CS229 Project Report: Cracking CAPTCHAs

Learning to Read Obscured and Distorted Text in Images

Ryan Fortune
rfortune@stanford.edu

Gary Luu
gluu@cs.stanford.edu

Peter McMahon
pmcmahon@stanford.edu

***Abstract*—In this report we discuss the application of machine learning techniques to the problem of reading text in CAPTCHAs. We first present techniques for reading a CAPTCHA that requires only single character classification and trivial segmentation, and show that we are able to successfully read such CAPTCHAs. We then show how we attempted to read a more difficult CAPTCHA; one that required significant effort to segment before individual character classification can be attempted. We detail our noise removal, line removal, and clustering-based segmentation techniques, and present our results for this second type of CAPTCHA.**

I. Introduction

CAPTCHAs are widely used on the internet as a security measure to prevent bots from automatically spamming registration forms. They essentially form a weak Turing Test in order to ensure that the user filling out a form on a website is human. By identifying weaknesses in current CAPTCHA techniques, researchers can enable the development of CAPTCHAs that are more resistent to attack by resourceful spammers.

Moreover, there are similarities between the tasks of reading CAPTCHAs and reading handwritten text, so the development of more effective techniques for reading CAPTCHAs may yield insights into how one can more effectively apply machine learning techniques to the problem of handwriting recognition.

There are many different CAPTCHA implementations, many of which are not proprietary. We used Eliot's PHP CAPTCHA generator [1] and BrainJar's .NET CAPTCHA generator [6], because of the availability of source code allowing us to generate data sets for supervised learning.

This report is divided into two main sections: the first concerning our attempt to read Eliot's CAPTCHAs, and the second concerning the more difficult task of reading BrainJar CAPTCHAs.

II. Eliot's PHP CAPTCHA System

Eliot's PHP CAPTCHA (EPC) system [1] is a CAPTCHA generating library freely available on the web under the GPL. An example of a CAPTCHA generated with EPC is shown in Figure 1.

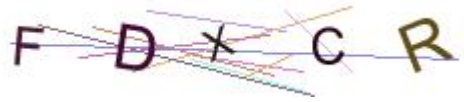

Fig. 1. CAPTCHA produced by Eliot's PHP CAPTCHA.

The important characteristics to note in this CAPTCHA are the following. First, the individual characters are placed in invisible equally-sized subdivisions of the image. Second, the characters use regular computer typeface (with random font color) without any transformations other than translation, rotation and size scaling. Third, lines have been randomly drawn across the image.

The EPC system thus produces CAPTCHA's solved by trivial segmentation techniques: to segment the image, we simply copy predetermined sections of the image. Solving EPC CAPTCHA's is thus a character recognition problem.

A. Character Classification using Raw Pixels as Features

We first attempted to build a character classifier using features represented by the raw data (image pixels). Specifically, we defined an m-by-n matrix X, where m is the number of training examples, n is the number of features ($n = wh$ where w and h are the width and height of each training image in pixels, respectively; throughout this report we assume $w = h = 50$), and X_{ij} is the grayscale value[1] of the jth pixel in the image of the ith training example. The jth pixel is defined in a way that allows the 2D training images to be represented using a 1D array of pixel features. An m-dimensional vector Y is defined that stores the actual class of each training example. $Y_i = j$ if the ith training example is the jth character.

Our first attempt at using these features was a classification algorithm based on Multinomial Logistic Regression. This approach yielded unsatisfactory results – even when using training sets where $m \gg n$ we experienced significant difficulties with obtaining convergence, both with gradient

[1]We use the opposite to the usual convention for representing grayscale pixel values: we convert a grayscale pixel value $(x, y)_{\text{old}}$ to our format by subtracting it from the maximum value: $(x, y)_{\text{new}} = 255 - (x, y)_{\text{old}}$. This has the more natural interpretation of "0" meaning a pixel is "off", and "255" meaning it is "on".

m	Linear Kernel Accuracy	RBF Kernel Accuracy
260	25.7%	56.54%
2600	98.8%	99.00%

TABLE I
IMPACT OF SVM KERNEL CHOICE ON ACCURACY. TRAINING SET SIZE m, AND TEST SET SIZE OF 26000.

descent and Newton's method.

B. SVM Classification

We used the libsvm [8] Support Vector Machine package to perform multi-class classification on the raw pixel features. We attempted to use the popular SPIDER package, which is native to MATLAB, but found that it was unable[2] to train datasets as large as ours.

