

# Rasterized Image Databases with LSH for Compression, Search and Duplicate Detection

Zoya Bylinskii  
MIT CSAIL

Andrew Spielberg  
MIT CSAIL

Maria Shugrina  
MIT CSAIL

Wei Zhao  
MIT CSAIL

## ABSTRACT

TODO.

## 1. INTRODUCTION

Large collections of images are ubiquitous in the modern digital world. According to one 2014 Internet Trends report, more than 1.8 billion images are uploaded to the internet every day [?]. Our work is inspired by the intuition that there must be a lot of redundancy in large image collections, and that this redundancy could be exploited for more efficient storage and for applications such as duplicate detection.

We focus on image redundancy on the patch level, assuming that large collections of images must have many patches which are nearly the same. Our goal is to store a set of images as a database of similar patches, where similar patches may be shared between images, such that we minimize the storage space while maintaining certain *quality* of reconstructed images. In effect, this results in lossy compression. More concretely, we aim to choose patch similarity criterion, and search and reconstruction algorithms such that:

- the database size is smaller than if full images were stored
- the images can be reconstructed from the database in real time
- the reconstructed images fulfill certain quality requirements (See Sec. 4)

These conflicts introduce a number of tradeoffs, such as size of the database versus image quality. The goal of this paper is as much to produce a working system as to build up the analytical foundations that allow making these tradeoffs.

In Section 3, we explain our method of “patchifying” images, storing them in a database, and reconstructing stored images. We also discuss image similarity, hashing schemes and indices in this core section. In Section 4, we provide analytical groundwork for selecting optimal image patch sizes,

similarity threshold, and the quality metrics appropriate for evaluation. Finally, in Section 5 we evaluate our system on real data, using analytical tools from the previous section. In addition, in Section 6, we briefly touch on applications that come naturally from a patch database, including similar image retrieval and duplicate detection.

## 2. RELATED WORK

Image databases, in particular rasterized.  
Image compression, in particular JPEGs.  
Image similarity functions.  
Image quality functions.  
Hashing, in particular [?].

## 3. METHOD

To limit the complexity, we assume each image to be a square of  $m^2$  pixels, and for all patches to be squares of the same size. We formally describe our method in Sec. 3.1. To summarize, we first segment each image into patches and store them in the `patches` table. Only patches sufficiently different from all the patches in `patches` are stored (See Sec. 3.2). (TODO: is this true?) In Sec. 3.3, we describe the search algorithm used to quickly retrieve similar patches, and in Sec. 3.4 we detail the indices that make image reconstruction faster. Finally, in Sec. 3.5 we provide implementation details.

### 3.1 Overview

Our method is governed by the following parameters:

- $m$  - The width and height of all images.
- $n$  - The width and height of all patches<sup>1</sup>.
- $i$  - The number of images in our database.
- $S$  - A similarity function  $S: \mathbb{I}_{n \times n} \rightarrow \mathbb{N}$ , where we define  $\mathbb{I}_{n \times n}$  to be the space of  $n \times n$  images. Section 3.2 details similarity measures.
- $T$  - A similarity threshold,  $T \in \mathbb{R}$ .

It is worth noting that we propose a lossy compression scheme. For the remainder of the paper, when we use the term *images* we are referring to entire images from our database. When we use the term *tiles* we are referring to small  $n \times n$  contiguous portions of images in our database. When we use the term *patches*, we are specifically referring

<sup>1</sup>We assume that  $m \bmod n$  is 0.

to tiles which we have chosen to store in our patch table and use for compression. Here we present an overview of our compression method; in subsequent subsections we delve into the details.

We begin our compression algorithm by creating a table of image patches to serve as our patch dictionary. These patches are chosen from randomly selected  $n \times n$  pixel tiles from the entire image database. During the compression step, we partition our images into  $\frac{m^2}{n}$  non-overlapping tiles, with the intent of mapping each tile to a patch in our patch table. Thus, rather than store the entire tile, we simply store a pointer to an entry in our pre-computed patch table. The patch we choose is the one which is closest to the original image in some similarity space  $S$ . If the closest patch in our patch table is less than  $T$  in similarity to the original image tile, we then store this tile as a new patch in our table and map the tile to itself. Assuming that our initial patch dictionary is distributed relatively uniformly over our image database, adding additional patches should be a rare procedure (TODO: how often?). Thus, the space savings come from only needing to store an effective pointer for each tile, rather than the entire tile data. Note that the minimum threshold on the similarity of tiles and patches guarantees that each compressed image is at least  $\frac{mT}{n}$  similar its original counterpart (TODO: is this true?). This procedure is shown in algorithm 1.

