COMP61342: Cognitive Robotics and Computer Vision Assignment

Aditya Vikram Nigam
University ID: 11624014
Department of Computer Science
University of Manchester
adityavikram.nigam@postgrad.manchester.ac.uk

1 Introduction

Object recognition is a central task in computer vision that involves identifying objects and categorising images and is fundamental to a wide range of robotic applications. Broadly, two main paradigms exist to tackle this problem: traditional feature-based methods and modern deep learning-based approaches. The idea behind traditional computer vision techniques is to manually formulate some way of representing the image by encoding the existence of various features that is suitable for classical machine learning algorithms. In contrast, convolutional neural networks (CNNs), inspired by the human visual system, learn hierarchical representations from the raw image and now form the backbone of state-of-the-art object recognition systems.

2 Datasets

This study uses two benchmark datasets: Fashion-MNIST and CIFAR-10. Fashion-MNIST [1] is a dataset of Zalando's fashion product images, designed as a more challenging alternative to the original MNIST handwritten digit dataset. It consists of 60,000 training and 10,000 test images, each of size 28×28 pixels in grayscale. The class labels in Fashion-MNIST are T-shirt/top, trouser, pullover, dress, coat, sandal, shirt, sneaker, bag, and ankle boot. CIFAR-10 [2] is one of the most commonly used datasets in computer vision, containing a total of 60,000 colour images of size 32×32 pixels, divided equally across 10 distinct classes. The dataset is split into 50,000 training images and 10,000 test images, with 6,000 images per class. The classes are airplane, automobile (but not truck or pickup truck), bird, cat, deer, dog, frog, horse, ship, and truck (but not pickup truck).

An important consideration in this selection was to include datasets with varying characteristics. Fashion-MNIST offers grayscale images, while CIFAR-10 consists of colour images.

3 Classification using Local features and CV algorithms

A traditional computer vision pipeline includes steps such as local feature detection and encoding, representation construction, and classifier selection. It offers a manual, fine-grained control to systematically explore and evaluate different strategies. By tuning each component independently, we gain insights into how specific design choices affect performance on different datasets. The code (python implementation) for this entire section is attached in Appendix A.

3.1 Local feature Detectors

The first and arguably most critical step in a traditional CV pipeline is interest point (keypoint) detection, which identifies image regions for further processing. The quality and density of keypoints significantly influence overall performance, as they form the basis for subsequent classification. Ideal features should be repeatable under varying imaging conditions, distinctive, and invariant to scale, rotation, and illumination. Identifying the appropriate region around the feature is a key step, which requires selecting the correct scale at which the feature is most distinctive and stable in the image. Figure 1 is an illustration of feature extraction on an image from the Fashion-MNIST dataset by the four different methods.

3.1.1 SIFT Keypoints

The SIFT (Scale-Invariant Feature Transform) [3], is a widely used algorithm for detecting and describing local features in images using the Difference of Gaussians (DoG) method, which is an efficient approximation of the Laplacian of Gaussian (LoG), a second-order derivative operator that detects blobs or areas of rapid intensity change. The

DoG is computed as:

$$D(x, y, \sigma) = L(x, y, k\sigma) - L(x, y, \sigma)$$

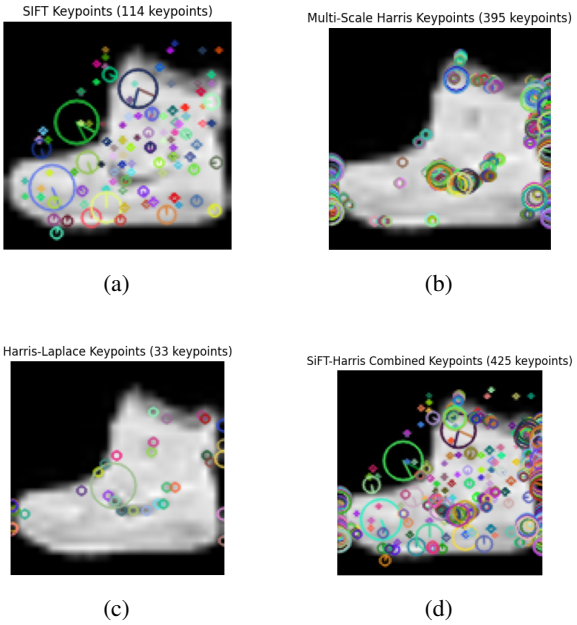

Figure 1: Feature Extraction Illustration

for some constant $k > 1$. Here, $L(x, y, \sigma)$ is the image blurred with a Gaussian of standard deviation σ, defined as:

$$L(x, y, \sigma) = G(x, y, \sigma) * I(x, y)$$

where $G(x, y, \sigma)$ is a Gaussian kernel with standard deviation σ, and $I(x, y)$ is the input image. Keypoints are detected by finding local extrema (minima or maxima) in different scales by comparing each pixel to its 26 neighbours (8 in the same scale and 9 in each adjacent scale).

3.1.2 Multi-Scale Harris

The multi-scale Harris corner detection method extends the original Harris corner detector [4] by adapting it to multiple scales. The corner response is computed using the second moment matrix M over a Gaussian window:

$$M = \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}$$

The Harris response function is given by $R = \det(M) - k \cdot (\text{trace}(M))^2$.

This multi-scale approach constructs a Gaussian scale-space pyramid, applying the Harris corner measure at multiple scales, and local extrema across both space and scale are selected.

