Traditional vs CNN Comparison Experiment

Introduction

Object recognition enables robots to understand and interact with their environments, making it a key component in modern robotics [1]. This project compares two approaches to object recognition: a traditional Bag of Visual Words (BoW) method using local features, and a modern deep learning approach using Convolutional Neural Networks (CNNs).

Both methods are evaluated on two datasets: iCubWorld, which simulates realistic robotic perception in controlled settings, and COCO, a widely used benchmark containing cluttered real-world scenes. By systematically varying hyperparameters and architectures, this project aims to assess the strengths and limitations of each approach. Results are further contextualized within the current state of the art in robotics.

Datasets

This project uses two datasets with different characteristics to evaluate object recognition methods in both controlled and complex environments: iCubWorld and COCO.

iCubWorld is a robotics vision dataset designed for evaluating object recognition in a controlled setting. It includes images of a human demonstrator holding objects in front of the iCub robot's camera. This project uses the "mixed" acquisition setting, which introduces variability in lighting, background, and demonstrators. A subset was used to stay within the computational limits of Google Colab, consisting of 7,200 training images, 1,800 validation images, and 1,000 test images. There are 20 object classes, each with 500 images, and all images are 640 × 480 pixels captured from the robot's left eye camera. The dataset is relatively clean and centered, making it well-suited for assessing recognition performance under robotics-friendly conditions.

In contrast, the COCO (Common Objects in Context) dataset provides real-world, cluttered scenes with multiple objects per image, making it a more challenging benchmark. For this project, the dataset was adapted for single-label classification by assigning each image the most frequent object class, and the number of classes was reduced to the 20 most common to align with iCubWorld. This subset included 8,000 training images and 2,000 validation images (test set is unreleased). Unlike iCubWorld, COCO suffers from class imbalance (categories like "person" dominate) and the visual complexity of the images makes recognition more difficult. The label simplification also introduces noise, as many images contain multiple relevant objects. Despite this, COCO serves as a strong benchmark for testing model robustness in realistic, unconstrained environments.

Traditional Method: Bag of Words

BoW Methods

The Bag of Visual Words (BoW) model represents images using histograms of image features, where each "visual word" corresponds to a recurring pattern or texture in the dataset. It is composed of four main stages: feature extraction, visual vocabulary (codebook) construction, image representation, and classification.

- ➢ Feature extraction

Although the Bag of Visual Words framework supports a variety of feature detectors, I chose to focus exclusively on local feature descriptors, as these are most effective for object recognition. Local features are well-suited for robotics tasks because they detect and describe small, distinctive regions of an image, and are robust to partial occlusion, background clutter, and variations in object pose.

I experimented with two widely used local feature methods: SIFT (Scale-Invariant Feature Transform) and ORB (Oriented FAST and Rotated BRIEF). SIFT was selected for its strong invariance to scale, rotation, and illumination changes, which makes it particularly effective in visually challenging scenarios such as those found in COCO and under variable lighting in iCubWorld. It produces dense, high-dimensional descriptors that capture detailed information about local image structure. However, it is computationally expensive.

In contrast, ORB was chosen for its speed and low computational overhead. It uses the FAST corner detector and BRIEF binary descriptors, making it highly efficient and suitable for real-time or resource-limited environments like embedded systems, or, in this case, Google Colab. However, ORB is less robust to scale and rotation, and tends to perform worse in complex or noisy scenes.

To prepare images for feature extraction, all inputs were resized to 256×256 pixels and converted to grayscale. This preprocessing step reduces computational load and standardizes input dimensions across the dataset.

- ➢ Visual vocabulary creation

To convert the variable-length descriptor sets into fixed-length representations, the feature descriptors were clustered into a visual vocabulary. Each cluster center represents a "visual word," or a local pattern/texture that is common across images in the dataset.
I chose to explore two clustering methods: MiniBatch KMeans and Gaussian Mixture Models (GMM). KMeans performs hard assignment of descriptors to cluster centroids and was chosen for its scalability and efficiency. GMM enables soft assignment, allowing descriptors to contribute probabilistically to multiple clusters, potentially capturing more complex information. To make GMM computationally feasible, Principal Component Analysis (PCA) was applied to reduce descriptor dimensionality before clustering, and clustering was limited to a random sample of 100,000 descriptors.
I tested three vocabulary sizes (100, 250, and 500 words) to evaluate the trade-off between expressive power and computational cost. Smaller vocabularies risk underfitting by failing to capture visual diversity, while larger vocabularies may overfit or create more sparse histograms.

- ➢ Image representation

Each image was represented as a histogram of visual word occurrences. This transformed the variable-length descriptor set into a fixed-length vector, enabling classification.
Histograms were not normalized before classification. While normalization (e.g., L2 norm) is commonly used to account for differences in keypoint count across images, I used raw counts for simplicity. This may have introduced some bias toward images with more detected features, particularly in cluttered scenes like COCO. Normalization is a recommended step for future work to improve robustness.

- ➢ Classification

I experimented with two different classifiers: Support Vector Machines (SVM) and Gaussian Naive Bayes (GNB). SVM was chosen for its effectiveness in high-dimensional, sparse data scenarios like BoW. It is a discriminative classifier that performs well even with relatively small datasets and is robust to overfitting. GNB, a generative model, assumes feature independence and Gaussian-distributed features. While this assumption is often violated in practice, GNB was included as a fast, low-resource baseline. Unlike SVMs, which learn decision boundaries between classes, GNB models how data is distributed within each class and uses Bayes' rule to predict class membership.
Experimental setup: For each dataset, 24 BoW configurations were tested by systematically varying the feature descriptor (SIFT or ORB), clustering method (KMeans or GMM), vocabulary size (100, 250, or 500), and classifier (SVM or GNB).

BoW Results
To determine the best traditional model for each dataset, I first evaluated all 24 configurations using macro-averaged F1 score on the validation set. F1 score was chosen as the primary selection metric because it balances precision and recall, which is particularly important for datasets with class imbalance, such as COCO. After identifying the top-performing configuration for each dataset, I conducted a more detailed evaluation by calculating macro-averaged precision, recall, and F1 score, and I generated a confusion matrix to visualize class-wise performance. For iCubWorld, this evaluation was performed on a held-out test set to assess generalization, while for COCO (whose test set is not publicly available), the evaluation and confusion matrix were based on the validation set.
The best-performing BoW configuration on iCubWorld used SIFT descriptors, GMM clustering, a vocabulary size of 500, and an SVM classifier. It achieved a validation F1 score of 0.549, and on the test set, it obtained a macro precision of 0.6355, macro recall of 0.628, and an F1 score of 0.632. These results indicate good generalization despite the limited training set and high-dimensional input.

