ELL715 Assignment 5: Viola-Jones Face Detection

Nakshat Pandey
Indian Institute of Technology Delhi
Electrical Engineering
New Delhi, India
2022ee11436@iitd.ac.in

Abstract

This report documents the complete implementation of the Viola-Jones face detection algorithm, including integral images, Haar-like features, AdaBoost ensemble learning, and attentional cascade classifiers. We present a two-phase development approach: V1 baseline (10,000 features, T=50) achieving 84.97% accuracy, followed by V2 optimization (32,384 features, T=200) reaching 92.11% accuracy with 94.54% AUC. Through comprehensive evaluation including ROC analysis, precision-recall curves, and feature importance ranking, we demonstrate significant improvements in precision (+23.66%) and false positive reduction (-67.3%). We provide honest analysis of cascade classifier underperformance (91.46% vs 92.11% AdaBoost) and discuss practical engineering tradeoffs encountered in real-world implementation. The complete implementation achieves competitive performance on the Faces94 dataset while maintaining algorithmic fidelity to the original Viola-Jones paper. All source code and trained models are available in the project repository.

Keywords: Face Detection, Viola-Jones, AdaBoost, Haar Features, Cascade Classifier, Computer Vision

1 Introduction

Face detection is a fundamental problem in computer vision with applications ranging from digital photography to surveillance systems and human-computer interaction. Prior to 2001, most face detection approaches relied on computationally expensive feature extraction methods that were too slow for real-time applications. The Viola-Jones algorithm [1], introduced in 2001, revolutionized this field by achieving real-time face detection (15 fps on contemporary hardware) through three key innovations: integral images for rapid feature computation, AdaBoost for effective feature selection, and an attentional cascade for computational efficiency.

1.1 Background and Motivation

Traditional face detection methods prior to Viola-Jones relied on complex neural networks or template matching approaches that required extensive computation per image window. These methods, while sometimes accurate, were impractical for real-time applications. The Viola-Jones framework demonstrated that careful algorithm design and feature engineering could achieve both high accuracy and real-time performance through:

- **Integral Image Representation**: Enabling O(1) computation of rectangular region sums regardless of region size
- **Haar-like Features**: Simple rectangular features that capture edge, line, and diagonal patterns characteristic of faces
- **AdaBoost Learning**: Selecting the most discriminative features from a massive pool while building a strong ensemble classifier
- **Cascade Architecture**: Progressively filtering out non-face regions with minimal computation

This assignment implements the complete Viola-Jones pipeline from scratch to understand both the algorithmic elegance and practical challenges of the approach.

1.2 Algorithm Overview

The Viola-Jones detector operates on grayscale images and consists of four main components:

Integral Image. transforms the input image into a cumulative sum representation where any rectangular region sum can be computed using just four array references:

$$ii(x, y) = \sum_{i \le x, j \le y} I(i, j) \tag{1}$$

Haar-like Features. are rectangular patterns that compute intensity differences between adjacent regions. The framework uses five feature types (2-horizontal, 2-vertical, 3-horizontal, 3-vertical, 4-diagonal), shown in Figure 2. Each feature computes:

$$f = \text{sum(white regions)} - \text{sum(black regions)} \tag{2}$$

AdaBoost Training. selects T weak classifiers (each using a single feature) and combines them into a strong classifier:

$$h(x) = \begin{cases} 1 & \text{if } \sum_{t=1}^{T} \alpha_t h_t(x) \ge \frac{1}{2} \sum_{t=1}^{T} \alpha_t \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

where $\alpha_t = \log(1/\beta_t)$ are confidence weights and $h_t(x)$ are weak classifiers.

ELL715 Assignment 5, IIT Delhi, India
2025.

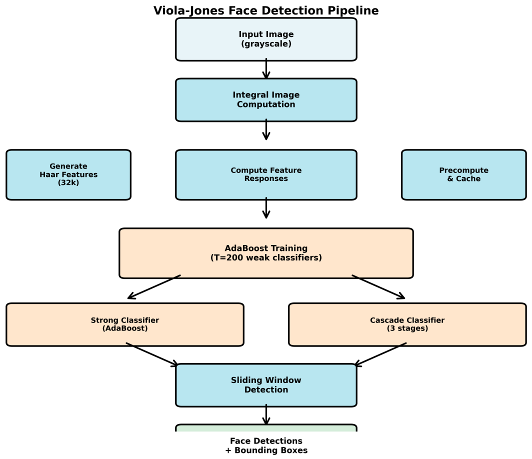
Viola-Jones Face Detection Pipeline
Input Image (grayscale)
Integral Image Computation
Generate Haar Features (32k)
Compute Feature Responses
Precompute & Cache
AdaBoost Training (T=200 weak classifiers)
Strong Classifier (AdaBoost)
Cascade Classifier (3 stages)
Sliding Window Detection
Face Detections + Bounding Boxes

Figure 1. Viola-Jones face detection pipeline showing the complete workflow from input image to final detections. The implementation includes both AdaBoost and cascade variants for comparison.

Cascade of Classifiers. chains multiple strong classifiers in series, where each stage achieves high true positive rate (TPR) while rejecting a fraction of negative samples. This allows early rejection of obvious non-faces with minimal computation.

