Research on Object Recognition with CNN and Local Features and Computer Vision Algorithm

Kunwei Song Student ID: 11537231

1 Introduction

Cognitive robotics and computer vision have long been pivotal subfields within computer science, with their frontier research representing the most advanced levels of the discipline. In the study of cognitive robotics and computer vision, the accurate recognition of everyday objects forms the basis for both human–machine interaction and autonomous decision-making. From the early development of traditional computer vision algorithms that extract local features[1] to the rapid rise of deep learning techniques, convolutional neural networks (CNNs) have, thanks to their powerful ability to represent image spatial structures, become the mainstream approach for visual classification tasks[2]. Accordingly, learning these two distinct detection methods is highly valuable for further consolidating our foundation in computer vision and for deepening our understanding of its essence.

Therefore, this report will investigate both traditional computer vision algorithms and convolutional neural networks. Through experimental comparisons and analysis of results, it will conduct an in-depth comparison and discussion of the two approaches and provide an outlook on the frontier technologies in this field.

2 Dataset Selection

In order to investigate the performance capabilities and differences of the two methods under varied scenarios, this report selects the iCubWorld1 dataset[3]—a robot-vision benchmark composed of images of various everyday objects—as the primary experimental subject. The training data of iCubWorld1.0 were collected in Human Mode (objects demonstrated by a human handler), with 3 object instances per category, 200 images per instance, and bounding boxes of size 160 × 160 pixels. The testing stage is divided into four complementary subsets—Background, Categorization, Demonstrator, and Robot—each designed to evaluate a different aspect of robustness: the Background subset tests performance on background-only or segmented inputs; the Categorization subset assesses recognition of novel object instances; the Demonstrator subset measures generalization to new demonstrators; and the Robot subset evaluates detection when objects are held by the robot.

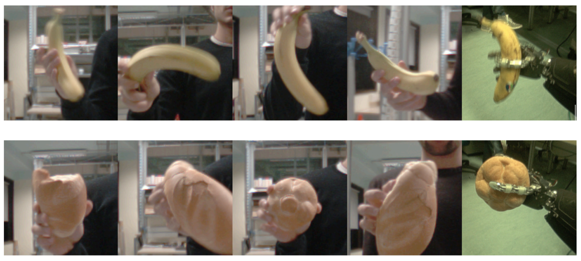

Figure1. Train and test samples of iCubWorld1.0 (From left to right: Train, Background, Categorization, Demonstrator, and Robot)

Additionally, because iCubWorld1.0 currently lacks a widely recognized baseline and comprises numerous test subsets, we also employ CIFAR-10 to assess traditional computer-vision models under conditions of small image size and a single, standard test set. CIFAR-10 is a small colour-image dataset with ten classes that is widely used for image-classification benchmarking[4].

3 Using CNN Methods for Object Recognition

3.1 CNN Methods Design

In this study, we apply convolutional neural networks (CNNs) to the iCubWorld1 dataset to recognize objects from multiple everyday categories. First, the iCubWorld1 dataset is split into training and validation sets at an 80%/20% ratio. The training set is used for model parameter learning, while the validation set is employed for hyperparameter tuning and early stopping. All input images are resized to 224 × 224 pixels, and during training random horizontal flips, random crops, and colour jittering are applied to improve robustness to illumination and viewpoint variations; during validation, only centre cropping and normalization are performed. Normalization uses the ImageNet channel means and standard deviations (mean = [0.485, 0.456, 0.406], std = [0.229, 0.224, 0.225]) to ensure effective transfer of pretrained weights.

Our starting point is a custom "SimpleCNN" architecture, inspired by prior work (Çalik & Demirci, 2018)**Error! Reference source not found.**. As shown in Figure 1, the network comprises three sequential convolutional blocks followed by two fully connected layers. The first, second, and third convolutional blocks use 32, 64, and 128 filters of size 3 × 3, respectively, all with stride 1 and padding 1 to ensure full spatial coverage while controlling parameter count. Each convolution is followed by batch normalization and ReLU activation, then by 2 × 2 max pooling to halve the feature-map dimensions—retaining key information while reducing computation and mitigating overfitting. After the third pooling, the resulting 128 × 28 × 28 feature maps are flattened into a 100 352-dimensional vector and fed to a fully connected layer of 512 neurons with ReLU activation and 50% dropout, encouraging the model to learn more robust representations. Finally, a linear layer mapping to 10 outputs, followed by softmax, yields the class-prediction probabilities.

We train this model for 200 epochs using the Adam optimizer (initial learning rate = 0.001) with a batch size of 128 and categorical cross-entropy loss. Standard data augmentation—random horizontal flip, resized crop, and colour jitter—is applied, and the ImageNet normalization is used throughout. A stratified grid search explores optimizer choice (Adam vs. SGD), learning rate (0.001, 0.01), and batch size (64, 128, 256), with validation accuracy and loss monitored at each epoch; the learning rate is decayed on plateau. The best hyperparameter combination yields an 81.5% validation accuracy.

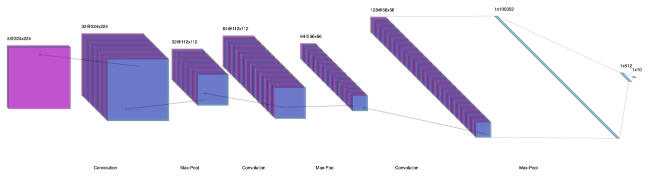

Figure 2. the Structure of Original CNN

Given that iCubWorld1 comprises only 6,000 training images while the original model has tens of millions of parameters—making the network relatively sparse—we introduce an enhanced architecture to improve generalization and reduce overfitting. After the third convolutional block, we replace the flatten-and-dense head with a global-pooling–based design: an AdaptiveAvgPool2d(output_size=(1,1)) layer collapses each of the 128 feature maps into a single scalar, producing a 128-dimensional vector[5]. This vector feeds into a lighter two-layer classifier—first a linear 128→256 layer with ReLU and 50% dropout, then a linear 256→10 layer with softmax. All other training settings (data augmentation, normalization, optimizer, schedule) remain the same to isolate the effect of this architectural change. Under these conditions, the improved model achieves an 87.5% validation accuracy—6 points higher than the original design.

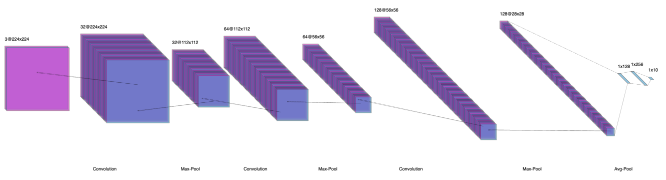

Figure3. the structure of Improved CNN

3.2 Analyse of CNN Methods Results

Throughout the entire experimental process, we experimented with a variety of strategies to identify the optimal hyperparameters. For the chosen network architecture, since iCubWorld1.0 is not a particularly large dataset and contains only ten classes, we determined that a simple deep neural network would suffice. Given the considerable advantages that convolutional neural networks hold over traditional feedforward neural networks in image processing[1], we adopted our current CNN and began manually evaluating its performance on this database. Ultimately, using an SGD optimizer with a learning rate of 0.001 and weight decay of 0.0001, configured for 60 epochs with early stopping at the 45th epoch, the model achieved 97.12% accuracy on the validation set. This result preliminarily confirmed the network's effectiveness.

We then conducted a systematic grid search over hyperparameters—learning rate, optimizer type, batch size, and learning rate decay. After 200 epochs, the optimal configuration was found to be: SGD optimizer, learning rate 0.0005, batch size 32, and learning rate decay 0.0001, yielding a peak validation accuracy of 98.08%. With these parameters fixed, we retrained the network for 30 epochs on the entire dataset, ultimately achieving 98.81% validation accuracy. However, evaluation on the test subsets revealed substantially lower performance: Background: 51.500%, Demonstrator: 78.960%, Categorization: 51.374%, Robot: 7.980%.

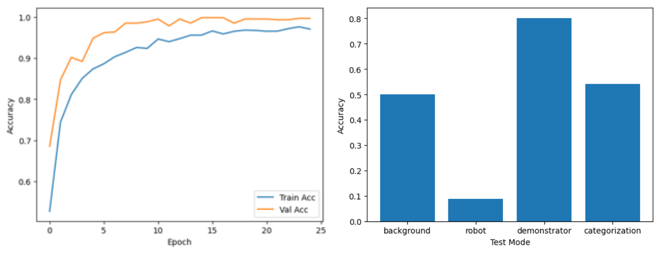

Figure 4. Initial CNN training curves and results

The results clearly indicate that none of the test subsets reached the validation-set accuracy. The Demonstrator subset performed best (78%), likely because its data most closely resemble the original training distribution, demonstrating that even with a new demonstrator, the model successfully transfers and exhibits robustness to occlusions by the demonstrator's hand. Next, the Background subset—composed of images from the training set on which the classifier had achieved 99% accuracy—was intended to verify whether the model relied solely on background cues. Its 50% accuracy, although low, suggests that the model did not simply exploit background information but learned object-specific features. The Categorization subset, containing unseen object instances, saw performance degradation, indicating that the model lacks strong generalization ability. Finally, the Robot subset yielded only 7%, revealing that under large viewpoint changes, occlusions, and interference from the robot arm, the learned features were insufficient for accurate classification.

In the original model, flattening the 128 × 28 × 28 feature maps produced a 100 352-dimensional vector, which was then passed through a fully connected layer with millions of parameters—resulting in an excessively large parameter count. Inspired by architectures such as ResNet, MobileNet, and EfficientNet, we replaced this flatten-and-dense classification head with a global pooling–based design. Specifically, following the final convolutional block, we inserted an AdaptiveAvgPool2d(output_size=(1,1)) layer to collapse each of the 128 channel feature maps into a single scalar, producing a 128-dimensional vector. This vector feeds into a lightweight two-layer classifier (linear 128→256 with ReLU and 50% dropout, followed by linear 256→10 and softmax). Although this discards spatial information, it sufficiently retains high-level channel features for our classification task, reducing parameter count and mitigating overfitting. We repeated the same evaluation and hyperparameter search procedures. The optimal settings are summarized in Table 1:

Table 1. Record of Optimal Parameters

Eopch	**lr**	**optimizer**	**batch_size**	**weight_decay**	**best_val_acc**
13	**0.0005**	**Adam**	**32**	**0.0000**	**0.9267**
15	0.0005	Adam	64	0.0000	0.9118
6	0.0010	Adam	64	0.0001	0.9101
7	0.0010	Adam	64	0.0000	0.910

The Final result in shown below:

Table 2. Final Result of CNN Method

Model	Accuracy (%)			
	Background	Robot	Demonstrator	Categorization
Updated CNN	53.000	16.459	88.506	58.021
Original CNN	51.500	7.980	80.410	51.374
ResNet 18	**64.000**	**25.062**	**93.503**	**81.159**

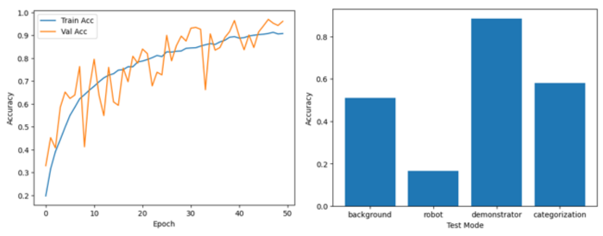

Figure 5. Training curves and results for the improved CNN

Analysis of the improved model shows a similar distribution of test-set accuracies, indicating persistent context dependency and robustness limitations. However, accuracies for all subsets improved; notably, the Robot subset accuracy increased to 16%, demonstrating that the optimized model learned more intrinsic object features. We attribute these improvements to three factors:

(1) parameter reduction—the new classification head contains approximately 30,000 parameters versus roughly 51 million previously, greatly suppressing overfitting;

(2) enhanced spatial invariance—global average pooling encourages the network to focus on channel activations rather than their exact spatial positions, bolstering robustness to object translation and background variation[6];

(3) more stable training dynamics—smaller dense layers facilitate smoother gradient flow and faster convergence. In summary, integrating global pooling into a lightweight CNN achieves both efficiency and high accuracy in classifying robot-vision data.