3.1.3 Harris-Laplace

This method draws inspiration from the Harris-Laplace detector [5]. It introduces the idea of the 'characteristic scale' of a local structure, indicated by a local extremum over the scale of normalised derivatives (Laplacian-based scale selection). A multi-scale representation for the Harris interest point detector is constructed, and then points at which the Laplacian is maximal over scales are selected. The DoG method is used for an efficient approximation.

3.1.4 SIFT-Harris Combined Keypoints

A novel method implemented in this study is a hybrid detector that integrates keypoints detected via SIFT (DoG) and multi-scale Harris corner detection to leverage their complementary strengths. Both detectors are independently applied across multiple scales to extract a rich set of candidate keypoints. To avoid redundancy and ensure distinctiveness, Harris keypoints that spatially overlap within 3 pixels of SIFT keypoints (duplicates) are removed. This is inspired by the idea that both blob-like structures and corners are interest points and potentially work better together down the pipeline.

3.2 Visual Vocabulary Construction and Classification

Local descriptors are computed out of the keypoints found to encode invariant neighbourhood information. Among various descriptors, like SURF, BRIEF, ORB etc, SIFT [3] continues to be widely used for both academic benchmarking and practical applications and hence has been chosen for this study. It constructs 128-dimensional vectors by computing weighted gradient orientation histograms over a 16×16 window centred at each keypoint. To enable classification, the variable-length descriptor sets need to be transformed into fixed-length feature vectors. The Bag-of-Words (BoW) model [6] addresses this by learning a visual vocabulary via clustering a large sample of descriptors, and each image is encoded as a histogram of visual word occurrences. Among various clustering algorithms, K-Means is the most widely used due to its simplicity and scalability, while alternatives like hierarchial K-means can be used for large databases. For classification, a Support Vector Machine (SVM) is employed due to its effectiveness in high-dimensional spaces, as demonstrated by prior work [6, 7].

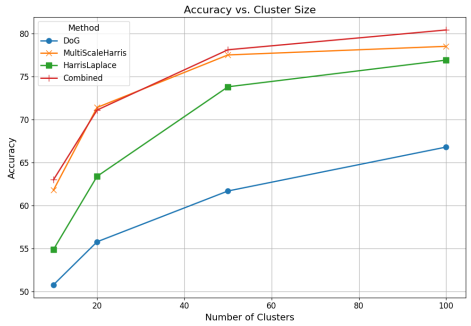

Figure 2: Accuracy vs Cluster size (Fashion-MNIST)

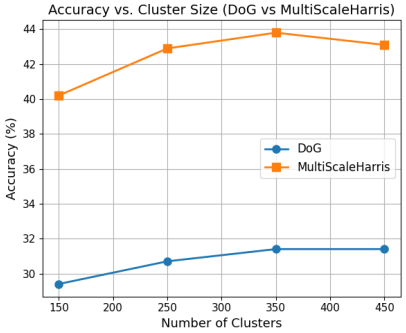

Figure 3: Accuracy vs Cluster size (Cifar10)

3.3 Experimental Design and Hyperparameter Exploration

In this study, the experiments were designed to focus on extensively exploring the early stages of the traditional pipeline – feature extractors and clustering. All 4 feature extraction methods were experimented on with both datasets. K-Means clustering was employed for quantisation, and an SVM was used for classification, justified by this combination's usage in almost all relevant research areas. The strategy was first searching for the optimal size of visual vocabulary, i.e., the number of clusters k for both datasets (Figure 2 and Figure 3). Fixing optimal k, individual hyperparameters were explored for the different feature extractors to analyse how these choices affect the classification accuracy specific to the dataset.

For SIFT-based methods, $contrast\_threshold$ controls the minimum local DoG response magnitude required to retain a keypoint; lower values increase sensitivity to low-contrast features. $edge\_threshold$ sets a curvature ratio limit to discard keypoints poorly localised along edges, with higher values retaining more keypoints. Figure 4 shows how the classification accuracy is affected by varying these parameters for the two datasets. For Harris Corner-based methods, $harris\_thresh$

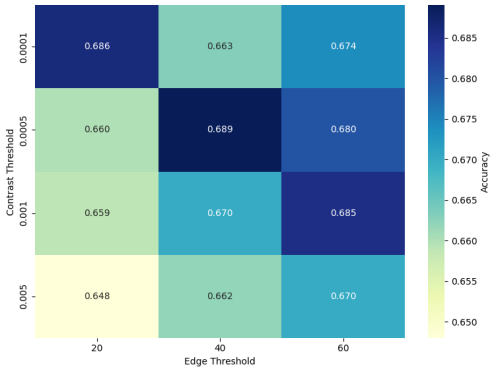

Figure 4: Heatmap (Fashion-MNIST)

determines the response sensitivity to corners, with higher values favouring fewer features having sharper corners. For comparison, basic parameters related to the number of scales and scale factor, common to all feature extraction methods, were set to commonly used values. A subset of 5000 images from both datasets was used for all the experiments in this section due to computational cost and training time. These images were upscaled 128×128 to be compatible with the feature extractors following standard practice.

3.4 Results

The optimal classification accuracies found after extensive hyperparameter exploration are described in TABLE 1. Interestingly, for both datasets, the same set of hyperparameters yielded the optimum results (Table 2).

Feature Detector	**Accuracy (Fashion-MNIST)**	**Accuracy (CIFAR-10)**
SIFT Keypoints	69	33
Multi-Scale Harris	81	48
Harris-Laplace	78	47
SIFT-Harris Combined	**82.5**	**50**

Table 1: Optimal classification accuracy for both datasets

For the MNIST dataset, the graph in Figure 2 clearly shows the optimal accuracy saturates for all methods around the values of 100 clusters, and further analysis showed less than a 0.1% increase before starting to drop. For CIFAR-10, Figure 3 shows the optimal value is 350, as the accuracy starts to dip afterwards. This demonstrates that the optimal number of clusters depends on the complexity of the datasets, where a small K leads to underfitting and loss of discriminative detail, whereas

an excessively large K may lead to sparsity and overfitting.