Table 1: Top 3 Best BoW Models for iCubWorld (Validation Set)

Descriptor	Clustering	Vocab size	Classifier	F1
SIFT	GMM	500	SVM	0.549
SIFT	KMeans	500	SVM	0.538
SIFT	GMM	250	SVM	0.474

Across experiments, SIFT-based models consistently outperformed ORB-based models. For example, the top SIFT+SVM models with GMM and KMeans reached validation F1 scores of 0.549 and 0.5377, respectively, while the best ORB-based configuration reached only 0.385. This supports the conclusion that SIFT's robustness to scale and rotation is valuable for robotics perception tasks involving varying viewpoints.
GMM generally outperformed KMeans in most configurations, especially at higher vocabulary sizes. This suggests that soft assignment provides a more expressive representation of local features. SVM classifiers outperformed Naive Bayes across nearly all settings, likely due to SVM's capacity to handle high-dimensional, sparse feature spaces without assuming feature independence.
The confusion matrix for the best iCub model showed strong class-wise performance overall, with occasional confusion between visually similar objects such as books and notebooks.

On the COCO dataset, overall performance was significantly lower due to the dataset's complexity, cluttered scenes, and the simplification of labels for single-label classification. The best BoW configuration used SIFT descriptors, KMeans clustering, a vocabulary size of 500, and SVM, achieving a validation F1 score of 0.107 with a macro precision of 0.113 and recall of 0.108.

Table 2: Top 3 Best BoW Models for COCO (Validation Set)

Descriptor	Clustering	Vocab size	Classifier	F1
SIFT	KMeans	500	SVM	0.107
SIFT	GMM	500	SVM	0.105
SIFT	GMM	500	NB	0.087

The BoW pipeline struggled with COCO due to its greater visual complexity, label noise from multi-object images, and class imbalance. The confusion matrix revealed frequent misclassification of many classes as "person", the most dominant class in the dataset. While SIFT again outperformed ORB, the performance gap was smaller, and all models exhibited signs of overfitting or poor generalization, likely due to the mismatch between COCO's visual complexity and BoW's limited capacity to model context.
Naive Bayes generally underperformed consistently, likely due to its assumption of feature independence, which is violated in histogram representations where bins can be correlated. Larger vocabulary sizes showed slight improvements in performance, but gains were small and sometimes plateaued or decreased due to increased sparsity.

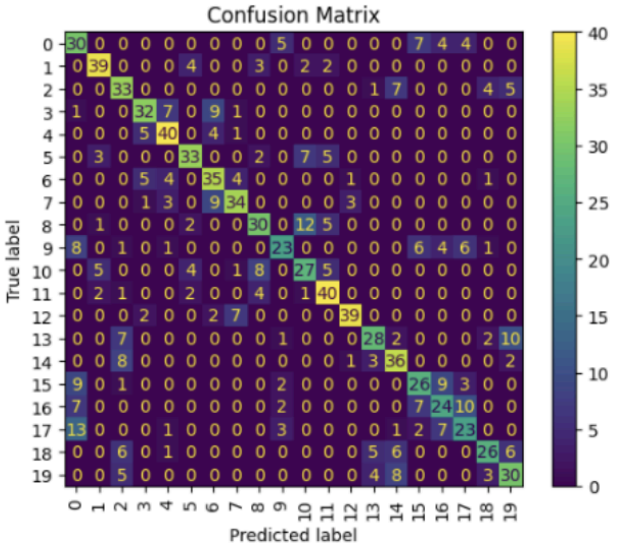

Figure 1: Confusion Matrix on the best BoW model for iCubWorld

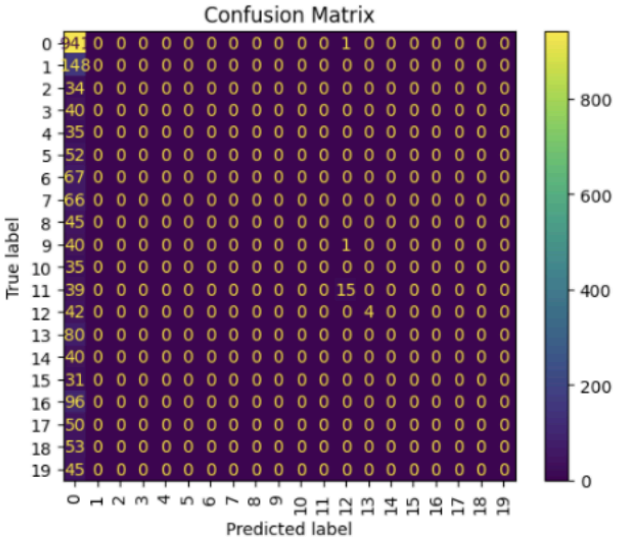

Figure 2: Confusion Matrix on the best BoW model for COCO

Limitations
The BoW approach worked reasonably well in controlled settings like iCubWorld but struggled in more realistic, cluttered environments like COCO. Performance could potentially be improved by applying L2 normalization to histograms, adapting COCO for multi-label classification, and sampling descriptors more intelligently for GMM to avoid underrepresentation of rare classes. Additionally, incorporating spatial information may help capture context that is lost in the histogram representation. Increasing the number of visual words could also improve discrimination by capturing finer-grained visual patterns, though this must be balanced against the risk of overfitting.

Deep Learning Approach: Convolutional Neural Network

CNN Methods

To compare against the traditional BoW approach, I implemented a deep learning pipeline using Convolutional Neural Networks (CNNs). Rather than training a model from scratch (which is computationally intensive and requires large amounts of labeled data), I used transfer learning with a pretrained ResNet-18 architecture. This allowed me to use visual features learned from the large-scale ImageNet dataset and fine-tune them for object recognition on my datasets.

- ➢ Model architecture

The base model was ResNet-18, a lightweight residual network pretrained on ImageNet. To adapt it for my task, I replaced the final fully connected layer with a custom classification head: a Dropout layer (dropout rate of 0.0, 0.3, or 0.5) followed by a Linear layer matching the number of classes in the dataset (20).
To balance efficiency with adaptability, I froze all layers except for the final residual block (layer4) and the classification head. This strategy allows the network to keep general-purpose low-level features (e.g., edges, textures) while enabling the deeper layers to adapt to the specific visual characteristics of iCubWorld and COCO. This is particularly useful given that these datasets differ from ImageNet in both content and scale.

- ➢ Data preprocessing

All input images were resized to 224×224 pixels, as required by ResNet. Pixel values were normalized using the ImageNet mean and standard deviation to match the input distribution of the pretrained network. To improve generalization, data augmentation (random horizontal flips and color jittering) was optionally applied to the training set. Validation and test sets remained unaugmented to ensure fair and consistent evaluation.

