Deep Learning vs. Local Features for Robust Robotic Object Classification

COMP61352: Cognitive Robotics and Computer Vision Assignment

Euan Baldwin 10818421
Department of Electrical and Electronic Engineering
University of Manchester
(Dated: April 25, 2025)

Robust object classification is vital for robotic perception. In this study, we compare a classical SIFT-based Bag-of-Words + SVM pipeline against a fine-tuned ResNet-50 CNN on 360 000 images from ten object classes from the iCubWorld Transformations dataset, captured under varied conditions during human–robot interaction. Quantitative results demonstrate that the CNN significantly outperforms the classical pipeline, achieving an accuracy of 94.3% compared to 67.7%, as well as substantially higher precision and recall. Grad-CAM visualisations indicate that the CNN not only attains superior class discrimination but also effectively localises task-relevant image regions, while the classical method often confuses visually similar objects. These findings confirm that deep learning provides greater robustness than local features for robotic object classification.

1. INTRODUCTION

Object recognition, the ability of a system to assign semantic labels to image regions corresponding to distinct object, is a cornerstone of both robotics and computer vision. Before the advent of deep learning, recognition pipelines typically relied on hand-engineered local descriptors and classical classifiers. A formative contribution was the Scale-Invariant Feature Transform (SIFT) algorithm, which detects and describes keypoints in a manner invariant to scale, rotation, and moderate illumination changes [1]. Building on SIFT, the Bag-of-Words (BoW) framework aggregates local descriptors into a fixed-length histogram of "visual words," analogous to term-frequency representations in text analysis [2]. These histograms serve as feature vectors for supervised classifiers, most commonly linear Support Vector Machines (SVMs), which separate object categories in high-dimensional space [3]. Such pipelines achieved state-of-the-art performance on early benchmarks throughout the 2000s [4].

Since the introduction of AlexNet in 2012 [5], however, convolutional neural networks (CNNs) have transformed image classification by learning hierarchical feature representations directly from data. At their core, CNNs apply successive layers of learnable convolutional filters and spatial pooling operations to extract increasingly abstract patterns, from edges and textures at early layers to object parts and global shape at deeper stages, and are trained end to end via back-propagation to minimise classification loss. Subsequent architectures such as VGG [6], ResNet [7], and EfficientNet [8], have progressively increased depth, capacity, and training efficiency, yielding significant improvements in accuracy on large-scale datasets. The combination of abundant labelled data and GPU acceleration has enabled CNNs to surpass traditional methods by substantial margins [9, 10].

Fine-tuning pretrained CNNs on new target domains has become a standard transfer-learning paradigm, delivering strong performance even under challenging conditions such as those encountered in robotic vision, including variable lighting, cluttered backgrounds and unconstrained object viewpoints. In this study, we employ a subset of the iCubWorld Transformations dataset [11] to perform a controlled comparison between a classical SIFT+BoW+SVM pipeline and a transfer-learned CNN for ten-class object recognition. Both approaches undergo rigorous hyperparameter optimisation to ensure fairness. We report quantitative metrics (accuracy, precision, recall, confusion matrices) alongside qualitative insights based on interpretability visualisations. Finally, we consider our results against the current state of the art in robotic object classification.

2. METHODOLOGY

2.1. Dataset

For this project, we selected the *iCubWorld Transformations* dataset [11, 12] as the sole benchmark for both approaches. The dataset comprises images captured by the iCub humanoid robot [12] as it observes a variety of objects in a laboratory and office environment. The Transformations subset was developed to evaluate object recognition under different organised variations in appearance within a robotic context. It contains 200 objects across 20 categories, with around ten instances per category [13]. An example of the intra-class semantic variability of the *mug* class is shown in Figure 1.

FIG. 1. Sample images for the different object instances in the *mug* category to provide a qualitative understanding of the semantic variability in iCubWorld [11].

Unlike generic vision datasets, iCubWorld is acquired from the robot's 'first-person' perspective during human–robot interaction. A human 'teacher' presents and manipulates each object, whilst the robot's cameras record sequences of frames. This produces imagery that reflects realistic operational conditions, featuring natural backgrounds, mild motion blur and varying illumination. The Transformations subset examines recognition performance under five specific conditions. The first, *2D Rotation*, involves in-plane rotation altering object orientation within the image plane; the second, *3D Rotation*, comprises out-of-plane rotations offering different viewpoints (e.g. front, side, back); the third, *Scale*, varies the distance

to the camera, thus changing the object's apparent size; the fourth, *Background Change*, entails movement of the object against varying backgrounds while maintaining approximately constant pose and scale; and the fifth, *Mixed*, constitutes a free-form combination of the preceding transformations, simulating natural interaction with arbitrary changes in angle, distance and background. An example of excerpts from five transformation sequences acquired for the *mug* instance is shown in Figure 2.

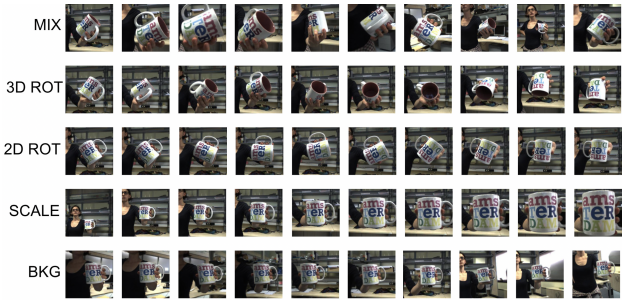

FIG. 2. Excerpts from the sequences acquired for one mug, representing the object while it undergoes visual transformations [11].

For each transformation, roughly 150 frames are captured, except in the mixed condition, which comprises about 300 frames. To assess the impact of illumination, all transformations are repeated under two distinct lighting setups on separate days, yielding ten sequences per object. As each of the ten classes contains ten objects, we hence have a total of 36 000 images per class. All images are recorded at 640 × 480 pixel resolution and include bounding-box annotations generated by the robot's attention system. In our experiments, we employed the provided cropped object images to ensure consistency.

To prepare the data, the raw archives were first extracted, and the dataset was divided into training and testing subsets using stratified sampling to maintain the class distribution. 80% of the images per class were allocated to training, and the remaining 20% to testing [14]. Each image was already cropped around the target object; we uniformly resized them to 224×224 pixels to ensure compatibility with both the classical feature-extraction pipeline and the CNN. Prior to any feature computation or network input, pixel values were scaled to [0, 1] and then normalised by subtracting the channel-wise mean and dividing by the channel-wise standard deviation [15].

2.2. *Traditional BoW+SVM Pipeline*

Our classical baseline follows a three stage pipeline - local descriptor extraction, visual vocabulary construction, and linear-SVM classification - chosen to balance discriminative power, computational efficiency, and ease of implementation. We begin by computing SIFT descriptors on each training image to exploit their proven robustness to scale and rotation variations, which are prevalent in our robot acquired views [1]. Because clustering the full set of 128-dimensional descriptors would be prohibitively expensive, we apply Principal Component Analysis (PCA) to project each descriptor onto its top 64 principal components; this reduction preserves over 95% of the original variance while dramatically lowering both memory demands and clustering runtime [16]. To build a compact yet expressive codebook, these reduced descriptors are then clustered with MiniBatch K-Means, a method well suited to large scale data, and the vocabulary size is treated as a tunable hyperparameter.

Optuna-driven cross-validation [17] over a range of 50 to 500 visual words revealed that a 500-word codebook achieves the best trade-off between representational richness and tractable model complexity. We therefore construct our final codebook by executing 100 iterations of MiniBatch K-Means on the full training set. Optuna uses a Bayesian-style optimiser (Tree-structured Parzen Estimator) to model validation accuracy as a function of hyperparameters. At each trial, it proposes a vocabulary size and an SVM regularisation constant, evaluates them on a held-out validation set, then updates its internal model to focus subsequent trials on the most promising regions of the hyperparameter space. Finally, we optimise the SVM regularisation constant, C, within a logarithmic range of 10^{-3} to 10^{2} using Optuna, arriving at $C = 2.33$ for our dataset.

