From SIFT–BoW to End-to-End CNNs: A Comparative Study on Standard Vision Benchmarks

By Shuyan Zhang

I. Introduction

Object recognition in robotics demands both robust feature extraction and efficient classification under varying imaging conditions. Classical pipelines such as SIFT–BoW extract local, scale- and rotation-invariant descriptors, quantise them into "visual words," and classify via simple probabilistic or distance-based models. More recently, deep CNNs automate feature learning end-to-end, preserving spatial relationships and exploiting hierarchical representations. This coursework investigates how these two frameworks perform on standard vision benchmarks (CIFAR-10, CIFAR-100, MS COCO) and a robotics-centric dataset (iCubWorld). Through systematic hyperparameter exploration and evaluation of computational requirements—training time, inference speed, memory footprint, and standard metrics (accuracy, F1, mAP)—we aim to elucidate the strengths and limitations of traditional versus deep learning methods in robotic vision contexts.

II. Dataset selection and preprocessing

This study employs both standard and robotics-oriented vision benchmarks to evaluate the efficacy of traditional computer-vision pipelines versus deep convolutional approaches. Datasets selected span from low-resolution, widely used benchmarks, ideal for rapid hyperparameter exploration, to more complex, robotics-relevant collections.

A. Benchmark options

1) CIFAR-10: The CIFAR-10 dataset consists of 60 000 color images of size 32 × 32 pixels distributed evenly across 10 classes, with 6 000 images per class. CIFAR-10's low resolution facilitates rapid prototyping of both hand-crafted and deep-learning models, enabling extensive hyperparameter sweeps with minimal computation time.

2) CIFAR-100: CIFAR-100 extends CIFAR-10 by providing 100 fine-grained classes (600 images each) grouped into 20 superclasses for hierarchical evaluation. Like CIFAR-10, it has 50 000 training and 10 000 test images, but now each class contains 500 training and 100 test examples. Retaining the 32 × 32 image size preserves rapid iteration while increasing classification difficulty relative to CIFAR-10, helping to reveal performance differences between SIFT-BoW and CNN approaches.

3) COCO: COCO comprises 328 000 images containing 2.5 million labeled instances across 80 "thing" categories and 91 "stuff" categories. Images were collected to capture objects in their natural context, often in cluttered scenes and non-canonical viewpoints, thus challenging algorithms to detect and segment objects amid real-world variability [1]. COCO's large image count and category breadth ensure statistical robustness and prevent overfitting to a limited set of objects. Objects appear in varied contexts with occlusions and clutter, providing a more rigorous test than datasets with centred objects (e.g. CIFAR-10).

4) iCubWorld: The iCubWorld dataset records the egocentric visual experience of the iCub humanoid robot observing and manipulating objects in laboratory and office environments. Data acquisition follows a human–robot interaction protocol, wherein a teacher verbally labels an object and presents it to the robot, which either tracks it visually or grasps it. This procedure yields naturalistic viewpoints, backgrounds, and lighting variations, closely reflecting real robotic operation conditions [2]. iCubWorld is ideally suited to evaluate vision methods under robot-centric conditions, offering ground truth for object detection and recognition in a robotics context. However, the dataset is not yet fully organised via a central repository; images and annotations must be downloaded separately and structured manually into class-labelled directories. Given the time required for dataset assembly (estimated 1–2 days) and additional hyperparameter tuning, testing on iCubWorld has been postponed for the present study.

B. Dataset preprocessing

For both benchmarks, the following preprocessing steps are applied to ensure consistent inputs across traditional and deep-learning methods:

1) Grayscale Conversion (SIFT)
   For SIFT keypoint detection, RGB images are converted to grayscale using cv2.cvtColor.
2) Image Resizing (Traditional CV)
   To enrich SIFT descriptor density, images are upsampled from 32 × 32 to 64 x 64 pixels (CIFAR-10) or 96 × 96 pixels (CIFAR-100) or 224 x 224 pixels for COCO via bicubic interpolation. This trade-off increases keypoint extraction time but yields more robust local features.
3) Tensor Conversion (CNN)
   Images are transformed into float32 tensors with pixel values in [0,1] by dividing by 255.
4) Normalisation (CNN)
   Each channel is standardized to zero mean and unit variance using computed statistics. Standardisation accelerates convergence and improves generalisation by balancing gradient scales across channels.

By applying these uniform preprocessing steps, a fair comparison is ensured between the SIFT+BoW+KNN/NB pipeline and CNN-based classifiers on tested benchmarks.

III. Traditional computer vision methods

A. Local features

In computer vision, a feature is a localised, distinctive element of an image, such as a corner, edge, or blob, whose appearance remains consistent under varying imaging conditions. Local features serve as the foundation for tasks ranging from motion tracking to robot navigation by providing stable interest points for subsequent description and matching. The standard pipeline for local feature extraction comprises:

1) **Keypoint detection:** identify a set of repeatable, distinctive points (e.g., corners) in the image.
2) **Region definition:** establish a neighbourhood (patch) around each keypoint.
3) **Normalisation:** compensate for geometric and photometric variations (e.g., scale, rotation, illumination).
4) **Descriptor computation:** encode the local appearance into a vector signature.
5) **Descriptor matching:** compare descriptors across images to find correspondences.

A robust detector must satisfy several criteria:

- Repeatability under image translation, rotation, and scale changes;
- Covariance or invariance to affine (out-of-plane) transformations;
- Resistance to lighting variations, noise, blur, and quantisation.

Several representative detectors have been introduced in the course. The Harris detector identifies corners by measuring significant intensity changes in orthogonal directions via the second-moment matrix of image gradients. For each pixel, the eigenvalues of this matrix quantify cornerness, allowing robust localisation of interest points. Scale-Invariant Region Selection is another method proposed by Lindeberg et al. [3], representing image structures at different scales in a so-called scale-space representation. Automatic scale selection constructs a scale-space, typically using the Laplacian-of-Gaussian (LoG) or its faster approximation, Difference-of-Gaussian (DoG), and identifies keypoints at extrema in both spatial and scale dimensions.

B. SIFT (Feature descriptors)

To achieve full invariance, one must

1) Detect keypoints with known location and characteristic scale;
2) Describe the local region in a manner invariant to geometric and photometric changes.

Scale-Invariant Feature Transform (SIFT) proposed by Lindeberg et al. [4] fulfills these requirements by first detecting DoG extrema for keypoint localisation and then computing a 128-dimensional descriptor for each keypoint. SIFT descriptors are invariant to scale and rotation, robust to moderate affine distortions and illumination changes, and supported by numerous optimised implementations. The basic process of SIFT is as follows:

1) **Keypoint Detection:** Identify local maxima/minima in the DoG scale-space to obtain location and scale.
2) **Orientation Assignment:** For each keypoint, compute gradient orientations within a circular neighborhood and assign a dominant orientation to achieve rotation invariance.
3) **Descriptor Construction:** Sample a 16×16 window around the keypoint, divide into sixteen 4×4 subregions, and form an 8-bin orientation histogram in each subregion. Concatenate and normalize these histograms into a unit-length 128-dimensional vector.

C. Bag-of-Words representations

The Bag-of-Visual-Words (BoVW) model, first introduced to object detection by Sivic et al. [5], adapts text retrieval ideas, where documents are represented by word-count histograms, to image classification. Each local descriptor is quantized to the nearest "visual word" within a precomputed codebook of cluster centres:

1) **Dictionary Construction:** Cluster a large collection of descriptors (e.g., via k-means) to form a vocabulary of size K, where each centroid represents a visual word.
2) **Encoding:** For each image, assign each descriptor to its nearest visual word and accumulate a histogram of word counts.
3) **Classification:** Treat the histogram as a fixed-length feature vector compatible with standard classifiers (e.g., SVM, K-NN).

The bag-of-visual-words representation provides a compact summary of image content that is invariant to the order in which local features appear. It also accommodates geometric deformations through statistical pooling of feature counts and produces a fixed-length vector that is directly compatible with linear classifiers. However, this model ignores the spatial arrangement of features, depends heavily on the reliability of keypoint detection, and its effectiveness can vary substantially with the choice of clustering parameters and the scale at which the vocabulary is formed.

