TEXT RECOGNITION USING THE DISCRETE COSINE TRANSFORM: A MACHINE LEARNING APPROACH

CHARLES PARK AND JAMIE MORROW

1. Abstract

In order to efficiently classify images of text as a particular letter, we built an algorithm to classify alphabetic characters based on their Discrete Cosine Transform (DCT). After building a training data set, we inputted 256x256 pixel images of e, o, t, w, m, x, and s and compared their 16x16 lowest DCT frequencies to our data set. We then found the lowest element-wise magnitude of distance between our input and the training set, classifying the input as the letter with the minimum distance and adding it to the comparison database. We achieved success with six of our seven cases, o, t, w, m, x, and s; e was incorrectly classified.

2. Motivation

Libraries and archives are constantly at odds between their two main priorities: preservation and dissemination of valuable documents. Preserving documents means controlling access and environment, whereas showing the information to the public inherently puts rare pieces in the position to be damaged. Digital distribution, the scanning and releasing of digital images of documents, is an elegant solution. It keeps rare and fragile documents safely in the controlled environment of archives, while giving access to the same or an arguably larger audience. The problem is that the raw scans of documents do not encode any of the documents' text that can be indexed or searched, the lifeblood of digital distribution. Getting this information from scans requires one of two things to happen. Either someone has to manually go through the menial task of transcribing the text, or we have to train computers to recognize and decode text.

Libraries are only one of many incredibly salient examples. Law enforcement uses character recognition to catch speeders with speed cameras. Character recognition even has the possibility to improve the mobility of the blind and visually impaired by translating found text in their daily lives into more accessible media. The crux of the issue, however, is that in this day and age, it is much quicker and easier to take a picture of text instead of transcribing it. Text recognition algorithms look to decode the "hidden" text in images.

The main problem with the proliferation of images is that no image codec will find the text in an image. It is quite obvious to most human observers that text is present in an image. Language barriers withheld, it should be easy enough for a user to comprehend. Computers fail at this because they only natively "see" the values of a particular pixel, and do not see the relations between pixels.

Date: December 17, 2014.

Where character recognition comes in is by building and implementing methods for computers to classify images of text as a particular letter or word.

There are two main approaches for character recognition through post-processing a static image. One treats images as a graph and uses properties that stem from graph theory, such as closed loops, hard corners, and shortest distance metrics, to classify characters based on these characteristics. These algorithms often use some sort of wavelet transform to distill these characteristics. One failure of these algorithms is that they often cannot correctly classify symbols outside the Latin or Arabic alphabets.[1][2]

Other techniques use wavelet or other discrete signal processing transformations and classify letters just by the properties of this data. These algorithms are often used for classification of more complex alphabets, such as Armenian and Hindi scripts. Since these are frequently more robust, they are used in handwriting recognition as well typed text.[3][4]

Through our work, we looked to use the latter techniques on the alphabet and to do so in a more computationally efficient way. By exploiting properties of the discrete cosine transform, we were able to do so with a large degree of accuracy at a level of $\frac{1}{256}$ compression and perfect accuracy for our data at a compression level of $\frac{1}{64}$.

3. Methods

3.1. **Input.** To begin, we input the desired image that we wish to interpret. For this example, as shown on the figure below, we input a simple 256x256 character using the font Arial. We represent this as a 256x256 matrix of intensity values.

A standard character s with Arial font.

It is important to note that we can use other techniques to obtain the images, such as using a camera, but it would need to go through a cleansing system to proportion and orient the image. By simplifying the problem to images of the recognizable dimensions, we can focus purely on the classification of letters.

---

[1]Lawgali, A., Bouridane, A., Angelova, M., Ghassemlooy, Z. (2011). Handwritten Arabic Character Recognition: Which Feature Extraction Method? International Journal of Advanced Science and Technology, 34.

[2]Legaspe, E. P., Silva, W. S., Fontana, C. F., Dias, E. M. (2011). Automatic Character Recognition Based on Graph Theory: A New Approach to Automation. World Scientific and Engineering Academy and Society (WSEAS), 71-79.

[3]Yang, Y. (2011). Handwritten Armenian Character Recognition Based on Discrete Cosine Transform and Artificial Immune System. Information Technology and Artificial Intelligence Conference (ITAIC), 2, 14-16.

