unCAPTCHA: AI Recognition of Distorted Text in Images

April 11th, 2021

Contents

Abstract

With the growth of web security, many methods to block web traffic are needed to keep a web business growing. CAPTCHA has become an important part of web security, since it helps prevent machines (bots) to spam forum posts and shops from overflowing traffic, if the bots are successful to push that limit, the website will lose business, and users. Though after careful analysis, wondering over the future of both machine learning and web security becomes a bit worrisome. Since, people can easily make machine learning possible from its open-source Python modules, like TensorFlow and PyTorch. By seeing if it's possible to *unCAPTCHA* and having the machine possibly bypass it will be worrisome to web business and web activity.

Introduction

Completely Automated Public Turing test to tell Computers and Humans Apart (CAPTCHA) is a simple test to distinguish between both humans and machines. It is used to help websites not create a major traffic load from the bots posting messages, video, audio, and other forms of media. In today's technology, a boost in machine learning (ML) has become popular. By testing out how efficient the use of CAPTCHA in websites is, ML will be needed to see if CAPTCHA does detect any attempts from a bot. There are different types of CAPTCHAs, from image-based to web cookies. To make it simple, CAPTCHA's grayscale distorted text images are used and with it, see if the machine can read and write the corresponding characters; decoding the distorted text. If the machine can at least identify most of the characters in the image, then creating a bot with this machine can not only overflow the amount of traffic to that website but also the businesses that are running that website. This could affect a small portion of websites that are easy to decode for the bot.

Research

Preprocessing

In any ML process, it is expected to create, retrieve, and use the data to make the ML algorithm work as intended. With distorted text, it can be easily generated by using FakeCaptcha.com, a free user-generated CAPTCHA like distorted text. By generating these fake CAPTCHAs, it is needed to create a program to extract each character needed for the ML algorithm to learn and recognize the characters given.

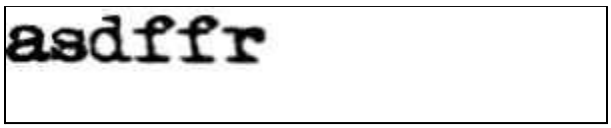
Thresholding

First, the program must be able to see clearly which pixels will be the main letters or numbers and which is the background. *Thresholding* can help create this much clearer coding for each pixel in the distorted text image. Thresholding is used in grayscale images to check if the pixel value is greater than a threshold value, assigning it to one value (maybe white), else it is assigned another value (maybe black)[1]. The program can

use different thresholding types, but the one chosen for the algorithm is the Adaptive Gaussian Threshold. The difference between adaptive and standard thresholding is that it can ignore shadows or light. Creating a much-fine-tuned image for the algorithm to learn. Originally, attempts of using more fixed thresholding on half the maximum gray level were made. However, after experimenting with different max and min value sets for each pixel, it is best to move forward with the Adaptive Gaussian Threshold, both easy to use and less time-consuming. When running the program, there are often characters being merged having the extracting processes difficult.

By using cv2, a Python module used to change image appearance using different filters, thresholding can be easily done using the cv2 module. Here is an example of using thresholding on distorted text:

Original image



Added thresholding

In here, the thresholding on ‘asdffr,’ made the image clearer and helpful for creating the connected components to find multiple character clusters.

Connected Components Labeling

When the thresholding is done, a cluster of the different characters from the generated CAPTCHA will be formed. It is needed to understand which cluster has either just one character or is joined with two or (rarely) three characters. To observe what characters are a joint cluster, we will use Connected Component Labeling (CCL), an algorithm used to help in labeling disjoint components of an image with unique labels[2]. When using CCL, we will expect each cluster to be having a colored label.

Image after CCL

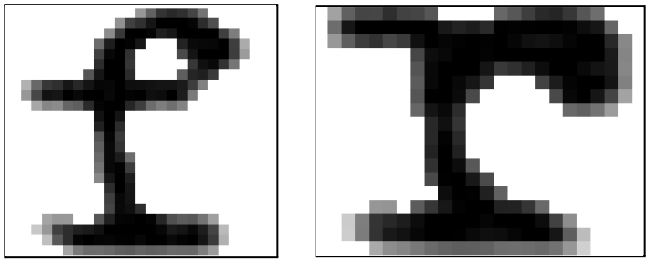
Above is an example of threshold clusters being labeled with colors. This helps the human better classify what has joined clusters or not. This will also help the algorithm best find the connected component and will split the joint characters into one single

character. By understanding what clusters the thresholding could find, the use of splitting joints of two characters will be helpful later on.