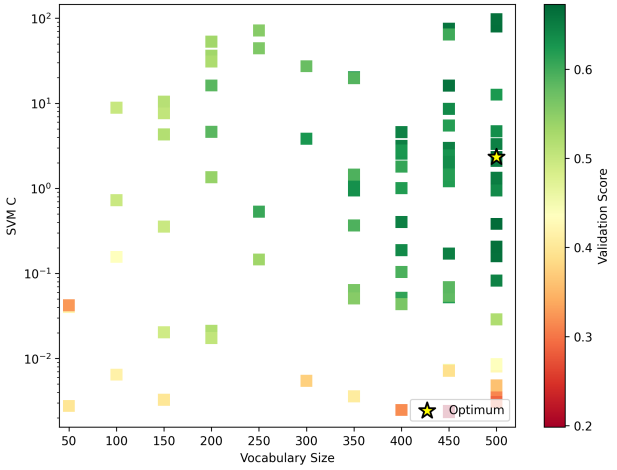

FIG. 3. Validation-score heatmap for the BoW+SVM search. Cells indicate accuracy for vocabulary size (x-axis) and SVM C (y-axis, log-scale), coloured red to green (low to high accuracy). The star marks the optimum (vocab size = 500; $C = 2.33$).

Figure 3 demonstrates that validation performance improves linearly with increasing vocabulary size: smaller codebooks (50 to 150 words) yield only moderate accuracy, whereas larger codebooks (400 to 500 words) produce markedly higher scores. We stopped at 500 word codebooks as pushing beyond this threshold led to a dramatic increase in clustering time and memory usage, ballooning CPU load and extending optimisation runtimes from minutes to several hours per trial. Given these steep computational costs and diminishing returns in accuracy gains, we deemed larger codebooks impractical for our cross-validation workflow. Regarding the regularisation constant, moderate values of C resulted in the optimal balance between under- and over-fitting. Too much regularisation (low C) underfits the data, while too little (very high C) incurs slight over-fitting without further gains. The optimal point, at 500 words and $C = 2.33$, highlights that an extensive visual vocabulary combined with moderate regularisation delivers the best validation accuracy in this pipeline.

Once the visual vocabulary is established, each image is encoded as a histogram of word occurrences by assigning its PCA-compressed descriptors to the nearest centroid and applying L_2-normalisation to mitigate the influence of dif-

fering keypoint counts [18]. We then adopt a linear SVM as our multi-class classifier, utilising its ability to generate sparse, high-margin decision boundaries while maintaining fast inference - an important consideration for on-robot deployment. Multi-class discrimination is handled via a one-vs-rest scheme, training ten binary SVMs and selecting the class with the highest confidence score at test time [19]. Lastly, the resulting optimal BoW+SVM pipeline is evaluated on held-out test images.

2.3. *Deep Learning CNN Pipeline*

In parallel, we developed a CNN pipeline grounded in transfer learning with ResNet-50 [7], whose deep residual connections both accelerate convergence and alleviate vanishing-gradient issues in very deep architectures. These qualities make it a robust backbone for robotic object classification. By initialising the network with ImageNet-pretrained weights and freezing all convolutional layers, we leveraged large-scale learned representations while avoiding overfitting on our comparatively small iCubWorld dataset and dramatically reducing training time [20, 21]. We replaced the original 1 000-way classification head with a Dropout layer followed by a fully connected layer tailored to our target classes, using dropout to regularise the head and mitigate co-adaptation of features in this low-data regime [22].

To match the pretrained filters' receptive fields and normalisation statistics, input images were uniformly resized to 224×224 pixels and normalised to ImageNet's channel means and variances, ensuring compatibility with ResNet-50's learned feature scales [15]. During training, we applied online augmentations - random horizontal flips at 50% probability and brightness/contrast jitter at 20% probability - to simulate viewpoint and illumination variability inherent in robotic perception, thereby improving the model's robustness to such variations without altering the underlying object semantics [10]. No augmentations were performed during validation or testing to preserve evaluation consistency.

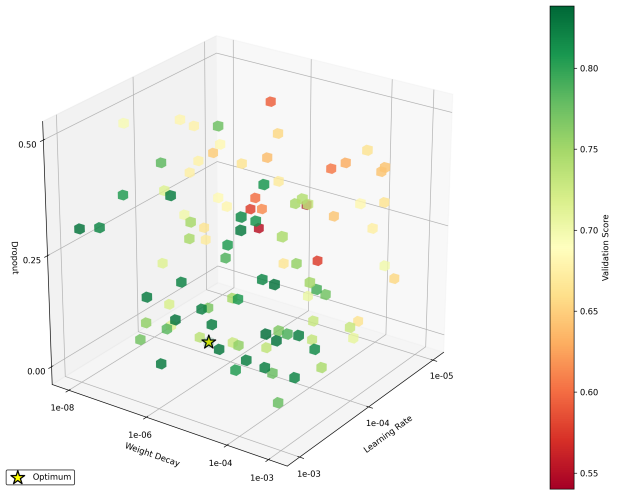

FIG. 4. Validation scores from the CNN fine-tuning hyperparameter search. Cubes represent trials at varying learning rates (x-axis, log-scale), weight decays (y-axis, log-scale) and dropout rates (z-axis), coloured red to green (low to high accuracy). The star marks the optimum (Learning rate = $9.25{\times}10^{-4}$; weight decay = $1.56{\times}10^{-5}$; dropout = 0.141).

We then tuned the classifier head's learning rate, weight decay and dropout rate via 100 Optuna [17] trials, sampling the learning rate logarithmically between 10^{-5} and 10^{-3}, weight decay between 10^{-8} and 10^{-3}, and dropout between 0.0 and 0.5. Figure 4 illustrates that the highest validation scores cluster at lower dropout rates, higher learning rates and moderate weight decay. High dropout (0.25–0.5) generally corresponds to lower accuracy, suggesting that fine-tuning benefits more from retaining network capacity than from aggressive unit dropping. Extremely low learning rates (10^{-5}) also degraded performance, indicating that a higher update step is critical for convergence. Similarly, very low weight decay (10^{-8}) under-regularises the model, while stronger weight decay (10^{-5}) yields superior generalisation. This search yielded a learning rate of 9.25×10^{-4}, weight decay of 1.56×10^{-5} and dropout of 0.141. These parameters balance sufficient gradient updates against over-regularisation in our fine-tuning setting.

We trained the head with the Adam optimiser, chosen for its adaptive moment estimation and rapid convergence in low-data contexts [23], for up to 50 epochs, employing early stopping when validation accuracy plateaued (improvement $< 0.2\%$ over three epochs) to prevent overfitting. Only the newly added layers were updated, preserving the general visual features in the frozen backbone. Implementation in PyTorch on NVIDIA A100 GPUs (accessed via Google Colaboratory [24]) allowed us to process nearly 200 images per second at inference, enabling rapid evaluation over our 49 000-image test set. By contrast, the BoW+SVM approach, while parallelised for SIFT extraction on CPU, remained markedly slower due to the inherently sequential nature of descriptor computation.

Finally, to verify that the network's decisions originate from semantically relevant regions rather than background artefacts, we applied Grad-CAM to the last convolutional block. Grad-CAM computes the gradient of the class score with respect to each feature-map channel, averages these gradients spatially to obtain importance weights, and then forms a weighted sum of the feature maps, yielding a localisation map. The resulting class-specific heatmaps can then be overlaid on the input images for qualitative inspection [25].

