FlareNet Adaptive Learning System - Complete Technical Guide

▯ Table of Contents

▯ System Overview

FlareNet's Adaptive Learning System implements **human-in-the-loop machine learning** for thermal image anomaly detection without retraining the core PatchCore model. Instead, it adapts OpenCV-based classification parameters based on user feedback.

Key Features

- ▯ **No Model Retraining**: Preserves the trained PatchCore segmentation model
- ▯ **Real-time Adaptation**: Parameters adjust immediately after user feedback
- ▯ **Comprehensive Tracking**: JSON and CSV logs of all parameter changes
- ▯ **Visualization**: Graphical trends of parameter evolution
- ▯ **Reset Capability**: Return to default parameters anytime
- ▯ **Multi-tier Integration**: React → Java → Python → Database

---

▯ Architecture Flow

```
┌─────────────────┐   ┌─────────────────┐   ┌─────────────────┐
│   React Frontend │───│   Java Backend   │───│  Python ML Engine│
│                 │   │  (Spring Boot)  │   │    (FastAPI)    │
└─────────────────┘   └─────────────────┘   └─────────────────┘
        │                     │                     │
        │                     │                     │
     User Annotations        Database Storage      Parameter Adaptation
        │                     │                     │
        v                     v                     v
┌─────────────────┐   ┌─────────────────┐   ┌─────────────────┐
│ JSON Annotations│───│PostgreSQL Tables│───│ JSON/CSV Tracking│
│                 │   │                 │   │                 │
└─────────────────┘   └─────────────────┘   └─────────────────┘
```

Process Flow:

1. **User Action**: Corrects annotations in React frontend
2. **Java Processing**: Saves to database, formats data
3. **Python Adaptation**: Receives feedback, updates parameters
4. **Tracking**: Logs changes in JSON/CSV format
5. **Future Analysis**: Uses updated parameters for new images

---

▯ Mathematical Foundation

1. Threshold Adaptation Formula

The system adapts the **percent_threshold** which controls detection sensitivity:

```
K = 1.1 + (percent_threshold / 100) × (2.1 - 1.1)
anomaly_threshold = mean(anomaly_map) + K × std(anomaly_map)
```

Where:

- `K ∈ [1.1, 2.1]` : Sensitivity multiplier
- Lower `percent_threshold` → Lower `K` → More sensitive detection
- Higher `percent_threshold` → Higher `K` → Less sensitive detection

2. Parameter Update Logic

False Negative Correction:

```
# Increase sensitivity (lower threshold)
new_threshold = max(10, current_threshold - 3)
new_min_area = max(0.0005, current_min_area × 0.8)
```

False Positive Correction:

```
# Decrease sensitivity (higher threshold)
new_threshold = min(90, current_threshold + 3)
new_min_area = min(0.005, current_min_area × 1.2)
```

3. HSV Color Classification

```
# Thermal warm detection
warm_hue = (H/180 ≤ 0.17) OR (H/180 ≥ 0.95)
warm_sat = S/255 ≥ 0.35
warm_val = V/255 ≥ 0.5

# Color severity classification
red_range = (H ≤ 10) OR (H ≥ 160)
orange_range = 10 < H ≤ 25
yellow_range = 25 < H ≤ 35
```

4. Geometric Classification Rules

```
# Loose joint detection
area_fraction = detection_area / total_image_area
is_loose_joint = (area_fraction ≥ loose_joint_area_min) AND
                 (overlap_fraction ≥ loose_joint_overlap_min)

# Wire overload detection
aspect_ratio = max(width, height) / min(width, height)
is_wire = (aspect_ratio ≥ wire_aspect_ratio) AND
          (area_fraction ≥ wire_overload_area)
```