- ➢ Training setup

The model was trained using the CrossEntropyLoss function, which is standard for multi-class classification problems. For optimization, I experimented with Adam and Stochastic Gradient Descent (SGD). I used Adam as it adjusts learning rates dynamically for each parameter and converges quickly, especially with noisy gradients. I used SGD with momentum (.9) since it can help accelerate convergence by smoothing out oscillations in the optimization path. Although it is often slower to converge than Adam, SGD can sometimes lead to better generalization in the long term.
A StepLR scheduler reduced the learning rate by a factor of 0.1 every 5 epochs to support stable convergence. Training was performed for 10 epochs with a batch size of 32, balancing memory usage and learning stability given the constraints of Google Colab.

- ➢ Hyperparameter Tuning

I conducted 24 experiments per dataset by varying the dropout rate (0.0, 0.3, 0.5), learning rate (1e-3 or 1e-4), optimizer (Adam or SGD), and whether data augmentation was enabled.

CNN Results

Like with BoW, to determine the best traditional model for each dataset, I first evaluated all 24 configurations using macro-averaged F1 score on the validation set. After identifying the top-performing configuration for each dataset, I conducted a more detailed evaluation by calculating macro-averaged precision, recall, and F1 score, and I generated a confusion matrix to visualize class-wise performance. For iCubWorld, this evaluation was performed on a held-out test set to assess generalization, while for COCO (whose test set is not publicly available) the evaluation and confusion matrix were based on the validation set.

The best CNN configuration for iCubWorld used a dropout rate of 0.5, learning rate of 0.001, the Adam optimizer, and no data augmentation. This model achieved a validation F1 score of 0.963. On the test set, it achieved a macro precision of 0.9582, macro recall of 0.9577, and an overall F1 score of 0.958. These results indicate excellent generalization and strong class balance, even in a relatively small dataset.

Table 3: Top 3 Best CNN Models for iCubWorld (Validation Set)

Dropout	Learning rate	Optimizer	Augmentation	F1
0.5	0.001	adam	False	0.963
0.5	0.001	adam	True	0.961
0.5	0.0001	adam	False	0.959

The confusion matrix showed consistently high classification accuracy across all 20 classes, with minimal confusion between categories. This is likely due to the clean, centered nature of iCubWorld images and the consistency in object presentation.
Several trends emerged from the 24 experiments. Adam consistently outperformed SGD, often at lower learning rates. The best Adam-based models achieved F1 scores at or above 0.95, while most SGD models remained below 0.92. A dropout rate of 0.5 yielded the best regularization effect, outperforming both 0.0 and 0.3. Interestingly, data augmentation slightly decreased performance in the majority of cases, possibly due to the already consistent and clean nature of the dataset. A learning rate of 0.001 was generally more effective, except when combined with no dropout, where overfitting was more likely.

The best configuration for COCO used dropout = 0.0, learning rate = 0.0001, the Adam optimizer, and data augmentation enabled. This model achieved a validation F1 score of 0.312, with macro precision of 0.2987 and macro recall of 0.3345.

Table 4: Top 3 Best CNN Models for COCO (Validation Set)

Dropout	Learning rate	Optimizer	Augmentation	F1
0	0.0001	adam	True	0.312
0.5	0.001	sgd	False	0.311
0.5	0.001	sgd	True	0.310

Compared to iCubWorld, performance on COCO was significantly lower due to its high visual complexity, cluttered scenes, and label simplification. The confusion matrix revealed frequent misclassification of objects as “person,” the most common class, highlighting the challenges of class imbalance and overlapping object boundaries.
In terms of trends, the best-performing model used the Adam optimizer with a low learning rate, no dropout, and data augmentation. However, overall performance between Adam and SGD was comparable, with multiple top-performing models using SGD. Data augmentation was present in two of the top three models, suggesting a possible benefit for generalization on COCO’s noisy, cluttered scenes. Notably, the best-performing model did not use dropout, which may indicate that augmentation alone provided sufficient regularization in this context. No clear pattern emerged with respect to learning rate, suggesting that its impact may depend on interactions with other factors such as optimizer choice or regularization.

Limitations
Although the CNN models outperformed the BoW pipeline across both datasets, several limitations remain. The COCO adaptation to single-label classification introduced noise, likely affecting model performance. Additionally, training was limited to 10 epochs due to computational constraints, and further improvements may be possible with longer training or fine-tuning earlier ResNet layers. Lastly, while augmentation helped on COCO, more advanced techniques could further enhance robustness.

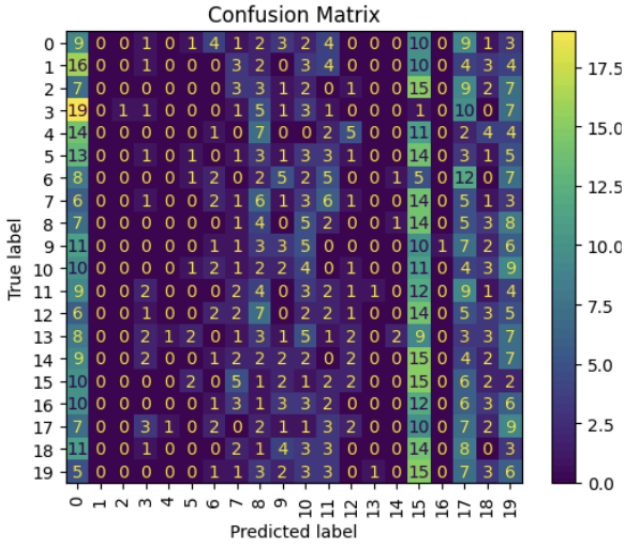

Figure 3: Confusion matrix on the best CNN model for iCubWorld

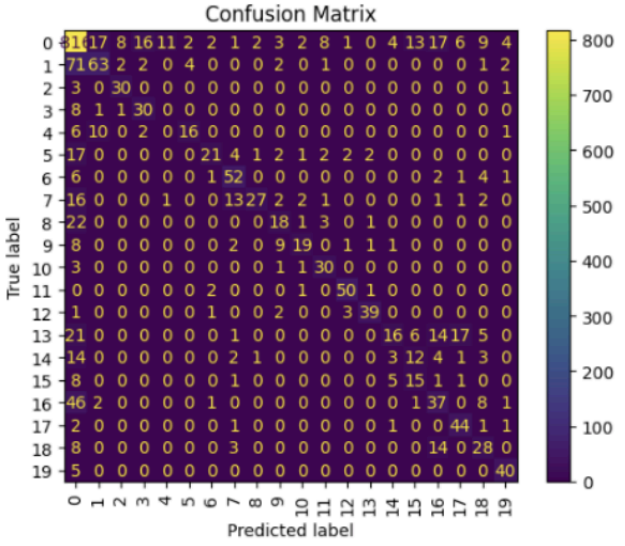

Figure 4: Confusion matrix on the best CNN model for COCO

Comparison between Methods

Overall, the CNN models consistently outperformed the traditional BoW pipeline in terms of classification accuracy and generalization. On iCubWorld, the best CNN achieved a macro-averaged F1 score of 0.963 on the test set, compared to 0.549 for the best BoW configuration. On COCO, the CNN also performed better (F1 = 0.312 vs. 0.107), despite the dataset's complexity and the label simplification used.