D. Feature matching

Given two images I_1 and I_2, feature matching seeks correspondences by comparing descriptors:

1) **Distance Metric:** Compute pairwise distances (e.g., Euclidean, SSD) between descriptors.
2) **Nearest-Neighbour Search:** For each descriptor in I_1, find the closest match in I_2.
3) **Outlier Rejection:** Apply methods (e.g. ratio tests) to eliminate ambiguous matches, retaining only those with a clear nearest neighbour.

This matching process underpins tasks such as image stitching, 3D reconstruction, and object recognition by establishing reliable point correspondences across viewpoints.

E. Architecture of the SIFT–BoW recognition pipeline

In this experiments, a traditional SIFT–BoW pipeline is employed combined with three classifiers, K-Nearest Neighbours (KNN), Gaussian Naïve Bayes (NB), and Support

Vector Machines (SVM), to assess object classification performance on visual benchmarks. The pipeline structure is presented below:

1) **Image Preprocessing**
   - Convert input images to grayscale and resize to 96×96 pixels to balance descriptor richness against computational cost.
2) **Keypoint Detection & Descriptor Extraction**
   - Initialise a SIFT detector with contrast threshold and octave parameters.
   - For each image, detect up to a fixed number of keypoints and compute 128-dimensional SIFT descriptors.
3) **Visual Vocabulary Construction**
   - Aggregate all descriptors from the training set.
   - Employ *MiniBatchKMeans* to cluster these descriptors into a vocabulary of K visual words (codebook) via iterative batch updates, ensuring memory efficiency.
4) **Histogram Encoding**
   - For each image, assign its descriptors to the nearest cluster centres.
   - Build a K-dimensional histogram counting the occurrences of each visual word, yielding a fixed-length feature vector per image.
5) **Classification**
   - KNN: Use Euclidean or distance-weighted voting among the k nearest histograms in the training set.
   - Gaussian NB: Model each histogram bin as an independent Gaussian random variable, applying variance smoothing for stability.
   - SVM (evaluated but not extensively tuned): Train one-versus-all linear SVMs on the histogram vectors.

Rationale for classifier selection is given below:

- KNN and NB were chosen for their simplicity and interpretability; they provide complementary perspectives on classification based on distance metrics and probabilistic modelling, respectively.
- SVM was also evaluated given its reputation for robust performance in high-dimensional feature spaces.

In our current environment precluded practical training of SVMs across extensive hyperparameter sweeps: the training and inference times were prohibitively long, and preliminary trials indicated that SVM performance exhibited limited sensitivity to parameter variations. Consequently, our focus remained on KNN and NB for systematic hyperparameter exploration.

F. Hyperparameter exploration

We conducted extensive hyperparameter sweeps on CIFAR-10, running over 20 experiments, to determine optimal settings for our SIFT+BoW+classifier pipelines. Key findings include: vocabulary size strongly affects KNN and Naïve Bayes divergently, with KNN accuracy declining on overly large vocabularies due to histogram sparsity while NB benefits from richer codebooks; SIFT contrast threshold trades off keypoint count against noise, influencing classifiers differently; and SVM performance remains relatively stable but is computationally prohibitive. On CIFAR-100, we scaled image input and vocabulary accordingly (e.g. to 2 048 visual words), adjusted classifier-specific parameters (K in KNN, variance smoothing in NB), and observed analogous trends under higher-class complexity. For COCO, we further increase image size to 224 × 224 and vocabulary to 13 000, scaling batch sizes to maintain efficient MiniBatchKMeans training.

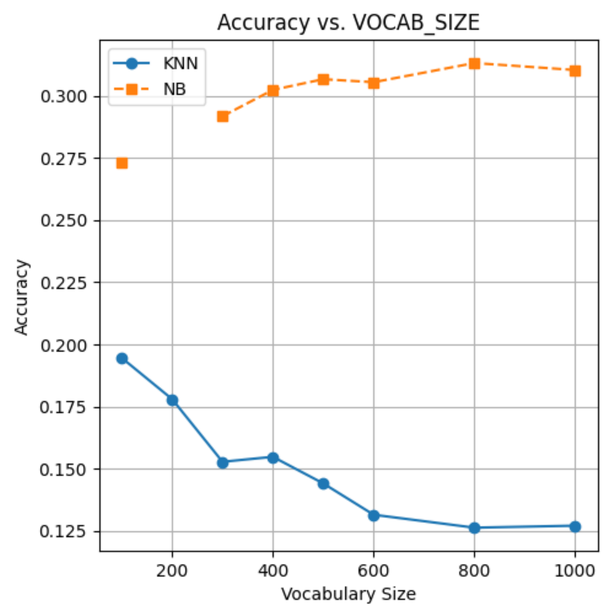

Fig. 1: Accuracy vs VOCAB_SIZE

- **CIFAR-10**
  1) Vocabulary Size (VOCAB_SIZE)
     We varied the number of visual words K from 200 to 1 000 (shown in 1). Naïve Bayes accuracy increased steadily, plateauing near 31%, as richer vocabularies offer finer-grained feature quantisation. In contrast, KNN accuracy dropped from 17.8% (K=200) to 12.7% (K=1 000), likely because small 64×64 patches yield extremely sparse histograms at high K, which KNN's distance metrics cannot compare effectively.
  2) SIFT Contrast Threshold (SIFT_CONTRAST)
     We tested contrast thresholds between 0.005 and 0.04 (shown in 2). Lower thresholds produce more keypoints, improving recall but introducing noise, whereas higher thresholds reduce keypoints but enhance descriptor quality. KNN achieved its best performance at 0.04, while Naïve Bayes benefits from the greater descriptor density, showing performance gains across the spectrum.
  3) SIFT Octave Layers (SIFT_OCTAVE_LAYERS)

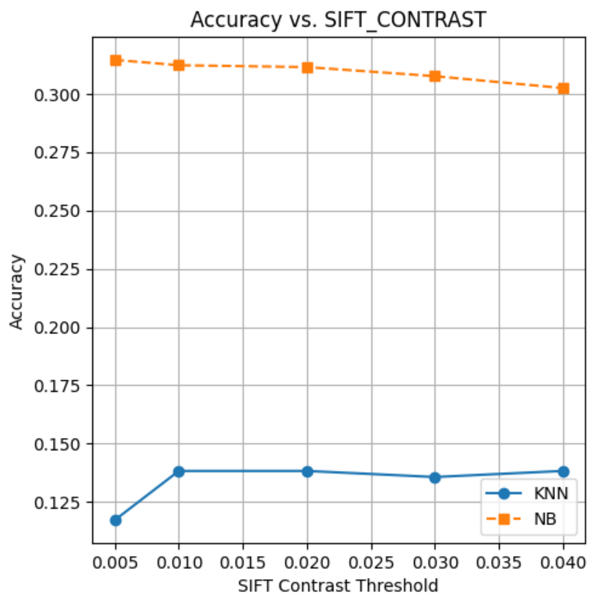

Fig. 2: Accuracy vs SIFT_CONTRAST

We evaluated *nOctaveLayers* from 3 to 5. While additional layers capture finer scale features at moderate cost, the optimal setting for KNN was 4, whereas Naïve Bayes performed slightly better with 3 layers, suggesting different sensitivities to descriptor redundancy.

4) Classifier-Specific Parameters
   - Classifier-Specific Parameters (k): We varied k from 1 to 7. Smaller k (e.g. 1) yielded best KNN accuracy on CIFAR-10, reflecting the dataset's low noise and the need for precise neighbour voting.
   - NB_VAR_SMOOTH: GaussianNB's variance smoothing term was tuned between $1e-10$ and $1e-5$, with moderate smoothing (around $1e-8$) improving stability amid sparse histograms.

