CIFAR-10 and STL-10 Image Classification

1. Introduction

Object recognition is a fundamental task in computer vision with applications in robotics and autonomous systems. This report compares deep learning and traditional computer vision approaches for image classification on two benchmark datasets: CIFAR-10 and STL-10. Both datasets have 10 object classes that largely overlap, with nine common objects: airplane, bird, cat, deer, dog, horse, ship, truck, and car (termed automobile in CIFAR-10), but differ in that CIFAR10 includes frog while STL-10 includes monkey. CIFAR-10 originally contained 50,000 training images and 10,000 test images [1], but was subsampled to 5,000 training images (500 per class) and 8,000 test images (800 per class) to match STL10's size and maintain balanced class representation. STL-10 is used in its entirety, with 5,000 training images (500 per class) and 8,000 test images (800 per class) [2], featuring higher resolution 96x96 images sourced from ImageNet. CIFAR-10 and STL-10 were chosen to evaluate the methods across different image characteristics: CIFAR-10s 32x32 images are low-resolution with centered subjects and minimal background clutter, while STL-10s images often include complex backgrounds and clutter, posing greater challenges for feature detection and classification. Using both datasets provides insights into how each approach handles varying levels of image complexity and resolution.

2. EfficientNet B0

The deep learning approach utilized EfficientNetB0, a CNN designed for efficiency through compound scaling of depth, width, and resolution [3]. Pre-trained on ImageNet, EfficientNetB0 leverages transfer learning. pre-trained weights from a large dataset (ImageNet) are reused to improve performance on smaller target datasets (CIFAR-10, STL-10) by initializing with learned features. This architecture, with approximately 237 layers, processes inputs sequentially, as shown in Figure 1. Starting with a 224×224×3 input, a 3×3 convolution layer extracts initial features. MBConv1 blocks at stages 2-3 apply depth wise separable convolutions for efficient feature extraction, followed by batch normalization to stabilize training and Swish activation for smooth nonlinearity. MBConv6 blocks at stages 4–6 deepen the network, using skip connections to enhance gradient flow. Stage 7 applies a 1×1 convolution, followed by global average pooling 2D to a 1×1280 vector, reducing overfitting. The architecture ends with two fully connected (FC) layers and softmax for ten object classes.

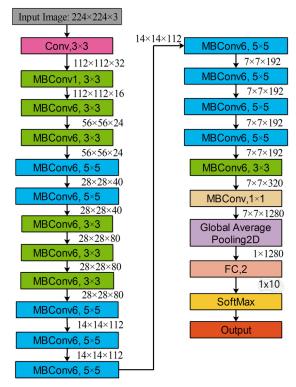


Fig. 1. EfficientNetB0 Architecture

EfficientNetB0's custom layers were tailored for this coursework. A resizing layer adjusted CIFAR-10 (32×32×3) and STL-10 (96×96×3) images to 224×224×3, matching EfficientNetB0s input requirements to leverage pretrained weights. A global average pooling layer minimized parameters, mitigating overfitting on small datasets . A dense layer with 128 units, ReLU activation, and He initialization captured patterns; 128 units were selected to provide sufficient capacity for the 10-class classification task. ReLU ensured non-linearity to learn complex features, and He initialization stabilized training by maintaining activation variance. Batch normalization accelerated convergence, and a tuned dropout layer prevented overfitting.

2.1 Data Preprocessing

224×224×3 match **Images** were resized to EfficientNetB0s input, a critical step for transfer learning. EfficientNets preprocessing function normalized pixel values to [- 1, 1] to align with pretrained weight distributions and ensure consistent feature extraction. Augmentation (i.e., $\pm 20^{\circ}$ rotations, 10% shifts, horizontal flips and 80%-120% contrast adjustments) improved robustness to variations (e.g., orientation, lighting) and reduced overfitting by increasing training diversity. A stratified 20% validation split yielded 4,000 training and 1,000 validation images (400 and 100 per class respectively).

2.2 Hyperparameter Exploration

Hyperparameter tuning involved a grid search to systematically explore learning rates, dropout rates, and batch sizes. Learning rates (0.001, 0.0005, 0.0001) were chosen based on typical transfer learning ranges for EfficientNetB0, starting with 0.001 to ensure rapid convergence of the pretrained model while testing lower rates to refine optimization and prevent overshooting on small datasets. Dropout rates (0.2, 0.3, 0.4) addressed overfitting risks, with 0.2 as a conservative baseline to maintain model capacity, increasing to 0.4 to counter the high risk of overfitting. Batch sizes (32, 64, 128) balanced gradient stability and computational efficiency, with 32 aligning with typical transfer learning setups for small datasets, 64 as a middle ground, and 128 testing larger batches for improved generalization. The Adam optimizer, chosen for its adaptive learning rates, optimized convergence; clipnorm=1.0 capped gradient norms to prevent instability with small batches, and weight decay=1e-4 promoted generalization by penalizing large weights. Training ran for 25 epochs, which proved adequate as the majority of runs stopped early (typically within 10-15 epochs) due to early stopping (patience of 5), indicating efficient convergence.

The best hyperparameter configurations are then fine-tuned by adhering to the standard EfficientNetB0 transfer learning protocol. Initially, all layers except the custom head were frozen. The last 25 layers were unfrozen for CIFAR-10 to adapt to its simpler, centered images, while the last 10 layers were unfrozen for STL-10 to handle its complex backgrounds, balancing adaptation and stability. Batch normalization layers remained frozen to preserve pre-trained statistics. A ReduceLROnPlateau scheduler adjusted the learning rate from 0.001, with fine-tuning extending to 40 epochs and early stopping (patience of 5). Implemented in Python with TensorFlow and Keras, the approach used a fixed seed (42) and epoch timing callbacks, ensuring reproducibility and monitoring.

2.3 Hyperparameter Analysis

The baseline EfficientNetB0 (learning rate=0.001, dropout=0.2, batch size=32) was trained and achieved 78.97% training accuracy, 82.70% validation accuracy, and 79.70% test accuracy. Augmentation drove generalization, but upscaling 32×32 images to 224×224 introduced artifacts, impacting classes like "cat" and "dog". STL-10 achieved 95.91% training, 94.60% validation, and 95.23% test accuracy. **Table 1** summarizes the hyperparameter tuning results, highlighting the best and worst configurations and performance ranges for both datasets.

Table 1: Key configurations from the 27-configuration grid search

STL-10 Test Accuracy (%)	95.74 (Best)	95.3	95.2	95.06 (Worst)
CIFAR-10 Test Accuracy (%)	82.27 (Best)	81.05	80.12	79.09 (Worst)
Batch Size	128	64	32	128
Dropout Rate	0.2	0.4	0.3	0.4
Learning Rate	0.001	0.001	0.0005	0.0001

For CIFAR-10, the best configuration achieved a test accuracy of 82.27%, improving the baseline test accuracy by 2.57%. The test accuracy spanned 79.09% to 82.27%, with a mean of approximately 80.98% and a standard deviation of 0.94%. The variance reflects CIFAR-10's sensitivity to hyperparameters, driven by upscaling artifacts and class overlaps. For STL-10, the best configuration achieved a test accuracy of 95.74%, a 0.51% improvement over the baseline. Test accuracies ranged from 95.06% to 95.74%, with a mean of 95.45% and a standard deviation of 0.22%. The low variance indicates robust performance, attributed to STL-10's higher resolution ImageNet-derived features.

Hyperparameter analysis shows that a learning rate of 0.001 outperformed lower rates (e.g., 79.09% for CIFAR-10, 95.30% for STL-10 at 0.0001), as higher rates better navigate the initial loss landscape. A dropout rate of 0.2 optimized generalization, with higher rates (0.4) reducing accuracies to 81.05% for CIFAR-10 and 95.06% for STL-10 due to excessive regularization. A batch size of 128 enhanced gradient stability, particularly for CIFAR-10, where noisy gradients benefit from larger batches .

2.4 Fine-Tuning Performance

Fine-tuning tailored the optimal EfficientNetB0 configurations to each dataset's unique characteristics, dataset-specific augmentation and partial leveraging of layers to enhance performance. For unfreezing CIFAR-10, the fine-tuned model achieved a test accuracy of 87.35%. This 9.65% improvement over the baseline reflects significant adaptation to CIFAR-10's low-resolution challenges. For STL-10, fine-tuning achieved a test accuracy of 95.91%. The classification report shown in Table 2.1 provides per-class insights of the fine-tuned model on CIFAR-10, where the macro-averaged precision, recall, and F1-score are 0.87, with "cat" having the lowest F1-score of 0.76 due to confusion with "dog" (F1=0.82). This is likely due to overlapping visual features like fur and shape. Stronger classes, such as "ship" (F1=0.94), benefit from distinct silhouettes.

The classification report shown in **Table 2.2** provides per-class insights of the fine-tuned model on STL-10, where the macro-averaged metrics are 0.96, with "bird" and "truck" excelling at F1=0.98, while "car" (F1=0.92, recall=0.89) underperforms slightly, likely due to variable backgrounds. CIFAR-10's deeper fine-tuning and adaptive learning rate scheduling explain its larger accuracy gain compared to STL-10's 0.68%, as more layers adapt to upscaling artifacts and class overlaps.

Table 2.1: Fine-tuned model classification report on CIFAR-10

Object	Precision	Recall	F1-Score
airplane	0.92	0.88	0.9
bird	0.91	0.95	0.93
automobile	0.85	0.84	0.85
cat	0.79	0.74	0.76
deer	0.87	0.81	0.84
dog	0.85	0.79	0.82
horse	0.81	0.95	0.87
frog	0.9	0.9	0.9
ship	0.93	0.95	0.94
truck	0.91	0.93	0.92
Macro Avg	0.87	0.87	0.87
Accuracy	0.87		

Table 2.2: Fine-tuned model classification report on STL-10

Object	Precision	Recall	F1-Score
airplane	0.97	0.97	0.97
bird	0.99	0.96	0.98
car	0.94	0.89	0.92
cat	0.94	0.97	0.95
deer	0.93	0.91	0.92
dog	0.93	0.96	0.95
horse	0.95	0.96	0.96
monkey	0.96	0.98	0.97
ship	0.95	0.96	0.96
truck	0.98	0.97	0.98
Macro avg	0.96	0.96	0.96
Accuracy	0.96		

The fine-tuning results provide granular insights into convergence as visualized in **Figure 2**. For CIFAR-10, training accuracy improved from 48.26% to 97.85%, with validation accuracy stabilizing at 87.90%. The ReduceLROnPlateau callback, reducing the learning rate four times due to stagnant validation loss, enabled precise weight updates and drove validation accuracy from 85.60% at Epoch 6 to 88.10%. Early stopping at Epoch 20 prevented overfitting. STL-10's training accuracy rose from 31.74% to

98.04% by Epoch 23, with validation accuracy peaking at 96.10% at Epoch 19. The fixed learning rate of 0.0001 ensured stability, but the lack of scheduling limited further gains. Generalization was maintained by the Early stopping at Epoch 23.

CIFAR-10's augmentation, incorporating 10% zoom, enhanced robustness to upscaling distortions by simulating variable object scales and contributed to the jump from 82.27% to 87.35% test accuracy. STL-10's broader $\pm 20^\circ$ rotation supported generalization without compromising its richer features, yielding high F1-scores.

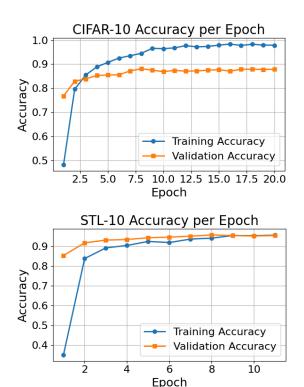


Fig. 2. Fine-Tuned Model performance on both datasets

2.5 Assessment and Future Directions

fine-tuned results underscore EfficientNetB0's adaptability to diverse datasets. STL-10's 95.91% test accuracy and balanced classification metrics in Table 2.2 highlight its suitability for transfer learning, leveraging 96×96 resolution and ImageNet-derived features. The low variance (standard deviation ≈0.2%,) and rapid convergence (validation accuracy reaching 95.70% by Epoch 9 in Figure 2) indicate that the pretrained weights require minimal The slight underperformance on "car" adjustment. (F1=0.92) suggests background clutter may obscure foreground features, which could be addressed by attention mechanisms such as CBAM to integrating prioritize relevant image regions. Adding augmentation, as used in CIFAR-10, could enhance STL-10's robustness to scale variations, potentially pushing accuracy beyond 96%.

CIFAR-10's fine-tuned 87.35% test accuracy, a 9.65% improvement over the baseline, reflects robust adaptation to low-resolution challenges, driven by unfreezing 25 layers, learning rate scheduling, and tailored augmentation. The classification report in Table 2.1 highlights persistent confusion between "cat" (F1=0.76) and "dog" (F1=0.82), where visual similarities lead to misclassifications, while classes like "ship" (F1=0.94) benefit from distinct features. The higher variance (standard deviation $\approx 0.94\%$) and slower convergence (validation accuracy peaking at 88.10% by Epoch 8 as shown in Figure 1) underscore the dataset's complexity, exacerbated by upscaling artifacts. The extended 25-epoch training, compared to fewer epochs in earlier experiments, allowed sustained validation accuracy gains and explains the improved 87.35%.

Future work could enhance CIFAR-10's performance by applying texture-enhancing augmentations such as edge detection filters to emphasize distinguishing features like fur patterns, potentially improving "cat" and "dog" F1-scores. For STL-10, ensemble methods combining EfficientNetB0 with ResNet50 could boost "car" performance by leveraging complementary feature representations. alternative architectures, such as ResNet50 alone, could validate EfficientNetB0's suitability or reveal better alternatives. Testing higher learning rates (e.g., 0.002) or larger batch sizes (e.g., 256) in future hyperparameter searches could uncover configurations better suited to CIFAR-10's complex feature space or STL-10's stable performance.

3. Traditional CV

This section covers two traditional computer vision approaches for object classification, employing the Bag of Words framework and comparing SIFT with Difference of Gaussians (DoG) against SIFT with Harris-Laplace. The BoW pipeline in both approaches comprise of five stages: (1) sparse keypoint detection to identify salient regions, (2) descriptor extraction to characterize these keypoints, (3) codebook generation to cluster descriptors, (4) histogram representation to encode image features, and (5) classification to assign category labels.

Sparse sampling is particularly effective for STL-10's complex, cluttered scenes and CIFAR-10's compact yet distinct object patterns, as it focuses on visually significant areas rather than dense, uniform sampling. Descriptors extracted from keypoints are processed using PCA to reduce dimensionality while retaining 90% of the variance. PCA enhances computational efficiency and ensures that the most informative components of the descriptors are retained, mitigating redundancy and noise. Codebooks are constructed using MiniBatchKMeans with a batch size of 4096, selected over traditional K-means for its scalability and reduced memory footprint when clustering vast

descriptor sets from both datasets. The batch size of 4096 was chosen as a practical compromise to ensure stable clustering without exhausting computational resources. Vocabulary sizes of 750, 1000, and 1250 were explored via grid search, allowing sufficient granularity to capture STL-10's diverse object appearances while promoting generalization across CIFAR-10's simpler, more uniform categories.

To address BoW's limitation in capturing spatial relationships, Spatial Pyramid Matching at level 1 is incorporated, providing a coarse representation of spatial layout without inflating dimensionality, which is a key advantage for CIFAR-10's low resolution and STL-10's cluttered images. L2 normalization is applied to the resulting histograms to ensure scale invariance and enhance the robustness of the subsequent classification step by mitigating the impact of varying feature magnitudes. For classification, SVM with a RBF kernel is employed due to its practical efficacy in high-dimensional, non-linear feature spaces produced by BoW histograms.

