

General solution for alphanumeric text-based CAPTCHAs using Convolutional Neural Networks

50.038 Computational Data Science

Joel Huang
1002530

Karthic Harish Ragupathy
1002265

Ivan Zhou Yue
1002238

December 3, 2018

Abstract

We align ourselves to the arguments of [1, 3, 11, 13], highlighting the ease of breaking alphanumeric CAPTCHAs today. CAPTCHAs are meant to be reverse Turing tests, implemented in order to differentiate humans from bots. Today, a number of sites still naively use alphanumeric CAPTCHAs as a first line of defence against automated attacks, flagging themselves as vulnerable targets to potential attackers. Recent advances in convolutional neural networks (CNNs) have produced the new state-of-the-art in object detection [2, 6]. Combined with long-known image segmentation and processing algorithms, old-school alphanumeric CAPTCHAs are solvable to a extremely high degree. In this paper, we aim to show that it is time to phase out alphanumeric text-based captchas once and for all.

1 Introduction

The problem of CAPTCHA breaking is important because it is essentially a contradiction to the web-based Turing test (and by extension, the reverse Turing test). In the last decade, the use of text-based CAPTCHAs to block automated requests have largely declined, due to improvements in the ability of paid automatic solvers, and the increasing accuracy of machine vision algorithms via convolutional neural networks, which makes it easy for any programmer to prepare and deploy a decently accurate model. However, many production-level systems have previously adopted text-based CAPTCHAs as security strategies and have not updated their systems with a more secure model. Some common types of text-based CAPTCHAs are displayed in Fig. 1.

1.1 Problem Framing

We frame the CAPTCHA-breaking task as a character-level object detection problem. A correct solution involves producing a subset of characters, chosen from a set of 62 alphanumeric (A-Z, a-z, 0-9) characters that exactly matches the characters that were used to generate the CAPTCHA. We constrain the solution further as follows: the predicted string must match the original string’s length and sequence; and all characters are mutually exclusive, meaning that the prediction must be case-sensitive, so the model must be able to handle hard-to-distinguish characters like ‘o’, ‘O’ and ‘0’.

1.2 Methodology

Our training stage will involve a training set, which we describe in the next section. Input images pass through an Otsu preprocessor, which thresholds the images to remove noise, then are fed into the neural network for training. The test stage is similar, where the test set will also be fed through the Otsu preprocessor, then into the neural network for inference. We use different metrics for character-level prediction (IOUs, class-level mAP) and for accurate CAPTCHA string sequencing ($mAP^{len(captcha)}$). We also introduce a novel step, which we term unseen testing. In this step, our pipeline is fed CAPTCHA images that are totally unseen, independent of the training and test sets. The same metrics apply to evaluate our model.














Type	Example	Source	Features
Solid CAPTCHA		Discuz!	Character independent, texture background, some interference
		Slashdot	A large number of interference lines and noise points
		Gimpy	Multiple strings, overlap, distortion
		Google	Unfixed length, distortion, adhesion
		Microsoft	Double-string, unfixed length, uneven thickness, tilting, adhesion
Hollow CAPTCHA		QQ	Hollow, shadows, interference shapes
		Sina	Hollow, adhesion, interference lines
		Yandex	Hollow, virtual contours, distortion, adhesion, interference lines
Three-dimensional CAPTCHA		Scihub	Hollow, shadows, interference lines, noise points
		Teabag	Grids, protrusion, distortion, background and character blending
		Parc	Colorful, shadow, rotation, zoom Special characters
Animation CAPTCHA		Program generating	Multiple characters jumping
		Hcaptcha	Multilayer character images blinking transformation

Figure 1: Common types of text-based CAPTCHAs used on the web

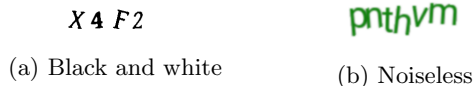


Figure 2: Organic CAPTCHA classes



Figure 3: Comparison between organic and synthetic

2 Dataset and Visualization

2.1 Terminology

All text-based CAPTCHAs are technically ‘synthetic’, having been rendered at some point using different libraries and languages. However, in principle, we have no knowledge of the exact methods particular sites use to generate their CAPTCHAs. We thus consider scraped or downloaded CAPTCHAs as in-the-wild, or organic data, having been generated and deployed for actual use on production-level systems. We contrast this organic data to images that we are able to generate ourselves, which we will term as synthetic data.