🐍 Python Components Deep Dive

1. `adaptive_params.py` - Parameter Management Core

```
class AdaptiveParams:
    def __init__(self):
        self.default_params = {
            "percent_threshold": 50,        # Detection sensitivity
            "min_area_factor": 0.001,       # Minimum blob area
            "hsv_warm_thresholds": {...},   # Color detection
            "color_classification": {...},   # Severity colors
            "geometric_rules": {...},        # Shape classification
            "severity_rules": {...},         # Fault classification
            "confidence_factors": {...}      # Confidence calculation
        }

    def adapt_from_feedback(self, feedback_analysis):
        """Core adaptation logic"""
        feedback_type = feedback_analysis["type"]

        if feedback_type == "false_negative":
            self._increase_sensitivity()
        elif feedback_type == "false_positive":
            self._reduce_sensitivity()
        elif feedback_type == "bbox_resize":
            self._adapt_geometric_rules()
        # ... other adaptations

    def _increase_sensitivity(self):
        """Make detection more sensitive for missed anomalies"""
        current = self.current_params["percent_threshold"]
        self.current_params["percent_threshold"] = max(10, current - 3)

    def reset_to_defaults(self):
        """Reset all parameters to original values"""
        self.current_params = self.default_params.copy()
        self.save_params()
```

Key Functions:

- `load_params()` : Load saved parameters or defaults
- `save_params()` : Persist parameters to JSON file
- `adapt_from_feedback()` : Main adaptation logic dispatcher
- `get_current_*()` : Parameter access methods
- `reset_to_defaults()` : Reset functionality

2. `feedback_handler.py` - User Feedback Analysis

```python
class FeedbackHandler:
    def process_user_feedback(self, image_id, user_id,
                             original_detections, user_corrections):
        """Main feedback processing pipeline"""

        # 1. Store parameters before changes
        params_before = adaptive_params.current_params.copy()

        # 2. Analyze feedback patterns
        feedback_analysis = self._analyze_feedback(
            original_detections, user_corrections
        )

        # 3. Apply parameter adaptations
        adaptations_applied = []
        for analysis in feedback_analysis:
            adaptive_params.adapt_from_feedback(analysis)
            adaptations_applied.append(analysis["type"])

        # 4. Track parameter changes
        if adaptations_applied:
            parameter_tracker.log_parameter_change(
                image_id, user_id, params_before,
                adaptive_params.current_params.copy(),
                adaptations_applied, detection_counts
            )

    def _analyze_feedback(self, original, corrected):
        """Detect feedback patterns"""
        analyses = []

        # Create ID mappings
        orig_by_id = {self._get_detection_id(det, i): det
                      for i, det in enumerate(original)}
        corr_by_id = {self._get_detection_id(det, i): det
                      for i, det in enumerate(corrected)}

        # Detect deletions (false positives)
        for orig_id, orig_det in orig_by_id.items():
            if orig_id not in corr_by_id:
                analyses.append({
                    "type": "false_positive",
                    "changes": {"deleted_detection": orig_det}
                })

        # Detect additions (false negatives)
        for corr_id, corr_det in corr_by_id.items():
            if corr_id not in orig_by_id:
                analyses.append({
                    "type": "false_negative",
                    "changes": {"added_detection": corr_det}
                })

        return analyses
```

Analysis Types:

- **false_positive**: User deleted a detection → Reduce sensitivity
- **false_negative**: User added a detection → Increase sensitivity
- **bbox_resize**: User resized bounding box → Adjust geometric rules
- **severity_change**: User changed severity → Adjust severity thresholds
- **category_change**: User changed category → Log for analysis

3. `parameter_tracker.py` - Change Tracking & Visualization

```python
class ParameterTracker:
    def log_parameter_change(self, image_id, user_id, params_before,
                            params_after, feedback_type, detection_counts):
        """Log parameter changes in JSON and CSV formats"""

        change_record = {
            "timestamp": datetime.now().isoformat(),
            "image_id": image_id,
            "user_id": user_id,
            "feedback_types": feedback_type,
            "detection_counts": detection_counts,
            "parameters_before": params_before,
            "parameters_after": params_after,
            "changes": self._calculate_changes(params_before, params_after)
        }

        # Save to JSON (full structure)
        self._save_json_log(change_record)

        # Save to CSV (flattened for analysis)
        self._save_csv_log(change_record)

        # Print formatted summary
        self._print_parameter_change(change_record)

    def create_visualization(self):
        """Generate parameter trend graphs"""
        # Load tracking data
        with open(self.json_file, 'r') as f:
            records = json.load(f)

        # Create trend plots
        df = pd.DataFrame(records)
        fig, axes = plt.subplots(len(param_columns), 1, figsize=(12, 8))

        for i, param in enumerate(param_columns):
            axes[i].plot(df['timestamp'], df[param],
                        marker='o', linewidth=2)
            axes[i].set_title(f'Parameter: {param}')

        plt.savefig("parameter_trends.png", dpi=300)
```

