HotSpot

Romanian National AI Olympiad 2025 - Problem Editorial

AI Olympiad Committee

June 23, 2025

Abstract

This editorial presents a comprehensive analysis of the HotSpot problem, an image segmentation challenge set in a dystopian future where WiFi signals represent areas of clean communication. The problem requires participants to identify geometric shapes (HotSpots) in noisy satellite imagery while dealing with various levels of interference and background noise. We explore multiple solution approaches ranging from simple thresholding to advanced region growing algorithms, analyzing their effectiveness across different subtasks.

Contents

1 Problem Overview

1.1 Context and Motivation

The HotSpot problem is set in the year 2147, where humanity has transitioned to a completely wireless society. In this dystopian world, WiFi signals serve as the primary source of energy, communication, and control. The challenge lies in identifying clean signal areas (HotSpots) from satellite imagery that is corrupted by radio interference and background noise.

We designed this problem to be something unique rather than just applying classical deep learning models. While HotSpot is fundamentally a segmentation problem dealing with basic geometric shapes, the variety of possible solution approaches is staggering. This diversity allows participants to explore different algorithmic paradigms and discover that sometimes simpler, more traditional computer vision techniques can outperform modern deep learning approaches.

1.2 Technical Specification

Participants must process four sets of 256×256 RGB images:

- **Satellite_Images-1**: Images with at least one object, minimal noise
- **Satellite_Images-2**: Images with multiple objects, minimal noise
- **Satellite_Images-3**: Images with moderate complexity
- **Satellite_Images-4**: Images with maximum complexity and interference

The goal is to generate binary masks identifying HotSpots, encoded using Run-Length Encoding (RLE) for efficient storage and evaluation.

1.3 Geometric Shapes

HotSpots appear as various geometric primitives:

- Circles and ellipses
- Rectangles and squares
- Convex and concave polygons

2 Suggested Solution Approaches

2.1 Baseline Solutions

2.1.1 Simple Thresholding (Subtasks 1-2)

For the first two subtasks, where images contain minimal noise, we suggest a straightforward approach that works effectively:

Algorithm 1 Simple Non-Black Pixel Detection

Require: RGB image I of size 256×256
Ensure: Binary mask M

for each pixel (i, j) in I **do**
  if $I[i, j] \neq (0, 0, 0)$ **then**
    $M[i, j] \leftarrow 1$
  else
    $M[i, j] \leftarrow 0$
  end if
end for
return M

This approach leverages the clean nature of early subtasks where HotSpots are clearly distinguished from the black background.

2.1.2 K-Means Clustering

A more sophisticated baseline suggestion uses K-means clustering to separate foreground objects from background:

Listing 1: K-Means Segmentation

```python
import numpy as np
from sklearn.cluster import KMeans

def kmeans_segmentation(image, k=2):
    # Reshape image to pixel array
    pixels = image.reshape(-1, 3)

    # Apply K-means clustering
    kmeans = KMeans(n_clusters=k, random_state=42)
    labels = kmeans.fit_predict(pixels)

    # Reshape back to image dimensions
    segmented = labels.reshape(image.shape[:2])

    # Identify foreground cluster (non-black)
    cluster_means = kmeans.cluster_centers_
    foreground_cluster = np.argmax(np.sum(cluster_means, axis=1))

    return (segmented == foreground_cluster).astype(int)
```

2.2 Advanced Suggestions

2.2.1 Region Growing with Similarity-Based Expansion

One of the most effective suggested approaches for complex subtasks employs region growing based on pixel similarity within local neighborhoods:

Algorithm 2 Region Growing Segmentation

Require: RGB image I, similarity threshold τ, neighborhood size N
Ensure: Binary mask M

```
Initialize M ← 0
Initialize seed points S using brightness thresholding
for each seed point s ∈ S do
    Initialize queue Q ← {s}
    M[s] ← 1
    while Q ≠ ∅ do
        p ← dequeue(Q)
        for each neighbor n of p in N-neighborhood do
            if M[n] = 0 AND similarity(I[p], I[n]) > τ then
                M[n] ← 1
                enqueue(Q, n)
            end if
        end for
    end while
end for
return M
```

The similarity function can be defined as:

$$\text{similarity}(p_1, p_2) = \exp\left(-\frac{\|p_1 - p_2\|_2^2}{2\sigma^2}\right)$$

where σ controls the sensitivity to color differences.