3. RESULTS

3.1. *Traditional BoW+SVM Pipeline*

The BoW+SVM pipeline was evaluated on the held-out split of 48 860 images from the iCubWorld Transformations benchmark. As summarised in Table I, this model achieved an overall accuracy of 67.7% (error rate 32.3%), macro-averaged F_1 of 0.67 and weighted F_1 of 0.68. Per-class F_1 values ranged from 0.58 for *sunglasses* to 0.75 for *book* and *hairclip*. The normalised confusion matrix in Figure 5 highlights the distribution of errors, with true-positive rates (diagonal) rarely exceeding 75% and several prominent off-diagonal cells exceeding 10%. In particular, *hairbrush* is most frequently mislabelled as *perfume* (14.8% of cases) or *pencilcase* (10.7%), and 13.6% of true *cellphone* images are predicted as *sunglasses*. These patterns indicate limited class separability when relying solely on hand-crafted local features.

Class	Precision	Recall	F_1-score	Support
book	0.78	0.73	0.75	5 091
cellphone	0.65	0.59	0.62	4 007
hairbrush	0.70	0.64	0.67	4 970
hairclip	0.76	0.75	0.75	5 442
mouse	0.74	0.75	0.74	5 195
pencilcase	0.62	0.66	0.64	5 686
perfume	0.66	0.74	0.70	5 693
ringbinder	0.70	0.65	0.67	4 280
sunglasses	0.56	0.60	0.58	4 263
wallet	0.60	0.59	0.60	4 233
Accuracy		0.68		48 860
Macro avg	0.68	0.67	0.67	48 860
Weighted avg	0.68	0.68	0.68	48 860

TABLE I. Test-set performance of the BoW+SVM pipeline on iCubWorld Transformations. Per-class precision, recall, F_1-score and support are provided, alongside overall metrics.

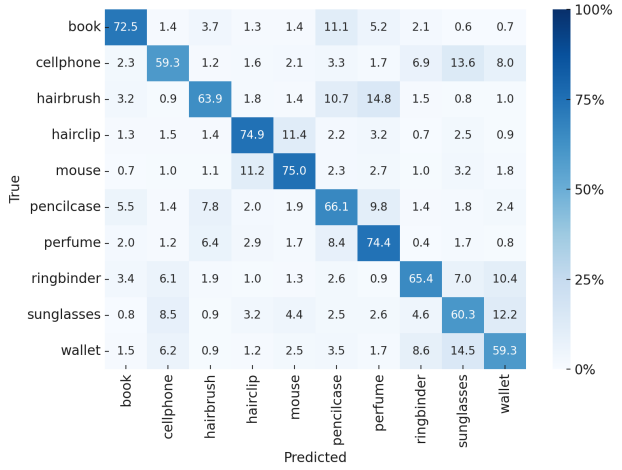

FIG. 5. Confusion matrix for the BoW+SVM pipeline on the test set. The diagonal entries indicate correct predictions, while off-diagonals highlight class confusions.

3.2. *Deep Learning CNN Pipeline*

Fine-tuning the ResNet-50 on the same training images led to a marked improvement. Over the course of fine-tuning, the model's training loss decreases rapidly from 0.52 at epoch one to 0.29 by epoch three, before settling into a more gradual decline ($\approx$ 0.25 by epoch eleven). Meanwhile, validation accuracy starts high ($\approx$ 90%) and climbs to a plateau of $\approx$ 94% by epoch eight, with only minor fluctuations thereafter. This pattern indicates that the classifier quickly learns the bulk of its representation in the early epochs, while further training yields diminishing returns. The early stopping mechanism was triggered after epoch eleven as the accuracy had failed to improve by $> 0.2\%$ over the previous three epochs.

Table II reports an overall accuracy of 94.3% (error rate 5.7%), macro-averaged F_1 of 0.94 and weighted F_1 of 0.94. All ten classes exceed 0.90 F_1, with the top recall observed for *hairclip* (98.2%) and *mouse* (95.7%). Figure 7 shows a confusion matrix strongly concentrated along the diagonal: true-positive rates surpass 90% for every class, and the largest off-diagonal entry (6.3%) corresponds to *cellphone* misclassified as *sunglasses*. Apart from this isolated confusion, non-diagonal cells rarely exceed 3%, demonstrating the network's high discriminative ability.

Relative to the BoW+SVM baseline, the CNN reduces the error rate from 32.3% to 5.7%, an 82% reduction, and raises macro-averaged F_1 by 0.27. The confusion matrices underscore these gains: diagonal dominance improves from a median of roughly 66% to over 95% across classes, while major off-diagonal confusions shrink by a factor of four to five. Thus, the CNN exhibits substantially stronger and more uniform performance across all object categories.

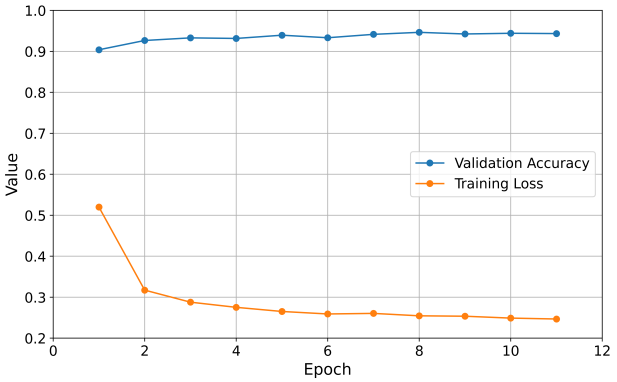

FIG. 6. Training loss (orange) and validation accuracy (blue) over 11 fine-tuning epochs for the ResNet-50 CNN on the iCubWorld Transformations dataset.

Class	Precision	Recall	F_1-score	Support
book	0.97	0.93	0.95	5 066
cellphone	0.98	0.87	0.92	3 991
hairbrush	0.97	0.93	0.95	4 959
hairclip	0.94	0.98	0.96	5 378
mouse	0.99	0.96	0.97	5 163
pencilcase	0.92	0.97	0.94	5 721
perfume	0.97	0.95	0.96	5 699
ringbinder	0.89	0.96	0.92	4 285
sunglasses	0.87	0.96	0.92	4 323
wallet	0.92	0.90	0.91	4 224
Accuracy		0.94		48 809
Macro avg	0.94	0.94	0.94	48 809
Weighted avg	0.95	0.94	0.94	48 809

TABLE II. Test-set classification performance of the CNN on iCubWorld Transformations. Per-class precision, recall, F_1-score and support are provided, alongside overall metrics.

4. DISCUSSION

The experimental results demonstrate that the CNN approach markedly outperforms the traditional BoW+SVM pipeline on the iCubWorld object recognition task. In this discussion, we analyse the underlying reasons for this performance gap, examine error patterns, assess robustness and computational trade-offs, and consider implications for robotic applications.

At the heart of the CNN's success lies its hierarchical feature extraction. Early convolutional layers detect simple primitives - edges, corners and small texture patches - while deeper successive convolution and pooling layers encode these into object parts and holistic shapes. Such compositionality proves critical when objects undergo transformations in scale, viewpoint or lighting. For instance, the network reliably recognises a book whether it is tilted, partially occluded or shadowed. The CNN's misclassifications are both infrequent and semantically plausible -

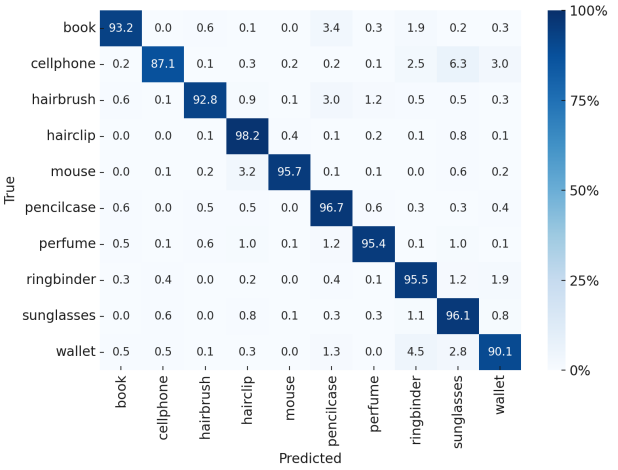

FIG. 7. Confusion matrix for the fine-tuned ResNet-50 CNN. The heatmap shows high true positive counts on the diagonal and minimal confusion between classes.

