Samuel Olatunde
Malle-Project 1st Iteration Report

Introduction

Duplicate and altered versions of digital content frequently appear across online platforms, often in the form of compressed, cropped, or otherwise transformed images. Traditional hashing techniques such as pHash or dHash struggle to recognize these modified copies because they rely on strict pixel-level similarity. This project explores a more robust, learning-based approach to detecting visually similar content using deep feature representations.

The first iteration of the system focuses on an embedding backend built with a pretrained ResNet-50 convolutional neural network. Each image is converted into a high-dimensional feature vector, and Facebook AI Similarity Search (FAISS) is used to efficiently retrieve the nearest matches. To evaluate how well deep embeddings handle real-world distortions, the system uses a subset of ImageNet Mini [2] as the source of “original” images and generates systematically augmented copies through cropping, rotation, blurring, color shifts, and compression. These modified versions serve as query images for testing retrieval accuracy under controlled perturbations. A baseline comparison against perceptual hashing is also included to quantify the performance gap between classical and learning-based methods.

This work operates exclusively on publicly available datasets and does not process real user content. The goal is to provide a technical demonstration of machine learning techniques for image similarity and to assess their robustness across a range of image transformations.

Background

Traditional near-duplicate detection relies heavily on perceptual hashing, with methods like pHash offering compact signatures that work well for exact or lightly altered copies but break down under stronger transformations such as cropping, rotation, or color shifts [5]. More recent approaches shift toward deep visual embeddings, where convolutional networks learn feature spaces that remain stable despite substantial edits. A key example is FaceNet, which demonstrated that embedding-based similarity learning using triplet loss can achieve far more robust matching than handcrafted methods [6]. These developments motivate the use of pretrained CNNs such as ResNet-50 for general-purpose similarity search, providing a stronger foundation for modified-image retrieval compared to classical hashing.

Methodology

This project integrates a set of tools and libraries that together support image preprocessing, feature extraction, similarity indexing, and evaluation. ImageHash is used to compute perceptual hashes (pHash), providing a lightweight baseline for comparing images based on coarse visual structure. Pickle serves as a serialization layer for storing embeddings, hash dictionaries, and intermediate artifacts, enabling reproducibility without repeated computation. FAISS functions as the core similarity-search engine, offering efficient indexing and retrieval of high-dimensional vectors at scale.

For image handling, Pillow manages loading, resizing, and basic transformations, while PyTorch and Torchvision supply the deep-learning backbone used for feature extraction. A pretrained ResNet-50 model is applied as a fixed encoder, producing 2048-dimensional embeddings for each image. NumPy supports fast array manipulation and numerical operations throughout the pipeline. Additionally, OS facilitates directory traversal and dataset management, and Shutil enables copying selected images for constructing controlled

The overall experimental workflow follows a structured pipeline that processes images, extracts embeddings, indexes them with FAISS, and evaluates retrieval performance on systematically modified image sets. This pipeline is designed to quantify how well deep visual embeddings retrieve near-duplicate content under realistic transformations.

1. Sampling Original Images

The first stage of the pipeline constructs a curated set of 200 reference images drawn from the ImageNet-Mini validation split. The extraction script begins by creating the directory malle_dataset/original_images and enumerating all class subfolders within imagenet-mini/val. A fixed random seed ensures that the sampling procedure is fully deterministic and reproducible across runs.

The workflow proceeds in two phases. In the first phase, the script randomly selects twenty class folders and copies every image within those folders into the output directory. To prevent filename collisions and to implicitly preserve class identity, each copied image is renamed to include its class (folder) name as a prefix. The script also records all copied file paths so that no image is duplicated later. After this initial pass, the script checks whether the target count of 200 images has been reached.

If the first phase yields fewer than 200 images, the second phase begins. Here, the script compiles a list of all remaining images across the entire validation split, filters out those already copied, and randomly samples from this pool to fill the remaining quota. These supplemental images are copied using the same collision-proof renaming

scheme. Since ImageNet-Mini is sufficiently large, the remaining pool always contains more than enough candidates to reach the exact target size.
The final output is a reproducible set of exactly 200 mixed-category images stored in a single directory, each following the standardized filename pattern {class_name}_{instance_id}.JPEG. This procedure guarantees diversity, avoids duplicates, preserves class metadata, and establishes a consistent foundation for downstream feature extraction and similarity-search experiments.

