
Politecnico di Torino

POLITECNICO DI TORINO

College of Computer Engineering, Cinema and Mechatronics

Master's Degree Thesis

# Similarity of Waste Image for Smart Bins Using Deep Learning

**Supervisors**
prof. Bartolomeo Montrucchio
dr. Antonio Costantino Marceddu

**Candidate**
Hossein Zahedi Nezhad

July 2025

# Abstract

This dissertation presents a deep learning-based framework for object-level change detection in cluttered visual scenes, focusing on identifying added or reconfigured items between temporally adjacent images. The motivation arises from real-world challenges in automated waste monitoring systems, where detecting changes in bin contents is critical for optimizing collection routes, improving recycling efficiency, and reducing operational costs. To address this need, two approaches were explored. Initially, a Siamese network trained with contrastive loss was implemented to evaluate the feasibility of image-level change detection based on pairwise similarity. While this approach demonstrated potential, it was limited to producing a single similarity score between images, without the ability to localize individual changes or determine the number of added objects. These limitations motivated the transition to a more expressive object-level triplet learning framework, where the network compares anchor–positive–negative tuples. This design enables fine-grained matching, robust feature discrimination, and estimation of added object count. In the triplet-based framework, the system combines polygon-guided object cropping, deep metric learning (i.e., training neural networks to learn a similarity-preserving embedding space), and cosine similarity analysis to compare object instances across image pairs. A COCO-style (Common Objects in Context) annotated dataset of 7,150 real-world waste bin images was used, capturing cluttered scenes with items such as cups, packaging, and organic waste. Objects were extracted using segmentation masks and polygon-fitting, then standardized into 224×224 crops. Four popular convolutional neural networks—ResNet-50, ResNet-101, MobileNetV2, and Xception—were repurposed from their original classification role to serve as backbone feature extractors. Each model was modified with a custom projection head that maps high-dimensional features into a 128-dimensional embedding space using global average pooling, dropout, a dense layer, and L2 normalization. This transformation enables angular similarity comparisons via cosine distance, making the architecture suitable for fine-grained object matching. A progressive training strategy was employed to enhance embedding quality and assess the impact of adaptation. In the zero-shot configuration, the backbone remained fully frozen to evaluate how well pre-trained features generalize to the object matching task. In Phase 1, only the projection head was trained while the backbone remained frozen, allowing rapid convergence and preserving general visual priors. In Phase 2, selective fine-tuning of deeper backbone layers was performed while continuing to train the projection head. Training relied on a margin-based triplet loss with a custom mining pipeline incorporating hard positive mining, hard and semi-hard negative mining, and stage-specific augmentations. Strong geometric and photometric transformations were applied to anchors and positives, while negatives were minimally perturbed. Model comparison and performance assessment across all architectures highlight the strengths and trade-offs of each model, confirming that the proposed approach enables accurate detection of added objects and reliable matching of existing instances across image pairs. The system is scalable, robust, and well-suited for integration into real-time waste monitoring systems.

# Acknowledgements

First and foremost, I would like to express my sincere gratitude to Prof. Bartolomeo Montrucchio and Dr. Antonio Costantino Marceddu for their guidance throughout the development of this thesis. In particular, I am deeply thankful to Dr. Marceddu, who consistently provided useful suggestions and invaluable support during the course of this project.

I would also like to extend my thanks to Federico Fedi, Mattia Brusamento and Simone Cavariani from NANDO (ReLearn) for their early support and for facilitating key resources that contributed to the successful completion of this work.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Background and Motivation

Municipal solid-waste (MSW) volumes are climbing faster than population growth. The World Bank's *What a Waste 2.0* estimates that the planet generated 2.24 billion tons in 2020 and will reach 3.88 billion tons per year by 2050 if current consumption patterns persist [1]. *UNEP's Global Waste-Management Outlook 2024* confirms a similar trajectory, estimating 2.1 billion tons of waste generated in 2023 and projecting an increase to 3.8 billion tons by 2050. The current direct cost of waste management services is estimated at US$252 billion per year. When externalities such as health care, environmental pollution, and climate-related impacts are accounted for, the total annual cost is projected to rise to US$640 billion by mid-century [2].

For city treasuries, the biggest drain is the collection truck. Routine collection and disposal already swallow 20–50% of the entire municipal budget in many low- and middle-income jurisdictions [3], and field audits show that collection alone can account for 50–80% of all solid-waste line items [4]. Inefficiencies amplify those costs: each needless revisit to a half-empty bin burns diesel (approx. 2.3 kg $CO_2$ eq/km for a compactor truck) [5], while overfilled or mis-sorted containers are now subject to contamination surcharges—up to US$25–150 per incident in California's Oro Loma district, automatically triggered by on-truck cameras [6].

To curb the haemorrhage of labour hours, fuel and tipping fees, cities and private haulers are turning to "smart-bin" systems that retrofit containers with embedded sensors and connectivity. Commercial platforms such as Sensoneo or Nordsense install ultrasonic or radar probes that ping the fill-level every 15–60 minutes and stream the data to cloud-based route-optimisation dashboards [7]. The promise is compelling: leaner truck fleets, pay-as-you-throw billing, and near-real-time feedback to residents and building managers.

However, a crucial blind spot remains. Almost all deployed smart-bin sensors reduce the container to a single coarse metric—"% full". Camera systems fare only slightly better, typically classifying the entire frame as "empty", "half" or "full" or computing one global similarity score between two snapshots. A 2024 survey of more than one hundred IoT-dustbin papers concludes that the overwhelming majority stop at volume estimation or single-label classification, with almost no work tackling object-level localisation of new items between visits [8]. Consequently, operators still lack answers to three mission-critical questions:

1. What was deposited since the last collection?
2. How many discrete items are new?
3. How many and which items remained unchanged between collections?

Solving that triad is not mere academic curiosity; it is the key to postponing unnecessary pickups, prioritising high-risk loads (food waste, e-waste, sharps), and issuing contamination

warnings before the truck rolls. This thesis therefore frames the problem as an object-level change-detection challenge and proposes a deep metric-learning solution that can track individual waste items across successive images and flag newly added objects in cluttered, low-light conditions.

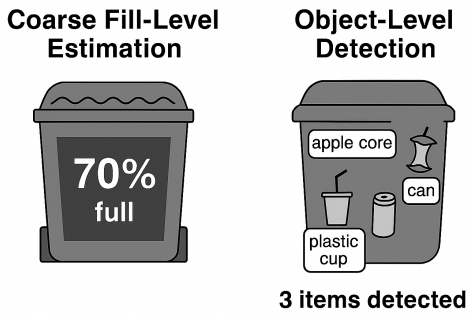



Figure 1.1: Traditional smart-bin systems output a coarse "percentage full" value (left), which provides no insight into item types or counts. In contrast, the proposed approach enables object-level detection, identifying individual waste items and tracking which ones are new or unchanged between collections.

## 1.2 Problem Statement

Despite advances in smart-bin technologies and object-detection pipelines, existing systems fundamentally lack the capacity to perform object-level reasoning across successive images taken from inside a bin. Most commercially deployed solutions rely on fill-level estimation (e.g., percentage fullness from ultrasonic sensors) or produce a single global similarity score between images, offering no insight into how many individual items have been added between visits. These approaches cannot differentiate between small rearrangements and meaningful content changes, nor do they provide structured outputs that enable selective tracking or counting of newly deposited objects.

This lack of object-level change detection imposes critical limitations on downstream operations. Without fine-grained information about bin contents, collection routes cannot be optimised dynamically, contamination cannot be flagged early, and behaviour-based feedback (e.g., for recycling compliance or pay-as-you-throw billing) remains crude or infeasible.

From a technical perspective, the core challenge lies in the inability of current visual models to generate discriminative, object-level embeddings that remain robust across cluttered scenes, viewpoint variation, partial occlusion, and diverse material textures. Preliminary experiments with a Siamese network trained using contrastive loss support this diagnosis: while the model was able to distinguish similar and dissimilar image pairs in aggregate, it lacked the resolution to localise individual changes or count newly added objects. These limitations were especially evident in scenes involving overlapping items, transparent containers, or reflective surfaces such as foil and glass.

Together, these observations motivate the development of a fine-grained, embedding-based approach that can detect and track individual waste items across image pairs. The proposed solution must generalize across object categories while remaining sensitive to instance-level differences and capable of capturing subtle, temporally localised changes in real-world bin environments.

This research was conducted in collaboration with NANDO (ReLearn), a technology company specializing in AI-based waste management solutions. As part of this collaboration, NANDO provided a proprietary dataset of annotated waste-bin images captured in real-world environments. These data served as the foundation for all experimental evaluations in this thesis and reflect practical challenges encountered in operational smart-bin systems.

## 1.3 Aim and Objectives

The central aim of this thesis is to develop a deep metric-learning framework capable of matching individual waste objects across temporally spaced bin images and accurately detecting newly added items under real-world conditions such as clutter, occlusion, and lighting variation.

To support this aim, the following research objectives were pursued:

1. **Dataset Utilization:** Employ a COCO-style annotated dataset of real-world waste-bin images—provided by NANDO (ReLearn)—containing instance-level segmentation masks, which serve as the basis for generating object crops and computing temporal object-level changes.
2. **Baseline Implementation:** Design and evaluate a Siamese neural network trained with contrastive loss as an initial baseline, assessing its performance and limitations in binary object-matching across successive image pairs.
3. **Triplet-Loss Pipeline:** Develop an advanced triplet-based embedding framework with custom hard and semi-hard mining strategies, optimized to compute 128-dimensional embeddings that support instance-level comparison and identification across complex visual conditions.
4. **Model Benchmarking:** Benchmark four CNN backbones—ResNet-50, ResNet-101, MobileNetV2, and Xception—under three training strategies: (i) zero-shot inference with frozen backbones, (ii) projection-head fine-tuning, and (iii) partial fine-tuning of deeper layers, to identify the optimal accuracy–efficiency trade-off for practical deployment.

## 1.4 Thesis Structure

This thesis is organized into five chapters, as follows:

- **Chapter 1 – Introduction:** Presents the background, motivation, problem statement, dataset provider, and key research objectives that guide this study.
- **Chapter 2 – Background and Models:** Reviews prior work on image similarity, Siamese and Contrastive loss, the evolution of convolutional neural networks, and the CNN model architectures explored in this thesis.
- **Chapter 3 – Methodology:** Describes the dataset, object-extraction pipeline, similarity-learning framework, triplet sampling strategies, fine-tuning steps, augmentation settings, inference visualisation, and training procedures.
- **Chapter 4 – Results and Evaluation:** Presents experimental findings, including evaluation metrics, backbone benchmarking, and analysis of added-object detection and object-matching performance.
- **Chapter 5 – Conclusion:** Summarizes key contributions and results, discusses limitations, and outlines possible extensions for future research and deployment.

# Chapter 2

# Background and Models

## 2.1 Metric Learning for Visual Similarity

Retrieving visually similar objects from a large image collection requires more than assigning category labels: it demands a continuous embedding space in which the distance between any two object crops faithfully represents their perceptual resemblance. Off-the-shelf convolutional neural networks trained for classification optimize their penultimate activations to separate discrete classes, not to rank instances by similarity. As a result, two objects of the same category may lie far apart in feature-space if they differ in color, texture, or pose, while visually distinct instances of different labels can inadvertently cluster together.

Metric learning reframes this task by directly optimizing an embedding function so that similar pairs are drawn closer and dissimilar pairs pushed farther apart. However, as we will see, the limited capacity and training dynamics of a pairwise Siamese setup proved insufficient for our large-scale, fine-grained retrieval task. Consequently, we complement that baseline with higher-capacity CNN backbones—ResNet-50/101, Xception, and MobileNetV2—trained via the triplet protocol detailed in Chapter 3.10.1.

In the remainder of this chapter, we first survey prior art in image similarity (Section 2.2), then detail our initial Siamese approach with contrastive loss (Section 2.3), and finally describe the CNN-based embedding backbones (Section 2.5).

## 2.2 Related Work on Image Similarity

The problem of retrieving visually similar images has been addressed through a progression of techniques, from hand-crafted descriptors to deep neural embeddings trained with metric objectives. Below we survey seminal and recent contributions in five categories.

### 2.2.1 Classical Feature Descriptors

Lowe's Scale-Invariant Feature Transform (SIFT) introduced local keypoints with orientation and scale normalization, enabling robust matching under viewpoint and illumination changes; it remains a cornerstone for early image retrieval systems [9]. Bay et al. proposed Speeded-Up Robust Features (SURF), accelerating descriptor computation via integral images and approximated Hessian detectors [10]. Mikolajczyk and Schmid further evaluated these descriptors on large benchmarks, highlighting trade-offs in repeatability and distinctiveness [11].

### 2.2.2 Early CNN-Based Feature Extractors

Razavian et al. showed that off-the-shelf CNN features from networks pre-trained on ImageNet transfer effectively to retrieval tasks, outperforming hand-crafted descriptors on multiple datasets [12].

Babenko et al. introduced R-MAC pooling to aggregate convolutional activations into a compact global descriptor, significantly boosting retrieval accuracy with minimal dimensionality [13]. Gordo et al. advanced this by learning the pooling regions end-to-end, further improving landmark retrieval performance [14].

### 2.2.3 Metric-Learning Paradigms

Early metric-learning frameworks sought to shape the embedding space directly. Hadsell et al. formalized the contrastive loss, penalizing similar pairs for large distances and dissimilar pairs for distances below a margin, thus enforcing pairwise constraints in the learned space [15]. Chopra et al. applied this within a Siamese CNN for verification tasks, demonstrating its efficacy on signature and face data [16]. Weinberger and Saul proposed Large Margin Nearest Neighbor (LMNN), optimizing Mahalanobis distances with a convex objective to improve k-NN classification [17]. Schroff et al.'s FaceNet introduced triplet loss with online hard-example mining, achieving state-of-the-art face recognition by directly optimizing relative distances among anchor, positive, and negative samples [18].

### 2.2.4 Recent Siamese and Pairwise Variants

Koch et al. adapted Siamese networks for one-shot learning, training on diverse image classes so the model learns a generic similarity function that generalizes to unseen categories [19]. Bertinetto et al. developed SiamFC, a fully-convolutional Siamese tracker that computes cross-correlation between exemplar and search region feature maps for real-time object tracking [20]. Sohn proposed improved pair mining strategies with multi-similarity loss, which dynamically weights positive and negative pairs to stabilize convergence [21]. More recently, Ge et al. introduced Circle Loss, unifying angle and cosine margin constraints to boost both intra-class compactness and inter-class separability in embedding learning [22].

### 2.2.5 Hybrid and Attention-Based Approaches

With the rise of attention mechanisms, Li et al. incorporated non-local blocks into CNN backbones to capture long-range dependencies for retrieval [23]. Radenović et al. combined local and global features using regional attention mining, yielding descriptors that adaptively focus on salient image regions [24]. These hybrid models represent the current frontier by blending metric objectives with sophisticated network modules.

**From survey to method.** Having reviewed existing descriptors and metric-learning frameworks, we now turn to our own implementation of a Siamese network with contrastive loss to establish a performance baseline.

## 2.3 Siamese Networks and Contrastive Loss

Siamese neural networks are a specialized architecture designed explicitly for learning similarities or differences between pairs of inputs. Introduced initially in the early 1990s for signature verification tasks [25], Siamese networks have since gained popularity due to their effectiveness in various metric-learning tasks, including facial recognition, object tracking, and image retrieval [19].

Unlike traditional neural networks, Siamese networks consist of two identical subnetworks with shared weights. Each subnetwork processes one input of a pair separately, and the resulting embeddings are then compared using a predefined similarity or distance measure, such as Euclidean distance or cosine similarity [16]. The primary advantage of this architecture lies in its ability to directly learn feature representations optimized for similarity-based comparisons, thus making it particularly suitable for applications that require measuring semantic closeness between images or other data points [26].

### 2.3.1 Architecture of Siamese Networks

The fundamental architecture of a Siamese network involves two identical neural network branches. These branches share the same configuration, structure, and weights. Given a pair of inputs $(x_1, x_2)$, each input passes through its respective subnetwork to generate embedding vectors $(f(x_1), f(x_2))$. The similarity or dissimilarity between these embeddings is then quantified through a specific metric. By sharing parameters across both subnetworks, the network ensures that similar inputs produce embeddings that are closer together in the feature space, while dissimilar inputs yield embeddings that are farther apart [19].

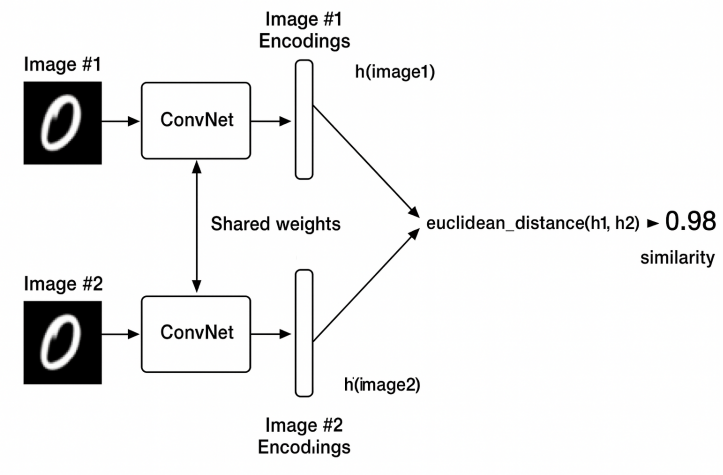



Figure 2.1: Siamese network architecture (adapted from Rosebrock [27]).

### 2.3.2 Contrastive Loss

To effectively train Siamese networks, a specialized loss function known as contrastive loss is frequently employed. Contrastive loss explicitly encodes the notion of similarity and dissimilarity by penalizing the network based on whether input pairs belong to the same or different classes [15].

Mathematically, contrastive loss ($L_{contrastive}$) is defined as follows:

$$L_{contrastive}(y, x_1, x_2) = y \cdot \frac{1}{2}[D(x_1, x_2)]^2 + (1 - y) \cdot \frac{1}{2}[\max(0, m - D(x_1, x_2))]^2 \tag{2.1}$$

where:

- $y$ is the binary indicator: $y = 1$ if inputs are similar, and $y = 0$ otherwise.
- $D(x_1, x_2)$ represents the distance between the embeddings of inputs $x_1$ and $x_2$.
- $m$ is a margin hyperparameter that determines the enforced minimum distance between embeddings of dissimilar inputs.

The intuitive interpretation of this loss function is that similar pairs ($y = 1$) are penalized if their embeddings are far apart, thus pushing them closer in the embedding space. Conversely, dissimilar pairs ($y = 0$) incur a penalty only if their embeddings are closer than the margin $m$, encouraging them to be separated by at least this distance [15].

### 2.3.3 Training Dynamics

The success of a Siamese network trained with contrastive loss heavily depends on effective sampling of input pairs. Typically, training involves balanced batches containing equal proportions of similar and dissimilar pairs to prevent bias towards one type. During training, hard examples—those challenging to classify correctly—are particularly valuable, as they provide stronger gradients and improve the discriminative ability of the model [18].

#### Impact of Weight Sharing

Weight sharing significantly contributes to the stability and convergence of Siamese network training. By constraining both subnetworks to have identical weights, the network drastically reduces the number of free parameters, thereby mitigating the risk of overfitting. This shared parameterization ensures consistent feature extraction across both inputs, reinforcing the learning of invariant and robust representations. Additionally, it provides symmetrical gradients during backpropagation, promoting stable convergence and preventing divergence, particularly in scenarios with limited training data.

#### Gradient Computations and Backpropagation Dynamics

Consider two inputs $x_1$ and $x_2$ passing through identical subnetworks to obtain embeddings $f(x_1)$ and $f(x_2)$. The loss function, typically contrastive loss, is computed as:

$$L_{contrastive}(y, x_1, x_2) = y \cdot \frac{1}{2}[D(x_1, x_2)]^2 + (1 - y) \cdot \frac{1}{2}[\max(0, m - D(x_1, x_2))]^2 \tag{2.2}$$

where $D(x_1, x_2) = ||f(x_1) - f(x_2)||_2$ represents Euclidean distance.

During backpropagation, the gradient with respect to each embedding vector is computed. For similar pairs ($y = 1$):

$$\frac{\partial L}{\partial f(x_i)} = (f(x_i) - f(x_j)), \quad \text{where } i, j \in 1{,}2, i \neq j \tag{2.3}$$

For dissimilar pairs ($y = 0$) with $D(x_1, x_2) < m$:

$$\frac{\partial L}{\partial f(x_i)} = -(m - D(x_1, x_2)) \frac{(f(x_i) - f(x_j))}{D(x_1, x_2)} \tag{2.4}$$

These gradients then propagate backward through the network, updating shared parameters simultaneously. The symmetry in gradient updates further contributes to the network's stability.

#### Implicit Metric Learning

Siamese networks implicitly perform metric learning by structuring their embedding space according to a learned distance metric. The contrastive loss function explicitly guides the network to arrange embeddings such that semantically similar inputs are closer together, and dissimilar inputs maintain a sufficient margin of separation. Through shared weight optimization, Siamese networks learn a nonlinear mapping from input space to a metric-structured embedding space, where distances correspond directly to semantic similarity. This implicit metric learning capability allows Siamese networks to generalize well to unseen classes or categories without explicit class labels during inference.

### 2.3.4 Comparison with Triplet Networks

Although Siamese networks and triplet networks share similar motivations—learning discriminative embeddings—they differ in their training structure. Triplet networks simultaneously process three samples (an anchor, a positive, and a negative), explicitly enforcing relative distances among embeddings. In contrast, Siamese networks handle pairs of inputs at a time, making them simpler to implement but potentially less effective in capturing complex semantic relations due to fewer constraints on the learned embedding space [18].

Nonetheless, Siamese networks offer computational advantages due to their simpler architecture and are effective in many practical scenarios, especially when the primary objective is pairwise similarity discrimination [19].

### Applications and Practical Considerations

Siamese networks have successfully been applied across various domains. In face recognition, Siamese architectures enable highly accurate face verification by directly comparing facial embeddings. Similarly, in object tracking, Siamese networks help maintain object identity across frames by embedding similarity metrics [20].

When applying Siamese networks, several practical considerations should be noted:

- Effective selection of the margin hyperparameter ($m$) significantly impacts the embedding space structure.
- Training pairs should be carefully selected to balance easy and hard examples, ensuring stable convergence.
- Embedding dimensionality should be chosen thoughtfully, balancing computational efficiency and representational power.

In summary, Siamese networks trained with contrastive loss provide a robust and efficient framework for similarity-based learning tasks, leveraging deep neural architectures to generate meaningful embeddings tailored for semantic comparison. However, to further improve representational capacity and capture finer visual distinctions, we next explore a variety of modern CNN backbones that serve as high-capacity encoders for our embedding pipeline.

## 2.4 Evolution and Principles of Convolutional Neural Networks

Convolutional Neural Networks (CNNs) have revolutionized the field of computer vision, emerging as the foundational architecture for modern image analysis and feature extraction tasks. Originally inspired by biological processes in the visual cortex, CNNs were introduced in the late 1980s by LeCun et al. with the creation of LeNet, a model designed for handwritten digit recognition [28]. Despite their early promise, initial adoption of CNNs was limited due to computational constraints and insufficient training data.

A significant breakthrough occurred in 2012 with the advent of AlexNet by Krizhevsky et al. [29], which demonstrated that CNNs, when combined with large-scale datasets such as ImageNet and powerful GPUs, could significantly outperform traditional hand-crafted image features. AlexNet's success sparked rapid innovation and led to deeper and more sophisticated CNN architectures such as VGGNet [30], GoogleNet [31], and ResNet [32]. These networks consistently pushed the boundaries of accuracy by introducing novel architectural elements like deeper convolutional layers, inception modules, and residual connections.

At the core of CNNs is the convolutional operation—a process involving sliding small filters across the input image to produce feature maps that encode spatial hierarchies. Early layers typically detect simple patterns such as edges and textures, while deeper layers capture complex semantic features like objects, parts, and their interactions. Convolutional layers are interleaved with pooling operations, reducing spatial dimensions and computational requirements while maintaining salient information. Non-linear activation functions such as Rectified Linear Units (ReLU) provide CNNs with the capacity to learn complex decision boundaries by introducing non-linearity into the model.

In recent years, CNNs have become increasingly efficient and specialized, leading to architectures tailored explicitly for deployment in resource-constrained environments. Networks such as MobileNet [33] and Xception [34] utilize depthwise separable convolutions, drastically reducing computational overhead while preserving strong representational power. This evolution highlights the shift towards creating CNNs that balance performance with computational efficiency, meeting the demands of diverse real-world applications.

Today, CNNs underpin most state-of-the-art visual recognition and embedding tasks, forming the basis for systems ranging from autonomous driving to medical imaging diagnostics. The continued refinement and innovation within CNN architectures ensure their pivotal role in advancing the frontiers of computer vision research and application.

## 2.5 CNN-Based Embedding Backbones

This section presents the different deep learning architectures explored as backbone encoders for the object embedding task. Multiple convolutional neural networks were investigated, each evaluated in terms of their capacity to generate compact and discriminative object embeddings suitable for object-level change detection.

### 2.5.1 ResNet50

ResNet50 is a deep convolutional neural network consisting of 50 layers and approximately 25.6 million parameters, specifically designed to facilitate the training of very deep models by leveraging residual learning [35]. The architecture begins with an initial convolutional layer and is followed by four main stages of convolutional operations, referred to as conv2_x through conv5_x. Each stage contains multiple residual blocks that incorporate shortcut connections, also known as skip connections. These connections allow the input of a block to be directly added to its output, forming a residual mapping defined by $F(x) = H(x) - x$. This structure helps to preserve gradient flow during backpropagation and greatly enhances training stability. In its original configuration for image classification tasks, the network concludes with a global average pooling layer followed by a fully connected classifier.

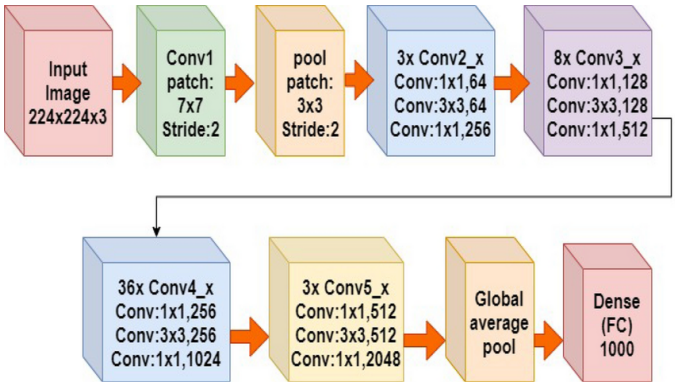



Figure 2.2: Basic architecture of ResNet-50, illustrating the sequence of convolutional and residual blocks across the network.[1]

### ResNet-50 Bottleneck Residual Block

The bottleneck residual block used in ResNet-50 is composed of three sequential convolutional layers: a $1 \times 1$ convolution for reducing the number of channels, a $3 \times 3$ convolution for spatial processing, and another $1 \times 1$ convolution for expanding the channels back to the original depth. In a typical configuration from the conv2_x stage, the input tensor has 256 channels. The first $1 \times 1$ convolution compresses this to 64 channels, followed by a $3 \times 3$ convolution operating on these 64 channels. The final $1 \times 1$ convolution increases the channel dimension back to 256, thereby preserving the original depth of the block. Each convolutional operation is followed by batch normalization and a ReLU activation, except for the final $1 \times 1$ layer, which omits the activation function. The original input (with 256 channels) is then added to the output of the three convolutional layers. A ReLU activation is applied to the result of this addition, producing the final output of the block. This residual connection enables the block to learn modifications to its input, making it easier to train very deep networks.

[1] *Image Sources:* Figure 2.2 Source: Adapted from ResearchGate, "Basic architecture of ResNet50," `https://www.researchgate.net/figure/Basic-architecture-of-ResNet50_fig5_363265826`.

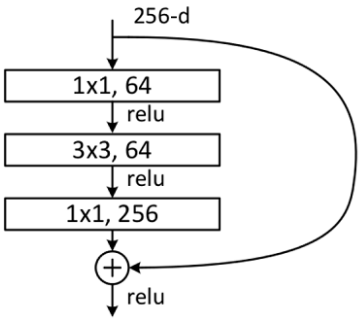



Figure 2.3: Structure of a bottleneck residual block used in ResNet-50 [32].

Having described the structure of the ResNet-50 bottleneck residual block, we now provide a detailed breakdown of each stage in the network, beginning with the initial convolution and progressing through all residual stages.

#### Initial Convolution and Pooling (conv1)

"The ResNet-50 model begins with an initial convolutional layer, referred to as `conv1`, which applies a $7 \times 7$ convolution (64 filters, stride = 2, padding = 3). This $7\times7$ conv is followed by BatchNorm and a ReLU activation, reducing the input resolution from $224 \times 224$ to $112 \times 112$. A $3 \times 3$ max pooling layer (stride = 2, padding = 1) then further down-samples the spatial dimensions to $56 \times 56$. This forms the stem of the network and prepares the input for the residual stages."

#### Residual Stage 1: conv2_x

The first residual stage, `conv2_x`, processes $56 \times 56$ feature maps and contains three bottleneck blocks. Each block follows the bottleneck structure described earlier. The output of this stage consists of feature maps with 256 channels. To ensure compatibility for residual addition, the first bottleneck block includes a $1 \times 1$ convolution on the shortcut path. This projects the 64-channel input (from the stem) to 256 channels, aligning the shortcut with the output of the block.

#### Residual Stage 2: conv3_x

The `conv3_x` stage includes four bottleneck blocks. The first block performs downsampling using a stride-2 convolution, reducing the feature map resolution to $28 \times 28$. The output channel depth is doubled to 512. As with the previous stage, a projection shortcut with a $1 \times 1$ convolution and stride 2 is used to match the dimensions for residual addition.

