COMP64301: Cognitive Robotics and Computer Vision Assignment

Zhikun Peng (14247379)

December 3, 2025

1 Introduction

1.1 Background

Object recognition is a basic ability for cognitive robots that enables them to communicate with their surrounding environments via navigation, manipulation, and human-robot interaction (HRI) [1]. Previously, this area of research focused on the use of hand-crafted feature extraction techniques like the Scale-Invariant Feature Transform (SIFT) [2] and Histogram of Oriented Gradients (HOG), commonly aggregated in the Bag of Visual Words (BoW) model [3]. Although these methods provided interpretability and robustness to geometric transformations, they often struggle with high-level semantic variations.

In recent years, the new generation of deep learning methods – mainly focusing on CNNs – have become an emerging trend that outperforms many previous models. Models such as ResNet [4] have revolutionized computer vision by learning hierarchical feature representations directly from data, achieving state-of-the-art performance on large-scale benchmarks like ImageNet. But understanding how the classical and novel solutions match is still necessary to deploy efficient vision systems on the limited resources robotics platforms.

1.2 Research Objectives

The main goal of this report is to analyse the effectiveness of Deep Learning (ResNet18) and Traditional Computer Vision (SIFT+BoW). The research will concentrate on evaluating how well feature learning (CNN) and feature engineering (BoW) perform at different levels of difficulty. The impact of hyperparameters such as vocabulary size (BoW), batch size (CNN), etc., on classification accuracy and computation efficiency were explored systematically. Then the transfer learning's generalization capability on domain-specific small dataset was analyzed.

1.3 Introduction to the Dataset

To strictly evaluate the performance robustness of the mentioned algorithm, two kinds of datasets are utilized to simulate various appearance problems existing in robotic vision:

A subset of the Caltech 101 dataset [5] is used as the experiment benchmark. It includes various categories with relatively large inter-class differences, such as Airplanes, Motorbikes, Bonsai, and so on. The images belonging to different classes have different structures, making it a relatively coarse-grained classification. This dataset can be taken to verify whether the algorithm is correct.

The sub-dataset of the Oxford-IIIT Pet data set [6] was selected for the more challenging task (representing a finer classifying task), which included dogs and cats of different breeds, characterized by large intra-class variances (e. g., pose, scale) and small inter-class variances (e. g., similar fur texture and shape). As such, this was more analogous to a complex real-world scenario where a robot had to identify visually similar objects.

2 Methodology

Study used an effective comparative scheme and examined both feature extraction methods: traditional feature engineering (BoW) and modern feature learning using CNNs with transfer learning. It also made sure that all the images were

divided up before they were saved as train, validation and test splits in a ratio of 70:15:15. A fixed random seed of 42 was applied during this operation.

2.1 Deep Learning Approach: Convolutional Neural Networks (CNN)

2.1.1 Model Selection and Architecture

The ResNet18 architecture [4] was chosen for its efficient convergence and robust performance. It offers a favorable balance between computational efficiency on the NVIDIA RTX 4060 GPU and high performance for image classification tasks.

2.1.2 Transfer Learning Strategy

Since the dataset remains too small to generate ideal results, a transfer learning strategy is adopted to prevent overfitting and enhance computational efficiency. For instance, ImageNet pre-trained weights were used for initialization, with all convolutional backbone layers frozen to function as a fixed and reliable feature extractor. Only the final fully connected layer required optimization to map extracted features to ten target categories. The parameters of this fully connected layer were reset, enabling it to output the desired ten-class predictions.

2.1.3 Data Preprocessing and Optimisation

All input images were resized to 224×224 pixels, converted into PyTorch tensors, and normalized using the standard ImageNet values. To enhance generalization, random horizontal flipping was applied as a data augmentation technique during training. The training process employed the Adam optimizer in conjunction with the Cross-Entropy loss function. Additionally, systematic hyperparameter exploration was conducted to ensure the efficiency and stability of the convolutional neural network (CNN). Specifically, convergence behavior was evaluated by testing learning rates $\{0.001, 0.0005\}$, while batch sizes $\{32, 64\}$ were tested to analyze the trade-off between gradient stability and GPU utilization.

2.2 Traditional Approach:Bag of Visual Words(BoW)

2.2.1 Feature Extraction and Encoding

For Caltech101 and Oxford Pets datasets where objects' scales, orientations and illuminations vary greatly, SIFT [2] is chosen as local feature descriptor because of its robustness in facing aforementioned problems. All SIFT descriptors in the training set are pooled to learn a visual dictionary for describing images later on, which is represented by histograms after encoding every image.

2.2.2 Visual Vocabulary Construction and Hyperparameter Tuning

The visual vocabulary was constructed by clustering the aggregated SIFT descriptors using the K-Means algorithm. The number of clusters, denoted as K, represents the vocabulary size and serves as a critical hyperparameter governing the trade-off between discriminative power and computational efficiency. To quantify its impact, we systematically investigated $K \in \{50, 100, 200, 500\}$.

2.2.3 Classification via Support Vector Machines (SVM)

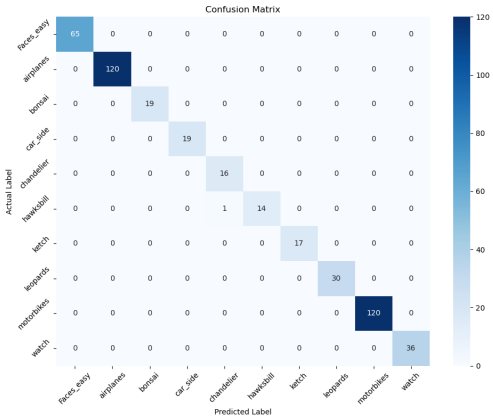
Using an SVM classifier based on radial basis functions (RBF), with BoF histograms as input. Two kernel functions (linear kernel and radial basis function kernel) were compared, with the RBF kernel ultimately selected as the subject of study for its ability to more accurately describe the distribution of nonlinear features. The regularization coefficient C was tested within the range $\{0.1, 1.0, 10.0\}$ to balance maximizing classification error with penalizing misclassified training points.

3 Experimental Setup and Results

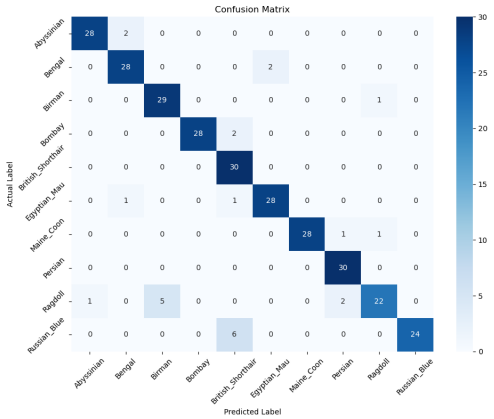
3.1 Experimental Configuration

The dataset was stratified by category proportion for training, validation, and testing sets to maintain balanced class distribution across each set. The final distribution was: training set 70%, validation set 15%, and testing set 15%. Model evaluation employed two primary metrics: test accuracy (%) as an effectiveness indicator and training time (seconds) as an efficiency indicator. All convolutional neural networks (CNNs) were trained on NVIDIA RTX 4060 GPUs, while the Bow experiment utilized only CPUs. From the Caltech101 dataset, the following 10 objects were selected:airplanes, motorbikes, faces_easy, watch, leopards, bonsai, car_side, ketch, chandelier, and hawksbill. In addition, ten cat or dog breeds were drawn from the Oxford Pet dataset, namely Abyssinian, Bengal, Birman, Bombay, British Shorthair, Egyptian Mau, Maine Coon, Persian, Ragdoll, and Russian Blue.

