CS 5100

8/12/25

Breaking CAPTCHA using Convolutional Recurrent Neural Networks

One of the internet's first AI benchmarks vs. a modern AI technique

Emilio Sanchez-Harris
sanchezharris.e@northeastern.edu

Introduction

CAPTCHA (Completely Automated Public Turing test to tell Computers and Humans Apart) systems first began to emerge back in 1997, when the search engine AltaVista found they had an issue with automated URL submissions. At the time, the search engine needed to maintain their "add URL" feature which helped broaden search coverage, however large numbers of URLs were being submitted in a clear attempt to skew its algorithm. To combat this rising tide of obvious bots, Andrei Broder[1] and his colleagues developed a system of generating random sequences of distorted alphanumeric characters which computer vision systems could not read, but humans could - within a year it had reduced the number of botted submissions by 95%.

Today, platforms like Twitter and Facebook are overrun by bots, indicating that CAPTCHA solutions like the ones Broder and his colleagues created, along with those that have followed in their footsteps, are not adequate in the face of todays modern AI techniques. In undertaking this project I aim to illustrate a modern approach used to break CAPTCHA systems, as well as hopefully highlight areas where this approach falls short, in the hopes of getting a better idea of what a better CAPTCHA system might comprise of.

Background

The approach used in this project comes from a 2016 paper by Baoguang Shi and their colleagues titled "An End-to-End Trainable Neural Network for Image-based Sequence Recognition and Its Application to Scene Text Recognition"[2] which is where the CRNN architecture was introduced. The technique achieves over 95% accuracy when applied to the SVT dataset, a dataset of images with text that has been distorted via blur, noise, and low resolution.

The novelty of the approach is that it creates pipeline made up of a number of machine learning techniques, namely Convolutional Neural Networks (CNN), Recurrent Neural Networks (RNN), and Bidirectional Long Short Term Memory (BiLSTM). We also utilize Connectionist Temporal Classification (CTC) in our implementation.

Convolutional Neural Networks extract spatial features from images through hierarchical learning. By passing an image through multiple convolutional layers, specific features like edges and curves are identified. Using pooling layers, we reduce the resolution of the feature map we've detected to make our system more robust to variations in these features as they arise. After passing images through these multiple layers the CNN learns increasingly complex and abstract features which make it an effective classifier. In this context CNN is used identify character patterns despite distortions arising from low resolution and blurring.

Recurrent Neural Networks processes sequential data and makes predictions as to what comes next by maintaining an internal memory state, which is where LSTM is utilized. By using bi-directional LSTM we treat the width of the feature map as a temporal sequence, allowing the model to reason about character order in both directions using context. This is important in CAPTCHA recognition because neighboring characters provide important cues for reasoning about each character.

The Connectionist Temporal Classification loss function, introduced by Alex Graves in 2006[3] is essential for this problem because it gives us a means of solving alignment problems that arise in sequence recognition. It uses a special "blank" character to represent potential gaps in the sequence and marginalizes over all possible alignments between the prediction and ground truth. This makes it an ideal loss function for CAPTCHAs considering that they are often intentionally spaced irregularly.

The combination of these techniques creates a system that is able to achieve end-to-end training on whole captcha images, with impressive results.

Methodology

My implementation utilizes the following frameworks and libraries:

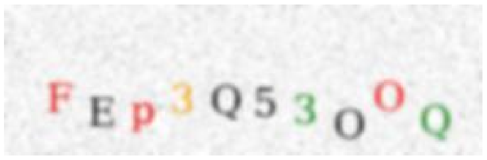
- PyTorch
- OpenCV
- Matplot
- Scikit-learn
- NumPy
- tqdm

My Captcha dataset comes from Kaggle and was uploaded by Aadhav Vignesh[4]. It is a library of 10,000 JPGs where the name of the file corresponds with the CAPTCHA solution.