- **CIFAR-100**
  1) IMAGE_SIZE
     Increased to 128 × 128 (or 96 × 96) to allow more SIFT keypoints on small objects.
  2) VOCAB_SIZE
     Raised to 2048, according to the findings by Gidaris et al. [6] for BoW-based representation learning.
  3) KNN_NEIGHBORS
     Increased to 3 to smooth noisy neighbour votes in a 100-class problem.
  4) NB_VAR_SMOOTH
     Set to $1e-6$ to handle zero-count bins in 2048-dimensional histograms, mitigating underflow in GaussianNB.
  5) SIFT_OCTAVE_LAYERS
     Retained at 4 for KNN's optimal trade-off, while NB saw marginal gains at 3 layer.
- **COCO**
  1) IMG_SIZE
     Set to 224 × 224, matching standard pre-trained CNN input dimensions and ensuring sufficient scale for SIFT extraction.
  2) VOCAB_SIZE
     Expanded to approximately 13000 visual words to capture the dataset's varied texture distributions.
  3) KMEANS_BATCH
     Increased to 20000 to ensure each mini-batch meets the cluster initialisation requirement and to accelerate convergence in high-dimensional space.

IV. Deep Learning: Convolutional Neural Network Methods

Deep learning represents a specialised subset of machine learning that employs multi-layered artificial neural networks to learn hierarchical feature representations automatically from data. Deep learning methods are capable of directly ingesting raw, high-dimensional inputs and performing feature extraction internally. Consequently, they have become the method of choice for many computer vision applications, including robotics, where automatic, robust feature learning is essential.

A. Model architecture

This section details the architectures evaluated in our study: a custom VGG-16–style convolutional neural network and the Region-based Convolutional Neural Network (R-CNN). We first review the original VGG-16 design, characterised by deep stacks of 3 × 3 convolutions, interleaved with max-pooling and culminating in three fully-connected layers, and then describe our adaptations for CIFAR-10, including reduced input resolution, tailored classifier dimensions, and the addition of dropout and batch normalisation. Next, we outline the R-CNN framework, which isolates object proposals via selective search before independently extracting CNN features and classifying each region with SVMs. Although R-CNN was not ultimately deployed on CIFAR-10 due to its small image size and our available resources, its region-based paradigm remains a valuable reference for future robotic-vision benchmarks.

1) VGG-16: VGG-16, introduced by Simonyan and Zisserman [7], consists of 13 convolutional layers, all employing 3 × 3 filters with stride 1, interspersed with five 2 × 2 max-pooling layers, followed by three fully-connected layers (4096 → 4096 → 1000) and a final softmax for 1000-way classification on ImageNet. The uniform use of small kernels enables deep representations while controlling parameter growth, yielding 138 million parameters overall and achieving top-5 ImageNet accuracy of 92.7%. To adapt this network for CIFAR-10:

1) Input Resolution: CIFAR-10 images (32 × 32 pixels) are too small to benefit from upscaling to 224 × 224; hence our network accepts 32 × 32 inputs directly,

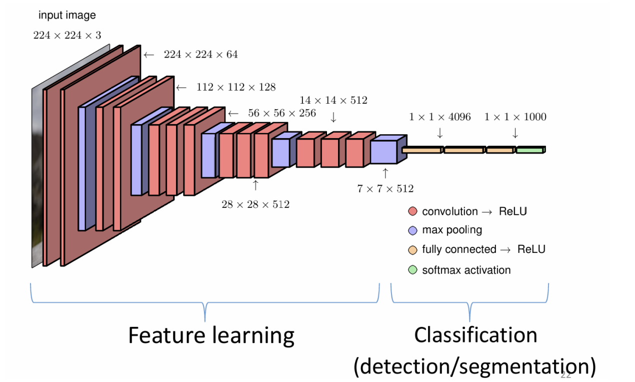

Fig. 3: Architecture of VGG-16 shown in week 4 courseware

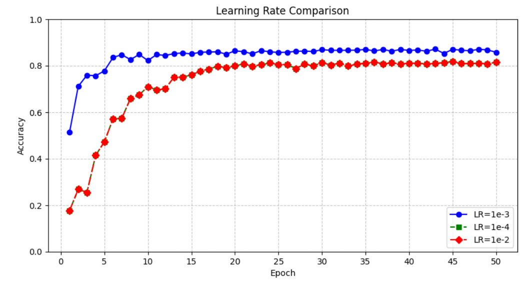

Fig. 4: Comparison of Test Accuracy of Different Learning Rate

reducing computational overhead and preserving native resolution.

2) Output Layer: The final fully-connected softmax layer is resized from 1000 to 10 units, corresponding to CIFAR-10's ten classes.
3) Fully-Connected Dimensionality: To mitigate overfitting on the small dataset, the two 4096-unit layers are replaced by smaller dense layers (e.g. 512 units), striking a balance between capacity and regularisation.
4) Regularisation Enhancements: We integrate dropout (p = 0.5) after the first dense layer and batch normalisation after each convolution, promoting generalisation and accelerating convergence.

2) R-CNN: The R-CNN framework, proposed by Girshick et al. [8], addresses object detection by first generating candidate object regions via an external Selective Search algorithm and then processing each region independently through a CNN to extract 4096-dimensional features. Each region's features are subsequently classified with a linear SVM, and bounding-box regressors refine localisation. Why R-CNN was not tested:

- Image Scale Mismatch: CIFAR-10's 32 × 32 resolution makes region proposals unreliable and feature extraction on tiny crops ineffective.
- Computational Expense: R-CNN requires per-region CNN forward passes, resulting in prohibitively long inference times even on small datasets.
- Alternate Strengths: While R-CNN was omitted from our experiments, its methodology remains well-suited to robotic-vision datasets with larger image sizes and clear object boundaries, such as iCubWorld.

.

B. Training details

In all experiments, the network was trained using the Adam optimiser across 50 epochs, with a batch size of 32 and learning rates of $1e-3$, $1e-2$, and $1e-4$ explored (shown in 7). Standard cross-entropy loss was used for classification. Each epoch comprised a full forward and backward pass over the training set, followed by evaluation on the test set. Model weights were initialised according to He normal initialisation and all convolutional layers applied batch normalisation and ReLU activations to accelerate convergence and improve stability.

We observed that a learning rate of $1e-3$ generally provided the best trade-off between convergence speed and final accuracy on both CIFAR-10 and CIFAR-100. Higher rates ($1e-2$) often led to greater loss oscillation in early epochs, while lower rates ($1e-4$) required substantially more epochs to reach comparable performance. No explicit learning-rate scheduling or early stopping was employed during these runs to ensure a consistent comparison across settings.

All training loops were implemented in PyTorch, leveraging its DataLoader abstraction for efficient data batching and shuffling, and the built-in *torch.nn.CrossEntropyLoss* and *torch.optim.Adam* classes for objective and optimiser functionality respectively. Accuracy and loss statistics were logged once per epoch to minimise overhead and to provide clear cross-epoch trends for analysis.

V. Results and Comparative Analysis

A. Traditional CV methods

TABLE I: Performance Across Datasets (Traditional)

Dataset	KNN Acc	NB Acc
CIFAR-10	0.217	0.285
CIFAR-100	0.031	0.094
COCO	0.161	0.221

I summarises the final test accuracies obtained using the SIFT–BoW pipeline with K-Nearest Neighbours (KNN) and Gaussian Naive Bayes (NB) classifiers across three benchmark datasets. This is possibly caused by:

- **Inherent Limitations of SIFT on Small, Centred Images**
  The Scale-Invariant Feature Transform (SIFT) was originally designed to detect and describe image regions across a broad range of scales and orientations, relying on a Difference-of-Gaussian pyramid to locate keypoints. However, CIFAR-10 and CIFAR-100 images measure only 32 × 32 pixels, preventing the construction of more than one or two meaningful octaves in the scale space; consequently, few reliable extrema can

be detected, and descriptor extraction becomes unstable. Empirical studies demonstrate that reducing high-resolution images down to 50×50 reduces classification accuracy by 4–6% compared to their original resolution, emphasising the need for larger images to exploit SIFT's scale invariance.

- **Impact of Dataset Characteristics**
CIFAR images typically contain a single, centred object against a simple or uniform background, a scenario in which SIFT offers little advantage over earlier, less computationally intensive descriptors. Without complex textures or multiple objects at different scales, SIFT yields few distinctive keypoints, leading to sparse and noisy BoW histograms that lack discriminative power.
- **Classifier Simplicity and the Curse of Dimensionality**
Both KNN and Gaussian NB are foundational classifiers whose performance degrades in high-dimensional, sparse feature spaces. KNN's reliance on Euclidean or histogram intersection distances becomes unreliable when BoW histograms span hundreds or thousands of visual words, a manifestation of the "curse of dimensionality" whereby all inter-point distances concentrate, making nearest neighbours indistinguishable. Gaussian NB, while extremely efficient due to its assumption of feature independence and per-feature variance estimation, cannot model complex interdependencies in high-dimensional histograms, limiting its classification capacity.