2.2 Gathering Organic Data

The organic dataset is made up of roughly 15k text-based CAPTCHA images retrieved from the web. The organic portion of our dataset is merged from two online sources. We collected 15k real-world labelled CAPTCHAs of 2 different dimensions from these sources (see Fig. 2):

- 10k black and white, WordPress PHP plugin with over 1+ million downloads, of dimensions (72×24). These are scaled up to (200×75) for training.
- 5k noiseless CAPTCHAs, scraped from online sources, of dimensions (200×75)

2.3 Learning labels from Synthetic Data

While our organic data is immensely valuable as real-world ground truths, we are unable to localize characters within the images in the current dataset. We hypothesize that synthetic data will be able to help us to solve this problem, inspired by synthetic data learning [12].

For this task, we have written a custom Python CAPTCHA generation library that is able to generate character-based CAPTCHAs with variations in font, color, and spacing; carry out transformations like warping, vertical offset, rotation; and create irregularities in the form of background variation,

point and line noise, and line and arc occlusion. Most importantly, we are able to create pixel-perfect localization labels for our synthetic data.

2.4 Deep learning for annotation

In order to validate our hypothesis that we should be able to learn labels for our organic dataset, we analyzed the PHP portion of our dataset. The dataset draws characters exclusively from a set of variants of Gentium (regular, italic, bold and bold italic) with replacement, including random rotations and spacing within certain thresholds. We then generate 50k synthetic samples, which we argue are highly similar to the 10k organic samples, and learn localization labels on the organic set from the ones in the synthetic set, using the same architecture as our final model, RetinaNet [6]. We then manually fixed any labels that were wrong, and fixed bounding boxes that were off. This task was made largely trivial because our proxy annotation model did an extremely good job, and enabled us a semi-automated method of gaining bounding box information for our entire organic dataset.

2.5 Synthetic Data

Early on, we noted that in this problem, in-the-wild, real-world CAPTCHAs have extremely small dissimilarity to synthetic data: the procedure to generate synthetic data is almost identical to the procedures used to generate ‘production-grade’ alphanumeric CAPTCHAs. Typically, a 4-6 character alphanumeric string is generated at pseudo-random, then rendered together with variable font families, font weights and sizes, positional offsets, typographic artifacts such as negative kerning, affine transformations, and filtering. There have been many instances of successful learning with synthetic data [3, 12].

To create diversity sufficient to replace the lack of organic data available, we modified the image parameters as follows:

2.5.1 Uniform random character sampling

We randomly sample alphanumeric characters, using the full range of 62 upper and lower case ASCII and single digit numerals.

2.5.2 Noise generation

We design **curves**, arc and line occlusions that occur randomly, creating 0 – 3 horizontal cuts that pass through at least 50% of the letters. We also introduce **dots**, creating 20 – 35 dots scattered throughout the image. By creating these artifacts, we hope to increase the invariance to noise in our classification step.

2.5.3 Color randomization

Colors are randomly allocated within the RGB spectrum. For visibility, we constrain background colors to values above (238, 238, 238) and foreground colors below (200, 200, 200).

2.5.4 Font selection

We render the most common types of fonts in this experiment. In general, there are a few broad classes of fonts, including sans-serif, serif, slab, and monospace. We choose the most common sans-serif and serif fonts from the Google Fonts open-source directory. We picked Droid Sans Mono and Roboto for sans-serifs; and Gentium and Merriweather for serifs. We also introduce different font weights within the font families.

2.5.5 Alignment and size

The alignment of the strings are varied. Each of the characters may or may not be in the same line. We define curves with randomized parameters that are used to offset character heights.

2.6 Visualizations

Our final combined dataset consists of 50,000 CAPTCHA images, with three main divisions: color, noiseless, and black and white. The table below describes each class of CAPTCHA and its characteristics, and the distribution is shown in Fig. 4.