Hyperparameter	Optimal Value
contrast_threshold	0.0005
edge_threshold	40
harris_thresh	0.005

Table 2: Optimal Hyperparameter Values

The first striking difference noticed from Table 1 is that the accuracy on the CIFAR-10 dataset is significantly lower (by around 40%) as compared to the Fashion-MNIST dataset. This suggests that the traditional CV methods experimented on the coloured CIFAR-10 dataset struggle to build a desirable feature pool for better classification. This can be attributed to the relatively higher complexity and nature of images. The optimal feature extraction method for both datasets was the novel method that used the combined Harris and DoG features to give the highest accuracies. This indicates that these features can complement each other and provide a rich and diverse feature pool for object recognition. Figure 4 suggests that a value for thresholds ($contrast\_threshold$ and $edge\_threshold$) that is low enough to not eliminate important features while not including too many features as noise is the optimum SIFT setting. On the other hand, the Harris detector accuracy kept on increasing as the threshold was lowered before saturating. The saturation point was taken as the optimal threshold in this case.

Method	No. of Features	Extraction Time
DoG (SIFT)	500K	50
Multi-Scale Harris	1500K	200
Harris-Laplace	150K	110
SIFT-Harris Combined	1900K	300

Table 3: Feature extraction volume (in thousands) and speed (in seconds) on Fashion-MNIST (5000 images)

The Harris-Laplace detector, although being slightly less accurate than the Multi-Scale Harris detector, provides that high accuracy with just one-tenth the number of features (Table 3), suggesting high quality of features is obtained with the characteristic scale concept. In terms of feature extraction time, optimized Harris-Laplace was almost 2 times faster than multi-scale Harris, the combined feature method was the slowest, and SIFT was the fastest.

4 Classification using CNN

4.1 Network Architecture

A Convolutional Neural Network (CNN) architecture is defined by the arrangement of different kinds of layers on top of each other. Convolutional and pooling layers are designed for feature extraction in a hierarchical fashion. The feature maps outputted by these layers are flattened and fed into fully connected (dense) layers that perform affine transformations for global reasoning and classification.

4.1.1 Convolutional Layers

Each convolutional layer applies a set of learnable kernels across the spatial dimensions of the input. Each filter convolves over the input to compute dot products over local receptive fields, producing activation maps that encode spatially local correlations. For an input X and filter W, the output at location (i, j) is:

$$Y_{i,j} = \sum_{m,n} X_{i+m,j+n} \cdot W_{m,n} + b$$

Rectified Linear Unit (ReLU) is typically applied post-convolution for non-linear activation. Batch normalization is used to normalize activations across a mini-batch to zero mean and unit variance to improve numerical stability while training.

4.1.2 Pooling and Dropout Layers

Pooling layers reduce spatial resolution of feature maps over a window by selecting the dominant activation in each region. The stride defines how many pixels the window moves across the input at each step. The most commonly used technique is max pooling, while average pooling is also used in some architectures. Pooling allows the network to learn hierarchical, increasingly abstract representations while also reducing dimensionality, which favors generalization.

Dropout is a regularisation technique that randomly deactivates a proportion of neurons during training and is primarily used to reduce overfitting. It can be used in both convolutional and fully connected layers.

4.2 Experimental Design and Hyperparameter Exploration

This study explores hand-crafted network architectures from scratch, instead of pre-trained state-of-the-art models, for the purpose of classification on

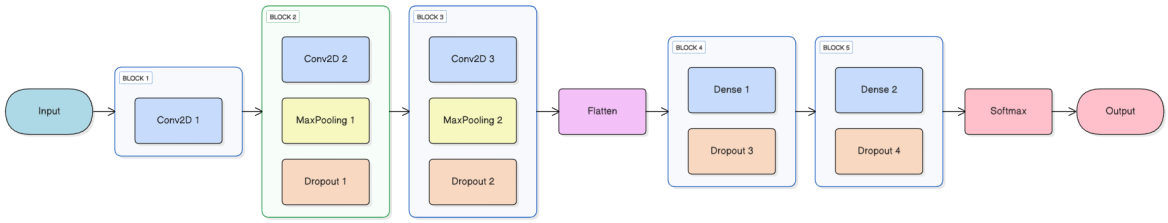

Figure 5: 3 Layer CNN Architecture with 2 Pooling Layers

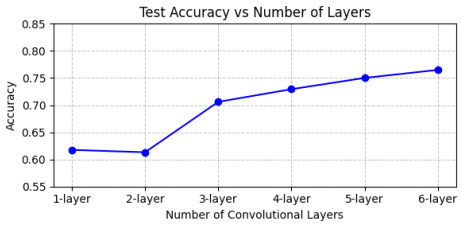

Figure 6: Test Accuracy vs Number of layers (Cifar10)

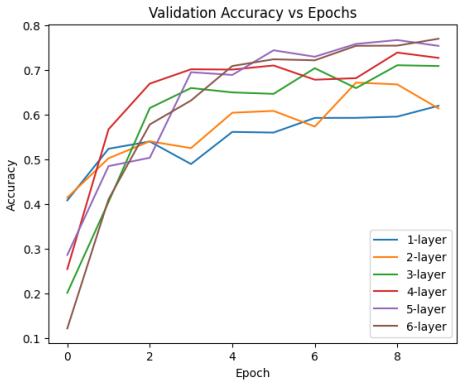

Figure 7: Training graph for different models (Cifar10)

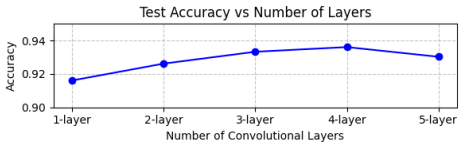

Figure 8: Test Accuracy vs Number of layers (Fashion-MNIST)

the two datasets. This decision was to focus on more experimentation and insights rather than on very high accuracies and longer training times. To design an effective CNN architecture, we first investigated how the depth of the architecture (the number of convolutional layers) impacts classification performance.