#### Residual Stage 3: conv4_x

The third residual stage, `conv4_x`, consists of six bottleneck blocks. The spatial resolution is further reduced to $14 \times 14$ through stride-2 convolution in the first block, and the output depth increases to 1024 channels. Projection shortcuts are again applied where required to match dimensions between the main and shortcut paths.

#### Residual Stage 4: conv5_x

The final residual stage, `conv5_x`, contains three bottleneck blocks. The first block performs downsampling, reducing the *spatial dimensions* of the feature map to $7 \times 7$, while increasing the number of channels to 2048. As in previous transitions, the first block includes a $1\times1$ convolution with stride 2 in the shortcut path to match the dimensions for residual addition.

## Classification Head

In its standard configuration, the ResNet-50 architecture is designed for large-scale image classification tasks such as the ImageNet benchmark [35]. After the final convolutional block (`conv5_x`), the network applies a global average pooling (GAP) operation to the $7 \times 7 \times 2048$ feature map, which reduces it to a 2048-dimensional vector.

This vector is then passed to a fully connected (Dense) layer with 1000 output units, followed by a softmax activation function to produce class probabilities. This canonical configuration has demonstrated strong performance on image classification benchmarks and is widely used as a backbone for transfer learning in vision tasks.

| Layer Name | Output Size | 18-Layer | 34-Layer | 50-Layer |
|---|---|---|---|---|
| cov1 | $112 \times 112$ | $7 \times 7$, 64, stride 2 | | |
| | | $3 \times 3$ max pool, stride 2 | | |
| cov2_x | $56 \times 56$ | $\begin{bmatrix} 3 \times 3, & 64 \\ 3 \times 3, & 64 \end{bmatrix} \times 2$ | $\begin{bmatrix} 3 \times 3, & 64 \\ 3 \times 3, & 64 \end{bmatrix} \times 3$ | $\begin{bmatrix} 1 \times 1, & 64 \\ 3 \times 3, & 64 \\ 1 \times 1, & 256 \end{bmatrix} \times 3$ |
| cov3_x | $28 \times 28$ | $\begin{bmatrix} 3 \times 3, & 128 \\ 3 \times 3, & 128 \end{bmatrix} \times 2$ | $\begin{bmatrix} 3 \times 3, & 128 \\ 3 \times 3, & 128 \end{bmatrix} \times 4$ | $\begin{bmatrix} 1 \times 1, & 128 \\ 3 \times 3, & 128 \\ 1 \times 1, & 512 \end{bmatrix} \times 4$ |
| cov4_x | $14 \times 14$ | $\begin{bmatrix} 3 \times 3, & 256 \\ 3 \times 3, & 256 \end{bmatrix} \times 2$ | $\begin{bmatrix} 3 \times 3, & 256 \\ 3 \times 3, & 256 \end{bmatrix} \times 6$ | $\begin{bmatrix} 1 \times 1, & 256 \\ 3 \times 3, & 256 \\ 1 \times 1, & 1028 \end{bmatrix} \times 6$ |
| cov5_x | $7 \times 7$ | $\begin{bmatrix} 3 \times 3, & 512 \\ 3 \times 3, & 512 \end{bmatrix} \times 2$ | $\begin{bmatrix} 3 \times 3, & 512 \\ 3 \times 3, & 512 \end{bmatrix} \times 3$ | $\begin{bmatrix} 1 \times 1, & 512 \\ 3 \times 3, & 512 \\ 1 \times 1, & 2048 \end{bmatrix} \times 3$ |
| | $1 \times 1$ | average pool, 1000-d fc, softmax | | |
| FLOPs | | $1.8 \times 10^9$ | $3.6 \times 10^9$ | $3.8 \times 10^9$ |

Figure 2.4: Detailed architecture of various ResNet models, including ResNet-50, showing the number and type of convolutional layers in each residual block, output sizes, and computational cost (FLOPs). ResNet-50 uses bottleneck blocks consisting of $1 \times 1$, $3 \times 3$, and $1 \times 1$ convolutions. Figure adapted from [32].

## Custom Embedding Head (ResNet-50)

Following the convolutional backbone of ResNet-50, which yields a $7 \times 7 \times 2048$ feature map from the `conv5_x` stage, a custom projection head is applied to transform the extracted features into low-dimensional embeddings suitable for similarity learning. The first component of this head is a Global Average Pooling 2D layer [36], which operates on the $7 \times 7 \times 2048$ output from the `conv5_x` stage. This layer performs spatial averaging over each feature map, reducing the tensor to a single 2048-dimensional vector. This operation compresses the spatial information into a compact global descriptor while retaining the semantic richness of the high-level features.

Subsequently, a Dense (fully-connected) layer with 128 output units is used to project the 2048-dimensional vector into a lower-dimensional embedding space. This layer produces a vector $\mathbf{e} \in \mathbb{R}^{128}$ for each input image. The embedding size of 128 dimensions was selected as a balanced choice between representation power and computational efficiency. This dimensionality is also widely used in deep metric learning literature for applications such as object retrieval and face recognition.

Notably, the output of the Dense layer does not pass through any non-linear activation function; it remains a purely linear transformation. This decision ensures that the full expressiveness of the embedding space is preserved. A linear output allows the model to produce embeddings with both positive and negative components, which is important for maintaining the geometric integrity of distance-based comparisons. Introducing a non-linear function such as ReLU would eliminate negative values and restrict the embedding space to $\mathbb{R}^{128}_{\geq 0}$, thereby distorting distance metrics such as cosine similarity or Euclidean distance. By avoiding this constraint, the learned embeddings retain directional and magnitude information necessary for fine-grained similarity learning

### 2.5.2 ResNet-101: Deeper Residual Architecture

ResNet-101 is a deeper variant of the ResNet family of convolutional neural networks introduced by He et al. [32], designed to enhance representational capacity through increased depth while maintaining efficient gradient propagation via residual connections. It retains the same architectural principles and bottleneck design as ResNet-50 but significantly extends the number of layers, particularly in the intermediate stages of the network. ResNet-101 contains approximately 44.5 million parameters, compared to 25.6 million in ResNet-50.

Structurally, ResNet-101 consists of 101 layers, including the initial convolution and pooling layers, followed by a series of bottleneck residual blocks distributed across four main stages: `conv2_x`, `conv3_x`, `conv4_x`, and `conv5_x`. The primary architectural difference from ResNet-50 lies in the `conv4_x` stage, which includes 23 bottleneck blocks in ResNet-101 compared to only 6 in ResNet-50. This substantial increase more than doubles the depth of the network, thereby enhancing its ability to capture intricate hierarchical features and subtle visual distinctions in complex scenes.

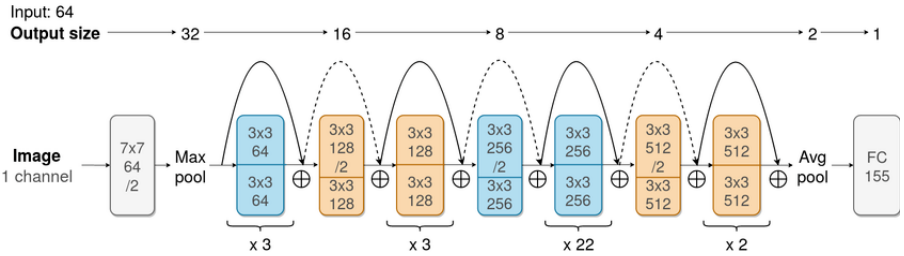



Figure 2.5: High-level schematic of ResNet-101. Each colored box represents one bottleneck block; the numbers above indicate output spatial size. In `conv2_x` there are 3 blocks (3×3×64), in `conv3_x` there are 4 blocks (3×3×128), in `conv4_x` there are 22 blocks (3×3×256), and in `conv5_x` there are 2 blocks (3×3×512). *Note: Although the diagram shows 22 blocks in `conv4_x`, the correct count for ResNet-101 is 23, as described in the original ResNet paper [32].*

| Layer Name | Output Shape | 50-Layers | 101-Layers |
|---|---|---|---|
| *conv*1 | 112×112 | 7×7, 64, stride 2 | 7×7, 64, stride 2 |
| *conv*2_*x* | 56×56 | 3×3 max pool, stride 2 | 3×3 max pool, stride 2 |
| | | [1×1, 64; 3×3, 64; 1×1, 256] × 3 | [1×1, 64; 3×3, 64; 1×1, 256] × 3 |
| *conv*3_*x* | 28×28 | [1×1, 128; 3×3, 128; 1×1, 512] × 4 | [1×1, 128; 3×3, 128; 1×1, 512] × 4 |
| *conv*4_*x* | 14×14 | [1×1, 256; 3×3, 256; 1×1, 1024] × 6 | [1×1, 256; 3×3, 256; 1×1, 1024] × 23 |
| *conv*5_*x* | 7×7 | [1×1, 512; 3×3, 512; 1×1, 2048] × 3 | [1×1, 512; 3×3, 512; 1×1, 2048] × 3 |
| | 1×1 | 3×3 average pool, 1000-d fc | 3×3 average pool, 1000-d fc |

Figure 2.6: Block-level breakdown of ResNet-50 and ResNet-101 architectures. The table highlights the number of residual blocks and the specific convolutional layer shapes in each stage. Notably, ResNet-101 includes 23 blocks in the `conv4_x` stage compared to 6 in ResNet-50. Figure adapted from [37].

#### Bottleneck Residual Blocks in ResNet-101

ResNet-101 uses the same bottleneck residual block design as ResNet-50 to maintain training stability across a deeper network. Each bottleneck block contains three convolutional layers in

the sequence: a $1 \times 1$ convolution for reducing the number of channels, a $3 \times 3$ convolution for spatial processing, and a final $1 \times 1$ convolution for restoring the channel depth. A shortcut connection adds the input to the block's output to form a residual mapping. This structure is critical for mitigating vanishing gradients in very deep networks.

## Classification Head

Similar to ResNet-50, the canonical ResNet-101 architecture concludes with a classification head tailored for large-scale image recognition tasks. After the final residual stage (`conv5_x`), the network outputs a $7 \times 7 \times 2048$ feature map. This is followed by a global average pooling (GAP) layer, which computes the mean across the spatial dimensions of each feature map, resulting in a 2048-dimensional vector.

This vector is then passed to a fully connected (Dense) layer with 1000 output units, each corresponding to a class in the ImageNet dataset. A softmax activation function is applied to produce a normalized probability distribution over the 1000 classes. This configuration forms the standard ResNet-101 architecture as introduced in [32], and is optimized for supervised image classification tasks on large datasets.

## Custom Embedding Head (ResNet-101)

To adapt ResNet-101 for similarity learning tasks, we replace its original classification head with the same custom projection head described in Section 2.5.1. This projection module, originally introduced in the ResNet-50 configuration, includes a global average pooling layer followed by a 128-dimensional dense projection layer.

No architectural changes were made to the projection head when used with ResNet-101; this ensures that any observed differences in embedding performance are attributable solely to the depth and structure of the backbone network.

### Motivation and Training Considerations

The rationale for including ResNet-101 in this study is to investigate the impact of increased depth on embedding quality and object-level change detection. In theory, deeper networks should be more capable of capturing subtle visual cues and hierarchical patterns, particularly in scenarios involving small, overlapping, or occluded objects. However, this benefit comes at the cost of increased computational complexity, model size, and inference time. As such, we evaluate ResNet-101 alongside its shallower counterparts to assess whether the performance gains justify the additional resource requirements in practical deployment settings. Training ResNet-101 required longer convergence time, and regularization techniques such as dropout and data augmentation were crucial due to the model's higher capacity and increased risk of overfitting.

Table 2.1 summarizes the key architectural and computational differences between ResNet-50 and ResNet-101 used in this study.

| **Feature** | **ResNet-50** | **ResNet-101** |
|---|---|---|
| **Total Layers** | 50 | 101 |
| **conv4_x Block Count** | 6 | 23 |
| **Total Parameters** | ∼25.6 million | ∼44.5 million |
| **FLOPs (224×224 input)** | 4.1 GFLOPs | 7.8 GFLOPs [38] |
| **Backbone Output Dimensionality** | 2048 | 2048 |

Table 2.1: Comparative summary of ResNet-50 and ResNet-101 characteristics and performance.

With the next two architectures, MobileNetV2 and Xception, we shift from standard convolutional blocks to a more efficient design paradigm: *depthwise separable convolutions*. To enable a clearer understanding of their design philosophy and performance advantages, we provide a brief explanation of this technique below.

### Depthwise Separable Convolutions

Depthwise separable convolutions are a widely used architectural innovation designed to reduce computational complexity and model size, without significantly compromising performance [34], [39]. They have become a core building block in many efficient convolutional neural networks, including MobileNet and Xception.

This technique involves decomposing a standard convolution into two operations:

- A **depthwise convolution**, which applies a spatial filter to each input channel independently, extracting localized patterns.
- A **pointwise convolution** (a $1\times1$ convolution), which combines the output of the depthwise step across all channels to produce a richer representation.

By decoupling spatial and cross-channel processing, depthwise separable convolutions dramatically reduce the number of parameters and floating-point operations compared to traditional convolutions, while retaining much of their expressive power.

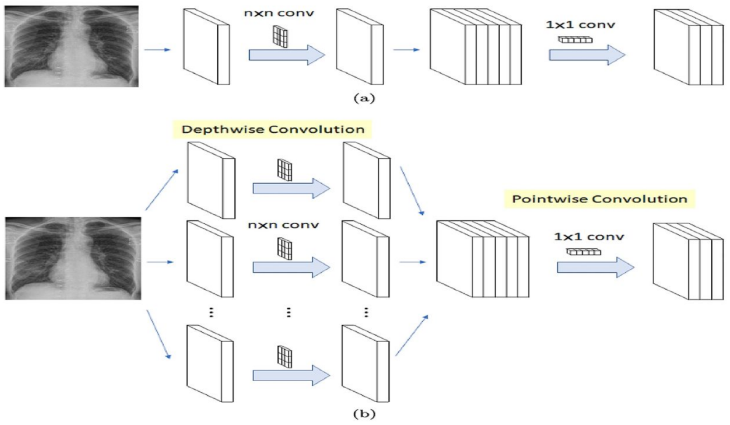



Figure 2.7: Side-by-side visualization of standard convolution (top) and depthwise separable convolution (bottom), highlighting how spatial and channel-wise operations are decoupled. Adapted from [40].

### 2.5.3 MobileNetV2

MobileNetV2 is a convolutional neural network (CNN) architecture designed to deliver high classification accuracy while maintaining exceptional computational efficiency, making it particularly suitable for mobile and embedded vision systems. Originally introduced by Sandler et al. [33], it extends the ideas of its predecessor, MobileNetV1, by incorporating two major innovations: inverted residual connections and linear bottlenecks. These mechanisms, combined with the use of *depthwise separable convolutions*—a technique that decomposes standard convolutions into lightweight spatial and channel-wise operations—enable the model to operate effectively on resource-constrained platforms.

With approximately 3.4 million parameters [33], MobileNetV2 has been widely adopted across computer vision tasks such as image classification, object detection, and semantic segmentation. Compared to conventional architectures like ResNet or Xception, it delivers competitive accuracy at a fraction of the computational cost. In the biomedical domain, for instance, MobileNetV2 has been successfully applied to the classification of systemic sclerosis skin images, achieving higher accuracy than traditional CNNs while significantly reducing training time [41].

## Initial Convolution Layer

The MobileNetV2 architecture begins with a standard convolutional layer designed to process raw input images. Specifically, this layer applies a $3 \times 3$ convolution operation with a stride of 2 and 32 output channels. The use of stride 2 reduces the spatial resolution of the input image by half, which lowers the computational burden in subsequent layers while preserving essential spatial structure.

This initial convolution plays a dual role: it acts as a low-level feature extractor by capturing edge and texture information, and it increases the channel dimensionality from 3 (RGB) to 32, allowing the network to process richer feature maps in the later bottleneck blocks. A `ReLU6` activation function typically follows this convolution to introduce non-linearity while ensuring compatibility with low-precision computation on mobile hardware. Unlike the standard ReLU, which is unbounded, `ReLU6` caps activations at 6. This constraint prevents excessively large values that could destabilize quantized inference, improving numerical stability and efficiency for 8-bit integer hardware deployments [33].

This layer sets the foundation for the efficient processing pipeline that follows in MobileNetV2, ensuring that the early-stage features are appropriately compressed and enhanced for the network's lightweight bottleneck modules.

## Inverted Residual Blocks

The core innovation of the MobileNetV2 architecture lies in its use of *inverted residual blocks*, which serve as the principal building units throughout the network. These blocks are engineered to maximize efficiency and representational capacity while minimizing computational overhead, particularly on low-resource devices.

Each inverted residual block is composed of three primary stages:

- **Expansion Layer:** The input tensor, which typically has a low channel dimensionality, is first passed through a $1 \times 1$ pointwise convolution that expands the number of channels by a predefined expansion factor (commonly $t = 6$). This step increases the capacity of the model to learn complex features by lifting the input to a higher-dimensional feature space. A ReLU6 activation and batch normalization follow this expansion.

- **Depthwise Convolution:** The expanded feature map undergoes a $3 \times 3$ *depthwise convolution*, which applies a single filter per input channel, rather than combining information across channels. This technique drastically reduces the number of parameters and floating-point operations compared to standard convolutions. It is also followed by batch normalization and ReLU6 activation.

- **Projection Layer (Linear Bottleneck):** Finally, a $1 \times 1$ pointwise convolution projects the high-dimensional features back to a lower-dimensional output space. This projection is followed by BatchNorm, with no ReLU6 activation afterward. Importantly, this design choice—referred to as a *linear bottleneck*—helps preserve representational integrity and avoids information loss that could occur if a non-linear activation were applied to low-dimensional projections.

The term "inverted" residual block arises from the reversal of the conventional ResNet bottleneck structure: instead of compressing and then expanding features, MobileNetV2 expands first and compresses at the end. When the input and output dimensions match and the stride is 1, the input is added element-wise to the output of the projection layer, forming a skip connection that enables gradient flow and mitigates vanishing gradient issues during training. The block structure is illustrated in Figure 2.8 which highlights the expansion–depthwise–projection pipeline and skip connection. This structure allows MobileNetV2 to construct deep networks that are both memory- and computation-efficient, while still maintaining high classification accuracy [33].

| Input Size | Operator | Expansion Factor (t) | Output Channels (c) | Repeats (n) | Stride (s) |
|---|---|---|---|---|---|
| $112 \times 112 \times 32$ | Bottleneck | 1 | 16 | 1 | 1 |
| $112 \times 112 \times 16$ | Bottleneck | 6 | 24 | 2 | 2 |
| $56 \times 56 \times 24$ | Bottleneck | 6 | 32 | 3 | 2 |
| $28 \times 28 \times 32$ | Bottleneck | 6 | 64 | 4 | 2 |
| $14 \times 14 \times 64$ | Bottleneck | 6 | 96 | 3 | 1 |
| $14 \times 14 \times 96$ | Bottleneck | 6 | 160 | 3 | 2 |
| $7 \times 7 \times 160$ | Bottleneck | 6 | 320 | 1 | 1 |
| $7 \times 7 \times 320$ | $1 \times 1$ Conv | - | 1280 | 1 | 1 |
| $7 \times 7 \times 1280$ | Global Avg Pool | - | 1280 | 1 | - |
| $1 \times 1 \times 1280$ | Fully Connected | - | Num. of Classes | 1 | - |

Table 2.2: Layer-wise configuration of MobileNetV2, adapted from [42]. The table begins with the first inverted residual block, immediately following the initial $3 \times 3$ convolutional layer applied to the $224 \times 224 \times 3$ input image.

## Classification Head

Once all inverted–residual blocks (each ending with its linear bottleneck) have processed the input, MobileNetV2 concludes with:

- **Global Average Pooling:** Reduces $h \times w \times 1280 \to 1 \times 1 \times 1280$.
- **Fully Connected + Softmax:** Maps the 1280-dim vector to $C$ outputs (the number of classes), followed by a softmax activation.

## Custom Embedding Head (MobileNetV2)

After passing through the sequence of inverted residual blocks, MobileNetV2 concludes its feature extraction pipeline with a compact and efficient embedding head. Unlike the original classification architecture—which includes a global average pooling (GAP) layer followed by a fully connected layer with softmax activation—our configuration replaces the classifier with a lightweight projection module suitable for metric learning.

**Global Average Pooling (GAP):** The feature map emerging from the last bottleneck block is first reduced using a global average pooling layer. This operation compresses the $h \times w \times c$ tensor into a fixed-length $1 \times 1 \times c$ vector (with $c = 1280$), summarizing spatial information without introducing dense parameters. GAP has been shown to improve generalization and is particularly effective in translation-invariant feature learning [36].

**Dense Projection and Normalization:** The pooled feature vector is passed through a dense layer that projects it into a lower-dimensional embedding space (e.g., 128 dimensions), followed by an $L_2$ normalization layer. This configuration ensures that output vectors lie on a unit hypersphere, which is crucial for cosine similarity-based comparison. The use of $L_2$ normalization aligns the training objective with angular distance metrics typically used in contrastive and triplet loss formulations.

This modification makes the architecture suitable for embedding-based applications such as object matching and change detection, rather than classification. The fully connected softmax layer used in traditional MobileNetV2 classifiers is excluded in our pipeline to ensure the output remains suitable for similarity learning tasks.

This final structure is intentionally lightweight and optimized for edge inference. The use of global average pooling and a streamlined projection head significantly reduces model size and complexity without compromising performance, especially when combined with pretraining or fine-tuning on domain-specific datasets [43], [44].

Figure 2.8 illustrates the standard classification architecture of MobileNetV2, which is adapted later by replacing the final classification head with a custom embedding projection module.

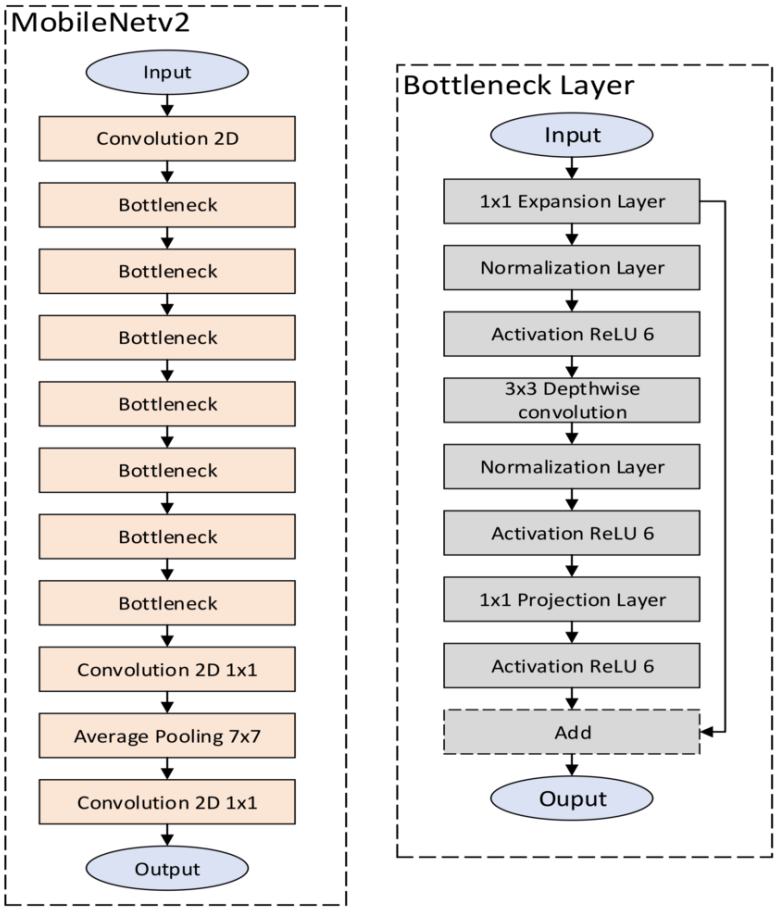



Figure 2.8: Overall architecture of MobileNetV2, showing the initial convolution, inverted residual blocks, and the final classification head. Adapted from [45].

### 2.5.4 Xception

Xception, short for "Extreme Inception," is a deep convolutional neural network architecture introduced by François Chollet [34]. It builds upon the Inception family of models but significantly simplifies the design by replacing the complex Inception modules with depthwise separable convolutions. This architectural shift allows for more efficient computation and better performance on image classification tasks, while also maintaining a streamlined and modular design that is easier to optimize.

Depthwise separable convolutions, the core building block of Xception, break down standard convolutions into two separate steps: a depthwise convolution, which applies a spatial filter to each input channel individually, followed by a pointwise $(1 \times 1)$ convolution that merges information across channels. This separation drastically reduces the number of parameters and computational cost, enabling efficient yet expressive feature learning.

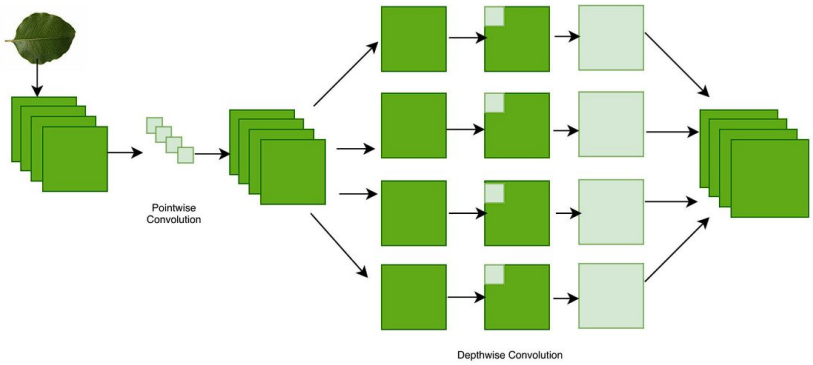



Figure 2.9: Illustration of a depthwise separable convolution used in the Xception architecture. Figure adapted from [46].

The total number of parameters in the Xception model is approximately 22.9 million, making it slightly smaller than ResNet-50 (25.6M) while achieving comparable or better performance on standard benchmarks such as ImageNet.

In terms of computational cost, Xception requires approximately 8.4 GFLOPs for a $299 \times 299$ input, which is nearly double that of ResNet-50 (4.1 GFLOPs at $224 \times 224$) but provides improved accuracy and architectural simplicity [34]. While more computationally intensive than lightweight models like MobileNetV2 (0.3 GFLOPs), Xception offers a better balance between expressiveness and efficiency, especially in scenarios where model capacity is more critical than minimal inference time.

Due to its balance between accuracy and efficiency, Xception has become a widely adopted backbone for tasks such as image classification, object detection, and transfer learning applications.

## Structure

The Xception architecture is composed of a total of 14 distinct modules that are organized into three main functional blocks: the Entry Flow, the Middle Flow, and the Exit Flow. This modular division enables the network to progressively extract hierarchical features, from low-level textures to high-level semantic concepts, in a structured and computationally efficient manner [34]. Each of these three flows plays a critical role in enabling Xception to efficiently model complex image features while maintaining a manageable number of parameters and operations.

We now proceed to review each of the three major flows—Entry, Middle, and Exit—in greater detail, highlighting their architectural components, functionality, and impact on feature learning.

### Entry Flow

The Entry Flow is responsible for initial feature extraction and spatial resolution reduction. It begins with two standard convolutional layers that operate directly on the input image, which has a resolution of $299 \times 299 \times 3$. The first convolution layer uses 32 filters of size $3 \times 3$ with a stride of $2 \times 2$, thereby reducing the spatial dimensions and extracting low-level edge and texture features. This is followed by a second $3 \times 3$ convolutional layer with 64 filters and a stride of $1 \times 1$, further enriching the representation. Both layers are followed by batch normalization and the ReLU activation function to introduce non-linearity and maintain stable gradients during training.

After the initial convolutions, the architecture applies a sequence of modified depthwise separable convolution layers, each followed by batch normalization and ReLU. These operations are

interleaved with $3 \times 3$ max pooling layers with a stride of $2 \times 2$, which further downsample the feature maps while retaining salient features. To ensure effective gradient propagation and feature reuse, residual connections are added in parallel to the depthwise separable convolutions. These skip connections are implemented using $1 \times 1$ convolutions to match the spatial dimensions and channel depth, allowing for seamless addition of the residual paths.

By the end of the Entry Flow, the network has transformed the high-resolution input image into a compact, high-dimensional feature map that preserves essential spatial and semantic information for deeper stages of processing [34].

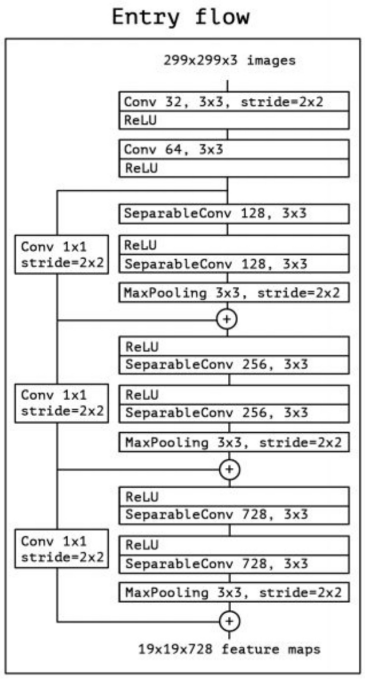



Figure 2.10: Visualization of the Entry Flow in the Xception architecture.Figure adapted from [34].

**Middle Flow:**

The Middle Flow is the core feature extraction stage of the Xception architecture. It consists of eight identical modules, each made up of three depthwise separable convolution layers with $3 \times 3$ kernels and 728 filters. Each layer is followed by batch normalization and a ReLU activation. Residual connections span each module, preserving gradient flow and enabling the efficient training of deep representations.

Unlike the Entry and Exit flows, the spatial dimensions of the feature maps remain unchanged throughout the Middle Flow. This design allows the network to refine and enrich intermediate feature representations without altering the resolution. By repeating the same architectural unit eight times, the model enhances its abstraction capacity while keeping the number of additional parameters minimal. This repeated structure has been shown to improve generalization on tasks such as image classification and transfer learning.