The libsvm library provides support for several kernels. We determined the classification accuracy with libsvm using the linear[3] kernel, and the radial basis function[4] (RBF) kernel. The libsvm authors recommend that the RBF kernel be used with kernel parameter γ, and penalty parameter[5] C, chosen by cross-validation. Our results are shown in Table I. The linear kernel provides good results, and has the advantage that when selected, the SVM will train far more quickly than with the RBF kernel. The default cross-validation training implementation in libsvm produces excellent results, but requires considerable computing resources for large training sets such as ours. The $m = 2600$ case took approximately 36 CPU-hours to train. libsvm tests 272 configurations of C and γ values using a basic grid search to determine those that are optimal. Retraining the SVM 272 times on such a large dataset is very time consuming.

Our results were obtained by training on two different set sizes, and in both cases testing against an independent set with 26000 examples (1000 of each letter in the latin alphabet). With the linear kernel we used the libsvm default $C = 1$, and with the RBF kernel we used libsvm's grid search optimization of C and γ.

C. Features

Due to the significant computational resources required to train the SVM with large datasets when using the raw pixels as features, we investigated two possibilities for reducing the dimension of the feature space. We obtained a reduced space using principal component analysis (PCA), and we developed a custom set of features based on statistical properties of the image pixels.

[2]With $n = 2500$ features, the SPIDER SVM could train on at most $m = 20$ examples.

[3]$K(\mathbf{x}_i, \mathbf{x}_j) = \mathbf{x}_i^T \mathbf{x}_j$

[4]$K(\mathbf{x}_i, \mathbf{x}_j) = \exp(-\gamma \, ||\mathbf{x}_i - \mathbf{x}_j||^2)$

[5]The parameter C is that in the SVM optimization problem: $\min_{\mathbf{w},b,\xi} \frac{1}{2}\mathbf{w}^T\mathbf{w} + C\sum_{i=1}^{m} \xi_i$.

New Dimension k	Test Set Accuracy
100	3.1%
250	18.5%
1500	26.2%
2500[a]	56.5%

TABLE II
IMPACT OF NEW FEATURE SPACE DIMENSION ON ACCURACY AFTER APPLYING PCA. TRAINING SET SIZE $m = 260$ ON SVM WITH RBF KERNEL, AND TEST SET SIZE OF 2600. ORIGINAL FEATURE DIMENSION $n = 2500$.

[a]This line shows the accuracy when the feature space dimension is not reduced.

To determine the efficacy of PCA for our problem, generated three new training sets using PCA, by mapping the original n-dimensional feature space onto a new k-dimensional space, with $k = 100$, $k = 250$ and $k = 1500$ respectively. We trained the SVM with these new training sets, and in each case computed its accuracy using a common test set. Table II shows the results. Reducing the feature set dimension using PCA clearly has a significant negative impact on accuracy, so we did not continue to use PCA.

We calculated a smaller set of "median" features that would hopefully capture trends of a given character with many fewer features. The key to this approach was reducing a two dimensional image into one dimension. We computed row sums and column sums separately, and found the median pixel. This is the pixel where the sum of the values of pixels to the left equals the sum of the values of pixels to the right with the same approach for the vertical direction. We also computed 25% and 75% points as well as a few linear combinations of these three values. Once we had the median point we also calculated radial features using the distance from the median point as the axis so the sum of pixels closer to the median point than the radial median equals the sum of those further away. Altogether this created 18 features.

Because these features only represent trends of the data we supplemented these with other features that merged together pixels in the original image. We merged cells to reduce a 50x50 character into a 8x8 character. This left us with 82 features, much fewer than before. With two-class logistic regression, we were able to distinguish between two character classes with over 95% accuracy, however these features did not perform well once we moved to multi-class classification.

D. End-to-End Results

We have shown that we are able to classify individual characters with high accuracy (99+%) when the characters do not have any lines through them. Since segmentation is trivial in Eliot's CAPTCHA system, we would expect to be able to correctly read a 5-character CAPTCHA with no lines

"Median" Features Accuracy	Custom Features Accuracy	Raw Pixels Feature Accuracy
11.9%	25.0%	56.5%

TABLE III
IMPACT OF CUSTOM FEATURES ON ACCURACY. TRAINING SET SIZE $m = 260$ ON SVM WITH RBF KERNEL, AND TEST SET SIZE OF 26000.

Lines	Character Accuracy	CAPTCHA Accuracy
0	98.8%	94.2%
10	90.8%	62.0%

TABLE IV
ACCURACY OF CLASSIFICATION OF WHOLE CAPTCHAS. USED MODEL GENERATED WITH AN SVM WITH RBF KERNEL, AND TRAINING SET SIZE $m = 2600$. TEST CAPTCHAS HAD EITHER NO LINES OR TEN RANDOMLY PLACED LINES. THE TEST SETS HAD 50 CAPTCHAS EACH, EACH 5 CHARACTERS LONG.

with probability greater than $0.99^5 \approx 0.95$. We conducted an experiment to verify this by generating 50 CAPTCHAs with no lines, segmenting them, classifying each character, and then counting how many CAPTCHAs were correctly read (we define a CAPTCHA being correctly read as one that has all of its characters correctly classified).