---

**Algorithm 1** Insert Image  $I$  into database

---

```

1: Patches  $\leftarrow$  CutIntoPatches( $I$ , patch_size= $n$ )
2: for  $P$  in Patches do
3:   SimPat  $\leftarrow$  FindLikelySimilarPatches( $P$ )
4:    $P_{closest} \leftarrow argmin\{S(P, P_i)\}$ 
5:   if then  $S(P, P_i) > T$ 
6:     insert  $P$  into Patches
```

---

With a large table of patches, finding the closest patch can be computationally expensive. In order to speed up the search, we employ *locality sensitive hashing* (LSH). Although this softens the constraint that we always find the closest patch in the database for each tile, the closest patch is still found with very high probability, and in expectation the selected patch is still very similar. Section 3.3 details this technique.

Our compression problem formally can be stated as choosing a selection of tile to patch mappings which minimizes the storage space usage of our patch table, while constraining that each image tile is at least  $T$  similar to its mapped patch. In other words,

$$\begin{aligned} & \underset{P, M}{\text{minimize}} \quad c_p(i, |P|, m, n) \\ & \text{subject to} \quad S(I_j, M(I_j)) \geq T, \quad j = 1, \dots, i. \end{aligned}$$

where  $c_p(\cdot, \cdot, \cdot)$  is a cost function as defined in section ??,  $P$  is our set of patches, and  $M: \mathbb{I}_{n \times n} \rightarrow \mathbb{I}_{n \times n}$  is our surjective mapping from image tiles to patches.

Given our pointer representation, we are able to construct the compressed image quite efficiently. Given an image identifier, we iterate over all patch pointers stored with it. TODO: Is LSH used here?

The  $n \times n$  patches are stored as byte data in the *patches* table. We store the patch pointers for each image in the

*patch\_pointers* table. The full schema looks as follows:  
TODO: update this

```

patches(id int PRIMARY KEY,
        patch bytea);

images(imgid int PRIMARY KEY);

patch_pointers(imgid int REFERENCES images(imgid),
               patch_id int REFERENCES patches(id),
               x int,
               y int);
```

where *patch\_pointers.x* and *patch\_pointers.y* refer to the left top corner location of each patch in the image.

## 3.2 Patch Similarity

There are many image similarity metrics that have been developed for images (See [?] for a good survey.), and our method is applicable to any metric that involves Euclidean distance over image features, its stacked color channel pixel values being the simplest case.

For the purpose of this project, we choose to use squared Euclidean distance over (CIE)LUV color space. Given two  $n \times n$  patches  $P_1$  and  $P_2$ , we evaluate similarity  $S$  per color channel  $i$  as follows:

$$S(P_1, P_2, i) = \frac{\|P_1(i) - P_2(i)\|^2}{n^2}$$

where  $\|\cdot\|$  denotes standard Euclidean norm. We normalize by the dimensionality of the space to allow us to keep the similarity threshold independent of the patch size. See section 4.3 for more details. A benefit of using a Euclidean distance metric is that it allows us to use LSH to retrieve patches that are likely to be similar.

## 3.3 LSH for Patches

## 3.4 Image Reconstruction

In order to reconstruct images quickly, we

## 3.5 Implementation

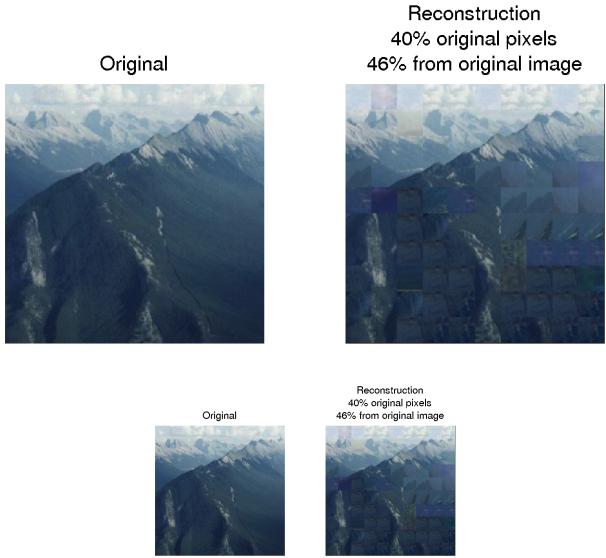
We used *postgresql* to construct our database, and used Java API to talk to the database from a custom executable. Locality sensitive hashing, image segmentation and reconstruction were all implemented in Java, and used to construct a hash table on patches in *postbresql*.

