USING SHAPE VECTORIZATION INSPIRED BY THE V4 AREA FOR OBJECT RECOGNITION

ANDO EMERENCIA

Abstract. The purpose of this research was to find out how well a shape vectorization inspired by the V4 area would perform in the area of object recognition. To this end, we wrote an implementation in Matlab, and tested it on the MPEG-7 dataset. This gave us good results, but there is room for improvement. It should also be noted that there are many different ways to implement this concept, and we only tested a limited number of possibilities (what seemed most logical). Disadvantages of our current implementation is that it is not indifferent to rotation, or to the textures used on objects; most misclassifications on the dataset are due to these facts. In this paper we will also discuss possible ways to deal with these problems. All in all, we can recognize specific sets of objects reasonably well, while using a transparent and easy to use and manipulate datastructure to represent our program's knowledge (in vectors), as opposed to most other methods of object recognition.

1. Introduction

People are good at visually recognizing objects, which is what inspired our approach. Much research has been done on the visual cortex of macaque monkeys. The visual cortex of man, being a fellow primate, is assumed to work in a similar way. So we use what we know from the visual cortex of macaque monkeys in trying to mimic human object recognition.

There are several visual cortical areas. The V4 area is part of the ventral stream (V1, V2, V4, IT), which is associated with form recognition and object representation [Unge82]. Neurons in the visual cortex fire action potentials when specific (combinations of) visual stimuli appear within their receptive field. Such visual stimuli can be very specific for a neuron – this property is called tuning.

Neurons in earlier visual areas are tuned to simple stimuli. For example, a neuron in V1 may fire in response to vertical lines. In higher visual areas, neurons are tuned to more complex stimuli. For example, in the inferior temporal cortex (IT), a neuron may only fire when a certain face appears in its receptive field.

Area V4 is an intermediate stage. Neurons in V4 are believed to be tuned to moderately complex boundary information (such as simple geometric shapes) at specific locations within larger shapes [Pasu01]. We will use a collection of such simple shapes, which we will call *primitive shapes*, to represent the shape of (parts of) objects.

We will make use of a grid of cells (each cell representing a neuron's receptive field). Researchers have found a linear relation between the distance to the center of gaze and the receptive field size [Gatt88]. This explains for example why human vision is more accurate (perceives more detail) around the center of gaze than near

Date: July 2008–February 2009, Rijksuniversiteit Groningen.

the edges. Inspired by this property, our program will use smaller cells near the center of the grid and larger cells near the edges.

Methods for V4 inspired object recognition already exist (e.g. [Murp07]), but these use a network or layers of weights to store the information learned from the training set. While such an approach may give good recognition results, the usability of a neural network remains limited. This is why our program uses *vectors* to represent images. Because once we have a collection of categorized vectors, we can use all sorts of methods from pattern recognition (e.g. KNN, tree learning, etc.) to classify a new sample. Moreover, we can perform operations on vectors such as calculating the mean vector of a category.

Section 2 lays out some of the assumptions made. We explain how we implemented the design in section 3 (‘Constructing The Image Vector’) and section 4 (‘Recognizing An Object’). In section 5 we present our results and in section 6 we interpret these results and discuss possible future work.

2. Methods

2.1. **What Is Recognized.**

Our program recognizes single objects against a monotone background color. The colors used in the images are not important, as we are using a *color indifferent* approach.

2.2. **Preprocessing.**

Input images are either in black/white (colors 0 and 1), or grayscale images (graylevel ranging from 0 to 255).

The first step in preprocessing the images is to apply histogram stretching. This transformation ensures that the full contrast range is used, allowing for more accurate calculations of mean grayscale values (since they are rounded to integers), which occurs later in the program.

Secondly, input images may be in any shape or size, as they are transformed (centered, scaled and possibly padded) to a preset image size (128×128 pixels) that is used throughout the program.

Each image contains a single object whose center coincides with the center of the image. We defined the center of an object to be the center of the smallest rectangular box (i.e. the tightest fit) that contains the whole object. We have also tried a different centering method, which centered the object around its center of mass, but that method performed worse during testing.

3. Constructing the image vector

3.1. **Fixed Vector Size.** The first step of our program is to compute a vector from an image, based on recognition of primitive shapes. Furthermore, since we might want to perform operations (such as taking the mean of a collection of vectors) on the vectors later, constructed vectors must all be compatible with one another. So they must all be of the same fixed length, and also, the meaning of a certain position in the vectors must be the same, so that it represents the same feature in all vectors. Had we not set these restrictions, then comparing vectors would be meaningless. This means that we cannot construct the vector based on the contents of the image or on the object we are recognizing. We need a representation that depends only on things such as what sort of shape is present at certain positions.