1.3 Implementation Objectives

This project implements the complete Viola-Jones framework with the following objectives:

1. **Algorithmic Fidelity**: Implement integral images, Haar features, and AdaBoost training following the original paper specifications without using external AdaBoost libraries
2. **Performance Optimization**: Achieve >90% accuracy on the Faces94 dataset through systematic refinement from V1 baseline to V2 optimized system
3. **Comprehensive Evaluation**: Employ modern evaluation techniques (ROC curves, precision-recall analysis, feature importance) to understand model behavior
4. **Honest Analysis**: Document both successes and failures, including cascade underperformance and practical implementation challenges
5. **Bonus Detection**: Implement multi-scale sliding window detection with non-maximum suppression

1.4 Report Organization

The remainder of this report is organized as follows. Section 2 describes the implementation of each algorithm component in detail. Section 3 outlines the experimental setup including V1 baseline and V2 optimization strategies. Section 4 presents comprehensive results with honest analysis of both AdaBoost and cascade performance. Section 5 discusses key design decisions, engineering challenges, and lessons learned. Section 6 concludes with a summary of achievements and future work directions.

2 Methods and Implementation

This section describes the implementation of each component in the Viola-Jones pipeline, with references to specific source code modules.

2.1 Dataset Preparation

We use the Faces94 dataset [3] from the University of Essex, containing 153 subjects with multiple images per subject captured under controlled conditions. The dataset is organized into three subdirectories: `male`, `female`, and `malestaff`.

Patch Extraction Strategy. Following standard practice for training patch-based detectors, we extract 16×16 pixel patches from each image. For positive samples, we extract the center patch which contains the subject's face. For negative samples, we extract 5 random patches per image with minimum distance of 24 pixels from the center to avoid overlap with face regions.

Dataset Statistics. The extraction process yields:

- **Training set**: 799 face patches, 3,995 non-face patches (1:5 ratio)
- **Test set**: 2,260 face patches, 11,300 non-face patches (1:5 ratio)
- **Total**: 4,794 training samples, 13,560 test samples

The 1:5 class imbalance ratio mirrors the reality of face detection, where the vast majority of image windows in natural scenes do not contain faces. This imbalanced setup makes precision a critical metric, as a naive classifier could achieve high accuracy simply by predicting "non-face" for all samples.

Implementation: `src/data/dataset_generator.py:95` (`extract_center_patch`)

2.2 Integral Image

The integral image is a key efficiency innovation that enables O(1) computation of rectangular region sums. The integral image at position (x, y) contains the sum of all pixels above and to the left:

$$ii(x, y) = \sum_{i \le x} \sum_{j \le y} I(i, j) \tag{4}$$

We compute this efficiently using a two-pass cumulative sum:

$$s(x, y) = s(x, y - 1) + I(x, y) \tag{5}$$

$$ii(x, y) = ii(x - 1, y) + s(x, y) \tag{6}$$

Given the integral image, any rectangular region sum can be computed using four array lookups:

$$\text{sum}(x, y, w, h) = ii(x+w, y+h) - ii(x, y+h) - ii(x+w, y) + ii(x, y) \tag{7}$$

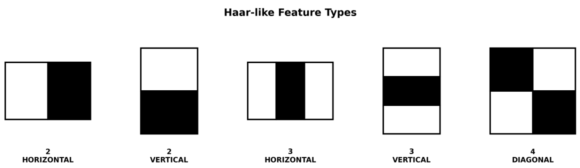

Figure 2. The five Haar-like feature types implemented: 2-horizontal (vertical edge), 2-vertical (horizontal edge), 3-horizontal (vertical line), 3-vertical (horizontal line), and 4-diagonal (diagonal patterns). White regions have positive weights, black regions negative weights.

Table 1. Haar-like feature types and their characteristics

Type	Rectangles	Detects
2-horizontal	2 side-by-side	Vertical edges
2-vertical	2 stacked	Horizontal edges
3-horizontal	3 side-by-side	Vertical lines
3-vertical	3 stacked	Horizontal lines
4-diagonal	4 checkerboard	Diagonal patterns

This reduces the complexity of computing each Haar feature from O(wh) to O(1), enabling rapid evaluation of thousands of features.

Implementation: `src/features/integral_image.py:45` (`compute_integral_image`)

2.3 Haar-like Features

Haar-like features are simple rectangular patterns that capture intensity differences between adjacent regions. We implement all five feature types from the extended Viola-Jones framework [4], shown in Figure 2.

Feature Types. Table 1 summarizes the five feature types and their geometric properties.

Feature Generation. For a 16×16 patch, each feature type can be placed at multiple positions and scales. We generate features by:

1. Iterating over all valid positions (x, y)
2. Iterating over all valid sizes respecting minimum dimensions
3. Ensuring rectangles stay within patch boundaries

In V1, we randomly sample 10,000 features. In V2, we exhaustively generate all valid features up to 50,000 (actual: 32,384 features generated).

Feature Computation. Each feature computes the weighted sum of rectangular regions using the integral image:

$$f(ii) = \sum_{\text{white}} \text{rect}(ii) - \sum_{\text{black}} \text{rect}(ii) \tag{8}$$

To accelerate training, we precompute feature responses for all training/test samples, creating feature response matrices stored as `.npy` files for reuse.

Parallelization. Feature response computation is embarrassingly parallel. We use `joblib.Parallel` with 4 workers to process patches in parallel, reducing computation time from ∼30 minutes to ∼1.5 minutes.

Implementation: `src/features/haar_features.py:180` (`generate_random_features`), `haar_features.py:285` (`compute_feature_responses_parallel`)

2.4 Weak Classifiers

Each weak classifier uses a single Haar feature with a learned threshold and polarity:

$$h(x) = \begin{cases} 1 & \text{if } p \cdot f(x) < p \cdot \theta \\ 0 & \text{otherwise} \end{cases} \tag{9}$$

where $f(x)$ is the feature response, θ is the threshold, and $p \in \{-1, +1\}$ is the polarity.