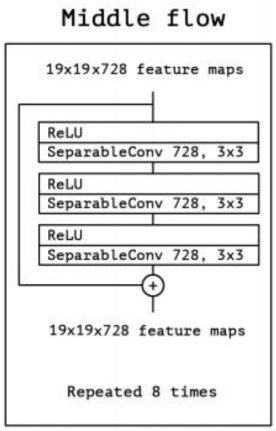



Figure 2.11: Illustration of the Middle Flow in the Xception architecture, which comprises eight repeated modules of depthwise separable convolutions with residual connections. This block enhances the network's ability to model increasingly abstract features.Figure adapted from [34].

### Exit Flow

The Exit Flow finalizes the feature extraction process by further refining the high-level representations produced by the Middle Flow. It begins with a depthwise separable convolution layer with 728 filters and continues through a series of separable convolutions with progressively increasing channel depths: 1024, 1536, and finally 2048 filters, each using $3 \times 3$ kernels. As in earlier flows, each convolution is followed by batch normalization and a ReLU activation function. However, in the final convolutional block, no residual connection is applied. This design choice allows the network to avoid mixing original and transformed signals, thereby encouraging the final feature maps to represent purely learned semantic abstractions without shortcut dependencies.

After the convolutional stack, the architecture applies a Global Average Pooling layer, which compresses the spatial dimensions of the feature maps into a single vector by averaging across each channel. This operation significantly reduces the number of parameters and improves generalization. The resulting vector is then passed to a fully connected layer with a logistic regression classifier, producing the final output probabilities over the target classes.

This flow ensures that the network captures the most abstract and semantically rich features while maintaining computational efficiency and preserving gradient flow during training through residual connections.

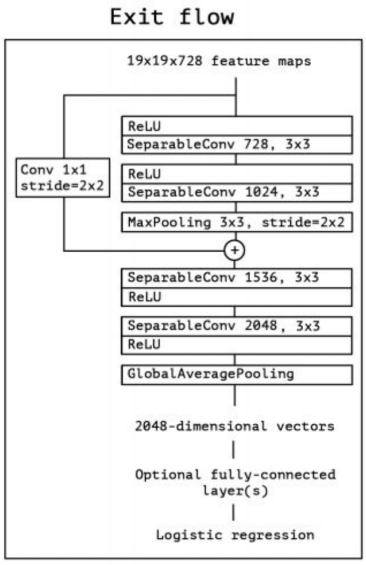



Figure 2.12: Diagram of the Exit Flow in the Xception architecture, which finalizes feature extraction using high-dimensional depthwise separable convolutions, followed by global average pooling and a dense classification layer. Figure adapted from [34].

#### Residual Connectivity

Residual connections run between every module except the very first and the final module. As detailed in the Entry, Middle, and Exit Flow descriptions, each skip consists of a $1 \times 1$ convolution (when spatial dimensions or channel depth change) so that the summed shortcut matches the module's output. These shortcuts preserve gradient flow and ensure stable training even with 36 convolutional layers in 14 modules, mitigating vanishing-gradient issues in a deep stack [32].

### Custom Embedding Head (Xception)

While the original Xception architecture concludes with a global average pooling (GAP) layer followed by a fully connected classification layer, our implementation replaces this classification head with a lightweight projection module suitable for embedding-based tasks.

After GAP, the resulting $1 \times 1 \times 2048$ feature vector is passed through a dense projection layer that maps it into a lower-dimensional embedding space (typically 128 dimensions). To align the output with cosine similarity–based training objectives, an $L_2$ normalization layer follows the projection. This normalization ensures that all embedding vectors lie on the unit hypersphere, enabling stable distance-based comparisons.

This modification transforms Xception from a classifier into an embedding model suitable for metric learning, where similarity is computed using angular distance. Such a design is critical for applications like object-level change detection, where feature proximity reflects semantic consistency across different image views.

## 2.5.5 Summary

In this chapter we reviewed Siamese networks with contrastive loss and showed why we transitioned to modern CNN backbones for embedding. In Chapter 3.10.1, we detail the triplet-based training pipeline that yields robust similarity descriptors.

| Model | Params (M) | FLOPs | Backbone Output Dim | Embed Dim |
|---|---|---|---|---|
| ResNet-50 | 25.6 | 4.1G | 2048 | 128 |
| ResNet-101 | 44.5 | 7.8G | 2048 | 128 |
| MobileNetV2 | 3.4 | 0.3G | 1280 | 128 |
| Xception | 22.9 | 8.4G | 2048 | 128 |

Table 2.3: Summary of parameter count, computational complexity, and dimensionality characteristics for each CNN backbone used in the system. The Backbone Output Dim refers to the size of the feature map produced after global average pooling in the pretrained model, while the Embed Dim denotes the 128-dimensional vectors generated by the projection head in our pipeline, used for similarity computation and training.

As outlined in Table 2.3, the selected backbone architectures vary considerably in depth, parameter count, and computational cost. These four models—ResNet-50, ResNet-101, MobileNetV2, and Xception—were chosen to explore a diverse design space that spans deep residual networks, lightweight mobile-friendly architectures, and depthwise separable convolution schemes. Each model is integrated into the triplet-based training framework described in Chapter 3, enabling a systematic evaluation of trade-offs between representational power, efficiency, and fine-grained matching accuracy. The detailed architectural review provided in this chapter was therefore essential for understanding their role and performance in the subsequent experimental pipeline.

# Chapter 3

# Methodology

In this chapter, we construct a complete object-level similarity learning pipeline using triplet loss and deep CNN embeddings. We detail each component of the system, including dataset preprocessing, object cropping, embedding generation, training strategy, and inference evaluation.

## 3.1 Dataset and Annotations

The dataset used in this study adopts the COCO (Common Objects in Context) format, a widely used schema for object detection and instance segmentation tasks. The COCO standard defines a structured annotation format that includes fields for image metadata, object categories, bounding boxes, and segmentation masks. Its hierarchical structure supports fine-grained object-level analysis and is particularly well suited for training deep learning models in vision tasks involving multiple objects per scene.

### Dataset Description

The dataset used in this study consists of real-world visual scenes captured from smart waste disposal environments. Each image depicts the interior of a household or industrial waste bin, containing multiple discarded items such as plastic containers, food packaging, paper waste, and metallic objects. These scenes exhibit naturally occurring clutter, partial occlusion, varied object orientations, and challenging lighting conditions. This dataset was provided by NANDO (ReLearn), a company specializing in AI-based waste management technologies.

The dataset is hosted in a remote cloud-based storage environment and is accessed programmatically through secure integration with the experimental training pipeline. This setup ensures efficient and scalable data retrieval without requiring local storage or manual intervention during model development.

Each image is paired with a structured annotation file adhering to a COCO-style schema. These annotations include object instance metadata such as segmentation masks, bounding boxes, and categorical labels. Additional custom fields are also embedded to support object-level change detection tasks across image pairs.

Images are preprocessed to a fixed resolution of $224 \times 224$ pixels and grouped into distinct training, validation, and test subsets. The dataset spans a wide spectrum of waste materials, container conditions, and spatial layouts, providing a robust basis for learning embeddings that generalize well across intra-class variability and real-world occlusions.


Figure 3.1: Example images from the dataset showing waste items with their segmentation masks.

### 3.1.1 Annotation Schema and Metadata

The annotation file used in this study adheres to a COCO-style structure and includes four primary components: `info`, `images`, `categories`, and `annotations`. These elements respectively encode dataset-level metadata, image descriptors, category definitions, and per-object annotation entries.

Each image entry contains a unique identifier along with its associated filename, height, and width. Every object annotation is linked to its corresponding image via the `image_id` field and includes both a bounding box (`bbox`) and a segmentation mask (`segmentation`) for precise spatial delineation. Category assignments are encoded using the `category_id`, which maps to human-readable labels defined in the `categories` section.

In addition to standard COCO fields, the annotation format incorporates custom attributes to support object-level change detection. These include flags such as `is_new`, which indicates whether the object appears only in the later image of a pair, and `is_occluded`, which identifies instances that are partially obscured. These metadata fields enable downstream modules to distinguish added objects and handle ambiguous cases during embedding comparison.

To illustrate the schema, a sample entry from each annotation component is shown below:

```
"images": [
  {
    "id": 1,
```

```
    "width": 1280,
    "height": 1024,
    "file_name":
        "real/2024_Week31_Tiny-0615_plastic_metal/data/2024-08-08___09-00-06.jpg",
    "license": 0,
    "flickr_url": "",
    "coco_url": "",
    "date_captured": 0
  }
],
"annotations": [
  {
    "id": 1,
    "image_id": 1,
    "category_id": 39,
    "segmentation": [[605.92, 965.16, 763.46, 804.04, ...]],
    "area": 117128.0,
    "bbox": [386.32, 429.29, 441.59, 535.87],
    "iscrowd": 0,
    "attributes": {"new": "yes", "occluded": false}
  }
],
"categories": [
  {
    "id": 1,
    "name": "ALUMINIUM CAN",
    "supercategory": ""
  }
]
```


Listing 3.1: Sample COCO-style JSON entries for image, annotation, and category fields

### 3.1.2 Category Set and Annotation Statistics

The dataset comprises 7,150 images and 36,739 object-level annotations, spanning 53 distinct semantic categories. Each object instance is associated with a unique `category_id` corresponding to one of the predefined classes, such as *paper cup*, *plastic bottle*, or *aluminium can*. These categories reflect typical waste types encountered in real-world disposal settings, particularly in office environments.

Table 3.1 and Figure 3.2 jointly illustrate the distribution of annotations across categories. A substantial imbalance is evident: dominant classes like *paper cup* and *crumbled tissue* appear over 6,000 times across more than 2,500 distinct images. In contrast, rare categories such as *COVID test* or *laptop charger* occur in only a single image each, often with a single annotated instance.

This class imbalance arises naturally from the frequency of certain disposal behaviors—items like paper cups and tissues are common in shared office spaces, whereas specialized items such as medical kits or electronics appear far less often. As a result, the dataset reflects realistic usage patterns but presents challenges for training generalizable models.

| Category ID | Category Name | Instance Count |
|---|---|---|
| 20 | PAPER CUP | 6,890 |
| 9 | CRUMBLED TISSUE | 5,576 |
| 22 | PAPER PACKAGING | 5,034 |
| 30 | PLASTIC BOTTLE | 2,548 |
| 39 | PLASTIC SNACK PACKAGING | 2,435 |
| 37 | PLASTIC PACKAGING | 1,935 |
| 33 | PLASTIC CUP | 1,927 |
| 1 | ALUMINIUM CAN | 1,473 |
| 18 | ORGANIC SCRAPS | 1,071 |
| 32 | PLASTIC CAP | 911 |

Table 3.1: Top 10 most frequent object categories by instance count.

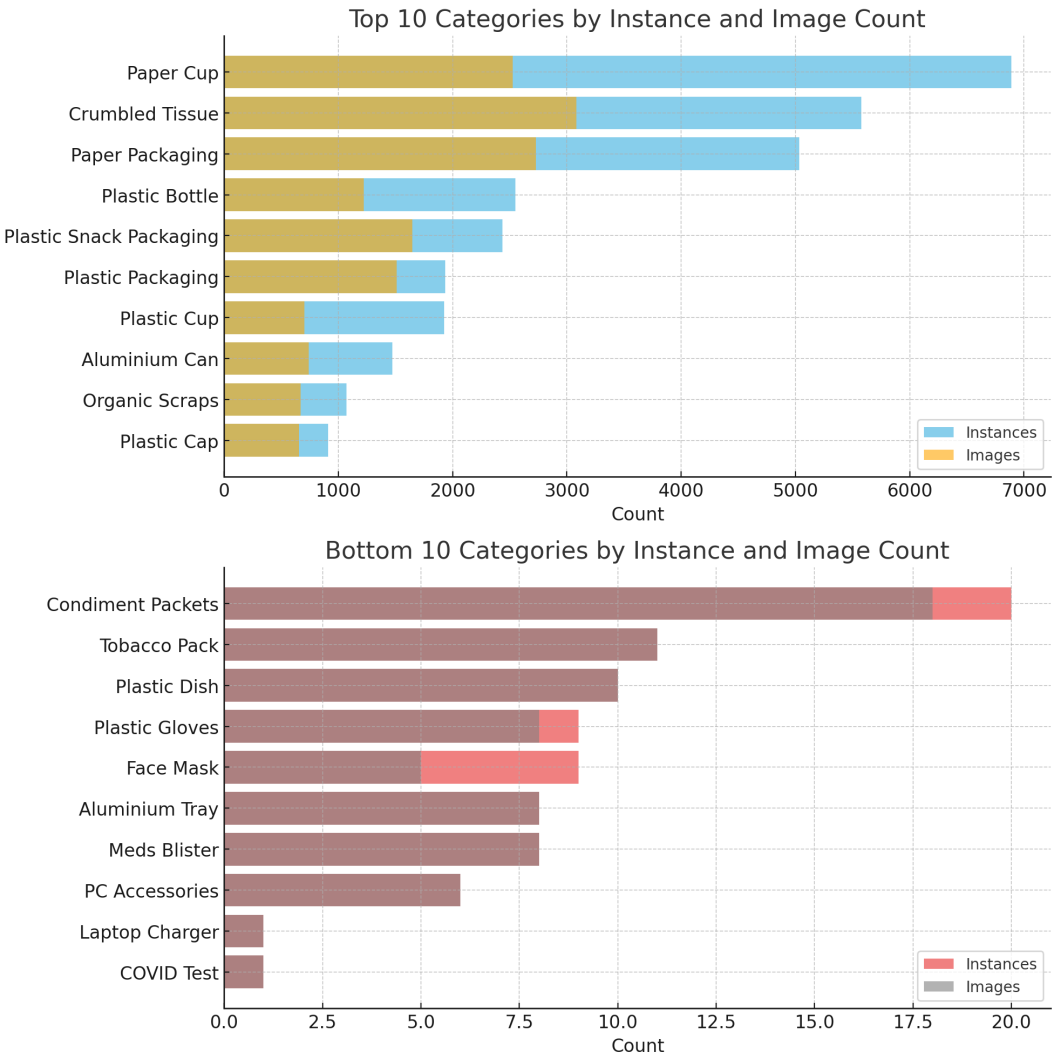



Figure 3.2: Annotation distribution across the top and bottom 10 object categories. The figure shows both the number of annotations and the number of distinct images per category.

### 3.1.3 Image Integrity Filtering

To ensure the reliability of the dataset before training, a systematic filtering step was implemented to detect and remove corrupted or unreadable images. Each image in the dataset was programmatically opened and verified using the Python Imaging Library (PIL). The verification step (‘img.verify()‘) ensures that each file is a valid image and not partially written, truncated, or in an unsupported format.

Images that triggered exceptions during this check were flagged as corrupted and excluded from the dataset. This step helped prevent downstream issues during preprocessing, model training, or augmentation.

The filtering process resulted in the exclusion of 192 invalid files (the exact filenames are logged and available for inspection). The remaining image set was stored in a cleaned image dictionary (`img_map`), which was used for all subsequent annotation processing and model input generation.

### 3.1.4 Dataset Splitting Strategy

Following image verification and the exclusion of 192 corrupted or invalid images, the remaining dataset was split into training, validation, and test subsets using a fixed 80/15/5 ratio. The final counts after filtering are summarized in Table 3.2. .

| Split | Image Count | Percentage |
|---|---|---|
| Training Set | 5,566 | 80% |
| Validation Set | 1,044 | 15% |
| Test Set | 348 | 5% |
| **Total** | **6,958** | **100%** |

Table 3.2: Dataset split distribution by subset and percentage.

The resulting split assignment was saved in a dictionary format to ensure reproducibility and was used consistently throughout the training and evaluation pipeline.

## 3.2 Object Extraction and Cropping

Accurate object-level comparison begins with precise extraction of individual object instances from their source images. While several strategies exist—such as extracting raw bounding boxes, applying ROI-aligned crops, or using segmentation with fixed padding—this system adopts a **polygon-based cropping method** guided by segmentation masks. This approach has been empirically shown to improve visual consistency and model accuracy by reducing background noise and preserving object shape.

### 3.2.1 Polygon-Based Cropping

To enable object-level comparison, each annotated object is individually extracted from its source image. Rather than relying solely on bounding boxes, this system uses polygon segmentation masks provided in the COCO-style annotation file to define more precise object boundaries.

Each image entry is loaded from disk and if the image is corrupt or unreadable, it is safely skipped to ensure robustness in large-scale processing.

For each annotation, the ‘segmentation‘ field contains polygon coordinates that delineate the object’s shape. These polygons are parsed and reshaped into a set of $(x, y)$ coordinates. A binary mask is then created with the same dimensions as the image, where the region inside the polygon is filled with ones using OpenCV’s ‘fillPoly‘ function.

This binary mask is applied channel-wise to the image, effectively removing the background and isolating the object pixels. The system then computes the tightest rectangular bounding box around the nonzero mask area by identifying the minimum and maximum $x$ and $y$ coordinates of the active region.

From this region, a tight crop is extracted from the original image. To standardize input dimensions for deep learning models, the crop is resized to $224 \times 224$ pixels using TensorFlow's 'resize with pad'. This method preserves aspect ratio while padding the image to maintain square dimensions.

This approach ensures consistent and centered object crops with minimal background interference, crucial for accurate embedding generation in later stages.

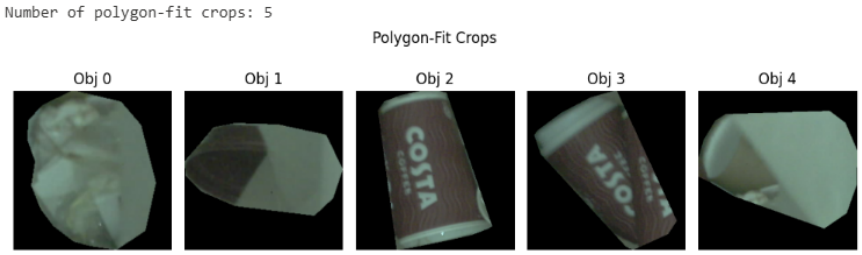



Figure 3.3: Polygon-fit object crops extracted from a single input image including five objects.

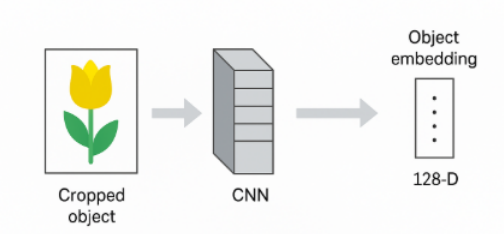
### 3.2.2 Metadata Management

Along with the cropped image patches, important metadata is recorded for each object to support matching, filtering, and triplet mining during training. The following fields are stored:

- **category_id**: Numerical identifier corresponding to the object class (e.g., bottle, paper, plastic).
- **is_new**: Boolean flag indicating whether the object is newly added in the second image of a pair. This value is extracted from the custom 'attributes' section of the annotation.
- **is_occluded**: Boolean flag indicating if the object is partially occluded. This is also retrieved from the optional 'attributes' field in the annotation.

The resulting crops and their associated metadata are returned together as Python lists, enabling downstream components to access both the visual and semantic properties of each object. This structured design supports later modules such as embedding comparison, similarity scoring, and change detection logic.

### 3.2.3 Embedding Generation for Cropped Objects




Once object crops have been extracted from each image using the polygon-based cropping strategy, the next step is to transform each visual crop into a compact vector representation that captures its semantic appearance. This is achieved using a pretrained or fine-tuned convolutional

neural network, which acts as an embedding backbone (e.g., ResNet-50, MobileNetV2, Xception) followed by a lightweight projection head.

Formally, let $\mathcal{C} = \{c_1, c_2, ..., c_N\}$ be the set of $N$ object crops extracted from a given image, where each $c_i$ is a tensor of shape $(224 \times 224 \times 3)$. These tensors are stacked into a batch and passed through the embedding model $\mathcal{F}_\theta$, yielding a corresponding set of embeddings:

$$\mathcal{E} = \{\mathbf{e}_1, \mathbf{e}_2, ..., \mathbf{e}_N\} = \mathcal{F}_\theta(\mathcal{C}) \quad \text{where} \quad \mathbf{e}_i \in \mathbb{R}^{128}$$

Each resulting vector $\mathbf{e}_i$ lies in a 128-dimensional embedding space and encapsulates the high-level visual features of the corresponding object crop. This embedding is the output of the final projection head attached to the CNN backbone, trained to produce normalized and semantically meaningful representations. The final output tensor has shape $(N, 128)$, where $N$ is the number of cropped objects in the image. These embeddings are later used for semantic comparison across object instances, forming the basis for matching, recognizing added objects, and triplet-based training strategies.

## 3.3 Cosine Similarity

Once object embeddings have been generated for a given image pair, the system performs pairwise comparison to identify semantically similar objects. This is achieved through a cosine similarity-based matching strategy, which computes angular similarity between embeddings while incorporating class-level consistency constraints.

Given two sets of embeddings—one from the first image and another from the second image—the system first applies $L_2$ normalization to each embedding vector. This projects all vectors onto the unit hypersphere, ensuring that similarity measurements are based purely on direction rather than magnitude. Cosine similarity is then computed between each embedding in the first image and every embedding in the second image, yielding a similarity matrix of shape $(N_A, N_B)$ , where $N_A$ and $N_B$ denote the number of objects in each image, respectively. Each element in the matrix reflects the cosine similarity score between a pair of object embeddings $(\mathbf{a}_i, \mathbf{b}_j)$, computed as:

$$\text{sim}(\mathbf{a}_i, \mathbf{b}_j) = \frac{\mathbf{a}_i \cdot \mathbf{b}_j}{\|\mathbf{a}_i\|_2 \cdot \|\mathbf{b}_j\|_2} \tag{3.1}$$

Since the vectors are $L_2$-normalized prior to comparison, we have:

$$\|\mathbf{a}_i\|_2 = \|\mathbf{b}_j\|_2 = 1$$

Substituting into the cosine similarity formula yields:

$$\text{sim}(\mathbf{a}_i, \mathbf{b}_j) = \frac{\mathbf{a}_i \cdot \mathbf{b}_j}{1 \cdot 1} = \mathbf{a}_i \cdot \mathbf{b}_j$$

Thus, the similarity score reduces to the dot product between the normalized vectors, which reflects the cosine of the angle between them.

In implementation, cosine similarity is computed using the `cdist` function, which returns cosine *distance*—defined as:

$$\text{dist}(\mathbf{a}_i, \mathbf{b}_j) = 1 - \text{sim}(\mathbf{a}_i, \mathbf{b}_j) \tag{3.2}$$

To recover true similarity scores, we subtract the result from 1.0. This ensures that each value in the resulting matrix reflects semantic similarity, with higher values indicating closer correspondence between object embeddings. The following example demonstrates how the similarity matrix is constructed by comparing objects across an input image pair.

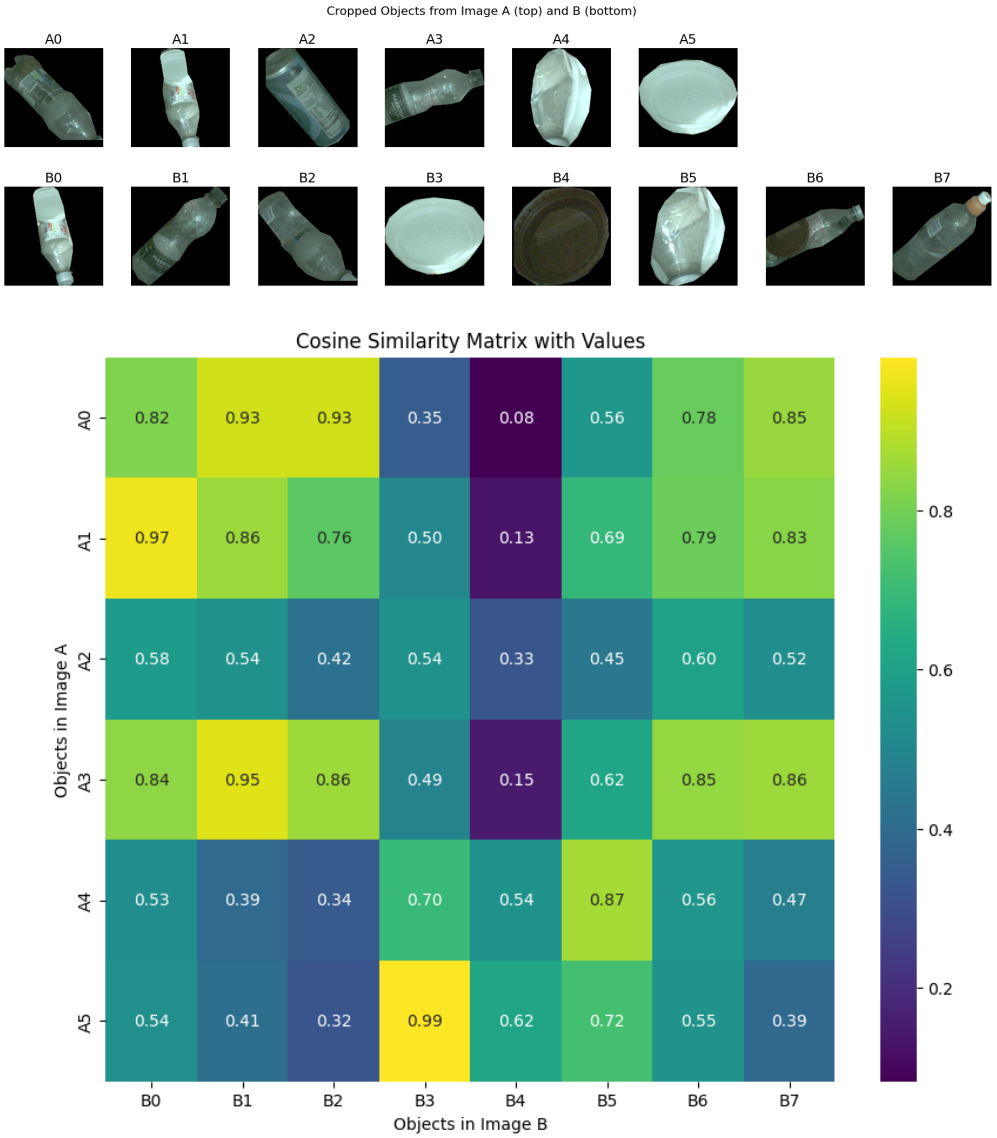



Figure 3.4: Example image pair used to construct the cosine similarity matrix. The system compares object embeddings from Image A (row) and Image B (column).

In this example, the system extracted 6 objects (A0–A5) from Image A and 8 objects (B0–B7) from Image B. For each cropped instance, $L_2$-normalized embeddings were computed and compared using cosine similarity, producing a $(6 \times 8)$ matrix of semantic similarity scores. The values indicate how closely each object in Image A corresponds to objects in Image B, with scores approaching 1 reflecting high alignment.

Cosine similarity constitutes a fundamental component of the proposed framework. It underpins triplet generation during both training phases (see Section 3.4), guides the object matching and added-object detection process (Section 3.7), and directly influences the evaluation outcomes (Section 4.3).

## 3.4 Triplet Learning and Triplet Loss

Triplet learning is a foundational approach in deep metric learning, designed to teach neural networks how to understand similarity relationships between data points. Rather than relying on

traditional classification labels, this method learns a continuous embedding space where samples from the same semantic class are mapped closer together, while samples from different classes are positioned further apart.

The learning process is based on structured training examples called triplets. Each triplet consists of three images: an *anchor* image, a *positive* image that is semantically similar to the anchor, and a *negative* image that is semantically dissimilar. The core objective is to ensure that the anchor is more similar (closer in the learned feature space) to the positive than to the negative.

This learning paradigm is particularly effective for tasks like face recognition, object re-identification, and fine-grained visual similarity, where the goal is not just to classify, but to reason about visual closeness. By focusing on relative comparisons rather than absolute labels, triplet learning enables the model to generalize well to unseen categories during inference.

In practical implementations, triplet learning is often coupled with techniques such as hard negative mining and custom sampling strategies to ensure that the selected triplets are informative and challenging, which accelerates convergence and enhances discriminative power.

### 3.4.1 Triplet Loss Formulation and Intuition

Triplet loss is a core concept in deep metric learning, designed to guide models to learn embeddings that reflect similarity relationships among inputs. Each training example consists of three samples: an **anchor** ($A$), a **positive** ($P$), and a **negative** ($N$). The anchor and positive belong to the same class, while the negative belongs to a different class. The objective is to ensure that the distance between the anchor and the positive is smaller than the distance between the anchor and the negative by at least a fixed margin $\alpha$. [18]

Mathematically, triplet loss can be expressed as:

$$\mathcal{L} = \sum_{i=1}^{m} \left[ \|f(x_a^i) - f(x_p^i)\|_2^2 - \|f(x_a^i) - f(x_n^i)\|_2^2 + \alpha \right]_+ \quad \text{(i.e., } \max(0, \cdot)\text{)}$$

Where:

- $f(x)$ is the embedding function (e.g., a deep neural network),
- $x_a^i$, $x_p^i$, $x_n^i$ are the anchor, positive, and negative samples,
- $\alpha$ is a margin hyperparameter that enforces a minimum distance between dissimilar pairs,
- $[\cdot]_+$ denotes the hinge function, i.e., $\max(0, \cdot)$.

This formulation encourages the model to learn an embedding space where similar items are pulled closer together and dissimilar items are pushed apart, improving the model's ability to differentiate between subtle variations in input data. It has been widely used in applications such as face verification, image retrieval, and fine-grained recognition tasks. [47], [48]

**Intuition Behind Triplet Loss:**

- Similar pairs (anchor and positive) should be embedded close together.
- Dissimilar pairs (anchor and negative) should be embedded far apart.

Through this learning mechanism, triplet loss enables the model to map raw input data into a representation space that captures semantic similarity relationships more effectively.

