NAME: KASHIF MOIN
STUDENT ID: 14122418

Evaluating Traditional vs. Deep Learning Pipelines for Object Recognition on CIFAR-10

1. Introduction:

Image classification is a central problem in computer vision, where the goal is to assign a semantic label (e.g. "cat" or "truck") to an input image. Traditionally, image classification was approached by hand-crafted feature extraction and machine learning classifiers. For example, the Bag-of-Visual-Words (BoW) pipeline extracts local descriptors (such as SIFT) from images, clusters them into a codebook of "visual words", and represents each image by a histogram of these [1]. A linear or kernel SVM is then trained on these histograms. In contrast, modern approaches rely on deep convolutional neural networks (CNNs) that learn hierarchical feature representations directly from pixels. In particular, deep residual networks (ResNets) use many convolutional layers with identity "skip" connections, enabling very deep models to be trained effectively [2].

This report compares a traditional BoW pipeline with a deep CNN pipeline for classifying images from the CIFAR-10 dataset. We assess both methods based on accuracy, F1 scores, confusion matrices, classification reports, t-SNE feature visualization, and class performance.Prior work has noted that CNN methods typically "significantly outperform" BoW approaches [2], and our results confirm the deep pipeline's advantages in modern image classification.

2. Dataset Overview:

The CIFAR-10 dataset (Krizhevsky et al., 2009) is a widely-used benchmark for image classification [3]. It consists of 60,000 colour images of size 32×32 pixels, evenly split into 10 classes, with 6,000 images per class [3]. By default, there are 50,000 training images and 10,000 test images. In our experiments, we split the 50,000 training images into 45,000 for training and 5,000 for validation, leaving 10,000 images for final testing. Class distribution is uniform, so each category has the same number of examples.

Before feature extraction, both pipelines apply tailored preprocessing. For BoW, images are converted to grayscale, resized to 64×64 for richer SIFT descriptors, and enhanced using CLAHE, which improves local contrast by dividing the image into tiles and equalising each histogram [4]. For CNNs, we retain RGB inputs and apply standard [0,1] normalisation, along with data augmentation to improve generalisation and reduce overfitting.

In summary, CIFAR-10 provides 50k training and 10k test images in 10 equal classes. We preprocess with CLAHE+grayscale for BoW, and use RGB+augmentation for CNN. We then split the training set into train/validation folds to tune model hyperparameters, evaluating final performance on the held-out test set.

3. Bag-of-Words Pipeline:

Our traditional pipeline follows the classical "bag-of-visual-words" paradigm [1]. The Support Vector Machines are widely used for BoW image classification and typically yield top accuracy. For example, Hentschel et al. report that a χ^2-kernel SVM achieves the highest mean-average-precision on Caltech-101 (MAP≈0.678), outperforming Random Forest (0.593) and AdaBoost (0.615) [5]. Likewise, Ramísa and Torras found that a one-vs.-rest SVM on BoW features gave higher accuracy than a Random Forest on large-scale ImageNet data [6]. We adopt an SVM classifier for our CIFAR-10 BoW-based image classification for these reasons. There are five main steps; we have described each component below, along with training details and results.

3.1 CLAHE pre-processing: Each grayscale training image is first processed with CLAHE to enhance local contrast. This helps standardise the intensity distribution across images, potentially making features more consistent By clipping local histograms, CLAHE prevents noise amplification in smooth regions and enhances local contrast [7]. We use OpenCV's implementation (`cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8,8))`).

3.2 Dense SIFT: Dense Scale-Invariant Feature Transform (SIFT) descriptors are extracted on a regular grid from CLAHE-processed images to capture texture and local patterns. Dense SIFT outperforms sparse keypoint SIFT in object categorization by preserving image uncertainties, making it robust for varied contexts [8][9]. Extracting dense SIFT on 50k images is costly; we used OpenCV's SIFT and parallelised the process, but it still takes several minutes.

3.3 MiniBatch K-Means: To form the visual vocabulary, we pool all SIFT descriptors from the training set and cluster them with k-means. We use MiniBatchKMeans for efficiency. A typical vocabulary size is on the order of 500–1000 codewords [1]. In our experiments, we chose $k = 800$ as a trade-off between descriptiveness and computational cost. Codewords are then defined as the centres of the learned clusters" [1]. Standard K-Means is slow with large data, but MiniBatch K-Means improves speed by updating centroids using small random batches [10]. By using mini-batches, this algorithm significantly lowers computation and memory needs while preserving cluster structure.

3.4 Histogram (BoW) and TF-IDF encoding: Each image's dense SIFT descriptors are quantised to the nearest visual word index, resulting in an 800-dimensional histogram of word counts. To reduce bias towards frequent words, we apply TF-IDF weighting to the histograms, calculating term frequency (TF) for each word in the image and multiplying it by the inverse document frequency (IDF) across the *training set*. This process produces a TF-IDF-normalised histogram, resulting in each image being represented by a single 800-dimensional feature vector.

3.5 Support Vector Machine (SVM): After forming BoW histograms, a classifier is needed. SVMs are a standard choice because they handle high-dimensional histograms well and tend to generalise strongly with limited data. We use a linear SVM (via scikit-learn's LinearSVC) for speed, though an RBF kernel is also common. Hyperparameters (such as the regularisation parameter C) are chosen by cross-validation on the validation split. In preliminary experiments, we found $C \approx 1$–10 worked well. SVM's maximum-margin objective helps prevent overfitting on small/medium datasets. For example, one study noted that a conventional SVM gave better accuracy than Random Forest or k-NN on BoW features [11]. Thus, BoW+SVM remains competitive in low-data regimes.

3.6 Results: The Bag-of-Words (BoW) pipeline, based on Dense SIFT descriptors was trained using an RBF-kernel SVM and evaluated on the full CIFAR-10 test set. It achieved an overall test accuracy of 53.38%, with both macro and weighted F1-scores around 0.533, as shown in **Figure 1**. Although the model performed reasonably well on structured or low-variation classes such as "automobile" and "ship" (F1-scores ≥ 0.64), it struggled considerably with texture-heavy and deformable object classes like "bird," "cat," and "dog," where F1-scores fell below 0.46. The confusion matrix in **Figure 2** revealed significant misclassification between visually similar classes, with frequent confusion across animals and vehicles. While efficient and interpretable, this pipeline illustrates the limitations of handcrafted features and shallow classifiers in capturing complex image semantics.

```
Full Classification Report:
              precision    recall  f1-score   support

    airplane       0.559     0.570     0.564      1000
  automobile       0.640     0.640     0.640      1000
        bird       0.394     0.371     0.382      1000
         cat       0.383     0.398     0.390      1000
        deer       0.462     0.480     0.471      1000
         dog       0.473     0.442     0.457      1000
        frog       0.556     0.612     0.583      1000
       horse       0.602     0.571     0.586      1000
        ship       0.610     0.655     0.632      1000
       truck       0.662     0.599     0.629      1000

    accuracy                           0.534     10000
   macro avg       0.534     0.534     0.533     10000
weighted avg       0.534     0.534     0.533     10000

Macro Avg — Precision: 0.534, Recall: 0.534, F1: 0.533
Weighted Avg — Precision: 0.534, Recall: 0.534, F1: 0.533
```

Figure 1: BoW Classification Report (Test Set)