Different pooling and dropout strategies were examined in these architectures. Different numbers of pooling layers were variously placed, and combinations of dropout layers with rates between 0.25 and 0.5 were experimented with. A 2×2 window for max-pooling was used, following standard design heuristics. Two hyperparameters, batch size and learning rate, which strongly affect the training dynamics and generalisation together, were chosen for exploration. A focused search was implemented to find the optimal combination of these parameters for classification. We also experimented systematically with architectural hyperparameters: initial number of filters, filter size, and final fully connected layer size to find a correct filter size and balance between underfitting and overfitting. Full datasets were utilised for training CNNs as using only 5000 images was highly overfitting the model.

4.3 Results

The optimal classification accuracy achieved using a custom-designed CNN on Fashion-MNIST was 12% higher than that in CIFAR-10, as shown in Table 4 with the corresponding hyperparameters. Figure 6 shows that on the CIFAR-10 dataset, the validation and test accuracy increases gradually and noticeably with increasing layers, plateauing at the optimum of 6 layers. Figure 7 shows a desirable stable increase in validation accuracy over the training epochs, with the 6-layer model having the smoothest progression. On the other hand, for the Fashion-MNIST dataset (Figure 8), there are only very slight improvements ($< 2\%$) when layers are increased. A 3-layer architecture (Figure 5) was deemed optimal, as a 4-layer model would double the parameter count with no tangible performance benefit. The trend difference between the two datasets reflects that the grayscale Fashion-MNIST, with its simpler, texture-based class separations, can be effectively modelled using shallow networks, whereas CIFAR-10's coloured, diverse object categories demand more expressive architectures and training strategies.

Figure 9 shows that a higher learning rate (2e-3) combined with a smaller batch size (32) performed

Configuration	Fashion-MNIST	CIFAR-10
Accuracy (%)	**94**	**82**
Learning Rate	0.0006	0.002
Batch Size	64	32
Convolutional Layers	3	6
Pooling Layers	2	3
Dense Layers	2 (512, 128)	2 (512, 128)
Epochs	15	20

Table 4: Optimal classification accuracy and CNN hyperparameters for each dataset

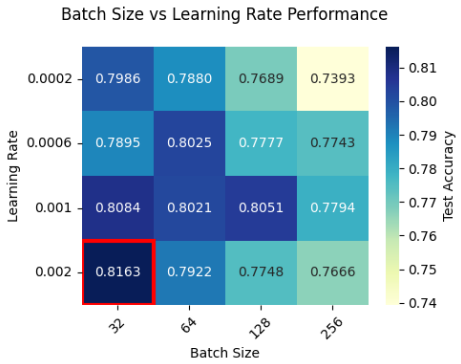

Figure 9: Heatmap (Cifar10)

the best, suggesting that rapid updates on smaller, diverse mini-batches helped navigate the complex loss landscape more effectively. On the other hand, Fashion-MNIST benefitted from a lower learning rate (6e-4) and a batch size of 64, which stabilised learning and avoided overshooting in its feature space. A combination of alternate layers that included max pooling and dropout came out to be the best strategy. For the 3-layer CNN model used on Fashion-MNIST, the max pooling and dropout layer, when added to both $2nd$ and $3rd$ convolutional layers, performed better by balancing feature abstraction with regularisation and helping prevent overfitting. Dropout with a rate of 0.25 after convolutional blocks and with a higher rate of 0.5 after the dense layers performed the best to generalise the model and increase test accuracy.

Another observation was that the difference between test and training accuracy was a minor 2-3% on Fashion-MNIST but was around 8-10% on CIFAR-10, which was further decreased to 6-7% by tuning the CNN model with optimal learning rate and batch size. This indicated that although 90+ training accuracies were achieved on both datasets, the CIFAR-10 posed a generalisation challenge and limit to the CNN model.

5 Comparison of Results

The results clearly show that the CNN-based approach outperforms traditional feature CV approaches across both datasets. On Fashion-MNIST, the CNN achieves 94% accuracy, compared to 82.5% from the best traditional method (SIFT-Harris). The gap is even wider on CIFAR-10, where the CNN reaches 82%, while the top traditional method achieves only 50%. The difference stems from the fact that handcrafted features like SIFT or Harris rely on fixed descriptors and struggle with diverse textures and colour information, particularly when image resolution is low and object boundaries are less distinct. CNNs learn end-to-end hierarchical features directly from the data by focusing on local regions of the input data and parameter sharing across spatial locations, which allows them to recognise objects with translational invariance. However, it's noteworthy that on Fashion-MNIST, traditional CV methods using a carefully tuned combined feature detector achieved competitive results. This suggests that with appropriate engineering, classical CV pipelines still provide unique value to domain understanding in niche domains. This is the reason believed to enable better adaptation in complex and large datasets. Table 5 shows this comparison along with the pretrained state of the art model performance as well. This also highlights that the 3-layer architecture implemented and tuned from scratch in this study was able to perform almost as accurately as the state-of-the-art model.