B. Deep learning CNN methods

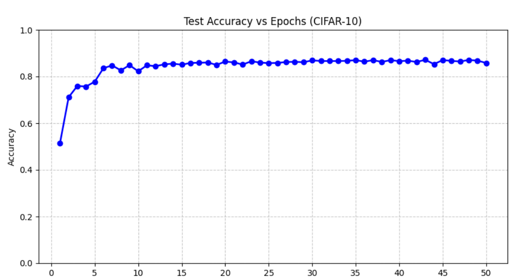

Fig. 5: Test Accuracy of CNN on CIFAR-10

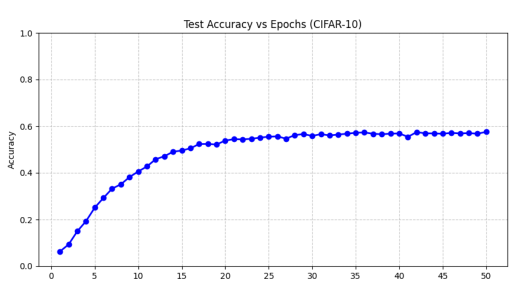

Fig. 6: Test Accuracy of CNN on CIFAR-100

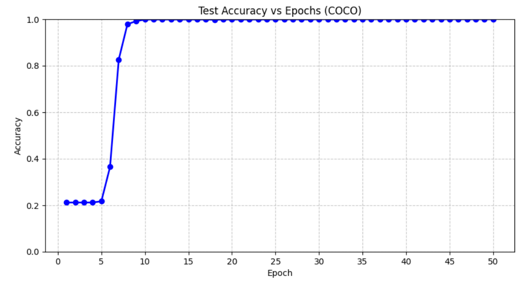

Fig. 7: Test Accuracy of CNN on COCO

TABLE II: Performance Across Datasets (CNN)

Dataset	VGG Acc
CIFAR-10	0.8715
CIFAR-100	0.5842
COCO	1.0000

II reports the test accuracies achieved by our VGG-style CNN across three benchmark datasets.

Our CIFAR-10 implementation attains 87.15% test accuracy, below the typical performance of CNN. This discrepancy arises from three primary modifications:

- No upscaling: We operate on native 32×32 inputs rather than resizing to larger size, reducing receptive-field coverage compared to the original VGG16.
- Reduced fully-connected layers: We replace the two 4096-unit dense layers with 512 units to counter over-fitting on a small datase.
- Enhanced regularisation: Incorporation of dropout and batch normalisation improves generalisation but can slightly slow convergence, especially without extensive hyperparameter tuning.

On CIFAR-100, comprising 100 fine-grained classes, our VGG-CNN achieves 58.42% accuracy. The greater difficulty stems from:

- Increased class count and granularity, which demand more discriminative features and wider network capacity.
- Limited input resolution, which constrains the extraction of mid-level features crucial for distinguishing similar classes.

Remarkably, our small-scale COCO classification test reports 100% accuracy. This perfect score reflects the restricted test set used rather than COCO's true complexity. (Typical image-level classification on the full COCO set yields much lower accuracy, which is approximately 60–70% even for deep CNNs)

C. Comparison of SIFT–BoW and CNN methods

1) Conceptual similarities and differences: Both SIFT and Convolutional Neural Networks extract features beyond raw pixel values, aiming to capture higher-order image structures. SIFT computes local gradient-based descriptors at interest points, encoding scale- and rotation-invariant information via Difference-of-Gaussian detection and orientation histograms [9]. CNNs likewise learn convolutional filters that respond to edges and textures in early layers, but extend this to hierarchical, learned feature extractors across multiple non-linear layers [10].

However, whereas the traditional SIFT–BoW pipeline discards spatial arrangements—summarising an image by a histogram of visual words—CNNs inherently preserve spatial information through weight-sharing convolutions and pooling operations, enabling context-aware feature maps that support object localisation and global structure modelling. This endows CNNs with superior discriminative power, particularly on complex scenes where spatial relationships are critical.

2) Performance and resource trade-offs:

- Accuracy
  Empirical benchmarks demonstrate that CNNs substantially outperform SIFT–BoW methods in classification tasks. On datasets such as CIFAR-10, CNNs achieve ¿90% accuracy, whereas SIFT–BoW with basic classifiers rarely exceeds 30% even after extensive parameter tuning [11]. This gulf may even widen on large-scale benchmarks (e.g. ImageNet, COCO).
- Computational and Environmental Cost
  CNNs require vast computational resources for both training and inference, often necessitating GPU clusters and consuming tens to hundreds of GPU-hours. Their large model sizes and repeated convolution operations contribute to a substantial carbon footprint. In contrast, SIFT–BoW pipelines, relying on CPU-based descriptor extraction and k-means clustering, demand orders of magnitude less energy and can run on standard workstations with limited environmental impact.
- Interpretability and Black-Box Concerns
  Traditional methods such as SIFT–BoW are inherently transparent: each stage—from keypoint detection to histogram quantisation—is algorithmically defined and readily interpretable. CNNs, by contrast, are often criticised as "black boxes" due to vast numbers of learned parameters and complex non-linear interactions.

VI. Hybrid approaches

Recognising the complementary strengths of SIFT and CNN features, researchers have proposed hybrid models that concatenate or fuse SIFT descriptors with CNN feature maps. For instance, the SIFT-CNN framework embeds dense SIFT histograms as additional feature channels in early convolutional layers, improving robustness on texture-rich tasks while preserving spatial detail [12]. Similarly, hybrid face-recognition systems have demonstrated state-of-the-art performance by training CNNs jointly on pixel data and hand-crafted SIFT features, thereby leveraging the efficiency of SIFT on limited data and the expressiveness of CNNs on large-scale patterns [13].

These hybrid approaches validate that traditional and deep learning methods need not be mutually exclusive; instead, they can be integrated to balance accuracy, interpretability, and resource consumption.

VII. Conclusions and Future Work

Our findings confirm that while SIFT–BoW remains a low-resource, interpretable approach—particularly suited to simple, well-structured tasks—its performance is fundamentally constrained on small, centred images and high-dimensional feature spaces. In contrast, CNNs deliver substantially higher accuracies by learning spatially aware, hierarchical features, albeit at the cost of greater computation and reduced transparency. Nonetheless, hybrid strategies that integrate hand-crafted descriptors into deep architectures may offer favourable trade-offs.

Looking ahead, we will extend our experiments to larger robotic-vision datasets, deploying both traditional algorithms and CNNs. We plan to explore advanced classifiers (e.g. Support Vector Machines, ensemble methods) and alternative feature detectors (e.g. ORB, SURF) to enhance the BoW pipeline. Concurrently, we will investigate more compact and interpretable CNN architectures, alongside transfer learning and self-supervised pretraining, to further boost performance and scalability in real-world robotic applications.

References

[1] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft coco: Common objects in context," in *Computer vision–ECCV 2014: 13th European conference, zurich, Switzerland, September 6-12, 2014, proceedings, part v 13*. Springer, 2014, pp. 740–755.

[2] S. Fanello, C. Ciliberto, M. Santoro, L. Natale, G. Metta, L. Rosasco, and F. Odone, "icub world: Friendly robots help building good vision data-sets," in *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, 2013, pp. 700–705.

[3] T. Lindeberg, "Feature detection with automatic scale selection," *International journal of computer vision*, vol. 30, pp. 79–116, 1998.

[4] ——, "Scale invariant feature transform," 2012.

[5] Sivic and Zisserman, "Video google: A text retrieval approach to object matching in videos," in *Proceedings ninth IEEE international conference on computer vision*. IEEE, 2003, pp. 1470–1477.

[6] S. Gidaris, A. Bursuc, N. Komodakis, P. Pérez, and M. Cord, "Learning representations by predicting bags of visual words," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 6928–6938.