CNNs proved to be more robust across both datasets, particularly in visually complex scenes like COCO, where their ability to learn hierarchical, spatially-aware features was crucial. In contrast, BoW struggled to handle overlapping objects and background clutter due to its reliance on local descriptors and histogram-based representations, which discard spatial information.

That said, BoW had some advantages. It was significantly less computationally demanding, requiring no GPU acceleration. Its pipeline is also highly interpretable since it allows inspection at each stage (for instance, by examining which visual words were activated). In contrast, CNNs require more computation and are often considered "black box" models, which can be a limitation in human-computer interactions and other situations where safety is critical.

Contextualization in the Robotics Field

Object recognition is a foundational capability in robotics, enabling systems to interact with their environments by identifying, manipulating, and reasoning with objects [1]. However, deploying vision systems in real-world settings remains challenging due to dynamic lighting, partial occlusion, different perspectives, cluttered backgrounds, and limited onboard computational power [2, 3].

Early approaches relied on handcrafted feature descriptors such as SIFT or HOG, typically paired with classifiers like SVMs or detection frameworks like the Viola-Jones cascade (for faces)[1]. These pipelines followed rigid stages, such as feature detection, region proposal, and classification, but often failed under environmental variability and required significant manual tuning for new object types [4].

The rise of deep learning, particularly Convolutional Neural Networks (CNNs), caused a huge shift. CNNs learn hierarchical features directly from data, enabling significantly higher accuracy and greater flexibility [5]. These models outperform traditional pipelines in complex scenes, offering robustness to occlusion, background noise, and intra-class variation (challenges that traditional local descriptors struggle to handle) [6]. Among the most influential advances is the YOLO (You Only Look Once) family of models, which reframed object detection as a single-stage process. YOLO splits an image into a grid and predicts bounding boxes and class probabilities for all objects in one forward pass, enabling real-time detection with high accuracy. This efficiency and speed have made YOLO widely adopted for applications such as robotics, autonomous vehicles, and video surveillance [7]. The latest YOLOv7 model achieves an average precision

(AP) of up to 56.8% on the COCO benchmark, making it one of the top-performing real-time object detectors available today [8].

More recently, transformer-based vision models have begun to influence robotics research. Models such as the Swin Transformer leverage hierarchical attention to capture long-range spatial dependencies and have set new benchmarks on datasets like COCO (e.g., Swin achieves 58.7 AP for detection) [9].

Another emerging direction is self-supervised learning (SSL), which aims to reduce dependence on labeled datasets. This would be a particularly valuable shift in robotics since data collection is straightforward but annotation is expensive. Techniques like contrastive learning (e.g., SimCLR) and masked image modeling (e.g., MAE) pretrain models on unlabeled data, which can later be fine-tuned for recognition tasks with minimal supervision. While still developing, SSL holds strong potential for low-resource robotics contexts [10].

Despite these advances, deep models have limitations. They require significant computational resources, are often considered "black-boxes", and can be brittle when deployed in settings that differ from their training environments [5]. This lack of interpretability makes it difficult to anticipate how a robot will respond in uncertain situations, raising concerns in contexts involving human interaction or safety. Ongoing research focuses on domain adaptation, model compression, efficient architectures to improve robustness, interpretability, and deployability [10].

Conclusion

This project compared traditional and deep learning approaches to object recognition across two contrasting datasets. While the BoW pipeline was lightweight and interpretable, it struggled in complex, real-world scenes. In contrast, the CNN model generalized well and consistently outperformed BoW across all metrics. Applying transfer learning by using a pretrained ResNet reflects best practices in robotics as it achieves strong performance without large-scale training. Overall, CNNs proved to be the more robust and adaptable choice for modern robotic vision tasks.

References:

[1] L. Bo, X. Ren, and D. Fox, “Unsupervised Feature Learning for RGB-D Based Object Recognition,” pp. 387–402, Jan. 2013, doi: https://doi.org/10.1007/978-3-319-00065-7_27.

[2] S. R. e Zehra, A. Majid, M. S. Nizami, and S. Afridi, “Perception and Attention Applying Cognitive Psychology Principal to Improve AI Vision System,” *The Critical Review of Social Sciences Studies*, vol. 3, no. 1, pp. 3062–3077, Mar. 2025, doi: https://doi.org/10.59075/mwezn998.

[3] Valentyn Kropov, “Computer vision in robotics: The future of intelligent automation,” *Software Development Company - N-iX*, Aug. 16, 2024. https://www.n-ix.com/computer-vision-in-robotics/ (accessed May 09, 2025).

[4] C. Szegedy, A. Toshev, and Dumitru Erhan, “Deep Neural Networks for Object Detection,” *Advances in Neural Information Processing Systems*, pp. 1–9, Jan. 2013, Available: https://www.researchgate.net/publication/319770289_Deep_Neural_Networks_for_Object_Detection

[5] L. Alzubaidi *et al.*, “Review of deep learning: concepts, CNN architectures, challenges, applications, future directions,” *Journal of Big Data*, vol. 8, no. 1, Mar. 2021, doi: https://doi.org/10.1186/s40537-021-00444-8.

[6] N. O' Mahony *et al.*, “Deep Learning vs. Traditional Computer Vision,” 2019. Available: https://arxiv.org/pdf/1910.13796

[7] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, “You Only Look Once: Unified, Real-Time Object Detection,” *arXiv.org*, Jun. 08, 2015. https://arxiv.org/abs/1506.02640

[8] C.-Y. Wang, A. Bochkovskiy, and H.-Y. M. Liao, “YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors,” *arXiv:2207.02696 [cs]*, Jul. 2022, Available: https://arxiv.org/abs/2207.02696

[9] Z. Liu *et al.*, “Swin Transformer: Hierarchical Vision Transformer using Shifted Windows,” *arXiv:2103.14030 [cs]*, Mar. 2021, Available: https://arxiv.org/abs/2103.14030

[10] A. Ciocarlan, S. Lefebvre, S. L. Hégarat-Mascle, and A. Woiselle, “Self-Supervised Learning for Real-World Object Detection: a Survey,” *arXiv.org*, 2024. https://arxiv.org/abs/2410.07442 (accessed May 09, 2025).