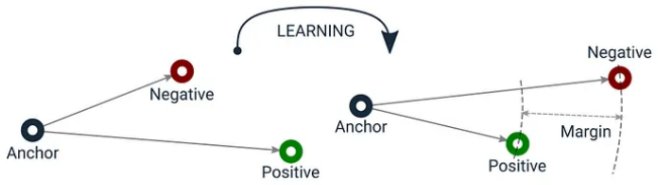



Figure 3.5: Visual explanation of triplet loss: training pulls anchor and positive samples closer together while pushing negatives apart by at least the margin $\alpha$ [49].

### Margin Selection

The margin $\alpha$ in the triplet loss function is a critical hyperparameter. A small margin may not enforce sufficient discrimination between positive and negative pairs, while a large margin can make it difficult for the model to satisfy the loss constraint, slowing or destabilizing training. Empirically, margins in the range $[0.1, 1.0]$ are often used and are selected via cross-validation.

### Triplet Mining

Efficient training with triplet loss relies heavily on how triplets are selected. Random triplet selection is typically ineffective, as most triplets will trivially satisfy the margin condition, yielding little to no loss. Instead, more informative mining strategies are employed:

- **Hard Negative Mining:** This strategy selects triplets where the negative is closer to the anchor than the positive, i.e., $\|f(x_a) - f(x_n)\|_2^2 < \|f(x_a) - f(x_p)\|_2^2$. These are the most difficult examples and lead to the strongest gradients. However, they can introduce noise and instability, especially early in training.

- **Semi-Hard Negative Mining:** This method selects negatives that are farther from the anchor than the positive, but still within the margin:

  $$\|f(x_a) - f(x_p)\|_2^2 < \|f(x_a) - f(x_n)\|_2^2 < \|f(x_a) - f(x_p)\|_2^2 + \alpha$$

  Semi-hard negatives are informative but less noisy than hard negatives, making them suitable for stable convergence.

- **Hard Positive Mining:** This strategy selects the most distant positive sample from the anchor, i.e., the same-class sample with the largest embedding distance:

  $$\|f(x_a) - f(x_p)\|_2^2 = \max_{\text{pos}} \|f(x_a) - f(x_p)\|_2^2$$

  Hard positives force the model to learn tighter intra-class clustering by pulling even difficult same-class samples closer to the anchor. However, they may introduce noise if the positives are mislabeled or contain significant artifacts.

- **Semi-Hard Positive Mining:** This method selects positives that are more distant than average but not the hardest. These samples represent natural intra-class variation without being outliers, making them useful for consistent gradient updates and better generalization.

- **Batch-All vs. Batch-Hard:** In **Batch-All** mining, all valid triplets within a mini-batch are used, increasing the number of training signals. In **Batch-Hard** mining, only the hardest positive and hardest negative per anchor are used. The choice depends on dataset size, batch size, and computational resources. [47]

Proper triplet mining ensures that the network continuously encounters challenging comparisons, which improves generalization and embedding discriminativeness. Mining strategy design is a key research area in deep metric learning and has a direct impact on final model performance.

### 3.4.2 Triplet Mining Strategy for Embedding Generation

To train an effective metric learning model, we implemented a structured triplet mining pipeline that extracts meaningful anchor-positive-negative tuples from the training dataset. Each triplet is formed by selecting:

- **Anchor:** an object crop from an image $A$.
- **Positive:** a semantically matching object from image $B$, with the same category label and high similarity to the anchor.
- **Negative:** a challenging distractor, ideally from the same category but with low similarity to the anchor.

This design encourages the model to distinguish fine-grained visual differences even within the same object class. Our triplet generation pipeline ensures both semantic and geometric diversity while maintaining control over difficulty.

#### Positive Sampling Strategy

Given a pair of images $A$ and $B$, we generate embeddings for all detected object crops in both images. For each object $a_i$ in image $A$, we compute cosine similarity with all objects in $B$. Positives are selected by:

- Matching `category_id` between $a_i$ and $b_j$,
- Cosine similarity $\geq \tau_{\text{pos}}$ (empirically set to 0.67),
- Ensuring each $b_j$ is used at most once.

Among candidates, the $b_j$ with highest similarity is chosen as the positive. This hard-positive mining ensures the model learns to focus on subtle differences even among visually similar objects.

#### Negative Sampling Strategy.

After selecting the anchor-positive pair $(a_i, b_j)$, the negative $n_k$ is sampled using the following logic:

- Prefer same-category negatives with similarity $\leq \tau_{\text{neg}}$ (set to 0.4),
- If no suitable negative exists in image $B$, a fallback negative is randomly sampled from a different image containing at least one valid crop.

This ensures that the triplet always contains an informative negative, either hard (same class, dissimilar) or semi-hard (random, diverse).

#### Efficiency Considerations.

To increase dataset coverage and avoid bias, triplet mining is executed over multiple passes. A limit of 20 triplets per anchor image is enforced to balance memory usage and difficulty diversity.

### 3.4.3 Triplet Dataset Construction and Loading

To enable efficient model training using triplet loss, the system constructs a TensorFlow-based pipeline that wraps anchor, positive, and negative samples into a batched dataset structure. This format allows high-throughput, GPU-accelerated loading and transformation of training examples.

For each CNN-based embedding model—including ResNet-50, ResNet-101, MobileNetV2, and Xception—a dedicated triplet generation phase is performed. In this phase, embeddings are extracted from object crops using the respective pre-trained backbone with frozen weights. Based on similarity scores computed in the embedding space, triplet samples are created and categorized into training and validation sets.

The training triplets are derived from the training portion of the dataset, whereas the validation triplets are constructed exclusively from the validation split. This separation ensures a clear distinction between learning and evaluation phases, promoting reliable generalization assessment. Once generated, these triplets are serialized and stored as NumPy arrays on disk for future reuse.

During model training, the serialized datasets are loaded into memory and passed through a modular pipeline that performs the following operations:

- Anchors, positives, and negatives are stacked into tensor groups with consistent shape.
- The training dataset is shuffled to promote randomization and prevent memorization.
- The validation dataset is kept unshuffled to maintain consistency during evaluation.
- All datasets are batched to ensure memory efficiency and accelerated computation.

This strategy ensures that each backbone model receives its own triplet dataset tailored to the structure of its embedding space. By decoupling the triplet generation process from the training loop, the system achieves both high flexibility and scalability, supporting consistent experimentation across different CNN architectures.

#### Triplet Generation Statistics

To support comparative training across architectures, separate triplet datasets were generated for each CNN-based embedding model. The following table summarizes the total number of training and validation triplets created for each model:

| Model | Training Triplets | Validation Triplets |
|---|---|---|
| ResNet-50 | 9,184 | 1,428 |
| ResNet-101 | 9,094 | 2,691 |
| MobileNetV2 | 8,860 | 990 |
| Xception | 8,005 | 1,486 |

Table 3.3: Summary of training and validation triplet counts generated for each CNN-based embedding model.

These model-specific triplet sets form the basis for the first phase of training described in Section 3.10. During this phase, the frozen backbones of each CNN architecture are used to train the projection layers using the generated triplets. The corresponding validation triplets enable performance monitoring and early stopping, ensuring that each embedding model is optimized under consistent and architecture-specific training conditions.

#### Visualization of Training and Validation Triplets

To qualitatively inspect the effectiveness of our triplet generation strategy, we visualized a selection of anchor-positive-negative triplets sampled from both the training and validation datasets. These

visualizations provide intuitive insight into the learning objective and help verify the semantic and visual consistency of the generated triplets.

For each triplet:

- The **anchor** is a reference object crop from an image in the dataset.
- The **positive** is a visually and semantically similar object from a paired image.
- The **negative** is a visually distinct or semantically dissimilar object, which may belong to the same category or a different one, but exhibits low embedding similarity to the anchor.

These visualizations confirm that:

1. Positives maintain high intra-class similarity in shape, texture, or pose.
2. Negatives provide meaningful contrast by introducing subtle or significant variation, which encourages the network to learn more discriminative embeddings
3. No obvious annotation or sampling errors are present in the generated triplets.

The following figures display multiple triplets from the training and validation sets respectively, arranged in rows of three (anchor, positive, negative). This visual feedback is instrumental in validating the quality of the embedding training pipeline before proceeding to model training.

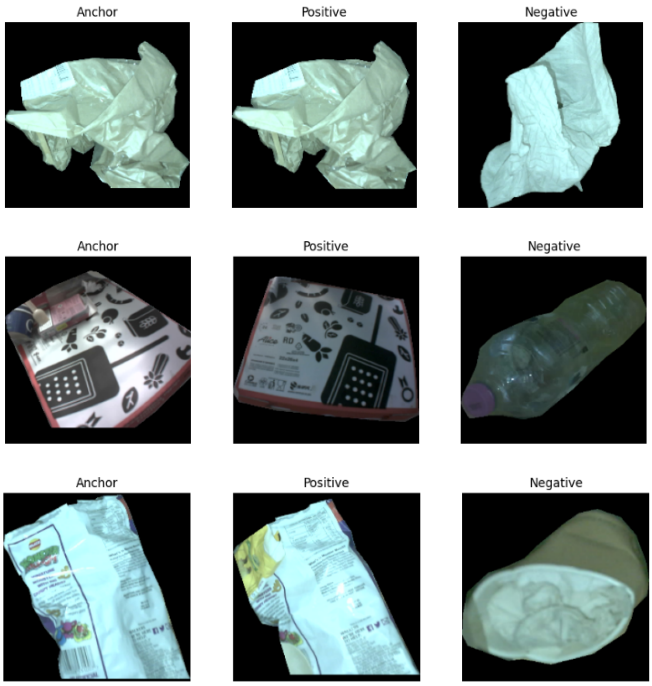



Figure 3.6: Visualization of three training triplets generated using ResNet-50 embeddings.

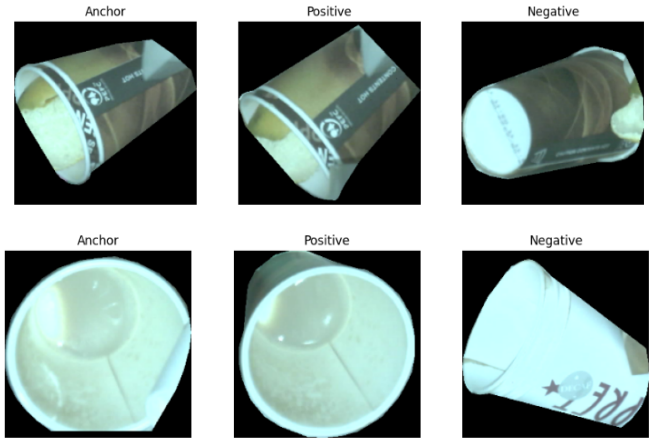



Figure 3.7: Visualization of two validation triplets constructed using ResNet-50 embeddings

### 3.4.4 Cosine-Based Triplet Loss

While the standard triplet loss is commonly defined using Euclidean (L2) distance, our implementation adopts a cosine similarity-based variant to better capture angular relationships between embeddings. This approach is particularly effective when using L2-normalized vectors, as it emphasizes directional alignment rather than magnitude.

To optimize the embedding space such that semantically similar objects are positioned closer together and dissimilar ones farther apart, we use a margin-based triplet loss defined over cosine similarity. This loss formulation is consistently applied throughout both Phase 1 and Phase 2 training stages (see Section 3.10), where it serves as the primary optimization objective for learning robust and discriminative object embeddings.

Given an anchor sample $A$, a positive sample $P$ (same category), and a negative sample $N$ (different or dissimilar object), the objective is to satisfy:

$$\mathcal{L}_{\text{triplet}} = \max\left(0,\ \alpha - \text{sim}(A, P) + \text{sim}(A, N)\right)$$

where:

- $\text{sim}(A, P)$ and $\text{sim}(A, N)$ represent cosine similarity scores,
- $\alpha$ is the enforced margin (empirically set to 0.3 in our implementation).

## 3.5 Fine-Tuning Step

Building on the foundations of transfer learning, we initialize our models with strong visual priors using pre-trained backbones, ResNet-50, ResNet-101, MobileNetV2 and Xception. In the second training phase, we selectively unfreeze and fine-tune the deeper layers of each architecture to better adapt to the object-level challenges present in our dataset, such as partial occlusions, ambiguous visual similarity, and high clutter within disposal environments.

For the ResNet-based models, this includes unfreezing layers within the `conv4_x` and `conv5_x` blocks, which are responsible for capturing high-level semantic features. In MobileNetV2, we fine-tune the final inverted residual blocks—specifically Blocks 14 to 16. For the Xception architecture, we target Blocks 11 to 14, which correspond to the latter stages of the Middle Flow and the entire Exit Flow. These fine-tuning choices are discussed in greater detail in Section 3.10.

To effectively train these layers and expose the models to realistic object variation, we apply an enhanced triplet mining strategy in combination with strong image augmentation. This strategy

dynamically generates new, diverse triplets during training, simulating fresh and variable conditions across epochs. Such an approach allows the network to learn from a more representative distribution of intra-class variation and inter-class contrast.

The specific methodology used for sampling and augmenting triplets in this fine-tuning phase is described in the sections that follow.

### 3.5.1 Enhanced Triplet Generation for Fine-Tuning

During the second phase of training, the triplet generation process was refined to support robust metric learning under realistic and challenging visual conditions. The improvements were introduced to enhance intra-class flexibility, harden inter-class discrimination, and better simulate environmental clutter. Below, we outline the key enhancements individually:

- **Positive Sampling Strategy:** The threshold for accepting a sample as a positive was reduced from 0.67 to 0.6. This relaxation allowed the inclusion of more varied intra-class examples—objects that, while semantically similar, may differ in appearance due to real-world factors such as lighting shifts, partial deformation, or pose variation. A positive was selected from the second image of a pair if it:
  - Shared the same `category_id` as the anchor,
  - Had a cosine similarity score $\geq 0.6$ with the anchor's embedding,
  - Had not already been selected for another triplet in the same batch.

  This selection strategy promotes intra-class diversity within training batches and improves the model's ability to learn consistent embeddings across varying visual appearances.

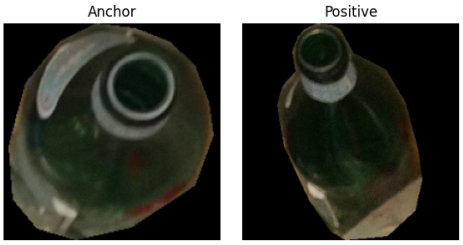



Figure 3.8: Example of anchor–positive pair

- **Negative Sampling Strategy:** To enforce a more discriminative embedding space, we adopted a hard negative mining policy. A candidate negative was selected from the second image of a pair if it:
  - Belonged to the same `category_id` as the anchor,
  - Had a cosine similarity score $\leq 0.5$ with the anchor's embedding,
  - Represented a distinct object not previously selected in the current batch.

  These candidates are considered "hard" due to their visual closeness yet semantic dissimilarity. If no valid hard negative was found, a fallback strategy selected a random negative from a different category and image.

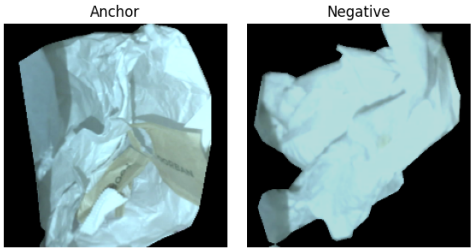



Figure 3.9: Example of a hard negative sample.The anchor and negative objects both belong to the `paper` category but exhibit distinct shapes and textures. Despite their semantic dissimilarity, their visual resemblance leads to a low cosine similarity score ($\leq 0.5$), qualifying this pair as a hard negative.

- **Occlusion Inclusion:** The fine-tuning phase maintained this policy by continuing to include partially occluded objects as valid anchors and positives. This choice reflects a deliberate design decision to simulate real-world visual clutter common in waste bin environments, where objects are frequently overlapped or partially hidden. Allowing occlusion enhances the model's robustness and its ability to learn embeddings that generalize under imperfect visibility conditions.

## Triplet Dataset for the Fine-Tuning step

Triplets were constructed from temporally adjacent image pairs in the dataset. For each anchor object detected in the first image of a pair, a positive and a negative crop were selected based on predefined sampling strategies. This process was iteratively applied across all valid object instances, with a cap of 15 triplets per image pair to ensure sample diversity and avoid redundancy.

Once generated, the triplets were integrated into the training pipeline, where they were dynamically batched, augmented, and passed through the network for optimization. Separate triplet datasets were created for each backbone model: training triplets—extracted from the training split—were used to fine-tune the deeper layers of the models, while validation triplets—generated from the validation split—served for performance monitoring during training. The following table summarizes the number of training and validation triplets produced per model:

| **Model** | **Training Triplets** | **Validation Triplets** |
|---|---|---|
| ResNet-50 | 10,606 | 2,652 |
| ResNet-101 | 10,607 | 2,652 |
| MobileNetV2 | 10,072 | 2,518 |
| Xception | 10,286 | 2,572 |

Table 3.4: Number of triplets generated for each model, split by training and validation sets.

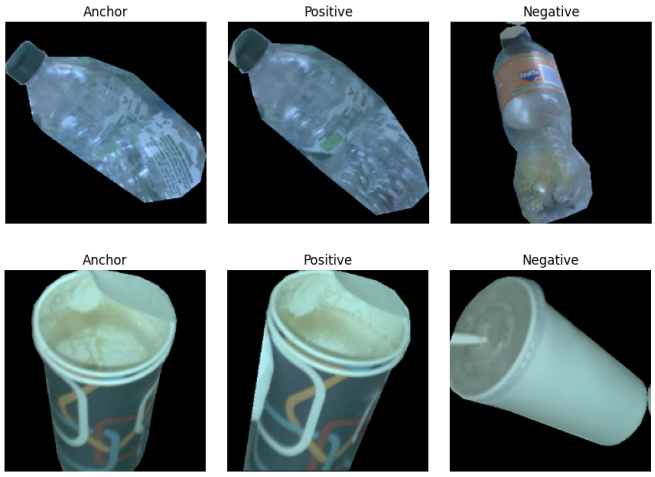



Figure 3.10: Enhanced triplet examples used in Fine-Tuning step

## 3.6 Augmentation

In deep metric learning, particularly when using triplet loss, data augmentation is not merely a technique for increasing data volume—it is a foundational strategy for improving model robustness, enhancing generalization, and simulating real-world object variability. The effectiveness of an embedding model heavily depends on its ability to remain invariant to superficial changes in input data while still distinguishing between truly distinct classes.

### General Importance of Augmentation in Deep Learning

Augmentation introduces controlled randomness to training data, generating realistic variations in object appearance. These variations—such as geometric transformations, occlusions, or lighting shifts—force the model to extract more abstract, invariant features rather than memorizing fixed spatial or color patterns. This mechanism acts as a regularizer, reducing overfitting, improving generalization, and enabling more stable convergence during training. These effects have been extensively validated in computer vision research [50].

### Augmentation Relevance in Triplet Loss Framework

In contrastive and triplet-based metric learning, the network is trained to compare image embeddings rather than predict fixed labels. Each training step presents an anchor image, a positive sample (same class), and a negative sample (different class). If the anchor and positive appear too similar (e.g., same lighting, orientation), the learning signal becomes weak and may lead to collapsed embeddings. Conversely, if negatives are overly perturbed, they risk overlapping with the positives semantically.

Therefore, a well-balanced augmentation policy is essential:

- **Enhancing generalization:** Strong augmentations prevent the network from learning trivial correlations such as background, color tint, or camera angle. This ensures that embeddings are shaped by semantic content rather than pixel-level features [50].
- **Simulating intra-class variability:** In real-world scenarios, objects from the same class often differ due to rotation, compression, lighting, or partial occlusion. If a model has only seen unvaried examples, it may incorrectly treat these variations as inter-class dissimilarity. By augmenting anchor and positive samples with such transformations, the model learns to encode class-level consistency despite these distortions.

### 3.6.1 Augmentation Strategy: Motivation and Design

As observed in the raw data samples (see section 3.1), our dataset contains significant intra-class variation caused by:

- Severe object rotation due to camera angle or disposal orientation.
- Spatial translation within the bin and compression from surrounding waste.
- Occlusions due to overlap between waste items or bin edges.
- Variations in lighting and plastic transparency, altering visual appearance.

These challenges make it difficult for a non-augmented model to generalize well. Thus, to simulate these conditions and increase tolerance to such distortions, we integrate synthetic transformations as part of the triplet generation process.

Overall, our augmentation policy is both empirically motivated by visual inspection of the dataset and theoretically grounded in deep metric learning literature. The approach ensures that embeddings for similar objects remain consistent under variation while reinforcing contrast for dissimilar ones, ultimately improving the discriminative power of the learned representation space.

To implement a robust augmentation strategy tailored to our task, we divide the transformations into two separate policies:

- **Advanced augmentations** are applied to *anchor* and *positive* samples to simulate realistic intra-class variation.
- **Minor augmentations** are applied to *negative* samples to introduce slight variability without compromising semantic distinctiveness.

The following subsections detail the specific augmentations used for each case.

#### Anchor and Positive Augmentation: Advanced Transformations

To simulate real-world intra-class variation and enhance the model's robustness, we apply a set of advanced augmentations to both anchor and positive samples during triplet training. These transformations are designed to introduce controlled perturbations that reflect common sources of visual distortion observed in the dataset, such as object rotation, spatial shifts, illumination changes, scale variations, and occlusion. The augmentations are implemented using a combination of OpenCV and NumPy within a TensorFlow-compatible preprocessing pipeline.

The augmentation function introduces the following transformations sequentially:

- **Rotation:** Each input image is rotated by a random angle sampled uniformly between $-50°$ and $+50°$. This simulates disposal scenarios where objects appear in arbitrary orientations within the bin.
- **Shift (Translation):** To account for spatial displacements within the bin, images are shifted randomly along both the horizontal and vertical axes by up to $\pm 35$ pixels. The transformation matrix is generated using NumPy and applied via OpenCV's affine warping.
- **Brightness Adjustment:** Lighting conditions inside disposal environments often vary due to reflective surfaces or shadows. We simulate this by scaling the pixel intensity values by a factor between 0.7 and 1.3. This operation ensures variability in photometric appearance while preserving semantic content.

- **Zooming:** The object scale is randomly adjusted between 0.7× (zoom-out) and 1.4× (zoom-in). Depending on the zoom factor, the resulting image is either padded (for zoom-out) or center-cropped (for zoom-in) to restore the original dimensions. This step allows the model to become invariant to object size changes due to proximity or camera viewpoint.

- **Occlusion:** With a 50% probability, a random rectangular patch is blacked out to simulate partial occlusion. The size of the occlusion ranges from 20 pixels to half the image width and height, and its position is randomly selected. This simulates realistic scenarios where objects are partially covered by other items or obstructed by bin edges.

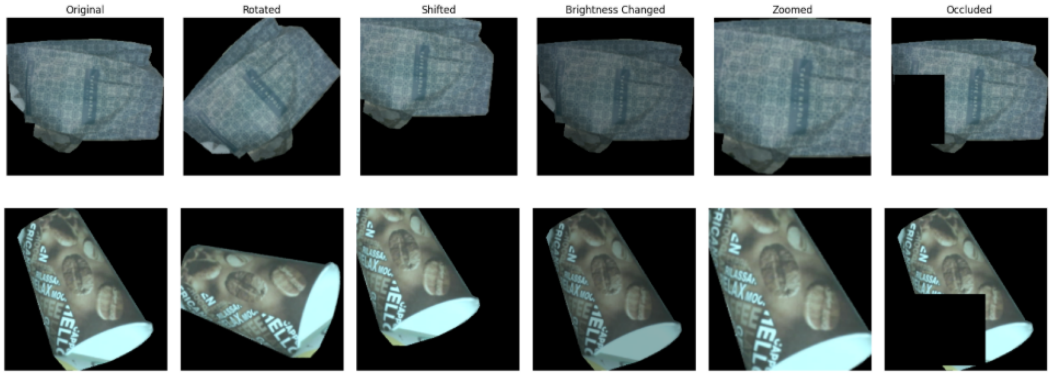



Figure 3.11: Illustration of advanced data augmentations applied to anchor and positive samples. Each row displays a distinct object crop undergoing the five transformations: rotation, translation, brightness adjustment, zooming, and occlusion.

The transformations described above are not applied in isolation but are executed sequentially in a combined augmentation pipeline. By applying these operations together, we emulate the complex and compounding effects that often occur in real disposal bins—where objects can be randomly oriented, partially hidden, variably illuminated, and distorted due to environmental or physical interaction. However, while such aggressive augmentation increases diversity and promotes generalization, it also introduces the risk of semantic distortion if pushed too far. Excessive transformations—such as extreme occlusion, over-rotation, or low-brightness clipping—can cause the altered sample to deviate significantly from its original semantic identity. To mitigate this, each augmentation is bounded within empirically defined limits, ensuring that the transformed samples remain visually realistic and semantically consistent with their original category. This balance enables the network to learn robust embeddings without introducing label noise or collapsing intra-class similarity.

## Anchor and Negative Augmentation: Subtle Variations for Semantic Integrity

In contrast to anchor and positive samples, negative images in triplet learning represent semantically dissimilar instances. The goal during training is to ensure that these negatives remain sufficiently distinct in the embedding space. Excessive or aggressive augmentation on negative samples could distort their appearance to the point where they may unintentionally resemble positives, leading to embedding collapse or ambiguous gradients.

To mitigate this, we employ a minimal augmentation policy for negatives. This implementation designed to simulate mild variations without altering the fundamental identity of the object.

The two augmentation steps applied are:

- **Shift (Translation):** A mild translation is applied by randomly shifting the image horizontally and vertically by up to ±10 pixels. This low-magnitude transformation introduces spatial diversity without deforming the object or affecting its semantic identity.

- **Brightness Adjustment:** The image brightness is scaled by a small factor between 0.8 and 1.2. This accounts for natural variation in lighting across scenes while keeping the object easily recognizable as a distinct instance. The intensity scaling is clipped to maintain valid pixel values within the 8-bit range $[0, 255]$.

These restrained augmentations maintain the visual realism and semantic integrity of negative samples while still contributing a slight stochastic element to the training process. This design choice prevents overfitting while safeguarding the core triplet loss objective: ensuring that anchor–negative distances remain significantly larger than anchor–positive distances.

Overall, by adopting this asymmetric augmentation strategy -strong perturbations for anchor-positive pairs and mild noise for negatives - we preserve the contrastive nature of the loss while enhancing the robustness of the embedding space. This approach aligns with empirical best practices in contrastive and metric learning [18], [47].

#### Augmentation Implementation Pipeline

All augmentations are applied dynamically during training, rather than precomputed and stored. This on-the-fly strategy introduces fresh transformations at every epoch, improving generalization and reducing memory overhead. By avoiding static augmented datasets, the model is continually exposed to diverse input conditions throughout training.

Such online augmentation guarantees higher data diversity without increasing dataset size or memory footprint, and supports better convergence behavior during metric learning.

### 3.6.2 Overview of Triplet Construction and Augmentation Policies

To summarize the fine-tuning strategy adopted in this work, we present a consolidated overview of the triplet construction logic, along with the augmentation policy applied to each component of the triplet. The goal is to concisely outline how anchor, positive, and negative samples were selected and transformed, and how these combined strategies support robust deep metric learning.

| Triplet Role | Selection Criteria | Augmentations |
|---|---|---|
| **Anchor** | Random object from the first image of a pair. | **Advanced** |
| **Positive** | Object from the second image with the same `category_id` and cosine similarity $\geq 0.6$. | **Advanced** |
| **Negative (hard)** | Same `category_id`, cosine similarity $\leq 0.5$, not previously used in batch. | **Minor** |
| **Negative (semi-hard)** | Different category or image (used only if no valid hard negative is found). | **Minor** |

Table 3.5: Triplet roles, selection criteria, and augmentation policy used during fine-tuning.

The augmented triplets generated using this combined sampling and transformation strategy were directly employed for fine-tuning the deeper layers of each backbone model, as described in Section 3.10. These triplets served as the core input to the Phase 2 training process, providing enriched and semantically controlled examples to improve the network's ability to generalize across intra-class variability and inter-class similarity.

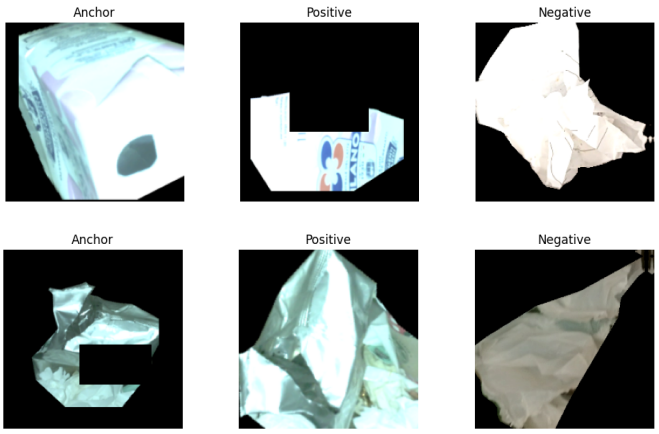



Figure 3.12: Samples of Augmented Triplets Used for training the second phase

## 3.7 Matching Process

After generating embeddings for object instances in a temporally aligned image pair, the system aims to identify semantic correspondences between these objects. This process is crucial for detecting whether specific objects persist, disappear, or newly appear across frames. The matching pipeline is composed of three key stages:

1. Computing cosine similarity between embedding vectors.
2. Enforcing category consistency during pairwise matching.
3. Identifying unmatched instances in the second image as added objects.

The following subsections provide a detailed account of each of these stages.

### 3.7.1 Matching Algorithm

To ensure semantic correctness in the matching process, the system imposes an additional constraint based on object class labels. Specifically, for a candidate match to be considered valid, both objects must share the same category. This prevents the erroneous pairing of visually similar yet semantically distinct objects—such as confusing a plastic cup with a bottle—by ensuring that only category-consistent comparisons are allowed.