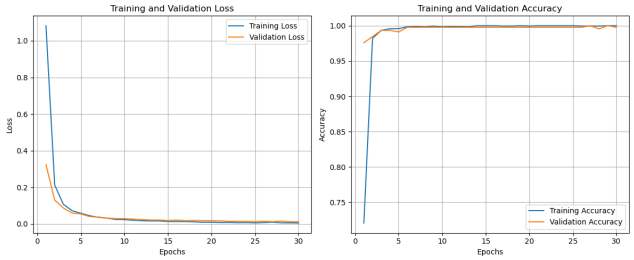
3.2 Results: CNN Hyperparameter Exploration



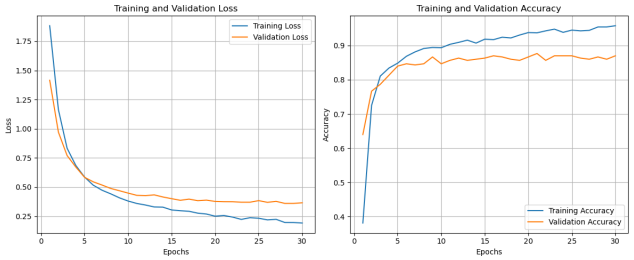
(a) Caltech101: Confusion Matrix



(b) Oxford Pets: Confusion Matrix



(c) Oxford Pets: Loss & Accuracy



(d) Oxford Pets: Loss & Accuracy

Figure 1: Visualization of CNN performance with Learning Rate=0.001, Batch Size=64 and Epochs=30. The (a) and (b) show the Confusion Matrices, where the diagonal represents correct predictions. The (c) and (d) show the Training/Validation Loss and Accuracy over epochs.

The confusion matrix in Figure 1 (a) exhibits a perfect diagonal distribution with no off-diagonal elements. This indicates the model perfectly distinguishes coarse-grained categories like airplanes and motorcycles, demonstrating that ImageNet pre-trained features exhibit “overfitting” characteristics for such simple tasks. According to Figure 1 (c), ResNet18 converges extremely rapidly on the Caltech101 dataset. By the third epoch, validation accuracy had already saturated near 100%, and the loss function rapidly dropped to zero.

The situation for Oxford Pets is slightly different. Although the final accuracy is high (around 90%+), as shown in Figure 1 (d), the loss decreases more gradually than Caltech, and a slight gap exists between the training and validation curves, suggesting a minor risk of overfitting. The confusion matrix in Figure 1 (b) reveals specific errors: the model occasionally confuses visually similar breeds (e.g., “Russian Blue” and “British Shorthair”).

The performance of the ResNet18 model under parts of various hyperparameter configurations is summarized in Table 1. The results indicate a significant dichotomy in task difficulty between the two datasets.

Table 1: Selected CNN Experimental Results. The table contrasts the saturation on the coarse-grained task (Caltech101) with the hyperparameter sensitivity on the fine-grained task (Oxford Pets).

Dataset	Batch Size	LR	Epochs	Test Acc (%)	Time (s)
Caltech101	64	0.001	15	99.78	**76**
Caltech101	32	0.001	30	**100.00**	166
Oxford Pets	64	0.0005	15	88.33	116
Oxford Pets	32	0.001	30	90.67	234
Oxford Pets	32	0.0005	30	**92.67**	225

On the Caltech101 dataset in Table 1, the model performed nearly flawlessly across all settings, achieving a maximum accuracy of 100%. Even when training rounds were reduced from 30 to 15, accuracy remained remarkably high at 99.78%. This demonstrates that the pre-trained ResNet excels at extracting high-level features (such as the shapes of airplanes or motorcycles), with hyperparameters having minimal impact on results. The model is sufficiently robust to accomplish this task.

Even though there were only 30 images in each class of the training data, the model did not suffer too much from overfitting. The "frozen backbone" method is attributed for this kind of stability because it constrained the tunable parameters in our CNN model to the last fully-connected layer, a useful regularizer for this work.

The Oxford Pet Dataset presents greater challenges due to high inter-class similarity, with models exhibiting high sensitivity to hyperparameters (see Table 1). Using a lower learning rate of 0.0005 paired with a batch size of 32 achieved the highest accuracy of 92.67%, surpassing the 90.67% accuracy obtained with a learning rate of 0.001. This indicates that fine-grained classification tasks benefit from smaller step updates, which optimize decision boundaries and reduce overfitting. Additionally, the dataset requires 30 training epochs to fully converge, representing a roughly 4% improvement over 15 epochs. In terms of efficiency, GPU acceleration delivers outstanding performance: Caltech101 converged in just 76 seconds, while the Oxford Pets experiment completed in under four minutes.

3.3 Results: BoW Parameter Evaluation

3.3.1 Parameter Sensitivity on Caltech101

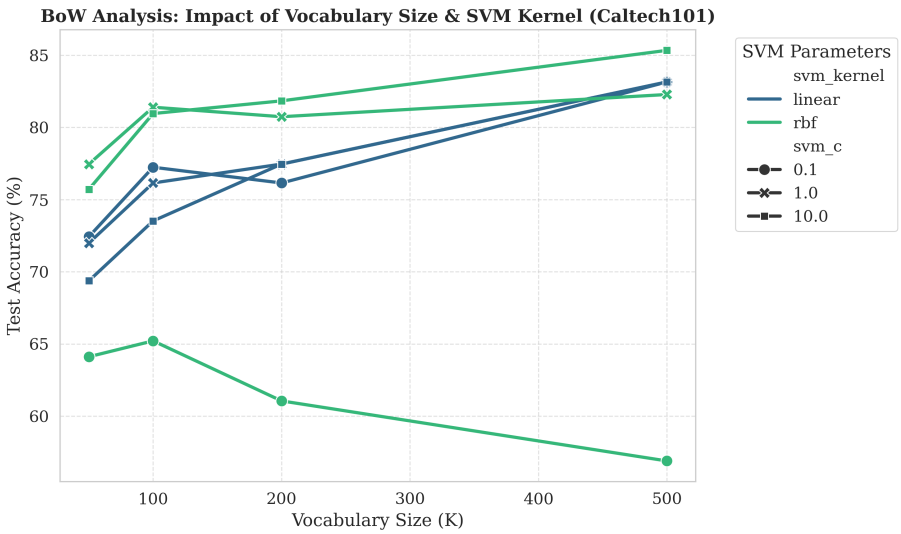


Figure 2: BoW Performance on Caltech101: Test Accuracy (%) vs. Vocabulary Size (K) for Linear and RBF SVM Kernels.