We also sought to determine what the impact of having random lines placed in the CAPTCHA is. We similarly generated 50 CAPTCHAs with 10 randomly drawn lines each, and attempted to read them. Our results for both experiments are shown in Table IV. We obtained the expected result for the unobscured text, but with lines in the CAPTCHA, the character classification accuracy drops by nearly 10%. This has a dramatic impact on the reading accuracy for the whole CAPTCHA, since now the probability of correctly classifying all 5 characters in a CAPTCHA is estimated as $0.908^5 \approx 0.6172$. Our observed result is in agreement with this.

III. MODIFIED BRAIN JAR CAPTCHA

We obtained a CAPTCHA program from the Brain Jar website that contained warped text with noise. In order to make classifying this CAPTCHA more difficult, we modified it to produce splines lying horizontally across the text. An sample output is shown in Figure 2. reCAPTCHA [7], the current reference standard CAPTCHA from the inventors of the field, uses text that is similarly obscured.

A. Individual Character Recognition

The BrainJar CAPTCHA is capable of using any characters in the text, but in this report we focus on recognition of the

Fig. 2. Example Modified Brain Jar CAPTCHA

Training set size m	Test Set Accuracy
500	52.4%
1000	49.6%
2000	56.0%
5000	72.2%
7000	76.2%

TABLE V
IMPACT OF TRAINING SET SIZE ON SVM ACCURACY USING *Linear Kernel*. TRAINING AND TEST SETS USED CHARACTERS WITH NO NOISE IN THE BACKGROUND OR FOREGROUND. THE TEST SET HAD 5000 EXAMPLES.

digits 0-9, purely due to computational resource constraints.

The character classification problem in BrainJar's CAPTCHA appears at first glance to be very similar to that in Eliot's CAPTCHA. Instead of individual characters being translated, rotated and scaled, they are warped. In addition, the BrainJar CAPTCHA also has noise both in the background and in the text (the foreground). As we discuss shortly, these differences have a significant iimpact on the individual character classification performance in the BrainJar CAPTCHA.

We first attempted to classify individual characters generated without any background or foreground noise. Table V shows our achieved accuracy as we increase the size of the training set. We used a linear kernel with the libsvm SVM. With the RBF kernel, with optimization of the parameters C and γ as before, we were able to obtain an accuracy of 32.1% using a training set size of $m = 200$. Available computational resources limited our ability to investigate the performance of this kernel using larger training sets, and far superior results were obtained using the linear kernel with training set sizes that were not tractable for us when using the RBF kernel. Hence our classification in this section is all based on an SVM using a linear kernel.

In all cases we have used raw pixel values as features. We tested our pixel statistics-based features, but these again yielded sub-optimal accuracy results (22.8% accuracy versus 38.9% accuracy when using raw pixel features, in one experiment that we conducted).

Table VI shows the how using training sets with noise versus those without noise affect accuracy on test sets that are noisy and not noisy. The objective of this analysis was to determine whether we should train our SVM on noisy data or on clean data. We also sought to determine the importance of noise removal in CAPTCHAs that we want to read. In this table, blank entries denote quantities we didn't conduct experiments to measure.

We found that it is best to train the SVM using clean data. This gives us an optimal accuracy of 76.2% if the test set is also clean. Our optimal accuracy on data with both

Training \ Test	No Noise	FG Noise Only	BG and FG Noise
No Noise	76.2%	65.6%	63.0%
FG Noise Only		52.1%	49.9%
BG and FG Noise			40.8%

TABLE VI
IMPACT ON ACCURACY OF NOISE IN TRAINING AND TEST SETS. SVM WITH LINEAR KERNEL. THE TRAINING SETS HAD $m = 7000$. THE TEST SETS HAD 10000 EXAMPLES EACH.

Fig. 3. k-means with radius and pixel intensity. Non-black pixels are made white.

background and foreground noise is 63.0%. It is interesting to note that when training on clean data, the accuracy decreases most significantly when foreground noise is included. The addition of background noise in addition does not have a large additional impact.

B. Noise Removal

Noise removal is important for line removal, segmentation and for character classification. We have shown how the presence of noise adversely affects our accuracy in single character classification. The following section briefly discusses the negative impact of noise on the efficacy of our line removal techniques.