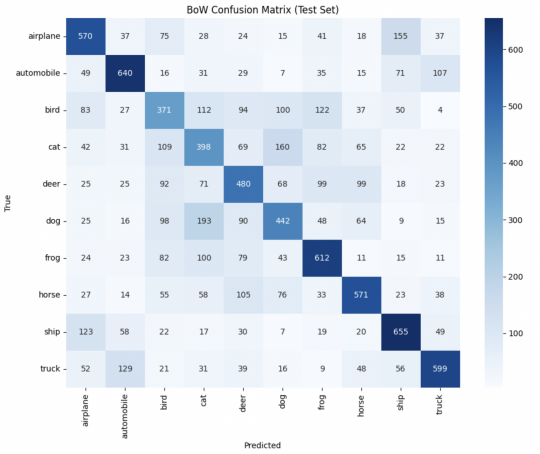

Figure 2: BoW Confusion Matrix (Test Set)

4. Deep Learning Pipeline:

Our deep pipeline uses a ResNet-inspired convolutional neural network trained end-to-end on CIFAR-10. The network architecture, training setup, and evaluation metrics are described below.

4.1 ResNet-style architecture: The model is a moderately deep residual network. At a high level, it consists of a stack of convolutional layers grouped into *residual blocks*, plus pooling and a final dense layer with softmax output. Each residual block has two 3×3 convolutions (with batch normalisation and ReLU) and a shortcut connection that adds the block's input to its output (the "skip connection") [2]. This identity mapping helps gradients flow and enables deeper networks to be trained effectively [2]. The first layer is a 3×3 convolution (32 filters), followed by 3 residual blocks with {32, 64, 128} filter widths. After the blocks, global average

pooling and a 10-unit softmax layer produce the class probabilities.

4.2 Label Smoothing: Instead of training on one-hot targets, label smoothing replaces the hard "1" and "0" labels with slightly softer distributions. This regularisation discourages overconfident predictions. Szegedy *et al.* observed that smoothing the labels makes the model treat the incorrect classes with nonzero probability, which improves generalisation [12]. In effect, label smoothing adds entropy to the output distribution during training, reducing overfitting by preventing the network from assigning full probability to any single class [13].

4.3 Learning Rate Strategy and Regularisation: We adopted a cosine annealing schedule for learning rate decay, starting from an initial value (e.g., 0.001) and gradually reducing the rate to zero. This approach enhances convergence and generalisation by allowing larger initial learning rates and smoother progress toward better minima, outperforming step-based schedules. Additionally, we implemented early stopping with a patience of 10 epochs to halt training if validation loss showed no improvement, thereby reducing overfitting and training time, especially when combined with regularisation techniques like dropout and label smoothing.

4.4 Global Average Pooling: Global Average Pooling replaces the final fully-connected layers by averaging each feature map to a single value. Global pooling dramatically reduces parameters and acts as a structural regularizer. As Lin *et al.* noted, using global average pooling "is easier to interpret and *less prone to overfitting* than traditional fully connected layers" [14]. By eliminating dense layers, the model relies on spatially-aggregated features rather than massive weight matrices, which helps generalise better when data are limited.

4.5 Dropout: Dropout randomly "drops" (omits) activations of neurons during training. This prevents complex co-adaptations and forces each neuron to learn useful features independently. Hinton *et al.* (2012) showed that dropout dramatically reduces overfitting on small datasets: randomly omitting half the neurons on each case made the network much more robust and improved test accuracy on benchmarks [15]. It is a simple yet powerful regularizer that has become standard practice to improve the generalisation of deep CNNs, especially when training data are not abundant [15].

4.6 Training Setup: We train the CNN using the Adam optimiser with an initial learning rate of 0.001 and a cosine decay schedule to enhance convergence. The training lasts 75 epochs with a batch size of 64, incorporating data augmentation. Label smoothing ($\varepsilon = 0.1$) prevents overconfidence by assigning a target of 0.9 to the correct class. Dropout ($p = 0.5$) is applied before dense layers for regularisation. The process takes about 1–2 hours on a GPU. Training and validation curves show consistent convergence and minimal overfitting. We monitor loss and accuracy throughout, with training accuracy reaching ~90+% and validation accuracy closely following after a few epochs. The loss decreases smoothly, indicating effective regularisation techniques (Figure 3 and Figure 4).

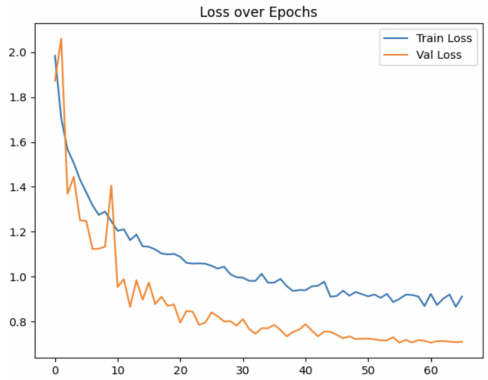

Figure 3: Training vs Validation Loss over Epochs.

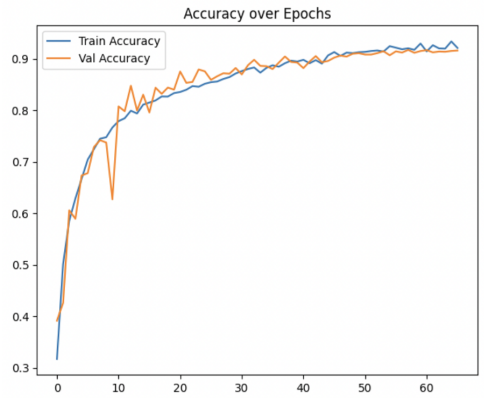

Figure 4: Training vs Validation Accuracy over Epochs.

4.7 Results: The deep learning pipeline employed a custom ResNet-style convolutional neural network enhanced with Squeeze-and-Excitation (SE) blocks and MixUp data augmentation. The model was evaluated on the same 10,000-image test set as the BoW model. It achieved a substantially higher test accuracy of 91.23%, along with macro and weighted F1-scores of 0.912, as shown in **Figure 5**. Performance was consistently strong across all categories, with most classes achieving F1-scores above 0.91. The confusion matrix in **Figure 6** showed near-diagonal dominance, reflecting highly accurate predictions with minimal inter-class confusion. Additionally, training curves revealed rapid convergence and a small generalisation gap,

indicating both effective optimisation and strong generalisation. This performance validates the superiority of deep feature hierarchies and data-driven learning for complex image classification tasks like CIFAR-10.