The grid search was performed manually on the test set by systematically evaluating all combinations hyperparameters. For SIFT with DoG, the grid included contrast thresholds (0.02, 0.04, 0.08) to control keypoint density, edge thresholds (7.5, 10, 12.5) to filter edge-related noise, vocabulary sizes (750, 1000, 1250) to adjust codebook granularity, and SVM C values (0.1, 1.0, 10.0) to tune regularization. For SIFT with Harris-Laplace, the grid was extended to include additional parameters: block sizes (3 and 4 for STL-10; 1, 2, and 3 for CIFAR-10) to adjust the scale of corner detection. sensitivity k values (0.05 and 0.07 for STL-10; 0.04 and 0.06 for CIFAR-10) to control corner strength, and sigma values ([1.0, 2.0, 4.0] for STL-10; [1.0, 1.5, 2.0] for CIFAR-10) to manage scale-space smoothing. The grid also maintained the same contrast thresholds (0.02, 0.04, 0.08), edge thresholds (7.5, 10, 12.5), vocabulary sizes (750, 1000, 1250), and SVM C values (0.1, 1.0, 10.0) as used in the DoG-based configuration.

3.1 DOG & Harris-Laplace

SIFT with DoG identifies keypoints by detecting extrema in the DoG scale space, an efficient approximation of the Laplacian-of-Gaussian (LoG). This method ensures scale invariance, making it well-suited for STL-10's diverse object scales and CIFAR-10's smaller, uniform objects. DoG operates by subtracting Gaussian-blurred images at adjacent scales, highlighting regions with significant contrast or texture changes, and is ideal for sparse, robust keypoint detection. The grid search tuned contrast thresholds to adjust keypoint density, with lower values retaining more points in STL-10's low-contrast scenes and higher values prioritizing quality in CIFAR-10's simpler images. Edge thresholds filter noisy keypoints near edges, with higher values reducing clutter in STL-10 and lower values preserving detail in

CIFAR-10. Descriptors, computed over 16x16 regions with orientation normalization provide rotation-invariant gradient patterns that are critical for both datasets' viewpoint variations.

SIFT with Harris-Laplace integrates multi scale Harris corner detection with LoG scale selection, combining precise spatial localization with scale invariance. Harris identifies corners via a second-moment matrix, detecting regions with multidirectional intensity changes, filtered at a 1% response threshold for sparsity. LoG assigns scales by locating extrema across smoothed scales and is proficient for handling STL-10's larger objects and CIFAR-10's finer details. The previously mentioned grid search tailored both CV approaches to each dataset's scale and complexity.

3.2 SIFT (DoG) Results

The best SIFT with DoG configuration on CIFAR-10 achieved an accuracy of 0.2559 with a contrast threshold of 0.04, edge threshold of 10.0, vocabulary size of 750, and SVM parameters C=1.0. **Table 4.1** summarizes the accuracy for a subset of configurations, selected to include the best performance and illustrate hyperparameter trends.

Table 4.1: Accuracy of SIFT with DoG on CIFAR-10 for a Subset of Configurations

Contrast Thresh.	Edge Thresh.	Vocab Size	C=0.1	C=1.0	C=10.0
0.02	7.5	750.0	0.2114	0.2519	0.2456
0.02	10.0	1000.0	0.2124	0.2547	0.2509
0.04	10.0	750.0	0.2221	0.2559	0.251
0.04	12.5	1250.0	0.2171	0.2542	0.2449
0.08	7.5	1000.0	0.2161	0.2442	0.233

The classification report for the best SIFT with DOG configuration in **Table 4.2** shows modest performance, with airplane (F1=0.34) and truck (F1=0.32) performing relatively well, while bird and deer (F1=0.18) are the most challenging.

Table 4.2: Classification Report for Best SIFT with DoG Configuration on CIFAR-10

Class	Precision	Recall	F1-Score
Airplane	0.37	0.32	0.34
Automobile	0.34	0.28	0.31
Bird	0.20	0.17	0.18
Cat	0.19	0.20	0.20
Deer	0.16	0.20	0.18
Dog	0.23	0.26	0.25
Frog	0.21	0.21	0.21
Horse	0.28	0.26	0.27
Ship	0.33	0.32	0.33
Truck	0.29	0.34	0.32
Macro Avg	0.26	0.26	0.26
Weighted Avg	0.26	0.26	0.26

The best SIFT with DoG configuration on STL-10 achieved an accuracy of 0.3877 with a contrast threshold of 0.02, edge threshold of 12.5, vocabulary size of 750, and SVM parameters C=1.0. **Table 4.3** summarizes the accuracy for a subset of configurations, selected to include the best performance and illustrate hyperparameter trends for brevity.

Table 4.3: Accuracy of SIFT with DoG on STL-10 for a Subset of Configurations

Contrast Thresh.	Edge Thresh.	Vocab Size	C=0.1	C=1.0	C=10.0
0.02	7.5	1000.0	0.2731	0.3695	0.3678
0.02	12.5	750.0	0.2825	0.3877	0.3812
0.04	10.0	750.0	0.2864	0.3759	0.3685
0.04	12.5	1050.0	0.2484	0.3777	0.3723
0.08	10.0	1250.0	0.2301	0.3438	0.3364

The classification report for the best configuration in **Table 4.4** shows improved performance compared to CIFAR10, with airplane (F1=0.56) and ship (F1=0.49) performing strongly, while automobile (F1=0.24) and dog (F1=0.28) are less accurate.

Table 4.4: Classification Report for Best SIFT with DoG Configuration on **STL-10**

Class	Precision	Recall	F1-Score
Airplane	0.58	0.55	0.56
Automobile	0.30	0.20	0.24
Bird	0.42	0.49	0.46
Cat	0.35	0.33	0.34
Deer	0.34	0.41	0.37
Dog	0.27	0.29	0.28
Frog	0.35	0.35	0.35
Horse	0.30	0.32	0.31
Ship	0.49	0.50	0.49
Truck	0.49	0.44	0.46
Macro Avg	0.39	0.39	0.39
Weighted Avg	0.39	0.39	0.39

4.3 SIFT (Harris-Laplace) Results

The best SIFT with Harris-Laplace configuration on CIFAR-10 achieved an accuracy of 0.4031 with a block size of 3, k=0.06, contrast threshold of 0.02, edge threshold of 7.5, vocabulary size of 1250, and SVM parameters C=1.0. Table **4.5** summarizes the accuracy for a subset of configurations, selected to include the best performance and illustrate hyperparameter trends for brevity.

Table 4.5: Classification Report for Best SIFT with DoG Configuration on CIFAR-10

Block Size	k	Contrast	Edge	Vocab Size	C=0.1	C=1.0
2	0.04	0.02	7.5	1250	0.3084	0.3915
2	0.06	0.02	7.5	1250	0.2990	0.3760
3	0.04	0.08	12.5	1000	0.3142	0.4006
3	0.06	0.02	7.5	1250	0.3114	0.4031
3	0.06	0.04	10.0	1250	0.3114	0.4031

The classification report for the best configuration in **Table 4.6** shows balanced performance, with ships (F1=0.53) and automobiles (F1=0.49) performing well, and cats (F1=0.28) and deer (F1=0.29) showing lower performance.

Table 4.6: Classification Report for Best SIFT with Harris-Laplace Configuration on CIFAR-10

Class	Precision	Recall	F1-Score
Airplane	0.49	0.42	0.45
Automobile	0.50	0.47	0.49
Bird	0.33	0.29	0.31
Cat	0.27	0.29	0.28
Deer	0.31	0.27	0.29
Dog	0.33	0.39	0.36
Frog	0.37	0.45	0.41
Horse	0.51	0.38	0.43
Ship	0.51	0.55	0.53
Truck	0.45	0.51	0.48
Macro Avg	0.41	0.40	0.40
Weighted Avg	0.41	0.40	0.40

The best SIFT with Harris-Laplace configuration on STL-10 achieved an accuracy of 0.5051 with a block size of 4, k=0.05, contrast threshold of 0.04, edge threshold of 7.5, vocabulary size of 1000, and SVM parameters C=10.0. **Table 4.7** summarizes the accuracy for a subset of configurations, selected to include the best performance and illustrate hyperparameter trends for brevity.

Table 4.7: Accuracy of SIFT with Harris-Laplace on STL-10 for a Subset of Configurations

k	Contrast	Edge	Vocab Size	C=0.1	C=1.0	C=10.0
0.05	0.02	7.5	1000	0.4015	0.4818	0.4843
0.05	0.04	10.0	1250	0.4041	0.4799	0.4816
0.07	0.02	7.5	1250	0.3970	0.4745	0.4714
0.05	0.02	10.0	1000	0.4111	0.5021	0.5051
0.05	0.04	7.5	1000	0.4111	0.5021	0.5051

The classification report for the best configuration shown in **Table 4.8** shows strong performance for ship (F1=0.67) and airplane (F1=0.63), driven by their distinct edges, while cat (F1=0.35) and dog (F1=0.36) are less accurate due to texture complexity. The balanced macro average (F1=0.50) and accuracy (0.5051) reflect robust performance, enhanced by the larger block size (4) and moderate vocabulary size (1000) on STL-10s 96×96 images.

Table 4.8: Classification Report for Best SIFT with Harris-Laplace Configuration on STL-10

Class	Precision	Recall	F1-Score
Airplane	0.59	0.69	0.63
Automobile	0.42	0.43	0.42
Bird	0.59	0.60	0.59
Cat	0.36	0.33	0.35
Deer	0.46	0.48	0.47
Dog	0.37	0.35	0.36
Frog	0.55	0.55	0.55
Horse	0.41	0.37	0.39
Ship	0.66	0.69	0.67
Truck	0.61	0.57	0.59
Macro Avg	0.50	0.51	0.50
Weighted Avg	0.50	0.51	0.50

For CIFAR-10, SIFT with Harris-Laplace (accuracy: 0.4031) significantly outperforms SIFT with DoG (accuracy: 0.2559). Harris-Laplace's multi-scale corner detection captures more discriminative keypoints in the low resolution 32×32 images. The optimal DoG configuration uses a moderate contrast threshold (0.04) and edge threshold (10.0) with a smaller vocabulary size (750), while Harris-Laplace benefits from a larger vocabulary size (1250), indicating a richer visual dictionary enhances its discriminative power. For STL-10, SIFT Harris-Laplace (accuracy: 0.5051) outperforms SIFT with DoG (accuracy: 0.3877). The larger block size (4) and moderate k (0.05) in the optimal Harris-Laplace configuration likely improve keypoint stability on STL-10s higher-resolution 96×96 images. The vocabulary size of 1000 strikes a balance between feature granularity and computational efficiency, with C=10.0 suggesting a tighter margin for SVM classification. The DoG configuration, with a lower contrast threshold (0.02) and smaller vocabulary size (750), is less effective, possibly due to fewer robust keypoints.

4.3 Comparison and Interpretation

both datasets, Harris-Laplace consistently outperforms DoG, likely due to its ability to detect corner like features across multiple scales, which is advantageous for both low- and high-resolution images. STL-10s higher resolution benefits both methods, but Harris-Laplace's performance gap is more pronounced, suggesting better adaptability to larger images. The SVM parameter C=1.0 or C=10.0 yields the highest accuracies, with C=0.1 underfitting. Classification reports show that classes with distinct edges (e.g., airplane, ship) achieve higher F1-scores, while textured classes (e.g., cat, deer) are challenging, particularly on CIFAR-10. The STL-10 Harris Laplace classification reports confirm this trend, with ship and airplane excelling, while cat and dog remain difficult.

5. Deep learning V.S. Traditional CV

The object recognition results demonstrate that the deep learning approach using EfficientNet B0 significantly outperforms traditional computer vision methods employing with Difference of Gaussians (DoG) and Harris-Laplace on both CIFAR-10 and STL-10 datasets. EfficientNet B0 achieved test accuracies of 87.35% on CIFAR-10 and 95.91% on STL-10, compared to 40.31% (Harris-Laplace) and 25.59% (DoG) on CIFAR-10, and 50.51% (Harris-Laplace) and 38.77% (DoG) on STL-10. This performance gap is driven by EfficientNet B0's use of transfer learning, leveraging pre-trained ImageNet weights to extract rich, adaptable features, while SIFT methods rely on hand-crafted, gradient-based features that struggle with CIFAR-10's low-resolution 32x32 images (F1-scores of 0.26 for DoG, 0.40 for Harris-Laplace) and only moderately improve on STL-10's 96x96 images (F1-scores of 0.39 for DoG, 0.50 for Harris-Laplace). Deep learning excels across all classes, capturing both edge and texture patterns (e.g., F1=0.97 for airplane, 0.95 for cat on STL-10), whereas traditional methods favor edge-distinct classes like ship (F1=0.67, Harris-Laplace on STL-10) but falter on textured classes like cat (F1=0.35) and dog (F1=0.36). Data augmentation and end-to-end optimization further enhance EfficientNet B0's generalization, unlike the manual tuning required for SIFT's feature extraction and SVM classification. While traditional CV offers computational efficiency, deep learning's superior adaptability and feature learning make it far more effective for complex image classification tasks in robotics and autonomous systems.

6. State of the Art in Computer Vision for Robotics

Contemporary deep learning methods in robotic vision encompass a diverse range of architectures tailored to specific tasks. CNNs remain the backbone of many vision-based robotic systems, excelling in object detection and image classification. Pre-trained models such as ResNet and DenseNet are commonly fine-tuned for robotic applications, achieving robust performance across various environments [4]. More recently, transformer architectures have been adapted for vision tasks through models like Vision Transformers (ViTs). These models process images as sequences of patches, offering scalability and robustness for complex scene understanding in robotics [5]. Additionally, deep generative models, including Variational Autoencoders, Generative Adversarial Networks, and Diffusion Models, are gaining traction for generating synthetic data and learning complex distributions for tasks like robotic grasping. These deep learning methods support a wide array of robotic applications. Object detection and recognition are vital for identifying and interacting with objects in real-time, with state-of-the-art models like YOLO and Faster R-CNN achieving high accuracy in diverse settings [6]. In navigation and path planning, deep learning enhances visual Simultaneous Localization and Mapping (SLAM) and obstacle avoidance, enabling robots to operate in dynamic environments [7].

Despite the transformative impact of deep learning, several challenges persist. One significant limitation is the requirement for large, annotated datasets, which are costly and time-consuming to acquire in robotics due to the variability of environments and tasks. To address this, researchers are exploring transfer learning, where models pre-trained on large datasets are fine-tuned on smaller, task-specific datasets, and synthetic data generation using simulation environments [8]. Generalization across different environments remains a concern, as models trained in specific settings may not perform well in new scenarios due to variations in lighting, object appearances, or other factors. Domain adaptation and robust training methods are active areas of research to improve model generalization [9].Recent advancements and emerging trends are shaping the future of robotic vision. Multimodal learning, which integrates vision with other sensory inputs such as language or tactile feedback, is gaining traction to enhance robot understanding and interaction. Neural Radiance Fields (NeRF) are being explored for high-fidelity 3D scene reconstruction, which can provide detailed environmental models for robotic navigation and manipulation [10]. These trends are expected to drive significant progress in the field, making robots more autonomous and capable in diverse, real-world settings.