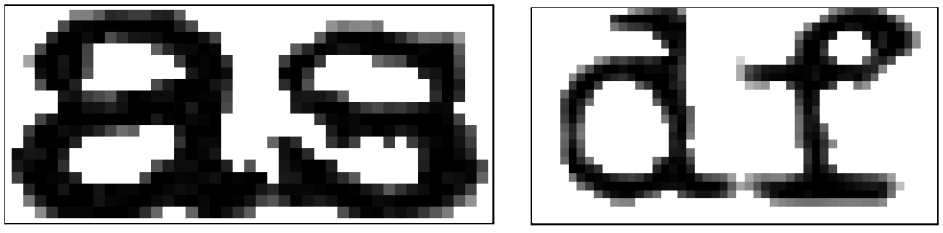
Extracting Single Characters

After understanding where each cluster is located, the process of extracting each of the clusters will be easier and proficient. To extract each cluster, the algorithm will detect the location of each cluster that has been labeled. Each of those pixels in that cluster will have their x and y position saved and will create a small image that the algorithm will create with extra white space for the user to observe what characters are extracted. With the locations of the x and y, it will be used to detect any type of patterns the generated CAPTCHA has made. Then the extraction will begin simultaneously within each character in order.

Back to the example used in both Thresholding and CCL, the extraction process went well. Both the ‘f’ and ‘r’ characters in the generated CAPTCHA are shown to be great, there are no other connected characters in one cluster and are shown visibly.



On the other hand, the other four characters are shown to have two characters joined together. In this case from the examples used, ‘as’ and ‘df’ are joined together. If ever there is one small pixel connected between two characters, in the case of ‘df’, it is shown as still one cluster.

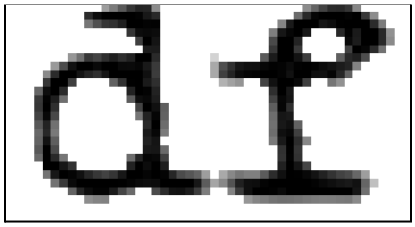


Data Conditioning

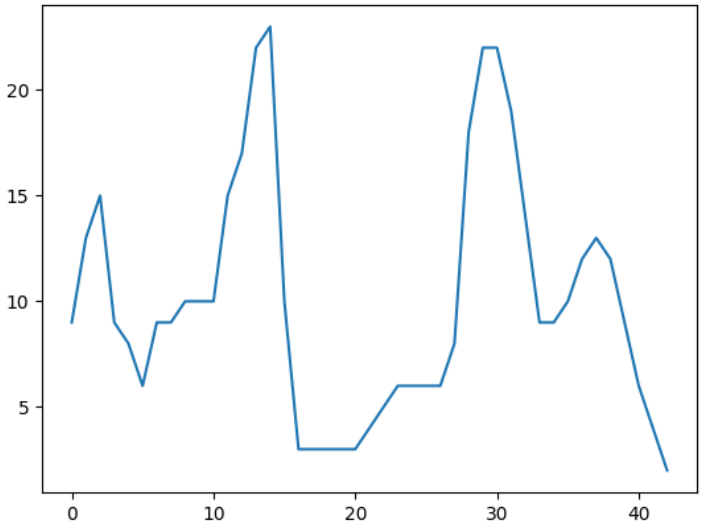
From the previous step, some contamination from a different character is expected because of the distortion. This contamination is found within our extraction of clusters. Occasionally, double characters will occur because of imperfect image segmentation. By using a histogram in the algorithm, it will detect a very wide postage stamp (extracted clusters) and will split them near the middle where the amount of print is minimum, finding the minimum numbers of black pixels in each column.

Below is the histogram used to find the minimum amount of black pixels with the connected characters. This will help the algorithm find what will be the best location to split that connection. In this case, using the 'df' cluster, the connected pixels in the very center have minimal black pixels.

Original cluster



Amount of Black Pixels in the 'df' Cluster



The algorithm will automatically cut the cluster in half to now retrieve the single character needed to add to our data set.