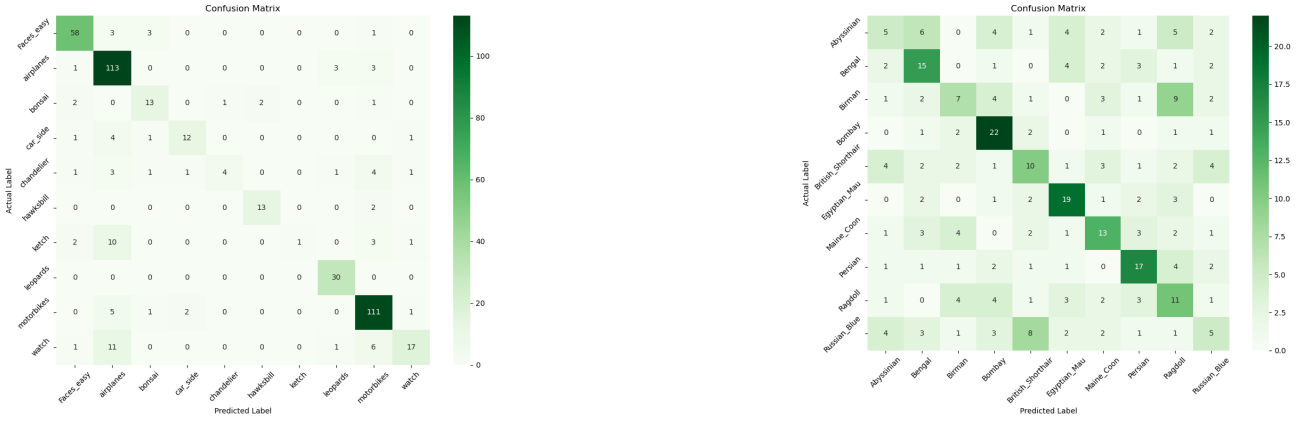
Figure 2 describes the Caltech 101 performance against two of the key hyperparameters, Vocabulary Size (K) and SVM Kernel. The experiments demonstrate that there is a high positive correlation between Vocabulary Size (K) and classification accuracy; the increase in K caused the increase in classification accuracy of 77.46% when K was 50(best accuracy) and 85.34% when K increased to 500(best accuracy).

The Radial Basis Function (RBF) kernel produced better results compared to Linear kernel with on average 3%-5% of the difference. In view of this, Bag of Words is represented as a higher dimensional feature space, thus suited to capturing

the inherent non-linear and complex relationship of High dimensional BoW space model, and therefore having more advantages than Linear kernel which cannot handle more complex functions.

When regularization coefficient (C) is set to very small value such as $C = 0.1$, the resulting prediction accuracy is quite low; such as only 56% accuracy for some K choices. This fact reflects an under-fitting situation when the penalty on model complexity is over-strong. On the contrary, the best model can be found when combining C=10.0 and RBF kernel to balance between maximizing the margin and allowing a few misclassified training points.

3.3.2 Analysis of Dataset Difficulty



(a) Caltech101: Confusion Matrix

(b) Oxford Pets: Confusion Matrix

Figure 3: Visualization of BoW performance with K=100, C=1.0 and Kernel=RBF. The (a) and (b) show the Confusion Matrices, where the diagonal represents correct predictions.

The confusion matrix for Oxford Pets (Figure 3 (b)) reveals that even with the optimal parameter combination ($K = 500$, RBF kernel), the bag-of-words model achieves a maximum accuracy of only 41.67% on the Oxford Pets dataset. The off-diagonal region exhibits numerous scattered misclassification points. This indicates the model struggles significantly to distinguish between different breeds of cats and dogs. The confusion is not confined to specific breed pairs but reflects a general inability to extract discriminative features.

4 Comparison and Discussion

4.1 Performance Comparison: Feature Engineering vs. Feature Learning

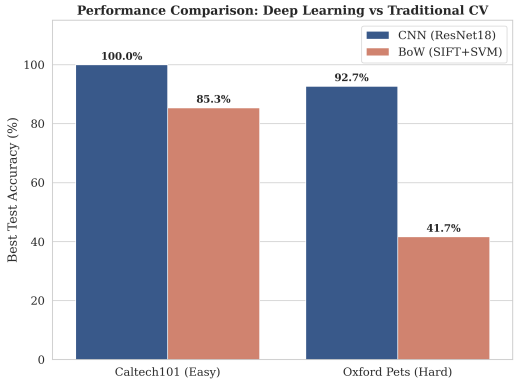


Figure 4: Performance Comparison between Deep Learning (CNN) and Traditional CV (BoW) across datasets. While both methods perform adequately on the coarse-grained task, a massive performance gap emerges on the fine-grained task.

A comparison of top scoring approaches as shown in Figure 4, the significant differences brought by the deep learning paradigm are clearly evident. Overall, on the coarse-grained Caltech101 dataset, performance variations exist but are not

substantial. While CNN achieves 100% accuracy, the BoW method still attains 85.3%. This suggests that for discriminative objects like those depicted here, manually designed features (such as SIFT) are sufficient to learn object patterns for retrieval.

While there is a difference in the coarse structure, the difference is huge when we consider the fine-grained Oxford Pet dataset. While the CNN still retains the accuracy at 92.7%, while the BoW's performance plummets to 41.7%. This gap is because the BoW model only pays attention to the frequency of occurrence of local features (e. g., corners, edges) and does not account for the location of these keypoint features, nor do they capture the semantics of each other. When classes are too similar (texture and shape wise), like fur, the "bag of words" cannot tell them apart. Instead, the hierarchical layers of ResNet18 can effectively abstract the high-level semantic features like ear shape, snout structure that serve as cues for breed discrimination.

4.2 Computational Efficiency and Resource Utilization

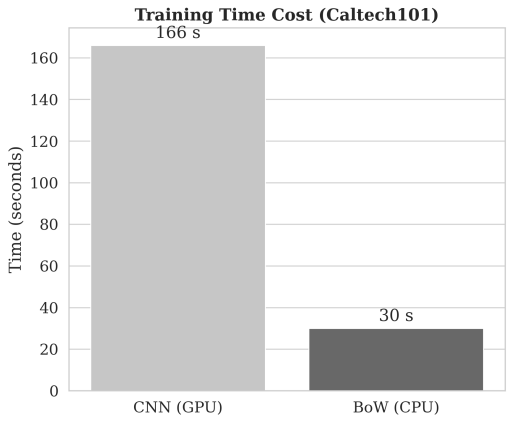


Figure 5: Training Time Cost Comparison on Caltech101. CNN training utilized an NVIDIA RTX 4060 GPU, while BoW training was performed on a AMD Ryzen 7 8845H CPU.