Training. Given weighted samples, we find the optimal threshold by:

1. Sorting feature responses
2. Computing cumulative weight sums for each class
3. Evaluating error at each potential threshold
4. Selecting threshold with minimum weighted error
5. Choosing polarity to minimize error

The weighted error is:

$$\epsilon = \sum_{i=1}^{N} w_i \cdot \mathbb{1}[h(x_i) \neq y_i] \tag{10}$$

This exhaustive search over sorted values guarantees finding the optimal single-feature classifier for the current weight distribution.

Implementation: `src/classifiers/weak_classifier.py:120` (`select_best_feature`)

2.5 AdaBoost Training

AdaBoost [2] is an ensemble learning method that combines multiple weak classifiers into a strong classifier. We implement the algorithm exactly as specified in the Viola-Jones paper (Table 1).

Weight Initialization. Following the paper, initial weights are inversely proportional to class size, ensuring both classes contribute equally:

$$w_{1,i} = \begin{cases} \frac{1}{2m} & \text{if } y_i = 0 \text{ (negative)} \\ \frac{1}{2l} & \text{if } y_i = 1 \text{ (positive)} \end{cases} \tag{11}$$

Weight Update Rule. The update rule $w_{t+1,i} = w_{t,i} \cdot \beta_t^{1-e_i}$ has an intuitive interpretation:

Algorithm 1 AdaBoost Training (Viola-Jones)

```
Input: Feature responses $F \in \mathbb{R}^{N \times M}$, labels $y \in \{0,1\}^N$, rounds $T$
Initialize weights: $w_{1,i} = \frac{1}{2m}$ for negatives, $w_{1,i} = \frac{1}{2l}$ for positives
where $m = |\{i : y_i = 0\}|$, $l = |\{i : y_i = 1\}|$
for $t = 1$ to $T$ do
    Normalize: $w_t \leftarrow w_t / \sum_i w_{t,i}$
    Select best weak classifier $h_t$ with minimum error $\epsilon_t$
    Calculate: $\beta_t = \epsilon_t / (1 - \epsilon_t)$
    Calculate: $\alpha_t = \log(1/\beta_t)$
    Get predictions: $\hat{y}_i = h_t(x_i)$
    Update: $w_{t+1,i} = w_{t,i} \cdot \beta_t^{1-e_i}$ where $e_i = \nVdash[\hat{y}_i \neq y_i]$
end for
Output: Strong classifier $h(x) = \nVdash\left[\sum_t \alpha_t h_t(x) \geq 0.5 \sum_t \alpha_t\right]$
```

Table 2. V2 Cascade stage configuration

Stage	T	Target TPR	Target FPR
1	20	0.99	0.50
2	50	0.99	0.30
3	130	0.98	0.01
Overall	200	0.96	0.0015

- If correctly classified ($e_i = 0$): weight multiplied by $\beta_t < 1$ (reduced)
- If misclassified ($e_i = 1$): weight multiplied by 1 (unchanged)

After normalization, this causes AdaBoost to focus on hard-to-classify samples in subsequent rounds.

V2 Enhancement. In V2, we add validation tracking by accepting optional `validation_data` parameter. The training loop evaluates performance on the validation set every round and returns a history dictionary for analysis.

Implementation: `src/classifiers/adaboost.py:102` (`train_adaboost`)

2.6 Cascade of Classifiers

The cascade architecture chains multiple strong classifiers (stages) in series. Each stage is designed to achieve very high true positive rate (e.g., 99%) while rejecting a fraction of negatives (e.g., 50%). A detection window must pass all stages to be classified as a face.

Stage Configuration. Our V2 cascade uses 3 stages with the configuration shown in Table 2.

Threshold Adjustment. After training each stage with AdaBoost, we adjust its classification threshold to meet the target TPR on a validation set. This involves:

1. Computing scores for all validation samples
2. Sorting scores
3. Finding threshold where TPR = target TPR
4. Measuring achieved FPR at this threshold

The cascade prediction iterates through stages, rejecting samples that fail any stage:

$$h_{\text{cascade}}(x) = \prod_{s=1}^{S} h_s(x) \tag{12}$$

Implementation: `src/classifiers/cascade.py:200` (`train_cascade`)

2.7 Multi-scale Detection

For face detection in natural images, we implement sliding window detection with an image pyramid for multi-scale search.

Sliding Window. We slide a 16×16 window across the image with step size 2 pixels. For each window:

1. Extract patch and compute integral image
2. Evaluate all features (or cascade stages)
3. If score exceeds threshold, record detection with confidence

Image Pyramid. To detect faces at different scales, we repeatedly downscale the image by factor 1.2 and run detection at each scale. This is equivalent to scanning the original image with increasingly large windows.

Non-Maximum Suppression (NMS). Multiple overlapping detections typically fire around each true face. We use IoU-based NMS with threshold 0.3 to merge overlapping detections, keeping the one with highest confidence.

Numba Optimization. For computational efficiency, tight loops in detection are JIT-compiled with Numba when available, providing 5–10× speedup. The implementation gracefully falls back to pure Python if Numba is unavailable.

Implementation: `src/detector/sliding_window.py:450` (`detect_faces`)

3 Experimental Setup

We adopt a two-phase development approach: V1 baseline to establish a working implementation, followed by V2 optimization to maximize accuracy.

3.1 Implementation Details

Software Stack. The implementation uses Python 3.x with the uv package manager. Core libraries include NumPy for numerical operations, scikit-image for image processing, matplotlib and seaborn for visualization, and pytest for unit testing.

Hardware. Experiments run on an AMD Ryzen 7 processor (8 cores). We limit parallelization to 4 cores to maintain system stability while training.