Appendix

BAG OF WORDS CODE:

Dataset loading/preprocessing:

iCubWorld helpers –

```
def load_icub_from_drive():
    labels = []
    image_paths = []

    # Get all 'part*' folders (e.g., part1, part2, ...)
    dataset_path = '/content/drive/MyDrive/U of Manchester/Robotics Assignment/data/icubworld_data'

    part_folders = [f for f in os.listdir(dataset_path) if f.startswith('part')]

    for part in part_folders:
        part_path = os.path.join(dataset_path, part)

        # Loop over class folders (e.g., book, pencilcase, etc.)
        for class_name in os.listdir(part_path):
            class_path = os.path.join(part_path, class_name)
            if not os.path.isdir(class_path):
                continue

            # Loop over object instances (e.g., book1, book2, ...)
            for instance_name in os.listdir(class_path):
                instance_path = os.path.join(class_path, instance_name)

                if not os.path.isdir(instance_path):
                    continue

                # Only use 'MIX' transformation
                mix_path = os.path.join(instance_path, 'MIX')
                if not os.path.isdir(mix_path):
                    continue

                # Loop over days (e.g., day5, day6, ...)
                for day_name in os.listdir(mix_path):
                    day_path = os.path.join(mix_path, day_name)
                    if not os.path.isdir(day_path):
                        continue

                    # Use only left camera images
                    left_path = os.path.join(day_path, 'left')
                    if not os.path.isdir(left_path):
                        continue

                    # Loop through image files
                    for file in os.listdir(left_path):
                        if not file.lower().endswith(('.jpg', '.jpeg', '.png')):
                            continue

                        #now record label and image path for each image
                        img_path = os.path.join(left_path, file)
                        labels.append(class_name)  # use high-level class name
```

```
                    image_paths.append(img_path)
    return image_paths, labels
```

```
def sample_dataset(image_paths, labels, max_per_class=500, seed=42):
    #sample
    class_to_images = defaultdict(list)
    for path, label in zip(image_paths, labels):
        class_to_images[label].append(path)

    random.seed(seed)

    #get sample from each class
    sampled_image_paths = []
    sampled_labels = []

    for label, paths in class_to_images.items():
        random.shuffle(paths)
        selected = paths[:max_per_class]
        sampled_image_paths.extend(selected)
        sampled_labels.extend([label] * len(selected))

    # shuffle data -- avoid order bias
    combined = list(zip(sampled_image_paths, sampled_labels))
    random.shuffle(combined)
    sampled_image_paths, sampled_labels = zip(*combined)

    return list(sampled_image_paths), list(sampled_labels)
```

COCO helpers–

```
def build_image_label_map(coco, top_k=20, target_count=8000):
    img_ids = coco.getImgIds()
    temp_labels = {}
    cat_count = Counter()

    # most common object per image - no filtering yet
    for img_id in img_ids:
        anns = coco.loadAnns(coco.getAnnIds(imgIds=img_id))
        if not anns:
            continue
        cat_ids = [ann['category_id'] for ann in anns]
        most_common_cat = Counter(cat_ids).most_common(1)[0][0]
        fname = coco.loadImgs(img_id)[0]['file_name']
        temp_labels[fname] = most_common_cat
        cat_count[most_common_cat] += 1

    #get only the top k categories
    top_cats = set([cat for cat, _ in cat_count.most_common(top_k)])
    label_to_index = {cat_id: idx for idx, cat_id in enumerate(sorted(top_cats))}

    # get images from top_k categories only until target_count is reached
    image_label_map = {}
```

```
    class_image_counts = defaultdict(int)

    for fname, cat_id in temp_labels.items():
        if cat_id in top_cats:
            image_label_map[fname] = label_to_index[cat_id]
            class_image_counts[cat_id] += 1
            if len(image_label_map) >= target_count:
                break

    return image_label_map, label_to_index
def paths_and_labels(image_label_map, img_dir):
    paths = []
    labels = []
    for fname, label in image_label_map.items():
        path = os.path.join(img_dir, fname)
        if os.path.exists(path):
            paths.append(path)
            labels.append(label)
    return paths, labels
```

Feature extractors –

```
#CHOICE 1: SIFT
def get_sift_descriptors(image_paths, resize_to=(256, 256)):
    sift = cv2.SIFT_create()
    img_descriptors = []
    all_descriptors = []

    for path in tqdm(image_paths, desc="Extracting SIFT descriptors"):
        img = cv2.imread(path, cv2.IMREAD_GRAYSCALE)
        if resize_to:
            img = cv2.resize(img, resize_to)

        #this finds DoG interest point (sparse)
        keypoints, descriptors = sift.detectAndCompute(img, None)

        if descriptors is not None:
            img_descriptors.append(descriptors)
            all_descriptors.extend(descriptors)
        else:
            img_descriptors.append(np.array([]))

    return all_descriptors, img_descriptors
#CHOICE 2: ORB
def get_orb_descriptors(image_paths, resize_to=(256, 256), max_features=1500):
    orb = cv2.ORB_create(nfeatures=max_features)
    img_descriptors = []
    all_descriptors = []

    for path in tqdm(image_paths, desc="Extracting ORB descriptors"):
        img = cv2.imread(path, cv2.IMREAD_GRAYSCALE)
        if resize_to:
            img = cv2.resize(img, resize_to)
```

```
        keypoints, descriptors = orb.detectAndCompute(img, None)

        if descriptors is not None:
            img_descriptors.append(descriptors)
            all_descriptors.extend(descriptors)
        else:
            img_descriptors.append(np.array([]))

    return all_descriptors, img_descriptors
```

Clustering algorithm helpers–

```
#CHOICE 1: KMEANS
def build_histograms_kmeans(descriptors, kmeans, k):
    #initalize hist
    hist = np.zeros(k)
    if descriptors is not None and len(descriptors) > 0:
        words_in_image = kmeans.predict(descriptors)
        #update histogram (word is an id)
        for word in words_in_image:
            hist[word] += 1
    return hist
#CHOICE 2: GMM
def build_histograms_gmm(descriptors, gmm, k):
    hist = np.zeros(k)
    if descriptors is not None and len(descriptors) > 0:
        words_in_image = gmm.predict(descriptors)
        for word in words_in_image:
            hist[word] += 1
    return hist
```