Looking at Figure 2, we can see that there is noise across the image in both the background and in the characters themselves. We tried two variations of features to use for k-means clustering to remove the noise. First, we tried clustering on the intensity of the pixel and a count of pixels within a short 2-pixel radius that had a similar intensity, producing Figure 3.

We next tried clustering using the pixel intensity as well as the size of the "Fill" region of the pixel, which is defined as the size in pixels of the largest contiguous area with similar pixel intensity that the pixel lies in. Figure 4 shows a sample result following the application of this method. This noise-removal technique clearly yielded superior results.

C. Line Removal

We attempted two methods of line removal: one using dynamic programming to optimize a path, and another using a probabilistic optimization procedure.

Fig. 4. k-means with pixel and "Fill" score

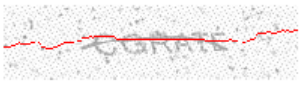

Fig. 5. Line Removal without Clustering Beforehand.

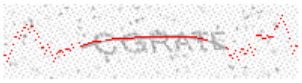

Fig. 6. Line Removal with Clustering Beforehand.

1) Slope Penalty and Dynamic Programming: Based on two observations, the fact that the splines typically spanned the image horizontally and generally would not have any sharp changes in slope, we devised a dynamic programming algorithm to find and remove the spline in the text by iterating across the columns of the image, and using the following recurrence

$$Score_{i,j} := \max_k \{Score_{i-1,k} + \gamma_{k,j}\}$$

where the farther k and j are apart, the lower the value of $\gamma_{k,j}$.

$$\gamma_{k,j} = ImageHeight - |j - k|$$

Figures 5 and 6 show the results of this algorithm, respectively with and without clustering before applying the algorithm.

2) Monte Carlo Line Removal: We used a probablistic method to remove lines. For every pixel, we computed the sum of its value and its immediate neighbors. We choose a starting point in the center column, and then progress to the left and right separately. Given a previous pixel we look at a column three pixels to the left/right of the previous one and only consider the five pixels vertically closest to the previous to maintain a continuous relatively flat line. We create a discount value equal to half of the previous pixel's sum, and subtract this from each of the five sums. We then randomly choose among these five by using the differences as weights. If all five differences are zero, we consider this the end of the line. We repeat 100 times and use the pixels that lead to the highest combined sum.

D. Recompletion

Once we know where the line is, we want to remove it from our images. Originally we set the value of any pixels on the line or immediately above and below to zero. Unfortunately this cut most of the letters in half. We implemented a simple method to reconnect these characters by looking at the pixels immediately above and below. Since clustering removes most of the noise, we added the pixels back to the image if both the above and below pixels have non-zero values. As to be expected, this restores vertical lines well, partially fixes curves, and does poorly replacing horizontal lines in characters.

Fig. 7. Successful Segmentation (LTFXQL)

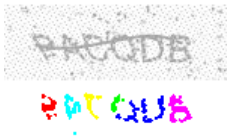

Fig. 8. Unsuccessful Segmentation (PACQDB)

E. Segmentation

The CAPTCHA we were testing did not have merged characters, so there was some white space between any two characters in the original image before noise was added. We assumed that this gap was at least two pixels. However, the image we analyze has had both foreground and background noise added so this assumption likely doesn't hold. k-means clustering allows us to remove much but not all of the background noise, but it also takes a few pixels out of the main characters.

For segmentation purposes we considered a pixel to be part of a character if two of four pixels in its immediate area have non-zero values. We then form groups of the non-zero characters pixels if there are other character pixels sufficiently close. We then take in to account where the line was removed and merge groups across the line. This leaves us with around 50 pixel groups for 6 character CAPTCHAS.

In order to merge more groups together, we look at the rightmost and leftmost ends of each character group. Since English characters tend to be vertically oriented, if any of the smaller groups ranges are relatively close to the range of one of the largest, but not close to any of the others, we merge them together. Ideally this prevents merging two separate characters together. After this, we take the six largest groups and consider them to be the six characters. We sort the groups based on their leftmost pixel to restore the ordering.

F. End-to-End Results

We performed an end-to-end test of our CAPTCHA cracking system on 100 CAPTCHAs generated using the modified BrainJar generator, each six characters long, using only the digits 0-9. Our experiment attempted to read each CAPTCHA by removing noise, removing the line (with recompletion), segmenting the image, and then performing individual character recognition on each character.

Table VII shows the results. We present a count of the number of test CAPTCHAs in which $0, 1, \ldots, 6$ characters were correctly identified. Not a single whole 6-digit CAPTCHA was correctly read. However, one CAPTCHA had five of its six digits correctly classified. In 41 cases, not a single character was correctly identified. Out of 600 digits in total, 113 were ultimately correctly classified.