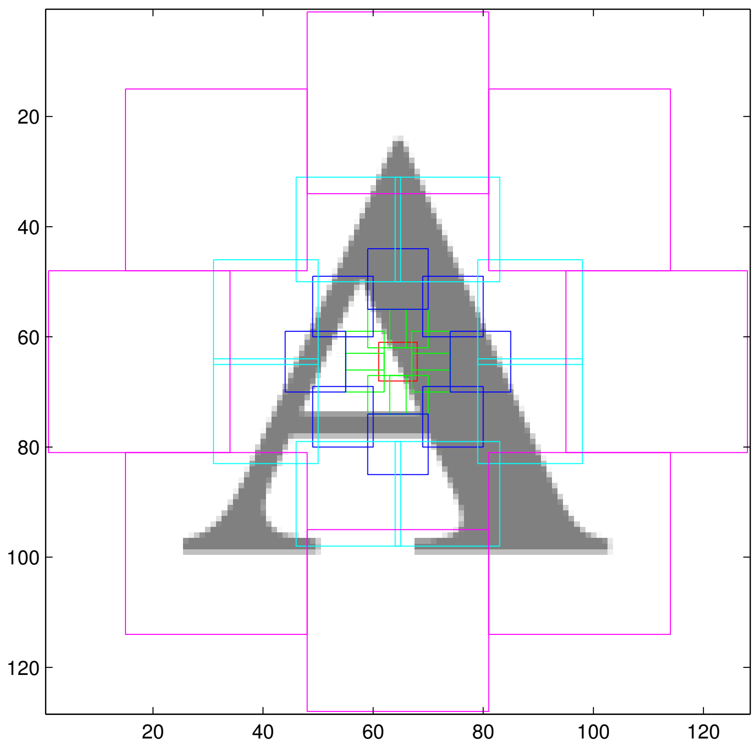

FIGURE 1. The grid used. Shown in the grid is the letter A.

3.2. **The Grid.** To mimic the human visual field, we used a grid, shown in Fig. 1. This grid consists of 33 cells. Most notably, cells near the center are small, while the cells near the edges are large.

From Fig. 1 we can see that we have 5 sets of cells (each in a different color), and all sets but the innermost one consist of 8 cells (the innermost set consists of a single cell), so that gives us $4 \times 8 + 1 = 33$ cells. Parameters used to construct the grid can be found in Table 1.

3.3. **Primitive shapes.** In cortical area V4, different neurons are tuned to different stimuli. Some may be selective for sharp angles in a certain direction, others to straight lines or to a combination of such features. To model this aspect of area V4, we consider a grid cell as the receptive field of 32 neurons, each tuned to a different primitive shape (see the 32 primitive shapes in Fig. 2). So for each grid cell, we compute 32 responses, one for each of the mentioned 32 model neurons. We compute the response of such a model neuron as a certain measure of similarity between the primitive shape assigned to that neuron and the part of the image that is visible in the receptive field of the neuron (the corresponding grid cell).

set nr.	**nr. of cells**	**size**	**midpoint radius**
1	1	8x8	0
2	8	8x8	7
3	8	12x12	15
4	8	20x20	26
5	8	34x34	47

Table 1. Parameters used to construct the grid shown in Fig. 1. The midpoint radius is the distance from the center of the grid to the centers of the cells.

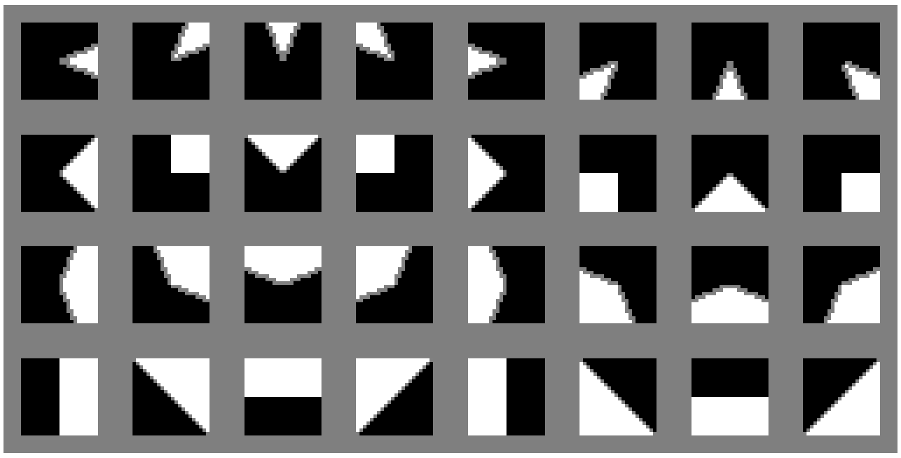

Figure 2. The primitive shapes used. The orientation of the primitive shapes changes in the horizontal direction and their opening angle changes in the vertical direction. Note that the grayscale values of the primitive shapes do not matter, as they are blended in with the image later (this is explained in section 3.5). Because of this fact, our set of primitives contains some doubles, causing straight lines to weigh more, but this does not influence our results in a negative way.

We use the same set of primitive shapes for all cells, but for every cell, we scale them according to the cell size.

3.4. **Computing the similarity.**

To compute the similarity of a given primitive shape with the part of an image that falls in a given cell of size $n \times n$, we first rescale the primitive shape to size $n' \times n'$, where $n' = \lceil 0.62n \rceil$ (the number 0.62 used here was chosen to be small enough to be robust against local noise in the image, and large enough to match only features that take up a significant part of the cell). Next we use the primitive shape as a template and use it for template matching in the concerned image cell, i.e., we put the template in a given position so that it fits in the image cell and compute the dissimilarity between the primitive shape template and the overlapping part of the image cell (Fig. 3).

We compute this dissimilarity using the L_1 norm (sum of pixel-wise absolute values of differences) and normalize it by dividing it by $255n'^2$. For each possible

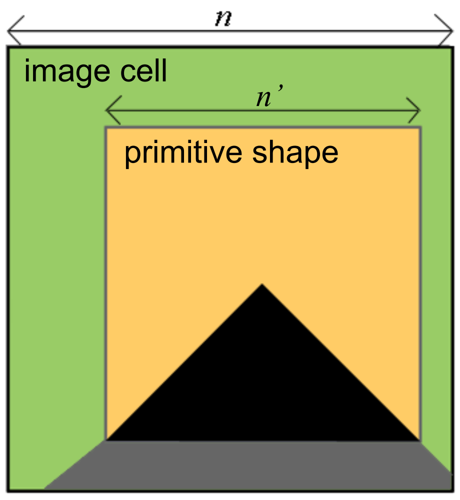

FIGURE 3. We scale the primitive shape in such a way, that inside the designated cell on the image, it has a few pixels on either side to move around in. So there are different positions within the cell where the primitive shape can be.

position of the primitive shape template within the image cell, we may get a different value of the dissimilarity. We take the minimum of all these values, subtract it from 1 and call the result the similarity between the concerned primitive shape and image cell.