Figure 5 describes the trade-off between accuracy and computational costs, we can observe that the BoW model trains relatively faster for the Caltech101 dataset, it takes around 30 seconds to be trained on the CPU only. Therefore, for very constrained scenarios where GPUs are not available(like some low-power microcontrollers), this model is still computationally appealing because of its low-computational requirements.

Contrariwise, the CNN needed 166 sec. (for the 30-epoch run) and was run using an NVIDIA RTX 4060 GPU. Though the actual training time for CNN is much more, the drastic reduction through the help of transfer learning should be noted here. It would have taken hours, even days, to train a brand new deep network from scratch, while simply finetuning the output layer made our CNN achieve almost state-of-the-art accuracy within three minutes of training under the modern cognitive robotic system with widespread applications of GPUs (e.g., NVIDIA Jetson).

4.3 Summary of Trade-offs

In summary, CNNs achieve high accuracy and robustness through automatic feature learning, but they rely on hardware acceleration and offer limited interpretability. In contrast, BoW methods are lightweight, intuitive, and interpretable, yet their performance is constrained in fine-grained classification due to the absence of spatial encoding, and they are sensitive to parameter selection. This creates a trade-off: CNNs sacrifice computational resources for superior performance, while BoW methods are efficient but struggle with complex visual tasks.

5 State of the Art: Contextualisation in Robotics

Although convolutional neural networks excel in controlled classification tasks (Section 4), their application to autonomous systems requires addressing broader perception challenges beyond simple recognition. This section contextualizes our findings within the forefront of robot vision research.

Historically, robotics relied on geometric features (SIFT/ORB) for tasks like SLAM [7]. While robust for mapping, these methods lacked the semantic understanding required for interaction. Deep learning revolutionized this landscape by enabling semantic perception. As demonstrated experimentally, convolutional neural networks extract high-level abstract information, empowering robots to perform semantic navigation and grasping tasks [8].

However, the field is rapidly shifting toward embodied AI and multimodal learning. Unlike passive recognition, embodied AI posits that intelligence arises from physical interactions between agents and their environment [9]. Recent advancements like CLIP and RT-2 (Robot Transformer 2) [10] fuse visual data with natural language instructions through visual-language models (VLMs). This multimodal approach enables robots to generalize to unseen objects (zero-shot learning) and execute complex commands like "pick up the extinct animal toy" (referring to a dinosaur)—a breakthrough unattainable by monomodal convolutional neural networks or bag-of-words models.

Despite these advances, deploying large foundational models on mobile platforms remains constrained by stringent SWaP (size, weight, and power) limitations. Consequently, current research focuses on transfer learning from simulated to real environments and efficient model distillation techniques to bridge the gap between high-level semantic reasoning and real-time physical control.

6 Conclusion

This report compares Deep Learning and traditional Computer Vision approaches in object recognition of cognitive robots, where we empirically demonstrated that SIFT+BoW is adequate for coarse-grained tasks; but it cannot be adapted to finer-grained recognition tasks. By comparison, using transfer learning, the ResNet18-based CNN achieves better results steadily compared with other models, which indicates that learning features outperforms the hard-coded ones. In sum, Deep Learning is deemed to be an advantageous choice when handling a much greater variety of situations encountered by autonomous agents in challenging environments, but a future work focus will probably be more emphasis on seeking approaches to lighten up the learning process, such as the use of MobileNet architecture to tackle smaller errors and trade-off between higher computational cost.

References

[1] A. Cangelosi and M. Asada, "What is cognitive robotics?" In *Cognitive Robotics*, The MIT Press, May 2022, ISBN: 9780262369329. DOI: 10.7551/mitpress/13780.003.0005. eprint: https://direct.mit.edu/book/chapter-pdf/2239481/c000600_9780262369329.pdf. [Online]. Available: https://doi.org/10.7551/mitpress/13780.003.0005.

[2] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International Journal of Computer Vision*, vol. 60, no. 2, pp. 91–110, 2004.

[3] G. Csurka, C. R. Dance, L. Fan, J. Willamowski, and C. Bray, "Visual categorization with bags of keypoints," *workshop on statistical learning in computer vision eccv*, 2004.

[4] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *IEEE*, 2016.

[5] F.-F. Li, M. Andreeto, M. Ranzato, and P. Perona, *Caltech 101*, Apr. 2022. DOI: 10.22002/D1.20086.

[6] O. M. Parkhi, A. Vedaldi, A. Zisserman, and C. V. Jawahar, "Cats and dogs," in *2012 IEEE Conference on Computer Vision and Pattern Recognition*, 2012, pp. 3498–3505. DOI: 10.1109/CVPR.2012.6248092.

[7] C. Cadena et al., "Past, present, and future of simultaneous localization and mapping: Toward the robust-perception age," *IEEE Transactions on Robotics*, vol. 32, no. 6, pp. 1309–1332, 2016. DOI: 10.1109/TRO.2016.2624754.

[8] N. Sünderhauf et al., "The limits and potentials of deep learning for robotics," *The International journal of robotics research*, vol. 37, no. 4-5, pp. 405–420, 2018.

[9] J. Duan, S. Yu, H. L. Tan, H. Zhu, and C. Tan, "A survey of embodied ai: From simulators to research tasks," *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 6, no. 2, pp. 230–244, 2022.

[10] B. Zitkovich et al., "Rt-2: Vision-language-action models transfer web knowledge to robotic control," in *Conference on Robot Learning*, PMLR, 2023, pp. 2165–2183.