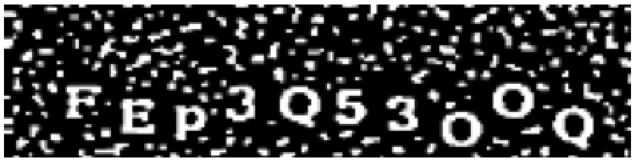
1. Preprocessing

I used OpenCV to apply a Gaussian blur, reducing noise in the images to as to avoid features being detected that aren't really there. I convert the image to black and white to reduce the complexity of the problem as well as use adaptive threshold to handle differing shades of white and black (pixel values of 0 and 1). Given the noise in the database I also use morphological operations to remove small bits of noise and fill any gaps in the characters. I also resized all the images to 200x50 so all values are easily divisible. Here is an example of an image before and after preprocessing:

Before



After



Then I split the dataset into 3 parts, 72% (80% of 90%) for training, 18%(20% of 90%) for hyperparameter tuning during training, and 10% for testing.

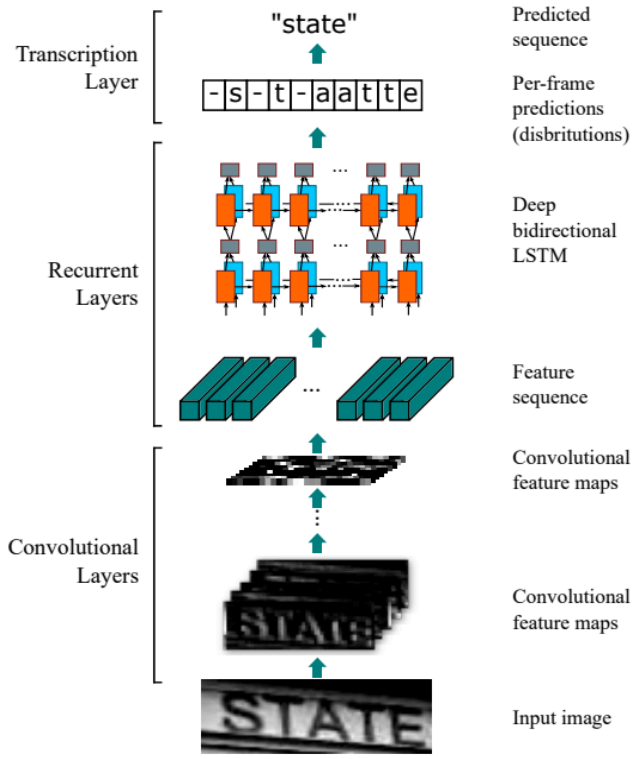
2. Character encoding

The encoder takes a set of characters, in this case all capital and lower case letters as well as digits 0-9, and creates a vocabulary of 36 characters encoded as integers, and adds the special CTC “blank” token at index 0 for a total of 37 unique integers.

3. CRNN

The CRNN implementation begins with a CNN. In my implementation the CNN component extract spacial features through 4 convolutional blocks, each doubling the amount of filters applied to go from detecting shapes to eventually detecting whole characters. After CNN processing the feature maps are transformed into a sequence format for RNN processing. This entails breaking up the image width into 25 time-steps each containing a total of 1536 features. The BiLTSM processes the feature sequence to capture spatial relationships and context that help segment and recognize individual characters to make a prediction. We include dropout to prevent overfitting. Finally, we permute our output for CTC training.

To better illustrate the implementation see the diagram[2] below.