To further address the remaining generalization shortcomings, future experiments may incorporate background-removed object images into the training set or include a small fraction of test-set images in training with appropriate data augmentation to enhance generalization. These aspects will be explored in future work.

4 Using Local Features and CV Algorithms for Object Recognition

4.1 Local Feature and CV Algorithm Method Design

In this section, we perform image recognition using local features and traditional computer-vision algorithms. To begin, the two datasets—iCubWorld1.0 and CIFAR-10—are first converted to grayscale to facilitate more reliable keypoint detection and to increase robustness.

We then detect interest points using the Harris[7] and Difference-of-Gaussian (DoG) methods introduced in class, selecting the detector best suited to each dataset. Next, we compute SIFT descriptors at each keypoint[8], exploring descriptor parameters to optimize performance. Once descriptors are obtained, we build a visual-bag-of-words representation: we choose an appropriate vocabulary size by testing several options, then cluster all SIFT descriptors with a trained K-means model to produce, for each image, a histogram of visual-word occurrences. These histograms are L2-normalized and fed into a classifier for final prediction. Throughout this process, we experimented with multiple combinations of detectors, descriptors, classifiers, and their hyperparameters, using a hybrid of manual tuning and grid search. Initial manual tests (Table 1) identified Harris as the detector, SIFT as the descriptor, SVM as the classifier, and a vocabulary size of 400 as the best configuration.

Table 3. Hyperparameter Grid Search

Descriptors	K	Classifier	Accuracy (%)			
			Background	Categorization	Demonstrator	Robot
Harris	***400***	***SVM***	99	**29.39**	22.89	12.47
Harris	*500*	*SVM*	100	28.39	22.29	**13.09**
DoG	*400*	*SVM*	99	21.74	**28.89**	10.85
DoG	*800*	*SVM*	100	21.79	28.09	10.22
Harris	*400*	*KNN*	100	21.29	18.14	11.35
DoG	*800*	*KNN*	87	21.74	17.24	11.72

We then performed a 50-iteration grid search to fine-tune the SVM parameters, yielding an RBF kernel with C = 1 and γ = 0.1. Under this configuration, the Categorization subset achieved 28.04% accuracy.

To further improve results, we incorporated techniques recommended in the course literature:

(1) **PCA Dimensionality Reduction & RootSIFT:** applying PCA to reduce SIFT descriptors from 128 to 64 or 32 dimensions before clustering, to remove noise and accelerate K-means[9].

(2) **MiniBatchKMeans(init= 'k-means++') with Increased n_init:** reducing sensitivity to initial centroids by running multiple initializations (e.g., n_init = 10).

(3) **TF–IDF Weighting + L2 Normalization:** replacing raw frequency histograms with TF–IDF–weighted histograms followed by L2 normalization, to suppress common "visual words" and emphasize discriminative ones[10].

(4) **Stop-Word List:** filtering out clusters whose visual words appear in more than 80% or less than 1% of the images, to remove noisy or uninformative words[10].

Method 1 (PCA/RootSIFT) reduced dimensionality but degraded accuracy and increased complexity, so it was discarded. Method 2 (MiniBatchKMeans) did not improve accuracy but produced more stable clustering. Method 3 (TF–IDF + L2) yielded a notable accuracy gain. Method 4 (stop-word filtering) had negligible effect on overall accuracy. The post-improvement results are summarized in Table 4.

Table 4. Comparison of Two Normalization Methods

Methods	Accuracy (%)			
	Background	**categorization**	**demonstrator**	**robot**
SVM(Tf–Idf +L2)	93.0	**31.08**	21.59	11.47
SVM(L2)	99.0	29.39	22.89	12.47

Finally, to assess traditional CV methods on a single, standardized test set, we trained on CIFAR-10 using the same manual plus automated tuning approach. After 50 iterations, the optimal settings were: DoG detector, SIFT descriptor, vocabulary size = 500, and SVM classifier (RBF kernel, C = 1.0, γ = 0.1), achieving 29.33% final accuracy.

4.2 Local Feature and CV Algorithm Analysis

The multiple modes of the iCubWorld1.0 test set allow us to thoroughly analyse our experimental outcomes. Table 5 summarizes the key results:

Table 5. Summary of Main Experimental Results

Methods	Accuracy (%)				
	Background	**Categorization**	**Demonstrator**	**Robot**	**Overall**
SVM (L2)	99.0	29.39	22.89	12.47	–
SVM (tf–idf + L2)	93.0	**31.08**	21.59	11.47	–
PCA→64 + RootSIFT + SVM (L2)	93.0	27.65	15.72	7.93	–
MiniBatchKMeans (init++, n_init=10)	99.10	29.51	23.02	12.51	–
StopWords + SVM (tf–idf)	**100**	28.04	**25.24**	**13.84**	–
KNN(L2)	100	21.29	18.14	11.35	
CIFAR-10 (DoG+SIFT, K=500, SVM)	–	–	–	–	**29.33**

From the table, we observe that the Background subset achieves very high scores across all methods, indicating that the models rely heavily on background textures rather than the objects themselves—and underscoring that BoVW, which ignores geometric information, is easily influenced by background "noise"[11].

For the Categorization subset, the introduction of TF–IDF weighting with L2 normalization confirms the benefit of suppressing common "visual words" and emphasizing rarer ones—improving accuracy by 1.6%. In the Demonstrator subset, changes in the demonstrator introduce hand occlusions, pose variations, and subtle background shifts that BoVW cannot capture. Here, the stop-word filtering method has the most pronounced effect, showing that appropriately removing noisy background words while retaining mid-frequency words helps the model recognize object-hand relationship patterns. Conversely, in the Robot subset, all detection methods perform very poorly (≈ 7–16%). Under the robot mode's large viewpoint changes, occlusions, and oversized bounding boxes, local feature matching fails, and BoVW's complete lack of spatial structure information causes near-total breakdown of detection.

Comparing classifiers, K-NN severely underfits the high-dimensional BoVW features and generalizes far worse than nonlinear[12].

On CIFAR-10, ROC curves and the confusion matrix show moderate AUC scores (0.7–0.8) and a clear diagonal, yet when the decision threshold is fixed at 0.5, accuracy remains just 20–40%. The model frequently confuses similar classes (e.g., cat vs. dog, automobile vs. truck), further demonstrating its failure to capture sufficient spatial information and distinctive category features.

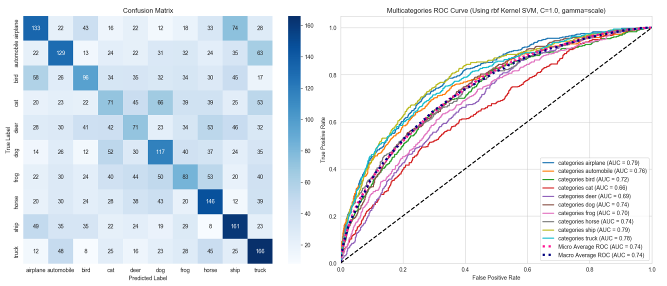

Figure 6. CIFAR-10 performance visualized under traditional CV methods

In summary, our experiments demonstrate that traditional CV-based models exhibit excessive background dependence, fail to learn intrinsic object features, and lack both spatial information and generalization ability—resulting in poor classification performance. To address these shortcomings in future work, we propose incorporating depth information fusion or augmenting the training set with background-removed object images, and exploring Spatial Pyramid Matching (SPM) to reintroduce spatial structure into the BoVW framework.

5 Interpretation and Comparison of Object Recognition Results for the Two Methods

Our experiments reveal a pronounced performance gap between the two approaches. The CNN method outperforms the traditional CV pipeline by roughly 50% on the critical Categorization and Demonstrator subsets.

Table 6. Performance Comparison of Different Methods

Methods	Accuracy(%)			
	Background	**Categorization**	**Demonstrator**	**Robot**
BoVW+SVM(tf–idf+L2)	93.0	31.08	21.59	11.47
BoVW+SVM(Stop-Words)	**100.0**	28.04	25.24	13.84
SimpleCNN（Flatten）	51.5	51.37	80.41	7.98
SimpleCNN（GAP）	53.0	58.02	88.51	16.46
ResNet-18	64.0	**81.16**	**93.50**	**25.06**

The traditional pipeline—detecting keypoints, extracting local descriptors, and classifying via a Bag-of-Visual-Words histogram—vectorizes each image based on dataset-specific features. When the dataset contains highly discriminative keypoints, the model performs well; otherwise, its reliance on those local features becomes a liability. By focusing on corners and high-contrast regions, SIFT struggles to capture large, smooth textured areas, losing the ability to learn and predict those features. Moreover, operating on grayscale images emphasizes edges and reduces background noise but sacrifices important colour information. Finally, converting to a BoW histogram discards all spatial context, leaving only word-occurrence frequencies; the model never truly learns intrinsic object representations, so its accuracy plummets under background interference or when generalization is required[13].

In contrast, a CNN ingests the full image end-to-end and applies hierarchical convolutions and pooling to extract both local edge details and global spatial patterns. Its superior background robustness is evident in the Background subset: the CNN learns object-centric features and suppresses spurious background cues, demonstrating that end-to-end feature learning yields more stable object recognition. These learned features are then passed through fully connected layers for classification—an opaque "black box" process that, unlike the traditional pipeline, sacrifices interpretability for predictive power.

From a training standpoint, CNNs require more epochs and larger datasets to acquire rich feature representations, trading additional computational cost and memory for higher accuracy. Therefore, in scenarios with tight compute constraints—such as edge devices—the traditional CV approach can still be advantageous. However, when a lightweight CNN is feasible, it remains the superior choice.