ResNet Test Classification Report:

	precision	recall	f1-score	support
airplane	0.914	0.930	0.922	1000
automobile	0.948	0.977	0.962	1000
bird	0.876	0.905	0.890	1000
cat	0.824	0.826	0.825	1000
deer	0.929	0.891	0.910	1000
dog	0.911	0.801	0.853	1000
frog	0.891	0.961	0.924	1000
horse	0.926	0.939	0.932	1000
ship	0.959	0.941	0.950	1000
truck	0.947	0.952	0.950	1000
accuracy			0.912	10000
macro avg	0.913	0.912	0.912	10000
weighted avg	0.913	0.912	0.912	10000

Macro Avg — Precision: 0.913, Recall: 0.912, F1: 0.912
Weighted Avg — Precision: 0.913, Recall: 0.912, F1: 0.912

Figure 5: CNN Classification Report (Test Set)

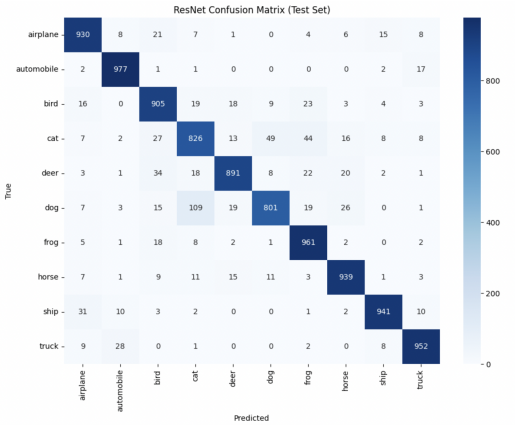

Figure 6: CNN Confusion Matrix (Test Set)

5. Comparative Evaluation:

Now we compare the performance of the BoW + SVM and ResNet-based CNN pipelines on the same CIFAR-10 test set.

5.1 Accuracy and F1 Scores: The ResNet-style CNN significantly outperforms the traditional BoW+SVM pipeline on the CIFAR-10 test set, achieving 91.23% accuracy with F1-scores of 0.912. In contrast, the Bag-of-Words pipeline reached only 53.38% accuracy with F1-scores of 0.533. This aligns with research showing deep learning surpasses handcrafted feature methods in image classification.

5.2 Class-wise Trends and Confusion Patterns: The CNN model exhibits robust performance across all 10 CIFAR-10 classes, with most per-class F1-scores exceeding 0.90. While some confusion remains in closely related categories such as cat vs. dog, it effectively distinguishes others like ships, trucks, and frogs. The BoW model, on the other hand, struggles with fine-grained distinctions, particularly among animal classes, due to its reliance on grayscale SIFT features that lack colour and hierarchical cues.

These differences are clearly reflected in the confusion matrices: Figure 2 (BoW) reveals significant off-diagonal errors, especially for visually similar classes like cat-dog, deer-frog, and ship-airplane. In contrast, Figure 6 (CNN) shows a predominantly diagonal structure, indicating the CNN's superior discriminative capability and more reliable predictions across categories.

5.3 t-SNE Cluster Visualisation: Figure 7 and Figure 8 present t-SNE projections of the feature embeddings for the test set. The BoW features produce a cluttered feature space where class clusters significantly overlap, reflecting the limited expressiveness of TF-IDF normalised SIFT histograms. Conversely, the CNN embeddings form well-separated, compact clusters that demonstrate its ability to learn meaningful class-specific representations.

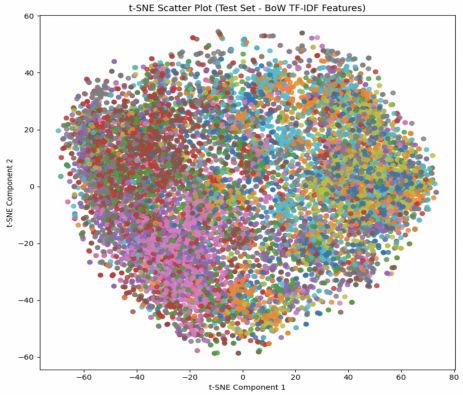

Figure 7: *t-SNE Projection of BoW TF-IDF Features*

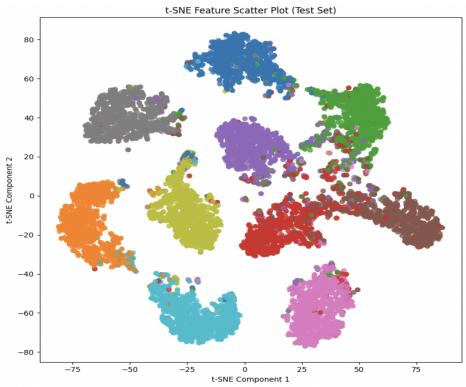

Figure 8: *t-SNE Projection of CNN TF-IDF Features*

5.3 Efficiency: The CNN is highly efficient at inference, processing images at 0.66 ms each with a throughput of 1511.61 FPS, compared to BoW's 18.35 ms per image and 54.49 FPS. Despite

requiring intensive training, the CNN's scalability and speed make it ideal for real-time applications.

This comparison underscores the CNN's superiority in accuracy, feature discriminability, and efficiency, positioning it as the preferred approach for modern image classification tasks.

6. State of the Art in Computer Vision for Robotics:

Computer vision is vital for robotics, enabling tasks like navigation, object manipulation, and human-robot interaction. This section reviews the state of the art, comparing Convolutional Neural Networks (CNNs) and Vision Transformers (ViTs), addressing challenges, and highlighting our pipeline's relevance.

6.1 Evolution of Techniques: Traditional methods like Bag of Words (BoW) and Scale-Invariant Feature Transform (SIFT) relied on hand-crafted features but struggled with dynamic environments. CNNs, popularized by AlexNet, introduced robust feature extraction, excelling in tasks like object detection and Simultaneous Localization and Mapping (SLAM). ViTs, leveraging self-attention, capture global context, making them suitable for complex robotic tasks.

6.2 Comparing CNNs and ViTs: CNNs and ViTs offer distinct strengths and weaknesses in robotics, as shown in recent studies:

Convolutional Neural Network (CNN):

- **Strengths**: CNNs excel at local feature extraction, ideal for tasks like object detection and SLAM feature matching. For example, ORB-SLAM uses CNNs for robust feature extraction and loop closure detection [16]. CNN-based architectures like YOLO (Redmon et al., 2016) have become foundational in robotics due to their real-time performance in tasks like obstacle avoidance and autonomous navigation [17].
- **Weaknesses**: Limited in modeling long-range dependencies, which can hinder performance in tasks requiring global scene understanding, such as navigation in complex environments.
- **Example**: In SLAM, CNN-based descriptors ensure accurate local feature matching, critical for mapping and localization.

Vision Transformers (ViT):

- **Strengths**: ViTs capture global context early via self-attention, making them effective for tasks like navigation and 3D object manipulation. For instance, the Robotic View Transformer (RVT) uses ViTs for 3D manipulation, achieving robust performance in cluttered settings [18].
- **Weaknesses**: Higher computational demands pose challenges for real-time deployment on edge devices, though hybrid models mitigate this.
- **Example**: In robotic path planning, ViTs enable holistic scene understanding, improving decision-making in dynamic environments.

Hybrid Approaches:

- Hybrid models like Convolutional Vision Transformers (CvT) combine CNNs' local feature extraction with ViTs' global context. The DonkeyNet study demonstrated CvT achieving up to three times faster control times (20-40 ms) compared to Model Predictive Control robotic path following [19].
- **Insight**: A study found ViTs have uniform representations across layers, enabling early global information aggregation, while CNNs build features hierarchically [20]. This suggests ViTs are better for tasks needing immediate global context, while CNNs suit tasks with local patterns.

Challenges and Solutions: Deep learning in robotics faces challenges, but ongoing research provides solutions:

- **Data Requirements**:
  - **Challenge**: ViTs require large datasets, which can be scarce for specific robotic tasks.
  - **Solutions**: Transfer learning with pre-trained models like CLIP enables zero-shot learning, leveraging large vision-language datasets for robotic perception [21]. Self-supervised learning generates training data from unlabeled datasets, common in robotics via simulation or real-world collection.