3.5. **Coloring.** When calculating the similarity of part of an image with a primitive shape, as explained in the previous section, we first blend the colors of the primitive shape in with the image.

We will now explain this coloring process, using Fig. 4. In *a* we see one of the primitive shapes. In *b* we mark the boundaries of the color zones in the primitive shape. In *c* we show these color zone markings, overlaying part of an image cell. These zone markings split the image part up in three segments. The image of *d* is constructed by replacing the pixel values in each segment from *c*, by their average pixel value taken per segment. For example, we can see in *c* that most of the pixels in the red-outlined segment are black, so in *d*, this area has a dark color; in contrast, most pixels in the blue-outlined segment in *c* are white, hence the lighter tone of the same area in *d*.

The image of *d* is the colored primitive shape, i.e. the actual primitive shape that will be compared with the image part (as shown in *c*, but without the color markings), as described in the previous section. Because of this coloring step, our approach is color indifferent (the color segments of the primitive shape take the color of the underlying image part), we get higher similarity scores on average (because the colors used in the primitive shape are taken as the means of sections of the image part), and we have that the better the different color zones of the primitive shape segment the image part, the higher the similarity score will be.

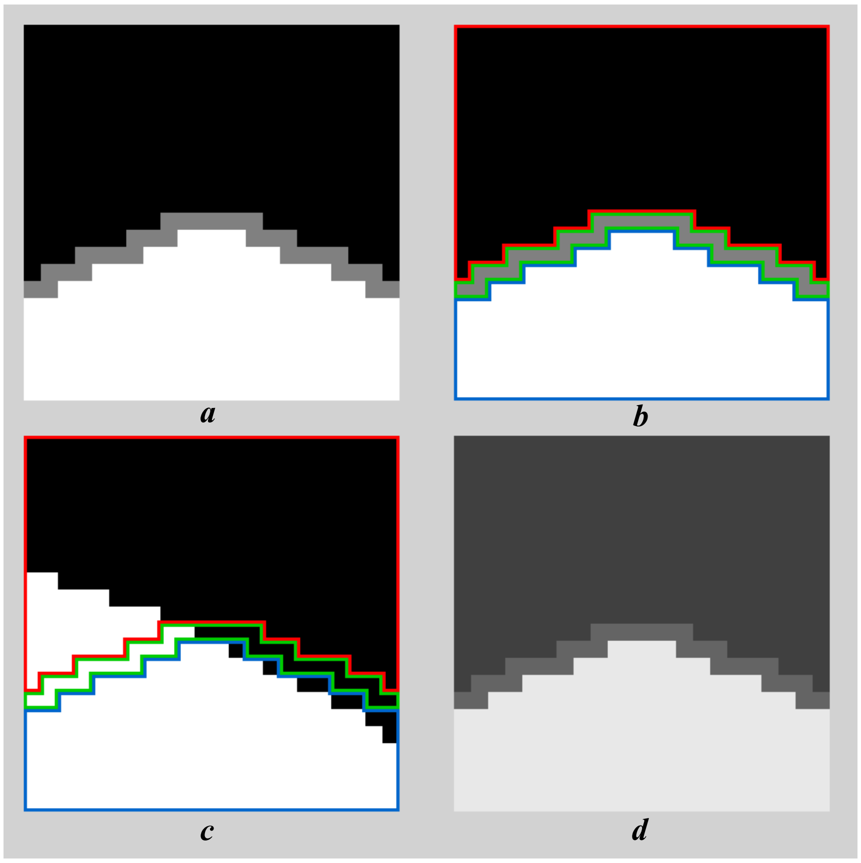

FIGURE 4. *a*: the primitive shape. *b*: marking the boundaries of the different colors used in the primitive shape. *c*: the segment-boundaries of the primitive shape shown overlaying part of an image. *d*: the segments colored according to the mean color value of the segments in *c*.

Note that if the primitive shape only uses two colors (i.e. it does not use gray), then two segments are identified and colored instead of three. This does not change the algorithm or its outcome.

3.6. **Detecting significant grayscale changes.** An image cell that is fully white would give a high similarity score for all primitive shapes, because all parts of a primitive shape would simply be colored white and match perfectly. However, this would also mean that all primitive shapes match equally well, and thus this gives us no information about what the image represents. As a solution we have decided to set to zero the similarity scores for all features of monotone (and monotone-alike) cells.

This leaves us with the problem of how to detect such image cells (image cells with no significant grayscale changes). We decided to test this by standard deviation. If the standard deviation is below a certain threshold, the colors of the image cell are too similar, and so we set the similarity scores of all features for this cell to zero. Experimental testing has led us to use a minimum standard deviation of 70 over the image cell (or: 27% of 255).

Aside from the standard deviation testing, we also set the similarity scores of an image cell to zero if more than 87% of its pixels are of the same grayscale value. We derived this value by looking at the primitive shapes and seeing what the maximum percentage a single color could take up in an image cell would be, while still being able to match a primitive shape.

3.7. **Examples.** In this section we show a few examples that illustrate how the similarity scores are distributed over our 32 primitive shapes, given a partial image visible in a cell.

Below we show three figures, each is structured in the same way. On the left is the part of an image visible in a certain cell shown. For convenience, its inverse is shown directly below it. On the right are the 32 primitive shapes. The primitive shapes and the image cell are drawn to scale, and so it is important to realize that different primitives may match different parts of the image cell. Below the primitive shapes are the associated similarity scores shown. These are marked with an asterisk (*) if they are above a certain threshold for binary conversion (see the next section).