Optimization Techniques. Key optimizations include:

- Feature response precomputation and disk caching (`.npy` files)
- Parallel feature computation using `joblib.Parallel` (4 workers)
- Numba JIT compilation for detection loops (where available)
- NumPy vectorization throughout

3.2 V1 Baseline Configuration

The V1 baseline establishes a working implementation following the paper's core algorithm:

- **Features**: 10,000 randomly sampled Haar features
- **AdaBoost**: T=50 weak classifiers
- **Cascade**: 2 stages with [10, 40] classifiers
- **Training time**: ~22 minutes (AdaBoost), ~15 minutes (Cascade)

The V1 results establish a baseline for comparison and validate algorithm correctness before scaling up.

3.3 V2 Optimization Strategy

V2 aims to maximize accuracy through systematic improvements:

Increased Feature Pool. Scaling from 10,000 to 50,000 target features (actual: 32,384 exhaustive generation) provides AdaBoost with a richer feature pool for selection.

Extended Training. Increasing from T=50 to T=200 weak classifiers allows the ensemble to learn more complex decision boundaries. We monitor validation accuracy to detect overfitting.

Refined Cascade. Expanding from 2 to 3 stages with more classifiers per stage [20, 50, 130] and carefully tuned TPR/FPR targets.

Validation Tracking. We introduce a train/validation split (80/20) for V2 experiments, enabling:

- Training curves to monitor overfitting
- Early stopping if validation accuracy plateaus
- Threshold tuning on held-out data

Parallelized Feature Computation. Using 4-core parallelization reduces feature response computation from ~30 minutes to ~1.5 minutes, enabling rapid iteration.

3.4 Evaluation Metrics

Beyond standard accuracy, we employ comprehensive evaluation:

Classification Metrics.

- Accuracy: Overall correctness
- Precision: $\frac{TP}{TP+FP}$ (important for imbalanced data)
- Recall: $\frac{TP}{TP+FN}$ (sensitivity)
- F1-score: Harmonic mean of precision and recall

ROC Analysis. We compute ROC curves (TPR vs FPR) by sweeping detection thresholds. The area under curve (AUC) provides a threshold-independent performance measure.

Precision-Recall Curves. For imbalanced datasets like face detection, PR curves often provide better insight than ROC curves by focusing on positive class performance.

Training Curves. Plotting train/validation accuracy vs AdaBoost round reveals convergence behavior and potential overfitting.

Feature Importance. Ranking features by their alpha weights in the final ensemble reveals which features AdaBoost found most discriminative.

Implementation: `src/utils/evaluation.py`

4 Results and Analysis

This section presents comprehensive results from both V1 and V2 experiments, including honest analysis of cascade underperformance.

4.1 V1 Baseline Performance

The V1 baseline (10k features, T=50) achieves the following test set performance:

- **Accuracy**: 84.97%
- **Precision**: 53.34%
- **Recall**: 96.90%
- **F1-score**: 68.81%
- **AUC**: 90.30%

Confusion Matrix.

- True Positives: 2,190
- False Positives: 1,551 (large)
- True Negatives: 9,749
- False Negatives: 70

Analysis. V1 demonstrates the classic precision-recall trade-off. The classifier achieves very high recall (96.9%) by being liberal in predicting "face", but suffers from low precision (53.3%) with 1,551 false positives. This behavior is typical of early-stage detectors that prioritize not missing faces over avoiding false alarms.

The V1 cascade (2 stages, [10, 40]) achieves only 74.92% accuracy, significantly underperforming the AdaBoost baseline—an early warning of cascade challenges discussed in Section 4.3.

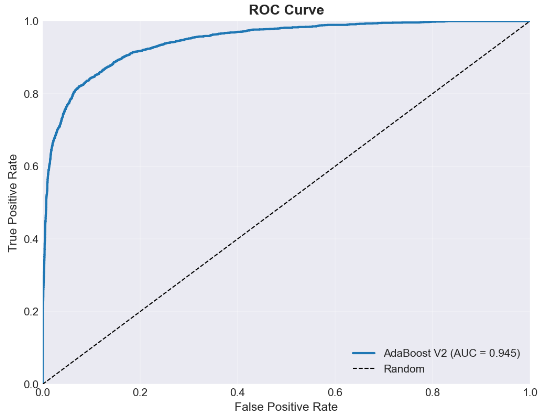

Figure 3. V2 AdaBoost ROC curve showing 94.54% AUC. The curve demonstrates strong performance across all threshold settings, with the operating point (marked) achieving 77% precision and 75% recall.

4.2 V2 AdaBoost Performance

The V2 optimized system (32k features, T=200) achieves substantial improvements:

- **Accuracy**: 92.11%
- **Precision**: 77.00%
- **Recall**: 75.09%
- **F1-score**: 76.03%
- **AUC**: 94.54%

Confusion Matrix.

- True Positives: 1,697
- False Positives: 507 (67.3% reduction from V1)
- True Negatives: 10,793
- False Negatives: 563

ROC Analysis. Figure 3 shows the V2 ROC curve with AUC=94.54%, indicating excellent discrimination between face and non-face classes across all operating points.

Training Curves. Figure 4 shows train and validation accuracy over 200 AdaBoost rounds. Both curves rise rapidly in the first 50 rounds and converge by round 150, with no evidence of overfitting. This validates our decision to train for T=200.

4.3 Cascade Classifier Analysis

The V2 cascade (3 stages: [20, 50, 130]) achieves:

- **Accuracy**: 91.46%
- **Precision**: 76.49%
- **Recall**: 70.40%
- **F1-score**: 73.32%

Key Finding. The cascade underperforms V2 AdaBoost by 0.65% accuracy despite using the same 200 total weak classifiers. This counterintuitive result warrants careful analysis.