In summary, for robotic vision applications demanding robust recognition under varied real-world conditions, convolutional neural networks should be the primary approach; but in well-controlled environments with limited computational resources, the BoVW-based pipeline offers a simple and interpretable fallback.

6 State-of-the-art in computer vision for robotics

In recent years, the intersection of robotics and computer vision driven by deep learning has made significant strides, giving rise to numerous frontier research directions. In this section, we explore some of these leading-edge techniques.

6.1 Vision Transformers and Architectural Innovations

Within the vanguard of computer-vision research, the emergence of the Vision Transformer (ViT) marks a paradigm shift. Traditional convolutional neural networks (CNNs) have dominated image tasks for years, yet the Transformer's success in natural-language processing inspired its application to vision. ViT[14] demonstrated that a pure self-attention–based model—trained on sufficiently large datasets—can match or even surpass CNNs in image-classification accuracy. By dividing an image into a sequence of patches and applying global self-attention, ViT captures long-range dependencies across the entire image. This approach both simplifies complex convolutional designs and endows the model with a truly global receptive field. Owing to its modularity and scalability, ViT can be deepened and scaled to massive data volumes, opening new avenues for precision gains in classification, detection, and segmentation tasks.

During 2024, much architectural innovation has focused on boosting ViT's efficiency and tailoring it to specific tasks. A key direction is hierarchical and sparse attention. Early ViT models incurred heavy computational costs as self-attention scales quadratically with token count. At CVPR 2024, Zhang et al. introduced the Less-Attention Vision Transformer (LaViT), which further reduces redundant attention computations[15]. They observe that full self-attention at every layer not only burdens computation but also leads to "attention saturation," where deeper layers merely re-emphasize similar patterns. LaViT divides the Transformer into stages: only the first few layers in each stage compute full attention maps; subsequent layers approximate and reuse these maps via lightweight linear transformations. This design ensures that each stage focuses on necessary attention patterns, cutting compute and avoiding repeated information saturation. LaViT relies almost exclusively on basic matrix multiplications, making it both simple and highly efficient for existing deep-learning frameworks. On ImageNet classification, COCO detection, and ADE20K segmentation, LaViT achieves performance on par with—or superior to—standard ViT, but with substantially lower compute, proving that "less attention" need not sacrifice accuracy.

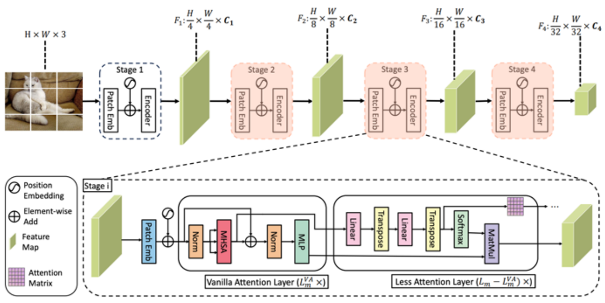
$H \times W \times 3$
$F_1: \frac{H}{4} \times \frac{W}{4} \times C_1$
$F_2: \frac{H}{8} \times \frac{W}{8} \times C_2$
$F_3: \frac{H}{16} \times \frac{W}{16} \times C_3$
$F_4: \frac{H}{32} \times \frac{W}{32} \times C_4$
Stage 1
Stage 2
Stage 3
Stage 4
Patch Emb
Encoder
Position Embedding
Element-wise Add
Feature Map
Attention Matrix
Stage i
Norm
MHSA
MLP
Linear
Transpose
Softmax
MatMul
Vanilla Attention Layer (L_m^{VA} ×)
Less Attention Layer ($L_m - L_m^{VA}$) ×)

Figure 7. The architecture of our Less-Attention Vision Transformer (LaViT)[14].

Beyond backbone optimizations, many innovations address specialized tasks with Vision Transformers. For instance, in semi-supervised semantic segmentation, vanilla ViT may overlook fine-grained details. Addressing this, Hu et al. presented S4Former at CVPR 2024—a Transformer framework tailored for semi-supervised segmentation[16]. S4Former embeds ViT as its backbone within a teacher–student paradigm and introduces three key innovations: PatchShuffle perturbations to enhance uncertainty, Patch-Adaptive Self-Attention (PASA) to modulate attention at fine scales, and a Negative Class Ranking (NCR) loss to constrain unannotated classes. Experiments on Pascal VOC 2012, COCO, and Cityscapes in semi-supervised settings show that S4Former outperforms prior methods by an average of 4.9 mIoU, setting new state-of-the-art results. Crucially, it retains the Transformer's simplicity and scalability, integrating seamlessly with existing segmentation pipelines. This underscores that task-driven customizations of ViT can fully leverage its strengths in specialized scenarios.

Overall, Vision Transformer architecture research is thriving: from general–purpose improvements for efficiency and scale (e.g., LaViT) to task-specific designs (e.g., S4Former). Together, these innovations are steering Transformers toward maturity and broader adoption in vision.

When compared to previous methods, these advances have propelled Vision Transformers and their variants to outperform traditional CNNs across many vision tasks. In image classification, ViT now matches or exceeds CNN performance when pretrained on large data. In object detection and instance segmentation, hierarchical Transformers (e.g., Swin Transformer) have become mainstream backbones, boosting detection AP by several points on COCO while maintaining comparable parameter counts and inference speeds to ResNet—delivering both performance and efficiency. Transformers also excel in multi-task learning: their global context modelling enables a single model to output multiple task predictions (e.g., classification + segmentation) from shared representations, surpassing single-task networks that require multi-branch designs. Furthermore, Transformers scale naturally: enormous models like ViT-22B (2.2 billion parameters) continue to push classification performance on JFT-300M data—an almost unimaginable scale for CNNs. In sum, architectural innovations have unlocked the full potential of Vision Transformers, driving a new wave of performance gains and generalization beyond classical convolutional networks.

Nevertheless, these models still face limitations. First, computational and memory costs remain high: even with optimizations like LaViT, self-attention on high-resolution feature maps is expensive, and further reducing complexity without accuracy loss is an open challenge. Second, data dependence persists: lacking the convolutional inductive bias for local patterns, Transformers can overfit when data are scarce. Recent work addresses this by embedding convolutions within Transformers or leveraging synthetic data and stronger augmentations. Third, interpretability demands more study: while CNNs benefit from filter-visualization techniques, attention maps in Transformers do not fully reveal decision rationale, calling for new analysis tools to understand large-scale models' internal representations. Finally, deploying Transformers in specialized domains (e.g., embedded devices, medical imaging) remains difficult due to their hardware demands and challenges in fine-tuning on small datasets. Encouragingly, the deep-learning community is tackling these issues through approaches like knowledge distillation into compact models and self-supervised pretraining to reduce annotation requirements. It is foreseeable that Vision Transformer architectures will become ever more efficient, robust, and transparent, with deep learning continuing to play a central role in this evolution.

6.2 General Robot Foundation Models (GRFM)

General Robot Foundation Models seek to bring the "large-model" paradigm from NLP into robotics by building massive, multimodal models that can serve many tasks and environments. Traditional robotic systems are often tailored to a single task, with limited generalization; by pretraining on vast, diverse datasets, foundation models promise greater flexibility and cross-task transfer. For example, DeepMind's Gato[17] first demonstrated that a single Transformer can, in a unified architecture, play video games, manipulate robotic arms, and process language—all from one set of shared parameters. Inspired by Gato, researchers now explore unified perception-and-decision models: pretrained on massive multimodal data, they are then adapted to new tasks with only a few examples, achieving true "one-model, many-skills" operation. This shift is driven by the need for future robots to operate in open, unpredictable environments and to handle a wide range of tasks without bespoke pipelines.

State-of-the-art GRFMs typically employ deep Transformer backbones to fuse vision, language, and control signals in an end-to-end fashion. One class of approaches directly trains the model to output low-level action sequences: Google Robotics' RT-1 and RT-2 use hundreds of thousands of human demonstration videos paired with textual instructions to teach a Transformer to map images and commands into continuous robot control outputs. Another class introduces an intermediate planning layer for better generality: Google's SayCan and PaLM-E leverage large language models to plan high-level steps, converting perceived scenes and instructions into executable action sequences. By using a language or code "medium" for planning, these methods gain interpretability and easier transfer—e.g., Ahn et al. use a language model to reason about next actions, while Driess et al. embed visual features into PaLM-E's 562B-parameter model to unify vision, language, and control[18]. Architecturally, GRFMs comprise a pretrained vision encoder (often a ViT), a language encoder (for commands and semantics), and an action decoder (to generate control signals). DeepMind's latest Gemini model exemplifies this trend, combining huge multimodal Transformers with millions of tokens of context to understand and generate long-horizon action sequences.

Compared to traditional modular pipelines, GRFMs exhibit far stronger cross-task generalization and data-efficient learning. For instance, Google's One-shot Open Affordance Learning (OOAL)[19] uses a vision-language foundation model to classify object affordances from a single example—using less than 1% of the data required by prior specialized models, yet surpassing their performance. In a complex real-world box-pushing task, fine-tuning Gemini on simulated demonstrations enabled zero-shot transfer to a real robot: it not only recovered most of an expert policy's performance but also showed semantic generalization—handling new objects, multilingual commands, and unseen room layouts. Both RT-2 and Gemini, trained purely in simulation, succeeded zero-shot on real hardware, illustrating the cross-environment and cross-task transfer that is out of reach for traditional reinforcement-learning or behavior-cloning approaches. By contrast, single-task models often demand extensive additional data and tuning whenever the environment or instructions change; foundation models deliver "one-for-all" capability that lifts both accuracy ceilings and application flexibility.

Yet GRFMs also face significant challenges. First, data scarcity remains critical: unlike internet-scale image and text corpora, real robot interaction data are scarce and costly, limiting model scale[17]. To address this, researchers pool data across institutions and leverage large-scale simulated interactions—e.g., the Proc4Gem initiative uses high-fidelity simulators to "convert compute into data" and augment real datasets. Second, true open-world generalization is still brittle: foundation models may behave unpredictably in extreme, out-of-distribution scenarios. Third, high computational and memory demands—billions of parameters and heavy self-attention costs—hinder real-time control. Recent work on knowledge distillation and hierarchical decision architectures aims to compress models and reduce inference latency. Finally, safety and reliability concerns arise when deploying such powerful models on physical robots. In sum, while GRFMs mark a leap forward for robotic perception and control, overcoming data, compute, and safety limitations will be key to their future success.

9 Conclusion

This study compared the traditional BoVW+SVM pipeline with the CNN approach on the iCubWorld1.0 and CIFAR-10 datasets, demonstrating that CNNs possess markedly stronger feature representation and generalization capabilities in complex scenarios (novel instances, new demonstrators, and robot viewpoints), whereas BoVW's performance is constrained by background dependence and the loss of spatial information. By investigating both methods, we have gained deeper insights into object-recognition research in robotics and vision. Recent advances suggest that future gains in robotic-vision accuracy and real-time performance can be achieved through multimodal fusion, Vision Transformer architectures, and General Robot Foundation Models.