[4]Hangarge, M., Santosh, K. C., Pardeshi, R. (2013). Directional Discrete Cosine Transform for Handwritten Script Identification. International Conference on Document Analysis and Recognition.

3.2. **Compute Discrete Cosine Transform.** Next, we compute the discrete cosine transform on the intensity values of the image. The transform projects our intensity values onto a matrix of 256x256 basis functions of increasing frequency in the $x-$ and $y-$directions from the $1,1$ entry. We compute the transform using the function:

$$D(i,j) = \frac{1}{\sqrt{2N}} C(i)C(j) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} \cos \frac{(2x+1)i\pi}{2N} \cos \frac{(2y+1)j\pi}{2n}$$

$$x, y = 0, 1, \ldots, N-1$$

$$C(u) = \begin{cases} \frac{1}{\sqrt{2}} & \text{if } u = 0 \\ 1 & \text{if } u > 0 \end{cases}$$

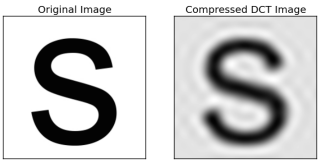
Where $D(i,j)$ is the value of the DCT matrix at row i and column j, and N is the total number of entries. As well as being computationally efficient, on the order of $n \log n$, the DCT has two particularly useful properties for our algorithm and its efficiency. The DCT is globally supported and has good energy compaction. Every entry in the DCT matrix contains some information about every pixel in the intensity domain, and by being very energy compact, most of the information is retained in the low frequency cosines. In other words, we can get a quite good reconstruction from a small number of DCT coefficients.

3.3. **Normalizing DCT Coefficients.** After we compute the DCT coefficients, we normalize it using the Frobenius norm. This step is important to implement as some letters have more area than others. For example, compare the letter i with letter w. By normalizing our DCT matrices, we make sure that the area of a letter does not disproportionately affect our classification. The equation of the Frobenius norm for a 256x256 matrix A with $a_{i,j}$, the i, j^{th} entry is:

$$\|A\|_F = \sqrt{\sum_{i=1}^{256} \sum_{j=1}^{256} |a_{i,j}|^2}$$

Additionally, in the future when expanding our inputs to contain more "raw" images, we are likely to encounter lighter or darker images. Normalizing will address these conditions as well.

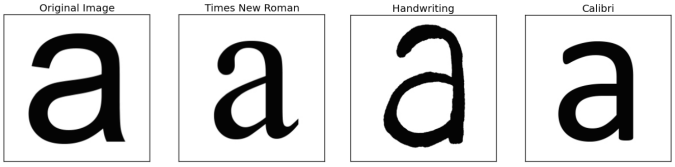
3.4. **Truncation to Optimize Data Set.** Next, we use truncation to optimize the computation. It is important to note that we do not actively truncate the DCT matrix, but by only comparing the top left 16x16 of the matrix, we significantly decrease the amount of comparisons done. This is essentially compression, where we get rid of the coefficients that represent the cosine waves with higher frequencies within the image. Following the DCT's energy compaction, even by comparing .39% of the DCT coefficients, the images we compare are still very recognizable. Although, the compressed image is a bit banded and blurred—missing the details we would get from the higher frequency cosines.



Reconstructing an s with the first 16x16 *entries of the DCT matrix.*

3.5. **Comparison.** And lastly, we compare the 16x16 DCT matrix to all of the other 16x16 matrices in our known data set. We do this by computing an $L_{1,1}$ norm on the difference between each element of our input and the corresponding element of each of our 3 cases of the 7 known letters in our data set.

$$\sum_{i=1}^{16}\sum_{j=1}^{16}|Input_{i,j} - Data_{i,j}|$$

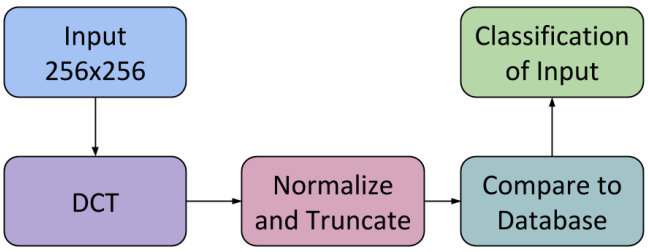


The letter a represented in 4 different fonts.

