions (e.g., `"number -- operator -- number"`), the system accurately performs the corresponding calculations while gracefully handling errors (such as division by zero).
- **High-Performing Recognition Models:** The digit recognition model, which I trained on the MNIST dataset using data augmentation and a robust CNN architecture, achieved an accuracy of approximately 98.72%. Similarly, the operator recognition model demonstrated an exceptional test accuracy of around 99.94%. These results confirm the reliability and robustness of the recognition models under various handwriting distortions.
- **Comprehensive System Design:** I provided detailed explanations of the CNN architectures, data augmentation techniques, and modular software components. This comprehensive approach not only clarifies the systems operation but also facilitates future improvements and integrations into other applications.

In summary, I successfully demonstrated the potential of modern computer vision and deep learning technologies to create an interactive tool for recognizing handwritten mathematical expressions. The high accuracy of the models and the intuitive gesture-based control of the system open up promising avenues for further development in educational, commercial, and research settings.
```

Appendix

The full Python code is listed below:

```python
import cv2
import mediapipe as mp
import numpy as np
from tensorflow.keras.models import load_model

print("Starting Air Calculator...")

digits_model = load_model("super_digit_classifier.keras")
operators_model = load_model("operators_model.keras")

mp_hands = mp.solutions.hands
mp_drawing = mp.solutions.drawing_utils

canvas = None
previous_position = None

DOT_IGNORE_AREA = 10
DOT_AREA_THRESHOLD = 40

def crop_and_predict_digit_or_dot(cropped):
    """Predict digit 0..9 or '.' or None if area too small. Uses verbose=1."""
    h, w = cropped.shape[:2]
    area = w*h
    if area < DOT_IGNORE_AREA:
        return None
    if area < DOT_AREA_THRESHOLD:
        return '.'
    max_side = max(w, h)
    scale = 20.0 / max_side
    new_w = int(w * scale)
    new_h = int(h * scale)
    resized = cv2.resize(cropped, (new_w, new_h), interpolation=cv2.INTER_AREA)
    out28 = np.zeros((28, 28), dtype=np.uint8)
    sx = (28 - new_w) // 2
    sy = (28 - new_h) // 2
    out28[sy:sy+new_h, sx:sx+new_w] = resized
    final_28 = out28.astype('float32') / 255.0
    x = final_28.reshape(1, 28, 28, 1)
    preds = digits_model.predict(x, verbose=1)  # verbose=1
    d = np.argmax(preds)
    return str(d)

def crop_and_predict_operator(cropped):
    """Predict + / * - with inversion. Uses verbose=1 for progress."""
    h, w = cropped.shape[:2]
    max_side = max(w, h)
    scale = 20.0 / max_side
    new_w = int(w * scale)
```

```
    new_h = int(h * scale)
    resized = cv2.resize(cropped, (new_w, new_h), interpolation=cv2.INTER_AREA)
    out28 = np.zeros((28, 28), dtype=np.uint8)
    sx = (28 - new_w) // 2
    sy = (28 - new_h) // 2
    out28[sy:sy+new_h, sx:sx+new_w] = resized
    out28 = 255 - out28
    final_28 = out28.astype('float32') / 255.0
    x = final_28.reshape(1, 28, 28, 1)
    preds = operators_model.predict(x, verbose=1)
    idx = np.argmax(preds)
    op_map = {0: '+', 1: '/', 2: '*', 3: '-'}
    return op_map.get(idx, '?')

def preprocess_and_find_contours(img):
    """Threshold + dilate => contours."""
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    _, bin_canvas = cv2.threshold(gray, 127, 255, cv2.THRESH_BINARY)
    kernel = np.ones((3,3), np.uint8)
    bin_dil = cv2.dilate(bin_canvas, kernel, iterations=1)
    contours, _ = cv2.findContours(bin_dil, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
    return bin_dil, contours

def split_line_into_groups(boxes, space_x_thresh=15):
    """Sort by x, group bounding boxes if gap <= space_x_thresh."""
    boxes.sort(key=lambda b: b[0])
    sublines = []
    current = [boxes[0]]
    for i in range(1, len(boxes)):
        x,y,w,h = boxes[i]
        px,py,pw,ph = boxes[i-1]
        gap = x - (px+pw)
        if gap > space_x_thresh:
            sublines.append(current)
            current = [boxes[i]]
        else:
            current.append(boxes[i])
    if current:
        sublines.append(current)
    return sublines

def process_subgroup(bin_dil, boxes):
    """Merge all bounding boxes => string of digits or '.'."""
    boxes.sort(key=lambda b: b[0])
    s = ""
    for (x,y,w,h) in boxes:
        cropped = bin_dil[y:y+h, x:x+w]
        c = crop_and_predict_digit_or_dot(cropped)
        if c is not None:
            s += c
    if s.endswith('.'):
        s = s[:-1]
```