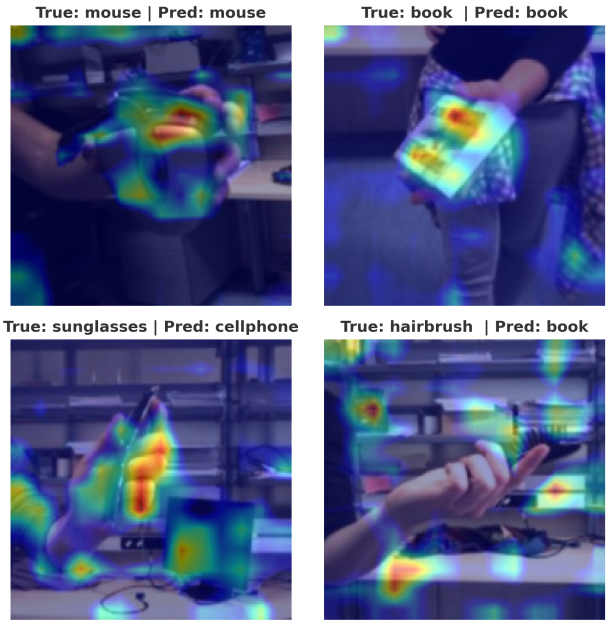

FIG. 8. Grad-CAM visualisations for sample test images. Top: correctly classified *mouse* and *book*, with attention centred on salient object features. Bottom: misclassified *sunglasses* (predicted as cellphone) and *hairbrush* (predicted as book).

most confusions occur between visually similar pairs such as wallets and sunglasses. By contrast, the BoW model aggregates SIFT descriptors without regard to their spatial arrangement, causing it to confuse objects that share local appearance but differ in configuration. The confusion between hairbrushes and perfume bottles (Figure 5) exemplifies this limitation: both may exhibit cylindrical, metallic regions, yet only the CNN can leverage spatial context to distinguish the bristles from the nozzle.

Transfer learning from a large-scale dataset further bolsters the CNN's performance. By initialising with weights trained on ImageNet, the ResNet-50 backbone already contains a broad range of edge, texture, and shape detectors. Fine-tuning on the ten iCubWorld categories adapts these general-purpose filters to the specific object set, requiring relatively few epochs to converge. In contrast, the BoW model constructs its descriptor vocabulary solely from the relatively limited training set, restricting its feature diversity.

To interpret the CNN's internal representations, we generated Grad-CAM visualisations for both correctly and incorrectly classified images (Figure 8). In the true positive examples, such as the *mouse* and *book*, the attention maps reliably highlight characteristic features including the mouse's buttons and base, and the book's cover and edges. Conversely, in misclassified cases the model's focus shifts to plausible but misleading regions. For instance, when *sunglasses* are mislabelled as *cellphone*, the heatmap centres on the hand rather than the eyewear. Similarly, a *hairbrush* predicted as *book* attracts attention to books in the background. These results demonstrate that, although the CNN generally learns to attend to semantically meaningful object parts, incidental shape similarities or background elements can occasionally divert its attention and lead to errors. In contrast, although SIFT visual words are in principle traceable to specific image patches, the sheer number of clusters renders manual inspection unwieldy and of limited practical interpretability value.

From a computational perspective, the two methods exhibit complementary profiles. The BoW+SVM pipeline front loads computation in SIFT extraction and clustering, which even when parallelised, remains time consuming on CPUs, effectively capping the visual vocabulary at 500 words during hyperparameter search. Subsequent SVM training and inference are efficient, however. The CNN demands significant GPU resources during training but delivers high throughput inference when deployed on modern hardware. For real time robotic applications, the CNN's inference efficiency and accuracy justify its higher training cost.

Despite the CNN's robustness, it still sometimes confuses visually similar classes such as wallets and sunglasses because in low resolution or tightly cropped views both appear as flat, dark rectangles with minimal internal detail, which highlights clear routes for refinement. Adopting multi-scale inputs or integrating attention mechanisms that model part–whole relationships could steer the network towards subtle cues (e.g. zip fasteners versus lens curvature). Targeted data augmentation that emphasises such fine-grained details may likewise improve discrimination under challenging conditions. Standard regularisation, including weight decay, dropout, augmentation and early stopping, has already curbed over-fitting, as evidenced by the high accuracy on a large held-out test set. In contrast, improving the BoW+SVM baseline would require richer encodings that restore spatial context (e.g. spatial pyramids [26]) or capture higher-order statistics (e.g. Fisher vectors [27]). Non-linear SVM kernels could offer additional flexibility, but any gains would likely be incremental relative to the performance gap already achieved by the CNN.

Additional simulations could sharpen our understanding of each method's limits. Systematic ablations of the augmentation pipeline would reveal which transformations most enhance robustness, and analysing performance separately for the five predefined iCubWorld conditions (2-D rotation, 3-D rotation, scale, background change and mixed) would isolate the transformations factors that remain challenging for the pipelines. Evaluating alternative backbones - such as EfficientNet [8] or Vision Transform-

ers [28] - under identical training regimes would clarify trade-offs between accuracy, parameter efficiency and inference speed. Finally, open-set and few-shot experiments would test each pipeline's ability to recognise novel objects without catastrophic forgetting, an essential capability for long-lived robotic platforms [29, 30].

Overall, in the context of robotics, the 94.3% accuracy attained by the fine-tuned ResNet-50, versus the 67.7% for the BoW+SVM baseline, reduces the error rate from 32.3% to just 5.7%, an 82% drop. A platform that relied on the classical pipeline would misidentify roughly one object in three, whereas a CNN-based system would make fewer than one mistake in twenty. Such reliability is indispensable for safe, effective human–robot interaction. These results reinforce the prevailing view in robotic vision: deep learning, bolstered by transfer learning and efficient inference hardware, remains the state-of-the-art solution for object recognition in real-world deployments.

5. STATE OF THE ART

While our experimental results demonstrate the benefits of deep CNNs for object recognition in the iCubWorld context, it is important to view these insights against the wider trajectory of robotic vision research. The last decade has witnessed a decisive transition in robotic object recognition: hand-crafted pipelines such as SIFT+BoW+SVM are now largely baselines, whereas deep, end-to-end models underpin state-of-the-art systems. Early CNN breakthroughs (AlexNet, VGG) were soon surpassed by architectures that eased optimisation bottlenecks - most notably residual connections in ResNet [7]. Later innovations improved parameter efficiency (EfficientNet's compound scaling [8]) and introduced patch-wise self-attention (Vision Transformers [28]). Self-supervised learning, including contrastive methods such as SimCLR [31] and BYOL [32], as well as masked-token reconstruction with MAE [33], now yields pretraining that transfers with little or no manual labelling.

In robotics, domain shift is a real challenge: robot-captured images often come with unusual angles, motion blur and cluttered backgrounds that are unusual in regular images. Fine-tuning with a few hundred labelled frames is usually sufficient, but meta-learning and few-shot methods can adapt with only one or two examples per class [29, 34]. Continual-learning algorithms mitigate catastrophic forgetting over a robot's lifespan [30], while open-set recognition adds a rejection option, vital for safety in unstructured settings [35]. However, classical local features have not vanished entirely. ORB, SURF and SIFT remain central to visual–inertial odometry and SLAM, where geometric consistency outweighs category-level semantics [36, 37]. In these pipelines, matching keypoints is essential for estimating camera pose, detecting loop closures and maintaining maps - tasks where deep descriptors remain too computationally demanding or not invariant enough.