Because no font will be an exact match, and some will overlap in different regions than others, we cannot determine the letter based on just one comparison. To get the correct classification, we compared our input to every letter in every font, and summed the distance norm for each letter across all known fonts.

4. Results

As an example, we send an Arial s through our system.



Classification System

The distance between our unknown input and each of our known characters are as follows.

Characters	Comparison
Times New Roman - E	0.017443841254305426
Times New Roman - O	0.01592133952436825
Times New Roman - T	0.018453059505304736
Times New Roman – W	0.020437071668311546
Times New Roman - M	0.02213423586817953
Times New Roman - X	0.016991847579674868
Times New Roman - S	0.013827460056745622

Characters	Comparison
Handwriting - E	0.017361727760317081
Handwriting - O	0.019205811478059331
Handwriting - T	0.01697044492948407
Handwriting - W	0.021431469653407476
Handwriting - M	0.025148416660158546
Handwriting - X	0.017769140518566928
Handwriting - S	0.015563086734111877

Magnitude of difference between an Arial s and other letters in Times New Roman and Jamie's Handwriting

Characters	Comparison
Calibri - E	0.015593954573660085
Calibri - O	0.015144700396888311
Calibri - T	0.019321149635743495
Calibri - W	0.019675985575986488
Calibri - M	0.021903080215224691
Calibri - X	0.016929214795709413
Calibri - S	0.010634988896754294

Characters	Comparison
All 3 Fonts Summed - E	0.0503995235883
All 3 Fonts Summed - O	0.0502718513993
All 3 Fonts Summed - T	0.0547446540705
All 3 Fonts Summed - W	0.0615445268977
All 3 Fonts Summed - M	0.0691857327436
All 3 Fonts Summed - X	0.051690202894
All 3 Fonts Summed - S	0.0400255356876

Magnitude of difference between an Arial s and other letters in Calibiri, and the sums of the distance for each letter in all fonts

In this example, because our input has the lowest summed distance from each known s, we choose s to be the classification of the input. One thing to note here is how small the distance magnitudes are. This is the result of normalizing a 256x256 matrix in its entirety, and then only comparing 256 values, rather than the 65536 entries that would sum to a norm of one.

This holds for the majority of our tested cases. For our unknown Arial letters, we classified t, w, m, x, s, and o correctly.

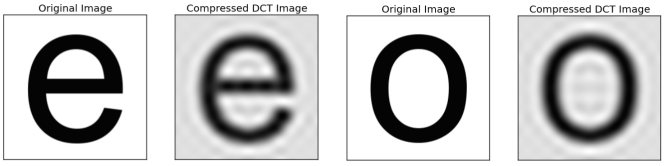
Input	**Classification**	**Margin of Victory**
t	t	0.02141
w	w	0.02022
m	m	0.00847
x	x	0.02450
s	s	0.01025
o	o	0.01513

Correct classifications of Arial letters by our training set.

4.1. **Failures.** One large flaw of our algorithm is that it improperly classifies e as o at 16x16 compression. This is fixed at the less compressed 32x32 truncation, but in turn quadruples the computing time.

	Input	Classification	Margin of Victory or Failure
At 16x16 **Compression**	o	o	0.01513
	e	o	-0.00513
	Input	**Classification**	**Margin of Victory or Failure**
At 32x32 **Compression**	o	o	0.00251
	e	e	0.00369

The incorrect classification of e at 16x16 *compression versus the correct classification at* 32x32*.*

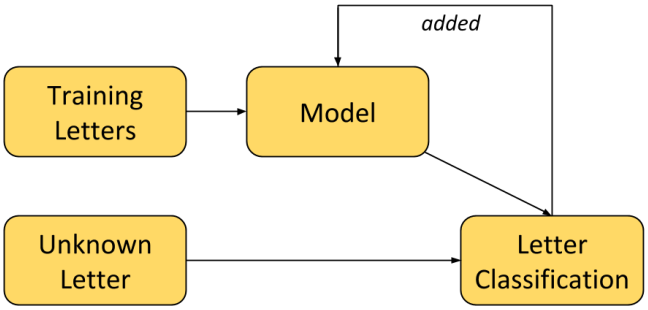


Reconstructing the image for letter e and o with the compressed DCT matrix.