Class	Character count	Character types
Color	4 – 6	Full alphanumeric
Noiseless	6	Lowercase and digits
BW	4	Uppercase and digits

2.6.1 Relationship analysis using PCA & t-SNE

In order to understand the relationship between the features in our dataset, we employ Principal Component Analysis (PCA) and t-distributed Stochastic

Neighbour Embedding (t-SNE) to reduce the dimension of the images. Each character in the CAPTCHAs is segmented, extracted and resized to fit a (60×60) pixels box (Fig. 5). Secondly, a threshold transformation is applied to convert the characters into black and white images with a singular RGB value, ranging from 0 to 255. This results in a 3600-dimensional vector of pixel values. PCA is used to reduce the dimensionality of each vector and to find directions (principal components) that contain the largest spread/subset of data points or information (variance) relative to all the other data points. t-SNE is used for the same purpose. A non-linear, probabilistic dimensionality reduction method whereby it aims to convert the Euclidean distances between points into conditional probabilities. A Student's-t distribution is then applied on these probabilities which serve as metrics to calculate the similarity between one datapoint to another. Our results can be seen in Fig. 6 and Fig. 7.

As observed from the PCA scatter plots, there exist a few discernible clusters evinced from the collective blotches of colors. However, there is no clear distinction between the clusters. The t-SNE plots are more useful in distinguishing the various characters. For example, the left section of the plot represents characters with mostly straight line segments (l, T, i) while the top section represents characters with loops (o, O, 8, b). Due to the close clustering and similarity between many of the characters, we conclude that the CAPTCHA images we're dealing with are challenging and hence, we expect reduced accuracy scores for certain characters that are identical to others, such as o & O, l & I.

3 Data Processing

3.1 Early attempts at algorithmic character segmentation

We consider several preprocessing techniques, primarily to reduce noise. The objective is to simplify the input to a state where the segmentation algorithms can extract the letters. Some preprocessing algorithms we attempted include:

- **Background/colour removal.** This is most useful against CAPTCHAs that use colour as a defence mechanism. It results in a grayscale image with the foreground pixels retaining their intensity and background pixels being white. However, there are CAPTCHA variants with gradient