Image classifier helpers –

```
def evaluate_classifier(X_train, y_train, X_val, y_val, classifier='svm'):

    if classifier == 'svm':
        model = SVC(kernel='linear', random_state=42)
    elif classifier == 'nb':
        model = GaussianNB()
    else:
        raise ValueError("Unsupported classifier.")

    model.fit(X_train, y_train)
    y_pred = model.predict(X_val)

    accuracy = accuracy_score(y_val, y_pred)
    precision = precision_score(y_val, y_pred, average='macro', zero_division=0)
    recall = recall_score(y_val, y_pred, average='macro', zero_division=0)
    f1 = f1_score(y_val, y_pred, average='macro', zero_division=0)

    print(f"{classifier.upper()} Results:")
    print(f"  Accuracy : {accuracy:.4f}")
    print(f"  Precision: {precision:.4f}")
    print(f"  Recall   : {recall:.4f}")
    print(f"  F1-score : {f1:.4f}")
```

```
   return model, {'accuracy': accuracy, 'precision': precision, 'recall':
recall, 'f1': f1}
```

Experiment code (runs 24 experiments for each dataset)--

```
vocab_sizes = [100, 250, 500]
classifiers = ['svm', 'nb']
results = []
cluster_methods = ['kmeans','gmm']
feature_methods = ['sift','orb']
data = 'coco' ##SWITCH DATASET HERE

if data == 'icubworld':
   #LOAD icubworld data
   image_paths, labels = load_icub_from_drive()
   sampled_image_paths, sampled_labels = sample_dataset(image_paths, labels, max_per_class=500)

   #label encoder helps map between labels and unique IDs -- easy to convert back and forth
   label_encoder = LabelEncoder()
   encoded_labels = label_encoder.fit_transform(sampled_labels)

   #split
   image_paths_trainval, image_paths_test, y_trainval, y_test = train_test_split(
   sampled_image_paths, sampled_labels, test_size=0.1, stratify=sampled_labels, random_state=42
   )
   image_paths_train, image_paths_val, y_train, y_val = train_test_split(
       image_paths_trainval, y_trainval, test_size=0.2, stratify=y_trainval, random_state=42
   )

elif data == 'coco':
   annotation_path_val = '/content/drive/MyDrive/U of Manchester/Robotics
Assignment/data/coco/annotations/instances_val2017.json'
   annotation_path_train = '/content/drive/MyDrive/U of Manchester/Robotics
Assignment/data/coco/annotations/instances_train2017.json'
   coco_val = COCO(annotation_path_val)
   coco_train = COCO(annotation_path_train)

   #coco only has train and val
   train_image_label_map, train_label_to_index = build_image_label_map(coco_train,
target_count=8000)
   val_image_label_map,   val_label_to_index   = build_image_label_map(coco_val,
target_count=2000)

   all_labels = set(train_image_label_map.values()) | set(val_image_label_map.values())
   label_encoder = LabelEncoder()
   label_encoder.fit(list(sorted(all_labels)))

   local_train_path = "/content/train2017"
   local_val_path   = "/content/val2017"

   # unzip
   if not os.path.exists(local_train_path):
```

```
        print("Unzipping training images...")
        !unzip -q "/content/drive/MyDrive/U of Manchester/Robotics
Assignment/data/train2017.zip" -d /content/
    if not os.path.exists(local_val_path):
        print("Unzipping validation images...")
        !unzip -q "/content/drive/MyDrive/U of Manchester/Robotics Assignment/data/val2017.zip"
-d /content/

    image_paths_train, y_train = paths_and_labels(train_image_label_map, local_train_path)
    image_paths_val, y_val = paths_and_labels(val_image_label_map, local_val_path)

    # encode labels
    train_labels = label_encoder.transform(train_labels)
    val_labels  = label_encoder.transform(val_labels)

print("Dataset: ", data)
print("Train images:", len(image_paths_train))
print("Val images:  ", len(image_paths_val))

#note no test labels are released for coco
if data == 'icubworld':
 print("Test images: ", len(image_paths_test))

for feature_method in feature_methods:
    # extract descriptors
    if feature_method == 'sift':
        all_descriptors_train, image_descriptors_train = get_sift_descriptors(image_paths_train)
        _, image_descriptors_val = get_sift_descriptors(image_paths_val)

    elif feature_method == 'orb':
        all_descriptors_train, image_descriptors_train = get_orb_descriptors(image_paths_train)
        _, image_descriptors_val = get_orb_descriptors(image_paths_val)

    all_descriptors_train = np.array(all_descriptors_train, dtype=np.float32)
     for clustering_alg in cluster_methods:
      #one run for each vocab size
      for k in vocab_sizes:
          if clustering_alg == 'kmeans':
              kmeans = MiniBatchKMeans(n_clusters=k, random_state=42, batch_size=1000,
verbose=1)
              kmeans.fit(all_descriptors_train.astype(np.float32))
              X = [] #each row is histogram of visual words for one image

              X_train = [build_histograms_kmeans(d.astype(np.float32), kmeans, k) if len(d) > 0
else np.zeros(k)
                         for d in image_descriptors_train]
              X_val = [build_histograms_kmeans(d.astype(np.float32), kmeans, k) if len(d) > 0
else np.zeros(k)
                       for d in image_descriptors_val]
          elif clustering_alg == 'gmm':
              #take samle of descriptors bc colab keeps crashing
              sample_size = 100000
              if len(all_descriptors_train) > sample_size:
```

```
                indices = np.random.choice(len(all_descriptors_train), sample_size,
replace=False)
                sampled_descriptors = all_descriptors_train[indices]
            else:
                sampled_descriptors = all_descriptors_train

            #add pca to reduce dimensionality
            pca = PCA(n_components=16 if feature_method == 'orb' else 64, random_state=42)
#why is this the case?
            pca.fit(sampled_descriptors)
            pca_all_descriptors = pca.transform(sampled_descriptors)

            gmm = GaussianMixture(n_components=k, random_state=42, verbose=1,
covariance_type='diag')
            gmm.fit(pca_all_descriptors)

            X_train = []
            for d in image_descriptors_train:
                if d is None or len(d) == 0:
                    X_train.append(np.zeros(k))
                else:
                    d_pca = pca.transform(d.astype(np.float32))
                    X_train.append(build_histograms_gmm(d_pca, gmm, k))
            X_val = []
            for d in image_descriptors_val:
                if d is None or len(d) == 0:
                    X_val.append(np.zeros(k))
                else:
                    d_pca = pca.transform(d.astype(np.float32))
                    X_val.append(build_histograms_gmm(d_pca, gmm, k))

        for classifier in classifiers:
            model, metrics = evaluate_classifier(X_train, y_train, X_val, y_val,
classifier=classifier)
            results.append({
                  'feature': feature_method,
                  'clustering': clustering_alg,
                  'vocab_size': k,
                  'classifier': classifier,
                  **metrics
              })
            print(f"[{feature_method.upper()} | {clustering_alg.upper()} | k={k} |
{classifier.upper()}] "
                    f"Accuracy: {metrics['accuracy']:.4f}, F1: {metrics['f1']:.4f}")
            #reduce memory
            del model, metrics
            gc.collect()
        #CLEANUP MEMORY -- colab is crashing from high ram
        del X_train, X_val
        if clustering_alg == 'kmeans':
            del kmeans
        elif clustering_alg == 'gmm':
            del gmm, pca
```

```
        gc.collect()

df = pd.DataFrame(results)
df.to_csv(f"{feature_method}_experiments_results.csv", index=False)
print("\nsaved results!'")
```