Output Files:

- `parameter_changes.json` : Full structured data
- `parameter_changes.csv` : Flattened for Excel/analysis
- `parameter_trends.png` : Visual trend graphs
- `reset_log.json` : Reset operation history

4. `model_core.py` - ML Integration

```python
def classify_anomalies_adaptive(filtered_img, anomaly_map=None):
    """Enhanced classification with adaptive parameters"""

    # Get current adaptive parameters
    hsv_params = adaptive_params.get_param("hsv_warm_thresholds")
    color_params = adaptive_params.get_param("color_classification")
    geom_params = adaptive_params.get_param("geometric_rules")

    # Adaptive threshold calculation
    k_adaptive = get_adaptive_k()
    if anomaly_map is not None:
        thresh = anomaly_map.mean() + k_adaptive * anomaly_map.std()
        bin_mask = anomaly_map > thresh

    # HSV classification with adaptive thresholds
    hsv = cv2.cvtColor(filtered_img, cv2.COLOR_BGR2HSV)
    h, w, _ = hsv.shape

    mask = np.zeros((h, w), dtype=np.uint8)
    for y in range(h):
        for x in range(w):
            H, S, V = hsv[y, x]
            hC, sC, vC = H/180.0, S/255.0, V/255.0

            # Use adaptive HSV thresholds
            warm_hue = (hC <= hsv_params["hue_low"]) or \
                       (hC >= hsv_params["hue_high"])
            warm_sat = sC >= hsv_params["saturation_min"]
            warm_val = vC >= hsv_params["value_min"]

            if warm_hue and warm_sat and warm_val:
                mask[y, x] = 1

def get_adaptive_k():
    """Convert percent threshold to k multiplier"""
    current_threshold = adaptive_params.get_current_percent_threshold()
    return percent_to_k(current_threshold)

def percent_to_k(percent):
    """Map threshold percentage to sensitivity multiplier"""
    percent = max(0, min(percent, 100))
    return 1.1 + (percent / 100.0) * (2.1 - 1.1)
```

5. `app.py` - FastAPI Server

```python
@app.post("/analyze")
async def analyze(file: UploadFile = File(...)):
    """Main analysis endpoint using adaptive parameters"""

    # Process uploaded image
    image = Image.open(io.BytesIO(await file.read()))
    image_np = np.array(image)

    # Run adaptive classification
    result = analyze_thermal_image(image_np, use_adaptive=True)

    # Include current parameters in response
    result["current_parameters"] = {
        "threshold": adaptive_params.get_current_percent_threshold(),
        "min_area_factor": adaptive_params.get_current_min_area_factor()
    }

    return result

@app.post("/adaptive-feedback")
async def adaptive_feedback(request: dict):
    """Receive feedback from Java backend"""

    try:
        # Extract data from Java backend format
        original_json = json.loads(request["originalAnalysisJson"])
        user_json = json.loads(request["userAnnotationsJson"])
        image_id = request["thermalImageId"]
        user_id = request["userId"]

        # Process through feedback handler
        result = feedback_handler.process_user_feedback(
            image_id, user_id, original_json, user_json
        )

        return {"status": "success", "adaptations": result}

    except Exception as e:
        return {"status": "error", "message": str(e)}

@app.post("/reset-parameters")
async def reset_parameters():
    """Reset all parameters to defaults"""

    try:
        current_params = adaptive_params.current_params.copy()
        adaptive_params.reset_to_defaults()

        return {
            "status": "success",
            "message": "Parameters reset to defaults",
            "previous_threshold": current_params.get("percent_threshold"),
            "new_threshold": 50
        }
    except Exception as e:
        return {"status": "error", "message": str(e)}
```