A Appendix: Source Code

A.1 Data Splitting Script

```python
#generate_splits_and_folders.py
import os
import glob
import pandas as pd
import numpy as np
import shutil
from sklearn.model_selection import train_test_split
from tqdm import tqdm

# ================ Configuration Area ================

# Original data root directory
RAW_DATA_DIR = r"D:\comp64301_cv\data\raw"
# Output directory (processed data will be saved here)
OUTPUT_DIR = r"D:\comp64301_cv\data\processed"

# 1. Caltech101 configuration
CALTECH_DIR = os.path.join(RAW_DATA_DIR, "caltech-101")
CALTECH_TARGET_CLASSES = [
    "airplanes", "motorbikes", "Faces_easy", "watch", "leopards",
    "bonsai", "car_side", "ketch", "chandelier", "hawksbill"
]

# 2. Oxford-IIIT Pet configuration
OXFORD_DIR = os.path.join(RAW_DATA_DIR, "oxfordiiit-pet", "images")
OXFORD_CLASS_COUNT = 10

# Split ratios (must sum to 1.0)
TRAIN_RATIO = 0.70
VAL_RATIO = 0.15
TEST_RATIO = 0.15
# ===================================================

def ensure_dir(directory):
    if not os.path.exists(directory):
        os.makedirs(directory)

def get_caltech_data():
    """Get Caltech101 data list"""
    data = []
    print(f"Scanning Caltech101: {CALTECH_DIR}...")

    # Note: Handle case sensitivity issues here, try to match folders
    if not os.path.exists(CALTECH_DIR):
        print(f"Warning: Path not found {CALTECH_DIR}")
        return pd.DataFrame()

    # Get the list of actual existing folders
    existing_folders = os.listdir(CALTECH_DIR)
    # Create a case-insensitive mapping dictionary
    folder_map = {f.lower(): f for f in existing_folders}

    for target_cls in CALTECH_TARGET_CLASSES:
        # Try to match
        actual_folder = folder_map.get(target_cls.lower())

        if actual_folder:
            class_path = os.path.join(CALTECH_DIR, actual_folder)
            images = glob.glob(os.path.join(class_path, "*.jpg"))
            for img_path in images:
                data.append({
                    "filepath": img_path,
                    "label": target_cls, # use original target class name
                    "dataset_source": "caltech101"
                })
        else:
            print(f"Warning: Folder for class '{target_cls}' not found in Caltech101")

    print(f"Caltech101: Found {len(data)} images.")
```

```
    return pd.DataFrame(data)

def get_oxford_data():
    """Get Oxford Pets data list"""
    data = []
    print(f"Scanning Oxford Pets: {OXFORD_DIR}...")

    if not os.path.exists(OXFORD_DIR):
        print(f"Warning: Path not found {OXFORD_DIR}")
        return pd.DataFrame()

    all_images = glob.glob(os.path.join(OXFORD_DIR, "*.jpg"))

    temp_list = []
    for img_path in all_images:
        filename = os.path.basename(img_path)
        # Oxford dataset filenames are usually Class_Name_Number.jpg
        # Method: find the last underscore and take the part to the left of it
        class_name = "_".join(filename.split("_")[:-1])
        temp_list.append((img_path, class_name))

    df = pd.DataFrame(temp_list, columns=["filepath", "label"])

    # Select top N classes in alphabetical order
    unique_classes = sorted(df["label"].unique())
    target_classes = unique_classes[:OXFORD_CLASS_COUNT]
    print(f"Oxford Pets selected top {OXFORD_CLASS_COUNT} classes: {target_classes}")

    # Filter data
    df_filtered = df[df["label"].isin(target_classes)].copy()
    df_filtered["dataset_source"] = "oxford_pet"

    print(f"Oxford Pets: Found {len(df_filtered)} images.")
    return df_filtered

def _safe_train_test_split(df, test_size, stratify_col="label", random_state=42):
    """Try stratified sampling, fallback to random sampling if fails (e.g., too few samples in a class)"""
    stratify = df[stratify_col] if stratify_col in df.columns else None
    try:
        return train_test_split(
            df, test_size=test_size, random_state=random_state, stratify=stratify
        )
    except ValueError as e:
        print(f"Warning: Stratified sampling failed, falling back to random sampling. Reason: {e}")
        return train_test_split(
            df, test_size=test_size, random_state=random_state, stratify=None, shuffle=True
        )

def organize_files_into_folders(df, dataset_name):
    """
    Based on the split column in the DataFrame, physically copy images into folder structures.
    Target structure: output_dir / dataset_name  / class_name / image.jpg
    """
    print(f"\nOrganizing file structure and copying images to: {os.path.join(OUTPUT_DIR, dataset_name)} ...")

    # Iterate over each row in the DataFrame
    # Use tqdm to show progress bar
    for _, row in tqdm(df.iterrows(), total=len(df), desc=f"Processing {dataset_name}"):
        src_path = row['filepath']
        split = row['split']   # train, val, test
        label = row['label']   # class name

        # Construct target directory
        # Example: D:\comp64301_cv\data\processed\caltech101\train\airplanes\
        target_dir = os.path.join(OUTPUT_DIR, dataset_name, split, label)

        # Ensure target directory exists
        os.makedirs(target_dir, exist_ok=True)

        # Construct target filename (to avoid name conflicts, keep original filename)
        filename = os.path.basename(src_path)
```

```
        target_path = os.path.join(target_dir, filename)

        # Copy file (copy2 preserves metadata like timestamps)
        try:
            shutil.copy2(src_path, target_path)
        except Exception as e:
            print(f"Copy failed: {src_path} -> {target_path}, Error: {e}")
def process_dataset(df, dataset_name):
    """Process a single dataset: split, save CSV, and organize folders"""
    if len(df) == 0:
        print(f"Error: {dataset_name} dataset is empty, skipping.")
        return

    print(f"\nProcessing dataset: {dataset_name}")
    print(f"Using ratios: train={TRAIN_RATIO}, val={VAL_RATIO}, test={TEST_RATIO}")
    # ================= Splitting logic =================
    parts = []

    # 1. Split Test set
    if TEST_RATIO > 0:
        train_val, test = _safe_train_test_split(df, test_size=TEST_RATIO, stratify_col="label")
        test = test.copy()
        test.loc[:, "split"] = "test"
        parts.append(test)
    else:
        train_val = df

    # 2. Split Train / Val
    remain_ratio = 1.0 - TEST_RATIO
    if remain_ratio > 0:
        # Calculate relative proportion of val in the remaining data
        # relative_val = target_val / (target_train + target_val)
        if (TRAIN_RATIO + VAL_RATIO) > 0:
            relative_val_size = VAL_RATIO / (TRAIN_RATIO + VAL_RATIO)

            train, val = _safe_train_test_split(train_val, test_size=relative_val_size,
    stratify_col="label")
            train = train.copy()
            val = val.copy()
            train.loc[:, "split"] = "train"
            val.loc[:, "split"] = "val"
            parts.extend([train, val])
        else:
            # Edge case handling
            train_val.loc[:, "split"] = "train" # Default to train
            parts.append(train_val)

    # Combine results
    final_df = pd.concat(parts, ignore_index=True)

    # ================= Save CSV =================
    csv_filename = f"{dataset_name}_split.csv"
    csv_path = os.path.join(OUTPUT_DIR, csv_filename)
    final_df.to_csv(csv_path, index=False)
    print(f"CSV saved: {csv_path}")
    print(f"Distribution stats:\n{final_df['split'].value_counts()}")

    # ================= Physical file copying =================
    organize_files_into_folders(final_df, dataset_name)
    print(f"{dataset_name} processing completed.")

if __name__ == "__main__":
    ensure_dir(OUTPUT_DIR)

    # 1. Caltech101
    df_caltech = get_caltech_data()
    process_dataset(df_caltech, "caltech101")

    # 2. Oxford Pets
    df_oxford = get_oxford_data()
    process_dataset(df_oxford, "oxford_pet")

    print("\nAll datasets processed successfully.")
```