Dataset	Local Features	Custom CNN	ResNet (Pretrained)
Fashion-MNIST	82.5%	94%	95.7
CIFAR-10	50%	82%	94.6%

Table 5: Performance comparison of object recognition methods across datasets

Future work associated with this study could include implementing data augmentation techniques for a better comparison of generalisability and testing local feature extractors with full datasets.

6 State of the art in Computer Vision

Modern robotics leverages Computer Vision for a range of applications, from object detection and recognition to semantic mapping, visual SLAM (Simultaneous Localisation and Mapping), LIDAR (Light Detection and Ranging) and grasp planning.

These technologies enable massive fields like autonomous navigation, industrial robots for precision tasks, agricultural automation and medical robots.

Deep learning has driven significant progress in robot vision by enabling perception and decision-making in complex environments. Convolutional Neural Networks (CNNs) are widely used for image processing and feature extraction, allowing robots to accurately identify and track target objects. For instance, industrial robots have shown enhanced precision in tasks like object sorting and assembly line optimisation by utilising CNNs and achieving 92.6% accuracy [8]. Recent architectures like ResNet [9] introduce residual connections, allowing for much deeper networks and improved gradient flow, while EfficientNet [10] proposed compound scaling strategies for optimal performance-efficiency trade-offs. For spatial understanding, state of the art architectures like Mask R-CNN [11] have enabled fine-grained semantic segmentation, allowing robots to differentiate between scene elements like tools, humans, and environmental structures. Lightweight detectors such as YOLOv9 [12] have increasing adoption in robotics due to their high inference speed and adaptability to low-power edge devices. These models are foundational for obstacle avoidance, context-aware navigation, and task-specific manipulation.

Recent advances incorporate transformer-based architectures into robotic vision. Vision Transformers (ViT) [13] apply self-attention over image patches, enabling the modelling of long-range dependencies. The Segment Anything Model (SAM) [14] is a promptable vision model developed by Meta AI for general-purpose image segmentation. Multimodal vision-language models such as CLIP [15] introduce new capabilities like zero-shot classification and natural language-conditioned perception. Deep reinforcement learning, including Q-learning, is also gaining traction [16] in object detection and tracking and autonomous driving, where reward-driven learning supports closed-loop control and adaptability.

Despite these advances, several challenges remain. Deep vision systems often require large annotated datasets, which are difficult to acquire in robotic domains. This motivates research in self-supervised learning, few-shot learning and domain adaptation to enable data-efficient generalisation. Other open problems include real-time inference under hardware constraints, robustness to occlusions and environmental variation, and the lack of explainability in applications such as healthcare and autonomous driving. As research continues, emphasis is placed on improving efficiency, interpretability, and cross-domain generalisation to bring intelligent robots into safety-critical and dynamic real-world environments. For the integration of computer vision, machine learning and robotics, [17] suggests that future efforts should prioritise optimising lightweight models with edge computing, improving decision-making interpretability, and developing a federated learning-driven data ecosystem.

References

[1] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms," *CoRR*, vol. abs/1708.07747, 2017.

[2] A. Krizhevsky, "Learning multiple layers of features from tiny images," *University of Toronto*, 05 2012.

[3] L. D. G., "Distinctive image features from scale-invariant keypoints," *Journal of Computer Vision*, vol. 60, no. 2, pp. 91–110, 2004.

[4] C. Harris, M. Stephens, *et al.*, "A combined corner and edge detector," in *Alvey vision conference*, vol. 15, pp. 10–5244, Citeseer, 1988.

[5] K. Mikolajczyk and C. Schmid, "Scale & affine invariant interest point detectors," *International journal of computer vision*, vol. 60, pp. 63–86, 2004.

[6] G. Csurka, C. Dance, L. Fan, J. Willamowski, and C. Bray, "Visual categorization with bags of keypoints," in *Workshop on statistical learning in computer vision, ECCV*, vol. 1, pp. 1–2, Prague, 2004.

[7] G. M. Foody and A. Mathur, "A relative evaluation of multiclass image classification by support vector machines," *IEEE Transactions on geoscience and remote sensing*, vol. 42, no. 6, pp. 1335–1343, 2004.

[8] S.-C. Chen, R. S. Pamungkas, and D. Schmidt, "The role of machine learning in improving robotic perception and decision making," *International Transactions on Artificial Intelligence*, vol. 3, no. 1, pp. 32–43, 2024.

[9] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015.

[10] M. Tan and Q. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in *International conference on machine learning*, pp. 6105–6114, PMLR, 2019.

[11] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask r-cnn," in *Proceedings of the IEEE international conference on computer vision*, pp. 2961–2969, 2017.

[12] C.-Y. Wang, I.-H. Yeh, and H.-Y. Mark Liao, "Yolov9: Learning what you want to learn using programmable gradient information," in *European conference on computer vision*, pp. 1–21, Springer, 2024.

[13] A. Dosovitskiy, Beyer, *et al.*, "An image is worth 16x16 words: Transformers for image recognition at scale," *arXiv preprint arXiv:2010.11929*, 2020.

[14] A. Kirillov, E. Mintun, N. Ravi, *et al.*, "Segment anything," pp. 4015–4026, 2023.

[15] A. Radford, J. W. Kim, C. Hallacy, *et al.*, "Learning transferable visual models from natural language supervision," *ICML*, 2021.

[16] A. M. Hafiz, S. A. Parah, and R. Bhat, "Reinforcement learning applied to machine vision: state of the art," *International Journal of Multimedia Information Retrieval*, vol. 10, no. 2, pp. 71–82, 2021.