□ API Connections & JSON Formats

1. Java to Python Integration

Java Backend (UserAnnotationService.java):

```java
@Service
public class UserAnnotationService {

    public void saveAnnotations(Long thermalImageId, String userAnnotationsJson,
                               String userId) {
        // 1. Save to database
        UserAnnotation annotation = new UserAnnotation();
        annotation.setThermalImageId(thermalImageId);
        annotation.setAnnotationsJson(userAnnotationsJson);
        annotation.setUserId(userId);
        userAnnotationRepository.save(annotation);

        // 2. Get original analysis for comparison
        AnalysisResult originalAnalysis = analysisResultRepository
            .findByThermalImageId(thermalImageId);

        // 3. Send to Python adaptive system
        if (originalAnalysis != null) {
            sendAdaptiveFeedback(originalAnalysis, annotation);
        }
    }

    private void sendAdaptiveFeedback(AnalysisResult original,
                                     UserAnnotation userAnnotation) {
        try {
            String pythonUrl = "http://localhost:5000/adaptive-feedback";
            RestTemplate restTemplate = new RestTemplate();

            Map<String, Object> feedbackData = Map.of(
                "originalAnalysisJson", original.getDetectionsJson(),
                "thermalImageId", userAnnotation.getThermalImageId(),
                "userAnnotationsJson", userAnnotation.getAnnotationsJson(),
                "userId", userAnnotation.getUserId()
            );

            ResponseEntity<String> response = restTemplate.postForEntity(
                pythonUrl, feedbackData, String.class);

        } catch (Exception e) {
            // Log error but don't fail the save operation
            log.warn("Failed to send adaptive feedback: " + e.getMessage());
        }
    }
}
```

2. JSON Data Formats

Original Analysis JSON (from model):

```
[
    {
        "x": 150,
        "y": 200,
        "width": 80,
        "height": 60,
        "category": "loose_joint",
        "severity": "Potentially Faulty",
        "confidence": 0.85,
        "color_analysis": {
            "red_percentage": 0.3,
            "orange_percentage": 0.4,
            "yellow_percentage": 0.2
        },
        "geometry": {
            "area_fraction": 0.12,
            "aspect_ratio": 1.33
        }
    }
]
```

User Annotations JSON (corrected):

```
[
    {
        "x": 145,
        "y": 195,
        "width": 90,
        "height": 70,
        "category": "loose_joint",
        "severity": "Faulty",
        "confidence": 0.9,
        "user_modified": true,
        "modification_type": ["bbox_resize", "severity_change"]
    },
    {
        "x": 300,
        "y": 150,
        "width": 50,
        "height": 45,
        "category": "hot_spot",
        "severity": "Potentially Faulty",
        "confidence": 0.7,
        "user_added": true
    }
]
```

Feedback Data to Python:

```
{
    "originalAnalysisJson": "[{...}]",
    "thermalImageId": "11",
    "userAnnotationsJson": "[{...}]",
    "userId": "H1210"
}
```

3. Database Schema Integration

analysis_result table:

```
CREATE TABLE analysis_result (
    id BIGSERIAL PRIMARY KEY,
    thermal_image_id BIGINT,
    detections_json TEXT,  -- Original ML results
    analysis_timestamp TIMESTAMP,
    status VARCHAR(50)
);
```

user_annotations table:

```
CREATE TABLE user_annotations (
    id BIGSERIAL PRIMARY KEY,
    thermal_image_id BIGINT,
    annotations_json TEXT,  -- User corrections
    user_id VARCHAR(255),
    created_at TIMESTAMP
);
```

⚙ Parameter Adaptation Logic

1. Adaptation Decision Tree

```
User Feedback Input
        │
        ▼
┌─────────────────┐
│ Feedback Analysis│
└─────────────────┘
        │
        ▼
┌─────────────────┐     ┌─────────────────┐     ┌─────────────────┐
│ False Negative  │────▶│ Increase Sensitivity │────▶│ threshold -= 3  │
│ (User Added)    │     │ Lower Threshold    │     │ min_area *= 0.8 │
└─────────────────┘     └─────────────────┘     └─────────────────┘
        │
        ▼
┌─────────────────┐     ┌─────────────────┐     ┌─────────────────┐
│ False Positive  │────▶│ Decrease Sensitivity│────▶│ threshold += 3  │
│ (User Deleted)  │     │ Higher Threshold   │     │ min_area *= 1.2 │
└─────────────────┘     └─────────────────┘     └─────────────────┘
        │
        ▼
┌─────────────────┐     ┌─────────────────┐     ┌─────────────────┐
│ Bbox Resize     │────▶│ Geometric Adapt   │────▶│ Adjust area     │
│ (Size Change)   │     │ Shape Rules       │     │ requirements    │
└─────────────────┘     └─────────────────┘     └─────────────────┘
        │
        ▼
┌─────────────────┐     ┌─────────────────┐     ┌─────────────────┐
│ Severity Change │────▶│ Classification    │────▶│ Adjust color    │
│ (Level Change)  │     │ Threshold Adapt   │     │ thresholds      │
└─────────────────┘     └─────────────────┘     └─────────────────┘
```

