Breaking captcha v1.0

Andre, Anthony, Martin, Rishi, Vishnu

Objective

To train and develop a machine learning model that is capable of inferring the alphabetical characters present in Captcha v1.0

1. Data Cleaning, Pre-Processing

- Initial Dataset: 8000 training + 2000 testing images
- Cleaned Dataset: 7747 training + 1947 testing images
- Removed: watermarks, non-alphanumeric characters

Image 0, 1: Example of Manually Removed Images

Noise Reduction

- Create **binary mask** using NumPy vectorization
- Analyze 8-neighbor pixels using **rolling window**
- Color replacement using weighted average of surrounding colors

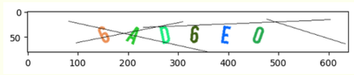

Image 3: Before Noise Reduction

Image 4: After Noise Reduction

2. Data Augmentation

Data augmentation using TensorFlow

- Brightness: ±20% variation
- Contrast: ±30% adjustment
- Translation: 10% in any direction
- Zoom: up to 20% scaling
- Rotation: ±2°

Results:

- Applied to **40% of dataset**
- Increased **training data variety**
- Maintained **character readability**

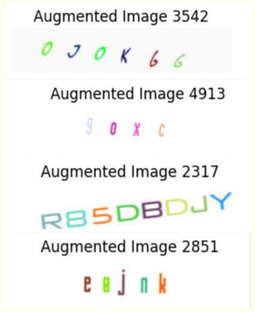

Image 4: Examples of Augmented Images

3. Character Tokenization

- We tried to identify individual character by **tokenizing letters**
- Each character = 1 Region of Interest (**ROI**)
- ROIs are extracted by:
  - **Gray-scaling** and **Binary Thresholding** to identify ROIs
  - Detecting **Contours** - boundaries of characters bigger than a fixed threshold size

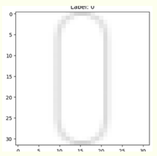

Image 5: Detecting '0'

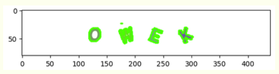

Image 6: Drawing Contours after thresholding

Image 7: Labeled Regions of interest

4. Training

- After tokenizing characters, we trained a Convolutional Neural Network (CNN).
- We employed **K-Fold Cross Validation** where **k=5** for a baseline model performance.
- Performed **hyper-parameter tuning** using GridSearchCV in order to reach higher training accuracies.

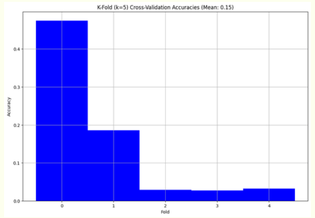

*Image 8: Cross Validation Accuracies, where lr=**0.01**, dropout=**0.5**, epoch=**10** mean accuracy = **0.15***

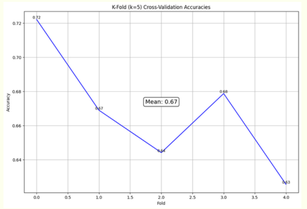

*Image 9: Cross Validation Accuracies, where lr=**0.001**, dropout=**0.2**, epoch=**10** mean accuracy = **0.67***

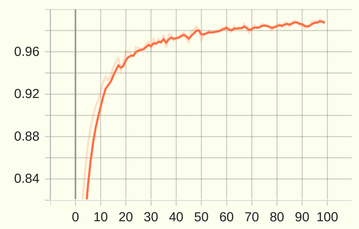

*Image 10: Plot of training accuracy over **100** epochs*

CNN Training Layers
2D Convolutional
2D Max Pooling
Batch Normalization
Flattening
Dense Layer
Dropout Layer
Output Dense Layer

5. Results

- However, we built the model to succeed on character match metric (i.e., percentage of how many characters are accurately identified), on which we scored around **>95% F1 score, Precision & Recall**
- Observation: Our F1 score is even **higher** - **almost 97%** without including the characters **"0" (zero), "o" (lowercase O), "1" (one), "l" (lowercase L), "i" (lowercase I)** which are similar-looking characters

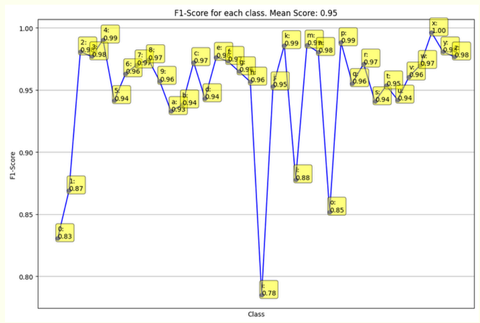

Image 11: Plot of F1 scores for each text class

6. Rejected Approaches, Learnings

- The biggest issue faced was with **over** and **under segmented characters** - multiple characters fused together or individual characters split apart
- We tried to use several methods to solve this issue:
  - **Color-based Segmentation**: Utilized **color quantization** and identify **most frequent colors** (*with K-Means Clustering*), and use that information to **separate all characters of the same color**. In theory, this would help split segmented or overlapping characters assuming they have differing colors. However, **certain characters had the exact same colors**. Additionally, the images were often **anti-aliased**, leading to **blended edges and outlines with subtle color variations**. This anti-aliasing effect, combined with varying outline colors, significantly reduced the method's reliability and effectiveness for accurate character segmentation.
  - **BRISK Algorithm:** Used the BRISK (Binary Robot Invariant Scalable Keypoints) detector to **detect key points** in the key ROIs in the image (like over-segmented characters) and **segment them using K-Means clustering** and extracted using a mask. However, despite these efforts, BRISK struggled to reliably segment letters due to **overlapping or similar features** among adjacent characters, making it ineffective for our use case.
  - **Watershed Algorithm:** Attempted to separate segments by removing noise then extracting foreground and background, and finally using watershed on the image to separate the various characters. However, this approach **proved ineffective for strongly merged letters**, as the algorithm could not reliably separate characters that were **heavily connected** or **lacked clear boundaries**, leading to inaccurate segmentation.

Image 12: Example of under-segmented characters

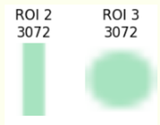

Image 13: Example of over-segmented characters

Image 14: Failed Color based Segmentation of H,O & 6

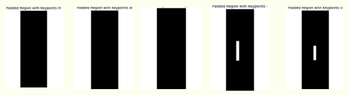

Image 15: Failed BRISK detection on segmenting "m, w, t, -, 0"

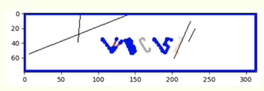

Image 16: Failed Watershed Separation