Reference:

[1] LeCun, Y., Bottou, L., Bengio, Y. & Haffner, P., 1998. Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11), pp. 2278–2324.

[2] Krizhevsky, A., Sutskever, I. & Hinton, G.E., 2012. ImageNet classification with deep convolutional neural networks. In: Advances in Neural Information Processing Systems (NeurIPS), Lake Tahoe, NV, USA, December 2012, pp.1097–1105.

[3] Fanello, S.R., Ciliberto, C., Santoro, M., Natale, L., Metta, G., Rosasco, L. & Odone, F., 2013. *iCub World: Friendly Robots Help Building Good Vision Data-Sets*. arXiv:1306.3560.

[4] Krizhevsky, A. & Hinton, G., 2009. *Learning Multiple Layers of Features from Tiny Images*. Technical report, University of Toronto.

[5] Lin, M., Chen, Q. & Yan, S., 2013. Network In Network. International Conference on Learning Representations (ICLR).

[6] Tan, M. & Le, Q.V., 2019. EfficientNet: Rethinking model scaling for convolutional neural networks. In: International Conference on Machine Learning (ICML), pp. 6105–6114.

[7] Harris, C. & Stephens, M., 1988. A combined corner and edge detector. Alvey Vision Conference, pp. 147–151.

[8] Lowe, D.G., 2004. Distinctive image features from scale-invariant keypoints. International Journal of Computer Vision, 60(2), pp. 91–110.

[9] Arandjelović, R. & Zisserman, A., 2012. Three things everyone should know to improve object retrieval. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Providence, RI, USA, June 2012. IEEE, pp. 2911–2918.

[10] Sivic, J. & Zisserman, A., 2003. Video Google: A text retrieval approach to object matching in videos. In: Proceedings of the Ninth IEEE International Conference on Computer Vision (ICCV), Nice, France, October 2003. IEEE, pp. 1470–1477.

[11] Csurka, G., Dance, C.R., Fan, L., Willamowski, J. & Bray, C., 2004. Visual categorization with bags of keypoints. In: ECCV Workshop on Statistical Learning in Computer Vision.

[12] Hentschel, C. & Sack, H., 2014. Does one size really fit all? Evaluating classifiers in bag-of-visual-words classification. In: Proceedings of the 14th International Conference on Knowledge Technologies and Data-driven Business (i-KNOW ’14), Graz, Austria, 16–19 September 2014. Article No. 7, pp. 1–8. ACM. doi:10.1145/2637748.2638424

[13] Lazebnik, S., Schmid, C. & Ponce, J., 2006. Beyond bags of features: Spatial Pyramid Matching for Recognizing Natural Scene Categories. In: Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR), New York, NY, USA, June 2006. IEEE, pp. 2169–2178.

[14] Dosovitskiy, A. et al., 2021. An image is worth 16×16 words: Transformers for image recognition at scale. In: International Conference on Learning Representations (ICLR).

[15] Zhang, S., Liu, H., Lin, S. & He, K., 2024. You only need less attention at each stage in vision transformers. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 10 June 2024.

[16] Hu, X., Jiang, L. & Schiele, B., 2024. Training vision transformers for semi-supervised semantic segmentation. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), pp. 4007–4017.

[17] Reed, S. et al., 2022. A generalist agent. arXiv:2205.06175.

[18] Driess, D. et al., 2023. PaLM-E: An embodied multimodal language model. arXiv:2303.03378.

[19] Li, G., Sun, D., Sevilla-Lara, L. & Jampani, V., 2024. One-Shot Open Affordance Learning with Foundation Models. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 16–20 June 2024, pp. 3086–3095.

Appendix A:

All Source Code of this Program in GitHub:

https://github.com/beebatter/Object-Recognition-with-CNN-and-Local-Features-and-Computer-Vision-Algorithm