References

- A. Krizhevsky, "Learning Multiple Layers of Features from Tiny Images," Jan. 2009,
 [Online]. Available: https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf
- [2] A. Coates, A. Ng, and H. Lee, 'An Analysis of Single-Layer Networks in Unsupervised Feature Learning', in Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, 11–13 Apr 2011, vol. 15, pp. 215–223.
- [3] M. Tan and Q. V. Le, "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks," arXiv.org, 2019. https://arxiv.org/abs/1905.11946
- [4] S. Sultana, Muhammad Mansoor Alam, Mazliham Mohd Su'ud, Jawahir Che Mustapha, and M. Prasad, "A Deep Dive into Robot Vision - An Integrative Systematic Literature Review Methodologies and Research Endeavor Practices," ACM computing surveys, vol. 56, no. 9, pp. 1–33, Apr. 2024, doi: https://doi.org/10.1145/3648357.
- A. Dosovitskiy et al., "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale," arXiv:2010.11929 [cs], Oct. 2020, Available: https://arxiv.org/abs/2010.11929
- [6] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," arXiv.org, Jun. 08, 2015. https://arxiv.org/abs/1506.02640
- [7] C. Cadena et al., "Past, Present, and Future of Simultaneous Localization And Mapping: Towards the Robust-Perception Age," *IEEE Transactions on Robotics*, vol. 32, no. 6, pp. 1309–1332, Dec. 2016, doi: https://doi.org/10.1109/TRO.2016.2624754
- [8] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel, "Domain Randomization for Transferring Deep Neural Networks from Simulation to the Real World," arXiv:1703.06907 [cs], Mar. 2017, Available: https://arxiv.org/abs/1703.06907
- E. Tzeng, J. Hoffman, K. Saenko, and T. Darrell, "Adversarial Discriminative Domain Adaptation," openaccess.thecvf.com, 2017.
 https://openaccess.thecvf.com/content_cvpr_2017/html/Tzeng_Adversarial_Discriminative_Domain_CVPR_2017_paper.html
- [10] B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng, "NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis," arXiv:2003.08934 [cs], Aug. 2020, Available: https://arxiv.org/abs/2003.08934

Appendix A

1. CIFAR-10 CNN Experimentation

```
import os
import warnings
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.preprocessing.image import
ImageDataGenerator
from tensorflow.keras.applications import
EfficientNetB0
from tensorflow.keras.applications.efficientnet
import preprocess input
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import Callback
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import itertools
from sklearn.metrics import classification report,
confusion matrix
from sklearn.model_selection import
train_test_split
import time
# Suppress TensorFlow warnings
os.environ["TF_CPP_MIN_LOG_LEVEL"] = "3"
os.environ["TF_XLA_FLAGS"] =
"--tf_xla_auto_jit=-1"
tf.config.optimizer.set jit(False)
warnings.filterwarnings("ignore",
category=UserWarning, module="keras")
# Set random seed for reproducibility
tf.random.set seed(42)
np.random.seed(42)
# Define class names
cifar10_classes = ['airplane', 'automobile', 'bird',
'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
NUM_CLASSES = 10
```

Custom callback for epoch timing class EpochTimeCallback(Callback):

def on epoch begin(self, epoch, logs=None):

```
self.start_time = time.time()
  def on epoch end(self, epoch, logs=None):
     end time = time.time()
     duration = end time - self.start time
     print(f"Epoch {epoch + 1} took {duration:.2f}
seconds")
# Load and preprocess CIFAR-10 with
subsampled training and test sets
def load_cifar10_data():
  Loads CIFAR-10, subsamples to 5,000 train
(500/class) and 8,000 test (800/class) (Kornblith
et al., 2019).
  try:
     (x_train, y_train), (x_test, y_test) =
cifar10.load_data()
     # Subsample training set to 5,000 (500 per
class)
     train_samples_per_class = 500
     x_train_sub, y_train_sub = [], []
     for cls in range(NUM CLASSES):
       cls_indices = np.where(y_train[:, 0] ==
cls)[0]
       selected indices =
np.random.choice(cls_indices,
train_samples_per_class, replace=False)
x_train_sub.append(x_train[selected_indices])
y_train_sub.append(y_train[selected_indices])
     x_train = np.vstack(x_train_sub)
     y_train = np.vstack(y_train_sub)
     # Subsample test set to 8,000 (800 per
class)
     test samples per class = 800
     x_test_sub, y_test_sub = [], []
     for cls in range(NUM_CLASSES):
       cls indices = np.where(y test[:, 0] ==
cls)[0]
       selected_indices =
np.random.choice(cls indices,
test_samples_per_class, replace=False)
x_test_sub.append(x_test[selected_indices])
y_test_sub.append(y_test[selected_indices])
     x_test = np.vstack(x_test_sub)
```

```
layers.Dense(NUM_CLASSES,
    y_test = np.vstack(y_test_sub)
                                                          activation='softmax')
    # Preprocess
                                                            1)
    x train = x train.astype('float32')
    x \text{ test} = x \text{ test.astype('float32')}
    x train = preprocess input(x train)
                                                          model.compile(optimizer=Adam(learning rate=lea
                                                          rning_rate, clipnorm=1.0, weight_decay=1e-4),
    x_test = preprocess_input(x_test)
    y_train = tf.keras.utils.to_categorical(y_train,
                                                                     loss='categorical crossentropy',
NUM CLASSES)
                                                                     metrics=['accuracy'])
                                                            return model
    y test = tf.keras.utils.to categorical(y test,
NUM CLASSES)
                                                          # Data augmentation
    # Verify class distribution in test set
                                                          def create data generator():
    class counts =
np.bincount(np.argmax(y test, axis=1))
                                                            Applies augmentation with contrast (Shorten &
     print("CIFAR-10 test set class distribution:",
                                                          Khoshqoftaar, 2019).
class_counts)
                                                            Rotation ±20°, shifts 10%, flips, brightness [0.8,
    assert all(count == test_samples_per_class
                                                          1.2].
for count in class counts), "Uneven class
distribution in CIFAR-10 test set"
                                                            return ImageDataGenerator(
                                                               rotation_range=20,
     print("CIFAR-10 x_train shape:",
                                                               width_shift_range=0.1,
                                                               height_shift_range=0.1,
x_train.shape, "y_train shape:", y_train.shape)
     print("CIFAR-10 x_test shape:",
                                                               horizontal_flip=True,
x test.shape, "y test shape:", y test.shape)
                                                               brightness range=[0.8, 1.2],
    return x_train, y_train, x_test, y_test
                                                               fill mode='nearest'
  except Exception as e:
                                                            )
    print(f"Error loading CIFAR-10: {str(e)}")
    return None, None, None, None
                                                          # Train and evaluate model
                                                          def train_model(model, x_train, y_train, x_val,
# Build EfficientNetB0 model
                                                         y_val, batch_size=32, epochs=25):
def build_model(learning_rate=0.001,
dropout rate=0.2):
                                                            Trains with augmentation, early stopping, and
                                                          epoch timing (Goodfellow et al., 2016).
  Constructs EfficientNetB0 with
BatchNormalization (Tan & Le, 2019; loffe &
                                                            datagen = create_data_generator()
Szegedy, 2015).
                                                            datagen.fit(x train)
  base_model =
                                                            early_stopping =
EfficientNetB0(weights='imagenet',
                                                          tf.keras.callbacks.EarlyStopping(
                                                               monitor='val_loss', patience=5,
include_top=False, input_shape=(224, 224, 3))
  base_model.trainable = False
                                                          restore_best_weights=True
  model = models.Sequential([
                                                            epoch_timer = EpochTimeCallback()
    layers.Input(shape=(32, 32, 3)),
    layers.Resizing(224, 224),
                                                            history = model.fit(
    base_model,
                                                               datagen.flow(x_train, y_train,
    layers.GlobalAveragePooling2D(),
                                                          batch size=batch size),
    layers.Dense(128, activation='relu',
                                                               validation_data=(x_val, y_val),
kernel initializer='he normal'),
                                                               epochs=epochs,
    layers.BatchNormalization(),
                                                               callbacks=[early_stopping, epoch_timer],
    layers.Dropout(dropout_rate),
                                                               verbose=1
```

```
plt.savefig(filename)
  return history
                                                              plt.close()
# Plot training results
                                                           # Run baseline and grid search for CIFAR-10
def plot results(history, title='CIFAR-10
                                                           def run cnn grid search():
EfficientNetB0 Performance'):
                                                              Runs baseline and grid search for CIFAR-10
                                                           with BatchNormalization (Sokolova & Lapalme,
  Generates loss/accuracy plots for report
visualization (Marking Criteria 2).
                                                           2009).
                                                              Baseline: LR=0.001, Dropout=0.2, BS=32.
  plt.figure(figsize=(12, 4))
                                                              Grid: LR=[0.001, 0.0005, 0.0001],
  plt.subplot(1, 2, 1)
                                                           Dropout=[0.2, 0.3, 0.4], BS=[32, 64, 128].
  plt.plot(history.history['loss'], label='Train Loss')
  plt.plot(history.history['val_loss'],
                                                              x_train, y_train, x_test, y_test =
label='Validation Loss')
                                                           load cifar10 data()
  plt.title(f'{title} - Loss')
                                                              classes = cifar10_classes
  plt.xlabel('Epoch')
  plt.ylabel('Loss')
                                                              if x train is None:
                                                                 return
  plt.legend()
  plt.subplot(1, 2, 2)
                                                              x_train, x_val, y_train, y_val = train_test_split(
  plt.plot(history.history['accuracy'], label='Train
                                                                 x_train, y_train, test_size=0.2,
Accuracy')
                                                           random_state=42, stratify=y_train
  plt.plot(history.history['val_accuracy'],
label='Validation Accuracy')
                                                              print("CIFAR-10 Train shape:", x train.shape,
                                                           "Validation shape:", x_val.shape)
  plt.title(f'{title} - Accuracy')
  plt.xlabel('Epoch')
  plt.ylabel('Accuracy')
                                                              # Baseline model
  plt.legend()
                                                              print("\n=== Training CIFAR-10 Baseline Model
                                                           (LR=0.001, Dropout=0.2, BS=32) ===")
                                                              baseline model =
  plt.savefig(f'{title.lower().replace(" ", "_")}.png')
  plt.close()
                                                           build_model(learning_rate=0.001,
                                                           dropout_rate=0.2)
# Generate confusion matrix
                                                              baseline_history =
def save_confusion_matrix(y_true, y_pred,
                                                           train_model(baseline_model, x_train, y_train,
classes, filename):
                                                           x_val, y_val, batch_size=32, epochs=25)
  Saves confusion matrix for detailed class
                                                              print("\nEvaluating baseline on test set...")
analysis (Marking Criteria 2).
                                                              baseline_loss, baseline_acc =
  ,,,,,,
                                                           baseline model.evaluate(x test, y test,
  cm = confusion_matrix(y_true, y_pred)
                                                           verbose=0)
  plt.figure(figsize=(10, 8))
                                                              print(f"Baseline Test Loss: {baseline_loss:.4f},
  plt.imshow(cm, interpolation='nearest',
                                                           Test Accuracy: {baseline acc:.4f}")
cmap=plt.cm.Blues)
  plt.title('Confusion Matrix')
                                                              y_pred = baseline_model.predict(x_test,
  plt.colorbar()
                                                           verbose=0)
  tick_marks = np.arange(len(classes))
                                                              y_pred_classes = np.argmax(y_pred, axis=1)
  plt.xticks(tick marks, classes, rotation=45)
                                                              y_true_classes = np.argmax(y_test, axis=1)
  plt.yticks(tick_marks, classes)
                                                              baseline report =
  plt.xlabel('Predicted Label')
                                                           classification_report(y_true_classes,
  plt.ylabel('True Label')
                                                           y_pred_classes, target_names=classes,
  plt.tight_layout()
                                                           output_dict=True, zero_division=0)
```

```
if (Ir in [0.001, 0.0001] and dr in [0.2, 0.4]
  baseline_f1 = baseline_report['weighted
                                                          and bs == 32) or (Ir == 0.0005 and dr == 0.3 and
avg']['f1-score']
                                                          bs == 64):
  plot_results(baseline_history, title='CIFAR-10
                                                                    title = f'CIFAR-10
Baseline EfficientNetB0')
                                                          LR={Ir}_DR={dr}_BS={bs}'
  save_confusion_matrix(y_true_classes,
                                                                    plot_results(history, title)
y_pred_classes, classes,
                                                                  print("\nEvaluating on test set...")
'confusion_matrix_baseline_cifar10.png')
                                                                  test loss, test acc =
  baseline result = {
                                                          model.evaluate(x_test, y_test, verbose=0)
                                                                  print(f"Test Loss: {test_loss:.4f}, Test
     'Model': 'Baseline',
    'Learning_Rate': 0.001,
                                                          Accuracy: {test_acc:.4f}")
     'Dropout Rate': 0.2,
    'Batch_Size': 32,
                                                                  y_pred = model.predict(x_test, verbose=0)
     'Test Loss': baseline loss,
                                                                  y pred classes = np.argmax(y pred,
     'Test Accuracy': baseline acc,
                                                          axis=1)
     'Avg_Precision': baseline_report['weighted
                                                                  y_true_classes = np.argmax(y_test,
                                                          axis=1)
avg']['precision'],
     'Avg_Recall': baseline_report['weighted
                                                                  report =
avg']['recall'],
                                                          classification_report(y_true_classes,
     'Avg_F1_Score': baseline_f1
                                                          y_pred_classes, target_names=classes,
                                                          output_dict=True, zero_division=0)
  }
                                                                  f1_score = report['weighted
pd.DataFrame([baseline_result]).to_csv('cifar10_b
                                                          avg']['f1-score']
aseline results.csv', index=False)
  print("Baseline results saved to
                                                                  if not results or f1_score > best_f1:
cifar10_baseline_results.csv")
                                                                    best_f1 = f1_score
                                                                    best_params = (lr, dr, bs)
  # Grid search
                                                                    best_model = model
  learning_rates = [0.001, 0.0005, 0.0001]
                                                                    best_y_pred = y_pred_classes
  dropout_rates = [0.2, 0.3, 0.4]
                                                                    best_y_true = y_true_classes
  batch_sizes = [32, 64, 128]
                                                                    save_confusion_matrix(y_true_classes,
                                                          y_pred_classes, classes,
  results = []
                                                          'confusion_matrix_best_cifar10.png')
  best f1 = 0.0
  best_params = None
                                                                  result = {
  best_model = None
                                                                     'Learning_Rate': Ir,
  best_y_pred = None
                                                                    'Dropout_Rate': dr,
  best_y_true = None
                                                                     'Batch_Size': bs,
                                                                     'Test Loss': test loss,
  for Ir, dr, bs in itertools.product(learning_rates,
                                                                    'Test_Accuracy': test_acc,
dropout_rates, batch_sizes):
                                                                    'Avg_Precision': report['weighted
     print(f'\n=== Training CIFAR-10 with LR: {Ir},
                                                          avg']['precision'],
Dropout: {dr}, Batch Size: {bs} ===')
                                                                     'Avg_Recall': report['weighted
                                                          avg']['recall'],
                                                                    'Avg_F1_Score': f1_score
    try:
       model = build_model(learning_rate=Ir,
                                                                  }
dropout_rate=dr)
                                                                  results.append(result)
       history = train_model(model, x_train,
                                                               except Exception as e:
y_train, x_val, y_val, batch_size=bs, epochs=25)
                                                                  print(f"Error in evaluation for LR={Ir},
                                                          DR={dr}, BS={bs}: {str(e)}")
```

```
print("\n=== Best Model Summary and Results
                                                          cifar10_grid_search_results.csv")
===")
  if best model and best params:
                                                             if results df.empty:
    Ir, dr, bs = best_params
                                                               print("Warning: CSV is empty! Check logs
    print(f"Best Model Parameters: LR={Ir},
                                                          and outputs.")
Dropout={dr}, Batch Size={bs}")
                                                               print("Output files:",
    best model.summary()
                                                          os.listdir('/kaggle/working'))
                                                               print("\nFinal Grid Search Results for
    print("\nGenerating classification report for
best model...")
                                                          CIFAR-10:")
    report = classification_report(best_y_true,
                                                               print(results_df.to_string(index=False))
best y pred, target names=classes,
output_dict=True, zero_division=0)
                                                          # Main execution
                                                          if name__ == '__main__':
     report df =
pd.DataFrame(report).transpose()
                                                             print("Starting CIFAR-10 EfficientNetB0
                                                          baseline and grid search with 8,000 test images
report df.to csv('classification report best cifar1
                                                          (800/class)...")
0.csv')
                                                             run_cnn_grid_search()
     print("Classification report saved to
classification_report_best_cifar10.csv")
     best result = next(r for r in results if
r['Learning_Rate'] == Ir and r['Dropout_Rate'] ==
dr and r['Batch Size'] == bs)
    print("\nBest Model Results:")
    print(f"Learning Rate:
{best result['Learning Rate']}")
    print(f"Dropout Rate:
{best_result['Dropout_Rate']}")
    print(f"Batch Size:
{best_result['Batch_Size']}")
    print(f"Test Loss:
{best_result['Test_Loss']:.4f}")
    print(f"Test Accuracy:
{best_result['Test_Accuracy']:.4f}")
     print(f"Average Precision:
{best result['Avg Precision']:.4f}")
    print(f"Average Recall:
{best result['Avg Recall']:.4f}")
     print(f"Average F1-Score:
{best_result['Avg_F1_Score']:.4f}")
  else:
    print("No best model found.")
  print("\nSaving CIFAR-10 grid search results to
CSV...")
  results_df = pd.DataFrame(results)
results_df.to_csv('cifar10_grid_search_results.csv
                                                              2. STL -10 CNN
```