Listing 1: generate_splits_and_folders.py

A.2 CNN Training Script

```python
#train_cnn.py
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, models, transforms
from torch.utils.data import DataLoader
import os
import time
import copy
import csv
from datetime import datetime
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
import seaborn as sns
import numpy as np

# ================= Setting Area =================
# Dataset directories
DATA_DIR_CALTECH101 = r"D:\comp64301_cv\data\processed\caltech101"
DATA_DIR_OXFORD_PET = r"D:\comp64301_cv\data\processed\oxford_pet"

# Model save directory
MODEL_SAVE_DIR = r"D:\comp64301_cv\results\cnn_models"
# Figure save directory
FIGURE_SAVE_DIR = r"D:\comp64301_cv\results\figures"
# Experiment results log file
LOG_CSV_FILE = r"D:\comp64301_cv\results\cnn_results.csv"

# Ensure save directories exist
os.makedirs(MODEL_SAVE_DIR, exist_ok=True)
os.makedirs(FIGURE_SAVE_DIR, exist_ok=True)
os.makedirs(os.path.dirname(LOG_CSV_FILE), exist_ok=True)

# Device (4060 cuda available)
DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")
# ================================================

def get_dataloaders(data_dir, batch_size=32):
    """
    Load data and apply normalization.
    ImageNet normalization parameters: mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]
    """
    data_transforms = {
        'train': transforms.Compose([
            transforms.Resize((224, 224)), # ResNet input standard size
            transforms.RandomHorizontalFlip(), # Data augmentation
            transforms.ToTensor(),
            transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
        ]),
        'val': transforms.Compose([
            transforms.Resize((224, 224)),
            transforms.ToTensor(),
            transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
        ]),
        'test': transforms.Compose([ # Test processing usually same as Val
            transforms.Resize((224, 224)),
            transforms.ToTensor(),
            transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
        ]),
    }

    image_datasets = {x: datasets.ImageFolder(os.path.join(data_dir, x), data_transforms[x])
                      for x in ['train', 'val', 'test']}

    # Pointing to 'test' subfolder
    test_dataset = datasets.ImageFolder(os.path.join(data_dir, 'test'), data_transforms['test'])

    # shuffle=False because no need to shuffle
    test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False, num_workers=0)

    # On Windows, num_workers=0 is usually the most stable to avoid multi-process errors
    dataloaders = {x: DataLoader(image_datasets[x], batch_size=batch_size, shuffle=(x=='train'),
```

```
    num_workers=0)
                   for x in ['train', 'val', 'test']}

    dataset_sizes = {x: len(image_datasets[x]) for x in ['train', 'val', 'test']}
    class_names = image_datasets['train'].classes

    return dataloaders, dataset_sizes, class_names

def initialize_model(num_classes, feature_extract=True):
    """
    Initialize ResNet18 model
    :param feature_extract: True = freeze backbone, only train FC layer; False = fine-tune entire
    network
    """
    print("Initializing ResNet18...")
    # Download/load pretrained weights
    model = models.resnet18(weights=models.ResNet18_Weights.DEFAULT)

    # If feature extracting, freeze parameters
    if feature_extract:
        for param in model.parameters():
            param.requires_grad = False # Key step: disable gradient computation

    # Replace the final fully connected layer (fc)
    # Newly created layers have requires_grad=True by default
    num_ftrs = model.fc.in_features
    model.fc = nn.Linear(num_ftrs, num_classes)

    return model

def train_model(model, dataloaders, dataset_sizes, criterion, optimizer, num_epochs=10):
    """
    Training loop function
    """
    since = time.time()

    best_model_wts = copy.deepcopy(model.state_dict())
    best_acc = 0.0

    # Used to record history (used for plotting)
    history = {'train_loss': [], 'train_acc': [], 'val_loss': [], 'val_acc': []}

    print(f"Starting training... Device: {DEVICE}")

    for epoch in range(num_epochs):
        print(f'Epoch {epoch+1}/{num_epochs}')
        print('-' * 10)

        # Each epoch has a training and validation phase
        for phase in ['train', 'val']:
            if phase == 'train':
                model.train()  # Training mode
            else:
                model.eval()   # Evaluation mode

            running_loss = 0.0
            running_corrects = 0

            # Iterate over data
            for inputs, labels in dataloaders[phase]:
                inputs = inputs.to(DEVICE)
                labels = labels.to(DEVICE)

                # Zero the gradients
                optimizer.zero_grad()

                # Forward propagation
                # Only track gradients in the training phase
                with torch.set_grad_enabled(phase == 'train'):
                    outputs = model(inputs)
                    _, preds = torch.max(outputs, 1)
                    loss = criterion(outputs, labels)

                    # Backpropagation + optimization (only in training phase)
```

```
                    if phase == 'train':
                        loss.backward()
                        optimizer.step()

                # Statistics
                running_loss += loss.item() * inputs.size(0)
                running_corrects += torch.sum(preds == labels.data)

            epoch_loss = running_loss / dataset_sizes[phase]
            epoch_acc = running_corrects.double() / dataset_sizes[phase]

            print(f'{phase} Loss: {epoch_loss:.4f} Acc: {epoch_acc:.4f}')

            # Record history
            history[f'{phase}_loss'].append(epoch_loss)
            history[f'{phase}_acc'].append(epoch_acc.item())

            # If in validation phase and accuracy improves, save model weights
            if phase == 'val' and epoch_acc > best_acc:
                best_acc = epoch_acc
                best_model_wts = copy.deepcopy(model.state_dict())

    time_elapsed = time.time() - since
    print(f'Training complete in {time_elapsed // 60:.0f}m {time_elapsed % 60:.0f}s')
    print(f'Best val Acc: {best_acc:.4f}')

    # Load best model weights
    model.load_state_dict(best_model_wts)
    # Modify return values: add time_elapsed
    return best_acc, model, history, time_elapsed

def evaluate_on_test(model, test_loader, device):
    """
    Evaluate model performance on the Test set
    """
    model.eval()   # Switch to evaluation mode

    correct = 0
    total = 0

    # To store all predictions and labels
    all_preds = []
    all_labels = []

    print("\nPerforming final evaluation on the Test set...")

    # Disable gradient calculation
    with torch.no_grad():
        for inputs, labels in test_loader:
            inputs = inputs.to(device)
            labels = labels.to(device)