Our code is available at:

[https://github.com/shumash/db\\_project](https://github.com/shumash/db_project)

## 4. ANALYSIS

A number of parameters can be tweaked to change the patch matching and storage, and different choices may be appropriate for different applications and performance requirements (both quantitative and qualitative). These parameters include the size of the input images, the size and shape of the patches extracted, the extraction strategy (non-overlapping vs overlapping patches), the sampling strategy used to seed the dictionary, the similarity metric and thresholds used to compare patches, as well as the parameters required for indexing and retrieving patch matches (approximate nearest neighbors). Here we discuss some of the parameter choices made and the experiments that lead up

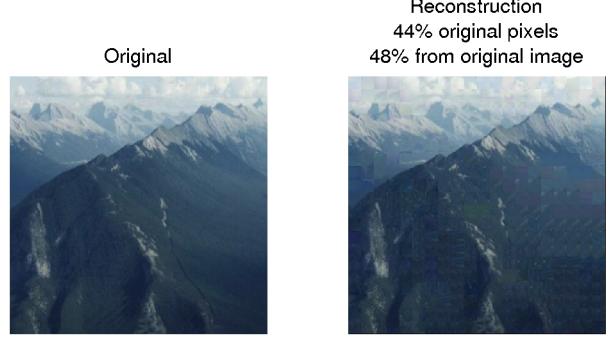


**Figure 1:** For demonstration purposes only, we choose a large patch size and low similarity threshold. Under these parameters, the original image is reconstructed to take up only 40% of its original size (in pixels). The 60% of the patches that have been replaced come either from the same image (46% of them), or from other images (the remaining 64%). Notice that when the size of the image and its reconstruction are halved, the artifacts already become visually insignificant, and would not impair a scene recognition or search task.

to these choices. Other possible choices are discussed in Sec.7.1.

Our quantitative performance metrics involve examining how the patch dictionary size grows with the addition of new images to the database (the growth function and rate) and the compression ratio per image (viewed as a distribution over compression ratios and summarized as the average compression ratio). Qualitative evaluations involve determining whether a human can spot compression artifacts and how salient they are in the images. The authors of this paper manually examined images reconstructed from the dictionary patches. A crowdsourced evaluation strategy involving Amazon’s Mechanical Turk may be appropriate for larger-scale studies, but was beyond the scope of this paper.

There will always be a trade-off between compression benefits (storage - patch dictionary size, speed - image reconstruction time) and reconstruction quality. For many computer vision tasks including scene recognition (and thus retrieval), imperfect reconstructions with artifacts may not be a problem as long as the overall scene structure does not change. For instance, [?] has shown that with images of pixel dimension 32x32, humans are already able to achieve over 80% scene and object recognition rate. See fig.1 for a demonstration of an image that has serious reconstruction artifacts, but when down sampled (to a thumbnail), they become insignificant, and thus do not necessarily impair visual recognition.



**Figure 2:** Compare this image reconstruction, computed with a dictionary of  $25 \times 25$  pixel patches with the reconstruction in fig.1, computed with  $50 \times 50$  patches. In both cases, a similar threshold is used (scaled to the patch size, as discussed in sec. ) but the visual artifacts are less noticeable because smaller patches have less contained structure, and are more likely to be homogenous in appearance.

## 4.1 Patch Size

TODO (written by Andy): I think this should be moved earlier, after which the analysis is added. Maybe we should add some stuff on the "quality gains" and add a plot for that, but there's not much we can probably do until later.

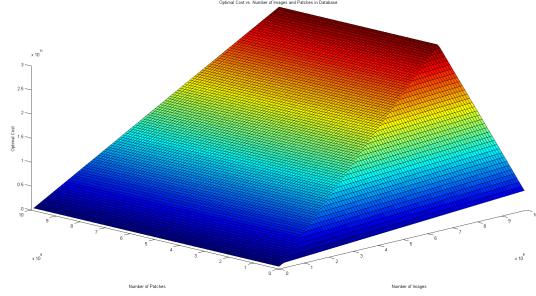
At larger patch granularities, each patch contains more image structure, and thus the probability that another patch contains the same or similar image content decreases with the number of pixels in a patch. At larger patch granularities it becomes increasingly harder to find matching patches in the patch dictionary, and the closest matching patches for textured regions might introduce artifacts (see fig.2). At the same time, patches that are too small do not offer as efficient a compression strategy. We must balance the costs of storing pointers to patches for each image in our database, as well as all the patches themselves, against the costs of storing the images in their original form. This calculation is investigated further below.