2. Parameter Bounds & Constraints

```
# Sensitivity constraints
MIN_THRESHOLD = 10    # Most sensitive
MAX_THRESHOLD = 90    # Least sensitive
DEFAULT_THRESHOLD = 50

# Area factor constraints
MIN_AREA_FACTOR = 0.0005   # Smallest allowed blobs
MAX_AREA_FACTOR = 0.005    # Largest minimum size
DEFAULT_AREA_FACTOR = 0.001

# Geometric rule bounds
MIN_LOOSE_JOINT_AREA = 0.05    # 5% minimum area
MAX_LOOSE_JOINT_AREA = 0.30    # 30% maximum area

# HSV bounds (normalized [0,1])
HSV_HUE_LOW = [0.10, 0.25]     # Range for warm detection
HSV_HUE_HIGH = [0.90, 1.00]    # Range for warm detection
HSV_SAT_MIN = [0.20, 0.50]     # Saturation minimum
HSV_VAL_MIN = [0.30, 0.70]     # Value minimum
```

3. Adaptation Rate Control

```
def _increase_sensitivity(self, changes):
    """Controlled sensitivity increase"""
    current = self.current_params["percent_threshold"]

    # Progressive adaptation - smaller steps as we get more sensitive
    if current > 40:
        step = 3      # Normal step
    elif current > 20:
        step = 2      # Smaller step
    else:
        step = 1      # Minimal step

    new_threshold = max(MIN_THRESHOLD, current - step)
    self.current_params["percent_threshold"] = new_threshold

    # Log adaptation reasoning
    print(f"Sensitivity increased: {current} → {new_threshold} (step: {step})")
```

󰄀 Data Tracking & Visualization

1. CSV Structure Explanation

The `parameter_changes.csv` contains flattened tracking data:

```
timestamp,image_id,user_id,feedback_types,original_detections,corrected_detections,
added_detections,before_percent_threshold,before_min_area_factor,
after_percent_threshold,after_min_area_factor,delta_percent_threshold,
delta_min_area_factor
```

Key Columns:

- `timestamp` : ISO format change time
- `image_id` : Thermal image identifier
- `user_id` : User who provided feedback
- `feedback_types` : Comma-separated list of feedback types
- `original_detections` : Count of ML detections
- `corrected_detections` : Count after user corrections
- `added_detections` : Count of user-added detections
- `before_* / after_*` : Parameter values before/after adaptation
- `delta_*` : Calculated parameter changes

2. JSON Structure (Full Data)

```json
[
    {
        "timestamp": "2025-10-21T16:16:49.854205",
        "image_id": "12",
        "user_id": "H1210",
        "feedback_types": ["false_negative"],
        "detection_counts": {
            "original": 3,
            "corrected": 4,
            "added": 1,
            "deleted": 0
        },
        "parameters_before": {
            "percent_threshold": 47,
            "min_area_factor": 0.0008,
            "hsv_warm_thresholds": {...},
            "color_classification": {...}
        },
        "parameters_after": {
            "percent_threshold": 44,
            "min_area_factor": 0.00064,
            ...
        },
        "changes": {
            "percent_threshold": {
                "from": 47,
                "to": 44,
                "delta": -3
            },
            "min_area_factor": {
                "from": 0.0008,
                "to": 0.00064,
                "delta": -0.00016
            }
        }
    }
]
```