            # Forward propagation
            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1)

            total += labels.size(0)
            correct += (predicted == labels).sum().item()

            # Store predictions and labels
            all_preds.extend(predicted.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())

    accuracy = 100 * correct / total
    print(f'Test Set Final Accuracy: {accuracy:.2f}%')

    #
    return accuracy, all_labels, all_preds

def plot_training_history(history, filename="training_curves.png"):
    """
    Plot and save the Loss and Accuracy curves during training
    """
    acc = history['train_acc']
```

```
    val_acc = history['val_acc']
    loss = history['train_loss']
    val_loss = history['val_loss']

    epochs_range = range(1, len(acc) + 1)

    plt.figure(figsize=(12, 5))

    # Plot Loss curves
    plt.subplot(1, 2, 1)
    plt.plot(epochs_range, loss, label='Training Loss')
    plt.plot(epochs_range, val_loss, label='Validation Loss')
    plt.legend(loc='upper right')
    plt.title('Training and Validation Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.grid(True)

    # Plot Accuracy curves
    plt.subplot(1, 2, 2)
    plt.plot(epochs_range, acc, label='Training Accuracy')
    plt.plot(epochs_range, val_acc, label='Validation Accuracy')
    plt.legend(loc='lower right')
    plt.title('Training and Validation Accuracy')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.grid(True)

    plt.tight_layout()

    save_path = os.path.join(FIGURE_SAVE_DIR, filename)
    plt.savefig(save_path)
    plt.close() # Close the figure to free memory
    print(f"Result curves saved to: {save_path}")

def log_experiment_result(csv_file, params, metrics):
    """
    Append a single experiment result to a CSV file
    :param csv_file: Path to the CSV file (e.g., 'experiment_logs.csv')
    :param params: Dictionary containing hyperparameters (e.g., lr, batch_size, model_name)
    :param metrics: Dictionary containing result metrics (e.g., val_acc, test_acc, time)
    """
    # Combine all data to be recorded
    # Add a timestamp to distinguish different batches of experiments
    record = {'timestamp': datetime.now().strftime("%Y-%m-%d %H:%M:%S")}
    record.update(params)   # Add hyperparameters
    record.update(metrics)  # Add results

    # 2. Check if the file exists (to decide whether to write the header)
    file_exists = os.path.isfile(csv_file)

    # 3. Get all column names (Header)
    fieldnames = list(record.keys())

    # 4. Write/append data
    try:
        with open(csv_file, mode='a', newline='', encoding='utf-8') as f:
            writer = csv.DictWriter(f, fieldnames=fieldnames)

            # If file does not exist, write the header
            if not file_exists:
                writer.writeheader()

            # Write the current experiment data
            writer.writerow(record)
            print(f"Experiment record saved to: {csv_file}")

    except Exception as e:
        print(f"Failed to write log: {e}")

def plot_confusion_matrix(y_true, y_pred, classes, filename="confusion_matrix.png"):
    """
    Calculate and plot the confusion matrix
    """
```

```
    # Calculate confusion matrix
    cm = confusion_matrix(y_true, y_pred)

    # Create figure
    plt.figure(figsize=(10, 8))

    # Use Seaborn to draw heatmap
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
                xticklabels=classes, yticklabels=classes)

    plt.title('Confusion Matrix')
    plt.xlabel('Predicted Label')
    plt.ylabel('Actual Label')
    plt.xticks(rotation=45)
    plt.yticks(rotation=45)
    plt.tight_layout()

    # Save the figure
    save_path = os.path.join(FIGURE_SAVE_DIR, filename)
    plt.savefig(save_path)
    plt.close()
    print(f"Confusion matrix saved to: {save_path}")

def run_experiment(dataset='caltech101', batch_size=32, learning_rate=0.001, num_epochs=5):
    """
    Main experiment function
    """
    print(f"\n=== Running experiment: Dataset={dataset} BS={batch_size}, LR={learning_rate},
    Epochs={num_epochs} ===")

    # 1. Prepare data
    if dataset == 'caltech101':
        dataloaders, dataset_sizes, class_names = get_dataloaders(DATA_DIR_CALTECH101, batch_size)
    elif dataset == 'oxford_pet':
        dataloaders, dataset_sizes, class_names = get_dataloaders(DATA_DIR_OXFORD_PET, batch_size)
    else:
        raise ValueError(f"Unsupported dataset: {dataset}")
    num_classes = len(class_names)
    print(f"Detected classes: {class_names}")

    # 2. Initialize model (freeze backbone)
    model = initialize_model(num_classes, feature_extract=True)
    model = model.to(DEVICE)

    # 3. Set optimizer
    optimizer = optim.Adam(model.fc.parameters(), lr=learning_rate)

    # 4. Set loss function
    criterion = nn.CrossEntropyLoss()

    # 5. Start training
    best_val_acc, model, history, duration = train_model(model, dataloaders, dataset_sizes,
    criterion, optimizer, num_epochs)

    # 6. Save model
    save_path = os.path.join(MODEL_SAVE_DIR, f"resnet18_{dataset}_lr{learning_rate}.pth")
    torch.save(model.state_dict(), save_path)
    print(f"Model saved to: {save_path}")

    # 7. Plot and save training results
    plot_filename = f"resnet18_{dataset}_lr{learning_rate}_bs{batch_size}_epochs{num_epochs}.png"
    plot_training_history(history, plot_filename)

    # 8. Load the best saved model for final testing
    best_model_path = os.path.join(MODEL_SAVE_DIR, f"resnet18_{dataset}_lr{learning_rate}.pth")
    model.load_state_dict(torch.load(best_model_path))
    test_acc, y_true, y_pred = evaluate_on_test(model, dataloaders['test'], DEVICE)
    # Plot confusion matrix
    cm_filename = f"cm_cnn_{dataset}_lr{learning_rate}_bs{batch_size}.png"
    plot_confusion_matrix(y_true, y_pred, class_names, cm_filename)

    # 9. Combine all data to be recorded into a CSV file
    # Prepare hyperparameters dictionary
    exp_params = {
```

```
        'dataset': dataset,
        'model': 'resnet18',
        'freeze_backbone': True,
        'batch_size': batch_size,
        'learning_rate': learning_rate,
        'epochs': num_epochs
    }

    # Prepare results dictionary
    exp_metrics = {
        'best_val_acc': round(best_val_acc.item(), 4) if torch.is_tensor(best_val_acc) else round(best_val_acc, 4),
        'test_acc': round(test_acc, 4),
        'training_time_sec': int(duration)
    }

    # CSV
    log_file = LOG_CSV_FILE
    log_experiment_result(log_file, exp_params, exp_metrics)

    return history

if __name__ == "__main__":
    # Modify parameters here for testing, including dataset selection, batch size, learning rate, and number of epochs
    lrs = [0.001] #0.001, 0.0005
    batches = [64] #32, 64
    epochs_list = [30] #15, 30

    for lr in lrs:
        for bs in batches:
            for epochs in epochs_list:
                run_experiment(dataset='oxford_pet', batch_size=bs, learning_rate=lr, num_epochs=epochs)
```