4. Training

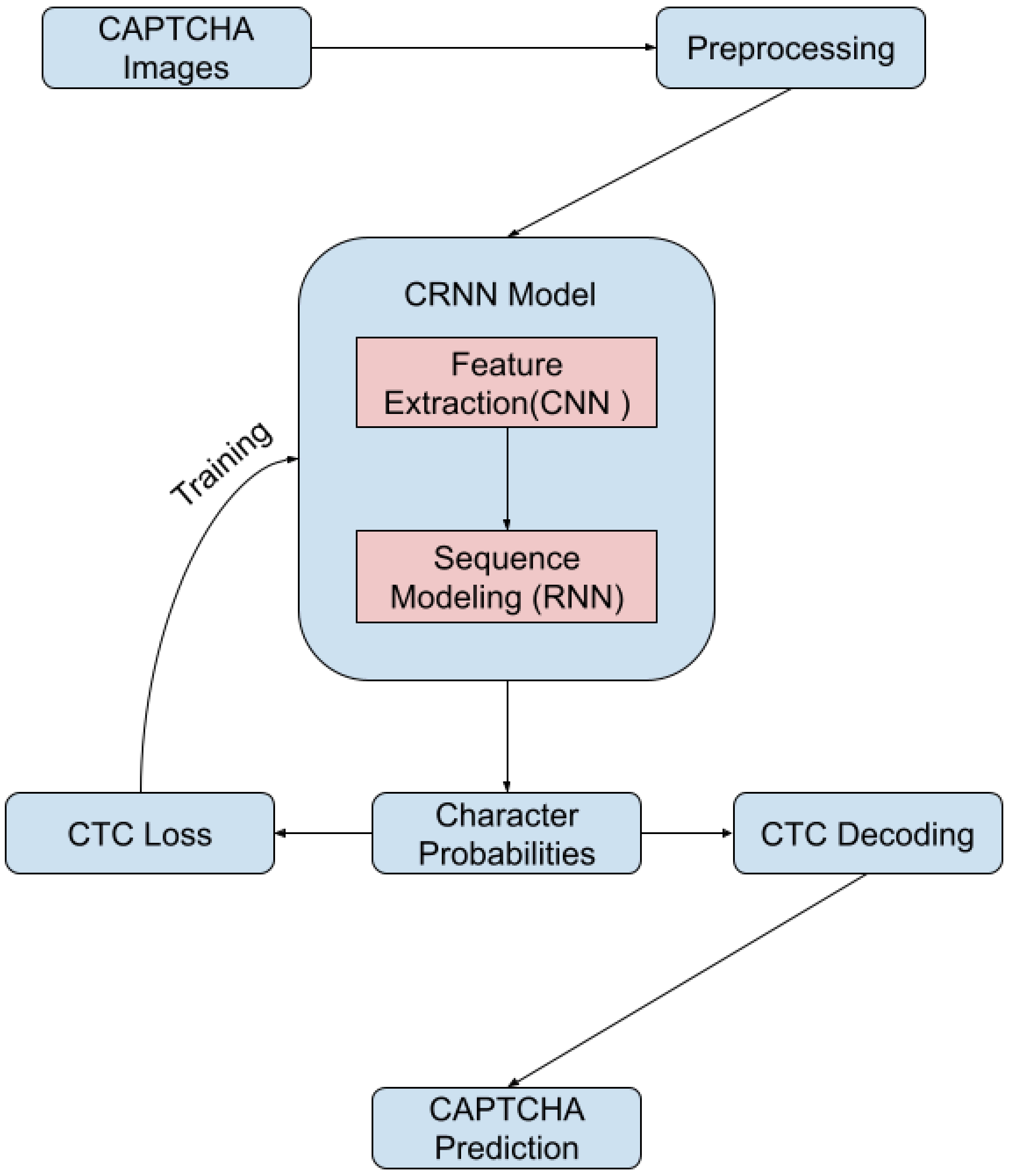
We start by creating batches for CTC training, permuting tensors where necessary. Our CTC loss function handles the alignment between the 25 time steps and 10 characters, returning a lower loss value as our system gives a higher probability of correct alignments, and visa versa. Although our CRNN model is composed of multiple components we can train it jointly with this single loss function.

To optimize the training process we utilize the adam optimizer to adjust learning rate at the parameter level, as well as the ReduceLROnPlateau() function to cut the learning rate in half if our validation accuracy begins to plateau. An early stopping condition is also included in case something goes wrong in the training process.