The contrast of the primitive shapes is rendered according to their similarity with the concerned partial image. This contrast scaling was normalized for better viewing. The fact that all similarity scores lie relatively close together is due to the coloring of the primitive shapes (see section 3.5).

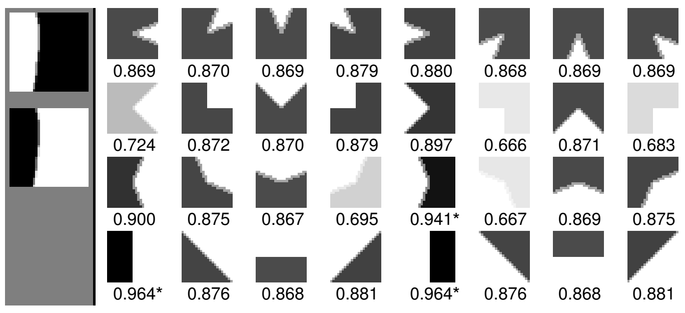

FIGURE 5. The image cell on the left is taken from the `horseshoe-20` image. The primitive shapes with the vertical line segments are easily the best matches. The primitive shape with the vertical line curved slightly to the left scores above the threshold as well. This is in accordance with what we would expect from this image segment.

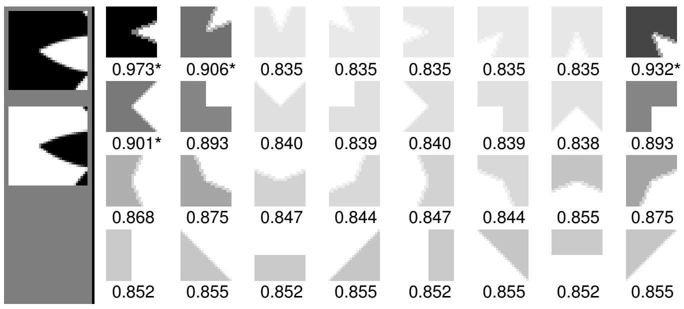

Figure 6. The image cell on the left is taken from the `device7-18` image. The image cell has a specific convex indentation to the right, allowing only primitives with an opening to the right to have high similarity scores. The best match is from the primitive shape with the sharp indentation to the right, as we would expect. The other primitive shapes that score above the threshold are the ones with the angle to the upper and lower right, and the primitive shape with a slightly wider angle to the right.

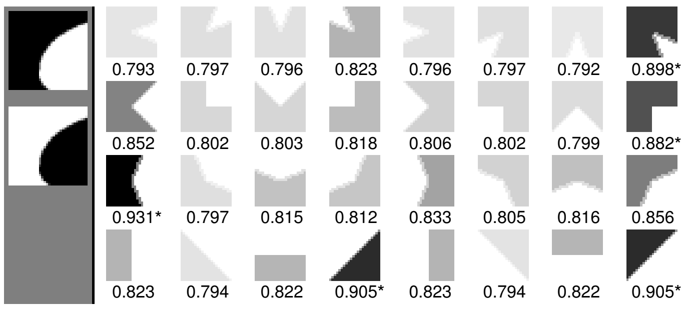

FIGURE 7. The image cell on the left is taken from the `chopper-16` image. This figure demonstrates that by retaining multiple similarity scores per image cell (i.e. similarity scores from different primitive shapes), it may happen that these correspond to different positions. So our description of an image cell may contain information about different parts of the image cell. This allows for more comprehensive matching. For this image, we can see how some primitive shapes match the curved vertical part of the object's boundary, while others correspond to the more straight segment sloping up towards the upper right. Although not strictly matching the object's boundary, the high similarity scores from the primitive shapes in the top two rows of the rightmost column tell us that it is likely that there is an area in the image cell where the upper left, upper right and lower left corners are of the same color; this is true for the upper left area of the image cell.

3.8. **Converting Vectors To A Binary Format.** To be able to compare these image vectors using methods described later (in section 4), we first need to convert them to a binary format (i.e. devise a threshold to filter out the few best scores, set those to 1, and the rest to 0).

Testing has led us to believe that the similarity scores for an image cell are distributed normally. Based on this assumption, we employed a threshold that lets an average of 5 out of the 32 similarity scores per image cell pass.

So $\frac{5}{32} = 0.15625$ is about 16%, and we need the area under the normal curve that is $1-0.15625 = 84.375\%$. From statistics we know that $0.84375 \approx \int_{-\infty}^{1.01} \frac{1}{\sqrt{2\pi}} e^{\frac{-z^2}{2}} dz \approx P(-\infty < Z \leq 1.01)$. So we used $1.01\times$ std + mean as threshold.

Assuming an average of 5 significant matches per image cell gave us the best results. A higher number gives image vectors that are too general; a lower number leads to more information loss and image vectors that are sparse, do not match others and are sensitive to small changes in the image.

Note that our threshold rule is better than employing a static threshold rule such as "only take the best 3 similarity scores", since not all of these are necessarily significant. An example would be an image cell where the top three similarity scores are e.g. 0.958, 0.943 and 0.275.

Another static threshold rule such as "only take the similarity scores that are within x distance of the highest similarity score", would also perform less in practice. The distribution of the similarity scores differs a lot per image cell and is based on its contents. Some may have all scores between, 0.55 and 0.57 for example, while others have scores that are widely spread between 0.2 and 0.9.

Only by taking into account the entire distribution of similarity scores for an image cell can we say something about which scores are statistically significant; e.g. those greater than the mean+std. This is exactly what our threshold rule does.