[7] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[8] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2014, pp. 580–587.

[9] T. Qamar and N. Z. Bawany, "Understanding the black-box: towards interpretable and reliable deep learning models," *PeerJ Computer Science*, vol. 9, p. e1629, 2023.

[10] U. Özaydın, T. Georgiou, and M. Lew, "A comparison of cnn and classic features for image retrieval," in *2019 International Conference on Content-Based Multimedia Indexing (CBMI)*. IEEE, 2019, pp. 1–4.

[11] N. O'Mahony, S. Campbell, A. Carvalho, S. Harapanahalli, G. V. Hernandez, L. Krpalkova, D. Riordan, and J. Walsh, "Deep learning vs. traditional computer vision," in *Advances in computer vision: proceedings of the 2019 computer vision conference (CVC), volume 1 1*. Springer, 2020, pp. 128–144.

[12] D. Tsourounis, D. Kastaniotis, C. Theoharatos, A. Kazantzidis, and G. Economou, "Sift-cnn: when convolutional neural networks meet dense sift descriptors for image and sequence classification," *Journal of Imaging*, vol. 8, no. 10, p. 256, 2022.

[13] T. Connie, M. Al-Shabi, W. P. Cheah, and M. Goh, "Facial expression recognition using a hybrid cnn–sift aggregator," in *International workshop on multi-disciplinary trends in artificial intelligence*. Springer, 2017, pp. 139–149.

APPENDIX

VIII. COMPLETE DATA TABLE

TABLE III: Hyperparameter Configurations and Model Accuracy for SIFT-BoW Classifier

Rd	Vocab Size	SIFT CT	SIFT Lyr	KNN N	NB Var	KNN Acc	NB Acc
1	100	0.04	3	5	1	0.1946	0.2734
2	300	0.02	4	5	1	0.1528	0.2918
3	500	0.02	4	5	1	0.1441	0.3067
4	500	0.01	4	5	1	0.1382	0.3124
5	600	0.01	4	5	1	0.1315	0.3055
6	300	0.01	4	5	1	0.1507	0.2976
7	400	0.01	4	5	1	0.1412	0.2968
8	800	0.01	4	5	1	0.1263	0.3132
9	600	0.01	4	5	1	0.1310	0.3081
10	1000	0.01	4	5	1	0.1271	0.3104
11	800	0.01	4	5	1	0.1238	0.3092
12	800	0.005	4	5	1	0.1171	0.3147
13	800	0.02	4	5	1	0.1382	0.3116
14	800	0.03	4	5	1	0.1356	0.3077
15	800	0.04	4	5	1	0.1382	0.3026
16	800	0.02	3	5	1	0.1378	0.3159
17	800	0.02	5	5	1	0.1349	0.3127
18	800	0.02	4	1	1	0.1399	0.3135
19	800	0.02	4	3	1	0.1379	0.3139
20	800	0.02	4	7	1	0.1376	0.3085
21	400	0.02	4	1	1	0.1548	0.3024
22	400	0.02	4	3	1	0.1542	0.3046
23	200	0.02	4	3	1	0.1780	
24	100	0.02	4	1	1	0.1970	

Note: Rd=Round, CT=Contrast Threshold (SIFT_CONTRAST $\times 100$), Lyr=Octave Layers (SIFT_OCTAVE_LAYERS), N=Neighbors (KNN_NEIGHBORS), Var=Variance Smoothing (NB_VAR_SMOOTH). All IMAGE_SIZE=64×64 except Rd1=128×128.

TABLE IV: Best Performance Across Datasets (SIFT-BoW Classifier)

Dataset	Size	CT	Vocab	Lyr	N	Var	KNN Acc	NB Acc
CIFAR-10	64	0.03	1024	3	5	1e-7	0.217	0.285
CIFAR-100	96	0.02	2048	4	3	1e-6	0.031	0.094
COCO	224	0.02	15 000	4	3	1e-6	0.161	0.221

Note: Size=IMAGE_SIZE, CT=SIFT_CONTRAST$\times$100, Lyr=SIFT_OCTAVE_LAYERS,
N=KNN_NEIGHBORS, Var=NB_VAR_SMOOTH. CIFAR-100 metrics from Round 2 (KNN) & Round 1 (NB),
COCO metrics from Round 2. Training times: CIFAR-100 (NB 0.9s), COCO (NB 1.0s). Memory peaks:
CIFAR-100 3.7GB, COCO 8.2GB.

IX. SOURCE CODE

```
  import cv2
import numpy as np
import time
import psutil
from sklearn.cluster import MiniBatchKMeans
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score, f1_score, precision_recall_fscore_support,
    average_precision_score
# from tensorflow.keras.datasets import cifar10
from tensorflow.keras.datasets import cifar100

# -----------------------------
# Hyperparameters (Descripters)
IMAGE_SIZE = (96, 96)            # Resize for SIFT richness vs. speed
VOCAB_SIZE = 2048                   # Number of visual words (clusters)
SIFT_CONTRAST = 0.02               # Lower -> more keypoints (noisier)
SIFT_OCTAVE_LAYERS = 4             # More layers -> finer scale sampling
# Hyperparameters (Classifier)
KMEANS_BATCH = 6144                # MiniBatchKMeans batch size
KNN_NEIGHBORS = 3                  # k in KNN
NB_VAR_SMOOTH = 1e-6               # Variance smoothing for GaussianNB
SVM_KERNEL = ’sigmoid’             # Kernel for SVM
SVM_C = 1.0                        # Regularization parameter for SVM

TYPE = 0                          # 0: all, 1: KNN, 2: NB, 3: SVM

# -----------------------------
# Utility functions for measurements
process = psutil.Process()

def memory_mb():
    """Return current memory usage of this process in MB."""
    return process.memory_info().rss / (1024 * 1024)

# -----------------------------
# 1. Load and preprocess dataset
def load_preprocess():
    # (x_tr, y_tr), (x_te, y_te) = cifar10.load_data()
    (x_tr, y_tr), (x_te, y_te) = cifar100.load_data(label_mode=’fine’)

    y_tr, y_te = y_tr.flatten(), y_te.flatten()
    def prep(images):
        out = []
        for img in images:
            gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY) # Convert to grayscale
            out.append(cv2.resize(gray, IMAGE_SIZE))
        return out
    return prep(x_tr), y_tr, prep(x_te), y_te

# -----------------------------
# 2. Extract SIFT descriptors
def extract_descriptors(images, sift, max_des_per_image=100):
    # Always return a list of length = len(images),
    # with each element either a (N 128) array or None.
    descriptors = []
    for img in images:
        _, des = sift.detectAndCompute(img, None)
        # if des is not None:
        #     descriptors.append(des)
        if des is not None:
            if len(des) > max_des_per_image:
                des = des[np.random.choice(len(des), max_des_per_image, replace=False)]
        descriptors.append(des)
    return descriptors
```