### 4.1.1 Cost Evaluation

Assume for now that we choose to store  $p$  patches in our auxiliary table. In practice, we choose  $p$  to be a function  $p: S \rightarrow \mathbb{N}$  which maps from our similarity metric to a number of patches to store. Assume also that each patch is square and composed of  $n^2$  pixels, where  $n$  is a user defined parameter. We further assume that each pixel requires 3 bytes to store and that each pointer is 8 bytes (a standard integer for a 64-bit system). Under this "image only" scheme, in the case where we have  $i$  images, the cost  $c_i$  to store all the images in our database is:

$$c_i(i, m) = 3im^2 \quad (1)$$

In the case where we store pointers to patches, we have two tables: one table to store pointers to image patch exemplars, and a second table to store the exemplar data themselves. Under this "patch pointer" scheme, in the case where we have  $i$  images and  $p$  patches, the cost  $c_p$  to store all the images in our database is:



**Figure 3:** A graph demonstrating how  $c_o$  changes with  $i$  and  $p$  for  $m = 100$  and  $n = 10$ . Note the line of discontinuity where  $p = 357.3i$  - this is the line where the costs of  $c_p$  and  $c_i$  intersect.

$$c_p(i, p, m, n) = 8i\left(\frac{m}{n}\right)^2 + 3pn^2 \quad (2)$$

The first term is the cost of storing the pointer data, while the second term is the cost of storing the patch exemplars themselves. The "8" comes from the fact that we are dealing with very large image and patch tables ( $> 2 \times 10^9$  patches), and thus a `bignum` type is required to store the patch references.

Given these two equations, for a fixed  $m$  and  $n$ , we can easily see that our compressive scheme becomes more space-efficient when:

$$p < \frac{m^2 * (3 * n^2 - 8) * i}{3 * n^4} \quad (3)$$

As long as we choose a similarity threshold such that new image patches get added at a rate that guarantees this inequality is satisfied, our compressive method of image storage will save space. Figure 3 shows an example of how the optimal storage cost changes with different patch and image counts, where the optimal cost is defined as  $c_o = \min(c_p, c_i)$ ; in other words, storing the images using the less expensive method.

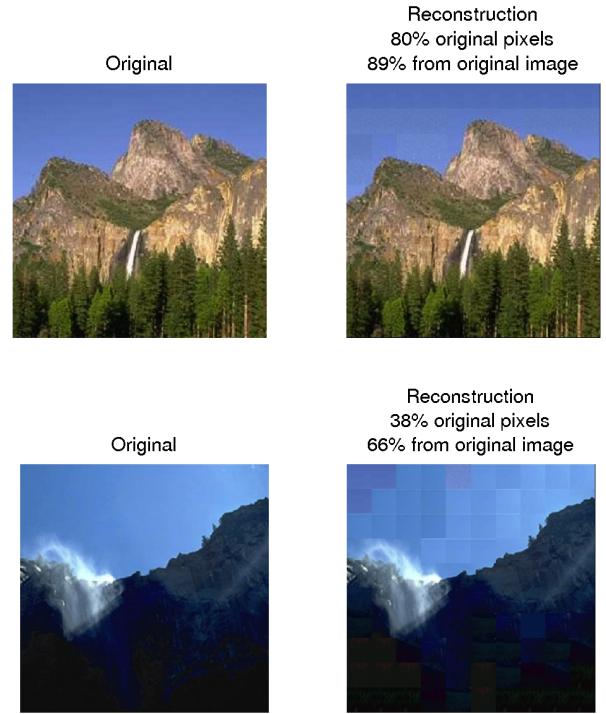
## 4.2 Sampling strategies

TODO: Now that this is mentioned earlier, should this even go here?

A patch dictionary can be built up incrementally, adding new patches as new images are added to the database. A potential problem with this approach is that image reconstruction quality will tend to decrease with the order in which images are added, such that images added to the database earlier will tend to have more patches that correspond to them (see Fig.4 for an example). A strategy with a more even distribution of reconstruction quality over images involves starting with a batch of images, and seeding the dictionary by randomly sampling patches from a set of images from the batch. This is the strategy we employ.