Since we have 33 cells on our grid, and 32 primitive shapes, we get $33 \times 32 = 1056$ comparison results for the entire grid, and this is indeed the size of the image vectors we use in our program. So the value of an element of an image vector is an indication of how well a certain primitive shape matches part of the image in a certain cell of the grid. The image vectors are then converted to a binary format, so that they only contain zeroes and ones. A zero meaning "this cell does not match this primitive shape", and a one meaning "this cell matches this primitive shape".

4. Recognizing an object

Now that we have our vectors, we come to the second part of our program, which is recognizing an object.

4.1. **The Procedure.** In our tests we considered a number of different objects. For each object, we used a number of images showing this object. Every image, along with the name of the object shown on the image, was then presented to our program. Then an unseen image was presented to the program, and the program had to determine which object was shown on the image.

There are a number of ways to do this. We chose to solve it by vector comparisons. We first construct the image vectors of all given images, then we match a new image vector against all the known image vectors. We then choose the object of the known image vector that was the nearest neighbor match (had the highest similarity score with the new image vector).

We have considered other ways to determine the object of the untagged image, other than just taking the nearest neighbor match. For example, we have considered choosing the class that had the highest *average* similarity score with the new image (the average taken over all images showing the same object). This approach gave worse results however. We think that this is because of the fact that while the images for some objects show great distinction, for each image there is usually at least one other image showing the same object that will give a pretty good match.

To calculate the similarity scores between the individual image vectors, we employed three different methods: the inverse Hamming distance, the Recall fraction and the Tanimoto coefficient. These methods are explained in the following three sections.

4.2. **Inverse Hamming Distance.** Since we have binary image vectors, a way to tell how much these vectors differ is given by the Hamming distance. In short, the Hamming distance is the number of bits that need to be flipped in order to turn one of the vectors into the other. The Hamming distance can easily be calculated by taking the element-wise `XOR` of both image vectors.

Computing the Hamming distance however, would give a measure of dissimilarity, not similarity; i.e. a high Hamming distance means the vectors are very different, *not* that they are very similar. This is why we use the inverse Hamming distance:

$$\text{inverse Hamming dist.} = \frac{\sum_{i=1}^{n} \texttt{NOT(XOR(}A_i, B_i\texttt{))}}{n}$$

where n is the image vector size, and A and B are the image vectors compared.

4.3. **The Recall Fraction.** The second method we use to determine similarity scores is the Recall fraction. We define the Recall fraction as the fraction of 1-bits of a test vector A that are also found in a stored vector B; formally:

$$\mathrm{R}(A, B) = \frac{A \cdot B}{|A|}$$

Unlike the other two methods, this method is asymmetric, meaning that R(A,B) is not the same as R(B,A).

4.4. **The Tanimoto Coefficient.** The Tanimoto coefficient is a very intuitive choice for a similarity score, since our vectors will be sparse (mostly zeros), and this method gives the matched features as a percentage of the total number of features detected (detected features are ones) in both vectors.

As we can see from the following formula, the Tanimoto coefficient has range [0,1]:

$$\mathrm{T}(A,B) = \frac{A \cdot B}{|A| + |B| - A \cdot B}$$

5. Results

From the MPEG-7 dataset [Late00], we used images from fourteen categories. See Fig. 8 to get an idea of what the images look like.

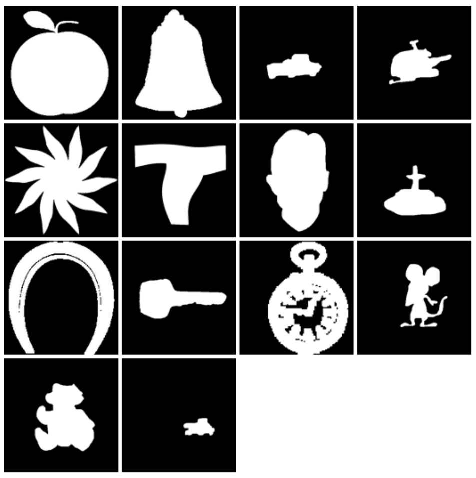

FIGURE 8. Sample images from the fourteen categories that we tested, prior to performing the centering and scaling step in the preprocessing. From left to right (and then top to bottom): `apple`, `bell`, `car`, `chopper`, `device7`, `device8`, `face`, `fountain`, `horseshoe`, `key`, `pocket`, `rat`, `teddy` and `truck`.

We found that all used samples matched at least one primitive shape somewhere. This is to be expected, as long as the images contain an object. The category that matched the least primitive shapes (number of ones in the image vector) on average was `apple`, averaging 34.7 matches per image. The image that matched the least primitive shapes was `apple-6`, with 29 matches. The `apple` category is one of the categories of which the objects have no details around the center, so this result is not very surprising.

We do want to mention here that without the centering and scaling step during the preprocessing, these numbers would be a lot lower (14.7 average from the `truck` category, with only 6 matches from `truck-17`). This is because without scaling, small objects cover fewer image cells.

Fig. 9 shows how the image vector encodes the color boundaries of an object.

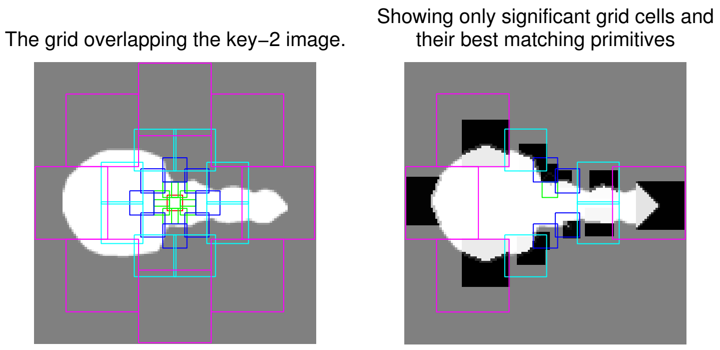
The grid overlapping the key−2 image.
Showing only significant grid cells and their best matching primitives

FIGURE 9. Illustrating the image vector of the `key-2` image. Left we see the cells of the grid overlaying the image. Right we see only the significant cells from the grid (those that contain part of the object's boundary). In each cell, the best matching primitive shape is shown (at its best matching position). Note that the actual image vector contains more information than is shown here, since an image cell may match more than one primitive shape.