[17] Y. Gao, Z. Zhang, X. Zhu, and S. Ding, "Research progress on the integration of robot vision, computer vision and machine learning: Technological evolution, challenges and industrial applications," *Int J Cur Res Sci Eng Tech*, vol. 8, no. 2, pp. 257–262, 2025.

A Code for Local features using CV (Section 3)

```
# Feature extraction methods

# SIFT
def extract_dog_features(image,
    contrastThreshold=0.0005,
    edgeThreshold=40):
    sift = cv2.SIFT_create(
        contrastThreshold=
        contrastThreshold, edgeThreshold
        =edgeThreshold)
    keypoints, descriptors = sift.
        detectAndCompute(image, None)
    return keypoints, descriptors

# Multi-Scale Harris

def multi_scale_harris(image,
    harris_thresh=0.01):
    """
    Harris-Laplace keypoint detector
        with SIFT descriptors.

    Parameters:
        image (np.ndarray): Grayscale
            image (uint8 or float32).
        num_scales (int): Number of
            scales.
        scale_factor (float): Scale
            increment per octave.
        harris_thresh (float): Threshold
             to keep strong Harris
            responses.
        k (float): Harris detector free
            parameter.

    Returns:
        keypoints (list of cv2.KeyPoint)
            : Final keypoints.
        descriptors (np.ndarray):
            Corresponding SIFT
            descriptors.
    """
    if image.dtype != np.float32:
        image = np.float32(image)

    num_scales=4
    scale_factor=1.4
    k=0.04

    harris_responses = []
    blurred_images = []

    # Step 1: Harris detection at
        multiple scales
    for scale in range(num_scales):
        sigma = scale_factor ** scale
        blur = cv2.GaussianBlur(image,
            (0, 0), sigmaX=sigma, sigmaY
            =sigma)
        harris = cv2.cornerHarris(blur,
            blockSize=2, ksize=3, k=k)
        harris_responses.append(harris)
        blurred_images.append(blur)

    keypoints = []

    # Step 2: Local maxima across scales
    for scale_idx in range(num_scales):
        harris = harris_responses[
            scale_idx]
        coords = np.argwhere(harris >
            harris_thresh * harris.max()
            )

        for y, x in coords:
            current_val = harris[y, x]
            is_max = True
            for neighbor_idx in [
                scale_idx - 1, scale_idx
                 + 1]:
                if 0 <= neighbor_idx <
                    num_scales:
                    neighbor =
                        harris_responses
                        [neighbor_idx]
                    if y < neighbor.
                        shape[0] and x <
                         neighbor.shape
                        [1]:
                        if neighbor[y, x
                            ] >=
                            current_val:
                            is_max =
                                False
                            break
            if is_max:
                sigma = scale_factor **
                    scale_idx
                keypoints.append(cv2.
                    KeyPoint(x=float(x),
                     y=float(y), size=
                    sigma * 5))  # <--
                    FIXED

    # Compute SIFT descriptors
    sift = cv2.SIFT_create()
    image_for_sift = (image * 255).
        astype(np.uint8) if image.max()
```

```
        <= 1.0 else image.astype(np.
        uint8)
    keypoints, descriptors = sift.
        compute(image_for_sift,
        keypoints)


    return keypoints, descriptors


# Optimized Harris Laplace



def compute_DoG(image, sigma1, sigma2):
    """
    Computes normalized Difference-of-
        Gaussians (DoG) approximation of
         LoG.
    Now includes scale normalization for
         accurate LoG approximation.
    """
    blur1 = cv2.GaussianBlur(image, (0,
        0), sigmaX=sigma1)
    blur2 = cv2.GaussianBlur(image, (0,
        0), sigmaX=sigma2)

    # Normalization factor: 1/((k-1)*
              ), where k =      /
    k = sigma2 / sigma1
    normalization = 1 / ((k - 1) *
        sigma1 ** 2)

    return (blur2 - blur1) *
        normalization  # Normalized DoG
            LoG

def harris_laplace_optimized(image,
                             harris_scale_fac
                                 =1.4,
                             dog_scale_factor
                                 =1.2,
                             num_scales
                                 =6,
                             harris_thresh
                                 =0.005,
                             dog_thresh
                                 =0.0001)
                                 :
    """
    Optimized Harris-Laplace with:
    - Precomputed Harris/DoG pyramids
    - DoG for fast scale-space extrema
    - Adaptive non-max suppression
    """
    if image.dtype != np.float32:
        image = np.float32(image)


    # 1. Precompute Harris and DoG
        pyramids

    scales = [harris_scale_factor ** i
        for i in range(num_scales)]
    harris_pyramid = []
    dog_pyramid = []

    for sigma in scales:
        # Harris at current scale
        blurred = cv2.GaussianBlur(image
            , (0, 0), sigmaX=sigma)
        harris = cv2.cornerHarris(
            blurred, blockSize=2, ksize
            =3, k=0.04)
        harris_pyramid.append(harris)

        # DoG for adjacent scales (finer
             and coarser)
        if sigma > scales[0]:  # Finer
            scale exists
            dog_finer = compute_DoG(
                image, sigma/
                dog_scale_factor, sigma)
        else:
            dog_finer = None

        dog_current = compute_DoG(image,
             sigma, sigma*
            dog_scale_factor)
        dog_pyramid.append((dog_finer,
            dog_current))


    # 2. Detect scale-space extrema

    keypoints = []
    for scale_idx, sigma in enumerate(
        scales):
        harris = harris_pyramid[