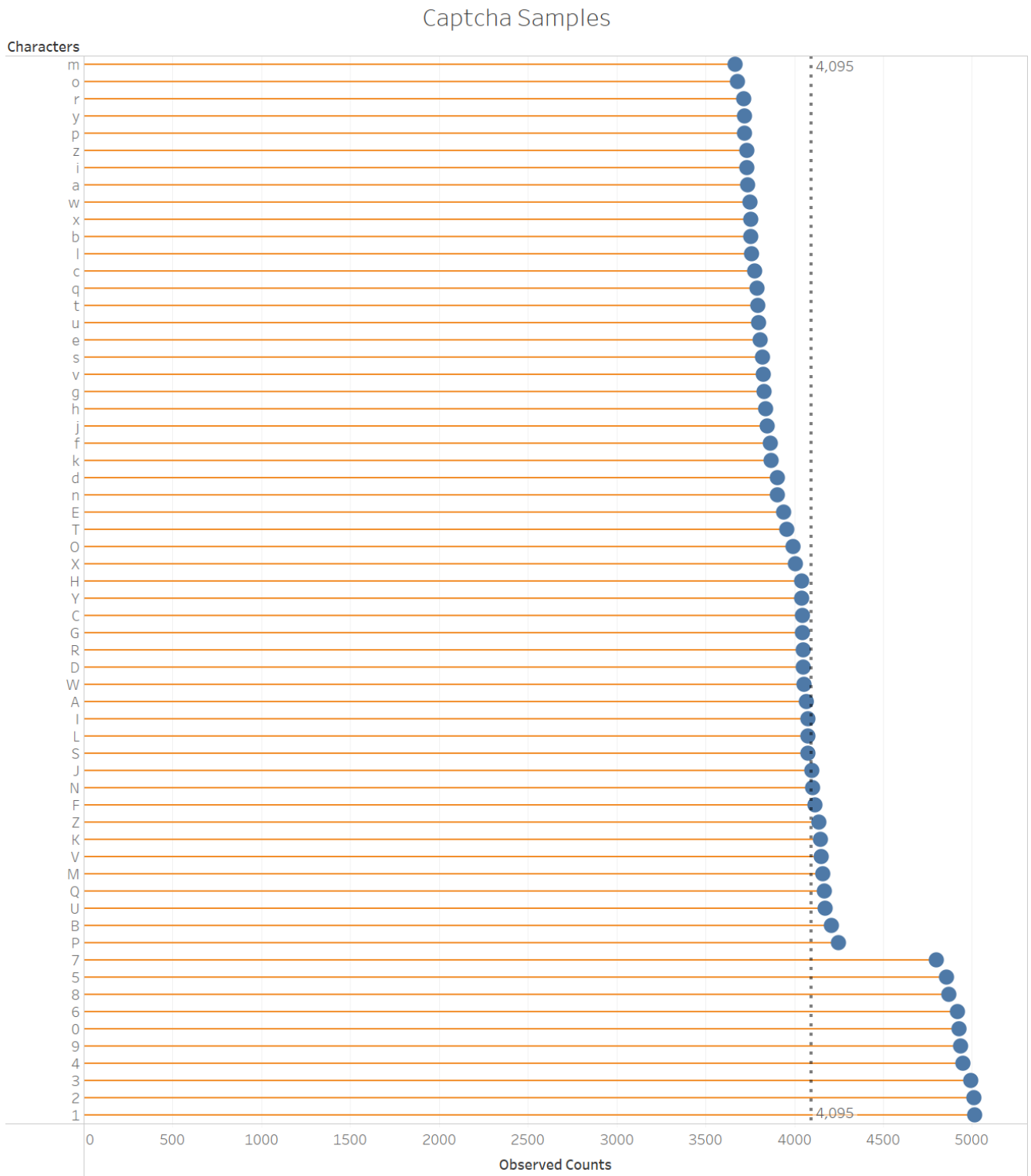


Figure 4: Distribution of characters in dataset. Notice that the overall dataset is not uniform distributed. The observed counts for letters are significantly less than digits. This is because the lowercase letters are not rendered for the black and white class, uppercase letters are not rendered for the noiseless class, while digits are rendered for all classes.