We tested our program using leave-one-out cross-validation of nearest neighbor classification, with 20 samples from each category. This means that we take one image vector, and compare it to all $14 \times 20 - 1$ other image vectors. We look at the nearest neighbor (the image vector with the highest similarity score), and determine if it was from an image that belongs to the same category, or to another category. So the classification error is the fraction of image vectors for which the nearest neighbor is from a different category.

We performed these tests using the three comparison methods described in section 4. The results are presented in the following three sections.

5.1. **Inverse Hamming Distance.** Table 2 shows the classification results for this method. The image of Fig. 10 shows nearest neighbor similarity scores.

category	**classification error [%]**
apple	0
bell	5
car	0
chopper	0
device7	0
device8	0
face	0
fountain	0
horseshoe	25
key	5
pocket	25
rat	0
teddy	0
truck	0
total	**4.29**

TABLE 2. Results from the leave-one-out cross-validation test using the inverse Hamming distance, listed per category.

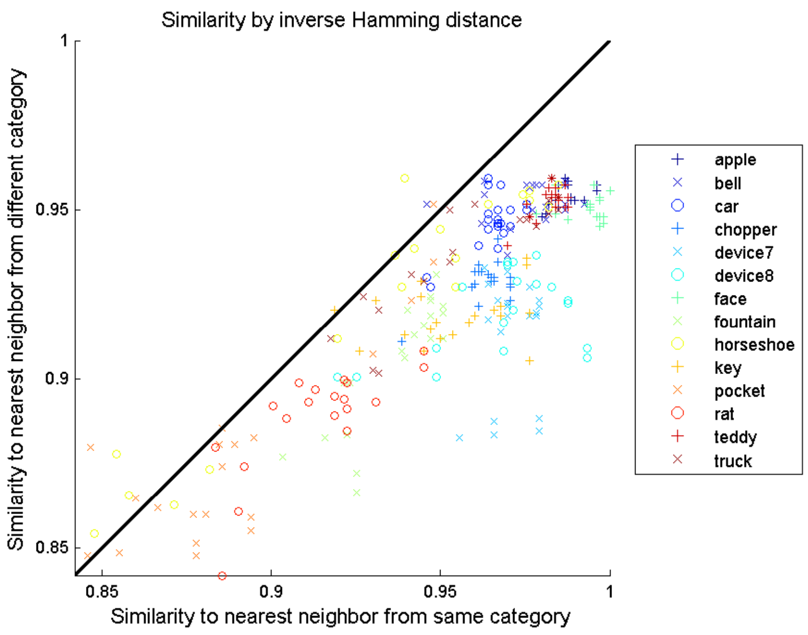

FIGURE 10. A 2D scatter plot showing nearest neighbor similarity scores of the 14 × 20 images used in the tests. The x (horizontal) and y dimensions represent the similarity to the nearest neighbor from the same category and a different category, respectively. The similarity scores shown here were computed using the inverse Hamming distance method.

5.2. **The Recall Fraction.** Table 3 shows the classification results for this method. The image of Fig. 11 shows nearest neighbor similarity scores.

category	**classification error [%]**
apple	0
bell	5
car	0
chopper	5
device7	0
device8	0
face	0
fountain	0
horseshoe	10
key	0
pocket	10
rat	0
teddy	0
truck	10
total	**2.86**

TABLE 3. Results from the leave-one-out cross-validation test using the Recall fraction, listed per category.

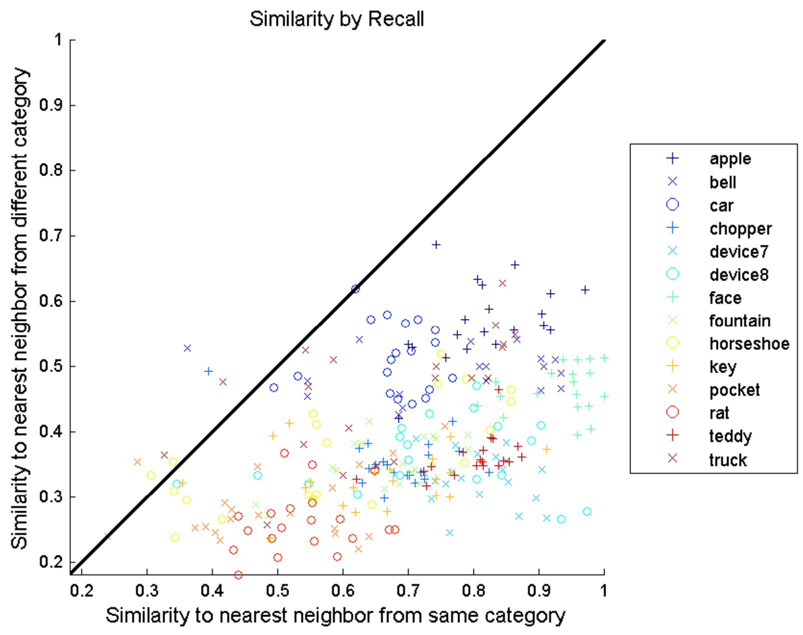

FIGURE 11. A 2D scatter plot showing nearest neighbor similarity scores of the 14 × 20 images used in the tests. The x (horizontal) and y dimensions represent the similarity to the nearest neighbor from the same category and a different category, respectively. The similarity scores shown here were computed using the Recall fraction method.