Listing 2: train_cnn.py

A.3 BoW Training Script

```
#train_bow.py
import os
import cv2
import numpy as np
from sklearn.cluster import MiniBatchKMeans
from sklearn.svm import SVC
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, confusion_matrix
import time
import csv
from datetime import datetime
from tqdm import tqdm
import matplotlib.pyplot as plt
import seaborn as sns

# ================ Configuration Area ================
# Dataset directories (Same as CNN)
DATA_DIR_CALTECH101 = r"D:\comp64301_cv\data\processed\caltech101"
DATA_DIR_OXFORD_PET = r"D:\comp64301_cv\data\processed\oxford_pet"

# Experiment results log file (Same file as CNN for easy comparison)
LOG_CSV_FILE = r"D:\comp64301_cv\results\bow_results.csv"
# Figure save directory
FIGURE_SAVE_DIR = r"D:\comp64301_cv\results\figures"

# Ensure log directory exists
os.makedirs(os.path.dirname(LOG_CSV_FILE), exist_ok=True)
os.makedirs(FIGURE_SAVE_DIR, exist_ok=True)
# ====================================================

def get_file_paths(root_dir):
    """Get file paths and labels from a directory"""
    file_paths = []
```

```
    labels = []
    classes = sorted(os.listdir(root_dir))

    for label_idx, class_name in enumerate(classes):
        class_dir = os.path.join(root_dir, class_name)
        if not os.path.isdir(class_dir):
            continue
        for fname in os.listdir(class_dir):
            if fname.lower().endswith(('.jpg', '.jpeg', '.png')):
                file_paths.append(os.path.join(class_dir, fname))
                labels.append(label_idx)
    return file_paths, labels, classes

def extract_sift_features(image_paths):
    print(f"Extracting SIFT features ({len(image_paths)} images)...")
    sift = cv2.SIFT_create()
    descriptors_list = []

    for path in tqdm(image_paths, desc="SIFT Extraction"):
        img = cv2.imread(path, cv2.IMREAD_GRAYSCALE)
        if img is None:
            descriptors_list.append(None)
            continue

        _, des = sift.detectAndCompute(img, None)
        descriptors_list.append(des)

    return descriptors_list

def build_vocabulary(descriptors_list, k=100):
    print(f"Building Visual Vocabulary (K={k})...")

    valid_descriptors = [des for des in descriptors_list if des is not None]
    if not valid_descriptors:
        raise ValueError("No features extracted. Check image paths.")

    all_descriptors = np.vstack(valid_descriptors)

    kmeans = MiniBatchKMeans(n_clusters=k, batch_size=1000, random_state=42, n_init='auto')
    kmeans.fit(all_descriptors)

    return kmeans

def extract_histograms(descriptors_list, kmeans_model):
    vocab_size = kmeans_model.n_clusters
    histograms = []

    for des in descriptors_list:
        if des is not None:
            # Predict which cluster each feature in this image belongs to (Visual Word)
            words = kmeans_model.predict(des)
            # Count the occurrences of each cluster
            hist, _ = np.histogram(words, bins=range(vocab_size + 1))
            # Normalize (L2 Norm) - This step is important for SVM
            norm = np.linalg.norm(hist)
            if norm == 0:
                histograms.append(hist)
            else:
                histograms.append(hist / norm)
        else:
            # If no features, give a zero vector
            histograms.append(np.zeros(vocab_size))

    return np.array(histograms)

def log_experiment_result(csv_file, params, metrics):
    record = {'timestamp': datetime.now().strftime("%Y-%m-%d %H:%M:%S")}
    record.update(params)
    record.update(metrics)

    file_exists = os.path.isfile(csv_file)
    fieldnames = list(record.keys())

    try:
```

```
        with open(csv_file, mode='a', newline='', encoding='utf-8') as f:
            writer = csv.DictWriter(f, fieldnames=fieldnames)
            if not file_exists:
                writer.writeheader()
            writer.writerow(record)
            print(f"Experiment record saved to: {csv_file}")
    except Exception as e:
        print(f"Failed to write log: {e}")

def run_experiment(dataset='caltech101', vocab_size=100, svm_c=1.0, svm_kernel='rbf'):
    """
    Main experiment function for BoW
    """
    start_time = time.time()
    print(f"\n=== Running BoW Experiment: Dataset={dataset}, K={vocab_size}, SVM_C={svm_c},
    SVM_Kernel={svm_kernel} ===")

    # 1. Select Dataset Path
    if dataset == 'caltech101':
        data_dir = DATA_DIR_CALTECH101
    elif dataset == 'oxford_pet':
        data_dir = DATA_DIR_OXFORD_PET
    else:
        raise ValueError(f"Unsupported dataset: {dataset}")

    train_dir = os.path.join(data_dir, "train")
    test_dir = os.path.join(data_dir, "test")

    train_paths, train_y, classes = get_file_paths(train_dir)
    test_paths, test_y, _ = get_file_paths(test_dir)

    print(f"Classes: {len(classes)}")

    # 2. Feature Extraction
    train_des = extract_sift_features(train_paths)
    test_des = extract_sift_features(test_paths)

    # 3. Clustering (Vocabulary Building)
    kmeans = build_vocabulary(train_des, k=vocab_size)

    # 4. Encoding
    print("Generating histograms...")
    X_train = extract_histograms(train_des, kmeans)
    X_test = extract_histograms(test_des, kmeans)

    # 5. Standardization
    scaler = StandardScaler()
    X_train = scaler.fit_transform(X_train)
    X_test = scaler.transform(X_test)

    # 6. SVM Training
    print(f"Training SVM (C={svm_c}, Kernel={svm_kernel})...")
    clf = SVC(kernel=svm_kernel, C=svm_c, random_state=42)
    clf.fit(X_train, train_y)

    # 7. Evaluation
    print("Evaluating on Test set...")
    y_pred = clf.predict(X_test)
    test_acc = accuracy_score(test_y, y_pred)

    # 8.Plot Confusion Matrix
    cm = confusion_matrix(test_y, y_pred)
    plt.figure(figsize=(10, 8))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Greens',
                xticklabels=classes, yticklabels=classes)
    plt.title("Confusion Matrix")
    plt.xlabel("Predicted Label")
    plt.ylabel("Actual Label")
    plt.xticks(rotation=45)
    plt.yticks(rotation=45)
    plt.tight_layout()
    plt.savefig(os.path.join(FIGURE_SAVE_DIR, f"cm_bow_{dataset}_K{vocab_size}_C{svm_c}_Kernel{
    svm_kernel}.png"))
    plt.close()
```

```
    duration = time.time() - start_time

    print("-" * 30)
    print(f"BoW Result: Test Acc = {test_acc*100:.2f}%, Time = {duration:.1f}s")
    print("-" * 30)

    # 9. Log Results
    exp_params = {
        'dataset': dataset,
        'method': 'BoW+SVM',
        'vocab_size': vocab_size,
        'svm_c': svm_c,
        'svm_kernel': svm_kernel
    }

    exp_metrics = {
        'test_acc': round(test_acc, 4),
        'training_time_sec': int(duration)
    }

    log_experiment_result(LOG_CSV_FILE, exp_params, exp_metrics)

if __name__ == "__main__":
    datasets = ['caltech101', 'oxford_pet']# 'caltech101', 'oxford_pet'
    vocab_sizes = [100]# 50, 100, 200, 500
    svm_cs = [1.0]# 0.1, 1.0, 10.0
    svm_kernels = ['rbf']# 'linear', 'rbf'

    for dataset in datasets:
        for vocab_size in vocab_sizes:
            for svm_c in svm_cs:
                for svm_kernel in svm_kernels:
                    run_experiment(dataset=dataset, vocab_size=vocab_size, svm_c=svm_c, svm_kernel
    =svm_kernel)
```

Listing 3: train_bow.py