2.2.2 Cellular Automata Segmentation

An interesting alternative approach employs cellular automata for image segmentation. This method treats each pixel as a cell in a 2D grid, where the cell's state evolves based on local neighborhood rules. Surprisingly, this bio-inspired approach often produces results very similar to K-means clustering while offering different computational characteristics.

Algorithm 3 Cellular Automata Segmentation

Require: RGB image I, iterations T, threshold θ
Ensure: Binary mask M

```
Initialize state matrix S based on pixel brightness
for t = 1 to T do
    S_new ← S
    for each pixel (i, j) in I do
        Count active neighbors N_active ← Σ_{(x,y)∈N(i,j)} S[x, y]
        Compute local brightness B ← ‖I[i, j]‖_2
        if B > θ AND N_active ≥ 2 then
            S_new[i, j] ← 1 {Cell becomes active}
        else if B ≤ θ AND N_active < 3 then
            S_new[i, j] ← 0 {Cell becomes inactive}
        end if
    end for
    S ← S_new
end for
M ← S
return M
```

The cellular automata approach uses simple local rules that create emergent segmentation behavior. The update rules can be customized based on the specific characteristics of the HotSpots:

Listing 2: Cellular Automata Implementation

```python
import numpy as np
from scipy.ndimage import convolve

def cellular_automata_segmentation(image, iterations=10,
    brightness_threshold=0.3):
    # Convert to grayscale and normalize
    gray = np.mean(image, axis=2) / 255.0

    # Initialize cellular automata state
    state = (gray > brightness_threshold).astype(int)

    # Define neighborhood kernel (Moore neighborhood)
    kernel = np.array([[1, 1, 1],
                       [1, 0, 1],
                       [1, 1, 1]])

    for iteration in range(iterations):
        # Count active neighbors
        neighbor_count = convolve(state, kernel, mode='constant', cval
            =0)

        # Apply cellular automata rules
        new_state = state.copy()

        # Rule 1: High brightness pixels with sufficient neighbors
            become active
        activate_mask = (gray > brightness_threshold) & (neighbor_count
            >= 2)
        new_state[activate_mask] = 1

        # Rule 2: Low brightness pixels with few neighbors become
            inactive
        deactivate_mask = (gray <= brightness_threshold) & (
            neighbor_count < 3)
        new_state[deactivate_mask] = 0

        # Rule 3: Stability rule - maintain state if conditions are
            ambiguous
        stable_mask = (neighbor_count >= 3) & (neighbor_count <= 5)
        new_state[stable_mask] = state[stable_mask]

        state = new_state

    return state
```

Interestingly, the cellular automata approach often converges to segmentation results that are remarkably similar to those obtained through K-means clustering. This similarity arises because both methods attempt to group pixels with similar characteristics, though through fundamentally different mechanisms: K-means through explicit distance minimization in feature space, and cellular automata through local neighborhood interactions that create emergent clustering behavior.

The key advantages of cellular automata segmentation include:

- **Inherent spatial coherence**: The local neighborhood rules naturally preserve spatial relationships
- **Noise robustness**: The iterative nature helps filter out isolated noise pixels
- **Computational efficiency**: Simple local operations can be easily parallelized
- **Adaptability**: Rules can be easily modified for different image characteristics

2.2.3 Brightness Thresholding with Morphological Operations

We suggest this approach that combines intensity-based segmentation with morphological post-processing:

Listing 3: Thresholding with Morphology

```python
import cv2
from skimage import morphology

def threshold_morphology_segmentation(image, threshold=0.3):
    # Convert to grayscale and normalize
    gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)
    normalized = gray.astype(float) / 255.0

    # Apply adaptive thresholding
    binary_mask = normalized > threshold

    # Morphological operations
    # Remove noise
    cleaned = morphology.remove_small_objects(binary_mask, min_size=50)

    # Fill holes
    filled = morphology.remove_small_holes(cleaned, area_threshold=100)

    # Smooth boundaries
    kernel = morphology.disk(2)
    final_mask = morphology.closing(filled, kernel)

    return final_mask.astype(int)
```

2.3 Machine Learning Suggestion

For subtasks 3 and 4, a supervised learning approach can be considered, though interestingly, this problem demonstrates that modern deep learning architectures like U-Net do not perform well due to the specific nature of the geometric shapes and noise patterns involved.