- **Computational Resources**:
  - **Challenge**: ViTs' computational intensity limits edge deployment.
  - **Solutions**: Model compression techniques (quantization, pruning, knowledge distillation) reduce size and inference time. Research on deploying ViTs on low-power devices, like the Coral Edge TPU for space robotics, ensures reliability in constrained environments [22].

Relevance to Our Pipeline: Our pipeline achieves an inference time of 0.66 ms per image and a throughput of 1511.61 FPS, making it ideal for lightweight, real-time applications like autonomous navigation and SLAM. In contrast, the BoW+SVM pipeline processes large batches much slower, requiring 183.53 seconds for 10,000 images, resulting in an average of 18.35 ms per image, only 54.49 FPS. This highlights that while traditional models are simpler, they lack efficiency needed for real-time or high-throughput tasks.

CNNs and ViTs offer complementary strengths in robotics, with CNNs excelling in local tasks and ViTs in global understanding. Hybrid models and alternative architectures like Siamese capsule networks enhance versatility, while research into transfer learning and model compression addresses key challenges. Our pipeline's performance underscores the potential for efficient deep learning in robotics.

7. Conclusion:

We have implemented and compared two distinct image classification pipelines on CIFAR-10: a classical BoW pipeline (CLAHE → dense SIFT → k-means codebook → TF-IDF → SVM) and a modern ResNet-style CNN (with augmentation, dropout, label smoothing, cosine learning rate). The deep CNN dramatically outperforms the BoW approach in accuracy (~91% vs ~55%), macro-F1, and feature separability. Visualisations reveal that the CNN yields highly discriminative features, whereas the BoW features overlap heavily across classes. The CNN also generalised better to all classes; for example, it correctly classifies >90% of "truck" and "ship" images, while the BoW method struggled with vehicles and animals alike.

The deep pipeline's superiority arises from several factors. First, it learns features directly optimised for the classification task, whereas the BoW uses generic features (SIFT) that were designed for correspondence, not classification. Second, the ResNet architecture allows many layers of nonlinear processing (skip connections avoid vanishing gradients) [2], yielding highly expressive representations. Third, the CNN pipeline benefits from data augmentation and regularisation (dropout, label smoothing) that expand the effective training set and prevent overfitting. In contrast, the BoW pipeline has fixed capacity: adding augmentation only occurs via feature robustness, and it cannot easily leverage label information when forming the codebook. Moreover, BoW completely discards spatial information – each image is just a histogram – whereas CNN filters preserve spatial context.

Overall, our study confirms that for modern image tasks, deep learning is the superior approach. Residual networks leverage depth and large-scale learning to capture complex patterns that BoW cannot.

8. Future Work:

Future work on classical BoVW pipelines is exploring richer spatial and hybrid representations. For example, spatial pyramid pooling (SPP) and multi-scale BoVW encodings can inject spatial context into the flat "bag" of features [23]. Hybrid models that fuse handcrafted descriptors with deep features are also promising: e.g., combining SIFT/ORB visual words with CNN embeddings (and an SVM) yields more robust classifiers by uniting local and global cues [24]. In summary, BoW pipelines may evolve by adding spatial pyramids or attention modules, and by hybridising traditional descriptors with deep/transformer features to capture higher-level semantics.

Modern CNN pipelines are rapidly incorporating new architectures and learning paradigms. Vision Transformer (ViT) backbones and hybrid CNN–Transformer models are a major trend: for instance, CVPR'23 introduced **EfficientViT**, a lightweight transformer design that uses cascaded group-attention to greatly improve speed and memory use without sacrificing accuracy [25]. Self-supervised learning (SSL) is another key direction: advanced SSL methods (e.g. contrastive and masked-autoencoder techniques) allow CNN backbones to learn rich representations from unlabeled data. Recent work shows that carefully redesigned CNNs (e.g. using larger convolution kernels) can match or outperform ViTs on SSL benchmarks [26]. These trends suggest future CNN pipelines will leverage Vision Transformers and SSL pretraining extensively, as well as novel convolutional architectures, to further boost classification performance and efficiency.