Once the cosine similarity matrix is computed, the system performs pairwise matching using a greedy best-match strategy. The algorithm proceeds as follows:

- For each object in the first image (represented as a row in the similarity matrix), the algorithm identifies the object in the second image (column-wise) that satisfies the following conditions:
  - The category id values for both objects are identical.
  - The cosine similarity score exceeds a predefined threshold of 0.66.
  - The candidate object from the second image has not already been assigned to a previous match.
- If such a match is found, the corresponding pair of indices is recorded, and both objects are marked as matched to prevent duplicate assignments.

This one-to-one matching logic guarantees that each object in the second image is matched to at most one object in the first image, avoiding redundant pairings. The output of the algorithm consists of:

- **Matched Pairs:** A list of index tuples `(i, j)` where object $i$ from the first image is matched to object $j$ in the second image.
- **Unmatched in Image A:** A list of indices for objects in the first image that did not find any valid match.
- **Unmatched in Image B:** A list of indices for objects in the second image that were not matched by any object in the first image.

The choice of 0.66 as the cosine similarity threshold was empirically determined based on validation experiments, striking a balance between match precision and coverage. Matches below this threshold were frequently observed to be visually ambiguous or semantically inconsistent. As a result, this cutoff serves to reject low-confidence pairings while preserving high-quality matches.

This algorithm forms the foundation for the system's object-level change detection logic. By explicitly aligning object instances across temporally adjacent image frames, it enables the detection of both matched and unmatched instances—information that is essential for identifying added or reconfigured objects in cluttered scenes.

### 3.7.2 Detection of Added Objects

Once object-level correspondences have been established between two temporally adjacent images, the system identifies newly introduced objects—i.e., those present in the second image but unmatched in the first. This is achieved by analyzing the unmatched indices resulting from the cosine similarity and category-aware matching process described in Subsection 3.7.1.

Let $U_B$ denote the list of unmatched object indices from the second image $B$, as determined by the matching algorithm. The total number of added objects is computed as:

$$N_{\text{added}} = |U_B| \tag{3.3}$$

This represents the cardinality of the unmatched set in image $B$ and directly quantifies objects that appear in $B$ without a valid match in image $A$. These objects are inferred as newly added instances in the visual scene.

## 3.8 Inference Visualization

To qualitatively assess the output of our trained CNN models, we present an example pair of real-world waste-bin images, along with their detected object crops and instance-level comparison results. This visualization illustrates how the system identifies semantically matching objects and flags newly added items based on cosine similarity and class alignment.

The objective is to showcase the system's ability to track object-level changes across time-separated snapshots, validating its effectiveness beyond numerical evaluation.


Figure 3.13: Full-resolution input images used for evaluation with ResNet-50 after Phase 2 training. Image A (left) and Image B (right) are shown side-by-side for visual comparison.

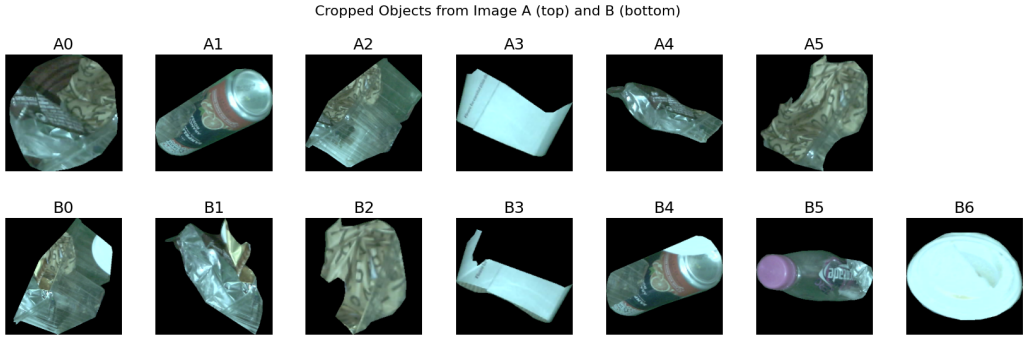



Figure 3.14: Detected object crops from Image A (top row, A0–A5) and Image B (bottom row, B0–B6).

**Phase 2 Output (ResNet-50):**

- **Matched Pairs:** 5
- **Matches:** A1 → B4, A2 → B0, A3 → B3, A4 → B1, A5 → B2
- **Unmatched in B:** (B5, B6)
- **Added Objects in B:** 2

Using the full-resolution input pair shown in Figure 3.13, the system detected **six** object instances in Image A (labeled A0–A5) and **eight** in Image B (labeled B0–B7). The corresponding cropped objects are shown in Figure 3.14.

Based on cosine similarity scores (Section 3.3), the ResNet-50 embedding model—after Phase 2 fine-tuning—identified **five matched object pairs**:

- A1 → B4
- A2 → B0
- A3 → B3
- A4 → B1
- A5 → B2

These pairs were selected as they exhibit the highest cosine similarity values among all cross-image object pairs, and they belong to the same predicted category, satisfying both the similarity and class consistency constraints defined in our matching process (see Section 3.7).

Object A0 from Image A remained unmatched, indicating it likely no longer appears in Image B. Conversely, objects B5 and B6 had no sufficiently similar counterpart in Image A. As a result, the system classified them as **added objects**—i.e., newly introduced items between frames.

This example illustrates how the trained model effectively captures object-level continuity and change, enabling both matching and added-object detection via learned embedding comparisons.

This example illustrates how the trained model effectively captures object-level continuity and change, enabling both matching and added-object detection via learned embedding comparisons.

### Cross-Model Comparison Samples

To complement the detailed single-pair analysis, we include two qualitative examples that compare the outputs of different CNN backbones. These examples use simplified outputs that summarize the number of **matched objects** and **added objects** detected in Image B. While the full matching pipeline operates at the instance level using embeddings and cosine similarity, these summaries offer an interpretable view of how each model generalizes across different bin conditions.

Each pair includes: (1) the original input images, and (2) a corresponding results comparing matched and added counts for all models after Phase 2 or in the zero-shot setting.

```
=== Comparison Summary ===
Phase 2 ResNet-50: Matched = 5 | Added in B = 4
Phase 2 ResNet-101: Matched = 3 | Added in B = 6
Phase 2 MobileNetV2: Matched = 4 | Added in B = 5
Phase 2 Xception: Matched = 4 | Added in B = 5
```

Figure 3.15: Model-wise output summary on a single image pair: matched and added objects detected by each CNN after Phase 2 training. The ground truth number of added objects is 4, which was correctly predicted only by the ResNet-50 model.

```
=== Comparison Summary ===
Pre-Trained ResNet-50: Matched = 4 | Added in B = 8
Pre-Trained ResNet-101: Matched = 2 | Added in B = 10
Pre-Trained MobileNetV2: Matched = 1 | Added in B = 11
Pre-Trained Xception: Matched = 1 | Added in B = 11
```

Figure 3.16: Full-resolution input images (top) and comparative performance summary (bottom) for a challenging bin scenario evaluated with pre-trained CNN models. The chart highlights the limitations of zero-shot inference, showing that pre-trained models struggle with object-level change detection and require task-specific training to improve performance.

## 3.9 Visualizing Siamese Model Predictions

To qualitatively evaluate the output of the Siamese network, we visualize example pairs of input images alongside their predicted similarity scores. This helps illustrate how the model distinguishes between similar and dissimilar trash bin scenes.

### Example Prediction: Dissimilar Pair

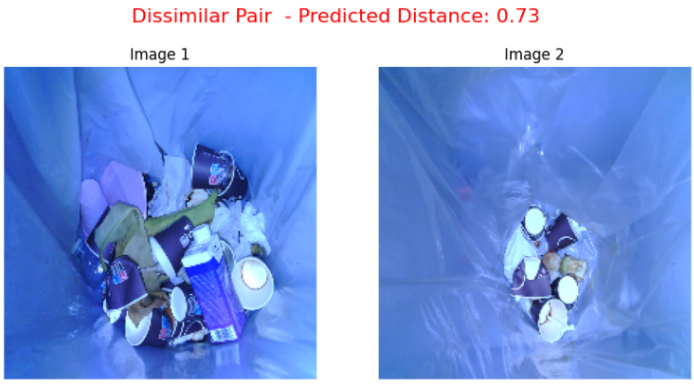



Figure 3.17: **Dissimilar Pair – Predicted Distance: 0.73.** The Siamese network correctly identifies this pair as dissimilar, assigning a high distance score due to significant differences in object content between the two trash bin scenes. Several items appear to have been removed.

### Example Prediction: Similar Pair

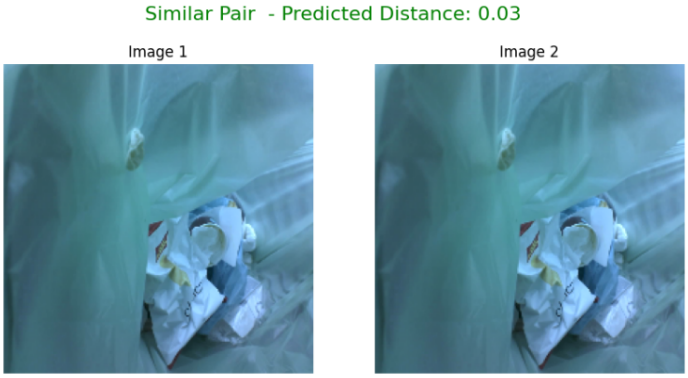



Figure 3.18: **Similar Pair – Predicted Distance: 0.03.** The Siamese model accurately detects high similarity in this pair, assigning a very low distance score. The object configuration is nearly identical across both images, indicating minimal or no change.

## 3.10 Training Strategy

This section presents the complete training methodology adopted for evaluating the object-level similarity task using both Siamese and Triplet-based learning paradigms. We describe how each convolutional backbone—ResNet-50, ResNet-101, MobileNetV2, and Xception—was integrated into a unified framework and progressively trained under consistent experimental settings. This part begins with a summary of the software libraries and hardware environment used to ensure reproducibility. Then, we detail the training procedures applied to each model.

### System and Software Environment

This project was implemented using Python and several widely adopted deep learning libraries. TensorFlow 2.17.0 served as the primary framework for model definition, training, and evaluation. Data manipulation and analysis were facilitated by NumPy 1.26.4 and Pandas 2.2.3. Scikit-learn 1.6.1 was used for evaluation metrics and preprocessing tasks, while Matplotlib 3.9.2 supported result visualization.

All experiments were conducted on a single NVIDIA Tesla V100 GPU with 16 GB of VRAM, using CUDA 12.2 and NVIDIA driver version 535.216.03 to accelerate training. This consistent environment ensured reliable and efficient computation throughout all training phases.

| Tool / Component | Description and Version |
|---|---|
| **Python** | General-purpose programming language for scripting and implementation (v3.10.14) |
| **TensorFlow** | Deep learning framework used for model construction and training (v2.17.0) |
| **NumPy** | Library for numerical operations and matrix manipulation (v1.26.4) |
| **Pandas** | Data analysis and tabular processing library (v2.2.3) |
| **Scikit-learn** | Toolkit for metrics, preprocessing, and ML utilities (v1.6.1) |
| **Matplotlib** | Visualization library for generating figures (v3.9.2) |
| **GPU** | NVIDIA Tesla V100 (16 GB VRAM), used for hardware acceleration |
| **CUDA Toolkit** | NVIDIA CUDA version 12.2 for GPU computation |
| **Driver Version** | NVIDIA Driver v535.216.03 |

Table 3.6: Summary of tools, libraries, and hardware environment used throughout this work.

In the following subsections, we present the detailed training configurations and phase-specific strategies applied to each backbone model individually. Each model's behavior in Phase 1 and Phase 2 is analyzed to enable a fair and structured comparison.

### 3.10.1 Siamese Network

In this part, we outline the implementation details of our Siamese network for object-level similarity learning. We describe the dataset preparation, pair generation, image preprocessing, model architecture, loss function, training procedure, and model persistence.

#### Dataset Preparation and Pair Generation

To train the Siamese network, we require pairs of images labeled as 'similar' (coming from the same class) or 'dissimilar' (from different classes). A Python generator constructs these training pairs dynamically during runtime:

- With 50% probability, it randomly selects two distinct images from the same class to form a *positive* pair (label = 0).
- Otherwise, it selects one image from one class and another from a different class to form a *negative* pair (label = 1).

By sampling pairs dynamically, we avoid storing a combinatorial number of pairs on disk and ensure a balanced mix of similar and dissimilar examples during each epoch.

#### Siamese Network Architecture

Our Siamese network consists of two identical branches sharing weights, followed by a distance computation layer.

Each branch processes an input image through:

1. A $7 \times 7$ convolution with 64 filters (stride 2), batch normalization, ReLU activation, and $3 \times 3$ max pooling (stride 2).

2. Two successive $3 \times 3$ convolutions (64 filters), each followed by batch normalization and ReLU.
3. A residual shortcut connection added after the second $3 \times 3$ convolution, followed by ReLU.
4. Global average pooling to produce a 2048-dimensional feature vector.
5. A dense projection to 128 dimensions (linear activation).
6. **L2 normalization** to project embeddings onto the unit hypersphere.

#### Distance Layer

We compute the Euclidean distance between the two 128-dimensional embeddings using a custom Euclidean Distance layer:

$$d = \sqrt{\max\big(\|\mathbf{e}_1 - \mathbf{e}_2\|_2^2,\, \varepsilon\big)}.$$

An infinitesimal constant $\varepsilon$ prevents numerical instability near zero.

#### Loss Function

We employ the contrastive loss function:

$$L = \frac{1}{N} \sum_{i=1}^{N} \Big[ (1 - y_i)\, D_i^2 + y_i\, \max(0, m - D_i)^2 \Big], \tag{3.4}$$

where $D_i$ is the Euclidean distance for pair $i$, $y_i = 0$ for similar and $y_i = 1$ for dissimilar pairs, and $m = 1.0$ is the margin.

#### Optimization and Training

The model is compiled with the Adam optimizer and the custom contrastive loss. During training, the model is trained for 10 epochs with 100 steps per epoch. Balanced batches of positive and negative pairs ensure stable convergence. We monitor the validation loss and employ early stopping to prevent overfitting.

### 3.10.2 Triplet-Based Training Strategy for CNN Backbones

In this section, we present the training methodology adopted for evaluating the object-level similarity performance of the four backbone architectures introduced earlier in Section 2.5: ResNet-50, ResNet-101, MobileNetV2, and Xception. Each model was initially used as a frozen feature extractor in a zero-shot configuration to assess its generalization capability without any task-specific adaptation. Subsequently, all models underwent two progressive training phases—Phase 1 and Phase 2—under controlled and consistent experimental settings to enable fair comparison across architectures.

To ensure the validity and comparability of results, the same training pipeline and loss formulation were applied to each model. Specifically, Phase 1 involved training only the projection head while keeping the backbone frozen, and Phase 2 selectively fine-tuned deeper layers of the backbone while continuing to train the projection head. Both phases used the triplet loss formulation discussed in Section 3.4, with a fixed margin $\alpha = 0.3$ applied uniformly across all models. For all models and training phases, the optimizer used was *Adam*, chosen for its adaptive learning rate and stable convergence behavior. A learning rate of 1e−3 was used in Phase 1, while a reduced rate of 1e−5 was applied in Phase 2 to stabilize fine-tuning. Early stopping was employed across all phases to prevent overfitting and ensure generalization, and the best model weights (based on validation loss) were saved during training to preserve optimal performance checkpoints.

The following table summarizes the global training configurations, hyperparameter settings, and optimization strategies applied consistently across all models in both Phase 1 and Phase 2.

| Parameter | Value Used in All Models |
|---|---|
| Optimizer | Adam |
| Phase 1 Learning Rate | 1e−3 |
| Phase 2 Learning Rate | 1e−5 |
| Triplet Loss Margin | 0.3 |
| Early Stopping | Enabled (patience = 6) |
| Best Weight Checkpoint | Enabled (based on validation loss) |

Table 3.7: Global training configuration applied uniformly across all models in Phase 1 and Phase 2.

### 3.10.3 Pretrained ResNet-50 as a Frozen Feature Extractor (Zero-Shot Baseline)

Before initiating any training or adaptation, a baseline experiment was conducted to evaluate the zero-shot performance of ResNet-50 on the object matching task. In this setup, the ResNet-50 model was loaded with pretrained weights from ImageNet, with its entire architecture left completely frozen. No projection head was added, and no fine-tuning or training was performed at this stage.

The model was used solely as a feature extractor. The cropped object images used in this evaluation were obtained using the segmentation-guided cropping pipeline described in Section 3.2. Each cropped object was passed through the frozen ResNet-50, and the resulting feature embeddings were used for similarity comparison without any task-specific training.In this configuration, the model's output corresponds to the 2048-dimensional feature vector produced by the final global average pooling (GAP) layer of ResNet-50.

To perform object matching between image pairs, cosine similarity was computed between the extracted embeddings. As detailed in Section 3.3, this similarity metric focuses on angular distance and benefits from an $L_2$ normalization step applied before comparison. However, it is important to note that this normalization occurs only during the matching phase, and not inside the model itself—since no training or embedding head was applied.

This zero-shot baseline serves as a reference point to assess how effectively a large pretrained convolutional model generalizes to a domain-specific matching task without any adaptation. The evaluation results for this baseline are reported in Section 4.3.

### 3.10.4 Phase 1 of ResNet-50: Frozen Backbone with Trainable Projection Head

As previously described in Section 2.5.1, ResNet-50 consists of a deep convolutional backbone composed of residual bottleneck blocks from `conv1` through `conv5_x`. To initialize a strong baseline for metric learning, the first training phase leverages a **frozen ResNet-50 backbone pre-trained on ImageNet**. In this configuration, the convolutional layers are preserved in their pretrained state and are excluded from gradient updates during training. This approach capitalizes on the rich visual representations already learned on large-scale classification tasks, while significantly reducing the number of trainable parameters and the risk of overfitting on the comparatively smaller dataset used in this study.

To adapt ResNet-50 for embedding generation, **the classification-specific layers are removed when loading the model**. **This excludes both the global average pooling layer and the final fully connected softmax classifier**, thereby retaining only the convolutional

feature extractor. The output of this backbone is a $7 \times 7 \times 2048$ feature map from the final residual stage.

This feature map is passed through a custom projection head composed of four layers:

- A **Global Average Pooling** layer compresses the spatial dimensions, yielding a 2048-dimensional vector that globally summarizes the feature map.
- A **Dropout** layer with a dropout rate of 0.2 is applied for regularization during training, mitigating overfitting in the projection head.
- A **Dense** (fully connected) layer projects the features into a 128-dimensional embedding space, producing an initial descriptor for similarity comparison.
- A $L_2$ **Normalization** layer constrains each 128-dimensional vector to have unit norm, projecting it onto the surface of a unit hypersphere.

The use of $L_2$ normalization aligns with the cosine similarity matching strategy detailed in Section 3.3. By enforcing unit norm constraints, the model ensures that similarity comparisons focus exclusively on the angular relationships between embedding vectors. This results in a geometrically regular hyperspherical embedding space that is highly effective for matching semantically similar object instances.

Overall, only the parameters in the Dense projection layer are updated during this phase, while the convolutional backbone remains static. This strategy ensures fast convergence and training stability, while producing a standalone embedding model whose performance is evaluated in comparison to other training stages.

The ResNet-50 embedding model was trained using triplet samples constructed according to the strategy detailed in Subsection 3.4.3. Specifically, a total of 9,184 training triplets and 1,428 validation triplets were generated for this backbone, as summarized in Table 3.3.

| **Layer Name** | **Output Shape** | **Trainable Params** | **Non-trainable Params** |
|---|---|---|---|
| ResNet-50 Backbone (fully frozen) | (None, 7, 7, 2048) | 0 | 23,587,712 |
| Global Average Pooling (GAP) | (None, 2048) | 0 | 0 |
| Dropout (rate = 0.2) | (None, 2048) | 0 | 0 |
| Dense (projection_dense) | (None, 128) | 262,272 | 0 |
| $L_2$ Normalization | (None, 128) | 0 | 0 |
| **Total** | — | **262,272** | **23,587,712** |

Table 3.8: Layer-wise parameter summary of the Phase 1 embedding model using a frozen ResNet-50 backbone. Only the projection head was trainable during this stage.

### 3.10.5 Phase 2 of ResNet-50: Layer-wise Fine-Tuning of Deeper Residual Blocks

To improve upon the baseline established in Phase 1, a second training stage was conducted in which selective fine-tuning was applied to the ResNet-50 backbone. As described in Section 2.5.1, the ResNet-50 architecture is composed of four residual stages: `conv2_x` through `conv5_x`, each consisting of multiple bottleneck residual blocks. In Phase 2, we unfreeze the deeper layers of the backbone to allow task-specific feature adaptation while retaining the generalization benefits of the earlier pretrained layers.

Specifically, only the convolutional layers within the `conv4_x` and `conv5_x` residual stages were unfrozen and made trainable. All earlier layers, including `conv1`, `conv2_x`, and `conv3_x`, remained frozen. This strategy is based on the intuition that high-level feature representations learned in

deeper stages are more relevant to the target domain, whereas early-layer filters often capture generic edge and texture information that generalize well across tasks. The projection head architecture from Phase 1 was retained without modification, consisting of a global average pooling layer, a dropout layer with rate 0.2, a dense layer projecting to a 128-dimensional embedding space, and an $L_2$ normalization layer. During training, the projection head and the unfrozen layers of the backbone were optimized using triplet loss.

A total of 13,258 augmented triplets—corresponding to 39,774 individual image crops—were generated for this fine-tuning phase using the enhanced sampling strategy described in Section 3.5. Of these, 80% (10,606 triplets; 31,818 images) were used for training, and 20% (2,652 triplets; 7,956 images) for validation. A detailed breakdown is provided in Table 3.6.2.

Overall, allowing gradient flow through selected residual stages, the network learns to refine its high-level representations to better capture the semantic nuances of object variation specific to the dataset. The earlier layers remain fixed to preserve stability and reduce overfitting risk.

| **Layer Name** | **Output Shape** | **Trainable Params** | **Non-trainable Params** |
|---|---|---|---|
| ResNet-50 Backbone (selectively unfrozen) | (None, 7, 7, 2048) | 22,084,608 | 1,503,104 |
| Global Average Pooling (GAP) | (None, 2048) | 0 | 0 |
| Dropout (rate = 0.2) | (None, 2048) | 0 | 0 |
| Dense (projection_dense) | (None, 128) | 262,272 | 0 |
| $L_2$ Normalization | (None, 128) | 0 | 0 |
| **Total** | — | **22,346,608** | **1,503,104** |

Table 3.9: Layer-wise parameter summary of the Phase 2 embedding model using ResNet-50 with fine-tuning applied to `conv4_x` and `conv5_x`.

### 3.10.6 Pretrained ResNet-101 as a Frozen Feature Extractor (Zero-Shot Baseline)

To explore the generalization capabilities of deeper convolutional backbones, a second zero-shot baseline experiment was conducted using ResNet-101. This model, like ResNet-50, was initialized with pretrained ImageNet weights but remained entirely frozen during the evaluation. No additional projection head was appended, and no parameter updates were performed. The resulting feature embeddings were 2048-dimensional, obtained after the final pooling layer, consistent with the ResNet-50 baseline.

This setup enables a direct comparison between shallower and deeper backbones in zero-shot conditions. By holding all other factors constant—input preparation, similarity metric, and downstream logic—this baseline serves to evaluate whether the increased representational depth of ResNet-101 leads to improved system performance without task-specific adaptation. Results from this experiment are reported in Section 4.3.

### 3.10.7 Phase 1 of ResNet-101: Frozen Backbone with Trainable Projection Head

To enable a fair comparison with the ResNet-50 baseline, a similar training procedure was applied to ResNet-101 in the first phase. The convolutional backbone was initialized with ImageNet-pretrained weights and kept entirely frozen during this stage, thereby preserving all learned filters across residual stages from `conv1` to `conv5_x`. This allows the model to function as a deep, fixed feature extractor while minimizing the risk of overfitting on the limited dataset.

Following the strategy described in Section 2.5.2, the frozen ResNet-101 backbone was coupled with the same projection head architecture used in the ResNet-50 setup. This includes a global

average pooling layer, dropout (rate = 0.2), a dense projection layer to 128 dimensions, and an $L_2$ normalization layer. As before, this setup supports cosine-based embedding comparisons, which are computed using angular distance between unit-normalized vectors (Section 3.3).

Notably, despite ResNet-101 being significantly deeper than ResNet-50, the projection head architecture and its trainable parameter count (262,272) remained exactly the same. This embedding model is trained using a total of 9,094 training triplets and 2,691 validation triplets generated specifically for the ResNet-101 backbone, as summarized in Table 3.3.

| **Layer Name** | **Output Shape** | **Trainable Params** | **Non-trainable Params** |
|---|---|---|---|
| ResNet-101 Backbone (frozen) | (None, 7, 7, 2048) | 0 | 42,658,176 |
| Global Average Pooling (GAP) | (None, 2048) | 0 | 0 |
| Dropout (rate = 0.2) | (None, 2048) | 0 | 0 |
| Dense (projection_dense) | (None, 128) | 262,272 | 0 |
| $L_2$ Normalization | (None, 128) | 0 | 0 |
| **Total** | — | **262,272** | **42,658,176** |

Table 3.10: Layer-wise parameter summary of the Phase 1 embedding model using a frozen ResNet-101 backbone. Only the projection head (Dense layer) was trainable during this stage.

### 3.10.8 Phase 2 of ResNet-101: Layer-wise Fine-Tuning of Deeper Residual Blocks

The second stage of training was conducted in which selective fine-tuning was applied to the deeper convolutional layers of the ResNet-101 backbone. The ResNet-101 introduces significantly more bottleneck blocks in the `conv4_x` stage compared to ResNet-50, resulting in a deeper and more expressive architecture. To exploit this potential while mitigating the risk of overfitting, only the layers corresponding to `conv4_x` and `conv5_x` were unfrozen and made trainable. All earlier layers (i.e., `conv1`, `conv2_x`, and `conv3_x`) remained frozen to retain generic low-level features.

The same projection head used in ResNet-50 Phase1 (Subsection 3.10.4) was retained here without modification. Joint optimization of the projection head and the deeper residual blocks (`conv4_x` and `conv5_x`) was performed using triplet loss.

In this phase, a total of 13,259 Augmented triplets were generated from annotated image pairs. These were divided into a training set of 10,607 triplets (80%) and a validation set of 2,652 triplets (20%), corresponding to 31,821 and 7,956 image crops respectively.

| **Layer Name** | **Output Shape** | **Trainable Params** | **Non-trainable Params** |
|---|---|---|---|
| ResNet-101 Backbone (selectively unfrozen) | (None, 7, 7, 2048) | 41,102,848 | 1,555,328 |
| Global Average Pooling (GAP) | (None, 2048) | 0 | 0 |
| Dropout (rate = 0.2) | (None, 2048) | 0 | 0 |
| Dense (projection_dense) | (None, 128) | 262,272 | 0 |
| $L_2$ Normalization | (None, 128) | 0 | 0 |
| **Total** | — | **41,365,120** | **1,555,328** |

Table 3.11: Layer-wise parameter summary of the Phase 2 embedding model using ResNet-101 with fine-tuning applied to `conv4_x` and `conv5_x`.

### 3.10.9 Pretrained MobileNetV2 as a Frozen Feature Extractor (Zero-Shot Baseline)

To further assess the efficacy of lightweight architectures under zero-shot conditions, MobileNetV2 was evaluated as a frozen feature extractor. Pretrained on ImageNet, the MobileNetV2 model was used in a fully frozen state, without further adaptation.

Input crops (Section 3.2), were individually passed through the MobileNetV2 backbone. Feature vectors were extracted after the final global average pooling (GAP) layer, resulting in 1280-dimensional descriptors for each object—lower in dimensionality compared to the 2048-dimensional outputs of ResNet-50 and ResNet-101.

This experiment enables a comparative analysis of compact versus deep backbones in zero-shot settings. While ResNet variants offer higher capacity, MobileNetV2 emphasizes efficiency, allowing investigation of the trade-off between model size and representation quality. Evaluation results are presented in Section 4.3.

### 3.10.10 Phase 1 of MobileNetV2: Frozen Backbone with Trainable Projection Head

The first training stage with MobileNetV2 follows a similar initialization strategy to that used in the ResNet-based configurations (Sections 3.10.4 and 3.10.7), where the convolutional backbone is retained in a frozen state while a task-specific projection head is trained for embedding generation. The goal remains consistent: to transform high-dimensional pretrained features into compact, semantically meaningful descriptors suitable for cosine similarity-based object matching.

In this configuration, MobileNetV2 serves purely as a frozen feature extractor. Its pretrained weights—learned on ImageNet—are preserved without modification to retain their general-purpose visual representations. The output of the backbone is a 1280-dimensional global feature vector, obtained via global average pooling over the final convolutional feature map. This vector is then passed through a custom projection head consisting of a Dropout layer (rate = 0.2), a dense layer that projects the feature vector to a 128-dimensional embedding space, and an $L_2$ normalization layer that ensures the embeddings lie on the unit hypersphere. This design is intentionally kept consistent with the projection heads used in ResNet-50, ResNet-101, and Xception Phase 1 models, ensuring a fair comparison across all architectures.