3. Visualization Generation

```
def create_parameter_visualization():
    """Generate trend analysis charts"""

    # Load tracking data
    df = pd.read_csv("parameter_tracking/parameter_changes.csv")
    df['timestamp'] = pd.to_datetime(df['timestamp'])

    # Create multi-parameter trend plot
    fig, axes = plt.subplots(3, 1, figsize=(14, 10))

    # 1. Threshold trend
    axes[0].plot(df['timestamp'], df['after_percent_threshold'],
                marker='o', linewidth=2, color='red')
    axes[0].set_title('Detection Threshold Evolution')
    axes[0].set_ylabel('Threshold Value')
    axes[0].grid(True, alpha=0.3)

    # 2. Min area trend
    axes[1].plot(df['timestamp'], df['after_min_area_factor'],
                marker='s', linewidth=2, color='blue')
    axes[1].set_title('Minimum Area Factor Evolution')
    axes[1].set_ylabel('Area Factor')
    axes[1].grid(True, alpha=0.3)

    # 3. Feedback type distribution
    feedback_counts = df['feedback_types'].value_counts()
    axes[2].bar(feedback_counts.index, feedback_counts.values, color='green')
    axes[2].set_title('Feedback Type Distribution')
    axes[2].set_ylabel('Count')

    plt.tight_layout()
    plt.savefig("parameter_tracking/comprehensive_trends.png", dpi=300)
```

🔄 Reset & Management Procedures

1. Command Line Reset Options

Option A: Direct Python Command

```
 cd "C:\Users\Yasiru Alahakoon\Desktop\FlareNet\python-backend"

# Quick reset
python -c "from adaptive_params import adaptive_params; adaptive_params.reset_to_defaults()"

# Reset with confirmation
python -c "
from adaptive_params import adaptive_params
import json
print('Current params:', json.dumps(adaptive_params.current_params, indent=2))
adaptive_params.reset_to_defaults()
print('Reset completed!')
"
```

Option B: Parameter Manager Script

```
# Show current status
python param_manager.py --show

# Reset parameters
python param_manager.py --reset

# Show tracking statistics
python param_manager.py --stats

# Create visualization
python param_manager.py --visualize
```

Option C: HTTP API Reset

```
# Reset via FastAPI endpoint
curl -X POST http://localhost:5000/reset-parameters

# Check current parameters
curl -X GET http://localhost:5000/current-parameters
```

2. Java Backend Integration

```java
@RestController
@RequestMapping("/api/adaptive")
public class AdaptiveParameterController {

    @PostMapping("/reset")
    public ResponseEntity<String> resetParameters() {
        try {
            String pythonUrl = "http://localhost:5000/reset-parameters";
            RestTemplate restTemplate = new RestTemplate();

            ResponseEntity<Map> response = restTemplate.postForEntity(
                pythonUrl, null, Map.class);

            if (response.getBody().get("status").equals("success")) {
                return ResponseEntity.ok("Parameters reset successfully");
            } else {
                return ResponseEntity.status(500)
                    .body("Reset failed: " + response.getBody().get("message"));
            }
        } catch (Exception e) {
            return ResponseEntity.status(500)
                .body("Connection error: " + e.getMessage());
        }
    }

    @GetMapping("/status")
    public ResponseEntity<Map> getParameterStatus() {
        try {
            String pythonUrl = "http://localhost:5000/current-parameters";
            RestTemplate restTemplate = new RestTemplate();

            ResponseEntity<Map> response = restTemplate.getForEntity(
                pythonUrl, Map.class);

            return ResponseEntity.ok(response.getBody());
        } catch (Exception e) {
            return ResponseEntity.status(500).body(
                Map.of("error", "Failed to get parameter status")
            );
        }
    }
}
```

3. Frontend Integration

React Component for Parameter Management:

```
const AdaptiveParameterManager = () => {
    const [parameters, setParameters] = useState(null);
    const [resetLoading, setResetLoading] = useState(false);

    const fetchCurrentParameters = async () => {
        try {
            const response = await fetch('/api/adaptive/status');
            const data = await response.json();
            setParameters(data);
        } catch (error) {
            console.error('Failed to fetch parameters:', error);
        }
    };

    const resetParameters = async () => {
        setResetLoading(true);
        try {
            const response = await fetch('/api/adaptive/reset', {
                method: 'POST'
            });

            if (response.ok) {
                alert('Parameters reset successfully!');
                fetchCurrentParameters(); // Refresh display
            } else {
                alert('Reset failed');
            }
        } catch (error) {
            alert('Reset error: ' + error.message);
        } finally {
            setResetLoading(false);
        }
    };

    return (
        <div className="parameter-manager">
            <h3>Adaptive Parameter Status</h3>
            {parameters && (
                <div className="parameter-display">
                    <p>Threshold: {parameters.percent_threshold}</p>
                    <p>Min Area: {parameters.min_area_factor}</p>
                </div>
            )}
            <button
                onClick={resetParameters}
                disabled={resetLoading}
                className="reset-button"
            >
                {resetLoading ? 'Resetting...' : 'Reset to Defaults'}
            </button>
        </div>
    );
};
```