5. Evaluation

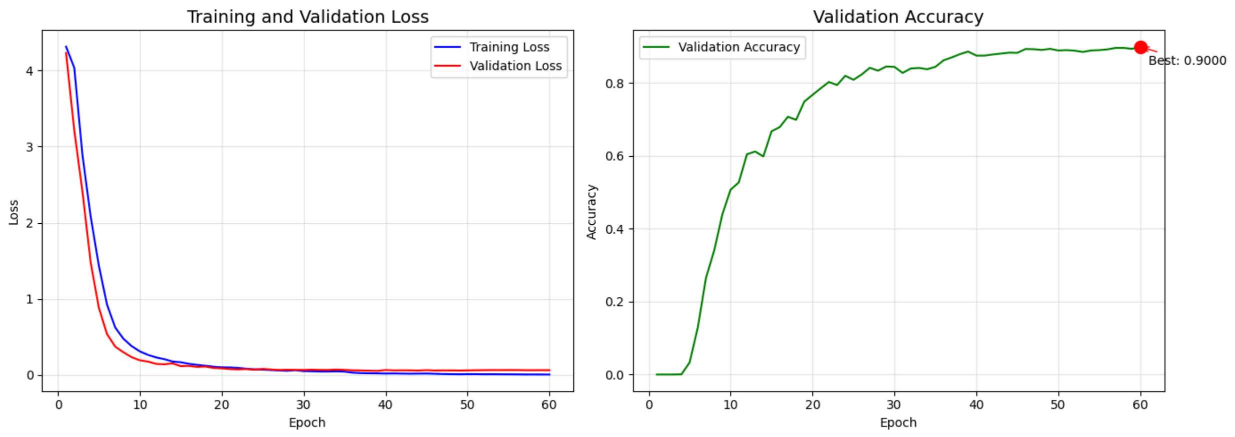
Finally we decode our systems output to text by removing the “blank” tokens, merging consecutive characters, and converting all indices back into their respective characters. We can then compare these prediction strings with their true values to get accuracy values for the whole CAPTCHA and at the character level.

The methodology can be summarized more succinctly with the following diagram:

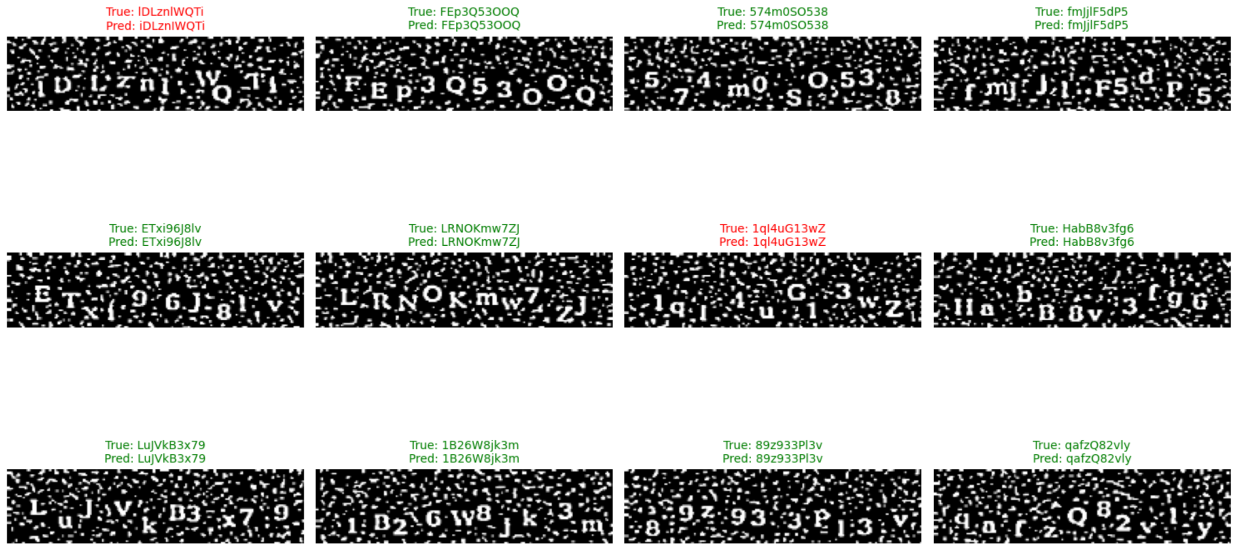


Results

The most impressive result from this project was the fact that the model achieved 90.00% accuracy at the whole-CAPTCA level on the final epoch of training. When we applied this model to our test set we achieved 88.00% accuracy at the whole-CAPTCHA level and 98.66% accuracy and the individual character level. We can see our improvements begin to plateau around epoch 45 around 88% percent accuracy.