```
    if s == "":
        s = "0"
    return s

def process_operator_subgroup(bin_dil, boxes):
    """Assume first bounding box is operator."""
    boxes.sort(key=lambda b: b[0])
    x,y,w,h = boxes[0]
    cropped = bin_dil[y:y+h, x:x+w]
    return crop_and_predict_operator(cropped)

def predict_all_in_one_line(canvas_img):
    """We read all bounding boxes as if they are in one single line."""
    bin_dil, contours = preprocess_and_find_contours(canvas_img)
    boxes = []
    for c in contours:
        x,y,w,h = cv2.boundingRect(c)
        boxes.append((x,y,w,h))
    if not boxes:
        return
    sublines = split_line_into_groups(boxes, space_x_thresh=15)
    if len(sublines) == 3:
        # interpret as equation
        left_str = process_subgroup(bin_dil, sublines[0])
        op_str = process_operator_subgroup(bin_dil, sublines[1])
        right_str = process_subgroup(bin_dil, sublines[2])
        try:
            lv = float(left_str)
            rv = float(right_str)
            res = None
            if op_str == '+': res = lv + rv
            elif op_str == '-': res = lv - rv
            elif op_str == '*': res = lv * rv
            elif op_str == '/':
                if rv == 0: res = "ERR(div0)"
                else: res = lv / rv
            if isinstance(res, float) and not isinstance(res, str):
                if abs(res - round(res)) < 1e-9:
                    res = int(round(res))
            print(f"{left_str} {op_str} {right_str} = {res}")
        except:
            print(f"{left_str} {op_str} {right_str} = ERR")
    else:
        # multiple subgroups => print them with semicolon
        results = []
        for grp in sublines:
            val = process_subgroup(bin_dil, grp)
            results.append(val)
        print("; ".join(results))

def is_thumb_extended(landmarks):
    thumb_tip = landmarks.landmark[4]
```

```
    thumb_base = landmarks.landmark[2]
    return abs(thumb_tip.x - thumb_base.x) > 0.05

def is_hand_open(landmarks):
    tip_ids = [8,12,16,20]
    base_ids = [6,10,14,18]
    for tip_id, base_id in zip(tip_ids, base_ids):
        if landmarks.landmark[tip_id].y > landmarks.landmark[base_id].y:
            return False
    return True

def erase_area(landmarks, canvas_img, shape):
    try:
        palm_ids = [0,1,5,9,13,17]
        pls = [landmarks.landmark[i] for i in palm_ids]
        xs = [int(lm.x*shape[1]) for lm in pls]
        ys = [int(lm.y*shape[0]) for lm in pls]
        x_min, x_max = min(xs), max(xs)
        y_min, y_max = min(ys), max(ys)
        cv2.rectangle(canvas_img, (x_min, y_min), (x_max, y_max), (0,0,0), -1)
    except:
        for fid in [8,12,16,20]:
            fx = int(landmarks.landmark[fid].x*shape[1])
            fy = int(landmarks.landmark[fid].y*shape[0])
            cv2.circle(canvas_img, (fx, fy), 20, (0,0,0), -1)

def main():
    global canvas, previous_position
    mp_hands = mp.solutions.hands
    mp_drawing = mp.solutions.drawing_utils
    cap = cv2.VideoCapture(0)
    with mp_hands.Hands(model_complexity=0, min_detection_confidence=0.8,
                        min_tracking_confidence=0.5) as hands:
        while True:
            ret, frame = cap.read()
            if not ret:
                break
            frame = cv2.flip(frame, 1)
            rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
            if canvas is None:
                canvas = np.zeros_like(frame)
            results = hands.process(rgb)
            if results.multi_hand_landmarks:
                for lm in results.multi_hand_landmarks:
                    mp_drawing.draw_landmarks(frame, lm, mp_hands.HAND_CONNECTIONS)
                    fx = int(lm.landmark[8].x * frame.shape[1])
                    fy = int(lm.landmark[8].y * frame.shape[0])
                    if is_hand_open(lm):
                        erase_area(lm, canvas, frame.shape)
                        continue
                    if is_thumb_extended(lm):
                        if previous_position is not None:
```

```
                            cv2.line(canvas, previous_position, (fx, fy), (255,255,255), 5)
                        previous_position = (fx, fy)
                    else:
                        previous_position = None
            else:
                previous_position = None
            combined = cv2.addWeighted(frame, 0.7, canvas, 0.3, 0)
            cv2.imshow("Air Draw Calculator", combined)
            key = cv2.waitKey(1) & 0xFF
            if key == ord('q'):
                break
            elif key == ord('r'):
                predict_all_in_one_line(canvas)
            elif key == ord('c'):
                canvas = np.zeros_like(frame)
        cap.release()
        cv2.destroyAllWindows()

if __name__ == "__main__":
    main()
```