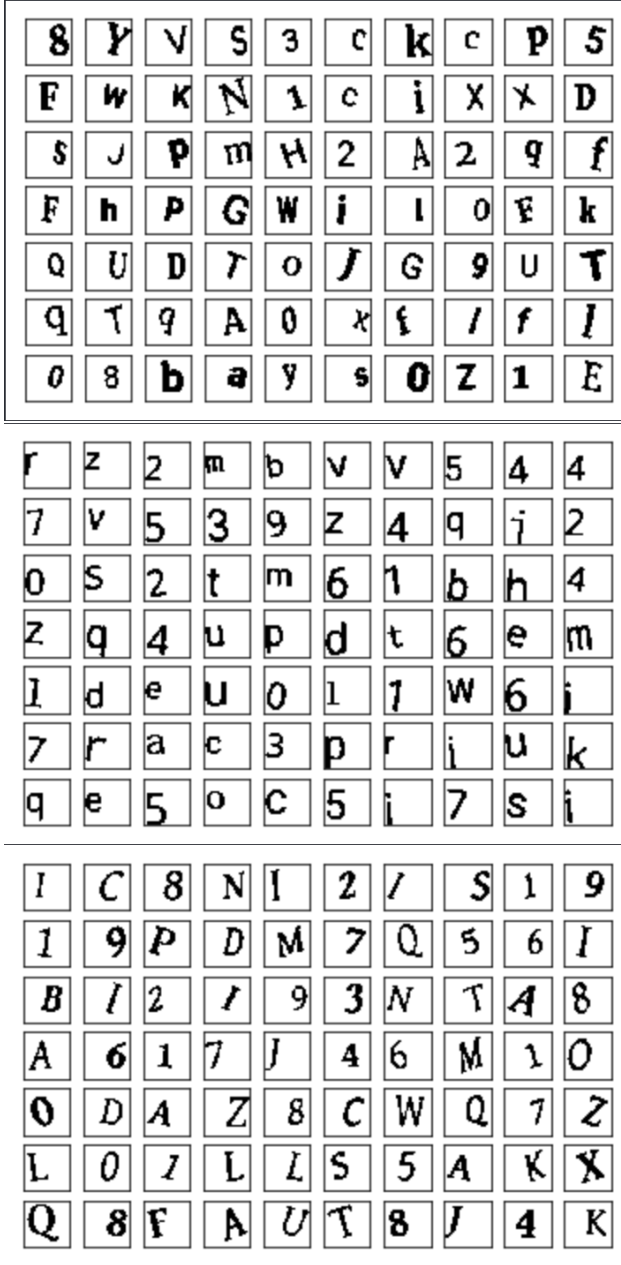


Figure 5: Color, noiseless and BW class characters for PCA and t-SNE



Figure 6: Principal components of the noiseless and BW sets

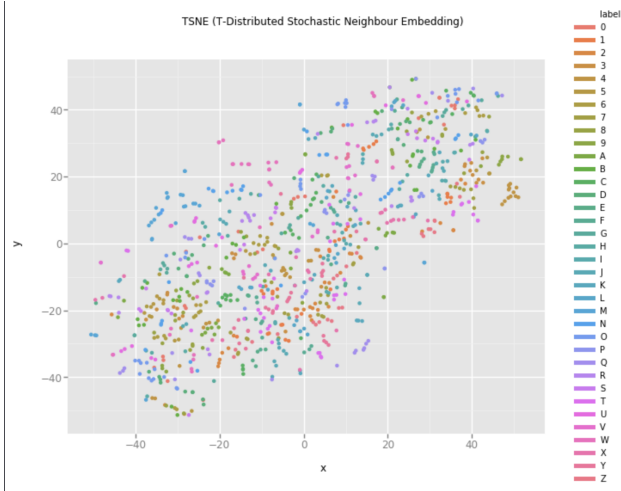


Figure 7: t-SNE plots of the color and BW sets

and pattern backgrounds, making an algorithmic background classification difficult.

- **Upsampling.** Each pixel of the image is divided into subpixels, thus allowing finer control over the area that is affected by the segmentation algorithm. While upsampling might be useful for complex images like detailed photographs, less complex images like ours do not require it.
- **Thresholding.** Removes pixels of low intensity (they are treated as noise), resulting in a binarized image. This is useful since most segmentation algorithms require a binarised input image. We eventually use an augmented form of thresholding in the final preprocessing algorithm.
- **Line removal.** Eliminate straight segments that are not part of characters. This is particularly hard as our dataset has arc and line occlusion.

Our objective is to segment text from the background and attempt to split it into single character blocks. The goal of segmentation is to identify the location of the characters and extract them from the rest of the image. Published attacks use techniques that are specific to a particular CAPTCHA scheme. We attempted flood-filling segmentation, where an object is taken as a connected component created from neighbouring black pixels. Flood-filling segmentation takes a binarized image and returns the collection of objects found. However, this nearest-neighbour method doesn't account for noise and compression artifacts. We found that some in-the-wild CAPTCHA sets contain JPEG artifacts, making flood-fill a poor choice as a boundary detector.

3.2 Otsu Thresholding

We choose Otsu thresholding [8] as it provides a robust binarization result, only occasionally letting noise through the filter. Consider a bimodal image, an image whose histogram has two peaks. For that image, Otsu thresholding computes a threshold value in the between the value of those peaks. Otsu's algorithm tries to find a threshold value t which minimizes the weighted within-class variance given by the relation:

$$\sigma_w^2 = q_1(t) \cdot \sigma_1^2(t) + q_2(t) \cdot \sigma_2^2(t) \quad (1)$$

where $q_i(t)$ is the cumulative sum of probabilities, and σ_i^2 is the variance. We first apply a 3x3 Gaussian blur to denoise the input, subsequently running Otsu thresholding on the filtered image. This results in a slightly better segmentation, as seen in Fig. 8.

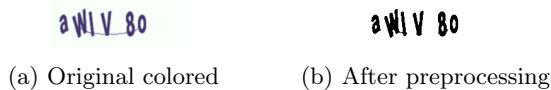


Figure 8: Pre- and post-Otsu thresholding

All images are binarized according to the results of the segmentation phase mentioned above. A classifier is then used to recognise each of the characters. While early attacks against CAPTCHAs were using simple criteria such as the pixel count of the resulted objects, more recent attacks utilise machine learning classifiers that are trained on the output of the segmentation algorithm.