print("Results saved to

Experimentation

', index=False)

```
seconds")
import os
import warnings
                                                          # Load and preprocess STL-10 using
import tensorflow as tf
                                                          tensorflow datasets
import tensorflow datasets as tfds # Added for
                                                          def load stl10 data():
STL-10 loading
from tensorflow.keras import layers, models
                                                            Loads STL-10 with 5,000 train (500/class) and
from tensorflow.keras.preprocessing.image import
                                                          8,000 test (800/class) using tfds.
ImageDataGenerator
from tensorflow.keras.applications import
                                                            try:
EfficientNetB0
                                                               # Load STL-10 dataset using
from tensorflow.keras.applications.efficientnet
                                                          tensorflow datasets
import preprocess input
                                                               ds_train, ds_test = tfds.load('stl10',
from tensorflow.keras.optimizers import Adam
                                                          split=['train', 'test'], as supervised=True,
from tensorflow.keras.callbacks import Callback
                                                          shuffle files=True)
import numpy as np
import matplotlib.pyplot as plt
                                                               # Convert to numpy arrays
import pandas as pd
                                                               x_train, y_train = [], []
import itertools
                                                               for image, label in tfds.as numpy(ds train):
from sklearn.metrics import classification report,
                                                                 x_train.append(image)
confusion matrix
                                                                 y train.append(label)
from sklearn.model_selection import
                                                               x_train = np.array(x_train)
train test split
                                                               y train = np.array(y_train)
import time
                                                               x_test, y_test = [], []
# Suppress TensorFlow warnings (including
                                                               for image, label in tfds.as_numpy(ds_test):
CUDA-related ones)
                                                                 x test.append(image)
os.environ["TF_CPP_MIN_LOG_LEVEL"] = "3"
                                                                 y_test.append(label)
os.environ["TF XLA FLAGS"] =
                                                               x_test = np.array(x_test)
"--tf xla auto jit=-1"
                                                               y_test = np.array(y_test)
tf.config.optimizer.set jit(False)
warnings.filterwarnings("ignore",
                                                               # Verify shapes
category=UserWarning, module="keras")
                                                               assert x_train.shape[0] == 5000, f"Expected
                                                          5000 training samples, got {x_train.shape[0]}"
# Set random seed for reproducibility
                                                               assert x_test.shape[0] == 8000, f"Expected
tf.random.set seed(42)
                                                          8000 test samples, got {x test.shape[0]}"
np.random.seed(42)
                                                               # STL-10 labels are 1-10, convert to 0-9 for
# Define class names for STL-10
                                                          consistency
stl10_classes = ['airplane', 'bird', 'car', 'cat', 'deer',
                                                               y_train = y_train - 1
'dog', 'horse', 'monkey', 'ship', 'truck']
                                                               y_test = y_test - 1
NUM_CLASSES = 10
                                                               # Preprocess
# Custom callback for epoch timing
                                                               x_train = x_train.astype('float32')
class EpochTimeCallback(Callback):
                                                               x \text{ test} = x \text{ test.astype('float32')}
  def on epoch begin(self, epoch, logs=None):
                                                               x_train = preprocess_input(x_train)
     self.start time = time.time()
                                                               x test = preprocess input(x test)
                                                               y_train = tf.keras.utils.to_categorical(y_train,
  def on_epoch_end(self, epoch, logs=None):
                                                          NUM CLASSES)
     end time = time.time()
                                                               y_test = tf.keras.utils.to_categorical(y_test,
     duration = end time - self.start time
                                                          NUM CLASSES)
```

print(f"Epoch {epoch + 1} took {duration:.2f}

```
# Data augmentation
    # Verify class distribution in test set
                                                          def create_data_generator():
    class counts =
np.bincount(np.argmax(y_test, axis=1))
                                                            Applies augmentation with contrast.
     print("STL-10 test set class distribution:",
                                                            Rotation ±20°, shifts 10%, flips, brightness [0.8,
class_counts)
                                                          1.2].
    assert all(count == 800 for count in
class counts), "Uneven class distribution in
                                                            return ImageDataGenerator(
STL-10 test set"
                                                               rotation range=20,
                                                               width shift range=0.1,
    print("STL-10 x_train shape:", x_train.shape,
                                                               height_shift_range=0.1,
"y_train shape:", y_train.shape)
                                                               horizontal_flip=True,
     print("STL-10 x test shape:", x test.shape,
                                                               brightness range=[0.8, 1.2],
"y_test shape:", y_test.shape)
                                                               fill mode='nearest'
     return x train, y train, x test, y test
  except Exception as e:
    print(f"Error loading STL-10: {str(e)}")
                                                          # Train and evaluate model
    return None, None, None, None
                                                          def train model(model, x train, y train, x val,
                                                          y_val, batch_size=32, epochs=25):
# Build EfficientNetB0 model
def build_model(learning_rate=0.001,
                                                            Trains with augmentation, early stopping, and
dropout rate=0.2):
                                                          epoch timing.
  Constructs EfficientNetB0 with
                                                            datagen = create_data_generator()
BatchNormalization.
                                                            datagen.fit(x train)
                                                            early_stopping =
  base model =
EfficientNetB0(weights='imagenet',
                                                          tf.keras.callbacks.EarlyStopping(
include_top=False, input_shape=(224, 224, 3))
                                                               monitor='val_loss', patience=5,
  base_model.trainable = False
                                                          restore_best_weights=True
                                                            )
  model = models.Sequential([
                                                            epoch_timer = EpochTimeCallback()
    layers.Input(shape=(96, 96, 3)), # STL-10
images are 96x96
                                                            history = model.fit(
    layers.Resizing(224, 224),
                                                               datagen.flow(x_train, y_train,
    base model,
                                                          batch_size=batch_size),
    layers.GlobalAveragePooling2D(),
                                                               validation_data=(x_val, y_val),
    layers.Dense(128, activation='relu',
                                                               epochs=epochs,
kernel_initializer='he_normal'),
                                                               callbacks=[early_stopping, epoch_timer],
    layers.BatchNormalization(),
                                                               verbose=1
    layers.Dropout(dropout_rate),
    layers.Dense(NUM_CLASSES,
                                                            return history
activation='softmax')
  ])
                                                          # Plot training results
                                                          def plot_results(history, title='STL-10
                                                          EfficientNetB0 Performance'):
model.compile(optimizer=Adam(learning_rate=lea
rning_rate, clipnorm=1.0, weight decay=1e-4).
                                                            Generates loss/accuracy plots for report
           loss='categorical crossentropy',
                                                         visualization.
           metrics=['accuracy'])
  return model
                                                            plt.figure(figsize=(12, 4))
                                                            plt.subplot(1, 2, 1)
```

```
plt.plot(history.history['loss'], label='Train Loss')
                                                              x_train, y_train, x_test, y_test =
  plt.plot(history.history['val_loss'],
                                                           load_stl10_data()
label='Validation Loss')
                                                              classes = stl10 classes
  plt.title(f'{title} - Loss')
  plt.xlabel('Epoch')
                                                              if x train is None:
  plt.ylabel('Loss')
                                                                return
  plt.legend()
                                                              x_train, x_val, y_train, y_val = train_test_split(
  plt.subplot(1, 2, 2)
                                                                x train, y train, test size=0.2,
  plt.plot(history.history['accuracy'], label='Train
                                                           random state=42
Accuracy')
                                                              )
  plt.plot(history.history['val_accuracy'],
                                                              print("STL-10 Train shape:", x_train.shape,
                                                           "Validation shape:", x val.shape)
label='Validation Accuracy')
  plt.title(f'{title} - Accuracy')
  plt.xlabel('Epoch')
                                                              # Baseline model
  plt.ylabel('Accuracy')
                                                              print("\n=== Training STL-10 Baseline Model
                                                           (LR=0.001, Dropout=0.2, BS=32) ===")
  plt.legend()
                                                              baseline model =
  plt.savefig(f'{title.lower().replace(" ", "_")}.png')
                                                           build model(learning rate=0.001,
  plt.close()
                                                           dropout rate=0.2)
                                                              baseline history =
# Generate confusion matrix
                                                           train_model(baseline_model, x_train, y_train,
def save_confusion_matrix(y_true, y_pred,
                                                           x_val, y_val, batch_size=32, epochs=25)
classes, filename):
                                                              print("\nEvaluating baseline on test set...")
                                                              baseline loss, baseline acc =
  Saves confusion matrix for detailed class
                                                           baseline_model.evaluate(x_test, y_test,
analysis.
  ,,,,,,
                                                           verbose=0)
  cm = confusion_matrix(y_true, y_pred)
                                                              print(f"Baseline Test Loss: {baseline_loss:.4f},
  plt.figure(figsize=(10, 8))
                                                           Test Accuracy: {baseline_acc:.4f}")
  plt.imshow(cm, interpolation='nearest',
cmap=plt.cm.Blues)
                                                              y_pred = baseline_model.predict(x_test,
  plt.title('Confusion Matrix')
                                                           verbose=0)
  plt.colorbar()
                                                              y_pred_classes = np.argmax(y_pred, axis=1)
  tick marks = np.arange(len(classes))
                                                              y_true_classes = np.argmax(y_test, axis=1)
  plt.xticks(tick_marks, classes, rotation=45)
                                                              baseline_report =
  plt.yticks(tick marks, classes)
                                                           classification_report(y_true_classes,
  plt.xlabel('Predicted Label')
                                                           y_pred_classes, target_names=classes,
  plt.ylabel('True Label')
                                                           output_dict=True, zero_division=0)
                                                              baseline f1 = baseline report['weighted
  plt.tight layout()
  plt.savefig(filename)
                                                           avg']['f1-score']
  plt.close()
                                                              plot results(baseline history, title='STL-10
                                                           Baseline EfficientNetB0')
# Run baseline and grid search for STL-10
def run_cnn_grid_search():
                                                              save_confusion_matrix(y_true_classes,
                                                           y pred classes, classes,
  Runs baseline and grid search for STL-10 with
                                                           'confusion_matrix_baseline_stl10.png')
BatchNormalization.
  Baseline: LR=0.001, Dropout=0.2, BS=32.
                                                              baseline result = {
  Grid: LR=[0.001, 0.0005, 0.0001],
                                                                'Model': 'Baseline',
Dropout=[0.2, 0.3, 0.4], BS=[32, 64, 128].
                                                                'Learning_Rate': 0.001,
                                                                'Dropout_Rate': 0.2,
```

```
'Batch_Size': 32,
                                                                  y_pred = model.predict(x_test, verbose=0)
     'Test_Loss': baseline_loss,
                                                                  y_pred_classes = np.argmax(y_pred,
     'Test Accuracy': baseline acc,
                                                          axis=1)
    'Avg_Precision': baseline_report['weighted
                                                                  y_true_classes = np.argmax(y_test,
avg']['precision'],
                                                          axis=1)
     'Avg Recall': baseline report['weighted
                                                                  report =
                                                          classification_report(y_true_classes,
avg']['recall'],
     'Avg_F1_Score': baseline_f1
                                                          y_pred_classes, target_names=classes,
                                                          output dict=True, zero division=0)
  }
                                                                  f1 score = report['weighted
pd.DataFrame([baseline_result]).to_csv('stl10_ba
                                                          avg']['f1-score']
seline_results.csv', index=False)
  print("Baseline results saved to
                                                                  if not results or f1 score > best f1:
stl10_baseline_results.csv")
                                                                    best_f1 = f1_score
                                                                    best params = (Ir, dr, bs)
  # Grid search
                                                                    best model = model
  learning_rates = [0.001, 0.0005, 0.0001]
                                                                    best_y_pred = y_pred_classes
  dropout rates = [0.2, 0.3, 0.4]
                                                                    best y true = y true classes
  batch_sizes = [32, 64, 128]
                                                                    save_confusion_matrix(y_true_classes,
                                                          y pred classes, classes,
  results = []
                                                          'confusion_matrix_best_stl10.png')
  best_f1 = 0.0
  best_params = None
                                                                  result = {
  best_model = None
                                                                    'Learning_Rate': Ir,
                                                                    'Dropout Rate': dr,
  best y pred = None
                                                                    'Batch Size': bs,
  best_y_true = None
                                                                    'Test_Loss': test_loss,
  for Ir, dr, bs in itertools.product(learning rates,
                                                                    'Test Accuracy': test acc,
dropout_rates, batch_sizes):
                                                                    'Avg_Precision': report['weighted
     print(f'\n=== Training STL-10 with LR: {Ir},
                                                          avg']['precision'],
Dropout: {dr}, Batch Size: {bs} ===')
                                                                    'Avg_Recall': report['weighted
                                                          avg']['recall'],
    try:
                                                                    'Avg_F1_Score': f1_score
       model = build_model(learning_rate=Ir,
dropout_rate=dr)
                                                                  results.append(result)
       history = train model(model, x train,
                                                               except Exception as e:
y_train, x_val, y_val, batch_size=bs, epochs=25)
                                                                  print(f"Error in evaluation for LR={Ir},
                                                          DR=\{dr\}, BS=\{bs\}: \{str(e)\}"\}
       if (Ir in [0.001, 0.0001] and dr in [0.2, 0.4]
and bs == 32) or (Ir == 0.0005 and dr == 0.3 and
                                                             print("\n=== Best Model Summary and Results
bs == 64):
                                                          ===")
          title = f'STL-10
                                                             if best model and best params:
LR={Ir}_DR={dr}_BS={bs}'
                                                               Ir, dr, bs = best_params
          plot_results(history, title)
                                                               print(f"Best Model Parameters: LR={Ir},
                                                          Dropout={dr}, Batch Size={bs}")
       print("\nEvaluating on test set...")
                                                               best_model.summary()
       test loss, test acc =
model.evaluate(x_test, y_test, verbose=0)
                                                               print("\nGenerating classification report for
       print(f"Test Loss: {test_loss:.4f}, Test
                                                          best model...")
Accuracy: {test_acc:.4f}")
```

```
report = classification_report(best_y_true,
best_y_pred, target_names=classes,
output dict=True, zero division=0)
     report df =
pd.DataFrame(report).transpose()
report_df.to_csv('classification_report_best_stl10.
csv')
     print("Classification report saved to
classification report best stl10.csv")
     best_result = next(r for r in results if
r['Learning Rate'] == Ir and r['Dropout Rate'] ==
dr and r['Batch_Size'] == bs)
     print("\nBest Model Results:")
     print(f"Learning Rate:
{best_result['Learning_Rate']}")
     print(f"Dropout Rate:
{best_result['Dropout_Rate']}")
     print(f"Batch Size:
{best_result['Batch_Size']}")
     print(f"Test Loss:
{best_result['Test_Loss']:.4f}")
     print(f"Test Accuracy:
{best result['Test Accuracy']:.4f}")
     print(f"Average Precision:
{best_result['Avg_Precision']:.4f}")
     print(f"Average Recall:
{best_result['Avg_Recall']:.4f}")
     print(f"Average F1-Score:
{best_result['Avg_F1_Score']:.4f}")
  else:
     print("No best model found.")
  print("\nSaving STL-10 grid search results to
CSV...")
  results df = pd.DataFrame(results)
results_df.to_csv('stl10_grid_search_results.csv',
index=False)
  print("Results saved to
stl10_grid_search_results.csv")
  if results_df.empty:
     print("Warning: CSV is empty! Check logs
and outputs.")
     print("Output files:",
os.listdir('/kaggle/working'))
  else:
     print("\nFinal Grid Search Results for
STL-10:")
     print(results_df.to_string(index=False))
```

```
# Main execution
if __name__ == '__main__':
    print("Starting STL-10 EfficientNetB0 baseline
and grid search with 8,000 test images
(800/class)...")
    run_cnn_grid_search()
```

