# Assessment Report

Candidate: Aman Gupta
Role Applied: Software Engineer - Vision & AI/ML
Company: Terafac Technologies

# Exact Image Search in a Bit-Level Image Database

## 1. Problem Statement

Given a dataset of 100×100 RGB images with 1-bit channels, the task was to design an efficient program to search whether a given image exists in the dataset. Additionally, I was asked to analyze time complexity, memory usage, and scaling behavior with increasing dataset sizes.

## Alternate Approaches Considered

- Brute-Force Matching: Compare each image pixel-by-pixel. Accurate but O(N) query time.
- Image Hashing with List Storage: Hash images and store in a list. Still O(N) due to list traversal.
- Exact Hash-Based Lookup with Dict: Hash images and store them in a Python dictionary for O(1) average-time lookup using hash table mechanics. Chosen for its high efficiency and simplicity for exact matches.

## 2.Chosen Approach: Exact Hash-Based Lookup

I opted for a high-efficiency method using image hashing combined with a Python dictionary for constant-time lookup.
Step-by-Step Strategy:

- Dataset Generation
  Generate N images with pixel values in {0,1} for all RGB channels using NumPy.
  ```
  def generate_bit_images(N):
      return np.random.randint(0, 2, (N, 100, 100, 3), dtype=np.uint8)
  ```

- Index Building
  Convert each image to bytes, hash it, and store it in a dictionary.
  ```
  def hash_bits(img):
      return hash(img.tobytes())

  def build_index(images):
      return {hash_bits(img): True for img in images}
  ```

- Image Lookup
  Hash the query image and check existence in the index.
  ```
  found = hash_bits(query_img) in index
  ```

## 3.Performance Evaluation

Logged generation time, memory usage, index build time, and query time for different dataset sizes using TensorBoard and DataFrames.

## 4.Observations

- Query Time: Remains O(1) due to hash table lookup.
- Memory Usage: Scales linearly with dataset size.
- Index Time: Also linear, but acceptable.

## 5.Shortcomings & Considerations

- Exact Match Only: Cannot handle slight variations, distortions, or lossy encoding.
- Hash Collisions: Though rare, possible.
- Memory Bound: At very high N, RAM becomes a bottleneck (especially in Colab). This limitation could potentially be addressed by using a disk-based key-value store like RocksDB, which would allow for scalable indexing and lookup without loading the entire dataset into memory. Currently I just flush the memory by manually calling the garbage collector.

### 7. Conclusion
The image data generation and memory footprint and image hash dictionary creation all of these had a time complexity of O(n). while the image search time complexity for exact match was O(1).

# Image Similarity Search using Machine Learning

## 1. Problem Statement
The objective of this assessment was to design and implement a system capable of finding visually similar images from a dataset using machine learning.

## 2. Initial Exploration and Approach Considerations
I had several approaches that i found for consideration

- Pretrained CNN Feature Extraction (e.g., ResNet, VGG): Using off-the-shelf CNNs to extract features, followed by L2/FAISS-based similarity search. Simple and effective, but lacks task-specific representation learning.
- Vision Transformers (ViTs): Architectures like DINOv2 or CLIP use self-supervised or contrastive methods to learn strong semantic features.
- Supervised Classification Features: Using labels to train a classifier and utilizing embeddings from penultimate layers. However, this approach requires labeled data.

Given the goal of building a similarity search engine with minimal labeled data, contrastive learning (SimCLR-style) was chosen as the most appropriate method. This self-supervised strategy enables the model to learn useful representations by comparing different augmentations of the same image (positives) against other images (negatives).

## 3. Chosen Approach: SimCLR with ResNet-18 Backbone

### 3.1 Dataset
The COCO-128 dataset was used, downloaded via Roboflow. This small subset of COCO contains 128 images in the training set. While suitable for proof-of-concept, it poses challenges for contrastive training due to its limited size.

### 3.2 Data Augmentation
SimCLR heavily relies on strong augmentations to generate different views of the same image. The following transformations were used:
```
T.RandomResizedCrop(224, scale=(0.2, 1.0))
T.RandomHorizontalFlip()
T.ColorJitter(0.4, 0.4, 0.4, 0.1)
T.GaussianBlur(5)
T.ToTensor()
T.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
```

### 3.3 Model Architecture
- Encoder: ResNet-18 without the final classification head.
- Projection Head: A small MLP with two layers to project 512-D features to a 128-D embedding space.
```
nn.Sequential(
  nn.Linear(512, 256),
  nn.ReLU(),
  nn.Linear(256, 128)
)
```
- Normalization: The output is normalized to lie on the unit hypersphere (important for cosine similarity).

### 3.4 Loss Function: InfoNCE
The contrastive InfoNCE loss compares all positive pairs against all negatives in the batch, using cosine similarity and a temperature-scaled softmax. Positive pairs are augmented views of the same image, and the rest serve as negatives.

### 3.5 Hyperparameters

Several key hyperparameters influenced model performance:

- Batch Size: 64. A modest size, constrained by dataset size (128 images) and compute. Larger sizes (e.g., 256+) would help create more negative pairs.
- Embedding Dimension: 128. Balances compactness and expressivity of learned representations.
- Temperature (τ): 0.1. Controls the sharpness of the softmax distribution in InfoNCE. Lower values increase emphasis on harder negatives.
- Learning Rate: Default for Adam optimizer, not specifically tuned here but can be explored.
- Epochs: 30. Chosen to ensure convergence given the small dataset.

### 3.6 Training

The model was trained for 30 epochs using a batch size of 64. Since the dataset contains only 128 images, training was effectively done in 2 batches per epoch. The Adam optimizer was used for optimization.

---

## 4. Evaluation Methodology

- Embedding Extraction: After training, all images were passed through the model to generate embeddings.
- FAISS Indexing: Embeddings were stored in a FAISS IndexFlatL2 structure for efficient nearest-neighbor search.
- Querying: A query image was passed through the model, and its embedding was used to search the FAISS index for the top-k similar images.

---

## 5. Limitations

- Dataset Size: The COCO-128 dataset is too small for effective contrastive training. Larger datasets like ImageNet or OpenImages would produce more robust representations.
- Batch Size: SimCLR benefits greatly from large batch sizes (e.g.,512+) to provide diverse negative samples. Although this experiment used a batch size of 64, the dataset size was only 128 images, resulting in just 2 batches per epoch—still limiting the diversity of negative samples compared to larger datasets.
- Training Time: Without proper hardware or augmentation caching, training was slow.
- No Hard Negative Mining: All negatives are equally treated, which could be improved by using semi-hard/hard negatives.

---

## 6. Alternative & Advanced Methods to Explore

### 6.1 DINO / DINOv2

- Self-supervised ViT-based method by Meta.
- Produces rich features useful for clustering, retrieval, segmentation.
- Requires more compute but no labels.

### 6.2 CLIP (Contrastive Language-Image Pretraining)

- Joint vision-language model by OpenAI.
- Learn aligned vision-text embeddings.
- Pretrained on large datasets, shows strong generalization.
- Could be used to find similar images based on textual query as well.

### 6.3 MoCo (Momentum Contrast)

- More memory-efficient alternative to SimCLR.
- Maintains a queue of negative samples instead of large batch sizes.

### 6.4 Supervised ViT Embeddings

- If labels are available, supervised ViTs (e.g., DeiT, Swin) can be used and embeddings from intermediate layers can be used.

---

## 7. Conclusion

Increasing batch size, using a larger and more diverse dataset, and experimenting with DINO or CLIP-based models would significantly improve performance and generalization.