4 Model Architecture

We refer the state-of-the-art in object detection, using the RetinaNet [6] meta-architecture for our classification and localization task. Typically, one-stage detectors like YOLO [9] and SSD [7] demonstrate fast inference with low accuracy. We experiment casually with them, but find that RetinaNet is a faster, one-stage detector with comparable accuracy to slower two-stage detectors, like Feature Pyramid Networks (FPN) [5] and variants of Faster-RCNN [10]. Here, we have two objectives, the classification of anchor boxes in the anchor set A , as one of 62 classes, and the regression from anchor boxes to ground-truth object boxes. We include the use of focal loss [6] to reduce the contribution of easy classification examples during training and focus the model on hard examples:

$$FL(p_t) = -(1 - p_t)^\gamma \log(p_t) \quad (2)$$

4.1 Backbone

The one-stage RetinaNet uses a CNN-FPN backbone, and two heads for classification and regression. The backbone takes the job of computing the convolutional feature maps over the input image. We theorize that we can build a better model by going deep, since we are able to control the size of the dataset easily. We choose ResNet50 [2] and augment it with a FPN, which creates a multi-scale feature pyramid instead of individual feature maps. This allows each layer to access different feature maps to detect objects at various scales.

4.2 Classification

The classification head predicts the probability of an alphanumeric character in the image, for each anchor $\in A$ and character class. The head is constructed with four 3x3 convolution layers with ReLU activations, followed by a final 3x3 convolution layer with $(classes \times A)$ filters, with a sigmoid activation for computing the probability score. The classification task on uses focal loss.

4.3 Regression

The regression head regresses the offset from each anchor box to the nearest ground truth object. The convolutional layers in the regression head are identical to the classification head, but the sub-network instead predicts the relative offset between the anchor and the true box. While the architecture is similar to the classification head, the parameters are kept separate. The regression task uses smooth L1 loss.

4.4 Training

We train with Keras, using the Adam [4] optimizer for 32k iterations with a batch size of 40, and image augmentations including small amounts of rotation, translation, shear, and scaling. Flipping is not used for obvious reasons.

5 Evaluation Results

We refer to the metrics discussed in Section 1.2. Our results are good on the testing set, reaching a mAP of 0.9624 after training for a total of 32k iterations. Based on Fig. 9, we perform well on the test set on a per-character basis. However, in order to actually break the CAPTCHA, we have to successfully predict all the characters in the image. On average, for a 6-character CAPTCHA, this would mean that we only score $mAP^{len(captcha)} = 0.9624^6 = 0.7945$. Running the tests visually on test samples, as in Figs. 10-12, we find that our predictions are extremely accurate. We then run inference on the unseen set, with an example shown in Fig. 13, to surprising results. Given these results, we are confident that with additional training on even more variants of text-based CAPTCHAs, this network will maintain its high mAP on test sets, while becoming even more robust to unseen data.