Main Code Implementation of Traditional CV Method in iCubWorld1.0

```
class Config:
    """Configuration class"""
    DATA_DIR = "Dataset/iCubWorld1/" # Dataset directory
    IMAGE_SIZE = (256, 256)     # image size
    VOCAB_SIZE = 400 # visual vocabulary size(K value)
    RANDOM_STATE = 42 # random state for reproducibility
    IMAGE_EXT = '.ppm' # image extension
    MAX_CORNERS = 200 # Harris
    HARRIS_MIN_DISTANCE = 4 # Harris
    HARRIS_QUALITY_LEVEL = 0.01 # Harris
    HARRIS_THRESH = 1e-4     # Harris
    DETECTOR_TYPE = 'HARRIS_SIFT'  #  SIFT, ‘ORB’, ‘HARRIS_SIFT’, ‘HARRIS_AFFINE’
    FEATURE_LIMIT_PER_IMAGE = 100 #
    VOCAB_SAMPLE_LIMIT = 6000 #

    # Harris parameters
    AFFINE_MAX_ITERATIONS = 20
    AFFINE_RESPONSE_THRESHOLD = 0.01
    AFFINE_EDGE_THRESHOLD = 10
    INIT_KEYPOINT_SIZE = 20

class BaseDataLoader:
    """basedata loader"""
    def __init__(self, data_dir: str = Config.DATA_DIR):
        self.data_dir = data_dir
        # check the dir
        if not os.path.exists(self.data_dir):
             raise FileNotFoundError(f"don't find the dir: {self.data_dir}")
        train_dir = os.path.join(self.data_dir, "train")
        if not os.path.exists(train_dir):
             raise FileNotFoundError(f"Train dir didn't find: {train_dir}")
        self.classes = self._load_classes()

    def _load_classes(self) -> List[str]:
        train_dir = os.path.join(self.data_dir, "train")
        classes = []
        for name in os.listdir(train_dir):
            full = os.path.join(train_dir, name)
            if os.path.isdir(full):
                classes.append(name)
        return sorted(classes)

    def _collect_images(self, directory: str, is_test: bool = False) -> Tuple[List[str],
List[int]]:
        paths, labels = [], []
        if not os.path.exists(directory):
            print(f"Warn: directory didn't exist: {directory}")
            return paths, labels
        class_name = ""
        if is_test:
            # `testset: test/modes/bananas/000000.ppm
             parts = directory.split(os.sep)
             if len(parts) >= 2:
                 class_name = parts[-1]
             else:
                 print(f"Warn: can't sure the categories name: {directory}")
                 return paths, labels

        else:
            # Trainset: train/bananas/bananas_1/000000.ppm

            parts = directory.split(os.sep)
            if len(parts) >= 2:
                 class_name = parts[-2]
            else:
                 print(f"Warn: can't sure the categories name{directory}")
                 return paths, labels


        try:
            class_idx = self.classes.index(class_name)
        except ValueError:

            print(f"Warn: Unknow class '{class_name}' in dir: {directory}")

            return paths, labels

        for fname in os.listdir(directory):
            if fname.endswith(Config.IMAGE_EXT):
                paths.append(os.path.join(directory, fname))
                labels.append(class_idx)

        return paths, labels

    def get_class_names(self) -> List[str]:
        return self.classes

class TrainDataLoader(BaseDataLoader):
    def __init__(self, data_dir: str = Config.DATA_DIR, max_samples_per_class: Optional[int] =
None):
        super().__init__(data_dir)
        self.max_samples_per_class = max_samples_per_class

    def load_data(self) -> Tuple[List[str], List[int]]:
        all_paths, all_labels = [], []
        train_root = os.path.join(self.data_dir, "train")

        print("Start load the train data...")
        for idx, cls in enumerate(tqdm(self.classes, desc="categories")):
            cls_dir = os.path.join(train_root, cls)
            class_paths_collected = []
            class_labels_collected = []
            for instance in os.listdir(cls_dir):
                instance_dir = os.path.join(cls_dir, instance)
                if os.path.isdir(instance_dir):
                    paths, labels = self._collect_images(instance_dir, is_test=False)
                    class_paths_collected.extend(paths)
                    class_labels_collected.extend(labels)

            if self.max_samples_per_class is not None and len(class_paths_collected) >
self.max_samples_per_class:
                indices = np.random.choice(len(class_paths_collected),
self.max_samples_per_class, replace=False)
                sampled_paths = [class_paths_collected[i] for i in indices]
                sampled_labels = [class_labels_collected[i] for i in indices]
                all_paths.extend(sampled_paths)
                all_labels.extend(sampled_labels)
            else:
                all_paths.extend(class_paths_collected)
                all_labels.extend(class_labels_collected)

        print(f"Finished the load.... Total {len(all_paths)} images。")
        return all_paths, all_labels


class TestDataLoader(BaseDataLoader):
    """Load the test data"""
    def __init__(self, data_dir: str = Config.DATA_DIR, max_samples_per_class: Optional[int] =
None):
        super().__init__(data_dir)
        self.max_samples_per_class = max_samples_per_class
        test_dir = os.path.join(data_dir, "test")
        if not os.path.exists(test_dir):
            raise ValueError(f"Test dir do not exist: {test_dir}")

        self.test_types = [d for d in os.listdir(test_dir) if
os.path.isdir(os.path.join(test_dir, d))]
        if not self.test_types:
             print(f"Warn: Can't find the sub dir under {test_dir} ")
        else:
            print(f"find the test type: {self.test_types}")


    def load_data(self) -> Dict[str, Tuple[List[str], List[int]]]:

        test_sets = {}
        test_root = os.path.join(self.data_dir, "test")

        print("Start load the test data...")
        for test_type in tqdm(self.test_types, desc="categories"):
            try:
                paths, labels = self._load_test_type(test_type)
                if paths:
                    test_sets[test_type] = (paths, labels)
                    self._print_test_set_stats(test_type, paths, labels)
                    if len(paths) != len(labels):
                         raise ValueError(f"path and labels do not match: {len(paths)} vs
{len(labels)}")
                else:
```

```python
                    print(f"Warn: Test type {test_type} didn't load any data")

            except Exception as e:
                print(f"erro in load {test_type} : {str(e)}")
                test_sets[test_type] = ([], [])

        print("Finished the loading of data")
        return test_sets

    def _load_test_type(self, test_type: str) -> Tuple[List[str], List[int]]:
        paths, labels = [], []
        test_type_dir = os.path.join(self.data_dir, "test", test_type)

        for cls in self.classes:
            cls_dir = os.path.join(test_type_dir, cls)

            if not os.path.exists(cls_dir):
                continue

            cls_paths, cls_labels = self._collect_images(cls_dir, is_test=True)

            if cls_paths:
                if self.max_samples_per_class is not None:
                    cls_paths, cls_labels = self._sample_data(cls_paths, cls_labels)

                paths.extend(cls_paths)
                labels.extend(cls_labels)

        return paths, labels

    def _sample_data(self, paths: List[str], labels: List[int]) -> Tuple[List[str], List[int]]:
        sample_size = min(len(paths), self.max_samples_per_class)
        if sample_size == 0:
             return [], []
        indices = np.random.choice(len(paths), sample_size, replace=False)
        return [paths[i] for i in indices], [labels[i] for i in indices]

    def _print_test_set_stats(self, test_type: str, paths: List[str], labels: List[int]) -> 
None:
        """print"""
        if not paths:
            return
        print(f"\n--- Testset: {test_type} ---")
        print(f"Total sample: {len(paths)}")


        unique_labels, counts = np.unique(labels, return_counts=True)
        label_counts = dict(zip(unique_labels, counts))

        for idx, cls in enumerate(self.classes):
            count = label_counts.get(idx, 0)
            percentage = (count / len(labels) * 100) if labels else 0
            print(f"  categories {cls}: {count} sample ({percentage:.1f}%)")
        print("-" * (len(f"--- testset: {test_type} ---")))

print("Finished the load of testloader")
```

```python
from tqdm import tqdm

# random seed
np.random.seed(Config.RANDOM_STATE)


train_loader = TrainDataLoader(data_dir=Config.DATA_DIR)
test_loader = TestDataLoader(data_dir=Config.DATA_DIR)

# load the data
train_paths, train_labels = train_loader.load_data()
test_sets = test_loader.load_data()

class_names = train_loader.get_class_names()

print(f"\n loded the data: ")
print(f" - Training smaples: {len(train_paths)}")
print(f" - Number of class: {len(class_names)}")
print(f" - class name: {class_names}")
print(f" - test mode: {list(test_sets.keys())}")


if not test_sets:
    print("\n warning: there is no test data loaded, please check the TestDataLoader logic and
test data directory structure.")
else:
    for test_type, (paths, _) in test_sets.items():
        print(f" - Testset '{test_type}' sample: {len(paths)}")


if train_paths:
    img_example = cv2.imread(train_paths[0])
    if img_example is not None:
        plt.imshow(cv2.cvtColor(img_example, cv2.COLOR_BGR2RGB))
        plt.title(f"sample image: {class_names[train_labels[0]]}")
        plt.axis('off')
        plt.show()
    else:
        print(f"Can't load the image: {train_paths[0]}")
else:
    print("there is no training data loaded, please check the TrainDataLoader logic and test
data directory structure.")


def extract_sift_features(image_path: str, feature_limit: Optional[int] = None) ->
Optional[np.ndarray]:

    try:
        img = cv2.imread(image_path)
        if img is None:
            print(f"Warning: can't load image: {image_path}")
            return None

        if len(img.shape) == 2 or (len(img.shape) == 3 and img.shape[2] == 1):
            gray = img
        elif len(img.shape) == 3 and img.shape[2] == 3:
            gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        elif len(img.shape) == 3 and img.shape[2] == 4:
            gray = cv2.cvtColor(img, cv2.COLOR_BGRA2GRAY)
        else:
            print(f"Warning: Image {image_path}can not identify the path: {img.shape}. Skip this
image。")
            return None


        sift = cv2.SIFT_create()

        keypoints = []
        descriptors = None

        if Config.DETECTOR_TYPE == 'SIFT':
            sift = cv2.SIFT_create(nfeatures=getattr(Config, 'FEATURE_LIMIT_PER_IMAGE', 0))
            keypoints, descriptors = sift.detectAndCompute(gray, None)

        elif Config.DETECTOR_TYPE == 'ORB':
            orb = cv2.ORB_create(nfeatures=getattr(Config, 'FEATURE_LIMIT_PER_IMAGE', 500))
            keypoints, descriptors = orb.detectAndCompute(gray, None)

        elif Config.DETECTOR_TYPE == 'HARRIS_SIFT':

            corners = cv2.goodFeaturesToTrack(
                gray,
                maxCorners=Config.MAX_CORNERS,
                qualityLevel=Config.HARRIS_QUALITY_LEVEL,