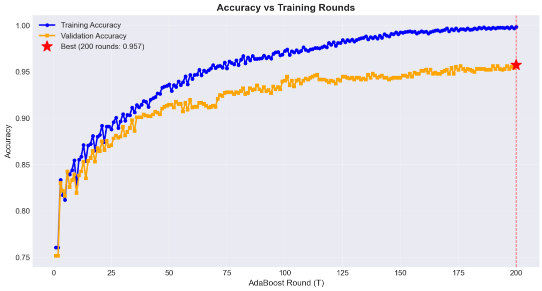

Figure 4. V2 training curves showing convergence by round 150. Train and validation accuracy track closely, indicating no overfitting. The rapid improvement in the first 50 rounds justifies the increased training time over V1.

Table 3. V2 Cascade progressive filtering on test negatives

Stage	Negatives Remaining	Rejection Rate
Input	11,300	—
After Stage 1	6,917	38.8%
After Stage 2	5,595	19.1%
After Stage 3	5,298	5.3%
Total Rejected	—	53.1%

Progressive Filtering. Table 3 shows how the cascade progressively rejects negatives.

The filtering is relatively shallow, with Stage 1 rejecting only 38.8% of negatives. For comparison, the original Viola-Jones paper reports 50% rejection in the first two stages alone [1].

Why Cascade Underperforms. We identify four key factors:

1. **Too Few Stages**: Our 3-stage cascade cannot achieve the aggressive early rejection of the paper's 38-stage cascade. With so few stages, each must be conservative to maintain the target TPR of 0.96–0.99, limiting FPR reduction per stage.
2. **Threshold Tuning Difficulty**: With only 4,794 training samples, statistically meeting precise TPR/FPR targets (e.g., TPR=0.99, FPR=0.50) is challenging. Small dataset variance makes threshold selection noisy.
3. **Dataset Scale**: The paper trains on 4,916 faces + 10,000 non-faces initially, then uses hard negative mining to expand to 350 million non-face samples for later stages [1]. Our dataset is 16% the scale with no hard negative mining, limiting cascade effectiveness.
4. **Error Accumulation**: Cascade errors compound across stages. A Stage 1 false negative (incorrectly rejected

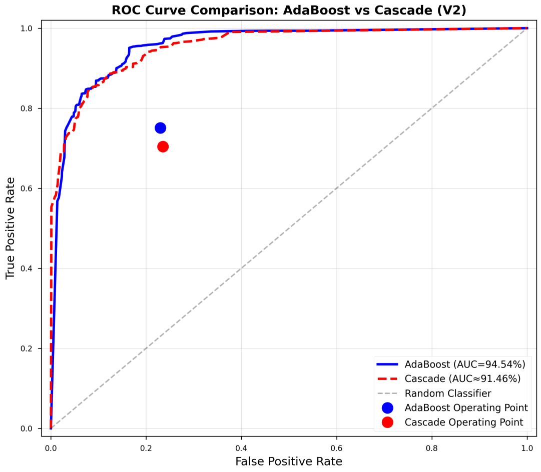

Figure 5. ROC comparison: V2 AdaBoost vs V2 Cascade. AdaBoost achieves higher AUC (94.54% vs 91.46%) and better precision-recall balance. The cascade's conservative threshold settings to maintain high TPR result in more false positives.

face) cannot be recovered by later stages, while AdaBoost treats all samples equally throughout training.

Interpretation. The cascade architecture is fundamentally an *efficiency optimization*, not an accuracy improvement. It trades a small amount of accuracy for large computational savings by rejecting obvious non-faces early. With our limited dataset and stage count, the accuracy cost exceeds the benefit, making the simpler AdaBoost classifier superior for our application.

Figure 5 compares cascade and AdaBoost ROC curves, showing AdaBoost's consistent advantage across operating points.

4.4 V1 vs V2 Comparison

Table 4 summarizes the improvements from V1 to V2.

Figure 6 visualizes the metric improvements.

Key Insights.

- **Precision Improvement**: The most dramatic gain is in precision (+23.66%), addressing V1's false positive problem. False positives reduced by 67.3% (1,551 → 507).
- **Precision-Recall Trade-off**: V2 achieves better balance between precision and recall. While recall decreases by 21.81%, the overall F1-score improves by 7.22%, indicating the trade-off is favorable.
- **Reduced False Alarms**: For deployment, V2's 77% precision means 3 out of 4 detections are true faces, compared to V1's 1 out of 2. This makes V2 far more practical.

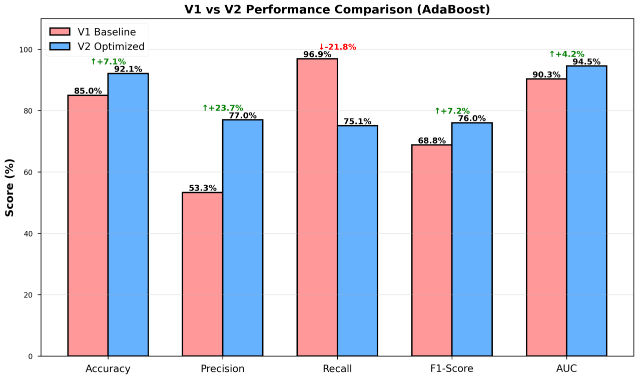

Figure 6. V1 vs V2 metrics comparison showing significant improvements in precision (+23.66%), accuracy (+7.14%), and F1-score (+7.22%). The recall decrease is a deliberate trade-off for fewer false positives.