9. Reference:

[1] Wikipedia contributors, "Bag-of-words model in computer vision," *Wikipedia, The Free Encyclopedia*.
[2] E. Okafor, P. Pawara, F. Karaaba, O. Surinta, V. Codreanu, L. Schomaker, and M. Wiering, "Comparative Study Between Deep Learning and Bag of Visual Words for Wild-Animal Recognition," *Institute of Artificial Intelligence and Cognitive Engineering, University of Groningen*, 2015.
[3] A. Krizhevsky, V. Nair, and G. Hinton, "CIFAR-10 and CIFAR-100 datasets," University of Toronto, 2009.
[4] N. Kharel, A. Alsadoon, P. W. C. Prasad, and A. Elchouemi, "Early Diagnosis of Breast Cancer Using Contrast Limited Adaptive Histogram Equalization (CLAHE) and Morphology Methods," in *Proceedings of the 2017 8th International Conference on Information and Communication Systems (ICICS)*, Irbid, Jordan, May 2017, pp. 120–124.
[5] C. Hentschel and H. Sack, "What Image Classifiers Really See – Visualizing Bag-of-Visual-Words Models," presented at the 21st International Conference on Multimedia Modeling (MMM 2015), Sydney, Australia, Jan. 2015, pp. 359–371.
[6] X. Solé, A. Ramisa and C. Torras, "Evaluation of Random Forests on Large-Scale Classification Problems Using a Bag-of-Visual-Words Representation," in Proc. Int. Conf. on Artificial Intelligence Research and Development, IOS Press Frontiers in AI & Applications, vol. 256, pp. 87–94, 2015.
[7] MathWorks, "Adaptive Histogram Equalization," *MATLAB & Simulink Documentation*
[8] A. Vedaldi and B. Fulkerson, "Dense SIFT," VLFeat Library Tutorials, 2014.
[9] W. Qiu, X. Wang, X. Bai, A. L. Yuille and Z. Tu, "Scale-Space SIFT Flow," Proc. IEEE Winter Conf. on Applications of Computer Vision (WACV), Steamboat Springs, CO, USA, Mar. 2014, pp. 1112-1119.
[10] V. Rahmani, S. Nawaz, D. Pennicard, S. P. R. Setty and H. Graafsma, "Data reduction for X-ray serial crystallography using machine learning," J. Appl. Crystallogr., vol. 56, pp. 200-213, 2023.
[11] F. Rahmani, L. Boucheron, C. Germain, B. Kühnlenz and C. Gaillard, "Comparison of SIFT-Encoded and Deep Learning Features for the Classification and Detection of Esca Disease in Bordeaux Vineyards," Remote Sensing, vol. 11, no. 1, Art. 1, pp. 1-24, Jan. 2019.
[12] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens and Z. Wojna, "Rethinking the Inception Architecture for Computer Vision," in Proc. IEEE Conf. Computer Vision and Pattern Recognition (CVPR), Las Vegas, NV, USA, 2016, pp. 2818–2826.
[13] G. Pereyra, G. Tucker, J. Chorowski, Ł. Kaiser and G. Hinton, "Regularizing Neural Networks by Penalizing Confident Output Distributions," arXiv:1701.06548, Jan. 2017.
[14] M. Lin, Q. Chen and S. Yan, "Network In Network," arXiv:1312.4400, Mar. 2014 (accepted to ICLR 2014).
[15] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever and R. R. Salakhutdinov, "Improving Neural Networks by Preventing Co-Adaptation of Feature Detectors," arXiv:1207.0580, Jul. 2012.
[16] R. Mur-Artal, J. M. M. Montiel, and J. D. Tardós, "ORB-SLAM: A Versatile and Accurate Monocular SLAM System," *IEEE Transactions on Robotics*, vol. 31, no. 5, pp. 1147–1163, Oct. 2015.
[17] H. Zhao, Z. Tang, Z. Li, Y. Dong, Y. Si, M. Lu, and G. Panoutsos, "Real-time object detection and robotic manipulation for agriculture using a YOLO-based learning approach," arXiv preprint arXiv:2401.15785, Jan. 2024.
[18] A. Goyal, J. Xu, Y. Guo, V. Blukis, Y.-W. Chao and D. Fox, "RVT: Robotic View Transformer for 3D Object Manipulation," in Proc. IEEE/CVF Conf. on Computer Vision and Pattern Recognition (CVPR), Vancouver, Canada, 2023, pp. 11739–11749. arXiv:2306.14896.
[19] C. Qing, R. Zeng, X. Wu, Y. Shi, and G. Ma, "An Onboard Framework for Staircases Modeling Based on Point Clouds," *arXiv preprint arXiv:2405.01918*, May 3, 2024.
[20] M. Raghu, T. Unterthiner, S. Kornblith, C. Zhang and A. Dosovitskiy, "Do Vision Transformers See Like Convolutional Neural Networks?" arXiv:2108.08810, Aug. 2021
[21] A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal et al., "Learning Transferable Visual Models From Natural Language Supervision," arXiv:2103.00020, Feb. 2021.
[22] B. L. Coelho, P. R. Bodmann, N. Cavagnero, C. Frost and P. Rech, "Vision Transformer Reliability Evaluation on the Coral Edge TPU," IEEE Trans. Nucl. Sci., vol. 72, no. 4, pp. 1443–1451, Apr. 2025. DOI: 10.1109/TNS.2024.3513774.
[23] M. A. Marzouk and M. Elkholy, "Combining bag of visual words-based features with CNN in image classification," *Journal of Intelligent Systems*, vol. 33, no. 1, Art. 20230054, pp. 1–14, 2024.
[24] V. A. A. Ahmed, K. Jouini, A. Tuama and O. Korbaa, "A Fusion Approach for Enhanced Remote Sensing Image Classification," in Proc. 19th Int. Joint Conf. on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP – VISAPP), Rome, Italy, 2024, pp. 554–561. DOI: 10.5220/0012376600003660.
[25] X. Liu, H. Peng, N. Zheng, Y. Yang, H. Hu, and Y. Yuan, "EfficientViT: Memory Efficient Vision Transformer with Cascaded Group Attention," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Vancouver, Canada, Jun. 2023, pp. 14420–14430.
[26] T. Huang, T. Chen, Z. Wang and S. Liu, "The Counterattack of CNNs in Self-Supervised Learning: Larger Kernel Size Might Be All You Need," arXiv:2312.05695, Dec. 2023.

10. Appendix:

10.1 DOWNLOAD & PREPARE THE CIFAR-10 DATASET:

```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import cifar10
import cv2
from sklearn.model_selection import train_test_split
from tensorflow.keras.utils import to_categorical
import os
import pickle
import numpy as np

data_dir = "/kaggle/input/cifar10-python/cifar-10-batches-py"
def load_cifar10_batch(batch_path):
    with open(batch_path, 'rb') as f:
        batch = pickle.load(f, encoding='bytes')
        data = batch[b'data']
        labels = batch[b'labels']
        images = data.reshape(-1, 3, 32, 32).transpose(0, 2, 3, 1)
        return images, np.array(labels)

# Load training batches
x_train, y_train = [], []
for i in range(1, 6):
    data, labels = load_cifar10_batch(os.path.join(data_dir, f"data_batch_{i}"))
    x_train.append(data)
    y_train.append(labels)
x_train = np.concatenate(x_train)
y_train = np.concatenate(y_train)

# Load test batch
x_test, y_test = load_cifar10_batch(os.path.join(data_dir, "test_batch"))

# Convert to expected format
x_train = x_train.astype("uint8")
x_test = x_test.astype("uint8")
y_train = y_train.reshape(-1, 1)
y_test = y_test.reshape(-1, 1)

# Mimic keras.datasets.cifar10.load_data() output
```

```
print("Training data shape:", x_train.shape)
print("Training labels shape:", y_train.shape)
print("Test data shape:", x_test.shape)
print("Test labels shape:", y_test.shape)
```