2. Generating Modified Copies

The second stage of the pipeline constructs a diverse set of modified images to evaluate the robustness of similarity-search methods under realistic distortions. The script begins by enumerating all 200 original images and seeding the random number generator to ensure reproducibility. Each image is assigned a number of modified variants, randomly chosen to be either 4 or 8. All images are first converted from PIL format into normalized float tensors in the [0,1] range to ensure consistent processing across transformations.
A wide range of transformations is supported, each chosen to simulate distortions commonly encountered when images are reposted, edited, or recompressed online. Available operations and their effects are highlighted in Table 1 below.

Table 1: Detailed Table of Image Modifications

Modification	What It Does	Values Used & Behavioral Notes
Center Crop	Removes edges and keeps the central region.	Crop sizes are randomly chosen from 256 or 384 px. Ensures all crops remain centered, enforcing consistent spatial bias. Helps test robustness against missing context.
Random Crop	Crop a patch from a random location.	Crop sizes 256 or 384 px, with padding if needed. Produces off-center subregions that may exclude key objects, stressing spatial invariance.

Random Resize	Rescales the entire image to a random size.	Output dimension sampled within [256, 384] px. Mimics differences in camera resolution or zoom. Prevents the model from overfitting to fixed-scale features.
Rotation	Rotates the image slightly clockwise or counterclockwise.	Angle sampled from –15° to +15°. These small rotations represent natural hand-held camera tilt.
Gaussian Blur	Softens the image using a Gaussian filter.	Kernel size depends on image dimensions: 3, 5, or 7, chosen based on min(height, width). Produces mild, medium, or strong blur—realistic for motion blur or low-quality cameras.
Brightness & Contrast Jitter	Randomly changes overall brightness and contrast.	Brightness factor 0.8–1.2, contrast factor 0.6–1.4. Allows significant lighting variation to test illumination invariance.
Hue & Saturation Jitter	Alters color tones and intensity.	Hue shift –0.5 to 0.5, saturation 0.6–1.4. Generates strong color deviations like filters, color drift, or sensor inconsistencies.
Watermark	Overlays semi-transparent text on the image.	Text: “Malle Project”, font size 80, opacity 120, random position. Models social-media watermarks or tampering with ownership marks.
JPEG Compression	Re-encodes image with lossy compression.	Quality randomly sampled from 30–80. Low values introduce block artifacts and noise like compressed uploads or messaging apps.
Occlusion	Covers a region with a black rectangular block.	Mask size 80×80 px, random (x, y) location. Simulates partial obstruction (hands, stickers, shadows, UI elements).

For every modified sample, the script randomly selects a subset of these transformations and applies them in a random order Once the sequence of transformations is complete, the final tensor is converted back to a PIL image and saved in the modified_images directory. .Each modified image is saved using a filename that encodes both its original class/instance identity and the full sequence of applied transformations. The naming structure follows the pattern

{class_name}_{instance_id}_{op1}_{op2}_..._{opN}.JPEG

e.g.,

n01496331_00037260_centCrop384_rotate_watermark_resize_color.JPEG,

where the prefix identifies the source image and the appended tags document the exact modification pipeline. This convention ensures complete traceability and allows transformations to be programmatically parsed during analysis. The resulting dataset contains a rich and controlled set of distortions that span changes in geometry, color, lighting, compression artifacts, and partial occlusion. This synthetic augmentation process provides a rigorous test bed for evaluating embedding robustness and retrieval performance across both learning-based and perceptual hashing methods.