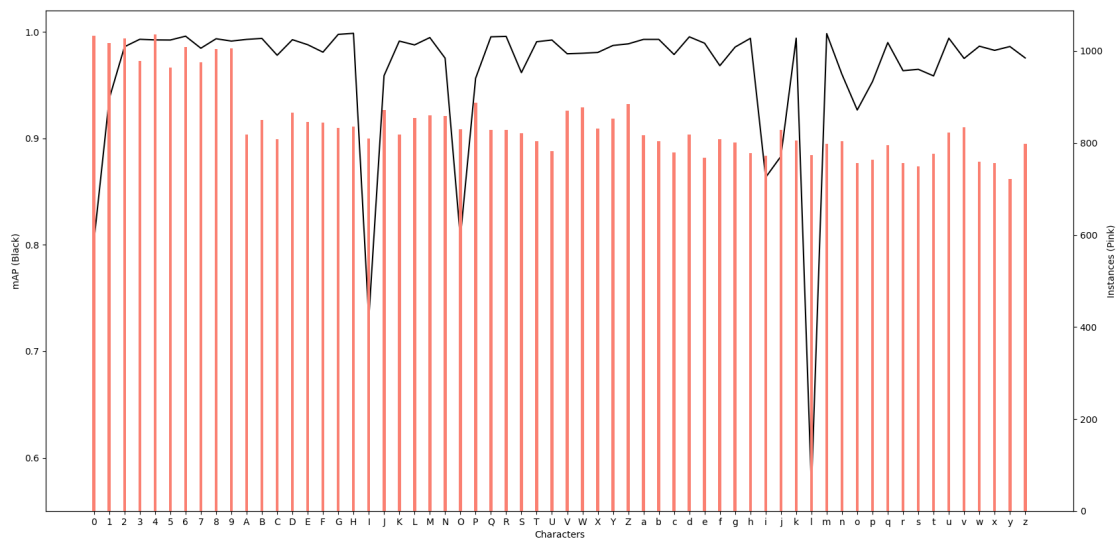


Figure 9: Results: mAP (black) and number test instances (pink) per character

```
processing time: 0.9927628040313721
3215_NkmHE.png [ 7.0694165 16.539825 34.918415 56.94571 ] 0.94628155 N
3215_NkmHE.png [42.829685 18.25738 63.0657 55.682297] 0.93504393 k
3215_NkmHE.png [ 74.97834 23.339264 104.62921 51.766293] 0.9279023 m
3215_NkmHE.png [112.32714 20.696232 135.26138 53.103733] 0.9107073 H
3215_NkmHE.png [144.27934 18.56451 166.302 55.27738] 0.89972794 E
```



Figure 10: A sample test result on the color set

```

processing time: 0.990349292755127
7028_jscfx0.png [71.56191 16.691833 96.61676 45.669228] 0.9665912 x
7028_jscfx0.png [32.815056 18.961489 56.590324 47.27707 ] 0.96031666 c
7028_jscfx0.png [54.833786 18.490793 74.0752 56.331753] 0.9540623 f
7028_jscfx0.png [14.531245 18.66003 37.0689 47.154495] 0.9128416 s
7028_jscfx0.png [ 3.900511 17.321436 16.856398 65.80125 ] 0.81445414 j
7028_jscfx0.png [ 94.613235 15.100686 121.926796 53.058437] 0.73534524 0
7028_jscfx0.png [ 95.67477 15.928685 121.65867 53.16189 ] 0.58240646 0

```



Figure 11: A sample test result on the noiseless set

```

processing time: 0.9923574924468994
228_0U8Bu.png [124.54958 24.29323 140.88493 51.235085] 0.9273528 u
228_0U8Bu.png [102.61326 20.232014 118.28107 54.24128 ] 0.8309154 B
228_0U8Bu.png [ 93.497345 20.227802 110.482 54.123363] 0.79476 8
228_0U8Bu.png [52.108143 17.708008 76.23358 55.38788 ] 0.7071569 b
228_0U8Bu.png [52.70465 17.71431 76.40289 55.34097] 0.64769226 U
228_0U8Bu.png [101.04989 20.560947 118.8774 54.045525] 0.5516697 8

```

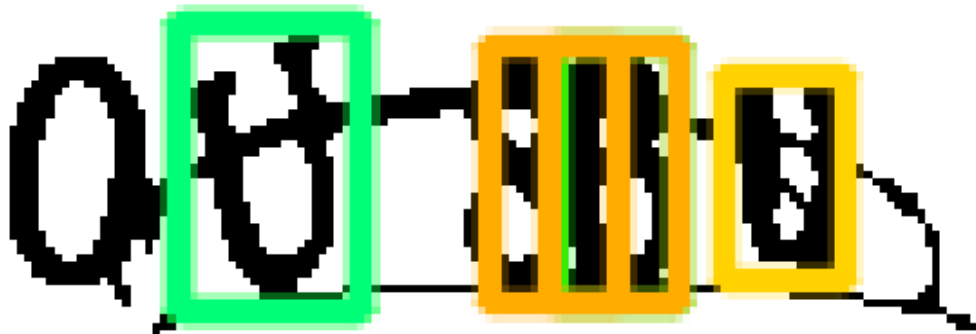


Figure 12: Failure case on the color set, the only set with noise. We are unable to produce a failure case example for the noiseless and BW sets.

```
processing time: 0.9941325187683105
854 [111.91172 22.89892 136.64429 58.312904] 0.968758 d
854 [ 0.6845457 23.548079 27.165869 56.476772 ] 0.8592202 3
854 [25.287416 26.798237 50.620438 63.643982] 0.80645734 q
854 [67.4503 24.552017 91.37746 70.68896 ] 0.744942 b
854 [ 0.85412043 24.150597 26.741817 56.48474 ] 0.6345513 s
854 [ 89.675415 37.658764 115.912384 67.861244] 0.53666806 e
```