Code Examples & Usage

1. Complete Workflow Example

```
 # 1. Initialize system
from adaptive_params import adaptive_params
from feedback_handler import feedback_handler
from parameter_tracker import parameter_tracker

# 2. Check current state
print("Current threshold:", adaptive_params.get_current_percent_threshold())
print("Current min area:", adaptive_params.get_current_min_area_factor())

# 3. Simulate user feedback
original_detections = [
    {"x": 100, "y": 100, "width": 50, "height": 50,
     "category": "loose_joint", "confidence": 0.8}
]

user_corrections = [
    {"x": 100, "y": 100, "width": 50, "height": 50,
     "category": "loose_joint", "confidence": 0.8},
    {"x": 200, "y": 200, "width": 60, "height": 60,
     "category": "hot_spot", "confidence": 0.9}  # User added this
]

# 4. Process feedback
result = feedback_handler.process_user_feedback(
    image_id="test_123",
    user_id="developer",
    original_detections=original_detections,
    user_corrections=user_corrections
)

print("Adaptations applied:", result['adaptations_applied'])
print("New threshold:", adaptive_params.get_current_percent_threshold())

# 5. Generate visualization
parameter_tracker.create_visualization()

# 6. Reset if needed
# adaptive_params.reset_to_defaults()
```

2. Testing API Endpoints

```
import requests
import json

# Test analysis with current parameters
def test_analysis():
    with open("test_image.jpg", "rb") as f:
        files = {"file": f}
        response = requests.post("http://localhost:5000/analyze", files=files)

    result = response.json()
    print("Current threshold used:", result.get("current_parameters", {}).get("threshold"))
    return result

# Test feedback processing
def test_feedback():
    feedback_data = {
        "originalAnalysisJson": json.dumps([
            {"x": 100, "y": 100, "width": 50, "height": 50,
             "category": "loose_joint"}
        ]),
        "thermalImageId": "999",
        "userAnnotationsJson": json.dumps([
            {"x": 100, "y": 100, "width": 50, "height": 50,
             "category": "loose_joint"},
            {"x": 200, "y": 200, "width": 60, "height": 60,
             "category": "hot_spot"}
        ]),
        "userId": "test_user"
    }

    response = requests.post("http://localhost:5000/adaptive-feedback",
                           json=feedback_data)
    print("Feedback result:", response.json())

# Test parameter reset
def test_reset():
    response = requests.post("http://localhost:5000/reset-parameters")
    print("Reset result:", response.json())

# Run tests
if __name__ == "__main__":
    print("=== Testing Analysis ===")
    test_analysis()

    print("=== Testing Feedback ===")
    test_feedback()

    print("=== Testing Reset ===")
    test_reset()
```

3. Parameter Monitoring Script

```python
import time
import json
from datetime import datetime
from adaptive_params import adaptive_params

def monitor_parameters(interval=30, duration=3600):
    """Monitor parameter changes over time"""

    start_time = time.time()
    monitoring_log = []

    print(f"Starting parameter monitoring for {duration/60:.1f} minutes...")

    while time.time() - start_time < duration:
        current_params = adaptive_params.current_params.copy()

        log_entry = {
            "timestamp": datetime.now().isoformat(),
            "threshold": current_params["percent_threshold"],
            "min_area_factor": current_params["min_area_factor"]
        }

        monitoring_log.append(log_entry)

        print(f"[{log_entry['timestamp']}] Threshold: {log_entry['threshold']}, "
              f"Min Area: {log_entry['min_area_factor']:.6f}")

        time.sleep(interval)

    # Save monitoring log
    with open(f"monitoring_log_{int(time.time())}.json", "w") as f:
        json.dump(monitoring_log, f, indent=2)

    print(f"Monitoring completed. Log saved with {len(monitoring_log)} entries.")

if __name__ == "__main__":
    monitor_parameters(interval=10, duration=600)  # 10 minutes monitoring
```