3. CIFAR-10 (CNN FINE TUNING)

import tensorflow as tf from tensorflow.keras import layers, models

```
from tensorflow.keras.applications import
                                                           y_test = np.vstack(y_test_sub)
EfficientNetB0
from tensorflow.keras.applications.efficientnet
                                                           # Preprocess
                                                           x_train = x_train.astype('float32')
import preprocess input
from tensorflow.keras.preprocessing.image import
                                                           x_{test} = x_{test.astype}(float32')
ImageDataGenerator
                                                           x train = preprocess input(x train)
from tensorflow.keras.optimizers import Adam
                                                           x_test = preprocess_input(x_test)
from tensorflow.keras.callbacks import
                                                           y_train = tf.keras.utils.to_categorical(y_train,
ReduceLROnPlateau
                                                         NUM CLASSES)
                                                           y test = tf.keras.utils.to categorical(y test,
import numpy as np
from sklearn.model_selection import
                                                         NUM CLASSES)
train_test_split
                                                           return x_train, y_train, x_test, y_test
# Set random seed for reproducibility
tf.random.set seed(42)
                                                         # Data augmentation (adjusted for CIFAR-10)
np.random.seed(42)
                                                         def create data generator():
                                                           return ImageDataGenerator(
# Define number of classes
                                                              rotation range=10,
NUM CLASSES = 10
                                                              width_shift_range=0.1,
                                                              height shift range=0.1,
# Load and preprocess CIFAR-10 with stratified
                                                              horizontal_flip=True,
                                                              brightness_range=[0.8, 1.2],
subsampling
def load_cifar10_data():
                                                              zoom_range=0.1,
  (x_train, y_train), (x_test, y_test) =
                                                              fill_mode='nearest'
tf.keras.datasets.cifar10.load data()
                                                           )
  # Subsample to match original setup (5,000
                                                         # Build and fine-tune model
train, 8,000 test)
                                                         def build and finetune model():
  train_samples_per_class = 500
                                                           base_model =
  x_train_sub, y_train_sub = [], []
                                                         EfficientNetB0(weights='imagenet',
                                                         include top=False, input shape=(224, 224, 3))
  for cls in range(NUM CLASSES):
     cls_indices = np.where(y_train[:, 0] == cls)[0]
    selected indices =
                                                           # Freeze BatchNormalization layers
np.random.choice(cls indices,
                                                           for layer in base model.layers:
train_samples_per_class, replace=False)
                                                              if isinstance(layer,
                                                         layers.BatchNormalization):
x train sub.append(x train[selected indices])
                                                                layer.trainable = False
y_train_sub.append(y_train[selected_indices])
                                                           # Initially freeze all layers, then unfreeze the
                                                         last 25 layers
  x train = np.vstack(x train sub)
                                                           base_model.trainable = False
  y_train = np.vstack(y_train_sub)
                                                           for layer in base_model.layers[-25:]:
  test samples per class = 800
                                                              layer.trainable = True
  x_{test\_sub}, y_{test\_sub} = [], []
  for cls in range(NUM_CLASSES):
                                                           # Build model with original architecture
    cls_indices = np.where(y_test[:, 0] == cls)[0]
                                                           model = models.Sequential([
    selected_indices =
                                                              layers.Input(shape=(32, 32, 3)), # CIFAR-10
                                                         native size
np.random.choice(cls_indices,
test_samples_per_class, replace=False)
                                                              layers.Resizing(224, 224),
    x_test_sub.append(x_test[selected_indices])
                                                              base model,
    y_test_sub.append(y_test[selected_indices])
                                                              layers.GlobalAveragePooling2D(),
```

x_test = np.vstack(x_test_sub)

```
layers.Dense(128, activation='relu',
kernel_initializer='he_normal'),
    layers.BatchNormalization(),
    layers.Dropout(0.2), # Best dropout rate
    layers.Dense(NUM_CLASSES,
activation='softmax')
  ])
  # Compile with a higher learning rate and
adjusted scheduler
model.compile(optimizer=Adam(learning_rate=1e
-3, clipnorm=1.0, weight decay=1e-4),
           loss='categorical_crossentropy',
           metrics=['accuracy'])
  print("\nFine-tuning model for CIFAR-10 (final
attempt with stratified sampling)...")
  return model
# Train and evaluate model
def train_model(model, x_train, y_train, x_val,
y_val, x_test, y_test):
  datagen = create_data_generator()
  datagen.fit(x train)
  early_stopping =
tf.keras.callbacks.EarlyStopping(
    monitor='val_loss', patience=5,
restore_best_weights=True
  )
  reduce Ir = ReduceLROnPlateau(
     monitor='val_loss', factor=0.3, patience=2,
min Ir=1e-6, verbose=1
  )
  model.fit(
    datagen.flow(x_train, y_train,
batch_size=128), # Best batch size
    validation data=(x val, y val),
    epochs=40, # Increased epochs for
fine-tuning
    callbacks=[early_stopping, reduce_lr],
    verbose=1
  )
  loss, accuracy = model.evaluate(x_test, y_test,
verbose=0)
  print(f"CIFAR-10 Test Accuracy after final
fine-tuning with stratified sampling: {accuracy *
100:.2f}%")
```

```
# Generate classification report
  y_pred = model.predict(x_test, verbose=0)
  y test classes = np.argmax(y test, axis=1)
  y pred_classes = np.argmax(y_pred, axis=1)
  # Define class names for STL-10
  class_names = ['airplane', 'bird', 'car', 'cat',
'deer', 'dog', 'horse', 'monkey', 'ship', 'truck']
  print("\nClassification Report for CIFAR-10 Test
Set:")
  print(classification_report(y_test_classes,
y_pred_classes, target_names=class_names))
  return loss, accuracy
# Main fine-tuning script for CIFAR-10
def run finetuning():
  # Load dataset
  print("Loading CIFAR-10...")
  x_train, y_train, x_test, y_test =
load_cifar10_data()
  x_train, x_val, y_train, y_val = train_test_split(
     x_train, y_train, test_size=0.2,
random_state=42, stratify=y_train
  )
  # Build and fine-tune model
  model = build_and_finetune_model()
  train_model(model, x_train, y_train, x_val,
y_val, x_test, y_test)
if name == ' main ':
  run_finetuning()
```

4. STL-10 (CNN FINE TUNING)

import tensorflow as tf import tensorflow_datasets as tfds from tensorflow.keras import layers, models

```
from tensorflow.keras.applications import
                                                            return ImageDataGenerator(
EfficientNetB0
                                                               rotation_range=20,
from tensorflow.keras.applications.efficientnet
                                                               width shift range=0.1,
import preprocess input
                                                               height shift range=0.1,
from tensorflow.keras.preprocessing.image import
                                                               horizontal flip=True,
ImageDataGenerator
                                                               brightness range=[0.8, 1.2],
from tensorflow.keras.optimizers import Adam
                                                               fill_mode='nearest'
import numpy as np
                                                            )
from sklearn.model selection import
train test split
                                                         # Build and fine-tune model
                                                         def build_and_finetune_model():
from sklearn.metrics import classification report
                                                            base_model =
# Set random seed for reproducibility
                                                          EfficientNetB0(weights='imagenet',
                                                          include_top=False, input_shape=(224, 224, 3))
tf.random.set_seed(42)
np.random.seed(42)
                                                            # Freeze BatchNormalization layers
# Define number of classes
                                                            for layer in base_model.layers:
NUM CLASSES = 10
                                                               if isinstance(layer,
                                                          layers.BatchNormalization):
# Load and preprocess STL-10
                                                                 layer.trainable = False
def load_stl10_data():
  ds_train, ds_test = tfds.load('stl10', split=['train',
                                                            # Initially freeze all layers, then unfreeze the
'test'], as_supervised=True, shuffle_files=True)
                                                          last 10 layers
  x_train, y_train = [], []
                                                            base_model.trainable = False
  for image, label in tfds.as numpy(ds train):
                                                            for layer in base model.layers[-10:]:
    x train.append(image)
                                                               layer.trainable = True
    y_train.append(label - 1) # Adjust labels
from 1-10 to 0-9
                                                            # Build model with original architecture
  x_train, y_train = np.array(x_train),
                                                            model = models.Sequential([
                                                               layers.Input(shape=(96, 96, 3)), # STL-10
np.array(y_train)
                                                         native size
                                                               layers.Resizing(224, 224),
  x_test, y_test = [], []
  for image, label in tfds.as_numpy(ds_test):
                                                               base model,
    x_test.append(image)
                                                               layers.GlobalAveragePooling2D(),
    y test.append(label - 1)
                                                               layers.Dense(128, activation='relu',
  x_test, y_test = np.array(x_test),
                                                          kernel_initializer='he_normal'),
np.array(y_test)
                                                               layers.BatchNormalization(),
                                                               layers.Dropout(0.2), # Best dropout rate
  # Preprocess
                                                               layers.Dense(NUM_CLASSES,
  x train = x train.astype('float32')
                                                          activation='softmax')
  x_test = x_test.astype('float32')
                                                            ])
  x_train = preprocess_input(x_train)
  x test = preprocess input(x test)
                                                            # Compile with a higher learning rate for
  y_train = tf.keras.utils.to_categorical(y_train,
                                                         fine-tuning
NUM CLASSES)
  y test = tf.keras.utils.to categorical(y test,
                                                          model.compile(optimizer=Adam(learning rate=1e
                                                          -4, clipnorm=1.0, weight_decay=1e-4),
NUM_CLASSES)
                                                                     loss='categorical crossentropy',
                                                                     metrics=['accuracy'])
  return x_train, y_train, x_test, y_test
# Data augmentation
                                                            print("\nFine-tuning model for STL-10...")
```

return model

def create_data_generator():

```
# Train and evaluate model
def train model(model, x train, y train, x val,
y_val, x_test, y_test):
  datagen = create_data_generator()
  datagen.fit(x train)
  early_stopping =
tf.keras.callbacks.EarlyStopping(
     monitor='val loss', patience=5,
restore_best_weights=True
  )
  model.fit(
     datagen.flow(x train, y train,
batch size=128), # Best batch size
     validation_data=(x_val, y_val),
     epochs=40, # Increased epochs for
fine-tuning
     callbacks=[early_stopping],
     verbose=1
  )
  # Evaluate model
  loss, accuracy = model.evaluate(x test, y test,
verbose=0)
  print(f"STL-10 Test Accuracy after fine-tuning:
{accuracy * 100:.2f}%")
  # Generate classification report
  y pred = model.predict(x test, verbose=0)
  y_test_classes = np.argmax(y_test, axis=1)
  y_pred_classes = np.argmax(y_pred, axis=1)
  # Define class names for STL-10
  class_names = ['airplane', 'bird', 'car', 'cat',
'deer', 'dog', 'horse', 'monkey', 'ship', 'truck']
  print("\nClassification Report for STL-10 Test
Set:")
  print(classification report(y test classes,
y_pred_classes, target_names=class_names))
  return loss, accuracy
# Main fine-tuning script for STL-10
def run finetuning():
  # Load dataset
  print("Loading STL-10...")
  x_train, y_train, x_test, y_test =
load_stl10_data()
  x_train, x_val, y_train, y_val = train_test_split(
```

```
x_train, y_train, test_size=0.2,
random_state=42
)

# Build and fine-tune model
model = build_and_finetune_model()
train_model(model, x_train, y_train, x_val,
y_val, x_test, y_test)

if __name__ == '__main__':
    run_finetuning()
```