Notably, this configuration introduces a moderate number of trainable parameters (163,698), which is lower than the Phase 1 configurations of ResNet-50 and ResNet-101 (both with 262,272). This reduced capacity reflects the lightweight nature of the MobileNetV2 architecture, while still enabling effective embedding learning through the final dense transformation. A total of 8,860 training triplets and 990 validation triplets were used for this model, as summarized in Table 3.3.

| **Layer Name** | **Output Shape** | **Trainable Params** | **Non-trainable Params** |
|---|---|---|---|
| Input Layer (image_input) | (None, 224, 224, 3) | 0 | 0 |
| Preprocessing (mobilenet_preproc) | (None, 224, 224, 3) | 0 | 0 |
| MobileNetV2 Backbone (frozen) | (None, 1280) | 0 | 2,257,984 |
| Dense Layer (proj_dense2, 128 units) | (None, 128) | 163,968 | 0 |
| $L_2$ Normalization | (None, 128) | 0 | 0 |
| **Total** | — | **163,968** | **2,257,984** |

Table 3.12: Layer-wise parameter summary of the Phase 1 embedding model using a frozen MobileNetV2 backbone

### 3.10.11 Phase 2 of MobileNetV2: Layer-wise Fine-Tuning of Upper Inverted Residual Blocks

To enhance the domain-specific representational capacity of MobileNetV2 while maintaining computational efficiency, Phase 2 training selectively unfreezes only the top three inverted residual blocks—`block_14`, `block_15`, and `block_16`. These deeper blocks, located toward the end of the network hierarchy, are responsible for learning abstract and high-level semantic features. All earlier layers, from the initial convolution through block 13, remain frozen to retain general-purpose visual filters and avoid overfitting.

This approach is conceptually aligned with the strategy used for ResNet-50 and ResNet-101 Phase 2, where fine-tuning was confined to the last two residual stages. While ResNet variants operate on residual blocks across fixed stages (`conv4_x` and `conv5_x`), the equivalent in MobileNetV2 corresponds to the final bottleneck blocks.

The projection head from Phase 1—consisting of a Global Average Pooling layer applied to the backbone output, a dropout layer with a rate of 0.2, a single 128-dimensional dense layer, and an $L_2$ normalization layer—was retained and jointly trained alongside the unfrozen blocks.

In this phase, a total of 12,590 augmented triplets were used, with 10,072 (80%) allocated to training and 2,518 (20%) to validation. Since each triplet consists of an anchor, a positive, and a negative crop this results in 37,770 object crops used in the embedding model. A detailed summary of the triplet distribution is provided in Table 3.6.2.

| **Layer Name** | **Output Shape** | **Trainable Params** | **Non-trainable Params** |
|---|---|---|---|
| Input Layer (image_input) | (None, 224, 224, 3) | 0 | 0 |
| Preprocessing (mobilenet_preproc) | (None, 224, 224, 3) | 0 | 0 |
| MobileNetV2 Backbone | (None, 1280) | 1,113,920 | 1,144,064 |
| Dense Layer (projection_dense) | (None, 128) | 163,968 | 0 |
| $L_2$ Normalization | (None, 128) | 0 | 0 |
| **Total** | — | **1,277,888** | **1,144,064** |

Table 3.13: Layer-wise parameter summary of the Phase 2 embedding model using MobileNetV2 with fine-tuning applied to blocks 14–16 and a trainable projection head.

### 3.10.12 Pretrained Xception as a Frozen Feature Extractor (Zero-Shot Baseline)

In line with the previously described baselines, a fourth zero-shot experiment was performed using the Xception architecture, which was originally introduced as an Inception-inspired model with depthwise separable convolutions and residual connections [34]. It was initialized with pretrained ImageNet weights and used without modification or training.

Each input object was passed through the frozen Xception backbone. Feature descriptors were extracted after the final convolutional block, where the $10 \times 10 \times 2048$ activation map was compressed via a Global Average Pooling (GAP) operation, yielding 2048-dimensional vectors.

This setup mirrors the protocols used for ResNet-50, ResNet-101, and MobileNetV2, allowing direct comparisons across architectures under identical preprocessing and matching conditions. Also this baseline helps isolate the representational power of its depthwise separable convolutional design. The corresponding evaluation results are reported in Section 4.3.

### 3.10.13 Phase 1 of Xception: Frozen Backbone with Trainable Projection Head

In Phase 1, the Xception model was utilized as a fixed feature extractor, following the same protocol applied to ResNet-50, ResNet-101, and MobileNetV2. The convolutional backbone was initialized with ImageNet-pretrained weights and kept entirely frozen during this stage.

To adapt Xception for embedding-based tasks, the original classification head was replaced with a compact projection module, consistent with the design used in the ResNet variants. Specifically, the output of the final convolutional block—shaped $7 \times 7 \times 2048$—was passed through a Global Average Pooling (GAP) layer, yielding a 2048-dimensional vector. This was then passed through a Dropout layer (rate = 0.2), followed by a dense projection layer of 128 units, and finally an $L_2$ normalization layer to ensure unit-length embeddings. This architecture enables cosine-based similarity comparisons during training and evaluation, as outlined in Section 3.3.

Notably, the projection head design in Xception replicates the single-layer structure used in the ResNet variants, yielding an identical parameter count (262,272). The model was trained using 8,005 triplets for training and 1,486 for validation, as reported in Table 3.3.

| **Layer Name** | **Output Shape** | **Trainable Params** | **Non-trainable Params** |
|---|---|---|---|
| Xception Backbone (frozen) | (None, 7, 7, 2048) | 0 | 20,861,480 |
| Global Average Pooling (GAP) | (None, 2048) | 0 | 0 |
| Dropout (rate = 0.2) | (None, 2048) | 0 | 0 |
| Dense (projection_dense) | (None, 128) | 262,272 | 0 |
| $L_2$ Normalization | (None, 128) | 0 | 0 |
| **Total** | — | **262,272** | **20,861,480** |

Table 3.14: Layer-wise parameter summary of the Phase 1 embedding model using a frozen Xception backbone and a trainable projection head.

### 3.10.14 Phase 2 of Xception: Layer-wise Fine-Tuning of High-Level Blocks

To improve upon the embedding performance of the zero-shot and Phase 1 Xception baselines, a second training phase was conducted in which selective fine-tuning was applied to the deeper layers of the Xception backbone. Specifically, convolutional blocks 11 through 14 were unfrozen, allowing gradient updates during training. These blocks span the final stages of the Middle Flow and the entire Exit Flow. The remaining lower-level blocks, which capture generic texture and edge information, remained frozen to preserve stability and generalization.

This configuration leverages Xception's efficient depthwise separable convolutions while tailoring the final semantic layers to the domain-specific task of object similarity learning.

The projection head followed the same structure used in the ResNet variants, consisting of global average pooling, dropout, a 128-unit dense layer, and $L_2$ normalization. Both the trainable Xception blocks and the projection head were jointly optimized using triplet loss.

A total of 12,858 augmented triplets were generated for this fine-tuning phase, comprising 10,286 training triplets and 2,572 validation triplets. A summary of the triplet construction is provided in Table 3.5.

| Layer Name | Output Shape | Trainable Params | Non-trainable Params |
|---|---|---|---|
| Xception Backbone (selectively unfrozen) | (None, 7, 7, 2048) | 7,654,840 | 13,206,640 |
| Global Average Pooling | (None, 2048) | 0 | 0 |
| Dropout | (None, 2048) | 0 | 0 |
| Dense (projection_dense) | (None, 128) | 262,272 | 0 |
| $L_2$ Normalization | (None, 128) | 0 | 0 |
| **Total** | — | **7,917,112** | **13,206,640** |

Table 3.15: Layer-wise parameter summary of the Phase 2 embedding model using Xception, with fine-tuning applied to convolutional blocks 11 through 14 and a trainable projection head.

## Summary of Phase 2 Configurations

To provide a comparative overview of the Phase 2 fine-tuning configurations across all evaluated models, Table 3.16 summarizes the key aspects of each architecture. These include the specific layers unfrozen for training, the structure of the projection head, the total number of trainable parameters, and the dataset scale used for training. This summary allows for direct comparison of model complexity, training scope, and architectural differences in the context of embedding-based object similarity learning.

| Model | Fine-Tuned Layers | Projection Head Components | Trainable Params | Triplets Used |
|---|---|---|---|---|
| **ResNet-50** | `conv4_x, conv5_x` | GAP<br>Dropout (0.2)<br>Dense (128)<br>$L_2$ Norm | 22,346,608 | 13,258 |
| **ResNet-101** | `conv4_x, conv5_x` | GAP<br>Dropout (0.2)<br>Dense (128)<br>$L_2$ Norm | 41,365,120 | 13,259 |
| **MobileNetV2** | `block 14-16` | GAP<br>Dropout (0.2)<br>Dense (128)<br>$L_2$ Norm | 1,277,888 | 12,590 |
| **Xception** | `blocks 11-14` | GAP<br>Dropout (0.2)<br>Dense (128)<br>$L_2$ Norm | 7,917,112 | 12,858 |

Table 3.16: Summary of fine-tuning configurations, trainable parameters, and triplets used per model.

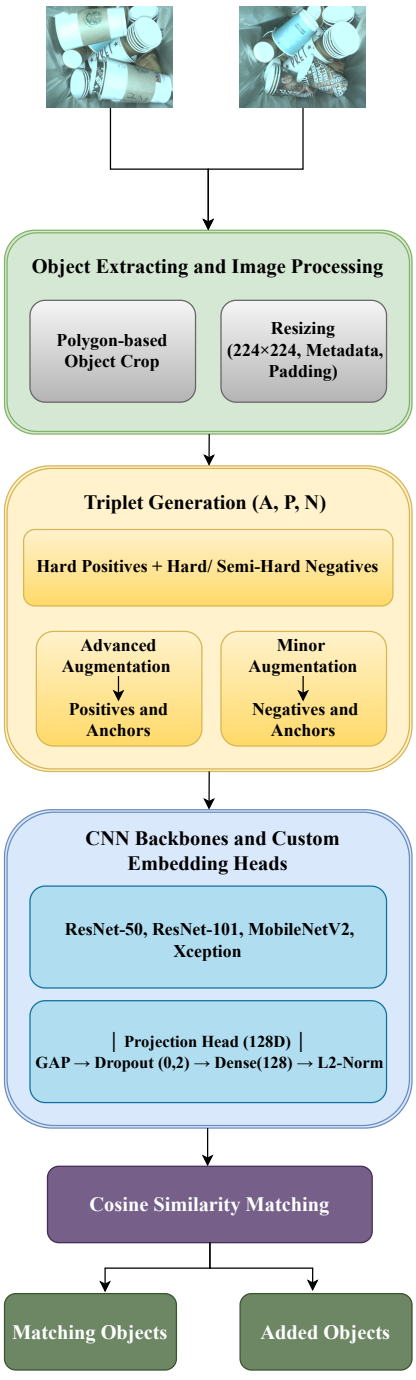



Figure 3.19: This diagram illustrates the overview of the Phase 2 pipeline, where objects are first extracted and grouped into triplets based on hard positive and hard/semi-hard negative criteria, then augmented to enhance variability. These triplets are used to train CNN backbones, resulting in a system capable of identifying matching and added objects.

# Chapter 4

# Results and Evaluation

In this chapter, we evaluate the effectiveness of our object-level change detection framework using a test set of 200 images. The evaluation covers both the Siamese baseline and the triplet-based CNN models. We begin by reporting the standalone evaluation results of the Siamese network, highlighting its strengths in pairwise verification. We then introduce a dual evaluation scheme—matched-object and added-object detection—to characterize the triplet-based models more comprehensively. Results are analyzed across all three training phases (zero-shot, projection head training, and fine-tuning), offering a comparative view of how embedding quality evolves and affects downstream tasks such as object matching and change identification.

## 4.1 Evaluation Metrics

To assess the performance of our object-level comparison system, we employ standard classification metrics—namely, **F1-score**, **Recall**, and **Precision**. These metrics are applied consistently but in different contexts: for the **Siamese baseline**, they quantify binary similarity prediction performance; for the **CNN-based triplet embedding models**, they are used to evaluate two downstream tasks: **object matching** and **added-object detection**.

**Precision** is defined as the proportion of correctly predicted positive instances among all predicted positives:

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}} \tag{4.1}$$

**Recall** measures the ability to identify all relevant ground truth positives:

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \tag{4.2}$$

**F1-score** is the harmonic mean of precision and recall, offering a balanced measure that penalizes both false positives and false negatives:

$$\text{F1} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{4.3}$$

## 4.2 Siamese Network

| Metric | Precision | Recall | F1-score |
|---|---|---|---|
| Siamese Network (threshold = 0.50) | 0.7751 | 0.6022 | 0.6778 |

Table 4.1: Evaluation performance of the Siamese network using a fixed distance threshold of 0.50 on cosine distance.

The evaluation uses a fixed cosine distance threshold of 0.50, which defines the binary decision boundary as follows:

- If the predicted cosine distance between two object crops is **below 0.50**, they are classified as *similar*.
- If the distance is **greater than or equal to 0.50**, they are classified as *dissimilar*.

Based on this configuration, the Siamese network achieves a **precision of 0.7751**, a **recall of 0.6022**, and an $F_1$ **score of 0.6778**. The high precision ($\approx 0.78$) indicates that the model has learned a strongly discriminative embedding: the vast majority of pairs it labels as "similar" are indeed true matches. This makes its predictions highly dependable for downstream applications that require reliable, high-confidence associations.

By contrast, the lower recall indicates that roughly 40% of genuine matches fall above the distance cut-off and are therefore missed. In other words, the network operates conservatively—it prefers to avoid false positives even if that means leaving some true positives unpaired.

### Distance-distribution analysis

To better understand the model's behavior, we examine the distribution of predicted distances in Figure 4.1.

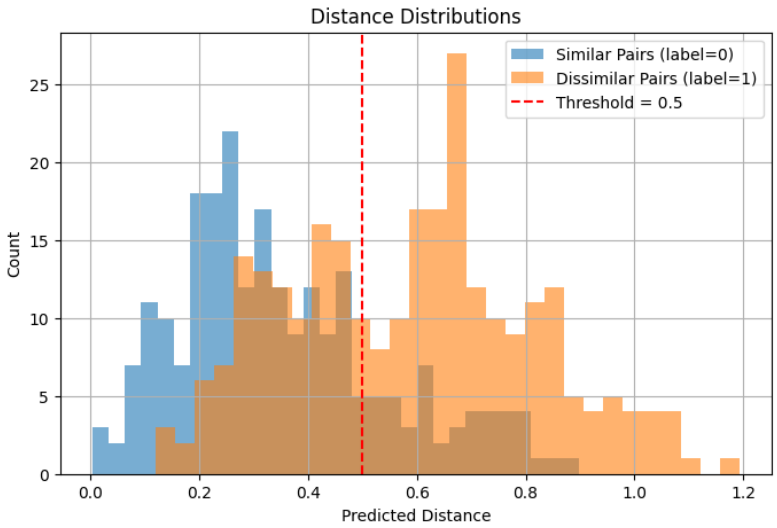



Figure 4.1: Distribution of predicted distances for similar (label=0) and dissimilar (label=1) image pairs under the Siamese network. The red dashed line indicates the decision threshold of 0.5.

Distances for similar pairs form a tight cluster between 0.1 and 0.4, whereas dissimilar pairs peak around 0.6–0.7; the clear separation shows that the network has organised the feature space into well-defined regions. The modest overlap in the 0.3–0.6 interval explains the recall value: some true matches land just beyond the conservative 0.50 cut-off. If higher coverage were needed, the threshold could be shifted leftward to capture those cases, trading a small amount of precision for additional recall. As configured, however, the Siamese model offers a robust, low-risk matching solution that prioritises reliability over exhaustive retrieval.

### Limitations and Transition to the Triplet-based CNN models

The Siamese architecture is highly attractive for tasks that demand binary, high-confidence verification: its single distance score yields very few false positives when two crops truly depict the same object. However, because that score is computed pair-by-pair, the model cannot seamlessly match *all* instances across two images or determine how many new objects have appeared—capabilities required for both the *matching* and *added-object* tasks addressed in this work. To satisfy these set-level matching and counting needs, we adopted a triplet-loss framework and trained four CNN backbones—ResNet-50, ResNet-101, MobileNetV2, and Xception—as described in our training methodology (Section 3.10). The resulting embeddings enable reliable identification and enumeration of added objects, and their performance is analyzed in the subsequent evaluation sections.

## 4.3 Triplet-Based CNN Models

To comprehensively evaluate our triplet-based object comparison system, we adopt a dual-metric strategy that includes both **matched-object evaluation** and **added-object detection**. These two evaluation modes reflect complementary aspects of scene understanding:

- **Object Matching:** Measures the model's ability to correctly identify objects that persist across two images.
- **Added Object Detection:** Measures the ability to detect newly introduced or missing objects between frames.

Although the final goal of our system is to identify newly added objects, evaluating both tasks is essential to fully characterize model behavior. Matching focuses on semantic continuity and identity preservation, while Added evaluation captures the model's responsiveness to genuine scene changes. Using only one of these perspectives can yield an incomplete or even misleading assessment. For instance, poor matching may artificially inflate added-object recall by misclassifying persistent objects as new.

This dual-perspective evaluation enables a robust and interpretable benchmarking of object-level change detection. In the following subsections, we present the evaluation results for both **matching** and **added-object detection**, across all three training phases—**zero-shot inference**, **Phase 1** (projection head training), and **Phase 2** (fine-tuning deeper layers).

### 4.3.1 Matching Performance of Pre-trained Models

This section presents the evaluation results of four pretrained CNN backbones: ResNet-50, ResNet-101, MobileNetV2, and Xception. These models were used as fixed feature extractors without task-specific fine-tuning.This highlights how each model performs in our metric learning and similarity task when applied in a zero-shot setting, without any training on the target dataset.

Embeddings were generated and matched using the procedure described in Section 3.3, and the predicted object matches were compared against ground truth annotations. Table 4.2 reports the F1 score, Recall, and Precision obtained by each model.

| Model | F1 Score | Recall | Precision |
|---|---|---|---|
| ResNet-50 | 0.6906 | 0.7471 | 0.6421 |
| ResNet-101 | 0.6866 | 0.6692 | 0.7049 |
| MobileNetV2 | 0.6736 | 0.6304 | 0.7232 |
| Xception | 0.6130 | 0.5486 | 0.6946 |

Table 4.2: Evaluation results for object matching performance using pre-trained convolutional models in the zero-shot setting.

Based on the zero-shot baseline results, **ResNet-50** emerges as the strongest model. Its 50-layer architecture is deep enough to capture rich, transferable features, yet not so large that those features become overly specialised to ImageNet; the result is the highest recall and the top overall $F_1$ (0.691). By contrast, **ResNet-101**, although deeper and slightly more precise, shows a small drop in recall that leaves its $F_1$ fractionally lower. **MobileNet-V2** and **Xception** trade depth for efficiency; their reduced parameter budgets limit representational capacity, yielding lower recall and, for Xception, the weakest $F_1$. Thus, within the purely zero-shot regime, a medium-depth backbone such as ResNet-50 offers the best balance of expressiveness and generalisation.

### 4.3.2 Matching Performance After Phase 1: Trainable Projection Head

This section reports the evaluation results of the embedding models trained in Phase 1, where the CNN backbone was kept frozen and only the projection head was optimized.

| Model | F1 Score | Recall | Precision |
|---|---|---|---|
| ResNet-50 | 0.6525 | 0.5953 | 0.7217 |
| ResNet-101 | 0.6940 | 0.6265 | 0.7778 |
| MobileNetV2 | 0.6374 | 0.5370 | 0.7841 |
| Xception | 0.5721 | 0.4708 | 0.7289 |

Table 4.3: Evaluation results for object matching performance using embedding models after Phase 1 training, where CNN backbones remain frozen and only the projection head is trained.

After projection-head fine-tuning, all four backbones exhibit the same general pattern: **precision increases whereas recall declines**. The models thus become more conservative—rejecting ambiguous matches more often (higher precision) yet overlooking a larger portion of true positives (lower recall). For ResNet-50, MobileNet-V2 and Xception the reduction in recall slightly outweighs the precision gain, producing a modest drop in $F_1$. **ResNet-101 is the lone exception**: its deeper, higher-capacity features allow the projection head to tighten decision boundaries without discarding as many borderline positives, yielding a small $F_1$ improvement.

This precision–recall trade-off is an expected consequence of the training regime. Triplet loss, applied to the compact 128-D projection head while the convolutional backbone remains frozen, drives intra-class embeddings closer together and pushes inter-class embeddings further apart. The resulting wider angular margins make the model more selective. MobileNetV2, as a lightweight backbone, suffers because its fixed features offer limited scope for the projection head to recover information that has been compressed away, whereas the high-capacity ResNet-101 retains redundant cues that the head can exploit.

Overall, the findings suggest that fine-tuning a minimal head boosts precision but may sacrifice coverage unless the underlying representation is sufficiently rich or additional backbone layers are unfrozen in later training phases.

### 4.3.3 Matching Performance After Phase 2: Layer-wise Fine-Tuning

This section presents the evaluation outcomes after Phase 2, where the deeper layers of each convolutional backbone were selectively unfrozen and fine-tuned alongside the projection head. This training stage enables the network to adjust high-level feature representations in response to the metric learning objective, allowing the embedding space to better capture object-level similarities specific to the task.

| Model | F1 Score | Recall | Precision |
|---|---|---|---|
| ResNet-50 | 0.8076 | 0.8249 | 0.7910 |
| ResNet-101 | 0.6850 | 0.6304 | 0.7500 |
| MobileNetV2 | 0.5972 | 0.4903 | 0.7636 |
| Xception | 0.6559 | 0.6342 | 0.6792 |

Table 4.4: Evaluation results for object matching performance after Phase 2 training, where both the projection head and selected backbone layers were fine-tuned.

Drawing on the metrics in Table 4.4, and comparing them with both the zero-shot baseline (Table 4.2) and the trainable projection head stage (Table 4.3), we observe the following trends for each model:

- **ResNet-50** benefits the most. With an additional 22 M trainable parameters (unfreezing `conv4_x`, `conv5_x`), its recall rebounds from the Phase 1 dip and surpasses the zero-shot level (0.825), pushing $F_1$ up to 0.808. The network is evidently large enough to adapt its filters to metric learning, yet still regularised by its moderate depth, avoiding over-fitting. , its recall rebounds from the Phase 1 dip and surpasses the zero-shot level (0.825), pushing $F_1$ up to 0.808.

- **ResNet-101** adds twice as many parameters (41 M), but the larger search space does not translate into better generalisation: precision rises to 0.750, recall slips to 0.630, and $F_1$ remains essentially flat (0.685 versus 0.687 zero-shot). The results suggest over-fitting once all high-capacity layers are released on a modest-sized triplet set.

- **MobileNetV2** (only 1.11 M new parameters) improves precision to 0.764 but suffers the lowest recall (0.490) and the weakest $F_1$ (0.597). The lightweight architecture lacks representational headroom to relearn discriminative features after fine-tuning, illustrating the limitations of extreme parameter efficiency for this task.

- **Xception** occupies a middle ground (7.6 M new trainable parameters). Fine-tuning boosts both recall (0.634) and precision (0.679), raising $F_1$ from 0.613 (zero-shot) and 0.572 (Phase 1) to 0.656. The moderate increase in capacity is sufficient to refine features without severe over-fitting.

Overall, Phase 2 shows that moderate additional capacity—exemplified by ResNet-50—yields the greatest gain, whereas too many unfrozen parameters (ResNet-101) risk over-fitting and too few (MobileNet-V2) leave the model under-adapted. These findings underscore the importance of aligning the extent of fine-tuning with both backbone size and dataset scale.

### 4.3.4 Visual Comparison of Matching Performance Across Training Phases

To complement the numerical evaluation in Tables 4.2–4.4, we present bar chart visualizations of matching performance for each backbone model. These plots illustrate the evolution of F1 Score, Recall, and Precision across the three training phases: zero-shot inference, projection-head fine-tuning (Phase 1), and deeper backbone fine-tuning (Phase 2).

By offering a visual comparison, these figures help highlight the relative strengths and weaknesses of each model throughout training, including trends such as recall–precision trade-offs and the impact of model capacity on fine-tuning effectiveness.

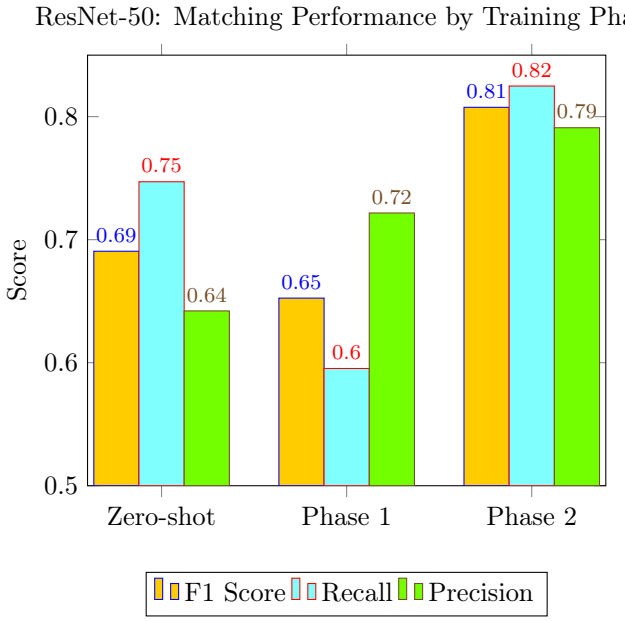



Figure 4.2: Matching performance of ResNet-50 across different training phases. F1 Score, Recall, and Precision are shown for the zero-shot baseline, projection-head fine-tuning (Phase 1), and partial backbone fine-tuning (Phase 2).

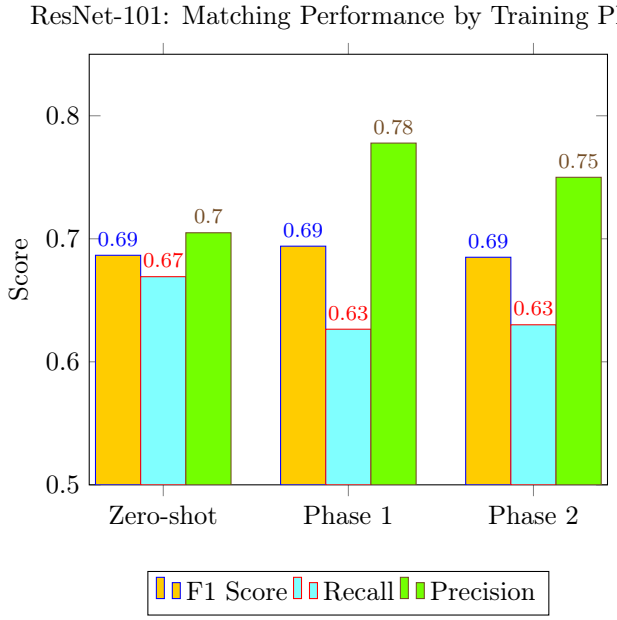



Figure 4.3: Matching performance of ResNet-101 across different training phases. F1 Score, Recall, and Precision are shown for the zero-shot baseline, projection-head fine-tuning (Phase 1), and partial backbone fine-tuning (Phase 2).

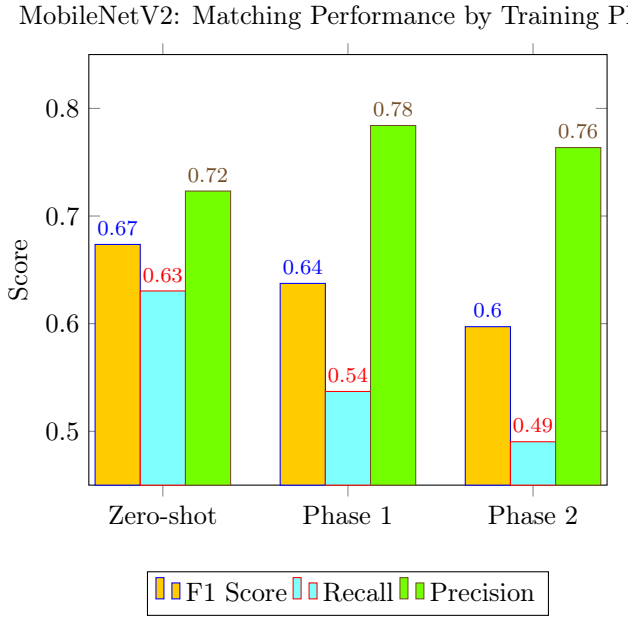



Figure 4.4: Matching Performance of MobileNetV2 across different training phases. F1 Score, Recall, and Precision are shown for the zero-shot baseline, projection-head fine-tuning (Phase 1), and deeper backbone fine-tuning (Phase 2).

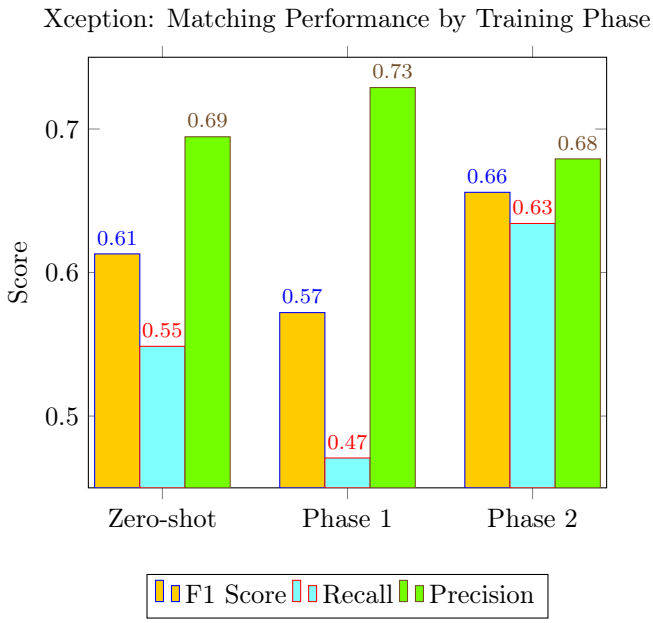



Figure 4.5: Matching Performance of Xception across different training phases. F1 Score, Recall, and Precision are shown for the zero-shot baseline, projection-head fine-tuning (Phase 1), and deeper backbone fine-tuning (Phase 2).