🔧 Troubleshooting Guide

Common Issues & Solutions

1. Parameters Not Changing

Problem: Adaptive system seems to run but parameters don't change **Diagnosis:**

```bash
# Check current parameters
python -c "from adaptive_params import adaptive_params; print(adaptive_params.current_params)"

# Check if feedback is reaching Python
curl -X POST http://localhost:5000/adaptive-feedback \
  -H "Content-Type: application/json" \
  -d '{"test": "data"}'
```

Solutions:

- Ensure FastAPI server is running on correct port
- Check Java backend is sending requests to right URL
- Verify JSON format matches expected structure
- Check parameter bounds aren't preventing changes

2. Tracking Files Not Generated

Problem: CSV/JSON tracking files missing **Diagnosis:**

```
# Check if tracking directory exists
ls -la parameter_tracking/

# Check if parameter_tracker is imported
python -c "from parameter_tracker import parameter_tracker; print('OK')"
```

Solutions:

- Create tracking directory manually: `mkdir parameter_tracking`
- Check file permissions
- Ensure matplotlib is installed for visualization
- Verify feedback_handler calls parameter_tracker

3. Java-Python Connection Issues

Problem: Java backend can't reach Python adaptive system **Diagnosis:**

```
// Test connection in Java
RestTemplate restTemplate = new RestTemplate();
try {
    String response = restTemplate.getForObject(
        "http://localhost:5000/health", String.class);
    System.out.println("Python connection: " + response);
} catch (Exception e) {
    System.out.println("Connection failed: " + e.getMessage());
}
```

Solutions:

- Check Python FastAPI server is running: `python app.py`
- Verify port configuration (default: 5000)
- Check firewall settings
- Ensure both services use same host/port configuration

4. Parameter Reset Not Working

Problem: Reset command doesn't restore defaults **Diagnosis:**

```
from adaptive_params import adaptive_params
print("Current:", adaptive_params.current_params["percent_threshold"])
adaptive_params.reset_to_defaults()
print("After reset:", adaptive_params.current_params["percent_threshold"])
```

Solutions:

- Check if adaptive_parameters.json file is write-protected
- Verify default_params dictionary is properly defined
- Ensure save_params() method is working
- Check file system permissions

5. Visualization Errors

Problem: Parameter trend graphs not generating **Diagnosis:**

```
import matplotlib.pyplot as plt
import pandas as pd
print("Matplotlib version:", plt.__version__)
print("Pandas version:", pd.__version__)
```

Solutions:

- Install missing dependencies: `pip install matplotlib pandas`
- Check if parameter_changes.json exists and has data
- Ensure proper datetime formatting
- Verify image save permissions

Performance Optimization

1. Reduce Parameter Tracking Overhead

```
 # Batch tracking updates
class OptimizedParameterTracker(ParameterTracker):
    def __init__(self):
        super().__init__()
        self.pending_changes = []
        self.batch_size = 10

    def log_parameter_change(self, *args):
        self.pending_changes.append(args)
        if len(self.pending_changes) >= self.batch_size:
            self.flush_changes()

    def flush_changes(self):
        for change_args in self.pending_changes:
            super().log_parameter_change(*change_args)
        self.pending_changes.clear()
```

2. Optimize Feedback Analysis

```
 # Cache detection ID calculations
def _get_detection_id_cached(self, detection, index):
    if not hasattr(self, '_id_cache'):
        self._id_cache = {}

    key = f"{detection.get('x', 0)}_{detection.get('y', 0)}_{index}"
    if key not in self._id_cache:
        self._id_cache[key] = self._calculate_detection_id(detection, index)

    return self._id_cache[key]
```

□ Summary

The FlareNet Adaptive Learning System provides a comprehensive solution for improving thermal anomaly detection through human feedback without model retraining. Key capabilities:

□ **Mathematical Foundation**: Robust threshold adaptation using statistical methods □ **Multi-Language Integration**: Python ML ↔ Java Backend ↔ React Frontend □ **Comprehensive Tracking**: JSON structure + CSV analytics + Visualizations □ **Production Ready**: Error handling, logging, reset capabilities □ **Extensible Design**: Easy to add new parameter types and adaptation strategies

The system successfully bridges the gap between automated ML detection and human expertise, creating a learning loop that improves detection accuracy over time while maintaining the integrity of the core trained model.

For technical support or questions about implementation details, refer to individual Python script documentation or contact the development team.

Documentation generated for FlareNet Milestone 03 - Adaptive Learning System Last Updated: October 21, 2025