Choose best model and evaluate on test set:

```
##choose run with the highest f1
best_idx = df['f1'].idxmax()
best_config = df.loc[best_idx]
print("Best on val:", best_config.to_dict())
#test the best model

#get descriptors on test set
if feature_method == 'sift':
   _, image_descriptors_test = get_sift_descriptors(image_paths_test)
else:  # orb
   _, image_descriptors_test = get_orb_descriptors(image_paths_test)

#get best hyperparams
feat   = best_config['feature']
clust  = best_config['clustering']
k      = int(best_config['vocab_size'])
clf_nm = best_config['classifier']

# merge train and val and get descriptors
paths_tv  = image_paths_train + image_paths_val
labels_tv = y_train + y_val

if feat == 'sift':
   all_desc_tv, descs_tv = get_sift_descriptors(paths_tv)
else:
   all_desc_tv, descs_tv = get_orb_descriptors(paths_tv)

all_desc_tv = np.array(all_desc_tv, dtype=np.float32)

# cluster to get codebook
if clust == 'kmeans':
   clust_model = MiniBatchKMeans(n_clusters=k, random_state=42, batch_size=1000)
   clust_model.fit(all_desc_tv)
else:  # GMM + PCA
   sample_size = 100000
   if len(all_desc_tv) > sample_size:
       indices = np.random.choice(len(all_desc_tv), sample_size, replace=False)
       sampled_descriptors = all_desc_tv[indices]
   else:
       sampled_descriptors = all_desc_tv
   pca = PCA(n_components=64 if feat=='sift' else 16, random_state=42)
   pca_all = pca.fit_transform(sampled_descriptors)
```

```
    clust_model = GaussianMixture(n_components=k, covariance_type='diag',
random_state=42)
    clust_model.fit(pca_all)

def make_hist(d):
return build_histograms_kmeans(d.astype(np.float32), clust_model, k) if len(d)>0 else
np.zeros(k)

def make_hist(d):
    if d is None or len(d)==0:
        return np.zeros(k)
    d_pca = pca.transform(d.astype(np.float32))
    return build_histograms_gmm(d_pca, clust_model, k)

# feature matrices for train+val and test
X_tv   = np.vstack([make_hist(d) for d in descs_tv])
X_test = np.vstack([make_hist(d) for d in image_descriptors_test])

#train classifier and evaluate on test
if clf_nm == 'svm':
    final_clf = SVC(random_state=42)
elif clf_nm == 'nb':
    final_clf = GaussianNB()

final_clf.fit(X_tv, labels_tv)
y_pred = final_clf.predict(X_test)

from sklearn.metrics import accuracy_score, f1_score
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

test_acc = accuracy_score(y_test, y_pred)
test_f1  = f1_score(y_test, y_pred, average='weighted')

cm = confusion_matrix(y_test, y_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm)
disp.plot(xticks_rotation='vertical')
plt.title("Confusion Matrix")
plt.show()
print(f"\nFINAL TEST- Acc: {test_acc:.4f}, F1: {test_f1:.4f}")
precision = precision_score(y_test, y_pred, average='weighted')
print(f"Precision: {precision:.4f}")
recall = recall_score(y_test, y_pred, average='weighted')
print(f"Recall: {recall:.4f}")

precision_micro = precision_score(y_test, y_pred, average='micro', zero_division=0)
recall_micro = recall_score(y_test, y_pred, average='micro', zero_division=0)
print(f"Micro-averaged Precision: {precision_micro:.4f}")
print(f"Micro-averaged Recall: {recall_micro:.4f}")
```

CNN code:

Preprocessing:

```
#make dataset class to feed data into pytorch
class ImageDataset(Dataset):
    def __init__(self, image_paths, labels, transform=None):
        self.image_paths = image_paths
        self.labels = labels
        self.transform = transform

    def __len__(self):
        return len(self.image_paths)

    def __getitem__(self, idx):
        image = Image.open(self.image_paths[idx]).convert("RGB")
        label = self.labels[idx]
        if self.transform:
            image = self.transform(image)
        return image, label
```

Load pretrained model:

```
def build_model(num_classes, dropout=.5):
    #load in ResNet-18
    model = models.resnet18(pretrained=True)

    # diable gradient updates for weights -- only want train the classifier part at the nd
    for param in model.parameters():
        param.requires_grad = False

    #replace the classifier layer (fully connected linear layer), add dropout
    model.fc = nn.Sequential(
        nn.Dropout(dropout),
        nn.Linear(model.fc.in_features, num_classes)
    )

    for name, param in model.named_parameters():
        if "layer4" in name or "fc" in name:
            param.requires_grad = True

    return model.to(device)
```

Training:

```
def train_model(model, train_loader, val_loader, optimizer, scheduler, criterion,
num_epochs=10, dropout=None, lr=None, optimizer_name=None, augment=None):
    #track best f1 score
    best_f1 = 0.0
    #starts new epoch
    for epoch in range(num_epochs):
        model.train() #TRAINING MODE
        running_loss, correct, total = 0.0, 0, 0
        #loop over batches
        for images, labels in tqdm(train_loader,
                                   desc=f"Epoch {epoch+1}/{num_epochs}",
                                   leave=False):
            images, labels = images.to(device), labels.to(device)
```

```
            optimizer.zero_grad() #reset gradient from last batch
            outputs = model(images) #forward pass
            loss = criterion(outputs, labels)
            loss.backward()#backward pass
            optimizer.step() #update weights

            running_loss += loss.item() #get total loss over all batches
            _, predicted = outputs.max(1) #get predicted class from softmax
            total += labels.size(0)
            correct += predicted.eq(labels).sum().item() #get how many predicts are correct

        #decide learning rate
        scheduler.step()
        #evaluate how model is learning
        val_acc, val_f1, _, _ = evaluate_model(model, val_loader)
        if val_f1 > best_f1:
            best_f1 = val_f1
            #save the best model for later
            torch.save(model.state_dict(),
f"best_model_dropout{dropout}_lr{lr}_{optimizer_name}_aug{augment}.pth")
            print(f"new best model with F1 = {val_f1:.4f}")
        print(f"Epoch {epoch+1}/{num_epochs}, Train Acc: {100*correct/total:.2f}%, Val F1:
{val_f1:.4f}")
    return best_f1
def evaluate_model(model, data_loader):
    model.eval()
    all_preds, all_labels = [], []
    with torch.no_grad(): #disable gradient tracking
        #load through validation dataset
        for images, labels in data_loader:
            images, labels = images.to(device), labels.to(device)
            #forward pass
            outputs = model(images)
            #compute validation accuracy
            _, preds = outputs.max(1)
            all_preds.extend(preds.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())
    acc = np.mean(np.array(all_preds) == np.array(all_labels)) * 100
    f1 = f1_score(all_labels, all_preds, average='macro')
    return acc, f1, all_labels, all_preds
```