## 4.3 Similarity Function

Many image (more specifically, patch) similarity functions are possible, each with its own distinct set of parameters that can be tweaked for the required application. Because



**Figure 4:** Example of a biased patch dictionary construction strategy, leading to non-uniformity in image reconstruction quality. Images added to the database earlier (top row) are better reconstructed (due to more patch samples in the database) than images added later (bottom row), constrained to be constructed out of patches added initially. The sky pixels in the image added later are borrowed from sky pixels of other images (44% of the pixels in this image come from other images, compared to only 11% in the image on the first row). Note: here we use a very low patch similarity threshold and large patch size for demonstration purposes only, to emphasize the artifacts created.

we are dealing with patches of a size specifically chosen to increase within-patch homogeneity, we do not consider cases of patches containing objects (the most we expect is an object boundary or simple texture), and thus do not need to consider complex image similarity functions (like SIFT matching, spatial relationship-preserving functions, etc.). We can constrain ourselves to color similarity, and split a patch  $P_i$  into 3 color channels:  $P_i(1), P_i(2), P_i(3)$ .

Then we consider two patches  $P_i$  and  $P_j$  similar when, given  $n \times n$  patches, all of the following are true:

$$\begin{aligned} \frac{1}{n^2} \|P_i(1) - P_j(1)\|^2 &< T_1 \\ \frac{1}{n^2} \|P_i(2) - P_j(2)\|^2 &< T_2 \\ \frac{1}{n^2} \|P_i(3) - P_j(3)\|^2 &< T_3 \end{aligned}$$

The  $\frac{1}{n^2}$  term allows us to normalize for patch size, so that the threshold values chosen becomes independent of patch size. Here we constrain the average similarity value of all the pixels to fit a threshold, whereas it is possible to have alternative constraints (whereby the maximal pixel difference, or the variance of the pixel differences, or some other quantity does not exceed a threshold).

Note additionally that if instead, we fix a single threshold for the sum of the Euclidean differences in the 3 color channels:

$$\frac{1}{n^2} [ \|P_i(1) - P_j(1)\|^2 + \|P_i(2) - P_j(2)\|^2 + \|P_i(3) - P_j(3)\|^2 ] < T$$

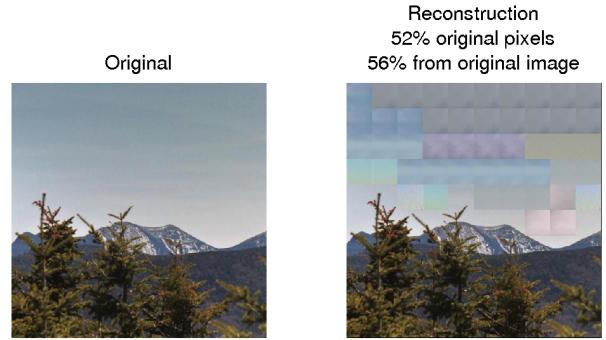
then the similarity in one color channel may compensate for the difference in another, producing skewed results (see fig. 5).

Multiple color channels  $P_i(1), P_i(2), P_i(3)$  are possible, but we choose to work in the (CIE)LUV color space, which is known to be more perceptually uniform than the standard RGB color space [?]. Additionally, our formulation makes it possible to impose separate similarity thresholds on each of the color channels ( $T_1, T_2, T_3$ ). However, for simplicity, we set  $T_1 = T_2 = T_3 = T$ .

#### 4.4 Similarity Threshold

Choosing a threshold  $T$  requires weighing the quantitative benefits of compression with the qualitatively poorer image reconstructions. We ran a number of experiments, varying the threshold, and quantitatively and qualitatively examining the results. In fig. 6 we plot a few small experiments (with 200 images) for demonstrative purposes. The images were  $500 \times 500$  pixels, and the patch size was  $25 \times 25$ . We chose this patch size due to the discussion in 4.1. Below we consider a number of quantitative indicators for patch compression.