3. Feature Extraction Using ResNet-50

The third stage of the pipeline computes deep feature embeddings for both the original (index) images and their modified counterparts (queries) using a pretrained ResNet-50 model configured as a feature extractor. The script begins by selecting the compute device (CPU or GPU), loading the pretrained ResNet-50, and removing its final fully connected classification layer. This exposes the 2048-dimensional output of the penultimate layer, which serves as a stable, high-level representation suitable for similarity search. A standard ImageNet preprocessing pipeline—resize, center crop, tensor conversion, and channel-wise normalization—is applied to ensure compatibility with the model's training distribution.

Image embedding is performed by the embed_image() function, which loads each image, applies the preprocessing transforms, forwards it through the truncated network, and l2-normalizes the resulting vector to unit length. This normalization is essential because FAISS later uses inner-product similarity, which becomes equivalent to cosine similarity on normalized embeddings. Metadata extraction—recovering class name, instance ID, and, for modified images, the full list of applied transformations—is implemented inside the embed_folder() function, which parses each filename as it iterates through the dataset.

The embed_folder() function computes embeddings for all images in a specified directory, aggregates them into a two-dimensional NumPy array (the format required by FAISS), and stores both the embeddings and metadata using Python's pickle module. When executed, the script produces two serialized embedding files: one for the 200 original reference images and one for all modified query images. These embedding files form the basis for the subsequent similarity indexing and retrieval experiments.

4. FAISS Indexing and Similarity Search

The similarity search stage loads the precomputed ResNet-50 embeddings for the 200 original images (the index set) and initializes a FAISS index for nearest-neighbor retrieval. Because the dataset is relatively small, the system uses faiss.IndexFlatIP, an exact inner-product index that computes pairwise similarities without approximation. Since all embeddings were l2-normalized during extraction, inner-product search is equivalent to cosine-similarity search, allowing efficient and mathematically consistent ranking. The dimensionality of the embeddings is inferred directly from the loaded index vectors, and all index embeddings are inserted into the FAISS structure before the index is serialized to disk for reuse.

The script then loads the embeddings corresponding to the modified images (the query set), along with metadata identifying each query image—its class name, instance ID, transformation history, and file path. Query embeddings are passed to FAISS using the index.search() interface, which returns the top-k most similar index vectors for every query. FAISS outputs two matrices: a similarity matrix S, containing cosine-similarity scores, and an index matrix I identifying which original images were retrieved for each query. To support downstream evaluation, the script iterates through these retrieval results and augments each FAISS match with the full metadata of the corresponding index image. This includes class and instance identifiers, enabling fine-grained evaluation at both class-level and instance-level resolution. The final output is a structured retrieval record for each query containing its top-k matched images and associated similarity scores. This complete result set is serialized with pickle and serves as the input to the recall and accuracy analysis conducted in the final stage of the pipeline.

5. Evaluation of Top-K Recall

5.1 FAISS Retrieval Evaluation

For evaluation, the system loads the previously generated FAISS retrieval outputs, which already contain each query image and its ranked top-K matches (with full metadata). The script iterates through these results and computes two metrics:

- **Class-level Recall@K**: counts a true positive if the correct class appears anywhere within the top-K list. A small guard (is_counted_already) ensures the class is counted at most once even if multiple retrieved images belong to the same class.
- **Instance-level Recall@K**: only counted when the correct class is retrieved and the correct instance ID appears within that class. This avoids false positives from cross-class instance-ID overlaps.

These metrics collectively measure how well the embedding-based retrieval system maintains semantic similarity (class) and exact identity matching (instance) under distortions.

5.2 pHash Retrieval Baseline

The pHash pipeline provides a classical, non-deep baseline. For each query image, the script computes the Hamming distance between its pHash and all index hashes, ranks index images by increasing distance, and records the top-K closest matches.

Evaluation mirrors the FAISS process:

- **Class-level Recall@K**: checks whether any top-K match shares the query's class.
- **Instance-level Recall@K**: checks for both the correct class and the matching instance ID within that class.

This allows a direct comparison between perceptual hashing and deep-feature embeddings in terms of robustness to distortions and fine-grained identity recovery.