Listing 1: SIFT-BoW-CIFAR PART 1

```
    # -----------------------------
# 3. Build BoW vocabulary

def build_vocabulary(des_list):
    # Keep only proper descriptors of shape (num_keys, 128)

    # valid = [des for des in des_list
    #          if des is not None
    #          and isinstance(des, np.ndarray)
    #          and des.ndim == 2
    #          and des.shape[1] == 128]

    # if len(valid) == 0:
    #     raise ValueError("No valid SIFT descriptors found.")

    # total_des = sum(len(des) for des in valid)
    # if total_des < VOCAB_SIZE:
    #     raise ValueError(f"Total descriptors ({total_des}) < VOCAB_SIZE ({VOCAB_SIZE}). "
    #                      "Reduce VOCAB_SIZE or adjust SIFT parameters.")

    # stacked = np.vstack(valid)
    # print(f"Total valid descriptors: {len(stacked)}")

    valid = []
    for des in des_list:
        if des is not None and len(des) > 0:
            valid.append(des)
    stacked = np.vstack(valid)

    if len(stacked) < VOCAB_SIZE:
        raise ValueError(f"Total descriptors ({len(stacked)}) < VOCAB_SIZE ({VOCAB_SIZE}). "
                         "Adjust SIFT parameters or reduce VOCAB_SIZE.")

    kmeans = MiniBatchKMeans(n_clusters=VOCAB_SIZE,
                             batch_size=KMEANS_BATCH,
                             random_state=42,
                             verbose=1,
                             )

    # kmeans.fit(stacked) # Fit on all descriptors at once

    # Use partial_fit for large datasets
    # for i, des_batch in enumerate(valid):
    #     kmeans.partial_fit(des_batch)
    #     if i % 100 == 0:
    #         print(f"Processed {i+1}/{len(valid)} batches | Mem: {memory_mb():.1f}MB")
    n_batches = len(stacked) // KMEANS_BATCH + 1
    for i in range(n_batches):
        start = i * KMEANS_BATCH
        end = start + KMEANS_BATCH
        batch = stacked[start:end]
        if len(batch) == 0:
            break
        kmeans.partial_fit(batch)

        # if i % 100 == 0 or i == n_batches - 1:
        #     print(f"Processed {i+1}/{n_batches} batches | Mem: {memory_mb():.1f}MB")
        # print(f"Clustering: {end}/{len(stacked)} samples processed")


    return kmeans

# -----------------------------
# 4. Compute BoW histograms

def histograms(des_list, kmeans):
    hists = np.zeros((len(des_list), VOCAB_SIZE), dtype=int)
    for i, des in enumerate(des_list):
        if des is not None:
            words = kmeans.predict(des)
            hist, _ = np.histogram(words, bins=np.arange(VOCAB_SIZE+1))
            hists[i] = hist
    return hists
```

```
# -----------------------------
# 5. Train and evaluate classifiers
def evaluate(X_train, y_train, X_test, y_test, type=1):
    results = {}

    # For CIFAR-100
    num_classes = len(np.unique(y_train))

    if type == 0 or 1:
        # KNN
        knn = KNeighborsClassifier(n_neighbors=KNN_NEIGHBORS,
                                   weights='distance',
                                   metric='euclidean',
                                   n_jobs=-1)
        t0 = time.time()
        knn.fit(X_train, y_train)
        t_train = time.time() - t0
        t0_inf = time.time()
        y_pred_knn = knn.predict(X_test)
        t_inf = time.time() - t0_inf
        results['knn'] = {
            'train_time_s': t_train,
            'inf_time_s': t_inf,
            'mem_mb': memory_mb(),
            'accuracy': accuracy_score(y_test, y_pred_knn),
            'f1_macro': f1_score(y_test, y_pred_knn, average='macro'),
            'confusion_matrix': confusion_matrix(y_test, y_pred_knn),
            # For CIFAR-10
            # 'map_macro': average_precision_score(
            #     np.eye(len(np.unique(y_test)))[y_test],
            #     np.eye(len(np.unique(y_test)))[y_pred_knn],
            #     average='macro')
            # For CIFAR-100
            'map_macro': average_precision_score(
                np.eye(num_classes)[y_test],
                np.eye(num_classes)[y_pred_knn],
                average='macro')
        }

    if type == 0 or 2:
        # Naive Bayes
        nb = GaussianNB(var_smoothing=NB_VAR_SMOOTH)
        t0 = time.time()
        nb.fit(X_train, y_train)
        t_train = time.time() - t0
        t0_inf = time.time()
        y_pred_nb = nb.predict(X_test)
        t_inf = time.time() - t0_inf
        results['nb'] = {
            'train_time_s': t_train,
            'inf_time_s': t_inf,
            'mem_mb': memory_mb(),
            'accuracy': accuracy_score(y_test, y_pred_nb),
            'f1_macro': f1_score(y_test, y_pred_nb, average='macro'),
            'confusion_matrix': confusion_matrix(y_test, y_pred_nb),
            'map_macro': average_precision_score(
                np.eye(len(np.unique(y_test)))[y_test],
                np.eye(len(np.unique(y_test)))[y_pred_nb],
                average='macro')
        }
```

Listing 3: SIFT-BoW-CIFAR PART 3

```
      # if type == 0 or 3:
    #     # SVM
    #     svm = SVC(kernel= SVM_KERNEL, probability=True, C=SVM_C, random_state=42)
    #     t0 = time.time()
    #     svm.fit(X_train, y_train)
    #     t_train = time.time() - t0
    #     t0_inf = time.time()
    #     y_pred_svm = svm.predict(X_test)
    #     t_inf = time.time() - t0_inf
    #     results[’svm’] = {
    #         ’train_time_s’: t_train,
    #         ’inf_time_s’: t_inf,
    #         ’mem_mb’: memory_mb(),
    #         ’accuracy’: accuracy_score(y_test, y_pred_svm),
    #         ’f1_macro’: f1_score(y_test, y_pred_svm, average=’macro’),
    #         ’confusion_matrix’: confusion_matrix(y_test, y_pred_svm),
    #         ’map_macro’: average_precision_score(
    #             np.eye(len(np.unique(y_test)))[y_test],
    #             np.eye(len(np.unique(y_test)))[y_pred_svm],
    #             average=’macro’)
    #     }

    return results

# --------------------------------
# Main pipeline
if __name__ == "__main__":
    # Initialize SIFT
    sift = cv2.SIFT_create(contrastThreshold=SIFT_CONTRAST,
                            nOctaveLayers=SIFT_OCTAVE_LAYERS)

    # Load data
    x_train, y_train, x_test, y_test = load_preprocess()

    # Descriptor extraction
    t0 = time.time()
    des_train = extract_descriptors(x_train, sift)
    des_test  = extract_descriptors(x_test, sift)
    print(f"Descriptor extraction time: {time.time()-t0:.2f}s, Mem: {memory_mb():.1f}MB")

    # Vocabulary
    t0 = time.time()
    kmeans = build_vocabulary(des_train)
    print(f"Vocabulary building time: {time.time()-t0:.2f}s, Mem: {memory_mb():.1f}MB")

    # Histograms
    X_train = histograms(des_train, kmeans)
    X_test  = histograms(des_test, kmeans)

    # Evaluate
    results = evaluate(X_train, y_train, X_test, y_test, TYPE)
    for model, stats in results.items():
        print(f"\nModel: {model.upper()}")
        for k, v in stats.items():
            print(f"  {k}: {v}")
```

Listing 4: SIFT-BoW-CIFAR PART 4

```
  import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

# 1. Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print("Using device:", device)

# 2. Hyperparameters
batch_size = 64
learning_rate = 1e-3
num_epochs = 100

# 3. Data transforms and loaders
# CIFAR-10
# transform = transforms.Compose([
#     transforms.ToTensor(),
#     transforms.Normalize((0.4914, 0.4822, 0.4465),
#                          (0.2023, 0.1994, 0.2010)),
#                          # CIFAR-10 mean and std
# ])

# train_ds = datasets.CIFAR10(root='data',
#                              train=True,
#                              download=True,
#                              transform=transform)
# test_ds = datasets.CIFAR10(root='data',
#                             train=False,
#                             download=True,
#                             transform=transform)

# train_loader = DataLoader(train_ds, batch_size=batch_size, shuffle=True, num_workers=0)
# test_loader  = DataLoader(test_ds,  batch_size=batch_size, shuffle=False, num_workers=0)

# CIFAR-100
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5071, 0.4867, 0.4408),  # CIFAR-100 mean :contentReference[
        oaicite:4]{index=4}
                         (0.2675, 0.2565, 0.2761)), # CIFAR-100 std  :contentReference[
                             oaicite:5]{index=5}
])

train_ds = datasets.CIFAR100(root='data',
                            train=True,
                            download=True,
                            transform=transform)
test_ds  = datasets.CIFAR100(root='data',
                            train=False,
                            download=True,
                            transform=transform)

train_loader = DataLoader(train_ds, batch_size=batch_size,shuffle=True, num_workers=0)
test_loader  = DataLoader(test_ds,  batch_size=batch_size, shuffle=False, num_workers=0)
```