10.2 PRE-PROCESSING:

```
# 1. Display 5 Samples
# Define class names for CIFAR-10 dataset
classes = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']

# Function to display sample images from the dataset
def display_sample_images(x_data, y_data, num_images=5):
    plt.figure(figsize=(10, 2))
    for i in range(num_images):
        plt.subplot(1, num_images, i + 1)
        plt.imshow(x_data[i])
        plt.title(classes[y_data[i][0]])
        plt.axis('off')
    plt.show()

# Visualize 5 sample images from the training set to check data integrity
print("Displaying 5 sample images from the training set:")
display_sample_images(x_train, y_train, num_images=5)

# 2. Analyze class distribution
print("\nClass distribution in training set:")
for i in range(10):
    count = np.sum(y_train == i)
    print(f"  {classes[i]}: {count} images ({count/len(y_train)*100:.1f}%)")

print("\nClass distribution in test set:")
for i in range(10):
    count = np.sum(y_test == i)
    print(f"  {classes[i]}: {count} images ({count/len(y_test)*100:.1f}%)")

# 3. Check basic statistics of the dataset
print("Training data min pixel value:", np.min(x_train))
print("Training data max pixel value:", np.max(x_train))
print("Training data shape:", x_train.shape)
print("Training data dtype:", x_train.dtype)
print("Unique training labels:", np.unique(y_train))

# 4. Normalize pixel values for CNN (using your normalization code)
x_train_normalized = x_train.astype('float32') / 255.0
x_test_normalized = x_test.astype('float32') / 255.0

# Verify normalization (your verification)
print("Normalized training data min pixel value:", np.min(x_train_normalized))
```

```
print("Normalized training data max pixel value:", np.max(x_train_normalized))
print("Normalized training data shape:", x_train_normalized.shape)

# 5. One-hot encode the labels for CNN
print("\nCreating one-hot encoded labels for CNN...")
y_train_onehot = to_categorical(y_train, 10)
y_test_onehot = to_categorical(y_test, 10)

# Display the original and one-hot encoded labels for verification
print("Original label format (first 5 examples):")
print(y_train[:5])
print("\nOne-hot encoded label format (first 5 examples):")
print(y_train_onehot[:5])
print("\nOne-hot encoded shape:", y_train_onehot.shape)

# 6. Prepare data for Traditional CV (Bag of Words) approach
print("\nPreparing data for Bag of Words approach...")
# Convert to grayscale and resize to 64x64 for better feature extraction
x_train_gray = np.zeros((x_train.shape[0], 64, 64), dtype=np.uint8)
x_test_gray = np.zeros((x_test.shape[0], 64, 64), dtype=np.uint8)

for i in range(len(x_train)):
    gray = cv2.cvtColor(x_train[i], cv2.COLOR_RGB2GRAY)
    x_train_gray[i] = cv2.resize(gray, (64, 64))

for i in range(len(x_test)):
    gray = cv2.cvtColor(x_test[i], cv2.COLOR_RGB2GRAY)
    x_test_gray[i] = cv2.resize(gray, (64, 64))

print("Grayscale training data shape:", x_train_gray.shape)
print("Grayscale test data shape:", x_test_gray.shape)

# 7. Create validation split for CNN
print("\nCreating validation split for CNN approach...")
x_train_cnn, x_val_cnn, y_train_cnn_onehot, y_val_cnn_onehot = train_test_split(
    x_train_normalized, y_train_onehot, test_size=0.1, random_state=42, stratify=y_train
)

# Also create a validation split with original labels (for metrics calculation)
_, _, y_train_cnn, y_val_cnn = train_test_split(
    x_train, y_train, test_size=0.1, random_state=42, stratify=y_train
)

print("CNN training data shape:", x_train_cnn.shape)
print("CNN validation data shape:", x_val_cnn.shape)
print("CNN training labels shape (one-hot):", y_train_cnn_onehot.shape)
print("CNN validation labels shape (one-hot):", y_val_cnn_onehot.shape)

# 8. Visualize the effect of preprocessing
plt.figure(figsize=(12, 6))
```

```
# 9.1 Original RGB image
plt.subplot(1, 3, 1)
plt.imshow(x_train[0])
plt.title("Original RGB")
plt.axis('off')

# 9.2 Grayscale image (for BoW)
plt.subplot(1, 3, 2)
plt.imshow(x_train_gray[0], cmap='gray')
plt.title("Grayscale (BoW)")
plt.axis('off')

# 9.3 Normalized image (for CNN)
plt.subplot(1, 3, 3)
plt.imshow(x_train_normalized[0])  # Will appear the same but pixel values are [0,1]
plt.title("Normalized (CNN)")
plt.axis('off')

plt.suptitle("Comparison of Preprocessing Steps")
plt.tight_layout()
plt.savefig('cifar10_preprocessing.png', dpi=300)
plt.show()

# 10. Save processed data for faster loading in the next steps
print("\nSaving processed data...")

# For traditional CV approach
# np.save('cifar10_train_gray.npy', x_train_gray)
# np.save('cifar10_test_gray.npy', x_test_gray)
np.save('cifar10_train_gray_64.npy', x_train_gray)
np.save('cifar10_test_gray_64.npy', x_test_gray)
np.save('cifar10_train_labels.npy', y_train)
np.save('cifar10_test_labels.npy', y_test)

# For CNN approach
np.save('cifar10_train_cnn.npy', x_train_cnn)
np.save('cifar10_val_cnn.npy', x_val_cnn)
np.save('cifar10_test_cnn.npy', x_test_normalized)
np.save('cifar10_train_cnn_labels.npy', y_train_cnn)
np.save('cifar10_train_cnn_onehot.npy', y_train_cnn_onehot)
np.save('cifar10_val_cnn_onehot.npy', y_val_cnn_onehot)
np.save('cifar10_test_onehot.npy', y_test_onehot)

# 11. Summary
print("\nDataset preparation complete!")
print("Summary:")
print(f"- Total training images: {len(x_train)}")
print(f"- Total test images: {len(x_test)}")
print(f"- Image dimensions: {x_train.shape[1]}x{x_train.shape[2]} pixels, {x_train.shape[3]} channels")
```

```
print(f"- Number of classes: {len(classes)}")
print(f"- Labels converted to one-hot format for CNN")
print(f"- Saved files for BoW and CNN approaches")
print("Ready to proceed with feature extraction and model training!")
```