            scale_idx]
        dog_finer, dog_current =
            dog_pyramid[scale_idx]

        # Get candidate points (Harris
            thresholding + 3x3 NMS)
        candidates = np.argwhere(harris
            > harris_thresh * harris.max
            ())
        for y, x in candidates:
            # Spatial NMS: Check 3x3
                neighborhood
            patch = harris[max(0,y-1):y
                +2, max(0,x-1):x+2]
            if harris[y, x] != np.max(
                patch):
                continue

            # DoG scale-space extremum
                verification
            val_current = dog_current[y,
                 x]
            val_finer = dog_finer[y, x]
                if dog_finer is not None
                 else -np.inf
            val_coarser = dog_pyramid[
                scale_idx+1][1][y, x] if
                 scale_idx < num_scales
                -1 else -np.inf

            if (val_current > val_finer
                and
                val_current >
                    val_coarser and
                abs(val_current) >
                    dog_thresh):
                keypoints.append(cv2.
                    KeyPoint(
                    x=float(x),
                    y=float(y),
                    size=sigma * 5,  #
```

```
                        size = 5
                    response=val_current
                        # Store DoG
                        strength
                ))


    # 3. Refine and return keypoints


    # Compute SIFT descriptors
    sift = cv2.SIFT_create()
    image_for_sift = (image * 255).
        astype(np.uint8) if image.max()
        <= 1.0 else image.astype(np.
        uint8)
    keypoints, descriptors = sift.
        compute(image_for_sift,
        keypoints)

    return keypoints, descriptors


# Combined Feature Extractor SIFT+Harris

def combine_features(image,
    harris_thresh=0.01,
    sift_contrast_thresh=0.0005,
    sift_edge_thresh=50):
    """
    Combine SIFT and Harris features,
        removing duplicates.

    Parameters:
        image: Input grayscale image
        sift_contrast_thresh: SIFT
            contrast threshold (lower=
            more features)
        sift_edge_thresh: SIFT edge
            threshold (higher=more
            features)
        harris_thresh: Harris threshold
            (lower=more features)

    Returns:
        combined_kp: Combined keypoints
            without duplicates
        combined_desc: Corresponding
            descriptors
    """
    # Extract features from both methods
    sift_kp, sift_desc =
        extract_dog_features(image,
        sift_contrast_thresh,
        sift_edge_thresh)
    harris_kp, harris_desc =
        multi_scale_harris(image,
        harris_thresh)

    # If either set is empty, return the
         other
    if not sift_kp:
        return harris_kp, harris_desc
    if not harris_kp:
        return sift_kp, sift_desc

    # Convert keypoints to numpy arrays
        for comparison
    sift_points = np.array([(kp.pt[0],
        kp.pt[1]) for kp in sift_kp])
    harris_points = np.array([(kp.pt[0],
         kp.pt[1]) for kp in harris_kp])

    # Find duplicates (points within 3
        pixels of each other)
    duplicate_mask = np.ones(len(
        harris_points), dtype=bool)
    radius = 3.0  # pixels

    for i, h_pt in enumerate(
        harris_points):
        distances = np.sqrt(np.sum((
            sift_points - h_pt)**2, axis
            =1))
        if np.any(distances < radius):
            duplicate_mask[i] = False

    # Filter out duplicate Harris points
    filtered_harris_kp = [kp for i, kp
        in enumerate(harris_kp) if
        duplicate_mask[i]]
    filtered_harris_desc = harris_desc[
        duplicate_mask]

    # Combine the features
    combined_kp = list(sift_kp) +
        filtered_harris_kp
    combined_desc = np.vstack((sift_desc
        , filtered_harris_desc))

    return combined_kp, combined_desc

# Function to train and evaluate

import time

def train_and_evaluate(images, labels,
    extractor_func, title, n_clusters,
    harris_thresh, contrastThreshold,
    edgeThreshold):
    # First pass: Extract descriptors
        and keep them for both
        clustering and histogram
        creation
    start_time = time.time()

    all_descriptors = []
    all_kp_descriptors = []  # Store (kp
        , desc) pairs for each image
    img_kp_counts = []
    total_kp_count = 0

    for img in images:
        if extractor_func ==
            multi_scale_harris:
            kp, desc =
                multi_scale_harris(img,
                harris_thresh=
                harris_thresh)
        elif extractor_func ==
            harris_laplace_optimized:
            kp, desc =
                harris_laplace_optimized
                (img, harris_thresh=
                harris_thresh)
        elif extractor_func ==
            combine_features:
            kp, desc = combine_features(
                img, harris_thresh=
                harris_thresh,
```

```
                sift_contrast_thresh=
                    contrastThreshold,
                sift_edge_thresh=
                    edgeThreshold)
        else:
            kp, desc =
                extract_dog_features(img
                , contrastThreshold,
                edgeThreshold)

        if desc is not None:
            all_descriptors.append(desc)
            all_kp_descriptors.append((
                kp, desc))
            img_kp_counts.append(len(kp)
                )
            total_kp_count += len(desc)
        else:
            all_kp_descriptors.append((
                None, None))
            img_kp_counts.append(0)


all_descriptors = np.vstack(
    all_descriptors)
feature_extraction_time = time.time
    () - start_time

print(f"{title} Feature Extraction
    Time: {feature_extraction_time
    :.2f} seconds")
print(f"Total Keypoints Detected: {
    total_kp_count}")