Deploying on edge devices favours lightweight backbones, such as MobileNet and ShuffleNet, alongside techniques including quantisation (which reduces the numerical precision of the network's weights), pruning and neural architecture search to meet latency and energy constraints on embedded GPUs or TPUs [38, 39]. More recently, "foundation" models have extended capabilities, CLIP [40] and ALIGN [41] offer vision–language embeddings for zero-shot recognition, while Segment Anything [42] supplies promptable masks that simplify annotation. Generative approaches, DALL·E and Stable Diffusion [43, 44], are also under investigation for synthesising rare viewpoints, although the sim-to-real gap remains a major hurdle.

These deep networks deliver unrivalled accuracy and adaptability, as shown by the 94.3% versus 67.7% performance gap between our fine-tuned ResNet-50 and the BoW+SVM baseline, but they bring new challenges including high training costs, energy demands and unexplained failure modes. Current research therefore targets efficient training, interpretability (e.g. Grad-CAM), robustness to harsh lighting and actuation noise, and formal safety guarantees. Overall, the consensus in robotic vision is clear: deep learning, complemented by local feature matching for SLAM and supported by edge-efficient deployment strategies, constitutes the present state of the art for perception in real-world robotic systems.

6. CONCLUSION

In this report, we conducted a comprehensive comparison between a traditional computer vision approach and a deep learning approach for object recognition on a robotics-oriented dataset. The traditional method (SIFT+BoW+SVM) achieved moderate success, with 67.7% accuracy and substantial confusion between classes. In contrast, the fine-tuned CNN (ResNet-50) delivered excellent performance, with 94.3% accuracy, greatly improving precision and recall for all object categories. Grad-CAM visualisations confirmed that the CNN learned to focus on the actual objects, whereas the BoW model, by its nature, lacked such focused understanding.

This comparative analysis provides concrete evidence that deep learning methodologies vastly outperform classic computer vision techniques for object recognition in complex, real-world imagery. The CNN's high accuracy and its ability to localise relevant image regions make it a superior choice for tasks including robust robotic object perception. In contrast, traditional feature-based methods, while historically important and still useful for certain tasks (including SLAM), cannot match the accuracy or adaptability of modern deep learning models in scenarios such as those in the iCubWorld dataset. The rapid progress in deep learning architecture design and training techniques continues to widen this performance gap, reinforcing that the future of robust robotic object classification lies in deep learning-based approaches.

[1] D. G. Lowe, in *Proceedings of the seventh IEEE international conference on computer vision*, Vol. 2 (Ieee, 1999) pp. 1150–1157.

[2] Sivic and Zisserman, in *Proceedings ninth IEEE international conference on computer vision* (IEEE, 2003) pp. 1470–1477.

[3] C. Cortes and V. Vapnik, Machine learning **20**, 273 (1995).

[4] M. Everingham *et al.*, International journal of computer vision **88**, 303 (2010).

[5] M. Z. Alom *et al.*, arXiv preprint arXiv:1803.01164 (2018).

[6] A. Dhillon and G. K. Verma, Progress in Artificial Intelligence **9**, 85 (2020).

[7] K. He *et al.*, in *Proceedings of the IEEE conference on computer vision and pattern recognition* (2016) pp. 770–778.

[8] M. Tan and Q. Le, in *International conference on machine learning* (PMLR, 2019) pp. 6105–6114.

[9] D. Ciregan, U. Meier, and J. Schmidhuber, in *2012 IEEE conference on computer vision and pattern recognition* (IEEE, 2012) pp. 3642–3649.

[10] A. Krizhevsky, I. Sutskever, and G. E. Hinton, Advances in neural information processing systems **25** (2012).

[11] G. Pasquale *et al.*, Robotics and Autonomous Systems **112**, 260 (2019).

[12] G. Metta *et al.*, Neural networks **23**, 1125 (2010).

[13] G. Pasquale, C. Ciliberto, L. Rosasco, and L. Natale, iCubWorld Transformations, https://robotology.github.io/iCubWorld/icubworld-transformations-modal (2016), accessed: 2025-04-22.

[14] F. Pedregosa *et al.*, the Journal of machine Learning research **12**, 2825 (2011).

[15] J. Deng *et al.*, in *2009 IEEE conference on computer vision and pattern recognition* (Ieee, 2009) pp. 248–255.

[16] I. T. Jolliffe, *Principal component analysis for special types of data* (Springer, 2002).

[17] T. Akiba *et al.*, in *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining* (2019) pp. 2623–2631.

[18] G. Csurka *et al.*, in *Workshop on statistical learning in computer vision, ECCV*, Vol. 1 (Prague, 2004) pp. 1–2.

[19] R. Rifkin and A. Klautau, Journal of machine learning research **5**, 101 (2004).

[20] K. Chatfield *et al.*, arXiv preprint arXiv:1405.3531 (2014).

[21] S. Kornblith, J. Shlens, and Q. V. Le, in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition* (2019) pp. 2661–2671.

[22] N. Srivastava *et al.*, The journal of machine learning research **15**, 1929 (2014).

[23] D. P. Kingma and J. Ba, arXiv preprint arXiv:1412.6980 (2014).

[24] Google Research, Google Colaboratory, `https://colab.research.google.com/` (2025), [Online; accessed 24-April-2025].

[25] R. R. Selvaraju *et al.*, in *Proceedings of the IEEE international conference on computer vision* (2017) pp. 618–626.

[26] S. Lazebnik, C. Schmid, and J. Ponce, in *2006 IEEE computer society conference on computer vision and pattern recognition (CVPR'06)*, Vol. 2 (IEEE, 2006) pp. 2169–2178.

[27] F. Perronnin, J. Sánchez, and T. Mensink, in *Computer Vision–ECCV 2010: 11th European Conference on Computer Vision, Heraklion, Crete, Greece, September 5-11, 2010, Proceedings, Part IV 11* (Springer, 2010) pp. 143–156.

[28] A. Dosovitskiy *et al.*, arXiv preprint arXiv:2010.11929 (2020).

[29] O. Vinyals *et al.*, Advances in neural information processing systems **29** (2016).

[30] M. De Lange *et al.*, IEEE transactions on pattern analysis and machine intelligence **44**, 3366 (2021).

[31] T. Chen, S. Kornblith, M. Norouzi, and G. Hinton, in *International conference on machine learning* (PmLR, 2020) pp. 1597–1607.

[32] J.-B. Grill *et al.*, Advances in neural information processing systems **33**, 21271 (2020).

[33] K. He *et al.*, in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition* (2022) pp. 16000–16009.

[34] C. Finn, P. Abbeel, and S. Levine, in *International conference on machine learning* (PMLR, 2017) pp. 1126–1135.

[35] A. Bendale and T. E. Boult, in *Proceedings of the IEEE conference on computer vision and pattern recognition* (2016) pp. 1563–1572.

[36] R. Mur-Artal, J. M. M. Montiel, and J. D. Tardos, IEEE transactions on robotics **31**, 1147 (2015).

[37] H. Bay, T. Tuytelaars, and L. Van Gool, in *Computer Vision–ECCV 2006: 9th European Conference on Computer Vision, Graz, Austria, May 7-13, 2006. Proceedings, Part I 9* (Springer, 2006) pp. 404–417.

[38] A. G. Howard *et al.*, arXiv preprint arXiv:1704.04861 (2017).

[39] X. Zhang *et al.*, in *Proceedings of the IEEE conference on computer vision and pattern recognition* (2018) pp. 6848–6856.

[40] A. Radford *et al.*, in *International conference on machine learning* (PmLR, 2021) pp. 8748–8763.