10.3 BOW PIPELINE:

```
import numpy as np
import cv2
from sklearn.cluster import MiniBatchKMeans
from sklearn.svm import SVC
from sklearn.metrics import (
    classification_report, accuracy_score, confusion_matrix,
    precision_recall_fscore_support
)
from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.model_selection import StratifiedKFold
from tqdm import tqdm
import matplotlib.pyplot as plt
import seaborn as sns
import pickle
import os
import time

# === CONFIGURATION ===
TRAIN_SIZE = 20000
TEST_SIZE = 10000
STEP_SIZE = 2
VOCAB_SIZE = 1000
SVM_C = 1.0
RESULT_DIR = "bow_results_final"
os.makedirs(RESULT_DIR, exist_ok=True)

# === Load Data ===
x_train_gray = np.load("cifar10_train_gray_64.npy")[:TRAIN_SIZE]
x_test_gray = np.load("cifar10_test_gray_64.npy")[:TEST_SIZE]
y_train = np.load("cifar10_train_labels.npy").ravel()[:TRAIN_SIZE]
y_test = np.load("cifar10_test_labels.npy").ravel()[:TEST_SIZE]

classes = ['airplane', 'automobile', 'bird', 'cat', 'deer',
           'dog', 'frog', 'horse', 'ship', 'truck']

# === Contrast Enhancement (CLAHE) ===
def apply_clahe(images):
    clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8, 8))
    return np.array([clahe.apply(img) for img in images])

x_train_clahe = apply_clahe(x_train_gray)
x_test_clahe = apply_clahe(x_test_gray)
```

```
# === Dense SIFT ===
def extract_dense_sift(images, step_size=STEP_SIZE):
    sift = cv2.SIFT_create()
    descriptors_list = []
    for img in tqdm(images, desc="Extracting Dense SIFT"):
        kp = [cv2.KeyPoint(x, y, step_size)
              for y in range(0, img.shape[0], step_size)
              for x in range(0, img.shape[1], step_size)]
        _, des = sift.compute(img, kp)
        descriptors_list.append(des if des is not None else np.array([]))
    return descriptors_list

train_desc = extract_dense_sift(x_train_clahe)
test_desc = extract_dense_sift(x_test_clahe)

# === Codebook with MiniBatchKMeans ===
def create_codebook(descriptors_list, k=VOCAB_SIZE, max_samples=200000):
    all_desc = np.vstack([d for d in descriptors_list if d.size > 0])
    if len(all_desc) > max_samples:
        all_desc = all_desc[np.random.choice(len(all_desc), max_samples, replace=False)]
    kmeans = MiniBatchKMeans(n_clusters=k, batch_size=1000, random_state=42)
    kmeans.fit(all_desc.astype(np.float32))
    return kmeans

codebook = create_codebook(train_desc, k=VOCAB_SIZE)

# === Histograms ===
def build_bow_histograms(descriptors_list, codebook, k):
    histograms = np.zeros((len(descriptors_list), k), dtype=np.float32)
    for i, descriptors in enumerate(descriptors_list):
        if descriptors.size > 0:
            words = codebook.predict(descriptors.astype(np.float32))
            for word in words:
                histograms[i, word] += 1
        else:
            histograms[i] = np.ones(k)
    return histograms

train_hist = build_bow_histograms(train_desc, codebook, VOCAB_SIZE)
test_hist = build_bow_histograms(test_desc, codebook, VOCAB_SIZE)

# === TF-IDF Normalization ===
tfidf = TfidfTransformer(norm='l2')
train_hist_tfidf = tfidf.fit_transform(train_hist).toarray()
test_hist_tfidf = tfidf.transform(test_hist).toarray()

# === Train SVM ===
start_time = time.time()
svm = SVC(kernel='rbf', C=SVM_C, gamma='scale', random_state=42)
```

```
svm.fit(train_hist_tfidf, y_train)
train_time = time.time() - start_time
# === Evaluate on Training Data ===
y_train_pred = svm.predict(train_hist_tfidf)

train_acc = accuracy_score(y_train, y_train_pred)
train_report = classification_report(y_train, y_train_pred, target_names=classes, digits=3)
train_cm = confusion_matrix(y_train, y_train_pred)

train_precision_macro, train_recall_macro, train_f1_macro, _ = precision_recall_fscore_support(
    y_train, y_train_pred, average='macro')
train_precision_weighted, train_recall_weighted, train_f1_weighted, _ = precision_recall_fscore_support(
    y_train, y_train_pred, average='weighted')

print(f"\nBoW Train Accuracy: {train_acc * 100:.2f}%")
print("\nTraining Classification Report:")
print(train_report)
print(f"\nMacro Avg — Precision: {train_precision_macro:.3f}, Recall: {train_recall_macro:.3f}, F1:
{train_f1_macro:.3f}")
print(f"Weighted Avg — Precision: {train_precision_weighted:.3f}, Recall: {train_recall_weighted:.3f},
F1: {train_f1_weighted:.3f}")

# === Confusion Matrix (Training Data) ===
plt.figure(figsize=(10, 8))
sns.heatmap(train_cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=classes, yticklabels=classes)
plt.title("Confusion Matrix (Training Set)")
plt.xlabel("Predicted")
plt.ylabel("True")
plt.tight_layout()
plt.savefig(f"{RESULT_DIR}/bow_train_confusion_matrix.png", dpi=300)
plt.show()

# === Evaluate on Test Data ===
start_time = time.time()
y_pred = svm.predict(test_hist_tfidf)
test_time = time.time() - start_test

acc = accuracy_score(y_test, y_pred)
report = classification_report(y_test, y_pred, target_names=classes, digits=3)
conf_matrix = confusion_matrix(y_test, y_pred)

precision_macro, recall_macro, f1_macro, _ = precision_recall_fscore_support(
    y_test, y_pred, average='macro')
precision_weighted, recall_weighted, f1_weighted, _ = precision_recall_fscore_support(
    y_test, y_pred, average='weighted')

print(f"\n313/313 [==============================] - {test_time:.0f}s {int((test_time % 1) *
1000)}ms/step")
```

```
print(f"\nBoW Test Accuracy: {acc * 100:.2f}%")
print(f"\nFull Classification Report:")
print(report)
print(f"\nMacro Avg — Precision: {precision_macro:.3f}, Recall: {recall_macro:.3f}, F1:
{f1_macro:.3f}")
print(f"Weighted Avg — Precision: {precision_weighted:.3f}, Recall: {recall_weighted:.3f}, F1:
{f1_weighted:.3f}")

# === Confusion Matrix (Test) ===
plt.figure(figsize=(10, 8))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues',
            xticklabels=classes, yticklabels=classes)
plt.title("BoW Confusion Matrix (Test Set)")
plt.xlabel("Predicted")
plt.ylabel("True")
plt.tight_layout()
plt.savefig(f"{RESULT_DIR}/cnn_test_confusion_matrix.png", dpi=300)
plt.show()

# === Clustering plot for BoW TF-IDF features (Test) ===
from sklearn.manifold import TSNE
import matplotlib.pyplot as plt
import seaborn as sns

def plot_tsne_bow(features, labels, dataset_name="BoW"):
    print(f"\nGenerating t-SNE scatter plot for {dataset_name} set...")

    # Apply t-SNE
    tsne = TSNE(n_components=2, perplexity=30, n_iter=1000, random_state=42)
    reduced = tsne.fit_transform(features)

    # Plot
    plt.figure(figsize=(10, 8))
    scatter = plt.scatter(reduced[:, 0], reduced[:, 1], c=labels, cmap='tab10', alpha=0.8)

    legend_labels = [plt.Line2D([0], [0], marker='o', color='w',
                                label=label, markerfacecolor=scatter.cmap(i / 10), markersize=10)
                     for i, label in enumerate(classes)]

    plt.legend(handles=legend_labels, title="Classes", bbox_to_anchor=(1.05, 1), loc='upper left')
    plt.title(f"t-SNE Scatter Plot ({dataset_name} Set - BoW TF-IDF Features)")
    plt.xlabel("t-SNE Component 1")
    plt.ylabel("t-SNE Component 2")
    plt.tight_layout()
    plt.savefig(f"tsne_bow_{dataset_name.lower()}.png", dpi=300)
    plt.show()

# === Generate t-SNE plots for BoW TF-IDF features ===
plot_tsne_bow(test_hist_tfidf, y_test, "Test")
```

10.4 CNN PIPELINE:

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import tensorflow as tf
from sklearn.metrics import (classification_report, confusion_matrix,
                             accuracy_score, precision_score, recall_score, f1_score)
from tensorflow.keras.models import Model
from tensorflow.keras.layers import (Input, Conv2D, BatchNormalization, ReLU, Add,
                                     GlobalAveragePooling2D, Dense, Dropout, Reshape, multiply)
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.optimizers.schedules import CosineDecay

# Load CIFAR-10 preprocessed .npy files
x_train = np.load('cifar10_train_cnn.npy')
x_val = np.load('cifar10_val_cnn.npy')
x_test = np.load('cifar10_test_cnn.npy')

y_train = np.load('cifar10_train_cnn_onehot.npy')
y_val = np.load('cifar10_val_cnn_onehot.npy')
y_test = np.load('cifar10_test_onehot.npy')

classes = ['airplane', 'automobile', 'bird', 'cat', 'deer',
           'dog', 'frog', 'horse', 'ship', 'truck']

# === Squeeze-and-Excitation Block ===
def se_block(input_tensor, ratio=16):
    filters = input_tensor.shape[-1]
    se = GlobalAveragePooling2D()(input_tensor)
    se = Dense(filters // ratio, activation='relu')(se)
    se = Dense(filters, activation='sigmoid')(se)
    se = Reshape((1, 1, filters))(se)
    return multiply([input_tensor, se])

# === Residual Block with SE ===
def residual_block(x, filters, downsample=False):
    shortcut = x
    stride = 2 if downsample else 1

    x = Conv2D(filters, (3, 3), strides=stride, padding='same')(x)
    x = BatchNormalization()(x)
    x = ReLU()(x)

    x = Conv2D(filters, (3, 3), padding='same')(x)
```

```
    x = BatchNormalization()(x)

    if downsample or shortcut.shape[-1] != filters:
        shortcut = Conv2D(filters, (1, 1), strides=stride)(shortcut)
        shortcut = BatchNormalization()(shortcut)

    x = Add()([x, shortcut])
    x = se_block(x)
    x = ReLU()(x)
    return x

# === ResNet Model ===
def build_resnet_model(input_shape=(32, 32, 3), num_classes=10):
    inputs = Input(shape=input_shape)
    x = Conv2D(32, (3, 3), padding='same')(inputs)
    x = BatchNormalization()(x)
    x = ReLU()(x)

    x = residual_block(x, 32)
    x = residual_block(x, 32)

    x = residual_block(x, 64, downsample=True)
    x = residual_block(x, 64)

    x = residual_block(x, 128, downsample=True)
    x = residual_block(x, 128)
    x = residual_block(x, 128)
    x = residual_block(x, 128)

    x = GlobalAveragePooling2D()(x)
    x = Dropout(0.5)(x)
    x = Dense(256, activation='relu')(x)
    x = Dropout(0.5)(x)
    outputs = Dense(num_classes, activation='softmax')(x)

    return Model(inputs, outputs)