                minDistance=Config.HARRIS_MIN_DISTANCE,
            )

            if corners is not None and len(corners) > 0:
                keypoints = [
                    cv2.KeyPoint(
                        float(pt[0][0]),  # x
                        float(pt[0][1]),  # y
                        20                # size
                    )
                    for pt in corners
                ]


                sift_descriptor_extractor = cv2.SIFT_create()

                keypoints, descriptors = sift_descriptor_extractor.compute(gray, keypoints)

            else:
                keypoints = []
                descriptors = np.array([])

        elif Config.DETECTOR_TYPE == 'HARRIS_AFFINE':
                corners = cv2.goodFeaturesToTrack(
                    gray,
                    maxCorners=Config.MAX_CORNERS,
                    qualityLevel=Config.HARRIS_QUALITY_LEVEL,
                    minDistance=Config.HARRIS_MIN_DISTANCE
                )
                if corners is None or len(corners) == 0:
                    return None

                init_kps = [
                    cv2.KeyPoint(float(c[0][0]), float(c[0][1]), Config.INIT_KEYPOINT_SIZE)
                    for c in corners
                ]

                sift_base = cv2.xfeatures2d.SIFT_create()
                affine_detector = cv2.xfeatures2d.AffineAdaptedFeatureDetector_create(
                    sift_base,
                    maxIterations=Config.AFFINE_MAX_ITERATIONS,
                    responseThreshold=Config.AFFINE_RESPONSE_THRESHOLD,
                    edgeThreshold=Config.AFFINE_EDGE_THRESHOLD
                )

                affine_kps = affine_detector.detect(gray, init_kps)

                keypoints, descriptors = sift_base.compute(gray, affine_kps)

        else:
            raise ValueError(f"Unknown DETECTOR_TYPE: {Config.DETECTOR_TYPE}")

            keypoints, descriptors = sift.detectAndCompute(img, None)



            if descriptors is None or len(descriptors) == 0:
                print(f"`info: Can't find SIFT descriptors in image
{os.path.basename(image_path)} .")
                return None

            if feature_limit is not None and len(keypoints) > feature_limit:
                indices = np.argsort([kp.response for kp in keypoints])[::-1][:feature_limit]
                descriptors = descriptors[indices]
                # keypoints = [keypoints[i] for i in indices]

        return descriptors.astype(np.float32)

    except Exception as e:
        print(f"Error: Wrong when process the image {image_path} : {e}")
        return None

if train_paths:
    print("extracting features from the first training image...")
    example_descriptors = extract_sift_features(train_paths[0],
feature_limit=Config.FEATURE_LIMIT_PER_IMAGE)
    if example_descriptors is not None:
        print(f"sample image extract  {example_descriptors[0]} descriptors, then the dimension
is {example_descriptors.shape[1]}")
    else:
        print("fail to extract the features, please check the image path and the image format.")
else:
    print("no training data, skip the feature extraction test.")

# --- Visual Vocabulary Construction---
```

```
def build_vocabulary(image_paths: List[str], vocab_size: int, sample_limit: Optional[int] =
None) -> Optional[
    MiniBatchKMeans]:
    all_descriptors = []
    print(f"Start extracting features to build vocabulary...")

    paths_to_process = image_paths
    if sample_limit is not None and len(image_paths) > sample_limit:
        print(f"Randomly sampling {sample_limit} images from {len(image_paths)} images to build
vocabulary...")
        paths_to_process = np.random.choice(image_paths, sample_limit, replace=False).tolist()

    for image_path in tqdm(paths_to_process, desc="Extracting vocabulary features"):
        descriptors = extract_sift_features(image_path,
feature_limit=Config.FEATURE_LIMIT_PER_IMAGE)
        if descriptors is not None:
            all_descriptors.append(descriptors)

    if not all_descriptors:
        print("Error: Could not extract features from any images to build vocabulary.")
        return None

    stacked_descriptors = np.vstack(all_descriptors)
    print(f"Extracted {stacked_descriptors.shape[0]} descriptors to build vocabulary.")

    if stacked_descriptors.shape[0] < vocab_size:
        print(
            f"Warning: Number of descriptors ({stacked_descriptors.shape[0]}) is less than
vocabulary size ({vocab_size}). Reducing vocabulary size to {stacked_descriptors.shape[0]}.")
        vocab_size = stacked_descriptors.shape[0]
        if vocab_size == 0:
            print("Error: Number of descriptors is 0, cannot perform clustering.")
            return None

    print(f"Starting MiniBatchKMeans to build visual vocabulary of size {vocab_size}...")
    kmeans = MiniBatchKMeans(n_clusters=vocab_size,
                             random_state=Config.RANDOM_STATE,
                             batch_size=256 * 8,  # adjustable parameter
                             init='k-means++',  # initialization method
                             max_iter=100,  # adjustable parameter
                             n_init=10,  # explicitly set n_init
                             verbose=1)  # can set to 1 to see progress

    kmeans.fit(stacked_descriptors)
    print("Visual vocabulary construction complete.")

    return kmeans


vocabulary_kmeans = build_vocabulary(train_paths, Config.VOCAB_SIZE,
sample_limit=Config.VOCAB_SAMPLE_LIMIT)

if vocabulary_kmeans:
    vocab_filename = f"visual_vocabulary_k{Config.VOCAB_SIZE}.pkl"
    with open(vocab_filename, 'wb') as f:
        pickle.dump(vocabulary_kmeans, f)
    print(f"Visual vocabulary saved to {vocab_filename}")
```

```python
else:
    print("Failed to build visual vocabulary. Following steps cannot proceed.")

# --- Image Representation (Bag-of-Words Histogram) ---

def compute_bow_histogram(descriptors: Optional[np.ndarray], kmeans_model: MiniBatchKMeans) ->
np.ndarray:

    vocab_size = kmeans_model.n_clusters
    histogram = np.zeros(vocab_size, dtype=np.float32)

    if descriptors is not None and descriptors.shape[0] > 0:
        # Find the nearest visual word (cluster center) for each descriptor
        visual_words = kmeans_model.predict(descriptors)

        # Calculate frequencies of visual words
        unique_words, counts = np.unique(visual_words, return_counts=True)
        histogram[unique_words] = counts

        # L2 normalization (optional but recommended)
        # norm = np.linalg.norm(histogram)
        # if norm > 0:
        #     histogram /= norm

    # L1 normalization (another option)
    # total_features = np.sum(histogram)
    # if total_features > 0:
    #     histogram /= total_features

    return histogram


# Computing BoW features for all training data
if vocabulary_kmeans:
    print("\nComputing BoW features for training data...")
    X_train_bow = []
    y_train = []  # List of corresponding labels

    for i in tqdm(range(len(train_paths)), desc="Computing training set BoW"):
        path = train_paths[i]
        label = train_labels[i]

        descriptors = extract_sift_features(path, feature_limit=Config.FEATURE_LIMIT_PER_IMAGE)
        bow_hist = compute_bow_histogram(descriptors, vocabulary_kmeans)

        X_train_bow.append(bow_hist)
        y_train.append(label)  # Keep labels
    X_train_bow = np.array(X_train_bow)
    n_images, n_clusters = X_train_bow.shape
    df = np.count_nonzero(X_train_bow > 0, axis=0)
    doc_freq = df / n_images
    mask = (doc_freq >= 0.001) & (doc_freq <= 0.95)
    hist_filtered = X_train_bow[:, mask]
    from sklearn.feature_extraction.text import TfidfTransformer

    tfidf = TfidfTransformer(norm='l2')
    hist_tfidf = tfidf.fit_transform(hist_filtered).toarray()
    X_train_bow = hist_filtered

    y_train = np.array(y_train)
    print(f"Training set BoW features completed. Feature matrix shape: {X_train_bow.shape}")

    # Computing BoW features for all test data
    X_test_bow_sets = {}
    y_test_sets = {}
    print("\nComputing BoW features for test data...")
    for test_type, (paths, labels) in test_sets.items():
        print(f"Computing test set '{test_type}' BoW features...")
        X_test_bow = []
        y_test = []
        for i in tqdm(range(len(paths)), desc=f"Computing {test_type} BoW"):
            path = paths[i]
            label = labels[i]
            descriptors = extract_sift_features(path,
feature_limit=Config.FEATURE_LIMIT_PER_IMAGE)
            bow_hist = compute_bow_histogram(descriptors, vocabulary_kmeans)
            X_test_bow.append(bow_hist)
            y_test.append(label)

        X_test_bow_sets[test_type] = np.array(X_test_bow)
        y_test_sets[test_type] = np.array(y_test)
        print(
            f"Test set '{test_type}' BoW features completed. Feature matrix shape:
{X_test_bow_sets[test_type].shape}")

else:
    print("Error: No visual vocabulary available, cannot compute BoW features.")
    X_train_bow, y_train = None, None
    X_test_bow_sets, y_test_sets = {}, {}


# --- Classifier Training and Recognition ---


if X_train_bow is not None and y_train is not None and X_train_bow.shape[0] > 0:
    print("\nStarting classifier training...")

    classifier = SVC(kernel='rbf', C=1.0, probability=False, random_state=Config.RANDOM_STATE)

    classifier.fit(X_train_bow, y_train)
    print("Classifier training completed.")

    classifier_filename = f"bow_classifier_{type(classifier).__name__}_k{Config.VOCAB_SIZE}.pkl"
    with open(classifier_filename, 'wb') as f:
        pickle.dump(classifier, f)
    print(f"Classifier saved to {classifier_filename}")

    print("\nEvaluating model performance on test sets...")
    if not X_test_bow_sets:
        print("No test set BoW features available for evaluation.")
    else:
        for test_type in test_sets.keys():
            print(f"\n--- Evaluating test set: {test_type} ---")
            if test_type not in X_test_bow_sets or X_test_bow_sets[test_type].shape[0] == 0:
                print(f"Skipping evaluation since test set '{test_type}' has no BoW features.")
                continue

            X_test = X_test_bow_sets[test_type]
            X_test_filt = X_test[:, mask]
            X_test_tfidf = tfidf.transform(X_test_filt).toarray()
            y_test = y_test_sets[test_type]

            # Make predictions
            y_pred = classifier.predict(X_test_filt)


            accuracy = accuracy_score(y_test, y_pred)
            print(f"Accuracy: {accuracy:.4f}")


            unique_test_labels = np.unique(y_test)
            current_target_names = [class_names[i] for i in unique_test_labels]

            report = classification_report(y_test, y_pred, labels=unique_test_labels,
target_names=current_target_names,
                                           zero_division=0)
            print("Classification Report:")
            print(report)

else:
    print("Error: No training data (BoW features) available, cannot train or evaluate
classifier.")


print("\n=== Pipeline Execution Complete ===")
```