[41] C. Jia *et al.*, in *International conference on machine learning* (PMLR, 2021) pp. 4904–4916.

[42] A. Kirillov *et al.*, in *Proceedings of the IEEE/CVF international conference on computer vision* (2023) pp. 4015–4026.

[43] A. Ramesh *et al.*, in *International conference on machine learning* (Pmlr, 2021) pp. 8821–8831.

[44] R. Rombach *et al.*, in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition* (2022) pp. 10684–10695.

APPENDIX

A. FULL PIPELINE CODE

Listing 1. Pipeline comparing SIFT–BoW–SVM with a fine-tuned ResNet-50 on the iCubWorld Transformations dataset.

```
#!/usr/bin/env python3
# ================================================================================
# iCubWorld-Transformations: End-to-End Experimental Pipeline
# ================================================================================
"""
A reproducible pipeline for evaluating traditional Bag-of-Words (BoW)
and modern deep-learning baselines on the iCubWorld-Transformations dataset.

Sections
--------
1.  Data preparation
2.  BoVW + linear-SVM
3.  CNN fine-tuning & hyper-parameter optimisation
4.  Grad-CAM visualisation
5.  Script entry-point
"""

# Standard and third-party library imports
import importlib, subprocess, sys

# Ensure required packages are installed (useful for Google Colab environments)
for _pkg in ("albumentations", "captum", "optuna"):
    try:
        importlib.import_module(_pkg)
    except ModuleNotFoundError:
        print(f"    Installing missing dependency: {_pkg}")
        subprocess.check_call([sys.executable, "-m", "pip", "install", "-qq", _pkg])

from __future__ import annotations  # Enables postponed evaluation of type hints

# Core modules and libraries
import os
import random
import shutil
import tarfile
from concurrent.futures import ThreadPoolExecutor
from pathlib import Path
from typing import Iterable, Mapping, Sequence

# Scientific and ML libraries
import cv2
import matplotlib.pyplot as plt
import numpy as np
import optuna
import seaborn as sns
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from albumentations import Compose, HorizontalFlip, Normalize, RandomBrightnessContrast,
    Resize
from albumentations.pytorch import ToTensorV2
from captum.attr import LayerGradCam
from sklearn.cluster import MiniBatchKMeans
from sklearn.decomposition import IncrementalPCA
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from torch.cuda.amp import GradScaler, autocast
from torch.utils.data import DataLoader
from torch.utils.tensorboard import SummaryWriter
from torchvision import datasets, models