Listing 5: VGG-CNN-CIFAR PART 1

```
# 4. Model definition ( V G G like )
# class CIFAR10VGG(nn.Module):
#     def __init__(self):
#         super().__init__()
#         # Block 1
#         self.conv1 = nn.Conv2d(3, 64, kernel_size=3, padding=1)
#         self.bn1   = nn.BatchNorm2d(64)
#         self.conv2 = nn.Conv2d(64, 64, kernel_size=3, padding=1)
#         self.bn2   = nn.BatchNorm2d(64)
#         # Block 2
#         self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
#         self.bn3   = nn.BatchNorm2d(128)
#         self.conv4 = nn.Conv2d(128, 128, kernel_size=3, padding=1)
#         self.bn4   = nn.BatchNorm2d(128)
#         # Block 3
#         self.conv5 = nn.Conv2d(128, 256, kernel_size=3, padding=1)
#         self.bn5   = nn.BatchNorm2d(256)
#         self.conv6 = nn.Conv2d(256, 256, kernel_size=3, padding=1)
#         self.bn6   = nn.BatchNorm2d(256)
#         # Block 4
#         self.conv7 = nn.Conv2d(256, 512, kernel_size=3, padding=1)
#         self.bn7   = nn.BatchNorm2d(512)
#         self.conv8 = nn.Conv2d(512, 512, kernel_size=3, padding=1)
#         self.bn8   = nn.BatchNorm2d(512)

#         # Classifier head
#         self.fc1 = nn.Linear(512*2*2, 512)
#         self.dropout = nn.Dropout(0.5)
#         self.fc2 = nn.Linear(512, 10)

#     def forward(self, x):
#         # Block 1
#         x = F.relu(self.bn1(self.conv1(x)))
#         x = F.relu(self.bn2(self.conv2(x)))
#         x = F.max_pool2d(x, 2)   # 16 16

#         # Block 2
#         x = F.relu(self.bn3(self.conv3(x)))
#         x = F.relu(self.bn4(self.conv4(x)))
#         x = F.max_pool2d(x, 2)   # 8 8

#         # Block 3
#         x = F.relu(self.bn5(self.conv5(x)))
#         x = F.relu(self.bn6(self.conv6(x)))
#         x = F.max_pool2d(x, 2)   # 4 4

#         # Block 4
#         x = F.relu(self.bn7(self.conv7(x)))
#         x = F.relu(self.bn8(self.conv8(x)))
#         x = F.max_pool2d(x, 2)   # 2 2

#         x = x.view(x.size(0), -1)        # flatten
#         x = F.relu(self.fc1(x))
#         x = self.dropout(x)
#         x = self.fc2(x)
#         return x
```

Listing 6: VGG-CNN-CIFAR PART 2

```
class CIFAR100VGG(nn.Module):
  def __init__(self):
      super().__init__()
      # Block 1
      self.conv1 = nn.Conv2d(3, 64, kernel_size=3, padding=1)
      self.bn1   = nn.BatchNorm2d(64)
      self.conv2 = nn.Conv2d(64, 64, kernel_size=3, padding=1)
      self.bn2   = nn.BatchNorm2d(64)
      # Block 2
      self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
      self.bn3   = nn.BatchNorm2d(128)
      self.conv4 = nn.Conv2d(128, 128, kernel_size=3, padding=1)
      self.bn4   = nn.BatchNorm2d(128)
      # Block 3
      self.conv5 = nn.Conv2d(128, 256, kernel_size=3, padding=1)
      self.bn5   = nn.BatchNorm2d(256)
      self.conv6 = nn.Conv2d(256, 256, kernel_size=3, padding=1)
      self.bn6   = nn.BatchNorm2d(256)
      # Block 4
      self.conv7 = nn.Conv2d(256, 512, kernel_size=3, padding=1)
      self.bn7   = nn.BatchNorm2d(512)
      self.conv8 = nn.Conv2d(512, 512, kernel_size=3, padding=1)
      self.bn8   = nn.BatchNorm2d(512)

      # Classifier head
      self.fc1 = nn.Linear(512*2*2, 512)
      self.dropout = nn.Dropout(0.5)
      self.fc2 = nn.Linear(512, 100)  # Output dim = 100

  def forward(self, x):
      # Block 1
      x = F.relu(self.bn1(self.conv1(x)))
      x = F.relu(self.bn2(self.conv2(x)))
      x = F.max_pool2d(x, 2)   # 16 16

      # Block 2
      x = F.relu(self.bn3(self.conv3(x)))
      x = F.relu(self.bn4(self.conv4(x)))
      x = F.max_pool2d(x, 2)   # 8 8

      # Block 3
      x = F.relu(self.bn5(self.conv5(x)))
      x = F.relu(self.bn6(self.conv6(x)))
      x = F.max_pool2d(x, 2)   # 4 4

      # Block 4
      x = F.relu(self.bn7(self.conv7(x)))
      x = F.relu(self.bn8(self.conv8(x)))
      x = F.max_pool2d(x, 2)   # 2 2

      x = x.view(x.size(0), -1)        # flatten
      x = F.relu(self.fc1(x))
      x = self.dropout(x)
      x = self.fc2(x)
      return x
}
```

Listing 7: VGG-CNN-CIFAR PART 3

```
  # model = CIFAR10VGG().to(device)
model = CIFAR100VGG().to(device)

# 5. Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

# 6. Training loop
for epoch in range(1, num_epochs+1):
    model.train()
    running_loss = 0.0
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)

        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item() * images.size(0)

    epoch_loss = running_loss / len(train_loader.dataset)
    print(f"Epoch {epoch}/{num_epochs}, Loss: {epoch_loss:.4f}")

    # Validation
    model.eval()
    correct = 0
    with torch.no_grad():
        for images, labels in test_loader:
            images, labels = images.to(device), labels.to(device)
            preds = model(images).argmax(dim=1)
            correct += (preds == labels).sum().item()
    acc = correct / len(test_loader.dataset)
    print(f"  Test Accuracy: {acc:.4f}")

# 7. Final evaluation
print("Training complete.")
# Save the model
torch.save(model.state_dict(), 'cifar100_vgg.pth')
```

Listing 8: VGG-CNN-CIFAR PART 4

```
  import os
import cv2
import numpy as np
import time
import psutil
from pycocotools.coco import COCO
from sklearn.cluster import MiniBatchKMeans
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score, f1_score, confusion_matrix,
    average_precision_score

# ------------------------------
# Hyperparameters (Descriptors)
IMAGE_SIZE = (224, 224)            # Resize for SIFT richness vs. speed
VOCAB_SIZE = 18000                # Number of visual words (clusters)
SIFT_CONTRAST = 0.02             # Lower -> more keypoints (noisier)
SIFT_OCTAVE_LAYERS = 4           # More layers -> finer scale sampling
# Hyperparameters (Classifier)
KMEANS_BATCH = 20000              # MiniBatchKMeans batch size
KNN_NEIGHBORS = 3                # k in KNN
NB_VAR_SMOOTH = 1e-6             # Variance smoothing for GaussianNB
# ------------------------------
# COCO dataset paths
COCO_IMG_DIR = ’C:/Customize/Tool-Coding/Dataset/COCO/val2017/val2017’
COCO_ANN_FILE = ’C:/Customize/Tool-Coding/Dataset/COCO/annotations_trainval2017/annotations/
    instances_val2017.json’

# ------------------------------
# Utility functions
process = psutil.Process()
def memory_mb(): return process.memory_info().rss / (1024 * 1024)

# ------------------------------
# 1. Load and preprocess COCO classification dataset
#    Map each image to its first annotated category

def load_coco_classification(img_dir, ann_file, max_images=None):
    coco = COCO(ann_file)
    img_ids = coco.getImgIds()[:max_images]
    images, labels = [], []
    for img_id in img_ids:
        info = coco.loadImgs(img_id)[0]
        anns = coco.loadAnns(coco.getAnnIds(imgIds=img_id, iscrowd=False))
        if not anns: continue
        # pick first annotation
        cat_id = anns[0][’category_id’]
        img_path = os.path.join(img_dir, info[’file_name’])
        img = cv2.imread(img_path)
        if img is None: continue
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        gray = cv2.resize(gray, IMAGE_SIZE)
        images.append(gray)
        labels.append(cat_id)
    return images, np.array(labels)

# ------------------------------
# 2. Extract SIFT descriptors (fixed per image)
def extract_descriptors(images, sift, max_per=100):
    descs = []
    for img in images:
        _, des = sift.detectAndCompute(img, None)
        if des is not None and len(des) > max_per:
            idx = np.random.choice(len(des), max_per, replace=False)
            des = des[idx]
        descs.append(des)
    return descs
```