### 4.3.5 Added Object Detection with Pre-trained Models

In addition to evaluating matched object pairs, we assess each model's ability to identify newly *added* objects—those present in one image of the pair but not the other. This task reflects the model's sensitivity to detecting novel content. The table below reports the performance of all four CNN backbones when used as frozen feature extractors, without any task-specific fine-tuning.

| Model | F1 Score | Recall | Precision |
|---|---|---|---|
| ResNet-50 | 0.7562 | 0.6485 | 0.9069 |
| ResNet-101 | 0.7929 | 0.8121 | 0.7746 |
| MobileNetV2 | 0.7933 | 0.8606 | 0.7358 |
| Xception | 0.7599 | 0.8727 | 0.6729 |

Table 4.5: Evaluation results on added object detection using pre-trained CNN backbones (zero-shot inference).

These results offer insight into how well each backbone generalises to out-of-distribution objects detected in a zero-shot setting:

- **ResNet-50** delivers the strongest precision (0.9069), indicating cautious yet highly accurate detections, though at the cost of the lowest recall (0.6485).
- **ResNet-101** provides a well-balanced performance ($F_1 = 0.7929$), pairing solid recall (0.8121) with good precision (0.7746).
- **MobileNetV2** achieves the highest recall (0.8606), effectively flagging most added objects, but this comes with more false positives (precision: 0.7358).

- **Xception** records the highest recall overall (0.8727), but its precision is the weakest (0.6729), signalling a tendency toward over-detection.

### 4.3.6 Added Object Detection after Phase 1 Training

Following the training of the projection head in Phase 1, we re-evaluate the models' ability to detect newly added objects. The convolutional backbones remain frozen, and only the projection layer learns task-specific embeddings via triplet loss. Table 4.6 presents the updated results.

| Model | F1 Score | Recall | Precision |
|---|---|---|---|
| ResNet-50 | 0.7676 | 0.8606 | 0.6927 |
| ResNet-101 | 0.7840 | 0.8909 | 0.7000 |
| MobileNetV2 | 0.7537 | 0.9273 | 0.6349 |
| Xception | 0.7260 | 0.9152 | 0.6016 |

Table 4.6: Evaluation results on added object detection after training the projection head (Phase 1).

Compared with the zero-shot baseline in Table 4.5, introducing a trainable projection head for each model leads to various performance shifts:

- **ResNet-50** – The jump in recall to 0.8606 means the model now finds many more added objects than in the zero-shot run, while its precision of 0.6927 remains acceptable. In practice, this backbone offers a strong balance between catching new items and limiting false alarms, though ResNet-101 slightly outperforms it across all metrics.

- **ResNet-101** – Recall rises to **0.8909** while precision remains moderate at **0.7000**, resulting in the highest $F_1$ score (**0.7840**) among all models. The detector becomes more eager to flag added objects yet maintains strong overall accuracy, making it suitable when slightly more false positives are tolerable.

- **MobileNetV2** – With recall soaring to 0.9273, the network misses almost no added objects. However, precision drops to 0.6349, so many extra regions are incorrectly marked; this setting favours exhaustive coverage over precision.

- **Xception** – Recall improves to 0.9152 while precision sinks to 0.6016. The model detects nearly all added objects but produces the highest rate of false positives, highlighting a strong bias toward coverage at the expense of accuracy.

### 4.3.7 Added Object Detection after Phase 2 Fine-Tuning

In Phase 2, a subset of backbone layers was unfrozen for each architecture to allow deeper adaptation to the metric learning objective. This section reports how this full fine-tuning stage affected the detection of added objects.

| Model | F1 Score | Recall | Precision |
|---|---|---|---|
| ResNet-50 | 0.8280 | 0.7879 | 0.8725 |
| ResNet-101 | 0.7760 | 0.8606 | 0.7065 |
| MobileNetV2 | 0.7242 | 0.9152 | 0.5992 |
| Xception | 0.7485 | 0.7758 | 0.7232 |

Table 4.7: Evaluation results on added object detection after partial backbone fine-tuning (Phase 2).

The Phase-2 results illustrate how each backbone responds once its deeper convolutional blocks are unfrozen, and they should be read alongside the zero-shot baseline in Table 4.5 and the projection-head stage in Table 4.6.

**ResNet-50** continues to improve: precision rebounds to 0.872 and recall remains strong at 0.788, lifting $F_1$ to 0.828. Releasing the extra 22 M weights therefore gives the network enough flexibility to adapt to added objects without over-fitting, confirming that a medium-depth backbone is well matched to the task. By contrast, **ResNet-101**—with more than 41 M newly trainable parameters—shows classic over-fitting behaviour: recall climbs further (0.860) but precision slips to 0.706, so $F_1$ falls below its zero-shot value.

**MobileNetV2** also deteriorates: despite retaining the highest recall (0.915), precision drops to 0.599 and $F_1$ slides to 0.724, suggesting that the lightweight architecture lacks enough capacity to refine its features once deeper layers are updated. Finally, **Xception** shifts toward a more balanced trade-off: recall decreases to 0.776 while precision rises to 0.723, yielding an $F_1$ of 0.749—higher than in Phase 1 and close to its zero-shot level.

These results emphasize that backbone capacity and fine-tuning depth must be aligned: deeper adaptation benefits moderate-sized models like ResNet-50 and Xception, while MobileNetV2 over-commits to recall at the expense of precision, and ResNet-101 tends to overfit without sufficient regularization.

### 4.3.8 Visual Comparison of Added Detection Performance Across Training Phases

To support the numerical results presented earlier, Figures 4.6 to 4.9 illustrate the evolution of performance metrics—F1 Score, Recall, and Precision—for added object detection across three training stages: zero-shot inference, projection-head fine-tuning (Phase 1), and deeper backbone fine-tuning (Phase 2).

These charts provide a clear, side-by-side visual comparison of how each model adapts to this auxiliary detection task, revealing different trade-offs. In particular, they highlight the extent to which each backbone generalizes to identifying unseen content during fine-tuning.

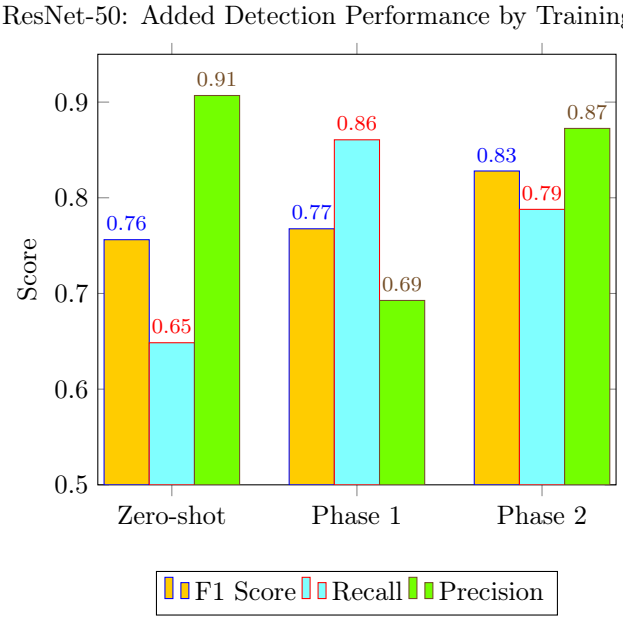



Figure 4.6: Added detection performance of ResNet-50 across different training phases.

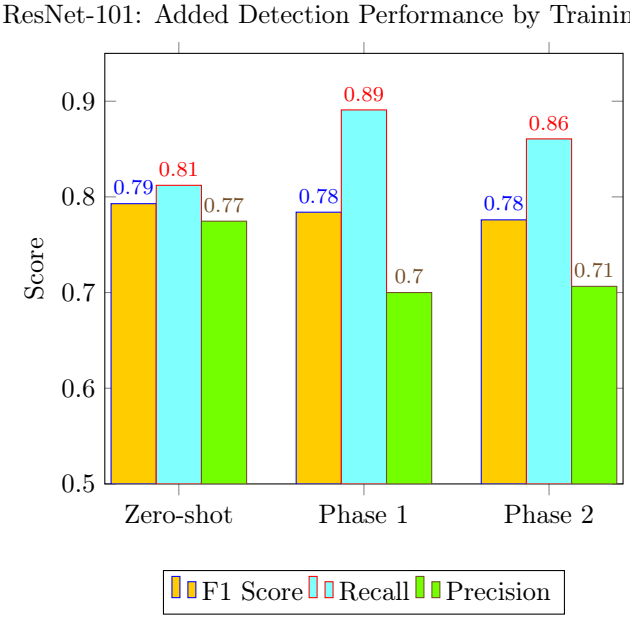



Figure 4.7: Added detection performance of ResNet-101 across different training phases.

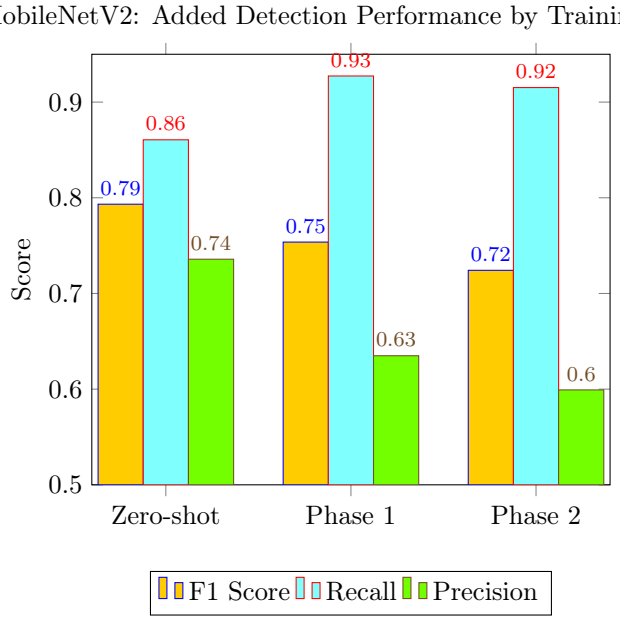



Figure 4.8: Added detection performance of MobileNetV2 across different training phases.

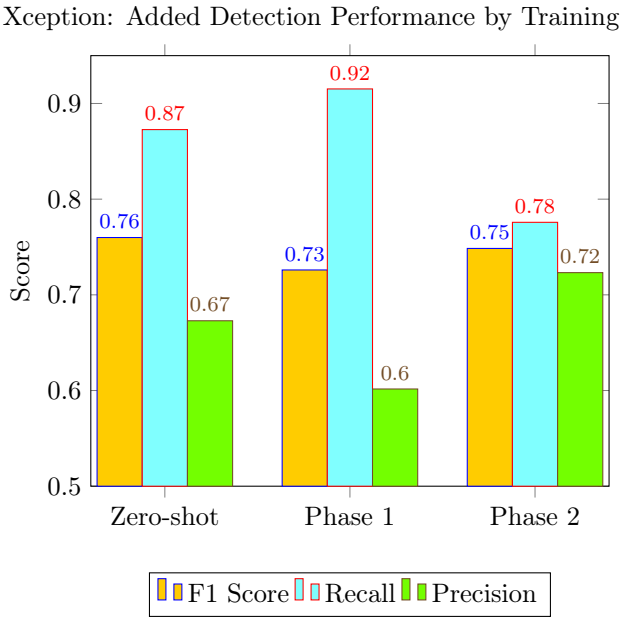



Figure 4.9: Added detection performance of Xception across different training phases.

### 4.3.9 Threshold Sensitivity Analysis

Using ResNet-50 as a case-study, we performed a grid-search over cosine-similarity cut-offs ($0.60 \leq \theta \leq 0.80$) for the *matching* and *added-object* tasks at three checkpoints: the frozen zero-shot model, the projection-head–only model (Phase 1), and the partially fine-tuned backbone (Phase 2).

As a representative example, we present the grid search results for the ResNet-50 model in Phase 2, showing how threshold variations affect precision, recall, and F1-score for both the matching and added-object tasks. Complete results for all other models and training stages are provided in Appendix A.

| Threshold | Matching | | | Added Object | | |
|---|---|---|---|---|---|---|
| | F1 | Recall | Precision | F1 | Recall | Precision |
| 0.60 | 0.7926 | 0.8327 | 0.7562 | 0.7960 | 0.7212 | 0.8881 |
| 0.62 | 0.7970 | 0.8327 | 0.7643 | 0.8013 | 0.7333 | 0.8832 |
| 0.64 | 0.7970 | 0.8249 | 0.7709 | 0.8078 | 0.7515 | 0.8732 |
| 0.66 | 0.8076 | 0.8249 | 0.7910 | 0.8280 | 0.7879 | 0.8725 |
| 0.68 | 0.7876 | 0.7938 | 0.7816 | 0.8224 | 0.8000 | 0.8462 |
| 0.70 | 0.7867 | 0.7821 | 0.7913 | 0.8171 | 0.8121 | 0.8221 |
| 0.72 | 0.7816 | 0.7588 | 0.8058 | 0.8176 | 0.8424 | 0.7943 |
| 0.74 | 0.7705 | 0.7315 | 0.8139 | 0.8148 | 0.8667 | 0.7688 |
| 0.76 | 0.7542 | 0.6926 | 0.8279 | 0.8120 | 0.9030 | 0.7376 |
| 0.78 | 0.7414 | 0.6693 | 0.8309 | 0.8000 | 0.9091 | 0.7143 |
| 0.80 | 0.7184 | 0.6304 | 0.8351 | 0.7784 | 0.9152 | 0.6771 |

Table 4.8: Grid search results for different cosine similarity thresholds in ResNet-50 Phase 2. Metrics are shown for both **Matching** and **Added-object** detection.

These grid searches enabled us to extract comparative plots that highlight different aspects of threshold behavior. To better visualize these effects, we plotted precision, recall, and F1-score variations with cosine similarity thresholds for ResNet-50 Phase 2 (Figure 4.10), and compared matching F1 performance across the zero-shot model, Phase 1, and Phase 2 (Figure 4.11).

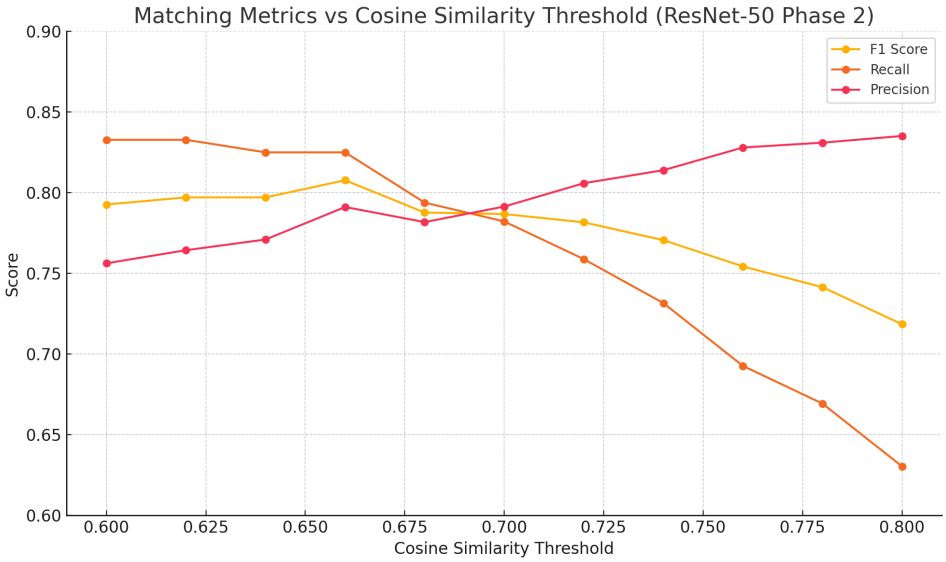



Figure 4.10: Effect of cosine similarity threshold on **precision**, **recall**, and **F1-score** in ResNet-50 Phase 2 for the *matching* task.

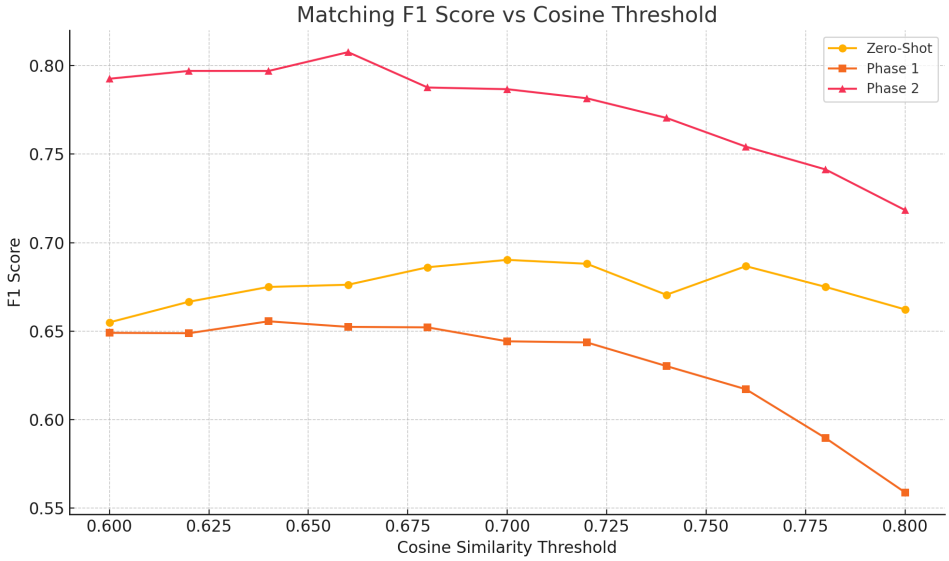



Figure 4.11: Threshold sensitivity of the **matching** F1 score in ResNet-50 across training stages.

Based on the results presented in Table 4.8, Figure 4.10, and Figure 4.11, two key trends can be observed:

1. **Trade-off Curve.** As expected, raising the threshold monotonically increases precision and decreases recall, producing a concave $F_1$ curve with a single peak for each phase. The peak therefore marks the point where the marginal gain in precision no longer compensates for the loss of recall.

2. **Shift of the optimal cut-off.** The threshold that maximizes $F_1$ is not fixed; it moves as the embedding space becomes more discriminative. This evolution is summarized in the table below:

| | Zero-shot | Phase 1 | Phase 2 |
|---|---|---|---|
| Best $\theta$ for *Match* $F_1$ | 0.70 | 0.64 | 0.66 |
| Best $\theta$ for *Added* $F_1$ | 0.76 | 0.62 | 0.66 |

   - In the **zero-shot** model, true-match similarities are relatively low, so a higher cut-off ($\approx 0.70$) is needed to suppress false positives.
   - After **projection-head tuning** (Phase 1), the head pulls positive pairs closer together; a lower cut-off ($\approx 0.64$) now balances precision and recall.
   - With **partial backbone fine-tuning** (Phase 2), both tasks peak at $\theta \approx 0.66$. The added capacity tightens intra-class clusters while widening inter-class gaps, causing the two tasks to converge on the same operating point.

We adopted $\theta = 0.66$ as a single working threshold because it is optimal for Phase 2—our final model—and remains close to the maxima in earlier stages. Nonetheless, the grid search underscores that re-calibrating the similarity cut-off after each training phase can yield measurable gains, especially when the downstream task places a premium on either exhaustive recall (lower $\theta$) or high precision (higher $\theta$).

### Effect of Cosine Threshold Across Backbone Models

To assess whether the optimal cosine similarity cut-off varies across backbone architectures, we plotted the matching $F_1$ score across thresholds for all Phase 2 models.

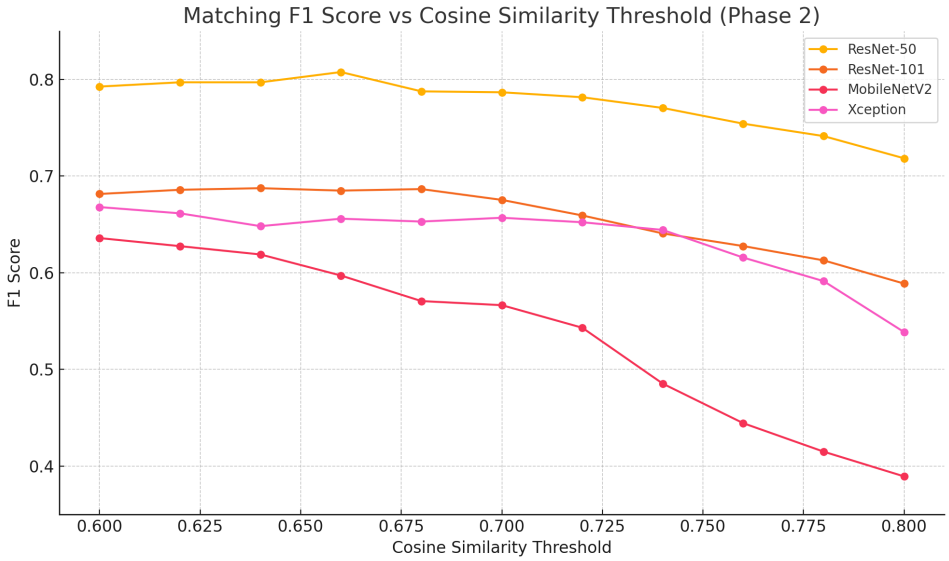



Figure 4.12: Comparison of **matching** $F_1$ score across cosine similarity thresholds for different Phase 2 backbones: ResNet-50, ResNet-101, MobileNetV2, and Xception.

Figure 4.12 shows that each backbone reaches its peak matching $F_1$ at a slightly different cosine cut-off (ResNet-50 at $\theta \approx 0.66$, ResNet-101 and MobileNetV2 at $\theta \approx 0.64$, and Xception at $\theta \approx 0.60$). While choosing a model-specific threshold would eke out an extra 1–6 % of $F_1$ for the lighter backbones, it would also introduce an additional hyper-parameter for every architecture, complicating both deployment and comparative analysis. For a *fair, single-pass evaluation* we therefore adopt a **common cosine threshold of 0.66** across all steps. This value is the plateau point for ResNet-50 and remains within a few percent of the individual optima for the other backbones, ensuring that the performance differences reported in Section 4.3 reflect the network architectures rather than threshold tuning.

# Chapter 5

# Conclusion

This final chapter consolidates the main outcomes of the project, highlighting key findings, model comparisons, and architectural insights gained from the experiments. It concludes by proposing several directions for future research and practical improvements to enhance the effectiveness and scalability of the proposed smart-bin change detection system.

## 5.1 Conclusion

This thesis presented a deep metric-learning framework for object-level change detection in smart waste-bin environments. The primary objective was to detect and track individual waste items across successive bin images using image embeddings, in order to identify newly added objects under real-world conditions of clutter, occlusion, and lighting variability.

To achieve this, we first implemented a Siamese network with contrastive loss as a baseline. While the model demonstrated strong precision (0.7751), it lacked the capability to detect multiple object-level changes, as it produced a single scalar distance per image pair and offered no insight into instance-level correspondence or count. This limitation motivated the transition to a triplet-loss framework, enabling the training of embedding models that support fine-grained object matching across images.

We developed and benchmarked four convolutional backbones—ResNet-50, ResNet-101, MobileNetV2, and Xception—under three training configurations. First, we evaluated each model in a *zero-shot* setting, using frozen feature extractors without any fine-tuning to assess their native embedding quality. In *Phase 1*, we trained a projection head on top of the frozen backbones to better adapt the embeddings to the object-matching task. Finally, *Phase 2* selectively fine-tuned deeper convolutional layers using strong augmentations tailored to waste-item variation and a custom triplet sampling strategy that combined hard and semi-hard mining. This progressive training strategy allowed us to evaluate trade-offs between model capacity, generalization, and computational efficiency across real-world bin conditions.

The selected models span a diverse range of architectural depths and computational profiles: ResNet-50 and ResNet-101 contain 50 and 101 layers respectively, with parameter counts of 25.6M and 44.5M and FLOPs of 4.1G and 7.8G (for $224 \times 224$ input), while MobileNetV2 offers a lightweight alternative with 3.4M parameters and only 0.3G FLOPs, benefiting from an efficient design based on depthwise separable convolutions and inverted residuals. In contrast, Xception—though also built on separable convolutions—incurs 8.4G FLOPs due to its greater depth and higher-resolution design. All models, except in the zero-shot setting, project into a common 128-dimensional embedding space, enabling consistent comparison across the training phases.

Evaluation results revealed distinct performance patterns across training phases and models, particularly for the two core tasks of *matching* (identifying unchanged objects across images) and *added-object detection.*

In the zero-shot setting, where models were evaluated using frozen pre-trained backbones, ResNet-50 achieved the highest F1-score for matching (0.6906), while MobileNetV2 slightly outperformed others in added-object detection (0.7933), likely benefiting from its lightweight regularization. Interestingly, Xception underperformed in both tasks in the zero-shot setting, indicating that its depthwise-separable architecture may be less effective at producing general-purpose embeddings without prior adaptation to the task.

Phase 1, which involved training only the projection head while keeping the feature extractor frozen, yielded modest or even reduced gains. ResNet-101 slightly improved in matching F1-score (from 0.6866 to 0.6940) due to increased precision, and ResNet-50 improved marginally in added-object F1-score (from 0.7561 to 0.7676). However, these results indicate that updating only the projection head is insufficient to adapt the embedding space, as the frozen backbone may not encode the visual variability specific to waste objects—especially under clutter, occlusion, or lighting shifts.

Phase 2, involving layer-wise fine-tuning of the convolutional layers, produced the most substantial performance gains. In matching, ResNet-50 exhibited a remarkable improvement of +11.7 points in F1-score (from 0.6906 to 0.8076), alongside robust recall (0.8249) and precision (0.7910), confirming its capacity to generalize under real-world variation. Xception also improved (F1 = 0.6559), but its final precision (0.6792) and recall (0.6341) suggest instability in fine-grained object comparison. Meanwhile, ResNet-101 showed limited change across phases (matching $F_1$ remained around 0.68–0.69), likely due to overfitting caused by its deeper architecture and more than 41 million trainable parameters, which can be excessive given the relatively small training set of 5,560 images. MobileNetV2, with its lightweight structure (only about 1.3 million trainable parameters in this phase), degraded in both matching F1-score (from 0.6736 to 0.5972) and added-object detection (from 0.7933 to 0.7424), indicating limited representational capacity for nuanced comparison.

For added-object detection in Phase 2, ResNet-50 again achieved the highest F1-score (0.8280) with balanced recall (0.7879) and precision (0.8725), outperforming all other models. Notably, it was the only model to exhibit consistent improvements across both matching and added tasks, underscoring its robustness. In contrast, Xception, despite improving its matching performance in Phase 2, did not keep progressing in added-object detection (F1 = 0.7485), suggesting that architectural differences—such as its heavy reliance on separable convolutions and potential mismatch with the $224 \times 224$ input resolution used in our pipeline—may introduce training instability or reduce alignment with instance-based objectives.

In summary, while deep or lightweight models like ResNet-101 and MobileNetV2 showed strengths in specific metrics, they failed to consistently improve or generalize across training regimes. These findings highlight the impact of both dataset limitations—such as limited training samples and class imbalance (see Figure 3.2)—and the alignment between model characteristics and task demands, including model depth, parameter count, and capacity to generalize from limited data. ResNet-50, with moderate depth and computational load (4.1G FLOPs), strikes the optimal balance between representation power and overfitting risk, emerging as the most reliable choice for object-level change detection in this domain.

Inference visualizations confirmed the models' ability to detect added items even in complex scenes with overlapping waste. The triplet-trained embeddings successfully clustered semantically similar objects, enabling reliable differentiation of new vs. existing items.

Overall, this work demonstrates the feasibility and effectiveness of using deep metric learning for object-level change detection in smart-bin systems, offering a substantial advance over traditional fill-level sensors and global similarity baselines.

Beyond the immediate application to smart-bin systems, this work contributes to the broader vision of intelligent resource monitoring in smart cities. By enabling fine-grained object-level change detection with affordable camera setups, the proposed framework opens avenues for automated waste auditing, behavioral analytics, and environmental compliance in recycling facilities. Furthermore, the embedding-based similarity approach could inspire extensions to industrial inspection, smart retail inventory tracking, and other domains that demand instance-level visual reasoning at scale.

## 5.2 Future Work

While the proposed framework showed promising results, several avenues remain open for future research and development.

First, real-world deployment would benefit from optimizing the inference pipeline for edge devices. Techniques such as model quantization, knowledge distillation, or pruning could reduce the memory and compute requirements of heavier models like Xception and ResNet-101, enabling onboard processing within smart-bin units.

Second, although the current evaluation focused on static image pairs, many practical applications involve continuous monitoring or video streams. Extending the system to track object changes over time sequences would improve temporal reasoning and help detect gradual contamination or delayed disposal behaviors.

Third, the current approach does not exploit segmentation masks during training. Incorporating segmentation-aware losses or attention mechanisms may enhance feature discrimination and spatial awareness, especially for small or partially occluded objects.

Fourth, the triplet generation strategy can be further improved. Currently, triplets are generated from cosine similarity with optional hard-mining heuristics. More adaptive sampling techniques—such as online batch-hard mining or reinforcement-based selection—could yield more challenging training examples and improve generalization.

Fifth, expanding the dataset to include a larger number of training images and a more balanced distribution of object categories would allow deeper models—such as ResNet-101—to better exploit their capacity without overfitting. A broader dataset would also improve robustness across diverse waste types and lighting conditions.

Finally, integrating this system into a full smart-bin management platform—complete with contamination alerts, user feedback, and route optimization—would offer valuable operational insight and support sustainable waste-handling policies such as pay-as-you-throw billing.

In summary, this thesis lays the foundation for intelligent, object-level monitoring in waste management and offers a flexible deep learning pipeline for future real-world deployment and academic extension.