```

```python
# === 1. Data Preparation ===

def mount_and_extract(archive_dir: Path, extract_dir: Path) -> None:
    """
    If running in Google Colab, this mounts Google Drive and extracts all
    *_cropped.tar archives from the source directory into the target extraction
    directory.
    """
    try:
        from google.colab import drive  # Only available in Colab
    except ModuleNotFoundError:
        print("Google Colab not detected     skipping drive mount.")
    else:
        drive.mount("/content/drive", force_remount=True)

    extract_dir.mkdir(parents=True, exist_ok=True)
    # Extract each archive file found in the source directory
    for tar_path in archive_dir.glob("*_cropped.tar"):
        print(f"Extracting '{tar_path.name}'     ")
        with tarfile.open(tar_path) as tarf:
            tarf.extractall(path=extract_dir)


def stratified_split_to_folders(
    source_roots: Sequence[Path],
    train_root: Path,
    test_root: Path,
    img_suffixes: set[str],
    test_fraction: float = 0.20,
) -> None:
    """
    Splits dataset folders into train and test sets while preserving the folder
    structure required by torchvision's ImageFolder class.
    """
    # Remove existing output directories to avoid data contamination
    for root in (train_root, test_root):
        shutil.rmtree(root, ignore_errors=True)
        root.mkdir(parents=True, exist_ok=True)

    # Iterate over each dataset partition (e.g. part1, part2)
    for partition in source_roots:
        for class_dir in filter(Path.is_dir, partition.iterdir()):
            # Collect image files with the specified suffixes
            images = [
                p for p in class_dir.rglob("*") if p.suffix.lower() in img_suffixes
            ]
            if not images:
                continue

            # Perform stratified split (shuffling for randomness)
            train_imgs, test_imgs = train_test_split(
                images, test_size=test_fraction, random_state=42, shuffle=True
            )

            # Copy each image to the appropriate train/test class sub-folder
            for subset, destination in ((train_imgs, train_root), (test_imgs, test_root)
                ):
                target_dir = destination / class_dir.name
                target_dir.mkdir(parents=True, exist_ok=True)
                for img_path in subset:
                    shutil.copy(img_path, target_dir / img_path.name)


# === 2. Bag-of-Words (BoW) + Linear SVM ===

def compute_sift_descriptors(
    image_paths: Iterable[Path], num_workers: int = 8
) -> Mapping[Path, np.ndarray]:
    """
```

```
    Computes SIFT descriptors for a set of images using multithreading.
    Skips images that yield no keypoints.
    """
    sift = cv2.SIFT_create()
    descriptor_cache: dict[Path, np.ndarray] = {}

    def _worker(path: Path):
        grey = cv2.imread(str(path), cv2.IMREAD_GRAYSCALE)
        _, desc = sift.detectAndCompute(grey, None)
        return (path, desc) if desc is not None and desc.size else None

    with ThreadPoolExecutor(num_workers) as pool:
        for result in pool.map(_worker, image_paths):
            if result:
                descriptor_cache[result[0]] = result[1].astype(np.float32)

    return descriptor_cache


def fit_incremental_pca(
    descriptors: Sequence[np.ndarray], *, n_components: int = 64, sample_cap: int = 20
        _000
) -> tuple[IncrementalPCA, np.ndarray]:
    """
    Applies PCA for dimensionality reduction using incremental fitting.
    Limits number of descriptors to a fixed cap to improve performance.
    """
    stacked = np.vstack(descriptors)
    if stacked.shape[0] > sample_cap:
        idx = np.random.choice(stacked.shape[0], sample_cap, replace=False)
        stacked = stacked[idx]

    pca = IncrementalPCA(n_components=n_components, batch_size=2048)
    pca.fit(stacked)
    return pca, pca.transform(stacked)


def descriptors_to_histograms(
    pca_descriptors: Mapping[Path, np.ndarray],
    kmeans: MiniBatchKMeans,
    image_paths: Sequence[Path],
    class_names: Sequence[str],
) -> tuple[np.ndarray, np.ndarray]:
    """
    Converts a list of PCA-reduced descriptors into fixed-length histograms
    by assigning descriptors to nearest visual words and counting frequencies.
    """
    histograms, labels = [], []
    for path in image_paths:
        desc = pca_descriptors.get(path)
        if desc is None:
            # If no descriptors exist, fill histogram with zeros
            hist = np.zeros(kmeans.n_clusters, dtype=np.float32)
        else:
            words = kmeans.predict(desc)
            hist = np.bincount(words, minlength=kmeans.n_clusters).astype(np.float32)
            hist /= np.linalg.norm(hist) + 1e-6  # Normalise to unit length

        histograms.append(hist)
        labels.append(class_names.index(path.parent.name))

    return np.vstack(histograms), np.asarray(labels)


def optimise_bovw_hyperparams(
    all_desc_reduced: np.ndarray,
    desc_cache_reduced: Mapping[Path, np.ndarray],
    train_paths: Sequence[Path],
    val_paths: Sequence[Path],
```

```
    class_names: Sequence[str],
    *,
    n_trials: int = 100,
) -> dict[str, float]:
    """
    Uses Optuna to optimise the BoVW vocabulary size and the regularisation
    strength of a linear SVM classifier by cross-validating on a held-out set.
    """
    y_val = np.asarray([class_names.index(p.parent.name) for p in val_paths])

    def objective(trial: optuna.Trial) -> float:
        vocab_size = trial.suggest_int("vocab_size", 50, 500, step=50)
        svm_c = trial.suggest_float("svm_C", 1e-3, 1e2, log=True)

        kmeans = MiniBatchKMeans(
            n_clusters=vocab_size, batch_size=vocab_size * 5, max_iter=50, random_state
                =42
        ).fit(all_desc_reduced)

        X_tr, y_tr = descriptors_to_histograms(desc_cache_reduced, kmeans, train_paths,
            class_names)
        X_val, _ = descriptors_to_histograms(desc_cache_reduced, kmeans, val_paths,
            class_names)

        clf = SVC(kernel="linear", C=svm_c).fit(X_tr, y_tr)
        return clf.score(X_val, y_val)

    study = optuna.create_study(direction="maximize")
    study.optimize(objective, n_trials=n_trials, n_jobs=4)
    return study.best_params


# === 3. Deep-learning pipeline ===

class AlbumentationsFolder(datasets.ImageFolder):
    """
    A wrapper around torchvision.datasets.ImageFolder that applies Albumentations
    transformations during data loading.
    Useful for applying modern augmentations efficiently.
    """
    def __init__(self, root: str | Path, transform: Compose) -> None:
        super().__init__(root, transform=None)
        self.albumentations = transform

    def __getitem__(self, index: int):
        path, label = self.samples[index]  # get image path and label from dataset
        image = np.asarray(self.loader(path))  # load image and convert to numpy array
        image = self.albumentations(image=image)["image"]  # apply albumentations
        return image, label


def make_dataloaders(
    train_dir: Path,
    test_dir: Path,
    *,
    image_size: int = 224,
    batch_size: int = 64,
    num_workers: int = 8,
) -> tuple[DataLoader, DataLoader]:
    """
    Prepares training and testing dataloaders with augmentations for training
    and only resizing/normalisation for testing.
    """
    # Define training transformations
    train_tf = Compose([
        Resize(image_size, image_size),  # resize to consistent input size
        HorizontalFlip(p=0.5),  # apply horizontal flip randomly
        RandomBrightnessContrast(p=0.2),  # augment brightness/contrast
        Normalize(),  # scale input using ImageNet mean and std
```

```
        ToTensorV2(),  # convert to PyTorch tensor
    ])

    # Define validation/test transformations (no augmentation)
    test_tf = Compose([
        Resize(image_size, image_size),
        Normalize(),
        ToTensorV2(),
    ])

    # Wrap datasets with our augmentation-aware class
    train_ds = AlbumentationsFolder(train_dir, train_tf)
    test_ds = AlbumentationsFolder(test_dir, test_tf)

    # Configure dataloader parameters
    kwargs = dict(batch_size=batch_size, num_workers=num_workers, pin_memory=True,
        persistent_workers=True)
    return DataLoader(train_ds, shuffle=True, **kwargs), DataLoader(test_ds, shuffle=
        False, **kwargs)


@torch.inference_mode(False)  # ensure gradients are calculated unless otherwise
    specified
def cnn_epoch(
    model: nn.Module,
    loader: DataLoader,
    *,
    optimiser: optim.Optimizer | None = None,
    scaler: GradScaler | None = None,
    device: torch.device,
) -> float:
    """
    Runs one epoch of training or evaluation depending on whether an optimiser is passed
        .

    If training: returns mean cross-entropy loss.
    If evaluating: returns mean accuracy.
    """
    is_training = optimiser is not None  # check if training or eval mode
    model.train(is_training)
    metric_accumulator = 0.0  # running total for either loss or accuracy

    for x_batch, y_batch in loader:
        x_batch, y_batch = x_batch.to(device), y_batch.to(device)  # move data to device

        with autocast(device_type=device.type):  # use automatic mixed precision
            logits = model(x_batch)  # forward pass
            loss = nn.CrossEntropyLoss()(logits, y_batch)  # compute loss

        if is_training:
            assert scaler is not None and optimiser is not None  # safety check
            scaler.scale(loss).backward()  # scale and backpropagate
            scaler.step(optimiser)  # update weights
            scaler.update()  # update scaler state
            optimiser.zero_grad(set_to_none=True)  # reset gradients
            metric_accumulator += loss.item()
        else:
            preds = logits.argmax(1)  # get predicted classes
            metric_accumulator += (preds == y_batch).float().mean().item()  # batch
                accuracy

    return metric_accumulator / len(loader)  # return average across batches


def optimise_cnn_hyperparams(
    train_dir: Path, test_dir: Path, num_classes: int, *, n_trials: int = 100
) -> dict[str, float]:
    """
    Uses Optuna to perform hyperparameter optimisation over learning rate,