5. CIFAR-10 CV Experimentation

import cv2 import numpy as np

```
import matplotlib.pyplot as plt
                                                               expected_train_size, expected_test_size =
                                                          5000, 8000
import os
from sklearn.cluster import MiniBatchKMeans
                                                               expected per class = [500, 800]
from sklearn.decomposition import PCA
from sklearn.svm import SVC
                                                            else:
from sklearn.metrics import accuracy score,
                                                               raise ValueError("Dataset must be 'cifar10'.")
classification_report
from sklearn.model selection import
                                                            train_images = train_images.astype(np.uint8)
StratifiedShuffleSplit
                                                            test images = test images.astype(np.uint8)
from sklearn.preprocessing import
StandardScaler
                                                            print(f"Training set size: {len(train_images)}
import tensorflow_datasets as tfds
                                                          images")
                                                            print(f"Test set size: {len(test_images)}
# Load and sample CIFAR-10 dataset with
                                                          images")
stratified sampling
                                                            train class counts = np.bincount(train labels,
def load data(dataset='cifar10'):
                                                          minlength=10)
  if dataset == 'cifar10':
                                                            test_class_counts = np.bincount(test_labels,
    ds, info = tfds.load('cifar10', split=['train',
                                                          minlength=10)
'test'], as supervised=True, with info=True)
                                                            print("Training class distribution:",
    train ds, test ds = ds[0], ds[1]
                                                          train class counts)
                                                            print("Test class distribution:",
    train images, train labels = [], []
                                                          test_class_counts)
    test images, test labels = [], []
    for image, label in tfds.as_numpy(train_ds):
                                                            if len(train_images) != expected_train_size or
       train images.append(image)
                                                          len(test images) != expected test size:
                                                               raise ValueError(f"Unexpected dataset size.
       train labels.append(label)
    for image, label in tfds.as_numpy(test_ds):
                                                          Expected {expected_train_size} train and
       test images.append(image)
                                                          {expected test size} test images.")
       test_labels.append(label)
                                                            if not all(count == expected_per_class[0] for
    train_images = np.array(train_images)
                                                          count in train_class_counts if count > 0) or not
                                                          all(count == expected per class[1] for count in
    train labels = np.array(train labels)
                                                          test_class_counts if count > 0):
    test_images = np.array(test_images)
                                                               raise ValueError(f"Class distribution is not
    test_labels = np.array(test_labels)
                                                          balanced. Expected {expected per class[0]} per
    # Sample 5000 train images from the 50,000
                                                          class in train, {expected_per_class[1]} per class in
train set
                                                          test.")
    sss train = StratifiedShuffleSplit(n splits=1,
train size=5000, random state=42)
                                                            return train images, train labels, test images,
    train_indices, _ =
                                                          test_labels
next(sss train.split(train images, train labels))
    train_images = train_images[train_indices]
                                                          # Extract SIFT features with scale-invariant
    train_labels = train_labels[train_indices]
                                                          detection
                                                          def extract sift features(images,
                                                          contrast_threshold=0.04, edge_threshold=10):
    # Sample 8000 test images from the 10,000
test set
     sss test = StratifiedShuffleSplit(n splits=1,
                                                          cv2.SIFT create(contrastThreshold=contrast thre
train_size=8000, random_state=42)
                                                          shold, edgeThreshold=edge_threshold)
    test indices, =
                                                            descriptors = []
next(sss test.split(test images, test labels))
                                                            for img in images:
    test images = test images[test indices]
                                                               gray = cv2.cvtColor(img,
    test_labels = test_labels[test_indices]
                                                          cv2.COLOR_BGR2GRAY)
```

```
keypoints, des =
                                                           transformed_descriptors =
sift.detectAndCompute(gray, None)
                                                         pca.fit_transform(descriptors)
    if des is not None and len(keypoints) > 0:
                                                           return transformed descriptors, pca
       descriptors.append(des)
  return np.vstack(descriptors) if descriptors else
                                                         # Build BoW vocabulary
np.array([])
                                                         def build vocabulary(descriptors, k,
                                                         batch_size=4096, use_pca=False, pca_dims=59):
# Compute PCA variance ratio with normalization
                                                           if use pca:
def compute variance ratio(images,
                                                              descriptors, pca = apply pca(descriptors,
contrast threshold=0.04, edge threshold=10,
                                                         n components=pca dims)
dataset='cifar10'):
                                                           else:
  descriptors = extract_sift_features(images,
                                                              pca = None
contrast threshold, edge threshold)
                                                           kmeans = MiniBatchKMeans(n clusters=k,
  if len(descriptors) == 0:
                                                         n_init='auto', random_state=42,
    print("No descriptors found")
                                                         batch size=batch size)
    return 0
                                                           kmeans.fit(descriptors)
  scaler = StandardScaler()
                                                           return kmeans, pca
  descriptors = scaler.fit transform(descriptors)
  pca = PCA(n components=min(128,
                                                         # Compute spatial histograms with SPM (L2
descriptors.shape[0]-1), random state=42)
                                                         normalization only)
  pca.fit(descriptors)
                                                         def compute_spatial_histograms(images,
  total_var = np.sum(pca.explained_variance )
                                                         kmeans, pca=None, spm_levels=[1],
  if total_var == 0:
                                                         contrast threshold=0.04, edge threshold=10):
    print("Warning: Total variance is zero, check
                                                           sift =
data or normalization")
                                                         cv2.SIFT create(contrastThreshold=contrast thre
                                                         shold, edgeThreshold=edge threshold)
    return 0
  variance_ratio =
                                                           histograms = []
np.cumsum(pca.explained variance ratio )
                                                           for img in images:
  plt.plot(range(1, len(variance_ratio) + 1),
                                                              gray = cv2.cvtColor(img,
variance ratio, 'b-')
                                                         cv2.COLOR_BGR2GRAY)
  plt.title(f'Cumulative Explained Variance Ratio
                                                              keypoints, des =
({dataset.upper()})')
                                                         sift.detectAndCompute(gray, None)
  plt.xlabel('Number of Components')
                                                              if des is None or len(keypoints) == 0:
  plt.ylabel('Cumulative Explained Variance
Ratio')
                                                         histograms.append(np.zeros(kmeans.n clusters *
  plt.grid(True)
                                                         sum(lvl**2 for lvl in spm_levels)))
                                                                continue
plt.savefig(f'/content/variance ratio {dataset} con
                                                              if pca is not None:
trast{contrast_threshold}_edge{edge_threshold}.p
                                                                des = pca.transform(des)
ng')
                                                              words = kmeans.predict(des)
  plt.close()
  ninety_percent_idx = np.argmax(variance_ratio
                                                              h, w = gray.shape
>= 0.9) + 1
                                                              img hist = []
  print(f"90% variance at ~{ninety_percent_idx}
                                                              for grid_size in spm_levels:
components")
                                                                grid_h, grid_w = h // grid_size, w //
                                                         grid_size
  return ninety_percent_idx
                                                                level_hist = np.zeros((grid_size *
                                                         grid_size, kmeans.n_clusters))
# Apply PCA and return fitted PCA object
def apply_pca(descriptors, n_components=59):
                                                                for i in range(grid size):
  pca = PCA(n_components=n_components,
                                                                   for j in range(grid_size):
random_state=42)
                                                                     mask = []
```

```
best config = None
            for kp_idx, kp in
enumerate(keypoints):
                                                            best_predictions = None
               kp x, kp y = kp.pt
                                                            best true labels = None
               if (i * grid h <= kp y < (i + 1) *
grid_h) and (j * grid_w \le kp_x < (j + 1) * grid_w):
                                                            for contrast threshold in contrast thresholds:
                                                               for edge threshold in edge thresholds:
                 mask.append(True)
                                                                 print(f"\nContrast Threshold:
                                                          {contrast_threshold}, Edge Threshold:
                 mask.append(False)
                                                          {edge threshold}")
            mask = np.array(mask)
            grid words = words[mask]
                                                                 pca dims =
            if len(grid words) > 0:
                                                          compute_variance_ratio(train_images,
                                                          contrast_threshold, edge_threshold,
               hist, _ = np.histogram(grid_words,
                                                          dataset=dataset)
bins=range(kmeans.n clusters + 1),
density=True)
                                                                 if pca_dims == 0:
               level_hist[i * grid_size + j] = hist
                                                                    continue
       img hist.append(level hist.flatten())
    img_hist = np.concatenate(img_hist)
                                                                 vocab_sizes = [750, 1000, 1250]
                                                                 spm configs = [[1]]
    # L2 normalization
    img hist = img hist /
                                                                 results = []
(np.linalg.norm(img_hist) + 1e-10)
                                                                 for k in vocab_sizes:
    histograms.append(img hist)
                                                                    print(f"\nVocabulary size: {k}")
                                                                    train descriptors =
                                                          extract_sift_features(train_images,
  histograms = np.array(histograms)
  return histograms
                                                          contrast threshold, edge threshold)
                                                                    if len(train descriptors) == 0:
# Save results to a file
                                                                      print("No descriptors found")
def save results(results,
                                                                      continue
                                                                    kmeans, pca =
filename='/content/experiment_results_sift_cifar1
                                                          build_vocabulary(train_descriptors, k,
0.txt'):
                                                          batch size=4096, use pca=True,
  with open(filename, 'a') as f:
    for result in results:
                                                          pca_dims=pca_dims)
       f.write(result + '\n')
                                                                    for spm in spm_configs:
                                                                      print(f"\nSPM Levels: {spm},
# Save classification report to a file
def save_classification_report(report,
                                                          Normalization: L2")
filename='/content/classification report cifar10.txt
                                                                      train hist =
'):
                                                          compute_spatial_histograms(
  with open(filename, 'w') as f:
                                                                         train_images, kmeans, pca,
    f.write(report)
                                                          spm levels=spm,
                                                          contrast_threshold=contrast_threshold,
# Main pipeline
                                                                         edge_threshold=edge_threshold
def main():
                                                                      )
  dataset = 'cifar10'
                                                                      test_hist =
  print(f"\nProcessing {dataset.upper()}")
                                                          compute_spatial_histograms(
  train images, train labels, test images,
                                                                         test images, kmeans, pca,
test_labels = load_data(dataset=dataset)
                                                          spm levels=spm,
                                                          contrast_threshold=contrast_threshold,
                                                                         edge threshold=edge threshold
  contrast thresholds = [0.02, 0.04, 0.08]
  edge thresholds = [7.5, 10, 12.5]
  best_accuracy = 0.0
                                                                      # SVM with rbf kernel only
```

```
for C in [0.1, 1.0, 10.0]:
                                                          from sklearn.svm import SVC
               svm = SVC(C=C, kernel='rbf',
                                                          from sklearn.metrics import accuracy_score,
gamma='scale', random state=42)
                                                          classification report
               svm.fit(train hist, train labels)
                                                          from sklearn.preprocessing import
               pred = svm.predict(test hist)
                                                          StandardScaler
               acc = accuracy score(test labels,
                                                          from sklearn.model selection import
                                                          StratifiedShuffleSplit
pred)
               result = f"Dataset: {dataset},
                                                          import tensorflow_datasets as tfds
Contrast: {contrast threshold}, Edge:
{edge threshold}, Vocab Size: {k}, SPM: {spm},
                                                          # Load and subsample CIFAR-10 dataset with
Norm: L2, SVM (Kernel=rbf, C={C},
                                                          stratified sampling
Gamma=scale) Accuracy: {acc:.4f}"
                                                          def load_data(dataset='cifar10'):
                                                             if dataset != 'cifar10':
               print(result)
               results.append(result)
                                                               raise ValueError("Dataset must be 'cifar10'.")
               # Track best configuration
                                                            # Load CIFAR-10 dataset
                                                             ds, info = tfds.load('cifar10', split=['train', 'test'],
               if acc > best_accuracy:
                 best accuracy = acc
                                                          as supervised=True, with info=True)
                 best config = result
                                                            train ds, test ds = ds[0], ds[1]
                 best predictions = pred
                 best_true_labels = test_labels
                                                            # Convert to numpy arrays
                                                            train_images, train_labels = [], []
                                                            for image, label in tfds.as_numpy(train_ds):
       save_results(results)
                                                               train_images.append(image)
                                                               train labels.append(label)
  # Generate and save classification report for
the best configuration
                                                            train images = np.array(train images)
  if best_config is not None:
                                                            train_labels = np.array(train_labels)
    print(f"\nBest Configuration: {best config}")
    print("\nClassification Report for Best
                                                            test_images, test_labels = [], []
Configuration:")
                                                            for image, label in tfds.as_numpy(test_ds):
     class names = ['airplane', 'automobile', 'bird',
                                                               test images.append(image)
'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
                                                               test_labels.append(label)
                                                            test_images = np.array(test_images)
    report =
classification_report(best_true_labels,
                                                             test_labels = np.array(test_labels)
best_predictions, target_names=class_names)
    print(report)
                                                            # Subsample with stratification
    save classification report(report)
                                                            target train size, target test size = 5000,
                                                          8000
if __name__ == "__main__":
                                                             samples_per_class_train,
                                                          samples per class test = 500, 800
  main()
                                                            # Stratified subsampling for training set
                                                             sss train = StratifiedShuffleSplit(n splits=1,
                                                          train_size=target_train_size, random_state=42)
                                                            train_idx, _ = next(sss_train.split(train_images,
# Import libraries
                                                          train labels))
                                                            train_images = train_images[train_idx]
import cv2
import numpy as np
                                                            train_labels = train_labels[train_idx]
import matplotlib.pyplot as plt
import os
                                                            # Stratified subsampling for test set
from sklearn.cluster import MiniBatchKMeans
                                                             sss_test = StratifiedShuffleSplit(n_splits=1,
from sklearn.decomposition import PCA
                                                          train_size=target_test_size, random_state=42)
```

```
# Harris corner detection
  test_idx, _ = next(sss_test.split(test_images,
test_labels))
                                                               harris = cv2.cornerHarris(gray,
  test images = test images[test idx]
                                                          blockSize=block size, ksize=3, k=k)
  test labels = test labels[test idx]
                                                               # Threshold Harris response to get initial
                                                          keypoints
  # Convert to uint8 for OpenCV
                                                               coords = np.where(harris > 0.01 *
  train_images = train_images.astype(np.uint8)
                                                          harris.max())
  test_images = test_images.astype(np.uint8)
                                                               keypoints = [cv2.KeyPoint(float(x), float(y),
                                                          1.0) for y, x in zip(coords[0], coords[1])]
  # Verify class balance
  print(f"Training set size: {len(train_images)}
                                                               if not keypoints:
images")
                                                                  descriptors.append(np.array([]))
  print(f"Test set size: {len(test images)}
                                                                  continue
images")
  train class counts = np.bincount(train labels,
                                                               # Scale selection with LoG
minlength=10)
                                                               best sigma = None
  test_class_counts = np.bincount(test_labels,
                                                               best_response = -float('inf')
minlength=10)
                                                               for sigma in sigma values:
  print("Training class distribution:",
                                                                  log = cv2.GaussianBlur(gray, (0, 0),
train class counts)
                                                          sigma)
  print("Test class distribution:",
                                                                  log_response = np.zeros_like(gray,
                                                          dtype=float)
test_class_counts)
                                                                  for y, x in zip(coords[0], coords[1]):
  if len(train_images) != target_train_size or
                                                                    log_response[y, x] = log[y, x] if 0 \le y \le y
                                                          gray.shape[0] and 0 \le x \le \text{gray.shape}[1] else 0
len(test images) != target test size:
     raise ValueError(f"Unexpected dataset size.
                                                                  max response =
Expected {target_train_size} train and
                                                          np.max(log_response[coords])
{target test size} test images.")
                                                                  if max response > best response:
  if not all(count == samples_per_class_train for
                                                                    best_response = max_response
count in train_class_counts if count > 0):
                                                                    best_sigma = sigma
     raise ValueError(f"Training class distribution
is not balanced. Expected
                                                               # Compute SIFT descriptors with refined
{samples_per_class_train} per class.")
                                                          scale
  if not all(count == samples per class test for
                                                               sift =
count in test class counts if count > 0):
                                                          cv2.SIFT create(contrastThreshold=contrast thre
     raise ValueError(f"Test class distribution is
                                                          shold, edgeThreshold=edge_threshold)
not balanced. Expected {samples_per_class_test}
                                                               for kp in keypoints:
per class.")
                                                                  kp.size = best_sigma * 2
                                                               _, des = sift.compute(gray, keypoints)
                                                               if des is not None and len(des) > 0:
  return train images, train labels, test images,
                                                                  descriptors.append(des)
test_labels
                                                             return np.vstack(descriptors) if descriptors and
# Extract Harris keypoints and refine with LoG for
                                                          descriptors[0].size else np.array([])
def extract_harris_log_features(images,
                                                          # Compute PCA variance ratio with normalization
block size=3, k=0.04, sigma values=[1.0, 1.5,
                                                          def compute variance ratio(images,
                                                          block_size=3, k=0.04, sigma_values=[1.0, 1.5,
2.0], contrast_threshold=0.04,
edge threshold=10):
                                                          2.0], contrast threshold=0.04,
                                                          edge_threshold=10, dataset='cifar10'):
  descriptors = []
  for img in images:
                                                             descriptors =
     gray = cv2.cvtColor(img,
                                                          extract_harris_log_features(images, block_size,
cv2.COLOR_BGR2GRAY)
```

```
k, sigma_values, contrast_threshold,
                                                           kmeans = MiniBatchKMeans(n clusters=k,
                                                         n_init='auto', random_state=42,
edge_threshold)
                                                         batch_size=batch_size)
  if len(descriptors) == 0:
    print("No descriptors found")
                                                           kmeans.fit(descriptors)
    return 0
                                                           return kmeans, pca
  scaler = StandardScaler()
  descriptors = scaler.fit_transform(descriptors)
                                                         # Compute spatial histograms with SPM (L2
  pca = PCA(n components=min(128,
                                                         normalization only)
descriptors.shape[0]-1), random state=42)
                                                         def compute spatial histograms(images,
                                                         kmeans, pca=None, spm levels=[1],
  pca.fit(descriptors)
  total var = np.sum(pca.explained variance )
                                                         block_size=3, k=0.04, sigma_values=[1.0, 1.5,
  if total_var == 0:
                                                         2.0], contrast_threshold=0.04,
    print("Warning: Total variance is zero, check
                                                         edge threshold=10):
data or normalization")
                                                           histograms = []
    return 0
                                                           for img in images:
                                                              gray = cv2.cvtColor(img,
  variance ratio =
np.cumsum(pca.explained_variance_ratio_)
                                                         cv2.COLOR_BGR2GRAY)
  plt.plot(range(1, len(variance ratio) + 1),
                                                              # Harris corner detection for keypoints
variance ratio, 'b-')
                                                              harris = cv2.cornerHarris(gray,
  plt.title(f'Cumulative Explained Variance Ratio
                                                         blockSize=block size, ksize=3, k=k)
({dataset.upper()})')
                                                              coords = np.where(harris > 0.01 *
  plt.xlabel('Number of Components')
                                                         harris.max())
  plt.ylabel('Cumulative Explained Variance
                                                              keypoints = [cv2.KeyPoint(float(x), float(y),
Ratio')
                                                         1.0) for y, x in zip(coords[0], coords[1])]
  plt.grid(True)
                                                              if not keypoints:
plt.savefig(f'/kaggle/working/variance_ratio_{datas
et} block{block size} k{k} contrast{contrast thre
                                                         histograms.append(np.zeros(kmeans.n clusters *
shold\_edge{edge_threshold\.png')
                                                         sum(lvl**2 for lvl in spm_levels)))
                                                                continue
  plt.close()
  ninety percent idx = np.argmax(variance ratio
>= 0.9) + 1
                                                              # Scale selection with LoG
                                                              best sigma = None
  print(f"90% variance at ~{ninety_percent_idx}
components")
                                                              best_response = -float('inf')
  return ninety_percent_idx
                                                              for sigma in sigma values:
                                                                log = cv2.GaussianBlur(gray, (0, 0),
# Apply PCA and return fitted PCA object
                                                         sigma)
def apply pca(descriptors, n components=59):
                                                                log_response = np.zeros_like(gray,
  pca = PCA(n_components=n_components,
                                                         dtype=float)
random state=42)
                                                                for y, x in zip(coords[0], coords[1]):
  transformed descriptors =
                                                                   log_response[y, x] = log[y, x] if 0 \le y \le y
pca.fit_transform(descriptors)
                                                         gray.shape[0] and 0 \le x \le \text{gray.shape}[1] else 0
  return transformed descriptors, pca
                                                                max response =
                                                         np.max(log_response[coords])
# Build BoW vocabulary
                                                                if max_response > best_response:
def build_vocabulary(descriptors, k,
                                                                   best response = max response
batch_size=4096, use_pca=False, pca_dims=59):
                                                                   best_sigma = sigma
  if use pca:
     descriptors, pca = apply_pca(descriptors,
                                                              # Compute SIFT descriptors for this image
n_components=pca_dims)
  else:
                                                         cv2.SIFT_create(contrastThreshold=contrast_thre
    pca = None
                                                         shold, edgeThreshold=edge_threshold)
```

```
for kp in keypoints:
                                                          def save results(results,
       kp.size = best_sigma * 2
                                                          filename='/kaggle/working/experiment_results_ha
     , des = sift.compute(gray, keypoints)
                                                          rris sift cifar10.txt'):
                                                             with open(filename, 'a') as f:
                                                                for result in results:
     if des is None or len(des) == 0:
                                                                  f.write(result + '\n')
histograms.append(np.zeros(kmeans.n_clusters *
sum(lvl**2 for lvl in spm_levels)))
                                                          # Save classification report to a file
       continue
                                                          def save classification report(report,
                                                          filename='/kaggle/working/classification report h
     # Transform descriptors and predict words
                                                          arris sift cifar10.txt'):
                                                             with open(filename, 'w') as f:
for this image
     if pca is not None:
                                                               f.write(report)
       des = pca.transform(des)
     words = kmeans.predict(des)
                                                          # Main pipeline
                                                          def main():
                                                             dataset = 'cifar10'
     # Compute spatial histogram
     h, w = gray.shape
                                                             print(f"\nProcessing {dataset.upper()}")
     img hist = []
                                                             train_images, train_labels, test_images,
     for grid size in spm levels:
                                                          test labels = load data(dataset=dataset)
       grid_h, grid_w = h // grid_size, w //
                                                             block sizes = [2, 3]
grid_size
       level_hist = np.zeros((grid_size *
                                                             k values = [0.04, 0.06]
grid_size, kmeans.n_clusters))
                                                             sigma_values_options = [[1.0, 1.5, 2.0]]
       for i in range(grid size):
                                                             contrast thresholds = [0.02, 0.04, 0.08]
          for j in range(grid_size):
                                                             edge_thresholds = [7.5, 10, 12.5]
            mask = []
            for y, x in zip(coords[0], coords[1]):
                                                             best accuracy = 0.0
               if (i * grid_h <= y < (i + 1) * grid_h)
                                                             best_config = None
and (j * grid_w \leq x < (j + 1) * grid_w):
                                                             best predictions = None
                                                             best true labels = None
                 mask.append(True)
               else:
                 mask.append(False)
                                                             for block_size in block_sizes:
            mask = np.array(mask)
                                                               for k in k_values:
            grid words = words[mask] if
                                                                  for sigma values in
len(words) > 0 else np.array([])
                                                          sigma_values_options:
            if len(grid_words) > 0:
                                                                     for contrast_threshold in
                                                          contrast thresholds:
               hist, _ = np.histogram(grid_words,
bins=range(kmeans.n_clusters + 1),
                                                                       for edge_threshold in
density=True)
                                                          edge thresholds:
               level_hist[i * grid_size + j] = hist
                                                                          print(f"\nBlock Size: {block_size},
       img_hist.append(level_hist.flatten())
                                                          k: {k}, Sigma Values: {sigma_values}, Contrast
     img hist = np.concatenate(img hist)
                                                          Threshold: {contrast threshold}, Edge Threshold:
                                                          {edge_threshold}")
     # L2 normalization
                                                                          pca_dims =
                                                          compute variance ratio(train images,
     img hist = img hist /
(np.linalg.norm(img_hist) + 1e-10)
                                                          block_size, k, sigma_values, contrast_threshold,
     histograms.append(img_hist)
                                                          edge_threshold, dataset=dataset)
                                                                          if pca_dims == 0:
  return np.array(histograms)
                                                                            continue
# Save results to a file
                                                                          vocab_sizes = [750, 1000, 1250]
```

```
result = f"Dataset:
              spm_configs = [[1]]
                                                         {dataset}, Block Size: {block_size}, k: {k}, Sigma:
              results = []
                                                         {sigma values}, Contrast: {contrast threshold},
              for k vocab in vocab sizes:
                                                         Edge: {edge_threshold}, Vocab Size: {k_vocab},
                                                         SPM: {spm}, Norm: L2, SVM (Kernel=rbf, C={C},
                 print(f"\nVocabulary size:
                                                         Gamma=scale) Accuracy: {acc:.4f}"
{k_vocab}")
                 train descriptors =
                                                                               print(result)
extract_harris_log_features(train_images,
                                                                               results.append(result)
block size, k, sigma values, contrast threshold,
edge threshold)
                                                                               if acc > best accuracy:
                 if len(train_descriptors) == 0:
                                                                                  best_accuracy = acc
                   print("No descriptors found")
                                                                                  best_config = result
                   continue
                                                                                  best predictions = pred
                 kmeans, pca =
                                                                                  best_true_labels =
build vocabulary(train descriptors, k vocab,
                                                         test labels
batch_size=4096, use_pca=True,
pca_dims=pca_dims)
                                                                        save_results(results)
                                                           if best config is not None:
                 for spm in spm_configs:
                   print(f"\nSPM Levels: {spm},
                                                              print(f"\nBest Configuration: {best config}")
                                                              print("\nClassification Report for Best
Normalization: L2")
                                                         Configuration:")
                   train hist =
compute_spatial_histograms(
                                                              class_names = ['airplane', 'automobile', 'bird',
                                                         'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
                      train_images, kmeans,
                                                              report =
pca, spm levels=spm, block size=block size,
                                                         classification report(best true labels,
k=k,
                                                         best_predictions, target_names=class_names)
sigma_values=sigma_values,
                                                              print(report)
contrast_threshold=contrast_threshold,
                                                              save_classification_report(report)
                                                         if __name__ == "__main__":
edge_threshold=edge_threshold
                                                           main()
                   test_hist =
compute_spatial_histograms(
                      test images, kmeans, pca,
spm_levels=spm, block_size=block_size, k=k,
sigma values=sigma values,
contrast_threshold=contrast_threshold,
edge_threshold=edge_threshold
                   for C in [0.1, 1.0, 10.0]:
                      svm = SVC(C=C,
                                                             6. STL-10 CV Experimentation
kernel='rbf', gamma='scale', random_state=42)
                      svm.fit(train_hist,
                                                         # Import libraries
train_labels)
                                                         import cv2
```