It is fairly obvious to see that the compressed e and o look very similar to each other. This is the result of artifacts in the DCT—banding that occurs around the letters. This effect occurs when you reconstruct an incomplete representation of the DCT matrix. The two letters are very similar to begin with in terms of shape, and the incomplete reconstruction adds artifacts to the o that mimic the crossbar of the e, and the crossbar is less accentuated in the truncated e. Since this is happening at different levels across different fonts, this leads to the algorithm misclassifying our e in favor of an o. This is one of the downside of compression; there exists a trade-off between resolution (and transitively accuracy) and memory allocation.

There are two solutions to fixing our problem. One one hand, we can use a larger data set to minimize the error. Currently, we are using only 3 different fonts with 7 different characters. This is not a big enough data set to obtain absolute certainty when comparing. Ideally, we want thousands of cases for all characters. It is important to keep in mind that there is a direct correlation between expanding our data set and increasing the amount of computational work, as there are more comparisons to be done against the input. On the other hand, we can use less compression. We reiterated our algorithm with a 32x32 compression, and the algorithm successfully classified e and o properly. However, by increasing the size of the DCT matrix, we also increase the amount of computational work by a factor of four. For this instance, there is a trade-off between the accuracy of our algorithm and the computational cost of running it. Additionally, this notion will become much more significant to consider as the size of our data set increases.

4.2. **Supervised Machine Learning.** In our algorithm, we utilized elements of supervised machine learning to have computers automate the classification. In supervised machine learning, we use a training set. In our case the training set was the known fonts, input that has been pre-interpreted by humans. We also decided how the algorithm responds when compared to the chosen data. Whereas in unsupervised machine learning, the computer does not take in a pattern. Rather, it develops a pattern on its own through many iterations. The decision to use supervision for the framework of our problem makes perfect sense.



A Basic supervised machine learning process.

If we were to implement our algorithm at a larger scale, we would include the addition of the classified letters beyond a single iteration into our model to truly utilize the power of machine learning. This has the wonderful benefit of distilling better characterizations of letters, while at the same time increasing the number of comparisons to which an input is tested on.

4.3. **Software.** For our implementation, we used an open source library called OpenCV (Open Source Computer Vision).[5] With its modular structure, the package included many shared or static libraries that allowed us to easily process and analyze images. More specifically, we used the functions *imread* and *dct*. *imread* allowed us to efficiently translate the inputted image to the corresponding intensity values. Whereas *dct* computed the discrete cosine transform given the matrix of intensity values. We attempted to implement our own DCT function, but the time efficiency lacked significantly compared to the one provided in OpenCV. In addition, we used NumPy, a notorious package in Python responsible for many scientific computing applications.[6] NumPy provided us with efficient tools to effectively tackle large, multi-dimensional arrays. For example, the use of matrix multiplication and matrix slicing would have added significant computational burdens without the efficiency of NumPy. NumPy proved to be invaluable for the normalizing and truncating steps in our algorithm.

[5]Bradski, G. (2008, January 15). Open Source Computer Vision. Retrieved from http://opencv.org/

[6]Jones, E., Oliphant, T., Peterson, P. (2001). SciPy: Open Source Scientific Tools for Python. Retrieved from http://www.scipy.org/

5. Conclusion and Future Work

Although we did not achieve perfect results at the 16x16 compression, our algorithm has the capacity to be generalized and extended. The points of improvement largely lie in the database itself. By extending our comparison set to contain more letters and more instances of them, in a purely statistical sense, we improve our chances of correct classification. Furthermore, we can improve our algorithm by implementing more advanced machine learning algorithms, such as support vector machines or the related nearest neighbor classification. In addition, because we are already computing the discrete cosine transform, we can use feature recognition based on the DCT with minimal increases in computation time.

Our algorithm also requires a relatively large image of a letter. 256x256 images that are entirely a single letter either need to be large letters, taken from a camera with a high pixel density, or be scaled up to size. As for pixel density, as the quality of cameras and their sensors advance inexorably forward, our seemingly large requirements will become more and more trivial. There are image processing tools that would be able to scale smaller text up to this size with relatively good detail retention, which could be addressed in the future.

In conclusion, we have shown that by using the discrete cosine transform, images of letters can be classified in a computationally efficient way. Our algorithm can easily be scaled up to include a larger database of characters, both in Latin script and otherwise, while simultaneously increasing its accuracy.