```

```
    weight decay, and dropout rate for fine-tuning a ResNet-50 model.
    """
    train_ld, val_ld = make_dataloaders(train_dir, test_dir, image_size=128)
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    def objective(trial: optuna.Trial) -> float:
        # Suggest values for each hyperparameter
        lr = trial.suggest_float("lr", 1e-5, 1e-3, log=True)
        wd = trial.suggest_float("weight_decay", 1e-8, 1e-3, log=True)
        dr = trial.suggest_float("dropout", 0.0, 0.5)

        # Load ResNet-50 and freeze all layers except final classification head
        net = models.resnet50(weights=models.ResNet50_Weights.DEFAULT)
        for p in net.parameters():
            p.requires_grad = False

        # Replace final fully connected layer with trainable head
        net.fc = nn.Sequential(nn.Dropout(dr), nn.Linear(net.fc.in_features, num_classes
            ))
        net.to(device)

        opt = optim.Adam(net.fc.parameters(), lr=lr, weight_decay=wd)  # optimiser for
            head only
        scaler = GradScaler()  # AMP scaler

        # Perform one training epoch and evaluate on validation set
        cnn_epoch(net, train_ld, optimiser=opt, scaler=scaler, device=device)
        val_acc = cnn_epoch(net, val_ld, device=device)
        return val_acc  # return validation accuracy as objective

    study = optuna.create_study(direction="maximize")  # aim to maximise accuracy
    study.optimize(objective, n_trials=n_trials)
    return study.best_params


# === 4. Grad-CAM visualisation ===

def gradcam_gallery(
    model: nn.Module,
    loader: DataLoader,
    class_names: Sequence[str],
    target_layer: nn.Module,
    *,
    num_examples: int = 8,
) -> None:
    """
    Visualises a selection of both correctly and incorrectly predicted images
    from the test set using Grad-CAM heatmaps.
    """
    device = next(model.parameters()).device  # determine device from model
    model.eval()
    cam = LayerGradCam(model, target_layer)
    dataset = loader.dataset  # assumes a well-formed dataset

    correct, incorrect = [], []  # lists to hold index of correct/incorrect predictions

    with torch.no_grad():
        for idx in range(len(dataset)):
            img, true_lbl = dataset[idx]  # get image and label
            pred = model(img.unsqueeze(0).to(device)).argmax().item()  # predict class
            (correct if pred == true_lbl else incorrect).append(idx)  # categorise

    # Randomly sample from both correct and incorrect predictions
    chosen = random.sample(correct, min(num_examples, len(correct))) + random.sample(
        incorrect, min(num_examples, len(incorrect))
    )
    random.shuffle(chosen)

    for idx in chosen:
```

```
        img_t, true_lbl = dataset[idx]
        img_t = img_t.to(device)
        pred_lbl = model(img_t.unsqueeze(0)).argmax().item()

        # Generate Grad-CAM heatmap
        attribution = cam.attribute(img_t.unsqueeze(0), target=pred_lbl).sum(1, keepdim=
            True)
        heat = F.interpolate(attribution, size=img_t.shape[-2:], mode="bilinear",
            align_corners=False)
        heat = torch.clamp(heat, min=0).squeeze().cpu().numpy()
        heat /= heat.max() + 1e-8

        # De-normalise image to convert back to displayable format
        mean = np.array([0.485, 0.456, 0.406])
        std = np.array([0.229, 0.224, 0.225])
        img_np = (img_t.cpu().permute(1, 2, 0).numpy() * std + mean).clip(0, 1)

        # Plot side-by-side: original image and heatmap overlay
        fig, (ax_img, ax_cam) = plt.subplots(1, 2, figsize=(8, 4))
        ax_img.imshow(img_np)
        ax_img.set_title(f"True : {class_names[true_lbl]}\nPred : {class_names[pred_lbl
            ]}")
        ax_img.axis("off")

        ax_cam.imshow(img_np)
        ax_cam.imshow(heat, cmap="jet", alpha=0.5)
        ax_cam.set_title(f"Grad-CAM ({target_layer._get_name()})")
        ax_cam.axis("off")
        plt.tight_layout()
        plt.show()


# === 5. Script Entry-Point ===

if __name__ == "__main__":
    # Define paths to dataset and processed folders (Google Drive & Colab layout)
    RAW_PATH = Path("/content/drive/MyDrive/colab/comp_vision")
    BASE_PATH = Path("/content/iCubWorldTransformations")
    TRAIN_PATH, TEST_PATH = BASE_PATH / "train", BASE_PATH / "test"
    PARTITIONS = [BASE_PATH / "part1_cropped", BASE_PATH / "part2_cropped"]
    IMG_SUFFIXES = {".png", ".jpg", ".jpeg"}  # supported image formats

    # Step 1: Prepare dataset by extracting archives and creating stratified splits
    mount_and_extract(RAW_PATH, BASE_PATH)
    stratified_split_to_folders(PARTITIONS, TRAIN_PATH, TEST_PATH, IMG_SUFFIXES)
    print("Dataset prepared")

    # Step 2: BoVW + SVM pipeline     compute SIFT features and optimise BoVW model
    class_labels = sorted([d.name for d in TRAIN_PATH.iterdir() if d.is_dir()])
    train_imgs_all = [
        p for cls in class_labels for p in (TRAIN_PATH / cls).iterdir()
        if p.suffix.lower() in IMG_SUFFIXES
    ]

    # Compute SIFT descriptors for training images
    sift_cache = compute_sift_descriptors(train_imgs_all)
    # Reduce descriptor dimensionality with PCA
    pca, stacked_reduced = fit_incremental_pca(list(sift_cache.values()))
    sift_cache_reduced = {p: pca.transform(d) for p, d in sift_cache.items()}

    # Split data into training and validation subsets for hyperparameter tuning
    paths_train, paths_val = train_test_split(
        list(sift_cache_reduced.keys()), test_size=0.2, random_state=42, shuffle=True
    )
    # Run Optuna to find best BoVW parameters
    best_bovw = optimise_bovw_hyperparams(
        stacked_reduced, sift_cache_reduced, paths_train, paths_val, class_labels
    )
    print(f"Best BoW params : {best_bovw}")
```

```
463
464     # Train final BoVW + SVM classifier with best parameters
465     kmeans_final = MiniBatchKMeans(
466         n_clusters=best_bovw["vocab_size"],
467         batch_size=best_bovw["vocab_size"] * 5,
468         max_iter=100,
469         random_state=42,
470     ).fit(stacked_reduced)
471     X_train, y_train = descriptors_to_histograms(sift_cache_reduced, kmeans_final,
        train_imgs_all, class_labels)
472     svm_final = SVC(kernel="linear", C=best_bovw["svm_C"]).fit(X_train, y_train)
473
474     # Evaluate SVM model on test data
475     test_paths = [
476         p for cls in class_labels for p in (TEST_PATH / cls).iterdir()
477         if p.suffix.lower() in IMG_SUFFIXES
478     ]
479     sift_test = compute_sift_descriptors(test_paths)
480     sift_test_reduced = {p: pca.transform(d) for p, d in sift_test.items()}
481     X_test, y_test = descriptors_to_histograms(sift_test_reduced, kmeans_final,
        test_paths, class_labels)
482     y_pred = svm_final.predict(X_test)
483
484     print(f"BoVW-SVM Test Accuracy : {accuracy_score(y_test, y_pred)*100:.2f}%")
485     print(classification_report(y_test, y_pred, target_names=class_labels))
486
487     # Show confusion matrix for SVM predictions
488     cm = confusion_matrix(y_test, y_pred)
489     plt.figure(figsize=(8, 6))
490     sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=class_labels,
        yticklabels=class_labels)
491     plt.title("BoVW-SVM Confusion Matrix")
492     plt.xlabel("Predicted"), plt.ylabel("True")
493     plt.tight_layout()
494     plt.show()
495
496     # Step 3: CNN hyperparameter optimisation using Optuna
497     best_cnn = optimise_cnn_hyperparams(TRAIN_PATH, TEST_PATH, len(class_labels))
498     print(f"Best CNN params : {best_cnn}")
499
500     # Step 4: Final CNN training with best parameters
501     train_ld, test_ld = make_dataloaders(TRAIN_PATH, TEST_PATH)
502     device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
503
504     model = models.resnet50(weights=models.ResNet50_Weights.DEFAULT)
505     for p in model.parameters():
506         p.requires_grad = False  # freeze backbone weights
507     model.fc = nn.Sequential(
508         nn.Dropout(best_cnn["dropout"]),
509         nn.Linear(model.fc.in_features, len(class_labels))
510     )
511     model.to(device)
512
513     optimiser = optim.Adam(model.fc.parameters(), lr=best_cnn["lr"], weight_decay=
        best_cnn["weight_decay"])
514     scaler = GradScaler()
515     tb = SummaryWriter("runs/final_experiment")  # log to TensorBoard
516
517     # Train with early stopping
518     patience, min_delta, best_val, epochs_no_improve = 3, 0.002, 0.0, 0
519     for epoch in range(1, 51):
520         train_loss = cnn_epoch(model, train_ld, optimiser=optimiser, scaler=scaler,
            device=device)
521         val_acc = cnn_epoch(model, test_ld, device=device)
522         tb.add_scalar("Loss/train", train_loss, epoch)
523         tb.add_scalar("Acc/val", val_acc, epoch)
524
525         print(f"Epoch {epoch:02d} | loss={train_loss:.4f} | val_acc={val_acc:.4f}")
526
```

```
        # Check for early stopping condition
        if val_acc > best_val + min_delta:
            best_val, epochs_no_improve = val_acc, 0
        else:
            epochs_no_improve += 1
            if epochs_no_improve >= patience:
                print("Early stopping     no improvement")
                break

    print(f"Final Val. Accuracy : {best_val:.4f}")

    # Final evaluation on test set
    all_preds, all_true = [], []
    with torch.no_grad():
        for xb, yb in test_ld:
            xb, yb = xb.to(device), yb.to(device)
            preds = model(xb).argmax(1)
            all_preds.extend(preds.cpu().numpy())
            all_true.extend(yb.cpu().numpy())

    final_acc = accuracy_score(all_true, all_preds)
    print(f"CNN Test Accuracy : {final_acc*100:.2f}%")
    print(classification_report(all_true, all_preds, target_names=class_labels))

    # Show confusion matrix for CNN predictions
    cm = confusion_matrix(all_true, all_preds)
    plt.figure(figsize=(8, 6))
    sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=class_labels,
        yticklabels=class_labels)
    plt.title("CNN Confusion Matrix")
    plt.xlabel("Predicted"), plt.ylabel("True")
    plt.tight_layout()
    plt.show()

    # Step 5: Run Grad-CAM to visualise CNN predictions
    gradcam_gallery(model, test_ld, class_labels, target_layer=model.layer3[-1])
```