import numpy as np

import os

import matplotlib.pyplot as plt

pred =

acc =

svm.predict(test_hist)

accuracy_score(test_labels, pred)

```
from sklearn.cluster import MiniBatchKMeans
                                                            print("Test class distribution:",
from sklearn.decomposition import PCA
                                                          test_class_counts)
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score,
                                                            if len(train images) != expected train size or
                                                          len(test images) != expected test size:
classification report
                                                               raise ValueError(f"Unexpected dataset size.
from sklearn.preprocessing import
StandardScaler
                                                          Expected {expected_train_size} train and
import tensorflow datasets as tfds
                                                          {expected test size} test images.")
                                                            if not all(count == expected per class[0] for
from sklearn.model selection import
StratifiedShuffleSplit
                                                          count in train class counts if count > 0) or not
                                                          all(count == expected_per_class[1] for count in
                                                          test_class_counts if count > 0):
# Load and verify STL-10 dataset
def load data(dataset='stl10'):
                                                               raise ValueError(f"Class distribution is not
  if dataset != 'stl10':
                                                          balanced. Expected {expected_per_class[0]} per
     raise ValueError("Dataset must be 'stl10'.")
                                                          class in train, {expected per class[1]} per class in
                                                          test.")
  ds, info = tfds.load('stl10', split=['train', 'test'],
as supervised=True, with info=True)
                                                            return train images, train labels, test images,
  train_ds, test_ds = ds[0], ds[1]
                                                          test labels
  train_images, train_labels = [], []
                                                          # Extract Harris keypoints and refine with LoG for
  for image, label in tfds.as_numpy(train_ds):
     train_images.append(image)
                                                          def extract_harris_log_features(images,
     train_labels.append(label)
                                                          block_size=3, k=0.04, sigma_values=[1.0, 2.0,
                                                          4.0], contrast threshold=0.04,
  train images = np.array(train images)
  train_labels = np.array(train_labels)
                                                          edge threshold=10):
                                                            descriptors = []
  test images, test labels = [], []
                                                            for img in images:
  for image, label in tfds.as_numpy(test_ds):
                                                               gray = cv2.cvtColor(img,
     test_images.append(image)
                                                          cv2.COLOR_BGR2GRAY)
     test labels.append(label)
                                                               # Harris corner detection
  test_images = np.array(test_images)
                                                               harris = cv2.cornerHarris(gray,
  test_labels = np.array(test_labels)
                                                          blockSize=block_size, ksize=3, k=k)
                                                               # Threshold Harris response to get initial
  expected_train_size, expected_test_size =
                                                          keypoints
5000, 8000
                                                               coords = np.where(harris > 0.01 *
  expected_per_class = [500, 800]
                                                          harris.max())
                                                               keypoints = [cv2.KeyPoint(float(x), float(y),
  train_images = train_images.astype(np.uint8)
                                                          1.0) for y, x in zip(coords[0], coords[1])]
  test images = test images.astype(np.uint8)
                                                               if not keypoints:
  print(f"Training set size: {len(train_images)}
                                                                  descriptors.append(np.array([]))
images")
                                                                  continue
  print(f"Test set size: {len(test_images)}
images")
                                                               # Scale selection with LoG
  train class counts = np.bincount(train labels,
                                                               best sigma = None
                                                               best_response = -float('inf')
minlength=10)
  test_class_counts = np.bincount(test_labels,
                                                               for sigma in sigma values:
                                                                  log = cv2.GaussianBlur(gray, (0, 0),
minlength=10)
  print("Training class distribution:",
                                                          sigma)
train_class_counts)
                                                                  log_response = np.zeros_like(gray,
```

dtype=float)

```
for y, x in zip(coords[0], coords[1]):
                                                           plt.ylabel('Cumulative Explained Variance
         log_response[y, x] = log[y, x] if 0 \le y \le y
                                                         Ratio')
gray.shape[0] and 0 \le x \le \text{gray.shape}[1] else 0
                                                           plt.grid(True)
       max_response =
np.max(log_response[coords])
                                                         plt.savefig(f'/content/Results/variance_ratio_{data
       if max response > best response:
                                                         set}_block{block_size}_k{k}_contrast{contrast_thr
         best_response = max_response
                                                         eshold\_edge\edge_threshold\.png')
         best_sigma = sigma
                                                           plt.close()
                                                           ninety percent idx = np.argmax(variance ratio
    # Compute SIFT descriptors with refined
                                                         >= 0.9) + 1
scale
                                                           print(f"90% variance at ~{ninety_percent_idx}
    sift =
                                                         components")
                                                           return ninety_percent_idx
cv2.SIFT create(contrastThreshold=contrast thre
shold, edgeThreshold=edge_threshold)
    for kp in keypoints:
                                                         # Apply PCA and return fitted PCA object
       kp.size = best_sigma * 2
                                                         def apply pca(descriptors, n components=59):
    _, des = sift.compute(gray, keypoints)
                                                           pca = PCA(n_components=n_components,
    if des is not None and len(des) > 0:
                                                         random state=42)
       descriptors.append(des)
                                                           transformed_descriptors =
  return np.vstack(descriptors) if descriptors and
                                                         pca.fit transform(descriptors)
descriptors[0].size else np.array([])
                                                           return transformed_descriptors, pca
# Compute PCA variance ratio with normalization
                                                         # Build BoW vocabulary
def compute_variance_ratio(images,
                                                         def build_vocabulary(descriptors, k,
block size=3, k=0.04, sigma values=[1.0, 2.0,
                                                         batch size=4096, use pca=False, pca dims=59):
4.0], contrast threshold=0.04,
                                                           if use pca:
edge_threshold=10, dataset='stl10'):
                                                              descriptors, pca = apply_pca(descriptors,
  descriptors =
                                                         n_components=pca_dims)
extract_harris_log_features(images, block_size,
                                                           else:
k, sigma_values, contrast_threshold,
                                                              pca = None
edge threshold)
                                                           kmeans = MiniBatchKMeans(n clusters=k,
  if len(descriptors) == 0:
                                                         n_init='auto', random_state=42,
    print("No descriptors found")
                                                         batch_size=batch_size)
    return 0
                                                           kmeans.fit(descriptors)
  scaler = StandardScaler()
                                                           return kmeans, pca
  descriptors = scaler.fit_transform(descriptors)
  pca = PCA(n_components=min(128,
                                                         # Compute spatial histograms with SPM (L2
descriptors.shape[0]-1), random_state=42)
                                                         normalization only)
  pca.fit(descriptors)
                                                         def compute_spatial_histograms(images,
  total var = np.sum(pca.explained variance )
                                                         kmeans, pca=None, spm levels=[1],
  if total var == 0:
                                                         block_size=3, k=0.04, sigma_values=[1.0, 2.0,
    print("Warning: Total variance is zero, check
                                                         4.0], contrast_threshold=0.04,
data or normalization")
                                                         edge threshold=10):
    return 0
                                                           histograms = []
  variance_ratio =
                                                           for img in images:
np.cumsum(pca.explained_variance_ratio_)
                                                              gray = cv2.cvtColor(img,
  plt.plot(range(1, len(variance_ratio) + 1),
                                                         cv2.COLOR_BGR2GRAY)
variance_ratio, 'b-')
                                                              # Harris corner detection for keypoints
  plt.title(f'Cumulative Explained Variance Ratio
                                                              harris = cv2.cornerHarris(gray,
({dataset.upper()})')
                                                         blockSize=block_size, ksize=3, k=k)
  plt.xlabel('Number of Components')
                                                              coords = np.where(harris > 0.01 *
                                                         harris.max())
```

```
keypoints = [cv2.KeyPoint(float(x), float(y),
                                                                  level_hist = np.zeros((grid_size *
1.0) for y, x in zip(coords[0], coords[1])]
                                                           grid_size, kmeans.n_clusters))
                                                                  for i in range(grid size):
     if not keypoints:
                                                                     for j in range(grid_size):
                                                                       mask = []
histograms.append(np.zeros(kmeans.n clusters *
                                                                       for y, x in zip(coords[0], coords[1]):
sum(lvl**2 for lvl in spm_levels)))
                                                                          if (i * grid_h <= y < (i + 1) * grid_h)
       continue
                                                           and (j * grid_w \le x < (j + 1) * grid_w):
                                                                             mask.append(True)
     # Scale selection with LoG
                                                                          else:
     best_sigma = None
                                                                             mask.append(False)
     best_response = -float('inf')
                                                                       mask = np.array(mask)
     for sigma in sigma values:
                                                                       grid words = words[mask] if
       log = cv2.GaussianBlur(gray, (0, 0),
                                                           len(words) > 0 else np.array([])
                                                                       if len(grid words) > 0:
sigma)
                                                                          hist, _ = np.histogram(grid_words,
       log_response = np.zeros_like(gray,
                                                           bins=range(kmeans.n_clusters + 1),
dtype=float)
       for y, x in zip(coords[0], coords[1]):
                                                           density=True)
          log_response[y, x] = log[y, x] if 0 <= y <
                                                                          level_hist[i * grid_size + j] = hist
gray.shape[0] and 0 \le x \le \text{gray.shape}[1] else 0
                                                                  img hist.append(level hist.flatten())
                                                                img_hist = np.concatenate(img_hist)
       max_response =
np.max(log_response[coords])
       if max_response > best_response:
                                                                # L2 normalization
          best_response = max_response
                                                                img_hist = img_hist /
                                                           (np.linalg.norm(img hist) + 1e-10)
          best_sigma = sigma
                                                                histograms.append(img_hist)
     # Compute SIFT descriptors for this image
     sift =
                                                             return np.array(histograms)
cv2.SIFT_create(contrastThreshold=contrast_thre
shold, edgeThreshold=edge_threshold)
                                                           # Save results to a file
     for kp in keypoints:
                                                           def save results(results,
       kp.size = best_sigma * 2
                                                           filename='/kaggle/working/experiment_results_ha
     _, des = sift.compute(gray, keypoints)
                                                           rris_sift_stl10.txt'):
                                                             with open(filename, 'a') as f:
     if des is None or len(des) == 0:
                                                                for result in results:
                                                                  f.write(result + '\n')
histograms.append(np.zeros(kmeans.n_clusters *
sum(lvl**2 for lvl in spm_levels)))
                                                           # Save classification report to a file
       continue
                                                           def save_classification_report(report,
                                                           filename='/kaggle/working/classification report h
     # Transform descriptors and predict words
                                                           arris sift stl10.txt'):
for this image
                                                             with open(filename, 'w') as f:
     if pca is not None:
                                                                f.write(report)
       des = pca.transform(des)
     words = kmeans.predict(des)
                                                          # Main pipeline
                                                           def main():
     # Compute spatial histogram
                                                             dataset = 'stl10'
     h, w = gray.shape
                                                             print(f"\nProcessing {dataset.upper()}")
     img\ hist = []
                                                             train_images, train_labels, test_images,
     for grid_size in spm_levels:
                                                           test_labels = load_data(dataset=dataset)
       grid_h, grid_w = h // grid_size, w //
                                                             block_sizes = [3, 4]
grid_size
```