In the set of graphs in the first column of fig. 6, we plot in blue the dictionary size against the number of images added to the database. We compare this to the total number of patches that would have been stored if no compression scheme was utilized (red dotted line). We can see that for all choices of threshold, the size of the patch dictionary grows slower than the total number of patches that would need to be added if all images were stored along with their original patches. The gap between the blue and red dotted lines is a measure of the storage savings. As the threshold becomes more stringent, the blue line approaches the red dotted line. Note additionally that as the threshold becomes smaller,



**Figure 5:** This is what happens when we do not separately constrain each of the color channels to match. We have patches that match in terms of general hue (average of the color channels), but are the wrong color and produce visible visual artifacts.

patches are more likely to be reused from the same image than from other images when compressing an image, because only patches from the same image will be similar enough from other patches from this image.

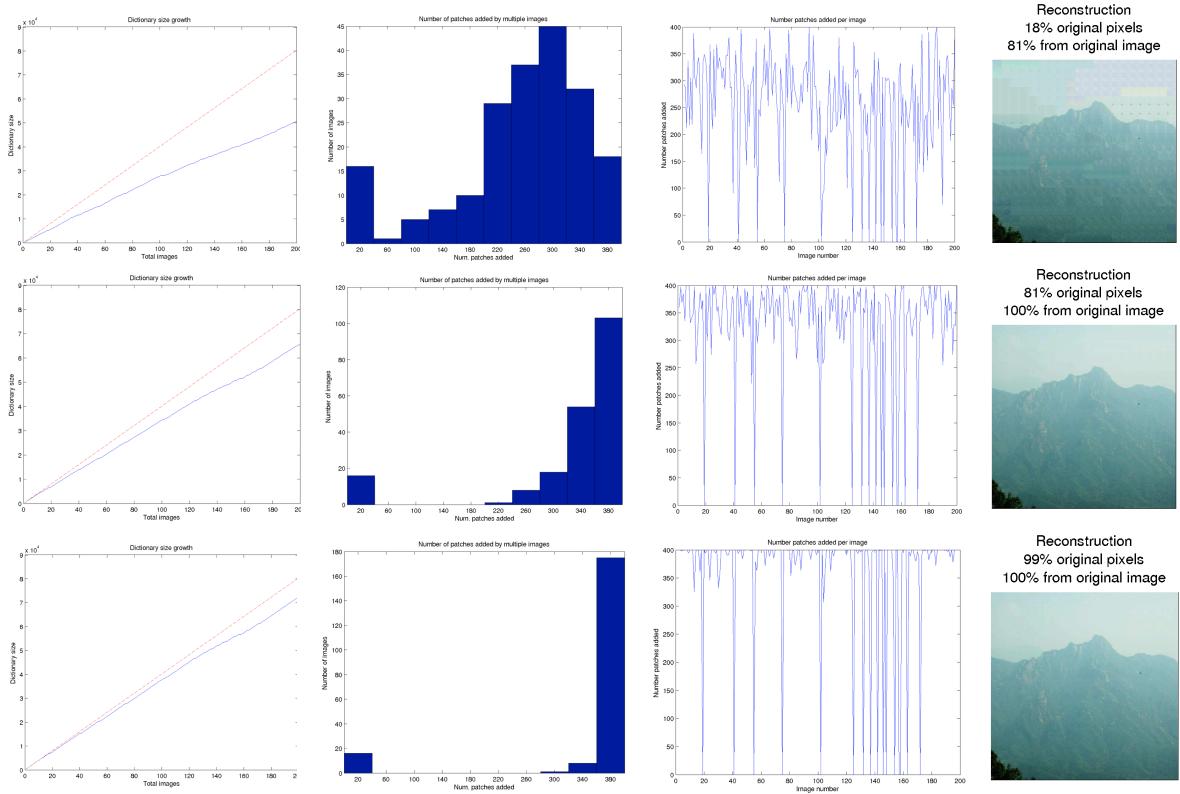
The second column of fig. 6 contains histograms indicating how many images contributed different amounts of new patches to the patch dictionary. When the threshold is very small (as in the histogram in the last row) more of the images contribute most of their patches (380-400 new patches added to the patch dictionary per image). Note additionally that the small number of images that is contributing 0 new patches to the dictionary accounts for the 10% duplicates that are present in the SUN database (more about this in sec. 6).

The third column of fig. 6 contains an insertion history: for each image inserted into the database, we track how many of its patches were added to the patch dictionary. We can see that as the patch similarity threshold decreases, most images contribute most of their pixels. This provides similar information to the histogram (which is merely the cumulative), but allows us to monitor any temporal changes. The spikes to 0 in this graph are indicative of duplicate images, and are discussed in sec. 6.