Now that the data is retrieved, the algorithm will set a specific size that our ML algorithm will need to learn. 28x28 is a good geometric ratio since it is expected for our neural network to learn. By using two-dimensional interpolation, the algorithm can create a much clearer image no matter the size. In images, two-dimensional interpolation uses values of only the 4 nearest pixels, located in diagonal directions from a given pixel, to find the appropriate color intensity values of that pixel[3]. Having the resizing less compressed and helpful for the neural network to work. The algorithm will also invert the image from black to white and vice versa. This will better help the neural network in learning with the MNIST dataset, in which the character is white and the background black.

Character Recognition

After successfully retrieving the data needed for the neural network, creating the ML algorithm will not be too much of a hassle since again the algorithm has made extracting characters very easy. With the use of Keras (a TensorFlow library), creating the algorithm for deep learning will be easier and more efficient.

Convolutional Neural Network (CNN)

CNN is a deep learning algorithm that can take in an input image, assign importance to various aspects/objects in the image and be able to differentiate one from the other[4]. This means that the network learns to optimize the filters (or kernels) through automated learning, whereas in traditional algorithms these filters are hand-engineered[5].

By using Keras, an API that uses TensorFlow, implementing CNN will be easier and efficient. When presented with a single character's postage stamp image, the model will output a vector of probabilities for each possible character 0-9. The character with the highest probability is the predicted character.

```
model = keras.Sequential(
    [
        keras.Input(shape=input_shape),
        layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Flatten(),
        layers.Dropout(0.5),
        layers.Dense(num_classes, activation="softmax"),
    ]
)

model.summary()
```

The deep CNN model presented above consists of the following layers:

- Input layer to accept training and test data (postage stamp images of single digits)
- 2D convolutional layer with 32 kernels, 3x3 pixels each to be learned with Rectified Linear Unit (ReLU) activation function. ReLU is a piecewise linear function that will output the input directly if it is positive, otherwise, it will output zero[6].
- 2D max-pooling layer to block average the output from the previous layer in areas 2x2 pixels.
- Another 2D convolutional layer with 64 kernels, 3x3 pixels each to be learned with RELU activation function.
- Another 2D max-pooling layer to block average the output from the previous layer in areas 2x2 pixels.
- Flattening step to go from 2D to 1D output.
- A dropout layer that simply forgets randomly selected half of all inputs.

- A dense layer to go from remaining inputs to a softmax probability estimate for each possible output.

Training

As mentioned before, the Modified National Institute of Standards and Technology (MNIST) dataset is a large database of handwritten digits (0-9) containing 60,000 training images and 10,000 testing images[6]. Using the MNIST data set, training the algorithm will be more efficient.

```
model.compile(loss="categorical_crossentropy", optimizer="adam",
metrics=["accuracy"])

model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs,
validation_split=0.1)

model.save('mnist_convnet.model')
```

To train the To minimize a loss function (compute the quantity that a model should seek to minimize during training) the algorithm will be using categorical cross-entropy, measuring the difference between two probability distributions for a given random variable or set of events[7]. Using this, the algorithm can best predict class membership. By using the Adam optimizer we can get similar results like the stochastic gradient descent, but it is much more sophisticated and has adaptive features to improve efficiency, convergence speed, etc. To track the progress, implementing a classification accuracy metric (the fraction of correct classification predictions) to the algorithm, will be more intuitive to humans than using cross-entropy. With this, saving the trained model will set files in a folder used for later without needing to train the algorithm again.

Testing and Validation

By splitting the MNIST dataset into a training set and testing set, the algorithm will better recognize the characters within the algorithm's testing data, helping determine the accuracy; this is a standard procedure. Then we will evaluate the accuracy, calling back to our saved directory from our training algorithm.

```
score = model.evaluate(x_test, y_test, verbose=0)
print("Test loss:", score[0])
print("Test accuracy:", score[1])

y_class = model.predict_classes(x_test[:10])
y_pred  = model.predict(x_test[:10])

for yc, yp, yt in zip(y_class, y_pred, y_test):
    print(yc, 10*' %.3f' % tuple(yp), yt)

for i in range(10):
    stamp = x_test[i,:,:]
```

```
    plt.imshow(stamp)
    plt.show()
```

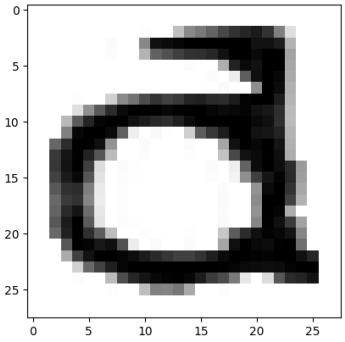
When running the algorithm, the accuracy was very high, around 98%. The input data was not handwriting data, using it as a proxy for distorted prints. It is not the same, attempting with 0 domain adaptation from handwriting to distorted print. Expecting a lower accuracy rate.