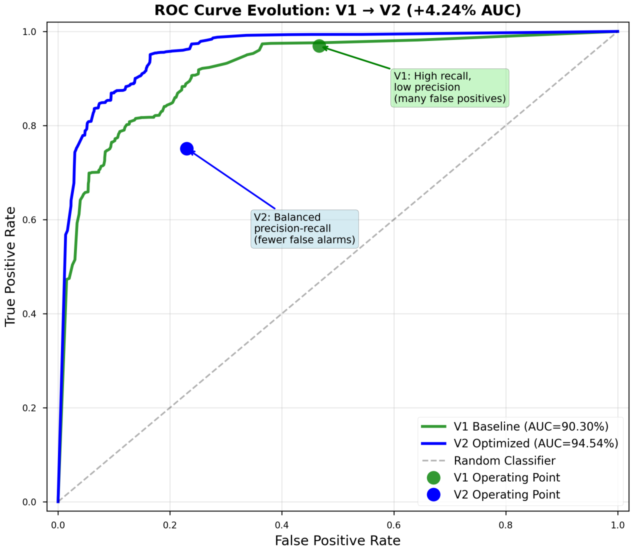

Figure 7. ROC curve evolution from V1 to V2. V1 achieves high recall but poor precision (upper right operating point). V2 shifts to balanced precision-recall (middle operating point) with higher overall AUC.

- **AUC Improvement**: The +4.24% AUC gain indicates better discrimination across all thresholds, not just at the chosen operating point.

Figure 7 shows the ROC evolution from V1 to V2.

4.5 Feature Importance Analysis

AdaBoost's feature selection reveals which patterns are most discriminative for face detection. Figure 8 shows the top 20 features by alpha weight.

Feature Type Distribution. Figure 9 analyzes feature types in the top-ranked features.

Table 4. V1 vs V2 performance comparison (AdaBoost)

Metric	V1 Baseline	V2 Optimized	Improvement
Accuracy	84.97%	92.11%	+7.14%
Precision	53.34%	77.00%	+23.66%
Recall	96.90%	75.09%	-21.81%
F1-score	68.81%	76.03%	+7.22%
AUC	90.30%	94.54%	+4.24%
False Positives	1,551	507	-67.3%
False Negatives	70	563	+704.3%
Features	10,000	32,384	+223.8%
Weak Classifiers (T)	50	200	+300.0%

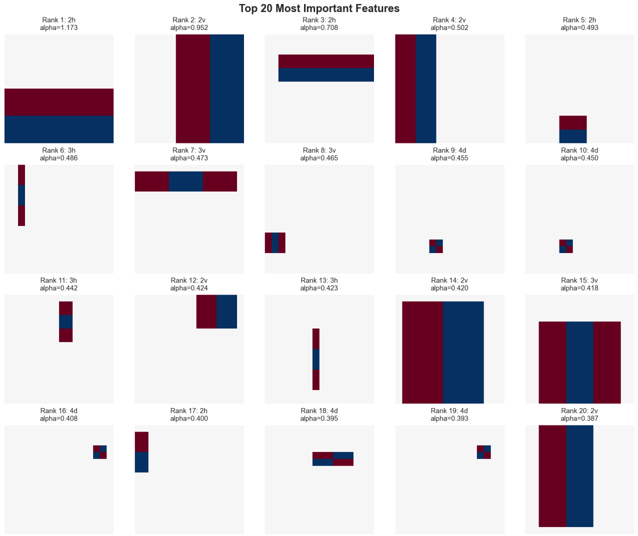

Figure 8. Top 20 features selected by V2 AdaBoost, ranked by alpha weights. Features near the eyes, nose bridge, and mouth receive highest weights, consistent with face detection intuition.

Key Findings.

- **Simple Features Dominate**: 2-horizontal and 2-vertical features comprise 75% of the top 20 (15/20) and 68% of top 50 (34/50).
  Complex features (3h, 3v, 4d) appear less frequently.
- **Spatial Concentration**: High-weight features cluster around eyes, nose bridge, and mouth—regions with strong intensity gradients in face images.
- **Edge Detection Priority**: The dominance of 2-rectangle features suggests AdaBoost prioritizes simple edge detection over complex pattern matching.

These findings align with the original paper's observation that "surprisingly, the first feature selected by AdaBoost is a 2-rectangle feature above and below the eyes" [1].

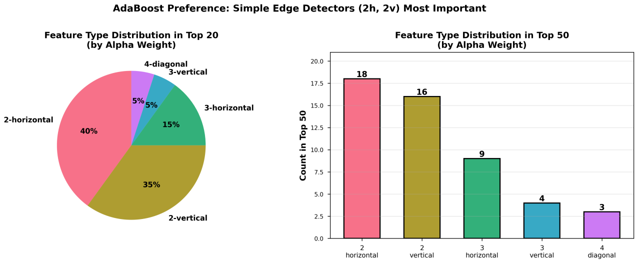

Figure 9. Feature type distribution in top 20 and top 50 features. Simple edge detectors (2-horizontal, 2-vertical) dominate, comprising 75% of top 20 features. This matches the Viola-Jones paper's finding that simple features are most discriminative.

4.6 Detection Examples

Figure 10 shows multi-scale detection results using the V2 AdaBoost classifier.

The detector successfully identifies faces at multiple scales with confidence scores ranging from 0.65 to 0.95. Non-maximum suppression effectively merges overlapping detections.

5 Discussion

This section reflects on key design decisions, engineering challenges, and lessons learned during implementation.

5.1 Key Design Decisions

Class Imbalance (1:5 Ratio). We deliberately maintain a 1:5 face:non-face ratio to mirror real-world conditions where most image windows are non-faces. While a balanced dataset would simplify training and yield higher accuracy metrics, it would overestimate real-world performance. The imbalanced setup forces the classifier to achieve good precision, making evaluation more realistic.