Below is an example of a given input CAPTCHA and the model's prediction:



I also did some error analysis to see where most the model's mistake come from, you can find another copy of these results in the logs folder.

```
=== Error Analysis ===
Total errors: 120

Top 10 character confusions:
  I → l: 13 times
  I → 1: 13 times
  l → i: 6 times
  1 → l: 6 times
  l → I: 5 times
  o → e: 4 times
  G → C: 3 times
  J → j: 3 times
  1 → I: 3 times
  i → I: 3 times

Most error-prone characters (top 10):
  '1': appears in 53 errors (44.2% of errors)
  '7': appears in 44 errors (36.7% of errors)
  '4': appears in 41 errors (34.2% of errors)
  '0': appears in 41 errors (34.2% of errors)
  '5': appears in 40 errors (33.3% of errors)
  '9': appears in 40 errors (33.3% of errors)
  'I': appears in 37 errors (30.8% of errors)
  '8': appears in 37 errors (30.8% of errors)
  '3': appears in 36 errors (30.0% of errors)
  '2': appears in 33 errors (27.5% of errors)

Confidence analysis:
  High confidence errors (>0.8): 120 (100.0%)
  Low confidence errors (<0.5): 0 (0.0%)
  Average error confidence: 0.993
```

Discussion

Our model was able to successfully guess the correct captcha with 90% accuracy, which shows how little trouble it would have bypassing a CAPTCHA of this type if it were to be used “in the wild” so to speak. Even more impressive is its level of individual character recognition with almost 99% accuracy. This is a glowing endorsement of the CRNN model in regards to its ability to parse text in images, and leads me to believe it will be very effective in its application in other OCR and CV fields beside CAPTCHA detection. It is, however, a very concerning statistic if you happen to be a party, like AltaVist who want anything but to have their systems be so easily compromised.

When considering the limitations of this approach what stands out to me is the prevalence of characters like ‘I’ (uppercase i), ‘l’ (lowercase L), and ‘1’ (number one), as well as the fact our system was *very* confidently wrong when it came to making these mistakes. What also stood out was the sheer prevalence of numbers in our errors, in fact they make up 9 of the top 10 most error-prone characters.

This leads me to the conclusion that potential improvements could come in the form of more character-specific training, or perhaps creating a separate model that classifies characters into either alphabetical or numerical before classifying within those categories. On the flip side, these results lead me to believe that potential improvements to our CAPTCHA systems may include:

- Utilizing commonly confused characters with greater frequency, and with an intent to take advantage of these limitations (using ‘l’ and ‘1’ more often with results like the ones I got in mind)

- A variety of the kinds of visual information present. The prevalence of numbers gave me the idea that systems like these might struggle with telling apart different *types* of entities, so it might be beneficial to mix the kinds of visual information included in a CAPTCHA. For example, along with numbers which can be confused with common a-Z alphabetical characters it might be beneficial to include more varied characters like ↑, ⟳, ❕ or even mix in images with the hopes of “spreading thin” the model - as I believe that’s what the numbers served to do in my case.

Conclusion

Upon completing this project, I found the results very pleasing (particularly the fact that the model hit 90% accuracy exactly). The character level accuracy is an even more impressive statistic, sitting practically at 99%. I learned a great deal when working on this project including: common image pre-processing techniques and why they work, how to train my own models using the CRNN architecture, how CTC can be used in a text recognition context, and most importantly I learned that ingenuity, creativity, and taking risks are key parts of research and their respective breakthroughs. When reading these papers it was very inspiring to see how an idea can become a reality. I look forward to learning more about AI in my time here and Northeastern University and hopefully, one day, being able to contribute to the field in the manor that people like Adrei Broder, Baoguang Shi, and Alex Graves did.

References

[1] Broder, Andrei Z., et al. *Method for Selectively Restricting Access to Computer Systems*. 27 Feb. 2001.
https://patents.google.com/patent/US6195698B1/en

[2] Shi, Baoguang, Xiang Bai, and Cong Yao. "An end-to-end trainable neural network for image-based sequence recognition and its application to scene text recognition." *IEEE transactions on pattern analysis and machine intelligence* 39.11 (2016): 2298-2304.

[3] Graves, Alex & Fernández, Santiago & Gomez, Faustino & Schmidhuber, Jürgen. (2006). Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural 'networks. ICML 2006 - Proceedings of the 23rd International Conference on Machine Learning. 2006. 369-376. 10.1145/1143844.1143891.

[4] Vignesh, Aadhav. “Captcha Images.” *Kaggle*, 30 July 2020, www.kaggle.com/datasets/aadhavvignesh/captcha-images.