k_values = [0.05, 0.07]	train_hist =
sigma_values_options = [[1.0, 2.0, 4.0]]	compute_spatial_histograms(
contrast_thresholds = [0.02, 0.04, 0.08]	train_images, kmeans,
edge_thresholds = [7.5, 10, 12.5]	pca, spm_levels=spm, block_size=block_size,
-	k=k,
best_accuracy = 0.0	
best_config = None	sigma_values=sigma_values,
best_predictions = None	contrast_threshold=contrast_threshold,
best_true_labels = None	_
	edge_threshold=edge_threshold
for block_size in block_sizes:)
for k in k_values:	test_hist =
for sigma_values in	compute_spatial_histograms(
sigma_values_options:	test_images, kmeans, pca,
for contrast_threshold in	spm levels=spm, block size=block size, k=k,
contrast_thresholds:	
for edge_threshold in	sigma_values=sigma_values,
edge_thresholds:	contrast_threshold=contrast_threshold,
<pre>print(f"\nBlock Size: {block_size},</pre>	
k: {k}, Sigma Values: {sigma_values}, Contrast	edge_threshold=edge_threshold
Threshold: {contrast_threshold}, Edge Threshold:)
{edge_threshold}")	,
pca_dims =	for C in [0.1, 1.0, 10.0]:
compute_variance_ratio(train_images,	svm = SVC(C=C,
block_size, k, sigma_values, contrast_threshold,	kernel='rbf', gamma='scale', random_state=42)
edge_threshold, dataset=dataset)	svm.fit(train_hist,
if pca_dims == 0:	train_labels)
continue	pred =
Continue	•
vocah sizos = [750, 1000, 1350]	svm.predict(test_hist)
vocab_sizes = [750, 1000, 1250]	acc =
spm_configs = [[1]]	accuracy_score(test_labels, pred) result = f"Dataset:
results = []	{dataset}, Block Size: {block_size}, k: {k}, Sigma:
for k_vocab in vocab_sizes:	{sigma_values}, Contrast: {contrast_threshold},
<pre>print(f"\nVocabulary size:</pre>	Edge: {edge_threshold}, Vocab Size: {k_vocab},
{k_vocab}")	SPM: {spm}, Norm: L2, SVM (Kernel=rbf, C={C},
train_descriptors =	Gamma=scale) Accuracy: {acc:.4f}"
extract_harris_log_features(train_images,	print(result)
block_size, k, sigma_values, contrast_threshold,	results.append(result)
edge_threshold)	
if len(train_descriptors) == 0:	if acc > best_accuracy:
print("No descriptors found")	best_accuracy = acc
continue	best_config = result
kmeans, pca =	best_predictions = pred
build_vocabulary(train_descriptors, k_vocab,	best_true_labels =
batch_size=4096, use_pca=True,	test_labels
pca_dims=pca_dims)	
	print(results)
for spm in spm_configs:	
print(f"\nSPM Levels: {spm},	if best_config is not None:
Normalization: L2")	<pre>print(f"\nBest Configuration: {best_config}")</pre>

```
print("\nClassification Report for Best
                                                               expected_train_size, expected_test_size =
Configuration:")
                                                          5000, 8000
     class names = ['airplane', 'automobile', 'bird',
                                                               expected per class = [500, 800]
'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
     report =
                                                             else:
classification report(best true labels,
                                                               raise ValueError("Dataset must be 'stl10'.")
best_predictions, target_names=class_names)
     print(report)
                                                             train images = train images.astype(np.uint8)
     save classification report(report)
                                                             test images = test images.astype(np.uint8)
if __name__ == "__main__":
                                                             print(f"Training set size: {len(train_images)}
  main()
                                                          images")
                                                             print(f"Test set size: {len(test images)}
import cv2
                                                          images")
import numpy as np
                                                             train class counts = np.bincount(train labels,
import matplotlib.pyplot as plt
                                                          minlength=10)
import os
                                                             test_class_counts = np.bincount(test_labels,
from sklearn.cluster import MiniBatchKMeans
                                                          minlength=10)
from sklearn.decomposition import PCA
                                                             print("Training class distribution:",
from sklearn.svm import SVC
                                                          train class counts)
from sklearn.metrics import accuracy_score,
                                                             print("Test class distribution:",
classification report
                                                          test_class_counts)
from sklearn.model selection import
train_test_split
                                                             if len(train_images) != expected_train_size or
                                                          len(test images) != expected test size:
from sklearn.preprocessing import
                                                               raise ValueError(f"Unexpected dataset size.
StandardScaler
import tensorflow_datasets as tfds
                                                          Expected {expected_train_size} train and
from sklearn.model selection import
                                                          {expected test size} test images.")
StratifiedShuffleSplit
                                                             if not all(count == expected_per_class[0] for
                                                          count in train_class_counts if count > 0) or not
# Load and sample STL-10 dataset with stratified
                                                          all(count == expected per class[1] for count in
                                                          test_class_counts if count > 0):
sampling
def load data(dataset='stl10'):
                                                               raise ValueError(f"Class distribution is not
  if dataset == 'stl10':
                                                          balanced. Expected {expected per class[0]} per
     ds, info = tfds.load('stl10', split=['train', 'test'],
                                                          class in train, {expected_per_class[1]} per class in
as supervised=True, with_info=True)
                                                          test.")
     train_ds, test_ds = ds[0], ds[1]
                                                             return train images, train labels, test images,
     train_images, train_labels = [], []
                                                          test_labels
     for image, label in tfds.as numpy(train ds):
       train_images.append(image)
                                                          # Extract SIFT features with scale-invariant
       train_labels.append(label)
                                                          detection
     train images = np.array(train images)
                                                          def extract sift features(images,
                                                          contrast_threshold=0.04, edge_threshold=10):
     train_labels = np.array(train_labels)
                                                             sift =
     test images, test labels = [], []
                                                          cv2.SIFT create(contrastThreshold=contrast thre
     for image, label in tfds.as_numpy(test_ds):
                                                          shold, edgeThreshold=edge_threshold)
       test images.append(image)
                                                             descriptors = []
       test labels.append(label)
                                                             for img in images:
     test images = np.array(test images)
                                                               gray = cv2.cvtColor(img,
     test_labels = np.array(test_labels)
                                                          cv2.COLOR_BGR2GRAY)
```

```
keypoints, des =
                                                           transformed_descriptors =
sift.detectAndCompute(gray, None)
                                                         pca.fit_transform(descriptors)
    if des is not None and len(keypoints) > 0:
                                                           return transformed descriptors, pca
       descriptors.append(des)
  return np.vstack(descriptors) if descriptors else
                                                         # Build BoW vocabulary
np.array([])
                                                         def build vocabulary(descriptors, k,
                                                         batch_size=4096, use_pca=False, pca_dims=59):
# Compute PCA variance ratio with normalization
                                                           if use pca:
def compute variance ratio(images,
                                                              descriptors, pca = apply pca(descriptors,
contrast threshold=0.04, edge threshold=10,
                                                         n components=pca dims)
dataset='stl10'):
                                                           else:
  descriptors = extract_sift_features(images,
                                                              pca = None
contrast threshold, edge threshold)
                                                           kmeans = MiniBatchKMeans(n clusters=k,
  if len(descriptors) == 0:
                                                         n_init='auto', random_state=42,
    print("No descriptors found")
                                                         batch size=batch size)
    return 0
                                                           kmeans.fit(descriptors)
  scaler = StandardScaler()
                                                           return kmeans, pca
  descriptors = scaler.fit transform(descriptors)
  pca = PCA(n components=min(128,
                                                         # Compute spatial histograms with SPM (L2
descriptors.shape[0]-1), random state=42)
                                                         normalization only)
  pca.fit(descriptors)
                                                         def compute_spatial_histograms(images,
  total_var = np.sum(pca.explained_variance )
                                                         kmeans, pca=None, spm_levels=[1],
  if total_var == 0:
                                                         contrast threshold=0.04, edge threshold=10):
    print("Warning: Total variance is zero, check
                                                           sift =
data or normalization")
                                                         cv2.SIFT create(contrastThreshold=contrast thre
                                                         shold, edgeThreshold=edge threshold)
    return 0
  variance_ratio =
                                                           histograms = []
np.cumsum(pca.explained variance ratio )
                                                           for img in images:
  plt.plot(range(1, len(variance_ratio) + 1),
                                                              gray = cv2.cvtColor(img,
variance ratio, 'b-')
                                                         cv2.COLOR_BGR2GRAY)
  plt.title(f'Cumulative Explained Variance Ratio
                                                              keypoints, des =
({dataset.upper()})')
                                                         sift.detectAndCompute(gray, None)
  plt.xlabel('Number of Components')
                                                              if des is None or len(keypoints) == 0:
  plt.ylabel('Cumulative Explained Variance
Ratio')
                                                         histograms.append(np.zeros(kmeans.n_clusters *
  plt.grid(True)
                                                         sum(lvl**2 for lvl in spm_levels)))
                                                                continue
plt.savefig(f'/content/variance ratio {dataset} con
                                                              if pca is not None:
trast{contrast_threshold}_edge{edge_threshold}.p
                                                                des = pca.transform(des)
ng')
                                                              words = kmeans.predict(des)
  plt.close()
  ninety_percent_idx = np.argmax(variance_ratio
                                                              h, w = gray.shape
>= 0.9) + 1
                                                              img hist = []
  print(f"90% variance at ~{ninety_percent_idx}
                                                              for grid_size in spm_levels:
components")
                                                                grid_h, grid_w = h // grid_size, w //
                                                         grid_size
  return ninety_percent_idx
                                                                level_hist = np.zeros((grid_size *
                                                         grid_size, kmeans.n_clusters))
# Apply PCA and return fitted PCA object
def apply_pca(descriptors, n_components=59):
                                                                for i in range(grid size):
  pca = PCA(n_components=n_components,
                                                                   for j in range(grid_size):
random_state=42)
                                                                     mask = []
```

```
best predictions = None
            for kp_idx, kp in
enumerate(keypoints):
                                                            best_true_labels = None
               kp x, kp y = kp.pt
               if (i * grid h <= kp y < (i + 1) *
                                                            for contrast threshold in contrast thresholds:
grid_h) and (j * grid_w \le kp_x < (j + 1) * grid_w):
                                                               for edge threshold in edge thresholds:
                                                                 print(f"\nContrast Threshold:
                 mask.append(True)
                                                          {contrast_threshold}, Edge Threshold:
                                                          {edge threshold}")
                 mask.append(False)
            mask = np.array(mask)
                                                                 pca dims =
            grid words = words[mask]
                                                          compute variance ratio(train images,
            if len(grid_words) > 0:
                                                          contrast_threshold, edge_threshold,
               hist, _ = np.histogram(grid_words,
                                                          dataset=dataset)
bins=range(kmeans.n clusters + 1),
                                                                 if pca dims == 0:
density=True)
                                                                    continue
               level hist[i * grid size + j] = hist
       img hist.append(level hist.flatten())
                                                                 vocab_sizes = [750, 1000, 1250]
    img_hist = np.concatenate(img_hist)
                                                                 spm_configs = [[1]]
    # L2 normalization
                                                                 results = []
    img hist = img hist /
                                                                 for k in vocab sizes:
                                                                    print(f"\nVocabulary size: {k}")
(np.linalg.norm(img_hist) + 1e-10)
    histograms.append(img_hist)
                                                                    train_descriptors =
                                                          extract_sift_features(train_images,
                                                          contrast_threshold, edge_threshold)
  histograms = np.array(histograms)
  return histograms
                                                                    if len(train descriptors) == 0:
                                                                      print("No descriptors found")
# Save results to a file
                                                                      continue
def save results(results,
                                                                    kmeans, pca =
filename='/content/experiment_results_sift_stl10.t
                                                          build_vocabulary(train_descriptors, k,
                                                          batch_size=4096, use_pca=True,
xt'):
  with open(filename, 'a') as f:
                                                          pca_dims=pca_dims)
    for result in results:
       f.write(result + '\n')
                                                                    for spm in spm_configs:
                                                                      print(f"\nSPM Levels: {spm},
# Save classification report to a file
                                                          Normalization: L2")
def save_classification_report(report,
                                                                      train_hist =
filename='/content/classification report stl10.txt'):
                                                          compute_spatial_histograms(
  with open(filename, 'w') as f:
                                                                         train_images, kmeans, pca,
    f.write(report)
                                                          spm_levels=spm,
                                                          contrast threshold=contrast threshold,
# Main pipeline
                                                                         edge_threshold=edge_threshold
def main():
                                                                      )
  dataset = 'stl10'
                                                                      test hist =
                                                          compute_spatial_histograms(
  print(f"\nProcessing {dataset.upper()}")
  train_images, train_labels, test_images,
                                                                         test_images, kmeans, pca,
test_labels = load_data(dataset=dataset)
                                                          spm levels=spm,
                                                          contrast_threshold=contrast_threshold,
  contrast thresholds = [0.02, 0.04, 0.08]
                                                                         edge_threshold=edge_threshold
  edge_thresholds = [7.5, 10, 12.5]
                                                                      )
  best_accuracy = 0.0
                                                                      # SVM with rbf kernel only
  best_config = None
                                                                      for C in [0.1, 1.0, 10.0]:
```

```
svm = SVC(C=C, kernel='rbf',
gamma='scale', random_state=42)
               svm.fit(train hist, train labels)
               pred = svm.predict(test hist)
               acc = accuracy_score(test_labels,
pred)
               result = f"Dataset: {dataset},
Contrast: {contrast_threshold}, Edge:
{edge threshold}, Vocab Size: {k}, SPM: {spm},
Norm: L2, SVM (Kernel=rbf, C={C},
Gamma=scale) Accuracy: {acc:.4f}"
               print(result)
               results.append(result)
               # Track best configuration
               if acc > best accuracy:
                  best_accuracy = acc
                  best config = result
                  best predictions = pred
                  best true labels = test labels
       save_results(results)
  # Generate and save classification report for
the best configuration
  if best config is not None:
     print(f"\nBest Configuration: {best_config}")
     print("\nClassification Report for Best
Configuration:")
     class_names = ['airplane', 'automobile', 'bird',
'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
     report =
classification_report(best_true_labels,
best_predictions, target_names=class_names)
     print(report)
     save_classification_report(report)
if __name__ == "__main__":
  main()
```