# Appendix

# Appendix A

# Grid Search Results for Cosine Thresholds

This appendix presents the detailed results of grid searches over cosine similarity thresholds ($\theta = 0.60$ to $0.80$) conducted for all models, including the zero-shot (pretrained) baseline as well as the two training stages (Phase 1 and Phase 2). For each architecture, we report performance metrics (**F1-score**, **Recall**, and **Precision**) for both the **matching** and **added-object** tasks. These results complement the main threshold analysis in Chapter 4 and offer transparency regarding per-model threshold sensitivity across different training configurations.

| **Threshold** | **Matching** | | | **Added Object** | | |
|---|---|---|---|---|---|---|
| | **F1** | **Recall** | **Precision** | **F1** | **Recall** | **Precision** |
| 0.60 | 0.654991 | 0.727626 | 0.595541 | 0.694030 | 0.563636 | 0.902013 |
| 0.62 | 0.666667 | 0.735409 | 0.609677 | 0.705882 | 0.581818 | 0.897196 |
| 0.64 | 0.675000 | 0.735409 | 0.623762 | 0.731183 | 0.618182 | 0.894737 |
| 0.66 | 0.672659 | 0.731518 | 0.628763 | 0.734982 | 0.630303 | 0.831596 |
| 0.68 | 0.686131 | 0.731518 | 0.646048 | 0.762887 | 0.672727 | 0.880952 |
| 0.70 | 0.688000 | 0.719844 | 0.663082 | 0.792079 | 0.727273 | 0.869565 |
| 0.72 | 0.688091 | 0.708171 | 0.669118 | 0.787097 | 0.739394 | 0.843179 |
| 0.74 | 0.670565 | 0.669261 | 0.671875 | 0.785276 | 0.775758 | 0.795031 |
| 0.76 | 0.686747 | 0.665370 | 0.709544 | 0.797654 | 0.842424 | 0.772727 |
| 0.78 | 0.675052 | 0.626459 | 0.731818 | 0.790055 | 0.866667 | 0.725888 |
| 0.80 | 0.662309 | 0.591440 | 0.752475 | 0.778947 | 0.896970 | 0.688372 |

Table A.1: Grid search results for cosine similarity thresholds in **ResNet-50 Pre-Trained**. Metrics are shown for both **Matching** and **Added-object** detection.

| Threshold | Matching | | | Added Object | | |
|---|---|---|---|---|---|---|
| | **F1** | **Recall** | **Precision** | **F1** | **Recall** | **Precision** |
| 0.60 | 0.649087 | 0.622568 | 0.677966 | 0.786127 | 0.824242 | 0.751381 |
| 0.62 | 0.648871 | 0.614786 | 0.686957 | 0.795455 | 0.848485 | 0.748663 |
| 0.64 | 0.655602 | 0.614786 | 0.702222 | 0.784314 | 0.848485 | 0.729167 |
| 0.66 | 0.652452 | 0.595331 | 0.721698 | 0.767568 | 0.860606 | 0.692683 |
| 0.68 | 0.652174 | 0.583658 | 0.738916 | 0.759894 | 0.872727 | 0.672897 |
| 0.70 | 0.644326 | 0.560311 | 0.757895 | 0.760224 | 0.903030 | 0.656388 |
| 0.72 | 0.643678 | 0.544747 | 0.786517 | 0.742574 | 0.909091 | 0.627615 |
| 0.74 | 0.630332 | 0.517510 | 0.806061 | 0.738609 | 0.933333 | 0.611111 |
| 0.76 | 0.617225 | 0.501946 | 0.801422 | 0.731591 | 0.933333 | 0.601562 |
| 0.78 | 0.589681 | 0.466926 | 0.800000 | 0.717593 | 0.933994 | 0.580524 |
| 0.80 | 0.558974 | 0.424125 | 0.819549 | 0.694878 | 0.945455 | 0.549296 |

Table A.2: Grid search results for cosine similarity thresholds in **ResNet-50 Phase 1**. Metrics are shown for both **Matching** and **Added-object** detection.

| Threshold | Matching | | | Added Object | | |
|---|---|---|---|---|---|---|
| | **F1** | **Recall** | **Precision** | **F1** | **Recall** | **Precision** |
| 0.60 | 0.792593 | 0.832685 | 0.756184 | 0.795987 | 0.721212 | 0.888060 |
| 0.62 | 0.797020 | 0.832685 | 0.764286 | 0.801325 | 0.733333 | 0.883212 |
| 0.64 | 0.796992 | 0.824903 | 0.770899 | 0.807818 | 0.751515 | 0.873239 |
| 0.66 | 0.807619 | 0.824903 | 0.791045 | 0.828025 | 0.787879 | 0.872483 |
| 0.68 | 0.787645 | 0.793774 | 0.781609 | 0.822430 | 0.800000 | 0.846154 |
| 0.70 | 0.786893 | 0.782101 | 0.791339 | 0.817073 | 0.812121 | 0.822086 |
| 0.72 | 0.781563 | 0.758755 | 0.805785 | 0.817647 | 0.842424 | 0.794286 |
| 0.74 | 0.770492 | 0.731518 | 0.813853 | 0.814815 | 0.866667 | 0.768817 |
| 0.76 | 0.754237 | 0.692607 | 0.827907 | 0.811989 | 0.903030 | 0.737624 |
| 0.78 | 0.741379 | 0.669261 | 0.830918 | 0.800000 | 0.909091 | 0.714286 |
| 0.80 | 0.718404 | 0.630350 | 0.835052 | 0.778351 | 0.915152 | 0.677130 |

Table A.3: Grid search results for cosine similarity thresholds in **ResNet-50 Phase 2**. Metrics are shown for both **Matching** and **Added-object** detection.

| Threshold | Matching | | | Added Object | | |
|---|---|---|---|---|---|---|
| | **F1** | **Recall** | **Precision** | **F1** | **Recall** | **Precision** |
| 0.60 | 0.655556 | 0.688716 | 0.625442 | 0.775920 | 0.703030 | 0.865672 |
| 0.62 | 0.652908 | 0.677043 | 0.630435 | 0.771242 | 0.715152 | 0.836879 |
| 0.64 | 0.655238 | 0.669261 | 0.641791 | 0.789089 | 0.751515 | 0.832215 |
| 0.66 | 0.651341 | 0.661479 | 0.641509 | 0.788644 | 0.755576 | 0.822368 |
| 0.68 | 0.653696 | 0.653696 | 0.653696 | 0.793846 | 0.781818 | 0.806250 |
| 0.70 | 0.661386 | 0.649885 | 0.673387 | 0.796407 | 0.806061 | 0.786982 |
| 0.72 | 0.643299 | 0.607004 | 0.684211 | 0.785311 | 0.842424 | 0.735450 |
| 0.74 | 0.632479 | 0.575875 | 0.701422 | 0.780762 | 0.884848 | 0.708738 |
| 0.76 | 0.622222 | 0.544747 | 0.725389 | 0.781491 | 0.921212 | 0.678571 |
| 0.78 | 0.604651 | 0.508537 | 0.751445 | 0.757946 | 0.939394 | 0.635246 |
| 0.80 | 0.589372 | 0.474708 | 0.777070 | 0.734118 | 0.945455 | 0.600000 |

Table A.4: Grid search results for cosine similarity thresholds in **ResNet-101 Pre-Trained**. Metrics are shown for both **Matching** and **Added-object** detection.

| Threshold | Matching | | | Added Object | | |
|---|---|---|---|---|---|---|
| | **F1** | **Recall** | **Precision** | **F1** | **Recall** | **Precision** |
| 0.60 | 0.704453 | 0.677043 | 0.734177 | 0.782609 | 0.818182 | 0.750000 |
| 0.62 | 0.710744 | 0.669261 | 0.757799 | 0.788732 | 0.848485 | 0.736842 |
| 0.64 | 0.690678 | 0.634241 | 0.758140 | 0.790191 | 0.877878 | 0.717822 |
| 0.66 | 0.693966 | 0.624659 | 0.777778 | 0.784000 | 0.890909 | 0.700000 |
| 0.68 | 0.665208 | 0.591440 | 0.760000 | 0.774869 | 0.896970 | 0.682028 |
| 0.70 | 0.653333 | 0.571984 | 0.761658 | 0.766067 | 0.903030 | 0.665179 |
| 0.72 | 0.644144 | 0.556242 | 0.764706 | 0.754430 | 0.903030 | 0.647826 |
| 0.74 | 0.628110 | 0.521401 | 0.770115 | 0.745098 | 0.921212 | 0.625514 |
| 0.76 | 0.596059 | 0.486381 | 0.771065 | 0.728571 | 0.927273 | 0.600000 |
| 0.78 | 0.555000 | 0.431987 | 0.776224 | 0.710766 | 0.945455 | 0.569943 |
| 0.80 | 0.541237 | 0.408560 | 0.801527 | 0.700655 | 0.957576 | 0.552448 |

Table A.5: Grid search results for cosine similarity thresholds in **ResNet-101 Phase 1**. Metrics are shown for both **Matching** and **Added-object** detection.

| Threshold | Matching | | | Added Object | | |
|---|---|---|---|---|---|---|
| | **F1** | **Recall** | **Precision** | **F1** | **Recall** | **Precision** |
| 0.60 | 0.681542 | 0.653696 | 0.711864 | 0.786127 | 0.824242 | 0.751381 |
| 0.62 | 0.688532 | 0.649805 | 0.726087 | 0.778409 | 0.830303 | 0.732620 |
| 0.64 | 0.687500 | 0.642023 | 0.739910 | 0.774373 | 0.842424 | 0.716495 |
| 0.66 | 0.684989 | 0.633063 | 0.750000 | 0.775956 | 0.860606 | 0.706468 |
| 0.68 | 0.686567 | 0.626459 | 0.759434 | 0.778378 | 0.872727 | 0.702439 |
| 0.70 | 0.675381 | 0.603113 | 0.767327 | 0.773684 | 0.890909 | 0.683721 |
| 0.72 | 0.659292 | 0.579767 | 0.764103 | 0.775194 | 0.900991 | 0.675676 |
| 0.74 | 0.640732 | 0.544747 | 0.777778 | 0.756219 | 0.921212 | 0.641350 |
| 0.76 | 0.627635 | 0.521401 | 0.788235 | 0.752427 | 0.933994 | 0.627530 |
| 0.78 | 0.612827 | 0.501946 | 0.785755 | 0.741627 | 0.939394 | 0.612648 |
| 0.80 | 0.588808 | 0.470817 | 0.785714 | 0.728972 | 0.945455 | 0.593156 |

Table A.6: Grid search results for cosine similarity thresholds in **ResNet-101 Phase 2**. Metrics are shown for both **Matching** and **Added-object** detection.

| Threshold | Matching | | | Added Object | | |
|---|---|---|---|---|---|---|
| | **F1** | **Recall** | **Precision** | **F1** | **Recall** | **Precision** |
| 0.60 | 0.692067 | 0.692067 | 0.692067 | 0.793846 | 0.781818 | 0.806250 |
| 0.62 | 0.695825 | 0.680934 | 0.711382 | 0.809524 | 0.824242 | 0.795322 |
| 0.64 | 0.699187 | 0.669261 | 0.731915 | 0.806916 | 0.848485 | 0.769231 |
| 0.66 | 0.703158 | 0.649085 | 0.766055 | 0.802198 | 0.884848 | 0.733668 |
| 0.68 | 0.695652 | 0.622568 | 0.783177 | 0.791557 | 0.909091 | 0.708955 |
| 0.70 | 0.677021 | 0.583658 | 0.806452 | 0.767677 | 0.921212 | 0.658089 |
| 0.72 | 0.627635 | 0.521401 | 0.788235 | 0.747573 | 0.933333 | 0.623482 |
| 0.74 | 0.597087 | 0.478599 | 0.793548 | 0.730679 | 0.945455 | 0.595420 |
| 0.76 | 0.589421 | 0.455253 | 0.835714 | 0.723982 | 0.969697 | 0.577617 |
| 0.78 | 0.555844 | 0.416342 | 0.835938 | 0.704846 | 0.969697 | 0.553633 |
| 0.80 | 0.497268 | 0.354086 | 0.834862 | 0.676533 | 0.960697 | 0.519481 |

Table A.7: Grid search results for cosine similarity thresholds in **MobileNetV2 Pre-Trained**. Metrics are shown for both **Matching** and **Added-object** detection.

| Threshold | Matching | | | Added Object | | |
|---|---|---|---|---|---|---|
| | **F1** | **Recall** | **Precision** | **F1** | **Recall** | **Precision** |
| 0.60 | 0.663774 | 0.595331 | 0.750000 | 0.777778 | 0.890909 | 0.698141 |
| 0.62 | 0.660574 | 0.579767 | 0.760841 | 0.768041 | 0.903030 | 0.668161 |
| 0.64 | 0.657596 | 0.564202 | 0.788043 | 0.763819 | 0.921212 | 0.652361 |
| 0.66 | 0.637413 | 0.536965 | 0.784091 | 0.753695 | 0.927273 | 0.634855 |
| 0.68 | 0.609524 | 0.498045 | 0.752576 | 0.749403 | 0.951515 | 0.618110 |
| 0.70 | 0.611650 | 0.490272 | 0.812903 | 0.744731 | 0.963636 | 0.606870 |
| 0.72 | 0.572864 | 0.443580 | 0.808511 | 0.725624 | 0.969697 | 0.579710 |
| 0.74 | 0.538860 | 0.404669 | 0.806202 | 0.710817 | 0.975758 | 0.559028 |
| 0.76 | 0.519894 | 0.381323 | 0.816667 | 0.696970 | 0.975758 | 0.542888 |
| 0.78 | 0.486188 | 0.342412 | 0.838095 | 0.679245 | 0.981818 | 0.519231 |
| 0.80 | 0.434783 | 0.291829 | 0.852273 | 0.659919 | 0.987879 | 0.495441 |

Table A.8: Grid search results for cosine similarity thresholds in **MobileNetV2 Phase 1**. Metrics are shown for both **Matching** and **Added-object** detection.

| Threshold | Matching | | | Added Object | | |
|---|---|---|---|---|---|---|
| | **F1** | **Recall** | **Precision** | **F1** | **Recall** | **Precision** |
| 0.60 | 0.638581 | 0.560311 | 0.742268 | 0.757732 | 0.890909 | 0.659193 |
| 0.62 | 0.627540 | 0.540856 | 0.747312 | 0.752525 | 0.903030 | 0.645022 |
| 0.64 | 0.618938 | 0.521401 | 0.761364 | 0.743842 | 0.915152 | 0.626556 |
| 0.66 | 0.597156 | 0.490272 | 0.763636 | 0.724221 | 0.915152 | 0.599206 |
| 0.68 | 0.579732 | 0.455253 | 0.764706 | 0.722611 | 0.939394 | 0.587121 |
| 0.70 | 0.566502 | 0.447471 | 0.771812 | 0.715935 | 0.939394 | 0.578358 |
| 0.72 | 0.543147 | 0.416342 | 0.781022 | 0.696629 | 0.939394 | 0.553571 |
| 0.74 | 0.485333 | 0.354086 | 0.771186 | 0.681034 | 0.957576 | 0.528428 |
| 0.76 | 0.444124 | 0.311824 | 0.776699 | 0.657098 | 0.957576 | 0.503185 |
| 0.78 | 0.414986 | 0.280165 | 0.800000 | 0.646341 | 0.963636 | 0.486239 |
| 0.80 | 0.389381 | 0.256809 | 0.804878 | 0.644000 | 0.975758 | 0.480597 |

Table A.9: Grid search results for cosine similarity thresholds in **MobileNetV2 Phase 2**. Metrics are shown for both **Matching** and **Added-object** detection.

| Threshold | Matching | | | Added Object | | |
|---|---|---|---|---|---|---|
| | **F1** | **Recall** | **Precision** | **F1** | **Recall** | **Precision** |
| 0.60 | 0.612326 | 0.599222 | 0.626016 | 0.779762 | 0.793939 | 0.766082 |
| 0.62 | 0.612576 | 0.587549 | 0.639831 | 0.768786 | 0.806061 | 0.734807 |
| 0.64 | 0.615063 | 0.571984 | 0.665158 | 0.759003 | 0.830303 | 0.699880 |
| 0.66 | 0.613043 | 0.548638 | 0.694581 | 0.759894 | 0.872727 | 0.672897 |
| 0.68 | 0.600897 | 0.521041 | 0.728995 | 0.743084 | 0.884848 | 0.640351 |
| 0.70 | 0.597222 | 0.501946 | 0.737143 | 0.732187 | 0.903030 | 0.615722 |
| 0.72 | 0.598086 | 0.486381 | 0.776398 | 0.722090 | 0.921212 | 0.593750 |
| 0.74 | 0.579853 | 0.459144 | 0.786667 | 0.717593 | 0.939394 | 0.580524 |
| 0.76 | 0.562025 | 0.431097 | 0.804348 | 0.702703 | 0.945455 | 0.559140 |
| 0.78 | 0.544041 | 0.408560 | 0.819353 | 0.693157 | 0.951515 | 0.545139 |
| 0.80 | 0.502703 | 0.361688 | 0.823009 | 0.678083 | 0.963636 | 0.523260 |

Table A.10: Grid search results for cosine similarity thresholds in **Xception Pre-Trained**. Metrics are shown for both **Matching** and **Added-object** detection.

| Threshold | Matching | | | Added Object | | |
|---|---|---|---|---|---|---|
| | **F1** | **Recall** | **Precision** | **F1** | **Recall** | **Precision** |
| 0.60 | 0.611236 | 0.529183 | 0.723404 | 0.746193 | 0.809090 | 0.641921 |
| 0.62 | 0.596811 | 0.509728 | 0.719780 | 0.740000 | 0.896970 | 0.629787 |
| 0.64 | 0.587963 | 0.494163 | 0.725714 | 0.737101 | 0.909091 | 0.619835 |
| 0.66 | 0.572104 | 0.470817 | 0.728916 | 0.725962 | 0.915152 | 0.601594 |
| 0.68 | 0.565947 | 0.459144 | 0.737500 | 0.715646 | 0.915152 | 0.587549 |
| 0.70 | 0.561743 | 0.451362 | 0.743590 | 0.708920 | 0.915152 | 0.578544 |
| 0.72 | 0.563725 | 0.447471 | 0.761589 | 0.709977 | 0.927273 | 0.575188 |
| 0.74 | 0.544081 | 0.420233 | 0.771429 | 0.701357 | 0.939394 | 0.559567 |
| 0.76 | 0.519481 | 0.389105 | 0.781250 | 0.696635 | 0.957576 | 0.546713 |
| 0.78 | 0.486631 | 0.354086 | 0.777778 | 0.679570 | 0.957576 | 0.526667 |
| 0.80 | 0.425770 | 0.295720 | 0.760000 | 0.655602 | 0.957576 | 0.498423 |

Table A.11: Grid search results for cosine similarity thresholds in **Xception Phase 1**. Metrics are shown for both **Matching** and **Added-object** detection.

| Threshold | Matching | | | Added Object | | |
|---|---|---|---|---|---|---|
| | **F1** | **Recall** | **Precision** | **F1** | **Recall** | **Precision** |
| 0.60 | 0.667954 | 0.673152 | 0.662835 | 0.722741 | 0.703030 | 0.743590 |
| 0.62 | 0.661479 | 0.661479 | 0.661479 | 0.726154 | 0.715152 | 0.737500 |
| 0.64 | 0.648221 | 0.638132 | 0.658635 | 0.732733 | 0.739394 | 0.726190 |
| 0.66 | 0.655936 | 0.634241 | 0.679167 | 0.748538 | 0.775758 | 0.723164 |
| 0.68 | 0.652977 | 0.618677 | 0.691304 | 0.755682 | 0.806061 | 0.711320 |
| 0.70 | 0.656004 | 0.610895 | 0.710487 | 0.749722 | 0.818182 | 0.688776 |
| 0.72 | 0.652268 | 0.587549 | 0.733810 | 0.750000 | 0.854545 | 0.668246 |
| 0.74 | 0.644295 | 0.560311 | 0.757895 | 0.739976 | 0.878788 | 0.638767 |
| 0.76 | 0.615741 | 0.517510 | 0.760000 | 0.717445 | 0.884848 | 0.603306 |
| 0.78 | 0.591346 | 0.478599 | 0.773585 | 0.709220 | 0.909091 | 0.581395 |
| 0.80 | 0.538653 | 0.420233 | 0.750000 | 0.684932 | 0.909091 | 0.549451 |

Table A.12: Grid search results for cosine similarity thresholds in **Xception Phase 2**. Metrics are shown for both **Matching** and **Added-object** detection.

# Bibliography


[1] S. Kaza, L. Yao, P. Bhada-Tata, and F. Van Woerden, *What a Waste 2.0: A Global Snapshot of Solid Waste Management to 2050*. World Bank Publications, 2018.

[2] United Nations Environment Programme, *Global waste-management outlook 2024*, Nairobi: UNEP, 2024.

[3] World Bank, *Solid waste management – urban development brief*, `https://www.worldbank.org/en/topic/urbandevelopment/brief/solid-waste-management`, 2020.

[4] E. Trapen *et al.*, «Environmental sustainability impacts of solid waste management», *Journal of Cleaner Production*, vol. 389, p. 136 862, 2023.

[5] A. López *et al.*, «Ghg emission from trucks during collection of solid wastes», *Eurasian Journal of Environmental Research*, vol. 2, no. 2, pp. 25–34, 2019.

[6] Oro Loma Sanitary District & Waste Management Inc., *Contamination and overage monitoring programme – factsheet*, 2023.

[7] Sensoneo, *Waste fill-level monitoring with smart sensors*, `https://www.sensoneo.com`, 2025.

[8] M. P. Arthur, S. Shoba, and A. Pandey, «A survey of smart dustbin systems using the iot and deep learning», *Artificial Intelligence Review*, vol. 57, no. 3, pp. 2135–2172, 2024.

[9] D. G. Lowe, «Distinctive image features from scale-invariant keypoints», in *International journal of computer vision*, vol. 60, Springer, 2004, pp. 91–110.

[10] H. Bay, T. Tuytelaars, and L. Van Gool, «Surf: Speeded up robust features», in *European conference on computer vision*, Springer, 2006, pp. 404–417.

[11] K. Mikolajczyk and C. Schmid, «A comparison of affine region detectors», in *International journal of computer vision*, vol. 65, Springer, 2005, pp. 43–72.

[12] A. S. Razavian, H. Azizpour, J. Sullivan, and S. Carlsson, «Cnn features off-the-shelf: An astounding baseline for recognition», *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, pp. 806–813, 2014.

[13] A. Babenko, A. Slesarev, A. Chigorin, and V. Lempitsky, «Aggregating local deep features for image retrieval», in *International conference on computer vision*, IEEE, 2015, pp. 1269–1277.

[14] A. Gordo, J. Almazán, J. Revaud, and D. Larlus, «End-to-end learning of deep visual representations for image retrieval», in *International journal of computer vision*, vol. 124, Springer, 2017, pp. 237–254.

[15] R. Hadsell, S. Chopra, and Y. LeCun, «Dimensionality reduction by learning an invariant mapping», in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE, 2006, pp. 1735–1742. DOI: `10.1109/CVPR.2006.100`.

[16] S. Chopra, R. Hadsell, and Y. LeCun, «Learning a similarity metric discriminatively, with application to face verification», in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE, 2005, pp. 539–546. DOI: `10.1109/CVPR.2005.202`.

[17] K. Q. Weinberger and L. K. Saul, «Distance metric learning for large margin nearest neighbor classification», in *Advances in neural information processing systems*, 2009, pp. 1473–1480.

[18] F. Schroff, D. Kalenichenko, and J. Philbin, «Facenet: A unified embedding for face recognition and clustering», *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 815–823, 2015.

[19] G. Koch, «Siamese neural networks for one-shot image recognition», University of Toronto, Tech. Rep., 2015, Technical Report.

[20] L. Bertinetto, J. Valmadre, J. F. Henriques, A. Vedaldi, and P. H. S. Torr, «Fully-convolutional siamese networks for object tracking», in *European Conference on Computer Vision (ECCV)*, Springer, 2016, pp. 850–865. DOI: 10.1007/978-3-319-48881-3_56.

[21] K. Sohn, «Improved deep metric learning with multi-class n-pair loss objective», in *Advances in neural information processing systems*, 2016, pp. 1857–1865.

[22] W. Sun, S. Feng, and Y. Li, «Circle loss: A unified perspective of pair similarity optimization», in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 6398–6407.

[23] X. Li, H. Hu, Z. Lin, Y. Wang, J. Ponce, and A. Yuille, «Revisiting non-local neural networks for image recognition and retrieval», in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 2032–2041.

[24] F. Radenović, J. Araújo, J. Almazán, and T. Darrell, «Fine-tuning cnn image retrieval with no human annotation», in *IEEE transactions on pattern analysis and machine intelligence*, vol. 41, 2018, pp. 1655–1668.

[25] J. Bromley, I. Guyon, Y. LeCun, E. Säckinger, and R. Shah, «Signature verification using a siamese time delay neural network», *International Journal of Pattern Recognition and Artificial Intelligence*, vol. 7, no. 4, pp. 669–688, 1993. DOI: 10.1142/S0218001493000339.

[26] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf, «Deepface: Closing the gap to human-level performance in face verification», in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE, 2014, pp. 1701–1708. DOI: 10.1109/CVPR.2014.220.

[27] A. Rosebrock, «Siamese network with keras, tensorflow, and deep learning», *PyImageSearch*, Nov. 2020, [Online; accessed 3-July-2025]. [Online]. Available: https://pyimagesearch.com/2020/11/30/siamese-networks-with-keras-tensorflow-and-deep-learning/.

[28] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, «Gradient-based learning applied to document recognition», in *Proceedings of the IEEE*, vol. 86, IEEE, 1998, pp. 2278–2324.

[29] A. Krizhevsky, I. Sutskever, and G. E. Hinton, «Imagenet classification with deep convolutional neural networks», in *Advances in Neural Information Processing Systems (NeurIPS)*, 2012, pp. 1097–1105.

[30] K. Simonyan and A. Zisserman, «Very deep convolutional networks for large-scale image recognition», in *International Conference on Learning Representations (ICLR)*, arXiv:1409.1556, 2015.

[31] C. Szegedy, W. Liu, Y. Jia, *et al.*, «Going deeper with convolutions», in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015, pp. 1–9.

[32] K. He, X. Zhang, S. Ren, and J. Sun, «Deep residual learning for image recognition», in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2016, pp. 770–778.

[33] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, «Mobilenetv2: Inverted residuals and linear bottlenecks», in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2018, pp. 4510–4520.

[34] F. Chollet, «Xception: Deep learning with depthwise separable convolutions», in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE, 2017, pp. 1251–1258. [Online]. Available: https://arxiv.org/abs/1610.02357.

[35] K. He, X. Zhang, S. Ren, and J. Sun, «Deep residual learning for image recognition», in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[36] M. Lin, Q. Chen, and S. Yan, «Network in network», in *International Conference on Learning Representations (ICLR)*, arXiv:1312.4400, 2014.

[37] S. Das, A. A. Fime, N. Siddique, and M. M. A. Hashem, «Estimation of road boundary for intelligent vehicles based on deeplabv3+ architecture», *IEEE Access*, vol. 9, pp. 114–125, 2021. DOI: 10.1109/ACCESS.2020.3048272.

[38] S. Bianco, R. Cadene, L. Celona, P. Napoletano, and R. Schettini, «Benchmark analysis of representative deep neural network architectures», *IEEE Access*, vol. 6, pp. 64 270–64 277, 2018.

[39] L. Sifre and S. Mallat, *Rigid-motion scattering for image classification*, arXiv preprint arXiv:1403.1687, 2014.

[40] B. Abraham and M. S. Nair, «Covid-19 detection using cnn transfer learning from x-ray images», *Computer Methods and Programs in Biomedicine*, vol. 199, p. 105 581, 2021, https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7759122/. DOI: 10.1016/j.cmpb.2020.105581.

[41] K. Shankar, J. S. Kumar, and S. R. Kumar, «Mobileskin: A lightweight mobilenetv2-based cnn for classification of systemic sclerosis skin images», *Biomedical Signal Processing and Control*, vol. 70, p. 102 993, 2021.

[42] GeeksforGeeks, *Mobilenet v2 architecture in computer vision*, Accessed: 2025-06-05, 2023. [Online]. Available: https://www.geeksforgeeks.org/mobilenet-v2-architecture-in-computer-vision/.

[43] S. Mehta, T. Lee, and et al., «Efficient deep learning models for classification of systemic sclerosis from skin images», *Computer Methods and Programs in Biomedicine*, vol. 199, p. 105 906, 2021.

[44] X. Zhang, X. Zhou, M. Lin, and J. Sun, «Shufflenet: An extremely efficient convolutional neural network for mobile devices», *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 6848–6856, 2018.

[45] A. T. Tragoudaras, P. Stoikos, K. Fanaras, and G. Stamoulis, «Design space exploration of a sparse mobilenetv2 using high-level synthesis and sparse matrix techniques on fpgas», *Sensors*, vol. 22, no. 12, p. 4567, 2022.

[46] R. Roopashree and R. Anitha, «Deepherb: A vision-based system for medicinal plant identification using xception deep learning model», in *2021 International Conference on Intelligent Technologies (CONIT)*, IEEE, 2021, pp. 1–6. [Online]. Available: https://www.semanticscholar.org/paper/DeepHerb%3A-A-Vision-Based-System-for-Medicinal-using-Roopashree-Anitha/cfc002389353d10ba7d6bef73714f948fc92d119.

[47] A. Hermans, L. Beyer, and B. Leibe, «In defense of the triplet loss for person re-identification», *arXiv preprint arXiv:1703.07737*, 2017.

[48] A. Rosebrock, *Deep metric learning with triplet loss and keras*, https://pyimagesearch.com/2020/04/13/deep-metric-learning-with-triplet-loss-and-keras/, 2020.

[49] U. Karn, *Triplet loss – advanced intro*, Accessed May 2025, 2020. [Online]. Available: https://medium.com/data-science/triplet-loss-advanced-intro-49a07b7d8905.

[50] C. Shorten and T. M. Khoshgoftaar, «A survey on image data augmentation for deep learning», *Journal of Big Data*, vol. 6, no. 1, pp. 1–48, 2019.