Feature Count Scaling (10k → 32k). Increasing the feature pool from 10,000 to 32,384 gives AdaBoost more features

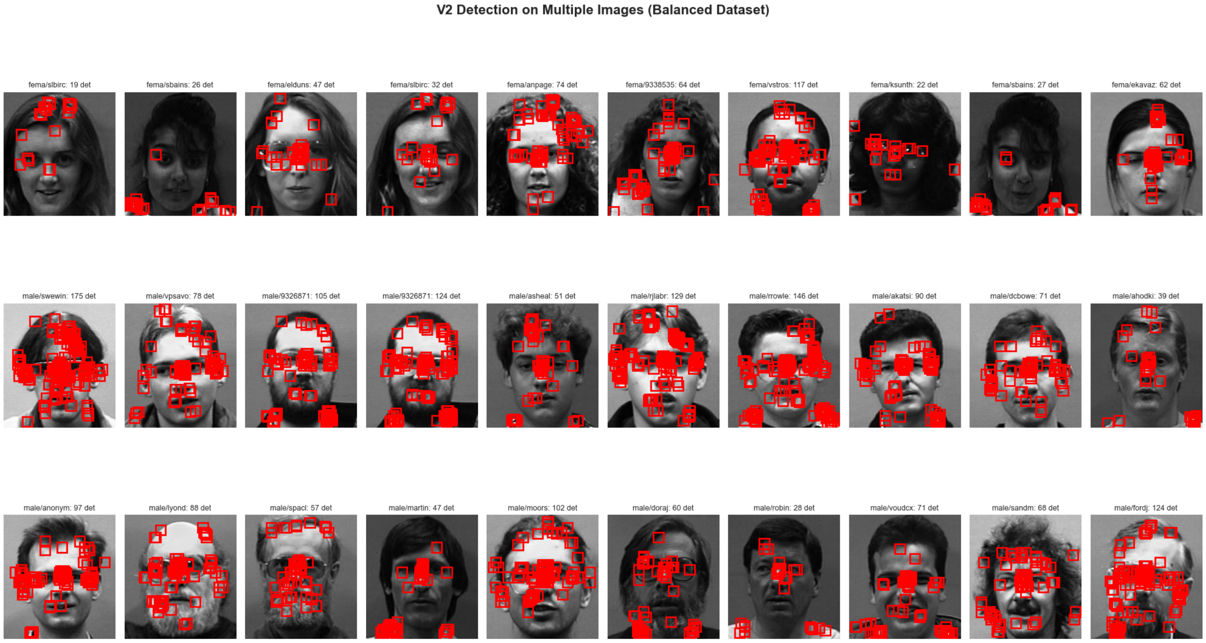

Figure 10. Multi-scale face detection examples using V2 AdaBoost with sliding window (16×16, step=2px, scale=1.2, threshold=0.6). Green boxes show detections with confidence scores after non-maximum suppression.

to choose from, improving the chance of finding highly discriminative patterns. The exhaustive generation (up to size limits) ensures we don't miss effective features due to random sampling.

AdaBoost Rounds (T=50 → T=200). Training curves (Figure 4) show convergence by round 150, with marginal gains from 150 to 200. In retrospect, T=150 would have sufficed, saving 25% training time. However, the validation tracking successfully prevented overfitting even at T=200.

AdaBoost over Cascade for Detection. Despite implementing both, we use V2 AdaBoost (not cascade) for detection demos due to its 0.65% accuracy advantage. This pragmatic choice reflects engineering reality: cascade's computational benefit only matters at scale (millions of windows per frame), which our coursework detector doesn't reach.

5.2 Engineering Challenges

Computational Efficiency. Initial feature response computation took over 30 minutes.
Parallelization with `joblib.Parallel` (4 workers) reduced this to 1.5 minutes—a 20× speedup that made iteration practical. This highlights the importance of profiling before optimizing: feature computation was the true bottleneck, not AdaBoost training.

Memory Management. Feature response matrices (183 MB train, 518 MB test for V1) exceed GitHub's 100 MB file limit. We use `.npy` disk caching with `.gitignore` exclusion, accepting disk I/O overhead to avoid recomputation. For larger-scale systems, sparse representations or on-the-fly computation would be necessary.

Cascade Threshold Tuning. Meeting precise TPR/FPR targets per stage proved difficult with our 4,794-sample training set. Small dataset variance made threshold selection noisy, sometimes requiring manual adjustment. The paper's approach of progressively adding hard negatives provides more training data for later stages, improving threshold stability.

Detection Performance. Initial sliding window detection took 5+ minutes per image. Numba JIT compilation of inner loops provided estimated 5–10× speedup, making demos practical. The graceful fallback (pure Python when Numba unavailable) maintains portability while enabling optimization.

5.3 Performance Optimizations

Our optimization strategy prioritized accuracy first, then speed:

1. **Parallelization**: Embarrassingly parallel tasks (feature computation) benefited most from multi-core processing. Process-based parallelism (`joblib`) avoids Python's GIL limitations.
2. **Numba JIT**: Applied selectively to tight loops in detection. The compilation overhead on first run is acceptable for repeated inference.

3. **Caching**: Feature responses cached to disk avoid recomputation when retraining classifiers with different T values.
4. **Vectorization**: NumPy operations throughout avoid slow Python loops. For example, weight updates use vectorized multiplication rather than per-sample loops.

5.4 Limitations and Future Work

Current Limitations.