Main Code Implementation of Traditional CV Method in Cifar-10:

```
class ConfigCIFAR10:

    IMAGE_SIZE = (32, 32)
    TARGET_IMAGE_SIZE = (64, 64)  # Target image size
    N_CLASSES = 10
    CLASS_NAMES = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',
'ship', 'truck']


    DETECTOR_TYPE = 'SIFT'  # Options: 'SIFT', 'ORB'
    FEATURE_LIMIT_PER_IMAGE = 100

    VOCAB_SIZE = 500
    VOCAB_SAMPLE_LIMIT = 5000

    SVM_KERNEL = 'rbf'  # SVM kernel function
    SVM_C = 1.0
    SVM_GAMMA = 'scale'
    TEST_SIZE = 0.2
    RANDOM_STATE = 42  # Random seed

    PLOT_ROC_AUC = True

    PERFORM_GRID_SEARCH = True

    GRID_SEARCH_CV_FOLDS = 3

    MODEL_DIR = "cifar10_models/"
    VOCAB_FILE = os.path.join(MODEL_DIR, f"vocab_k{VOCAB_SIZE}_{DETECTOR_TYPE}.pkl")
    SVM_MODEL_FILE = os.path.join(MODEL_DIR, f"svm_model_{DETECTOR_TYPE}_vocab{VOCAB_SIZE}.pkl")

    # 确保模型目录存在
    if not os.path.exists(MODEL_DIR):
        os.makedirs(MODEL_DIR)


config = ConfigCIFAR10()
print(f"the parameters has been set. the vocabulary size is {config.VOCAB_SIZE}, the detector is
{config.DETECTOR_TYPE}")
print(f"the model will be saved in {config.MODEL_DIR}")



from typing import Tuple, List
import numpy as np
import cv2
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
from tqdm import tqdm
import matplotlib.pyplot as plt

def load_cifar10_data() -> Tuple[np.ndarray, np.ndarray, np.ndarray, np.ndarray, List[str]]:
    print("Starting to load CIFAR-10 dataset (this may take several minutes)...")
    cifar10 = fetch_openml(data_id=40926, as_frame=False, parser='auto')
    print("CIFAR-10 dataset loaded.")

    X = cifar10.data.astype('uint8')
    y = cifar10.target.astype('uint8')
    class_names = config.CLASS_NAMES

    if X.shape[1] == 3072:
        X = X.reshape(-1, 3, 32, 32).transpose(0, 2, 3, 1)
    elif X.shape[1] == 3 and X.shape[2] == 32 and X.shape[3] == 32:
        X = X.transpose(0, 2, 3, 1)
    elif X.shape[1] == 32 and X.shape[2] == 32 and X.shape[3] == 3:
        pass
    else:
        raise ValueError(f"Unknown CIFAR-10 data shape: {X.shape}")

    print(f"Original image data shape: {X.shape}")
    print(f"Label data shape: {y.shape}")

    # Convert RGB images to grayscale

    print("Converting images to grayscale...")
    X_gray = np.array([
        cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
        for img in tqdm(X, desc="Grayscale conversion")
    ])
    print(f"Grayscale image data shape: {X_gray.shape}")

    # Resize images if needed
    if config.TARGET_IMAGE_SIZE != config.IMAGE_SIZE:
        print(f"Resizing grayscale images from {config.IMAGE_SIZE} to
{config.TARGET_IMAGE_SIZE}...")
        X_processed = np.array([
            cv2.resize(img, config.TARGET_IMAGE_SIZE, interpolation=cv2.INTER_CUBIC)
            for img in tqdm(X_gray, desc="Resolution adjustment")
        ])
        print(f"Resized image data shape: {X_processed.shape}")
    else:
        X_processed = X_gray
        print("Target image size matches original; no resizing performed.")


    X_train, X_test, y_train, y_test = train_test_split(
        X_gray, y,
        test_size=config.TEST_SIZE,
        random_state=config.RANDOM_STATE,
        stratify=y
    )
    print(f"Number of training images: {X_train.shape[0]}, Number of test images:
{X_test.shape[0]}")
    print(f"Number of training labels: {y_train.shape[0]}, Number of test labels:
{y_test.shape[0]}")

    return X_train, X_test, y_train, y_test, class_names

# Execute data loading
X_train_raw, X_test_raw, y_train, y_test, class_names = load_cifar10_data()



def plot_cifar_samples(images, labels, class_names_list, num_samples=10):
    plt.figure(figsize=(12, 5))
    for i in range(num_samples):
        plt.subplot(2, num_samples // 2, i + 1)
        plt.imshow(images[i], cmap='gray' if images[i].ndim == 2 else None)
        plt.title(f"Category: {class_names_list[labels[i]]}")
        plt.axis('off')
    plt.suptitle("CIFAR-10 Gray Image Samples", fontsize=16)
    plt.tight_layout(rect=[0, 0, 1, 0.96])
    plt.show()

plot_cifar_samples(X_train_raw, y_train, class_names, num_samples=10)

print(f"\nData loading and initial preprocessing complete. Class names: {class_names}")
unique_labels, counts = np.unique(y_train, return_counts=True)
print(f"Training set samples per class: {dict(zip([class_names[i] for i in unique_labels],
counts))}")
unique_labels_test, counts_test = np.unique(y_test, return_counts=True)
```

```python
print(f"Test set samples per class: {dict(zip([class_names[i] for i in unique_labels_test],
counts_test))}")
def create_detector(detector_type: str):
    """
    Create a feature detector/descriptor based on the specified type.
    Note: SIFT may require opencv-contrib-python in some OpenCV versions.
          Make sure it is installed: pip install opencv-python opencv-contrib-python
    """
    if detector_type == 'SIFT':
        # SIFT may perform poorly on small images (32x32), parameters may need adjustment
        try:
            return cv2.SIFT_create(
                nfeatures=config.FEATURE_LIMIT_PER_IMAGE,
                contrastThreshold=0.04,
                edgeThreshold=10
            )
        except AttributeError:
            print("Error: cv2.SIFT_create() is not available. Make sure opencv-contrib-python is
installed and compatible.")
            print("Try installing: pip install opencv-contrib-python")
            raise
    elif detector_type == 'ORB':
        # nfeatures: maximum number of features
        # scaleFactor: pyramid decimation ratio (>1), used to build the image pyramid
        # nlevels: number of pyramid levels
        return cv2.ORB_create(
            nfeatures=config.FEATURE_LIMIT_PER_IMAGE,
            scoreType=cv2.ORB_FAST_SCORE
        )
    else:
        raise ValueError(f"Unsupported detector type: {detector_type}")


def extract_features(images: np.ndarray, detector) -> Tuple[List[np.ndarray], List[int]]:

    all_descriptors = []
    descriptors_per_image_count = []
    print(f"Extracting features from {len(images)} images using {config.DETECTOR_TYPE}...")

    for img in tqdm(images, desc="Feature extraction"):
        # Ensure image is uint8
        if img.dtype != np.uint8:
            img_processed = cv2.normalize(img, None, 0, 255, cv2.NORM_MINMAX).astype('uint8')
        else:
            img_processed = img

        keypoints, descriptors = detector.detectAndCompute(img_processed, None)

        if descriptors is not None:


            if len(descriptors) > 0:
                all_descriptors.append(descriptors)
                descriptors_per_image_count.append(len(descriptors))
            else:

                descriptors_per_image_count.append(0)
        else:

            descriptors_per_image_count.append(0)

    if not all_descriptors:
        print(f"Warning: No descriptors extracted from any image. Check detector type
'{config.DETECTOR_TYPE}' and image quality/size.")
        return [], []

    total_images_with_desc = sum(1 for count in descriptors_per_image_count if count > 0)
    total_descriptors = sum(descriptors_per_image_count)
    print(f"Feature extraction complete. Extracted {total_descriptors} descriptors from
{total_images_with_desc} images.")
    avg_descriptors = (
        np.mean([c for c in descriptors_per_image_count if c > 0])
        if any(c > 0 for c in descriptors_per_image_count)
        else 0
    )
    print(f"Average descriptors per valid image: {avg_descriptors:.2f}")
    print(f"Number of descriptor arrays: {len(all_descriptors)}")

    # Visualize keypoints on the first image that has descriptors
    for i, desc in enumerate(all_descriptors):
        if desc is not None and len(desc) > 0:
            img_with_keypoints = cv2.drawKeypoints(
                images[i],
                keypoints,
                None,
                flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS
            )
            plt.imshow(img_with_keypoints, cmap='gray')
            plt.title(f"Keypoints of image {i} ({len(desc)} descriptors)")
            plt.axis('off')
            plt.show()
            break  # Show only the first example

    return all_descriptors, descriptors_per_image_count


# Initialize the detector
try:
    detector = create_detector(config.DETECTOR_TYPE)
    print(f"{config.DETECTOR_TYPE} detector created successfully.")


    num_vocab_samples = min(config.VOCAB_SAMPLE_LIMIT, len(X_train_raw))
    if num_vocab_samples < len(X_train_raw):
        print(f"Sampling {num_vocab_samples} images from the training set for vocabulary
construction.")
        sample_indices = np.random.choice(len(X_train_raw), num_vocab_samples, replace=False)
        X_vocab_samples = X_train_raw[sample_indices]
    else:
        print("Using all training images for vocabulary construction.")
        X_vocab_samples = X_train_raw

    print(f"Starting feature extraction for vocabulary construction ({len(X_vocab_samples)}
images)...")
```

```
    vocab_descriptors_list, _ = extract_features(X_vocab_samples, detector)

    if not vocab_descriptors_list:
        raise ValueError("No descriptors extracted for vocabulary construction. Pipeline cannot
continue. Try a different detector or check your images.")

    # Stack all descriptors into one large NumPy array for K-Means
    all_descriptors_for_vocab = np.vstack(
        [desc for desc in vocab_descriptors_list if desc is not None and len(desc) > 0]
    )
    print(f"Total descriptors for vocabulary: {all_descriptors_for_vocab.shape[0]}, Dimension:
{all_descriptors_for_vocab.shape[1]}")

except Exception as e:
    print(f"Error during feature extraction stage: {e}")
    all_descriptors_for_vocab = None  # Ensure variable exists to avoid downstream errors

def build_vocabulary(descriptors_matrix: np.ndarray, vocab_size: int, random_state: int) ->
MiniBatchKMeans:

    if descriptors_matrix is None or descriptors_matrix.shape[0] == 0:
        raise ValueError("Descriptor matrix is empty; cannot build vocabulary.")
    if descriptors_matrix.shape[0] < vocab_size:
        print(f"Warning: number of descriptors ({descriptors_matrix.shape[0]}) is less than
requested vocabulary size ({vocab_size}).")
        print(f"Setting KMeans n_clusters to {descriptors_matrix.shape[0]}.")
        actual_vocab_size = descriptors_matrix.shape[0]
    else:
        actual_vocab_size = vocab_size

    print(f"Starting to build visual vocabulary of size {actual_vocab_size} using
MiniBatchKMeans...")

    kmeans = MiniBatchKMeans(
        n_clusters=actual_vocab_size,
        random_state=random_state,
        batch_size=256 * 3,
        n_init='auto',
        max_iter=100,
        verbose=0
    )

    start_time = time.time()
    kmeans.fit(descriptors_matrix)
    end_time = time.time()
    print(f"Visual vocabulary built. Time elapsed: {end_time - start_time:.2f} seconds.")
    print(f"Vocabulary center shape (K, D): {kmeans.cluster_centers_.shape}")
    return kmeans


# Build or load vocabulary
vocabulary = None
if all_descriptors_for_vocab is not None and all_descriptors_for_vocab.shape[0] > 0:
    try:
        if os.path.exists(config.VOCAB_FILE):
            print(f"Loading precomputed vocabulary from file: {config.VOCAB_FILE}")
            with open(config.VOCAB_FILE, 'rb') as f:
                vocabulary = pickle.load(f)
            if not hasattr(vocabulary, 'n_clusters') or vocabulary.n_clusters !=
config.VOCAB_SIZE:
                print(
                    f"Warning: loaded vocabulary size ({getattr(vocabulary, 'n_clusters',
'unknown')}) "
                    f"does not match config ({config.VOCAB_SIZE}). Rebuilding.")
                vocabulary = None  # force rebuild
            else:
                print("Vocabulary loaded successfully.")

        if vocabulary is None:  # file missing or size mismatch
            print("Building a new visual vocabulary...")
            vocabulary = build_vocabulary(all_descriptors_for_vocab, config.VOCAB_SIZE,
config.RANDOM_STATE)
            print(f"Saving vocabulary to: {config.VOCAB_FILE}")
            with open(config.VOCAB_FILE, 'wb') as f:
                pickle.dump(vocabulary, f)
            print("Vocabulary saved.")

    except ValueError as ve:
        print(f"Error while building vocabulary: {ve}")
        print("Pipeline may not continue.")
    except Exception as e:
        print(f"Unknown error while building or loading vocabulary: {e}")
else:
    print("Error: no descriptors available for vocabulary construction. Please check the feature
extraction step.")

if vocabulary:
    print(f"Final vocabulary size (K): {vocabulary.n_clusters}")
else:
    print("Failed to build or load vocabulary.")


import numpy as np
from sklearn import svm
from tqdm import tqdm

def descriptors_to_bovw_histograms(
        image_descriptors_list: List[np.ndarray],
        vocabulary: MiniBatchKMeans,
        descriptors_counts: List[int]
) -> np.ndarray:

    vocab_size = vocabulary.n_clusters
    num_images = len(descriptors_counts)

    bovw_histograms = np.zeros((num_images, vocab_size), dtype=np.float32)

    print(f"Starting conversion of {len(image_descriptors_list)} descriptor sets (for
{num_images} images) to BoVW histograms...")

    desc_list_idx = 0
    for i in tqdm(range(num_images), desc="Images → BoVW vectors"):
        if descriptors_counts[i] > 0:  # If the current image has descriptors
            if desc_list_idx < len(image_descriptors_list) and \
               image_descriptors_list[desc_list_idx] is not None and \
               image_descriptors_list[desc_list_idx].shape[0] > 0:

                descriptors = image_descriptors_list[desc_list_idx]
                visual_words = vocabulary.predict(descriptors)

                # Compute the histogram of visual words
                hist, _ = np.histogram(visual_words, bins=np.arange(vocab_size + 1),
density=False)

                #  L1 normalize the histogram
                # hist = hist.astype(np.float32) / np.sum(hist)  # L1 normalization
                # hist = hist.astype(np.float32) / np.linalg.norm(hist)  # L2 normalization
                # if np.sum(hist) > 0:
                #     hist = hist.astype(np.float32) / np.sum(hist)

                bovw_histograms[i, :] = hist
                desc_list_idx += 1
            else:

                pass

    if desc_list_idx != len(image_descriptors_list):
        print(f"Warning: number of descriptor sets processed ({desc_list_idx}) does not match
the expected ({len(image_descriptors_list)}).")

    print("BoVW feature vectors construction complete.")
    print(f"BoVW feature matrix shape: {bovw_histograms.shape}")
    return bovw_histograms



if vocabulary is not None:

    print("Extracting features from training set to build BoVW vectors...")
    train_descriptors_list, train_desc_counts = extract_features(X_train_raw, detector)
    if not train_descriptors_list and sum(train_desc_counts) == 0:
        print("Error: No feature descriptors extracted from the training set. Cannot create BoVW
vectors.")
        X_train_bovw = np.array([])
    else:

        X_train_bovw = descriptors_to_bovw_histograms(train_descriptors_list, vocabulary,
train_desc_counts)
        print(f"Training set BoVW feature shape: {X_train_bovw.shape}")

    print("\nExtracting features from test set to build BoVW vectors...")
    test_descriptors_list, test_desc_counts = extract_features(X_test_raw, detector)

    if not test_descriptors_list and sum(test_desc_counts) == 0:
        print("Error: No feature descriptors extracted from the test set. Cannot create BoVW
vectors.")
        X_test_bovw = np.array([])
    else:
        X_test_bovw = descriptors_to_bovw_histograms(test_descriptors_list, vocabulary,
test_desc_counts)
        print(f"Test set BoVW feature shape: {X_test_bovw.shape}")
    # Verify that shapes match expectations
    if X_train_bovw.shape[0] != len(X_train_raw) or \
       (len(X_test_raw) > 0 and X_test_bovw.shape[0] != len(X_test_raw)):
        print("Warning: Number of BoVW vectors does not match number of original images! Please
check the pipeline.")
        print(f"X_train_raw: {len(X_train_raw)}, X_train_bovw: {X_train_bovw.shape[0]}")
        if len(X_test_raw) > 0:
            print(f"X_test_raw: {len(X_test_raw)}, X_test_bovw: {X_test_bovw.shape[0]}")

    if X_train_bovw.shape[0] > 0:
        print(f"\nExample BoVW histogram for one training image (first 50
words):\n{X_train_bovw[0, :50]}")
        print(f"Sum of histogram bins: {np.sum(X_train_bovw[0, :])} (equals number of keypoints
when not normalized)")

else:
    print("Error: Vocabulary is not defined. Cannot convert images to BoVW feature vectors.
Please rerun previous steps.")
    X_train_bovw = np.array([])
    X_test_bovw = np.array([])



def plot_roc_auc_multiclass(y_true, y_score, class_names, figsize=(10, 8)):

    n_classes = len(class_names)
    y_true_binarized = label_binarize(y_true, classes=np.arange(n_classes))
    if y_score.shape[1] != n_classes:
        raise ValueError(
            f"Number of columns in y_score ({y_score.shape[1]}) "
            f"must equal number of classes ({n_classes})"
        )

    plt.figure(figsize=figsize)


    for i in range(n_classes):
        fpr, tpr, _ = roc_curve(y_true_binarized[:, i], y_score[:, i])
        roc_auc = auc(fpr, tpr)
        plt.plot(
            fpr,
            tpr,
            lw=2,
            label=f'Class {class_names[i]} (AUC = {roc_auc:.2f})'
        )

    fpr_micro, tpr_micro, _ = roc_curve(y_true_binarized.ravel(), y_score.ravel())
    roc_auc_micro = auc(fpr_micro, tpr_micro)
    plt.plot(
        fpr_micro,
        tpr_micro,
        label=f'Micro-average ROC (AUC = {roc_auc_micro:.2f})',
        color='deeppink',
        linestyle=':',
        linewidth=4
    )
    all_fpr = np.unique(
        np.concatenate([
            roc_curve(y_true_binarized[:, i], y_score[:, i])[0]
            for i in range(n_classes)
        ])
    )

    mean_tpr = np.zeros_like(all_fpr)
    for i in range(n_classes):
        fpr_i, tpr_i, _ = roc_curve(y_true_binarized[:, i], y_score[:, i])
        mean_tpr += np.interp(all_fpr, fpr_i, tpr_i)
    # Average and compute AUC
    mean_tpr /= n_classes
    roc_auc_macro = auc(all_fpr, mean_tpr)
    plt.plot(
        all_fpr,
        mean_tpr,
        label=f'Macro-average ROC (AUC = {roc_auc_macro:.2f})',
        color='navy',
        linestyle=':',
        linewidth=4
    )


    plt.plot([0, 1], [0, 1], 'k--', lw=2)

    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title(
        f'Multiclass ROC Curve '
        f'(Using {config.SVM_KERNEL} Kernel SVM, C={config.SVM_C}, gamma={config.SVM_GAMMA})'
    )
    plt.legend(loc="lower right")
    plt.grid(True)
    plt.show()


import numpy as np
from sklearn import svm
import time
import os
import pickle
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
from typing import List
from sklearn.cluster import MiniBatchKMeans

def train_svm_classifier(X_train_features: np.ndarray, y_train_labels: np.ndarray, random_state:
int,
                         C_value: float = 1.0, gamma_value: str = 'scale') -> svm.SVC:

    if X_train_features.shape[0] == 0:
        raise ValueError("Training features are empty; cannot train SVM classifier.")
    if len(np.unique(y_train_labels)) < 2:
        raise ValueError(f"Fewer than 2 classes in training labels
({np.unique(y_train_labels)}); cannot train classifier.")

    print(f"Starting SVM classifier training... C={C_value}, gamma={gamma_value}")
    print(f"Training data shape: {X_train_features.shape}, Label shape: {y_train_labels.shape}")

    classifier = svm.SVC(
        kernel=config.SVM_KERNEL,
        C=C_value,
        gamma=gamma_value,
        random_state=random_state,
        probability=True if config.PLOT_ROC_AUC else False
    )

    start_time = time.time()
    classifier.fit(X_train_features, y_train_labels)
    end_time = time.time()

    print(f"SVM classifier training complete. Time elapsed: {end_time - start_time:.2f}
seconds.")
    return classifier


def evaluate_classifier(classifier, X_test_features: np.ndarray, y_test_labels: np.ndarray,
                        class_names_list: List[str]):  # , scaler=None):
    if X_test_features.shape[0] == 0:
        print("Test features are empty; cannot evaluate.")
        return

    # if scaler:
    #     X_test_features = scaler.transform(X_test_features)

    print("\nEvaluating classifier on test set...")
    y_pred = classifier.predict(X_test_features)

    accuracy = accuracy_score(y_test_labels, y_pred)
    print(f"Accuracy: {accuracy:.4f}")

    print("\nClassification Report:")
    report = classification_report(
        y_test_labels,
        y_pred,
        target_names=class_names_list,
        zero_division=0
    )
    print(report)

    print("\nConfusion Matrix:")
    cm = confusion_matrix(y_test_labels, y_pred)
    plt.figure(figsize=(10, 8))
    sns.heatmap(
        cm,
        annot=True,
        fmt='d',
        cmap='Blues',
        xticklabels=class_names_list,
        yticklabels=class_names_list
    )
    plt.xlabel('Predicted Label')
    plt.ylabel('True Label')
    plt.title('Confusion Matrix')
    plt.show()

    if config.PLOT_ROC_AUC and hasattr(classifier, "predict_proba"):
        try:
            y_proba = classifier.predict_proba(X_test_features)
            plot_roc_auc_multiclass(y_test_labels, y_proba, class_names_list)
        except Exception as e_roc:
            print(f"Error plotting ROC AUC curve: {e_roc}")



if 'X_train_bovw' in globals() and 'X_test_bovw' in globals() and \
        X_train_bovw.shape[0] > 0 and y_train.shape[0] > 0:

    # Ensure the number of training features matches the number of labels
    if X_train_bovw.shape[0] != y_train.shape[0]:
        print(f"Error: Number of training BoVW features ({X_train_bovw.shape[0]}) "
              f"does not match number of training labels ({y_train.shape[0]})!")
    else:
        svm_classifier = None
        try:
            # Train the SVM
            svm_classifier = train_svm_classifier(
                X_train_bovw,
                y_train,
                config.RANDOM_STATE,
                config.SVM_C,
                config.SVM_GAMMA
            )


            if X_test_bovw.shape[0] > 0 and y_test.shape[0] > 0:
                if X_test_bovw.shape[0] != y_test.shape[0]:
                    print(f"Warning: Number of test BoVW features ({X_test_bovw.shape[0]}) "
                          f"does not match number of test labels ({y_test.shape[0]})! Skipping
evaluation.")
                else:
                    evaluate_classifier(svm_classifier, X_test_bovw, y_test, class_names)
            elif X_test_bovw.shape[0] == 0 and y_test.shape[0] > 0:
                print("Test set BoVW features are empty; cannot evaluate.")
            else:
                print("Test set is empty or BoVW features were not generated; skipping
evaluation.")

        except ValueError as ve:
            print(f"Value error during SVM training or evaluation: {ve}")
        except Exception as e:
            print(f"Unexpected error during SVM training or evaluation: {e}")
else:
    print("Error: BoVW features were not generated successfully or training data is empty.
Cannot train or evaluate SVM classifier.")
    print(f"X_train_bovw exists: {'X_train_bovw' in globals()}, count: {X_train_bovw.shape[0] if
'X_train_bovw' in globals() else 'N/A'}")
```

```
    print(f"y_train exists: {'y_train' in globals()}, count: {y_train.shape[0] if 'y_train' in
globals() else 'N/A'}")


from sklearn.model_selection import GridSearchCV

def tune_svm_parameters(X_train_features: np.ndarray, y_train_labels: np.ndarray, random_state:
int) -> GridSearchCV:

    if X_train_features.shape[0] == 0:
        raise ValueError("Training features are empty; cannot perform parameter tuning.")
    if len(np.unique(y_train_labels)) < 2:
        raise ValueError("Fewer than 2 classes in training labels; cannot perform parameter
tuning.")

    print("Starting SVM parameter tuning (GridSearchCV)...")

    if config.SVM_KERNEL == 'linear':
        param_grid = {
            'C': [1,10,15]
        }
    else:  # e.g. 'rbf'
        param_grid = {
            'C': [1],
            'gamma': [0.01, 0.1, 1, 'scale', 'auto']
        }

    base_svm = svm.SVC(
        kernel=config.SVM_KERNEL,
        random_state=random_state,
        probability=config.PLOT_ROC_AUC
    )

    grid_search = GridSearchCV(
        estimator=base_svm,
        param_grid=param_grid,
        cv=config.GRID_SEARCH_CV_FOLDS,  # 3,5
        scoring='accuracy',
        verbose=2,
        n_jobs=-1
    )

    print(f"Parameter grid: {param_grid}")
    print(f"Performing search with {config.GRID_SEARCH_CV_FOLDS}-fold cross-validation...")

    start_time = time.time()
    grid_search.fit(X_train_features, y_train_labels)
    end_time = time.time()

    print(f"GridSearchCV complete. Time elapsed: {end_time - start_time:.2f} seconds.")
    print(f"Best parameters: {grid_search.best_params_}")
    print(f"Best cross-validation accuracy with best parameters: {grid_search.best_score_:.4f}")

    return grid_search
```

```
if config.PERFORM_GRID_SEARCH and 'X_train_bovw' in globals() and X_train_bovw.shape[0] > 0 and
y_train.shape[0] > 0:
    if X_train_bovw.shape[0] != y_train.shape[0]:
        print(
            f"Error: Number of training BoVW features ({X_train_bovw.shape[0]}) does not match
number of training labels ({y_train.shape[0]}); cannot perform parameter tuning.")
    else:
        try:
            print("\n--- SVM Parameter Tuning ---")
            grid_search_cv = tune_svm_parameters(X_train_bovw, y_train, config.RANDOM_STATE)

            best_svm_classifier = grid_search_cv.best_estimator_
            print("\n--- Evaluating best SVM model from GridSearchCV ---")

            if X_test_bovw.shape[0] > 0 and y_test.shape[0] > 0:
                if X_test_bovw.shape[0] != y_test.shape[0]:
                    print(
                        f"Warning: Number of test BoVW features ({X_test_bovw.shape[0]}) does
not match number of test labels ({y_test.shape[0]}); skipping evaluation.")
                else:
                    evaluate_classifier(best_svm_classifier, X_test_bovw, y_test, class_names)
            elif X_test_bovw.shape[0] == 0 and y_test.shape[0] > 0:
                print("Test set BoVW features are empty; cannot evaluate tuned model.")
            else:
                print("Test set is empty or BoVW features not generated; skipping tuned model
evaluation.")

            # model_filename =
f"best_svm_model_{config.DETECTOR_TYPE}_vocab{config.VOCAB_SIZE}.pkl"
            # print(f"Saving best SVM model to: {model_filename}")
            # with open(model_filename, 'wb') as f:
            #     pickle.dump(best_svm_classifier, f)

        except ValueError as ve:
            print(f"Value error during SVM parameter tuning: {ve}")
        except Exception as e:
            print(f"Unexpected error during SVM parameter tuning: {e}")
elif not config.PERFORM_GRID_SEARCH:
    print("\nParameter tuning is disabled in configuration (PERFORM_GRID_SEARCH=False). Skipping
this step.")
else:
    print("\nError: BoVW features not generated successfully or training data is empty; cannot
perform SVM parameter tuning.")
```

Appendix B:

Official Description of Test Set on iCubWorld1.0:

For training, we collected 3 objects per category. Each object has been acquired in human mode (see Acquisition Section), recording 200 images of size 640x480 from one iCub camera, subsequently cropped to a bounding box of size 160x160 around the object.

We provide the following 4 test sets:

DEMONSTRATOR

1 known instance per category acquired in human mode with a different demonstrator with respect to the training

CATEGORIZATION

1 new instance per category acquired in the same setting as for the training

ROBOT

1 known or new instance per category acquired in robot mode (the bounding box in this case is 320x320)

BACKGROUND

we select 10 images per category where the classifiers perform 99% of accuracy and provide the segmentation mask of the objects to test whether the classifier recognizes the object or the background