# Chars Correct	0	1	2	3	4	5	6
Counts	41	26	19	8	5	1	0

TABLE VII
CLASSIFICATION OF WHOLE CAPTCHAS: COUNTS OF CORRECT INDIVIDUAL CHARACTER CLASSIFICATION PER CAPTCHA. USED MODEL GENERATED WITH AN SVM WITH LINEAR KERNEL, AND TRAINING SET SIZE $m = 10000$. THE TEST SET HAD 100 CAPTCHAS, WITH 6 CHARACTERS (DIGITS 0-9) IN EACH.

The errors result from a combination of effects from three imperfect procedures. First, segmentation may fail. Second, recompletion after line removal may fail, which would result in a portion of a character being deleted. In some cases this is unimportant, but in others a significant horizontal portion of a character may be permanently removed. The fifth character in Figure 8 demonstrates this – a "D" was transformed into a "U". Third, the individual character classification is imperfect, and even more so in the presence of noise that is not completely removed.

IV. CONCLUSIONS

We have demonstrated that SVMs can be used to classify distorted individual characters with a success rate of 99%. This leads to the ability to read unobscured CAPTCHAs amenable to trivial segmentation with a success rate of approximately 95%. We have also presented methods for attacking the problem of segmentation in the most difficult type of CAPTCHA currently used.

A. Comparison with Related Work

In [3], Chellapilla et. al. present results for individual character recognition in CAPTCHAs using neural networks. They are able to achieve an accuracy of 99+%, even on warped text; this is considered the current optimal solution [4]. In an earlier paper, [2], Chellapilla et. al. were able to crack some CAPTCHAs featuring non-trivial segmentation characteristics with accuracy of approximately 4%. Yan and Al Ahmad [4] recently reported on their cracking of the Microsoft CAPTCHA. However, to our knowledge there has been no successful breaking of CAPTCHAs using warped text with splines, such as our modified BrainJar CAPTCHA, and reCAPTCHA [7].

Proper segmentation is a crucial step for any character recognition effort and as such there is much prior work that has already been done. Unfortunately for us, most of these address situations without foreground noise, meaning each character is contiguous, a situation we do not have. Some of the gaps and holes within our characters could be removed

by blurring the image, but doing so would make individual character recognition more difficult by obscuring character features. Khan's thesis [5] on segmentation suggests using a drop-fall algorithm for general purposes and also suggests ways of identifying disconnected characters. The drop fall algorithm described essentially reverses our approach and uses the whitespace instead of the character pixels themselves for segmentation, but this approach would similarly suffer with foreground noise. Khan's method handles disconnected characters in way that relies on the disconnect occurring in specific situations, which is not applicable in CAPTCHAs. Highly accurate segmentation remains an open problem for modern CAPTCHAs.

B. Future Work

One technique for reading CAPTCHAs, inspired by the way humans read CAPTCHAs, is to use a feedback mechanism between the segmentation and character classification stages. In the cases where the segmentation has been done incorrectly, the certainty of a prediction made by the SVM will be low. In this event, the segmenter should be required to reattempt segmenting that region of the image. One simple possible implementation of this idea is to slide a fixed-size window over the data and only accept a segmentation involving the current window position if the image in that window yields a character prediction with high certainty.

REFERENCES

[1] E. Eliot. PHP CAPTCHA. URL: http://www.ejeliot.com/pages/php-captcha

[2] Kumar Chellapilla, Patrice Y. Simard. *Using Machine Learning to Break Visual Human Interaction Proofs (HIPs).* NIPS 2004.

[3] Kumar Chellapilla, Kevin Larson, Patrice Simard and Mary Czerwinski. *Computers beat humans at single character recognition in reading based human interaction proofs.* Second Conference on E-mail and Anti-Spam (CEAS) 2005.

[4] Jeff Yan and Ahmad Salah El Ahmad. *A Low-cost Attack on a Microsoft CAPTCHA.*, ACM Conference on Computer and Communications Security, 2008.

[5] Salman Amin Khan. *Character Segmentation Heuristics for Check Amount Verification.* Ph.D. Thesis, MIT, 1998.

[6] Mike Hall. CAPTCHA Image URL: http://www.brainjar.com/dotNet/CaptchaImage/

[7] reCAPTCHA URL: http://recaptcha.net/

[8] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: a library for support vector machines. URL: http://www.csie.ntu.edu.tw/~cjlin/libsvm

[9] SPIDER SVM library for MATLAB. URL: http://www.kyb.mpg.de/bs/people/spider/