5.3. **The Tanimoto Coefficient.** Table 4 shows the classification results for this method. The image of Fig. 12 shows nearest neighbor similarity scores.

category	**classification error [%]**
apple	0
bell	5
car	0
chopper	0
device7	0
device8	0
face	0
fountain	0
horseshoe	5
key	0
pocket	10
rat	0
teddy	0
truck	0
total	**1.43**

Table 4. Results from the leave-one-out cross-validation test using the Tanimoto coefficient, listed per category.

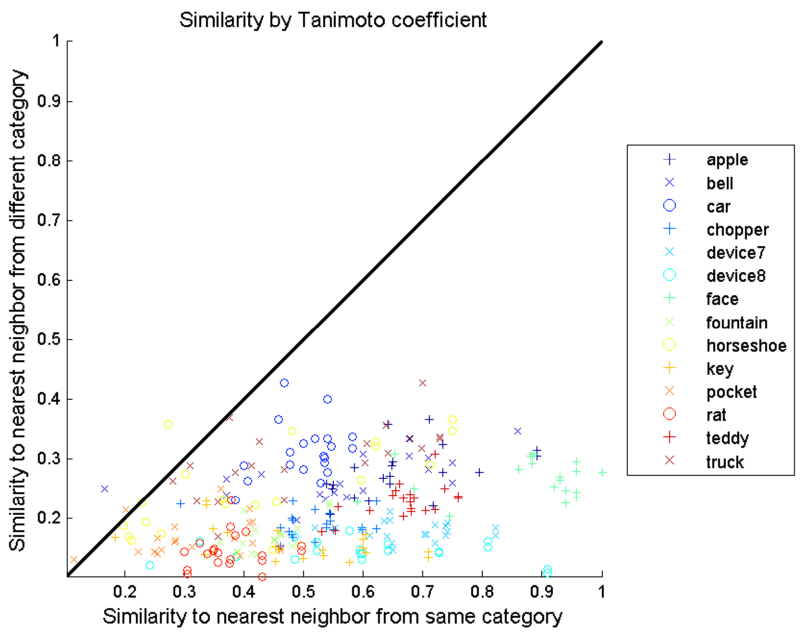

Figure 12. A 2D scatter plot showing nearest neighbor similarity scores of the 14 × 20 images used in the tests. The x (horizontal) and y dimensions represent the similarity to the nearest neighbor from the same category and a different category, respectively. The similarity scores shown here were computed using the Tanimoto coefficient.

5.4. **The errors.**

In this section, we take a closer look at the misclassifications that occurred using the best performing similarity score method, which is the Tanimoto coefficient.

The one misclassification in the `bell` category was due to rotation of the object in its image (i.e. a `bell` that was tilted 45 degrees), such that there was no similar sample in the dataset. It should be noted that other slight transformations (such as apples with a bite taken out of them), were classified correctly. Because of the centering and scaling step, the image of the very small `bell` (with respect to its background) was also classified correctly.

The categories `apple`, `car`, `chopper`, `device7`, `device8`, `face`, `fountain`, `key`, `rat`, `teddy` and `truck` all performed perfectly, which is impressive, since, for instance `device7` and `device8` show a large variety in shape, and `face` has no details near the center.

The one misclassification of the `horseshoe` category was one of the four `horseshoe` images that were upside down compared to the others. This means that it only had three other images of its category with whom it could find a proper match. Two of these were textured, decreasing the possibility of a correct match. Furthermore, an image of a `horseshoe` has no details near its center (where the majority of the image cells are located), so it is more prone to misclassifications. In this case, the round outer boundary of the `horseshoe` matched the shape of an `apple` (also a category with no details near the center).

The `pocket` category was the worst performing category in our tests. We attribute this to the fact that different pocketwatches have different textures near the center. So for instance a difference in time shown, shape of the hands, or backplate details could lead to a different image vector. One of the misclassifications was an image of a pocketwatch that was flipped on its side, which made it very distinct from all other `pocket` images.

6. Discussion

In general, most of the categories performed reasonably well. So for general, restricted object recognition, the method works. With 99% correctly classified, the Tanimoto coefficient proved to be the best method for computing similarity scores. For this method, eleven out of the fourteen categories tested had a perfect score. The worst performing category still had 90% correct.

The images that were misclassified often had multiple 'bad' properties:

- Our implementation is not indifferent to rotation (in 2D nor in 3D). Any object that is rotated compared to the other objects in its category, is very likely to be misclassified.
- Some categories are objects that have no details near their center. Since that is where the bulk of the image cells are located, these categories are more prone to misclassifications.
- The categories for which (some) images were not silhouettes (`horseshoe` and `pocket`), i.e. where background color was used inside the object to denote certain distinctive features on its surface, performed relatively poor. The reason is that these textures do not describe the general shape of an object (which is assumed to be invariant within a category), but properties of its surface (which may very well differ from object to object).

If we look at Fig. 12, it is interesting to see that for the Tanimoto coefficient method, there are no intercategory similarity scores above a certain threshold; i.e. there are no intercategory similarity scores above 0.45, while the majority of the intracategory similarity scores lies above 0.45. The other two methods do not share this property. A high Tanimoto score almost certainly means that the compared images are from the same category; this is a very good property to have.

The Tanimoto coefficient method performing well is within expectations. It is the only method that takes into account the number of ones in both image vectors compared. The Recall fraction method for example, is almost as good, but it may match an object with no details near its center to an object that does have details near its center, if the latter object also contained features of the former object in some way. This is the cause of some of the errors for this method, and makes it less suited than the Tanimoto coefficient. For example, for a certain `chopper` image, the closest match according to the Recall method was the image of a `fountain`, while the `chopper` only matched the bottom half of the `fountain`.