# Learning rate scheduler
cosine_schedule = CosineDecay(
    initial_learning_rate=0.001,
    decay_steps=75 * (len(x_train) // 64),
    alpha=1e-5
)

model = build_resnet_model()
model.compile(optimizer=Adam(learning_rate=cosine_schedule),
              loss=tf.keras.losses.CategoricalCrossentropy(label_smoothing=0.1),
              metrics=['accuracy'])
model.summary()
```

```
# === MixUp Data Augmentation ===
def mixup(generator, alpha=0.2):
    for x_batch, y_batch in generator:
        batch_size = x_batch.shape[0]
        l = np.random.beta(alpha, alpha)

        idx = np.random.permutation(batch_size)

        x_mix = l * x_batch + (1 - l) * x_batch[idx]
        y_mix = l * y_batch + (1 - l) * y_batch[idx]

        yield x_mix, y_mix

# === Data Generator with Augmentation ===
datagen = ImageDataGenerator(
    rotation_range=10,
    width_shift_range=0.1,
    height_shift_range=0.1,
    horizontal_flip=True
)
datagen.fit(x_train)
train_generator = mixup(datagen.flow(x_train, y_train, batch_size=64))

# === Callbacks ===
early_stop = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)

# === Training ===
history = model.fit(
    train_generator,
    steps_per_epoch=len(x_train) // 64,
    validation_data=(x_val, y_val),
    epochs=75,
    callbacks=[early_stop],
    verbose=2
)

# === Save model in HDF5 format ===
model.save('enhanced_resnet_cifar10_model.h5')
print("Enhanced ResNet model saved!")

# === Plot training curves ===
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(history.history['loss'], label="Train Loss")
plt.plot(history.history['val_loss'], label="Val Loss")
plt.title("Loss over Epochs")
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(history.history['accuracy'], label="Train Accuracy")
```

```
plt.plot(history.history['val_accuracy'], label="Val Accuracy")
plt.title("Accuracy over Epochs")
plt.legend()
plt.tight_layout()
plt.savefig("cnn_training_curves_final.png", dpi=300)
plt.show()

from sklearn.metrics import classification_report, confusion_matrix, accuracy_score, precision_score,
recall_score, f1_score
from tensorflow.keras.models import load_model

def evaluate_model(x_data, y_true_onehot, dataset_name="Test"):
    # Load the model
    # model = load_model('resnet_cifar10_model.h5')
    model = load_model('enhanced_resnet_cifar10_model.h5')
    start_time = time.time()
    y_pred_probs = model.predict(x_data)
    end_time = time.time()

    # Calculate metrics
    total_time = end_time - start_time
    num_images = x_test.shape[0]
    fps = num_images / total_time
    time_per_image = total_time / num_images

    print(f"🕒 Inference Time for {num_images} images: {total_time:.3f}s")
    print(f"⚡ Average Inference Time per Image: {time_per_image * 1000:.2f} ms")
    print(f"🎯 Throughput: {fps:.2f} FPS")

    y_pred_labels = np.argmax(y_pred_probs, axis=1)
    y_true_labels = np.argmax(y_true_onehot, axis=1)

    acc = accuracy_score(y_true_labels, y_pred_labels)
    macro_p = precision_score(y_true_labels, y_pred_labels, average='macro')
    macro_r = recall_score(y_true_labels, y_pred_labels, average='macro')
    macro_f = f1_score(y_true_labels, y_pred_labels, average='macro')

    weighted_p = precision_score(y_true_labels, y_pred_labels, average='weighted')
    weighted_r = recall_score(y_true_labels, y_pred_labels, average='weighted')
    weighted_f = f1_score(y_true_labels, y_pred_labels, average='weighted')

    print(f"\nResNet {dataset_name} Accuracy: {acc*100:.2f}%")
    print(f"\nResNet {dataset_name} Classification Report:")
    print(classification_report(y_true_labels, y_pred_labels, target_names=classes, digits=3))
    print(f"\nMacro Avg — Precision: {macro_p:.3f}, Recall: {macro_r:.3f}, F1: {macro_f:.3f}")
    print(f"Weighted Avg — Precision: {weighted_p:.3f}, Recall: {weighted_r:.3f}, F1: {weighted_f:.3f}")

    # Confusion Matrix
    cm = confusion_matrix(y_true_labels, y_pred_labels)
```

```
    plt.figure(figsize=(10, 8))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=classes, yticklabels=classes)
    plt.title(f"ResNet Confusion Matrix ({dataset_name} Set)")
    plt.xlabel("Predicted")
    plt.ylabel("True")
    plt.tight_layout()
    plt.savefig(f"confusion_matrix_{dataset_name.lower()}.png", dpi=300)
    plt.show()

# === Evaluate on Train Set ===
evaluate_model(x_train, y_train, "Train")

# === Evaluate on Test Set ===
evaluate_model(x_test, y_test, "Test")

# === Clustering Plot ===
from sklearn.manifold import TSNE
from tensorflow.keras.models import Model

def plot_tsne_scatter(x_data, y_data, dataset_name="Test"):
    print(f"\nGenerating t-SNE scatter plot for {dataset_name} set...")

    # Extract features
    model = load_model('enhanced_resnet_cifar10_model.h5')
    feature_extractor = Model(inputs=model.input, outputs=model.layers[-2].output)
    features = feature_extractor.predict(x_data)

    # Reduce with t-SNE (2D)
    tsne = TSNE(n_components=2, random_state=42, perplexity=30, n_iter=1000)
    reduced = tsne.fit_transform(features)

    y_labels = np.argmax(y_data, axis=1)

    # Plot
    plt.figure(figsize=(10, 8))
    scatter = plt.scatter(reduced[:, 0], reduced[:, 1], c=y_labels, cmap='tab10', alpha=0.8)
    legend_labels = [plt.Line2D([0], [0], marker='o', color='w',
                                label=label, markerfacecolor=scatter.cmap(i/10), markersize=10)
                     for i, label in enumerate(classes)]
    plt.legend(handles=legend_labels, title="Classes", bbox_to_anchor=(1.05, 1), loc='upper left')
    plt.title(f"t-SNE Feature Scatter Plot ({dataset_name} Set)")
    plt.xlabel("t-SNE Component 1")
    plt.ylabel("t-SNE Component 2")
    plt.tight_layout()
    plt.savefig(f"tsne_plot_{dataset_name.lower()}.png", dpi=300)
    plt.show()
# Generate t-SNE plots (Train)
plot_tsne_scatter(x_train, y_train, "Train")
# Generate t-SNE plots (Test)
plot_tsne_scatter(x_test, y_test, "Test")
```