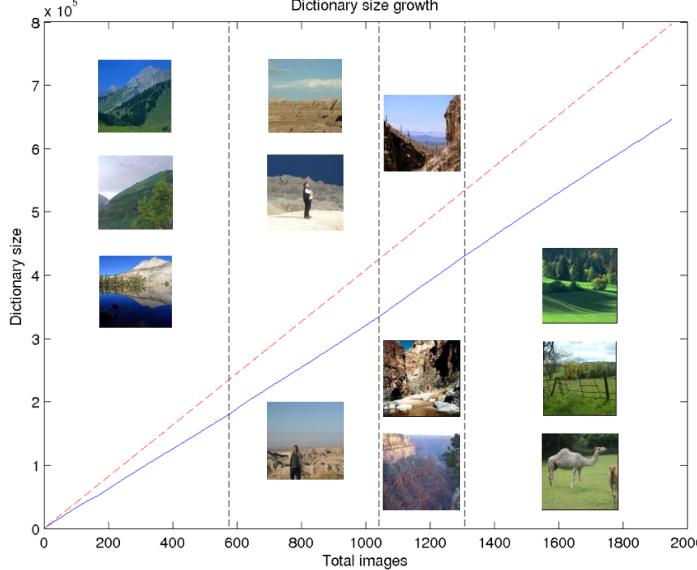
In the final column of fig. 6, we see a sample image reconstruction. We can see that the reconstruction quality increases, and visual artifacts decrease, as the similarity threshold decreases (becomes more stringent). At some point in the middle, the reconstruction is already indistinguishable from the original, but with significant database compression benefits. Thus, for further analysis we consider the threshold  $T = 8$  (recall that this is per color channel, and is independent of patch size).

#### 4.5 Growing the Database

In the preceding sections, we determined that for an image sized  $500 \times 500$ , a  $25 \times 25$  patch size with a  $T = 0.8$  patch similarity threshold is appropriate since compression savings are properly balanced against artifacts introduced during image reconstruction. For further experiments, we scale down the image size to  $100 \times 100$  and the patch size to  $5 \times 5$ , accordingly, maintaining the patch:image ratio. Note that the similarity threshold still applies as it is independent



**Figure 6:** Quantitative and qualitative results obtained by varying the patch similarity threshold,  $T$ , while extracting 400 patches from each image. In the first row,  $T = 80$ , the dictionary size is 50,822 patches, and the average compression per image is 36.73%. In the second row,  $T = 8$ , the dictionary size is 65,794 patches, and the average compression per image is 18.09%. In the third row,  $T = 0.8$ , the dictionary size is 72,445 patches, and the average compression per image is 9.80%.



**Figure 7:** A patch dictionary constructed from  $5 \times 5$  patches from  $100 \times 100$  images, with a  $T = 0.8$  patch similarity threshold, obtains an average compression of 18.95% per image. The average number of new patches added per image is 330.70 with a total dictionary size of 645,534 patches representing 1952 images. The dotted lines mark the addition of new image categories, (left to right) mountain (with dictionary growth rate 312.96, equivalent to compression of 21.76% per image), badlands (growth rate 330.50, compression 17.38%), canyon (growth rate: 360.61, compression 9.85%), and pasture (growth rate: 334.11, 16.47%). A few image samples from the 4 categories are included. Note that for this growth rate, storing image patches will always take less space than storing the entire image, since  $c_p < c_i$ .

of patch size. The reduction in image size allows us to consider larger experiments on databases of image thumbnails.

In fig. 7 we consider the growth of the patch dictionary as we add 2K images to our database. We add images from different categories, and observe a slight change in dictionary growth rate for every new category added. Although there is some variation across categories, the overall compression is 18.95% per image, demonstrating generalization across categories.

## 4.6 Image Type

TODO: Zoya will add experiments on different scene types

Not all image content is amenable to the same type of compression. For some types of images, particularly where there is a lot of spatial structure (e.g. indoors scenes with objects and parts), compression artifacts are much more noticeable and thus similarity thresholds should be more stringent. An interesting extension that is beyond the scope of this paper would be to have a content-aware similarity threshold (self-adjusting to content type such as indoor vs outdoor/natural).

## 5. PERFORMANCE

In this section we evaluate the performance of the implementation of our method on a real collection of X images of Y size.