2.3.1 Training Data Preparation

Using the clean results from subtasks 1 and 2 as training data:

Listing 4: Feature Extraction for ML

```python
def extract_features(image, window_size=5):
    features = []
    h, w = image.shape[:2]

    for i in range(window_size//2, h - window_size//2):
        for j in range(window_size//2, w - window_size//2):
            # Extract local window
```

```
            window = image[i-window_size//2:i+window_size//2+1,
                           j-window_size//2:j+window_size//2+1]

            # Compute features
            mean_rgb = np.mean(window, axis=(0,1))
            std_rgb = np.std(window, axis=(0,1))
            local_contrast = np.std(cv2.cvtColor(window, cv2.
                COLOR_RGB2GRAY))

            features.append(np.concatenate([mean_rgb, std_rgb, [
                local_contrast]]))

    return np.array(features)
```

2.3.2 Segmentation Classifier

A Random Forest or SVM classifier can be trained on these features:

Listing 5: Segmentation Classifier

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC

def train_segmentation_classifier(train_images, train_masks):
    X, y = [], []

    for img, mask in zip(train_images, train_masks):
        features = extract_features(img)
        labels = mask.flatten()

        X.extend(features)
        y.extend(labels)

    # Train classifier
    classifier = RandomForestClassifier(n_estimators=100, random_state
        =42)
    classifier.fit(X, y)

    return classifier

def predict_segmentation(classifier, image):
    features = extract_features(image)
    predictions = classifier.predict(features)

    return predictions.reshape(image.shape[:2])
```

3 Handling Noise and Interference

3.1 Types of Interference

The problem specifies two main sources of interference:

- **Random background noise**: Gaussian or salt-and-pepper noise
- **Semi-transparent stripes**: Linear interference patterns

3.2 Noise Reduction Strategies

3.2.1 Gaussian Filtering

Apply Gaussian blur to reduce high-frequency noise:

$$G_\sigma(x, y) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right)$$

3.2.2 Median Filtering

Effective for salt-and-pepper noise removal while preserving edges.

3.2.3 Stripe Detection and Removal

For semi-transparent stripes, use Hough transform or directional filtering:

Listing 6: Stripe Removal

```python
def remove_stripes(image, angle_range=(-90, 90)):
    # Convert to grayscale
    gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)

    # Apply Hough line detection
    lines = cv2.HoughLines(gray, 1, np.pi/180, threshold=100)

    # Create mask for detected lines
    line_mask = np.zeros_like(gray)

    if lines is not None:
        for line in lines:
            rho, theta = line[0]
            a = np.cos(theta)
            b = np.sin(theta)
            x0 = a * rho
            y0 = b * rho

            # Draw line on mask
            pt1 = (int(x0 + 1000*(-b)), int(y0 + 1000*(a)))
            pt2 = (int(x0 - 1000*(-b)), int(y0 - 1000*(a)))
            cv2.line(line_mask, pt1, pt2, 255, 2)

    # Inpaint to remove stripes
    result = cv2.inpaint(image, line_mask, 3, cv2.INPAINT_TELEA)
    return result
```

4 Suggested Implementation Strategy

4.1 Progressive Approach

We recommend the following progression:

1. **Subtasks 1-2**: Consider using simple thresholding to establish baseline results
2. **Subtask 3**: Try applying region growing or supervised learning trained on 1-2
3. **Subtask 4**: Experiment with combining noise reduction with advanced segmentation

4.2 Parameter Tuning Suggestions

Key parameters that may require optimization:

- Similarity threshold τ for region growing
- Brightness threshold for initial segmentation
- Morphological kernel sizes
- Noise reduction filter parameters
- Number of iterations for cellular automata

4.3 Validation Strategy

Consider using cross-validation on subtasks 1-2 to optimize parameters before applying to harder subtasks.

5 Conclusion

The HotSpot problem demonstrates that effective image segmentation doesn't always require the most sophisticated deep learning models. Through progressive difficulty across subtasks, participants can explore various algorithmic approaches and discover that traditional computer vision techniques often provide robust solutions for geometric shape segmentation tasks.