The inverse Hamming distance method performed predictably poor, as it does not take into account the number of ones in the image vectors at all.

6.1. **Centering and Scaling.**

As said before, we first wrote our implementation without the centering and scaling step during the preprocessing. We will now discuss the differences we see in the results.

Without the centering and scaling step, the number of errors was significantly higher for all methods but the Recall fraction (which had 8 errors without the centering and scaling step, up from 6). The inverse Hamming distance method had 26 errors (up from 12), and the Tanimoto coefficient method had 10 (up from 4). The reduction in the number of errors for these two methods is due to the fact that images with objects that would stand out because of their scale or position, are now more likely to be correctly classified. The increase in errors for the Recall fraction is explained later in this section.

The centering and scaling step also had an effect on the scatter plots for all methods: it had a moderating effect on the very bad cases (intercategory-intracategory > 0.3). This difference was significantly decreased for the worst performing (misclassified) images. We think that this is because without the centering and scaling step, relatively small objects would have only a few significant image cells, so a miss on one image cell would have a much greater impact on the total similarity score.

There is however another effect that this step has. It makes certain categories look more alike (e.g. `car` and `truck`, `chopper` and `fountain`). This makes some intercategory scores higher than they would be without centering and scaling (e.g. for the Tanimoto coefficient method, the maximum intercategory score went from 0.35 to 0.45). The Recall fraction method is especially susceptible for this, as it can match an object to parts of another object. This is why we saw an increase in the number of errors for the Recall fraction method when using the centering and scaling step. This step has not cause caused any additional errors for the Tanimoto coefficient method however.

Even if the centering and scaling step had introduced additional errors for the Tanimoto coefficient method, we still think that it is the proper way to compare images, since for example for people, the position of an object in an image does not give any clue to what kind of object it is, so our program should not use this information either. Without centering and scaling however, the program *would* use this information. If an error was introduced because two objects look more alike after being centered and scaled, then that error could just as well have been made by a human.

6.2. **Future Work.**

We have assumed that all images have a monotone background. To allow for this assumption to be dropped, we need to be able to do two things. Firstly, for the squaring step, we need to be able to stretch the background of images that have a non-monotone background. Although it is not the most trivial approach, we think that this problem is best solved by using seam carving techniques [Avid07]. Secondly, for the centering step, we need to be able to identify the edges of an object against a background that is e.g. a smooth gradient transition. We think that this can be done by using edge-finding algorithms with the right threshold.

Another suggestion for future work would be to make our implementation rotation indifferent. This would solve the problem of recognizing the `horseshoe` upside down. It is also a feature that is relatively easy to implement. Since we make use of a circular grid, we can approximate rotation indifference by checking for matches after rotating the grid. Vectors for the rotated grid could even be derived from the original vectors without having to rescan a rotated image. Since rotated versions of all primitive shapes are included in the set of primitive shapes, it would simply be a matter of switching vector cell contents around.

We noted that images that are not silhouettes are not always properly recognized by our implementation. This is because in these images, the fore- and background colors are not only used to denote the difference between the object and its background, but also to denote textures and features on the object's surface; and these may be different for different objects of the same category. If we may assume that the texture on an object's surface is not relevant for detecting its shape (where we only care about edges between object and background), we may have an easy solution. In our dataset for example, we could use some sort of filling algorithm, to make every image a silhouette: fill all regions of background color with foreground color, as long as they are not connected to the outer layer of background color. We think that this operation would significantly improve the results we have had on these categories. Although on the other hand, it might cause donuts to match apples. The best solution would be a precondition on the images that all of them have to be silhouettes.

Support for more advanced deformations (such as a pocketwatch that was flipped on its side) would be harder to implement, since our implementation does not use an internal 3D impression of the objects.

6.3. **Conclusion.**

In conclusion, as said before, the advantage of our approach over others is the ability to directly manipulate the image vectors. Our approach proved to be robust against small variations and deformations in images, as well as to translations and scaling. Our approach assumes a monotone background color. Poor results are due to certain 'bad' properties mentioned earlier in this section. The best results are found for images that are silhouettes and have some significant features near the center of the object.

Acknowledgments

I would like to thank T. van Laarhoven for peer reviewing this paper, and N. Petkov and G. Papari for comments and suggestions.

References

[Avid07] Avidan, S and A Shamir. 2007. Seam Carving for Content-Aware Image Resizing. *ACM Transactions on Graphics* 26(3):10.

[Gatt88] Gattass, R, APB Sousa and CG Gross. 1988. Visuotopic organization and extent of V3 and V4 of the macaque. *J Neuroscience* 8(6):1831–1845.

[Late00] Latecki, LJ, R Lakaemper and T Eckhardt. 2000. Shape Descriptors for Non-rigid Shapes with a Single Closed Contour. *Proc. of IEEE Conf. on Computer Vision and Pattern Recognition* 424–429.

[Murp07] Murphy, TM and LH Finkel. 2007. Shape representation by a network of V4-like cells. *Neural Networks* 20(8):851–867.

[Pasu99] Pasupathy, A and CE Connor. 1999. Responses to Contour Features in Macaque Area V4. *J Neurophysiol* 82(5):2490–2502.

[Pasu01] Pasupathy, A and CE Connor. 2001. Shape representation in Area V4: Position-Specific Tuning for Boundary Conformation. *J Neurophysiol* 86(5):2505–2519.

[Unge82] Ungerleider, LG, M Mishkin, DJ Ingle, MA Goodale and RJW Mansfield. 1982. Analysis of Visual Behavior. *Two Cortical Visual Systems* 549–586.