### 5.1 Quantitative

### 5.2 Qualitative

## 6. APPLICATIONS

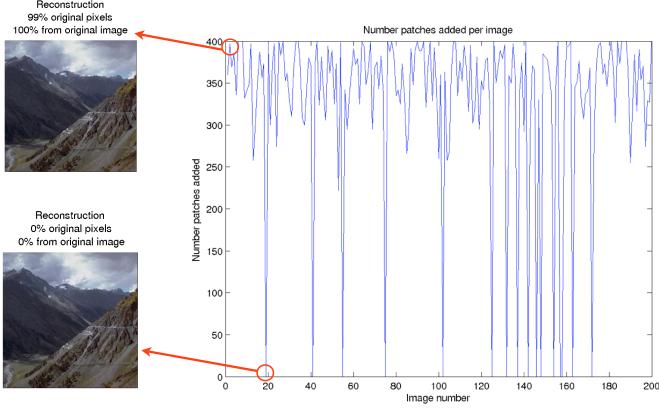
One of the appeals of this approximate patch-based approach is that it naturally lends itself to applications. In this section we describe methods to use our database for two applications - duplicate detection and similar image retrieval.

Encoding images as pointers to a collection of patches provides the ability to quickly spot images that contain large overlapping regions (composed of the same patches). In the extreme case, if multiple images point to the same set of patches, then we know these images are duplicates. Duplicates are a big problem in big computer vision datasets because they occur frequently and are hard to manually remove. They occur frequently (sometimes up to 10% of the time) because these datasets are automatically scraped from the internet, where the same image can occur under separate identifiers (on different websites, copied and uploaded by different users, etc.). The SUN database [?] used in this paper is no exception.

Duplicates are difficult to detect because not all duplicates are pixel-wise identical: the same image encoded using different standards or sized to different dimensions (even when resized to the same dimension later) will look almost identical to the human eye, but will contain different pixel values. Our patch similarity metric is forgiving to perturbation at the pixel-level as long as the patch is overall similar to another patch (see sec.4.3). If multiple images map to the same set of patches that means that the corresponding patches in those images are within a similarity threshold of each other (upper-bounded by  $2T$ ). If multiple images map to all of the same patches, then we have good guarantees that the images are near-duplicates. Otherwise, the probability that every single patch matched would be low (i.e. the images are similar locally, for multiple local locations - as many locations as patches).

We can use these properties to spot duplicates in our dataset on the fly. For instance, when an image is added to the database, we can measure how many new patches the image contributed to the patch dictionary (because similar-enough patches could not be found), and how much of the image was mapped to pre-existing dictionary patches. When an image is reconstructed fully from the dictionary patches, and the patches it is reconstructed from all come from a single other image in the database, we know that the newly-added image is a duplicate. This is depicted in fig. 8.

By the same logic, similar images are those that overlap in terms of the patches they share in common. We can easily check if the patches that match are found consecutively in the images (for instance, when only some local region of the images matches, like when they share an object) or are generally spread throughout (have a similar Gist). We can thus discover images that have different degrees of overlap with other images. Another, somewhat more whimsical application is in the automated creation of photomosaics (TODO: cite). If the patch database is seeded manually with image



**Figure 8:** This is an example of the first 200 consecutive insertion queries to an empty database: for each image inserted, we can measure how many new patches were added to the patch dictionary (out of 400 patches in the image). When we see that this number spikes down to 0 we know that the image has been fully reconstructed from patches from other images. We can check if all those patches came from a single other image. If that is the case, we know we have duplicate or near-duplicate image.

thumbnails, tiles can be replaced with patches close in similarity space. Thus, we map tiles to the closest thumbnails we can find in our patch table. In this case, the patches are typically disjoint from the image database (that is, not found anywhere within any of the images), and certainly not sampled from it. We provide an example of a photomosaic generated using our database application in figure [TODO].

## 7. CONCLUSION

### 7.1 Future Extensions

In this paper we considered a simple implementation of a patch-based image database compression scheme, where the images and patches were square and of fixed sizes. Patches were sampled in a regular, non-overlapping grid from each image. Alternative approaches include more flexible, context-aware, patch-sampling techniques. For instance, the patch granularity for sampling large homogenous sky and field regions may be different from the one used for sampling highly-textured regions like objects and structures (trees, buildings, people, etc.). Similarly, patches that do not cross object boundaries are likely to lead to less artifacts in future reconstructions. For this, approaches like Selective Search [?] that localize image regions likely to contain objects, may prove promising for sampling patches.