Results

Table 2: Instance Recall@K

K	ResNet-50	pHash
1	0.9425	0.5486
3	0.9662	0.6036
5	0.9763	0.6281
10	0.9814	0.6779
11	0.9822	0.6855

Table 3:Class Recall@K

K	ResNet-50	pHash
1	0.9518	0.5511
3	0.9679	0.6120
5	0.9797	0.6407
10	0.9831	0.6974
11	0.9839	0.7033

Analysis

As shown in Table 2 and Table 3, ResNet-50 markedly outperforms pHash across all evaluated K values, achieving Instance Recall@1 = 0.9425 and Class Recall@1 = 0.9518, compared with pHash's substantially lower 0.5486 and 0.5511, respectively.

Performance for ResNet-50 saturates rapidly—with Instance Recall increasing only from 0.9763 (K=5) to 0.9822 (K=11)—whereas pHash shows slower but continuous improvement, rising from 0.5486 (K=1) to 0.6855 (K=11) and indicating its dependence on larger retrieval windows.

Both methods exhibit slightly higher class-level than instance-level recall; however, the gap for ResNet-50 remains below 1% across all K values, while pHash maintains a wider 2–3% separation, demonstrating the CNN's stronger ability to preserve instance identity under distortion.

Collectively, the results in Tables 2 and 3 highlight that deep features extracted by ResNet-50 yield consistently strong performance—even exceeding 98% recall at K=10—whereas pHash's sensitivity to geometric and photometric perturbations limits its effectiveness for fine-grained image retrieval.

Conclusion

Although the initial plan was to scale the retrieval system to OpenAI's CLIP model, this direction was ultimately postponed. Given that ResNet-50 already achieved strong performance—over 94% Recall@1 and up to 98% Recall@10—there was limited practical benefit to introducing CLIP within the constraints of this project. CLIP is fully capable of operating on datasets it was not explicitly optimized for, but the additional setup complexity, combined with the already high retrieval accuracy from ResNet-50, made the transition unjustified for this iteration.

Future work will focus on evaluating CLIP in contexts where it excels: zero-shot retrieval on diverse, open-domain images far beyond ImageNet categories [7]. Scaling to CLIP will allow the system to handle more complex, unfamiliar, and semantically rich queries that traditional CNNs are not designed for.

Beyond model expansion, future development will involve moving away from ImageNet-Mini and coupling the system with Google's Image Search API, enabling large-scale web-image retrieval. This shift will support the long-term goal of transforming the pipeline into a usable product, complete with a frontend interface and public APIs for image upload, retrieval, and result visualization.

Overall, the strong results obtained with ResNet-50 provide a reliable foundation for scaling up, validating the system design and motivating the transition toward more flexible, production-level image retrieval in future work.

References

[1] PyTorch, “PyTorch documentation — PyTorch 2.7 documentation,” Pytorch.org, 2024. https://docs.pytorch.org/docs/stable/index.html

[2]Ilya Figotin, “ImageNet 1000 (mini),” Kaggle.com, 2020. https://www.kaggle.com/datasets/ifigotin/imagenetmini-1000?resource=download (accessed Dec. 11, 2025).

[3]“Getting started,” GitHub. https://github.com/facebookresearch/faiss/wiki/Getting-started

[4]DigitalSreeni, “348 - Image Similarity Search with VGG16 and Cosine Distance,” YouTube, Dec. 18, 2024. https://www.youtube.com/watch?v=dCcRWdmmgA0 (accessed Dec. 11, 2025).

[5]C. Zauner, “Implementation and Benchmarking of Perceptual Image Hash Functions,” Jul. 2010. Accessed: Dec. 11, 2025. [Online]. Available: https://www.phash.org/docs/pubs/thesis_zauner.pdf?utm_source=chatgpt.com

[6]F. Schroff, D. Kalenichenko, and J. Philbin, “FaceNet: A unified embedding for face recognition and clustering,” 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Jun. 2015, doi: https://doi.org/10.1109/cvpr.2015.7298682.

[7] openai, “GitHub - openai/CLIP: CLIP (Contrastive Language-Image Pretraining), Predict the most relevant text snippet given an image,” *GitHub*, 2025. https://github.com/openai/CLIP?tab=readme-ov-file