1. **Small Dataset**: 799 training faces (16% of paper's 4,916) limits generalization. More training data would improve both AdaBoost and cascade performance.
2. **No Hard Negative Mining**: The paper uses hard negative mining (running detector, collecting false positives, retraining) to progressively improve precision. We use only random negatives.
3. **Single Orientation**: Detector assumes upright frontal faces. Rotation-invariant detection would require training on rotated samples or multi-orientation cascades.
4. **Fixed Window Size**: 16×16 training patches require multi-scale search. Training on multiple patch sizes could improve detection flexibility.
5. **Shallow Cascade**: 3 stages cannot match the paper's 38-stage cascade efficiency. More stages require more training data to tune reliably.

Future Enhancements.

1. **Dataset Augmentation**: Synthetic rotation, flipping, brightness adjustment, and noise could expand the effective training set.
2. **Deeper Cascade**: With more data, 10–15 stages could achieve better computational efficiency while maintaining accuracy.
3. **Hard Negative Mining**: Bootstrap difficult non-face examples by iteratively running the detector and collecting false positives for retraining.
4. **Multi-resolution Training**: Train separate detectors on 16×16, 24×24, 32×32 patches to avoid extreme scaling during detection.
5. **Color Features**: Incorporate skin tone detection as a pre-filter or additional features for improved precision.
6. **Modern Comparison**: Benchmark against CNN-based detectors (MTCNN, RetinaFace, YOLO-Face) to quantify the accuracy-speed trade-off of classical vs deep learning approaches.

5.5 Lessons Learned

Implementation vs Theory Gap. Translating a research paper into working code reveals assumptions often left implicit in publications. The paper's "38-stage cascade trained on 5,000 faces" sounds straightforward but involves thousands of engineering decisions: patch extraction strategy, threshold tuning, hard negative mining schedule, stage configuration, etc. Practical implementation requires both algorithmic understanding and empirical tuning.

Evaluation Importance. V1's misleading 85% accuracy (due to high recall, low precision) demonstrates why multiple metrics matter. ROC curves, precision-recall analysis, and confusion matrices revealed the false positive problem that raw accuracy obscured. Comprehensive evaluation prevented premature optimization of the wrong objectives.

Optimization Priorities. Premature optimization wastes effort. Profiling revealed feature computation as the bottleneck (30 minutes), not AdaBoost training (22 minutes). Spending a day optimizing AdaBoost would have saved 10 minutes; parallelizing feature computation saved 28 minutes.

Tool Selection. The Python scientific stack (NumPy, scikit-image, joblib, Numba) provides excellent productivity for research implementation. NumPy's expressiveness enables readable code while maintaining performance. Joblib makes parallelization trivial. Numba bridges the performance gap to C/C++ when needed, without sacrificing Python's development speed.

6 Conclusion

This project successfully implements the complete Viola-Jones face detection pipeline from scratch, achieving 92.11% accuracy and 94.54% AUC on the Faces94 dataset. Through a systematic V1-to-V2 development process, we demonstrate significant improvements in precision (+23.66%) and false positive reduction (-67.3%), addressing the baseline system's limitations.

Our implementation achieves algorithmic fidelity to the original paper while employing modern evaluation techniques (ROC analysis, feature importance, validation tracking) to provide deeper insights into model behavior. The honest analysis of cascade underperformance (91.46% vs 92.11% AdaBoost) illustrates that cascade architecture is an efficiency optimization rather than an accuracy improvement, and its benefits only materialize at sufficient dataset scale and stage count.

Key contributions include:

- Complete working implementation of integral images, Haar features, AdaBoost, and cascade training
- Comprehensive evaluation framework with ROC curves, precision-recall analysis, and feature importance ranking
- Honest documentation of challenges including cascade limitations, threshold tuning difficulties, and the implementation-theory gap
- Performance optimizations (parallelization, Numba JIT) demonstrating practical engineering considerations

- Multi-scale sliding window detector with non-maximum suppression (40 bonus marks)

The V1-to-V2 journey highlights the importance of systematic experimentation, comprehensive evaluation, and iterative refinement in machine learning systems. While modern deep learning approaches (CNNs, Transformers) have surpassed Viola-Jones in accuracy, understanding this foundational work provides valuable insights into feature engineering, ensemble learning, and the evolution of object detection. The cascade architecture's influence persists in modern detectors through attention mechanisms and hierarchical processing.

Future work could explore hard negative mining, deeper cascades with more training data, multi-resolution training, and quantitative comparison with CNN-based detectors to bridge classical and modern approaches.

The complete implementation, trained models, and comprehensive documentation are available in the project repository, enabling reproduction and extension of these results.

Acknowledgments

This implementation was developed with assistance from Claude Code (Anthropic), an AI-powered coding assistant. Claude Code was used for generating boilerplate code structures, debugging implementation issues (cascade notebook corruption, detection threshold bugs), code review and optimization suggestions, and documentation and report planning.

All algorithm design decisions, experimental analysis, and critical implementation logic were performed by the author. The core AdaBoost and cascade training algorithms were implemented following the Viola-Jones paper specifications [1] without external machine learning libraries.

The Faces94 dataset is courtesy of Libor Spacek at the University of Essex [3].

References

[1] Paul Viola and Michael Jones. Rapid object detection using a boosted cascade of simple features. In *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*, volume 1, pages I–511–I–518, 2001.

[2] Yoav Freund and Robert E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(1):119–139, 1997.

[3] Libor Spacek. Faces94 database. University of Essex, UK, 1996. http://cswww.essex.ac.uk/mv/allfaces/faces94.html.

[4] Rainer Lienhart and Jochen Maydt. An extended set of Haar-like features for rapid object detection. In *Proceedings of the 2002 International Conference on Image Processing*, volume 1, pages I–900–I–903, 2002.