Experiment code helper:

```
def run_experiment(dropout, lr, optimizer_name, augment=False, num_epochs=10):
    if augment:
        train_transform = transforms.Compose([
            transforms.Resize((224, 224)),
            transforms.RandomHorizontalFlip(),
            transforms.ColorJitter(0.2, 0.2, 0.2),
            transforms.ToTensor(),
            transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
        ])
        #validation set should not be augmented
        val_transform = transforms.Compose([
```

```
        transforms.Resize((224, 224)),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])
    else:
        #resize and normalize data (basically to match pretrained model)----
        #must resize data to (224, 224) bc pretrained CNNS were trained on ImageNet (where all
images were 224x224 pixel)
        #normalize to speed up convergence during training
        #use mean and std from ImageNet to match what network was trained on
        train_transform = transforms.Compose([
            transforms.Resize((224, 224)),
            transforms.ToTensor(),
            transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
        ])
        #validation and training set need to have the same transformation
        val_transform = train_transform

    train_dataset = ImageDataset(train_paths, train_labels, train_transform)
    val_dataset = ImageDataset(val_paths, val_labels, val_transform)

    #create dataloaders -- helps feed batches into model
    train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True, num_workers=4,
pin_memory=True)
    val_loader = DataLoader(val_dataset, batch_size=32, shuffle=False, num_workers=4,
pin_memory=True)

    model = build_model(num_classes=len(label_encoder.classes_), dropout=dropout)

    #experiment with different optimizers
    trainable_params = filter(lambda p: p.requires_grad, model.parameters())

    if optimizer_name.lower() == 'adam':
        optimizer = optim.Adam(trainable_params, lr=lr)
    elif optimizer_name.lower() == 'sgd':
        optimizer = optim.SGD(trainable_params, lr=lr, momentum=0.9)
    #experiment with different learning rates
    scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=5, gamma=0.1)
    #use cross entropy loss function -- why?
    criterion = nn.CrossEntropyLoss()

    best_val_f1 = train_model(
        model, train_loader, val_loader, optimizer, scheduler, criterion, num_epochs,
        dropout=dropout, lr=lr, optimizer_name=optimizer_name, augment=augment
    )
    return {
        "dropout": dropout,
        "lr": lr,
        "optimizer": optimizer_name,
        "augment": augment,
        "val_f1": best_val_f1,
        "model": model,
        "criterion": criterion
```

```
  }
```

Run experiments

```
# - Dropout rate: [0.0, 0.3, 0.5] → to see how regularization affects
overfitting
# - Learning rate: [1e-3, 1e-4] → to compare fast vs slow learning
# - Optimizer: ['adam', 'sgd'] → to test adaptive vs momentum-based
optimization
# - Data augmentation: [False, True] → to see if transforms help
generalization
# In total: 3 x 2 x 2 x 2 = 24 experiments

results = []
##CHANGE THIS
data = 'icub'

if data == 'icub':
   dataset_path = "/content/drive/MyDrive/U of Manchester/Robotics
Assignment/data/icubworld_data"
   #get the data
   sampled_image_paths, sampled_labels = load_icub_data(dataset_path,
max_per_class=500)
   label_encoder = LabelEncoder()
   encoded_labels = label_encoder.fit_transform(sampled_labels)

   #split into training, validation, testing
   trainval_paths, test_paths, trainval_labels, test_labels =
train_test_split(
       sampled_image_paths, encoded_labels, test_size=0.1,
stratify=encoded_labels, random_state=42)
   train_paths, val_paths, train_labels, val_labels = train_test_split(
       trainval_paths, trainval_labels, test_size=0.2,
stratify=trainval_labels, random_state=42)

elif data == 'coco':
   annotation_path_val = '/content/drive/MyDrive/U of Manchester/Robotics
Assignment/data/coco/annotations/instances_val2017.json'
   annotation_path_train = '/content/drive/MyDrive/U of Manchester/Robotics
Assignment/data/coco/annotations/instances_train2017.json'
   coco_val = COCO(annotation_path_val)
   coco_train = COCO(annotation_path_train)
```

```
   #coco only has train and val
   train_image_label_map, train_label_to_index =
build_image_label_map(coco_train, target_count=8000)
   val_image_label_map,   val_label_to_index   =
build_image_label_map(coco_val, target_count=2000)

   all_labels = set(train_image_label_map.values()) |
set(val_image_label_map.values())
   label_encoder = LabelEncoder()
   label_encoder.fit(list(sorted(all_labels)))

   local_train_path = "/content/train2017"
   local_val_path   = "/content/val2017"

   #unzip
   if not os.path.exists(local_train_path):
       print("Unzipping training images...")
       !unzip -q "/content/drive/MyDrive/U of Manchester/Robotics
Assignment/data/train2017.zip" -d /content/
   if not os.path.exists(local_val_path):
       print("Unzipping validation images...")
       !unzip -q "/content/drive/MyDrive/U of Manchester/Robotics
Assignment/data/val2017.zip" -d /content/

   train_paths, train_labels = paths_and_labels(train_image_label_map,
local_train_path)
   val_paths, val_labels     = paths_and_labels(val_image_label_map,
local_val_path)

   # encode them
   train_labels = label_encoder.transform(train_labels)
   val_labels  = label_encoder.transform(val_labels)

#debug
print("Train images:", len(train_paths))
print("Val images:  ", len(val_paths))
if data == 'icub':
   print("Test images: ", len(test_paths))
#RUN the experiments
for dropout in [0.0, 0.3, 0.5]:
   for lr in [1e-3, 1e-4]:
       for opt in ['adam', 'sgd']:
           for augment in [False, True]:
```

```
                print(f"Running: dropout={dropout}, lr={lr}, optimizer={opt},
augment={augment}")
                result = run_experiment(dropout, lr, opt, augment)
                results.append(result)
#save
df = pd.DataFrame(results)
df.to_csv("experiment_results.csv", index=False)
df
```