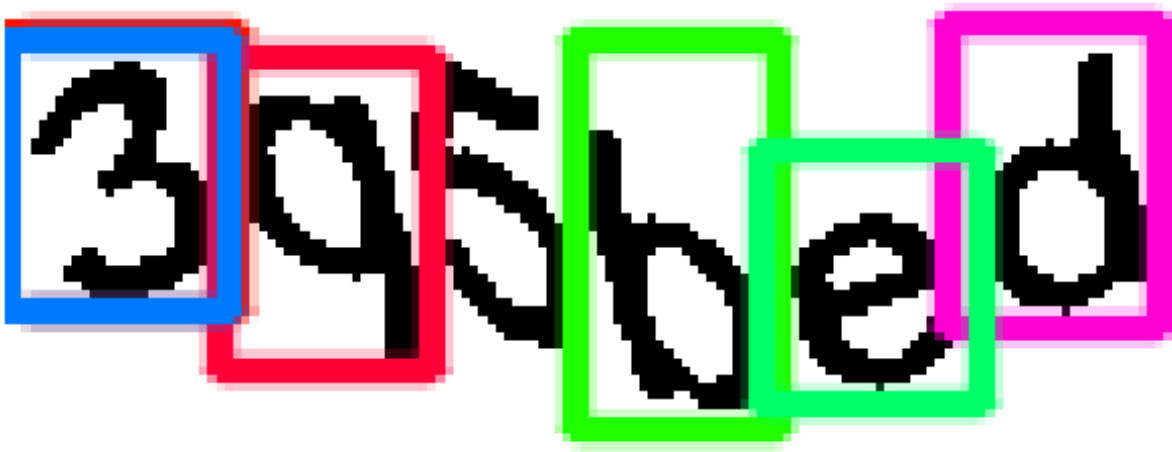


Figure 13: A test on the unseen set. This CAPTCHA type has never been seen by our model during training. Of the 6 characters in this string, 3q5bed, we are able to correctly predict 5 characters: 3, q, b, e, and d.

6 Conclusion

Websites must stop using alphanumeric CAPTCHAs as a first line of defence against automated attacks. We show that a simple method to solve alphanumeric CAPTCHAs, even on unseen data, is available to anyone with basic knowledge of image processing and machine learning. It is time to phase out alphanumeric text-based captchas once and for all.

References

- [1] R. Fortune, G. Luu, and P. McMahon. Cracking captchas.
- [2] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.
- [3] M. Jaderberg, K. Simonyan, A. Vedaldi, and A. Zisserman. Reading text in the wild with convolutional neural networks. *CoRR*, abs/1412.1842, 2014.
- [4] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.
- [5] T. Lin, P. Dollár, R. B. Girshick, K. He, B. Hariharan, and S. J. Belongie. Feature pyramid networks for object detection. *CoRR*, abs/1612.03144, 2016.
- [6] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár. Focal loss for dense object detection. *IEEE transactions on pattern analysis and machine intelligence*, 2018.
- [7] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. E. Reed, C. Fu, and A. C. Berg. SSD: single shot multibox detector. *CoRR*, abs/1512.02325, 2015.
- [8] N. Otsu. A threshold selection method from gray-level histograms. *IEEE transactions on systems, man, and cybernetics*, 9(1):62–66, 1979.
- [9] J. Redmon and A. Farhadi. Yolov3: An incremental improvement. *arXiv*, 2018.
- [10] S. Ren, K. He, R. B. Girshick, and J. Sun. Faster R-CNN: towards real-time object detection with region proposal networks. *CoRR*, abs/1506.01497, 2015.
- [11] F. Stark, C. Hazırbaş, R. Triebel, and D. Cremers. Captcha recognition with active deep learning. In *GCPR Workshop on New Challenges in Neural Computation*, volume 10, 2015.
- [12] R. Z. Tuan Anh Le, Atilim Gıneş Baydin and F. Wood. Using synthetic data to train neural networks is model-based reasoning. *Neural Networks (IJCNN) 2017 International Joint Conference*, pages 3514–3521, 2017.
- [13] N. Zhao, Y. Liu, and Y. Jiang. Captcha breaking with deep learning.