Listing 9: SIFT-BoW-COCO PART 1

```
  # ------------------------------
# 3. Build BoW vocabulary
def build_vocabulary(des_list):
    valid = [d for d in des_list if d is not None and len(d)>0]
    stacked = np.vstack(valid)
    if len(stacked) < VOCAB_SIZE:
        raise ValueError(f"Not enough descriptors: {len(stacked)} < {VOCAB_SIZE}")
    kmeans = MiniBatchKMeans(n_clusters=VOCAB_SIZE, batch_size=KMEANS_BATCH, random_state=42)
    # partial fit in batches
    for i in range(0, len(stacked), KMEANS_BATCH):
        kmeans.partial_fit(stacked[i:i+KMEANS_BATCH])
    return kmeans

# ------------------------------
# 4. Compute BoW histograms
def histograms(des_list, kmeans):
    H = np.zeros((len(des_list), VOCAB_SIZE), dtype=int)
    for i, des in enumerate(des_list):
        if des is not None and len(des)>0:
            words = kmeans.predict(des)
            hist, _ = np.histogram(words, bins=np.arange(VOCAB_SIZE+1))
            H[i] = hist
    return H

# ------------------------------
# 5. Train and evaluate classifiers
def evaluate(X_tr, y_tr, X_te, y_te):
    results = {}
    # KNN
    knn = KNeighborsClassifier(n_neighbors=KNN_NEIGHBORS, weights=’distance’, n_jobs=-1)
    t0 = time.time(); knn.fit(X_tr, y_tr); t_train = time.time()-t0
    t0 = time.time(); y_pred = knn.predict(X_te); t_inf = time.time()-t0
    results[’knn’] = {’train_time_s’:t_train,
                      ’inf_time_s’:t_inf,
                      ’accuracy’:accuracy_score(y_te, y_pred),
                      ’f1_macro’:f1_score(y_te, y_pred, average=’macro’)
                      # ’cm’:confusion_matrix(y_te, y_pred)
                      }
    # NB
    nb = GaussianNB(var_smoothing=NB_VAR_SMOOTH)
    t0 = time.time(); nb.fit(X_tr, y_tr); t_train = time.time()-t0
    t0 = time.time(); y_pred = nb.predict(X_te); t_inf = time.time()-t0
    results[’nb’] = {’train_time_s’:t_train,
                     ’inf_time_s’:t_inf,
                     ’accuracy’:accuracy_score(y_te, y_pred),
                     ’f1_macro’:f1_score(y_te, y_pred, average=’macro’)
                     # ’cm’:confusion_matrix(y_te, y_pred)
                     }
    return results

# ------------------------------
if __name__ == ’__main__’:
    # Initialize SIFT
    sift = cv2.SIFT_create(contrastThreshold=SIFT_CONTRAST, nOctaveLayers=SIFT_OCTAVE_LAYERS)
    # Load COCO images and labels
    x_train, y_train = load_coco_classification(COCO_IMG_DIR, COCO_ANN_FILE, max_images=5000)
    x_test,  y_test  = load_coco_classification(COCO_IMG_DIR, COCO_ANN_FILE, max_images=1000)
    print(f"Loaded train:{len(y_train)} test:{len(y_test)} imgs")
    # Extract descriptors
    des_tr = extract_descriptors(x_train, sift)
    des_te = extract_descriptors(x_test, sift)
    print("Descriptor extraction done")
    # Build vocabulary
    kmeans = build_vocabulary(des_tr)
    # Compute histograms
    X_tr = histograms(des_tr, kmeans)
    X_te = histograms(des_te, kmeans)
    # Evaluate
    res = evaluate(X_tr, y_train, X_te, y_test)
    for m,stats in res.items():
        print(f"\nModel: {m}", stats)
```

Listing 10: SIFT-BoW-COCO PART 2

```
  import os
from PIL import Image
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision.datasets import CocoDetection
from torchvision import transforms
from torch.utils.data import DataLoader, Dataset
from pycocotools.coco import COCO # type: ignore

# 1. Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print("Using device:", device)

# 2. Hyperparameters
batch_size    = 32
learning_rate = 1e-3
num_epochs    = 50
num_classes   = 80  # COCO has 80 object categories

# 3. Data transforms (resize + normalise for pretrained CNNs)
transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std =[0.229, 0.224, 0.225]),
])

# 4. Custom dataset subclassing CocoDetection correctly
class CocoClassification(CocoDetection):
    def __init__(self, img_folder, ann_file, transform=None, max_images=None):
        super().__init__(img_folder, ann_file, transforms=None)  # initialise base
        self.transform = transform
        # optionally limit size
        if max_images:
            self.ids = self.ids[:max_images]                     # crop to first N images

    def __getitem__(self, idx):
        # Leverage base class to load PIL image and raw annotations
        img, raw_anns = super().__getitem__(idx)               # returns PIL.Image and list
            of dicts
        if not raw_anns:
            # fallback if no annotation
            label = 0
        else:
            raw_cat_id = raw_anns[0]['category_id']
            label      = cat_to_idx[raw_cat_id]  # maps to [0,79]
        # apply transforms if provided
        if self.transform:
            img = self.transform(img)
        return img, label

# 5. Paths
data_root   = 'C:/Customize/Tool-Coding/Dataset/COCO/val2017/val2017'
ann_file    = 'C:/Customize/Tool-Coding/Dataset/COCO/annotations_trainval2017/annotations/
    instances_val2017.json'

# 6. Datasets and loaders
train_ds = CocoClassification(data_root, ann_file, transform=transform, max_images=5000)
test_ds  = CocoClassification(data_root, ann_file, transform=transform, max_images=1000)

train_loader = DataLoader(train_ds, batch_size=batch_size, shuffle=True,  num_workers=0)
test_loader  = DataLoader(test_ds,  batch_size=batch_size, shuffle=False, num_workers=0)

coco = COCO(ann_file)
cat_ids    = sorted(coco.getCatIds())
cat_to_idx = {cid: idx for idx, cid in enumerate(cat_ids)}

  }
```

Listing 11: VGG-CNN-COCO PART 1

```
# 7. Simple  V G G style  CNN
class SimpleVGG(nn.Module):
    def __init__(self, num_classes):
        super().__init__()
        self.features = nn.Sequential(
            # block1
            nn.Conv2d(3, 64, kernel_size=3, padding=1), nn.ReLU(inplace=True),
            nn.Conv2d(64,64, kernel_size=3, padding=1), nn.ReLU(inplace=True),
            nn.MaxPool2d(2),  # 112 112
            # block2
            nn.Conv2d(64,128,3,padding=1), nn.ReLU(inplace=True),
            nn.Conv2d(128,128,3,padding=1), nn.ReLU(inplace=True),
            nn.MaxPool2d(2),  # 56 56
        )
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(128*56*56, 1024), nn.ReLU(inplace=True), nn.Dropout(0.5),
            nn.Linear(1024, num_classes)
        )

    def forward(self, x):
        x = self.features(x)
        x = self.classifier(x)
        return x

model = SimpleVGG(num_classes).to(device)

# 8. Loss and optimiser
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

# 9. Training and evaluation loop
for epoch in range(1, num_epochs+1):
    model.train()
    running_loss = 0.0
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)
        optimizer.zero_grad()
        loss = criterion(model(images), labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item() * images.size(0)
    avg_loss = running_loss / len(train_loader.dataset)
    print(f"Epoch {epoch}/{num_epochs}     Training Loss: {avg_loss:.4f}")

    model.eval()
    correct = 0
    with torch.no_grad():
        for images, labels in test_loader:
            images, labels = images.to(device), labels.to(device)
            preds = model(images).argmax(dim=1)
            correct += (preds == labels).sum().item()
    accuracy = correct / len(test_loader.dataset)
    print(f"               Test Accuracy: {accuracy:.4f}")

print("Training complete.")

  }
```

Listing 12: VGG-CNN-COCO PART 2