Complete unCAPTCHA

Workflow: From Images to Text

Now, the algorithm will be doing the process of identifying and connecting other datasets to predict what digit from the MNIST data set will present from the extracted stamp. The point of all this laborious preprocessing was to get to this place where the algorithm will finally use the neural net to recognize single characters

Showing the Cluster with Predicted Result




```
PREDICTION: [3]
```

Demonstration

With the MNIST dataset, the algorithm must limit the scope, to handle five digits because splitting doubles may have created more than 5. While developing the algorithm, albeit using various examples of FakeCaptcha, with its use of graphics and text. The final script does not do that. The algorithm will only show the main results.

```
from tensorflow import keras
model = keras.models.load_model('mnist_convnet.model')
digits = '0123456789'

captcha_prediction = ''

for stamp_img, x_location in clean_stamps:
    # Resize and rescale gray levels so that we can present
    # the data to the deep neural net model as expected.
    resized_stamp_img = rescale_stamp(stamp_img)
    print('X location/size:', x_location, stamp_img.shape,
```

```
resized_stamp_img.shape)
    inp_data = np.expand_dims(resized_stamp_img, -1)
    out_data = model.predict(np.array([inp_data]))
    out_class = model.predict_classes(np.array([inp_data]))

    # output the prediction and show the cluster that
    # the algorithm has predicted.
    print('PREDICTION:', out_class, out_data)
    plt.imshow(1.-resized_stamp_img, cmap='gray')
    plt.show()

    # Assemble captcha text one character at a time
    captcha_prediction += str(out_class[0])
```

Though using the generated FakeCaptcha will be needed to recognize the said character. MNIST is extremely important for this algorithm since it is the only dataset used to help the neural network to predict and find better patterns for the stamps shown.

Results

Accuracy

The accuracy of the MNIST dataset was 98%, but the use of the FakeCaptcha was much lower than MNIST. Since it's handwriting and not print. But when running the algorithm, the print was similar to handwriting, which worked very well. Yet many examples did not work. Such as the prediction given in the *Workflow: From Images to Text*, where the prediction is odd from limited patterns given, the general shape of that 'd' cluster, has made the algorithm recognize it as a '3'. A possible reason is the distortion of the letter to resize it to its geometric size of 28x28.

Failure Modes

There are many factors in why the prediction of these clusters is shown to be unreliable. The print was thick in comparison to handwriting. Few examples have a font that is 'hollowed' out, with white pixels inside the black characters, having the algorithm split single digits that are not needed to happen. Segmentation fails within the algorithm, not having enough characters to return. Oftentimes, when generating the CAPTCHA, adding thresholding, and/or resizing the geometry of that cluster will create too much distortion. And lastly, again, when generating a CAPTCHA, the print will be too small for the algorithm to detect and cut each cluster.

Conclusion

From what the algorithm has created, split, learn, and train. It *is* possible to unCAPTCHA a certain print from the text made from using CAPTCHA. Though again, there are a few problems where the algorithm can only learn digits, split specific font types, and have a good dataset to have the neural network learn more proficient and effective. Though

results are shown, there is much needed to change to better improve the algorithm to better recognize each character, font, and distortion level.

Future Changes

As mentioned before, having the algorithm retain adequate training data will better improve its recognition rate from just patterns and distortion levels. Improve image segmentation, having the algorithm better recognize the different patterns and shapes. Though the amount the MNIST dataset has is good. Yet, having more data within the algorithm will improve results effectively. Also, experimenting on modifications within the neural network model will further improve the results. And lastly, obviously, is to extend the training data to handle all alphanumeric values, not just digits.

References

[1] https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_thresholding/py_thresholding.html#simple-thresholding
[2] https://en.wikipedia.org/wiki/Connected-component_labeling
[3] https://en.wikipedia.org/wiki/Bilinear_interpolation#Application_in_image_processing
[4] https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53
[5] https://en.wikipedia.org/wiki/Convolutional_neural_network
[6] https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/
[7] https://machinelearningmastery.com/cross-entropy-for-machine-learning/