# KMeans Clustering
kmeans_start = time.time()
kmeans = MiniBatchKMeans(n_clusters,
     batch_size=5000).fit(
    all_descriptors)
print(f"Clustering Time: {time.time
    () - kmeans_start:.2f} seconds")

# Compute Histograms using stored
    descriptors
hist_start = time.time()
histograms = []
for kp, desc in all_kp_descriptors:
    if desc is not None:
        words = kmeans.predict(desc)
        hist, _ = np.histogram(words
            , bins=n_clusters, range
            =(0, n_clusters))
    else:
        hist = np.zeros(n_clusters)
    histograms.append(hist)
print(f"Histogram Creation Time: {
    time.time() - hist_start:.2f}
    seconds")

# Prepare data
X = np.array(histograms)
y = np.array(labels)

# Train the SVM Classifier
clf = make_pipeline(StandardScaler()
    , SVC())
split_point = 4000
X_train = X[:split_point]
y_train = y[:split_point]
X_test = X[split_point:]
y_test = y[split_point:]

train_start_time = time.time()
clf.fit(X_train, y_train)
training_time = time.time() -
    train_start_time

print(f"{title} Training Time: {
    training_time:.2f} seconds")
acc = clf.score(X_test, y_test)
print(f"Accuracy: {acc * 100:.2f}%")
return acc

# Hyperparameter Search Example for
    Optimal Cluster



# Configuration for each feature
    extraction method
methods_config = {
    'DoG': {
        'extractor':
            extract_dog_features,
        'fixed_params': {
            'harris_thresh': 0,        #
                Not used for DoG
            'contrastThreshold': 0.0005,
                 # Fixed contrast
                threshold
            'edgeThreshold': 30        #
                Fixed edge threshold
        }
    },
    'MultiScaleHarris': {
        'extractor': multi_scale_harris,
        'fixed_params': {
            'harris_thresh': 0.01,     #
                Fixed Harris threshold
            'contrastThreshold': 0,    #
                Not used
            'edgeThreshold': 0         #
                Not used
        }
    },
    'HarrisLaplace': {
        'extractor':
            harris_laplace_optimized,
        'fixed_params': {
            'harris_thresh': 0.005,
                # Fixed Harris threshold
            'contrastThreshold': 0,    #
                Not used
            'edgeThreshold': 0         #
                Not used
        }
    },
    'Combined': {
        'extractor': combine_features,
        'fixed_params': {
            'harris_thresh': 0.01,     #
                Fixed Harris threshold
            'contrastThreshold': 0.0005,
                 # Fixed SIFT contrast
                threshold
            'edgeThreshold': 30        #
                Fixed SIFT edge
                threshold
        }
    }
```

```
}

# Cluster values to test
n_clusters_list = [10, 20, 50, 100]

# Store all results
all_results = []

# Test each method
for method_name, config in
    methods_config.items():
    print(f"\n=== Testing {method_name}
        ===")
    method_results = []

    for n_clusters in tqdm(
        n_clusters_list, desc=f"{
        method_name}"):
        acc = train_and_evaluate(
            images,
            labels,
            config['extractor'],
            f'{method_name} (k={
                n_clusters})',
            n_clusters=n_clusters,
            harris_thresh=config['
                fixed_params']['
                harris_thresh'],
            contrastThreshold=config['
                fixed_params']['
                contrastThreshold'],
            edgeThreshold=config['
                fixed_params']['
                edgeThreshold']
        )

        method_results.append({
            'method': method_name,
            'n_clusters': n_clusters,
            'accuracy': acc,
            **config['fixed_params']  #
                Include the fixed params
        })

    all_results.extend(method_results)

# Convert to DataFrame
results_df = pd.DataFrame(all_results)

# Plot results
plt.figure(figsize=(12, 8))
sns.set_style("whitegrid")
g = sns.lineplot(
    data=results_df,
    x='n_clusters',
    y='accuracy',
    hue='method',
    style='method',
    markers=True,
    dashes=False,
    markersize=10
)
plt.title('Feature Extraction Method
    Comparison\n(Varying Cluster Counts
    with Fixed Thresholds)')
plt.xlabel('Number of Clusters')
plt.ylabel('Accuracy')
plt.legend(title='Method',
    bbox_to_anchor=(1.05, 1), loc='upper
     left')
plt.grid(True)
plt.tight_layout()
plt.savefig('
    all_methods_cluster_comparison.png',
     bbox_inches='tight')
plt.show()

# Save results
results_df.to_csv('
    all_methods_cluster_results.csv',
    index=False)

# Find optimal cluster count for each
    method
optimal_clusters = results_df.loc[
    results_df.groupby('method')['
    accuracy'].idxmax()]
print("\nOptimal cluster counts per
    method:")
print(optimal_clusters[['method', '
    n_clusters', 'accuracy']])

# Sample Dataset Import

# Transform to resize and convert to
    tensor
transform = transforms.Compose([
    transforms.Resize((128, 128)),
    transforms.ToTensor()
])
dataset = FashionMNIST(root='./data',
    train=True, download=True, transform
    =transform)
data_loader = DataLoader(dataset,
    batch_size=1, shuffle=False)

# Extract 5000 images and labels
images = []
labels = []
for i, (img, label) in enumerate(
    data_loader):
    if i >= 5000:
        break
    gray = (img[0].numpy().squeeze() *
        255).astype(np.uint8)
    images.append(gray)
    labels.append(label.item())

# Function to show keypoints
def show_keypoints(image, keypoints,
    title="Keypoints"):
    image_with_kp = cv2.drawKeypoints(
        image, keypoints, None, flags=
        cv2.
        DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS
        )
    plt.figure(figsize=(4, 4))
    plt.imshow(image_with_kp, cmap='gray
        ')
    plt.title(f"{title} ({len(keypoints)
        } keypoints)")
    plt.axis("off")
    plt.show()

#
```