Gotcha CAPTCHA: Comparing CNN, CRNN, and Transformers

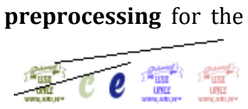
CS4243 Computer Vision and Pattern Recognition AY25/26 Semester 1

Preprocessing

We **manually** inspected the entire dataset and tagged problematic images into categories:

- **Mislabeled**: images where ground-truth text metadata is incorrect.
- **Overlapping words**: segmentation is non-trivial.
- **Non-alphanumerics:** Images to remove. (8000 → 7887 in train,
- 2000 → 1968 in test)
- **Low contrast**: characters hard to separate from background, making segmentation and recognition difficult

Rationale: manual categorization enables **targeted preprocessing** for the different issues to improve our dataset



Removal of Black Lines

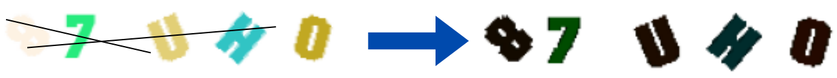
A key insight is that the **noise lines are the only black-colored** elements in the images, while the characters themselves are colored. We creates a binary pixel mask for pixels falling within a strict, near-black color threshold (RGB [0,0,0] to [8,8,8]). This mask effectively "selects" all parts of the noise lines. Then we intelligently reconstructs the masked areas hidden by the lines by propagating color information from the surrounding pixels effectively removing the noise and revealing the unobstructed characters underneath.

Enhancing Constrast

Applied **histogram equalization** to enhance image contrast while preserving chromatic information. To achieve this, images were first converted from the RGB colour space to the **YCbCr colour space**. Histogram equalization was applied to the Y (luminance) channel using cv2.equalizeHist(Y), then recombined with the original chroma channels and converted images back to RGB.

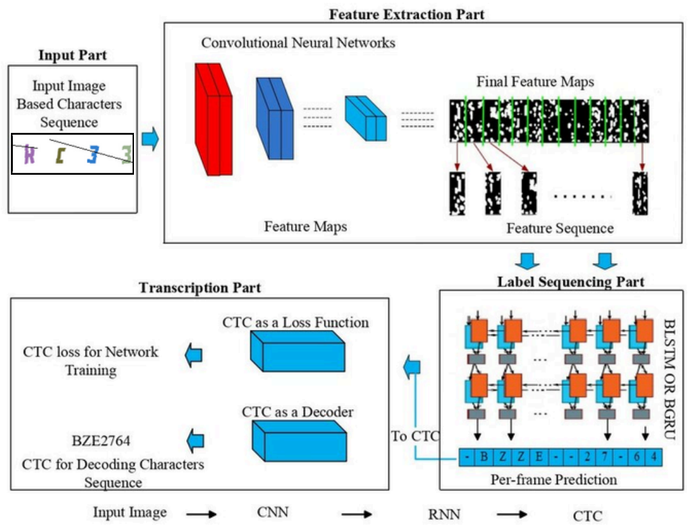
This approach produced the best results in our experiments. It was particularly effective for darker images and those exhibiting greater colour variability, as it **improved the perceptual distinction** between characters and background without distorting the hue relationships critical for later colour-based segmentation.

Transformation of preprocessing (before segmentation)



Model 1: CRNN

This architecture combines CNNs for spatial feature extraction and LSTMs for temporal modeling — a proven pattern for variable-length text recognition. The CTC loss removes the need for explicit alignment between character positions and image regions.



In evaluating the CRNN–CTC model, we found that the architecture itself trained stably and handled sequence length, distortion, and temporal ordering well. Most errors were not caused by noise or misalignment but by visually confusable character pairs such as o/0, i/l/1, and s/5. These glyphs become nearly identical after CAPTCHA warping and resizing, leading to consistent “almost right” predictions where one ambiguous character is wrong. Even when we introduced an auxiliary classification head to give the model extra supervised feedback at intermediate layers, the improvement was negligible. The auxiliary head strengthened feature learning but could not overcome the fundamental issue: the visual signal itself is too similar for these paired characters, so the model cannot disambiguate them reliably.

This pushed us toward a segmentation-based approach. Since the CRNN outputs the whole string at once, one ambiguous glyph breaks the entire result, and the auxiliary head cannot fix ambiguity rooted in the character shapes themselves. Segmenting and classifying characters individually removes alignment issues and improves fine-grained discrimination, making it the more effective next step.

Data augmentation

Augmenting training dataset

- **Smoothening edges**: used **edgePreservingFilter** → smooths flat areas but keeps strong edges sharp, so characters don’t become blurry.
- **Random rotations**: rotation angle between -20° and +20°
- **Random zoom**: zoom in or our randomly by 20% using linear interpolation
- **Adding Gaussian Noise**: convert original image to float → adding gentle random variation → converting back to pixels.

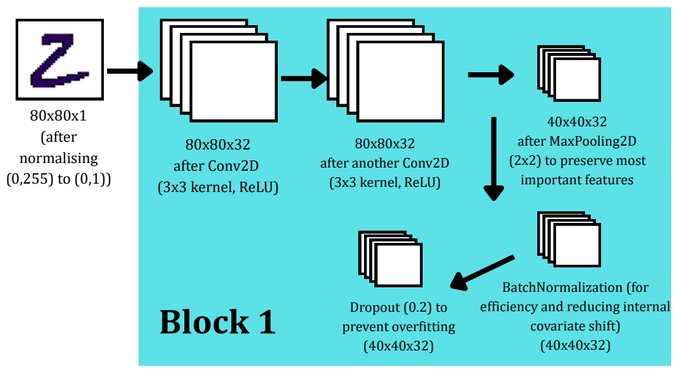
Segmentation

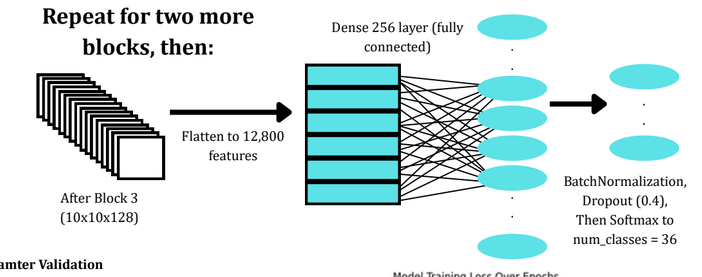
We notice that each character has a different colour, and our original implementation is to set the number of clusters based on the number of characters. However, we figured that the same segmentation needs to be done on the test set as well, and we should not peek into the label to get the number of characters, so the new segmentation method should be smart enough to split the characters without knowing the count.

We need to account for the fact that characters can overlap. The improved method:

1. **Detect connected components** (this isolates most non-overlapping characters)
2. **Detect suspiciously large components** based on some threshold, run **local KMeans colour clustering**
3. The algorithm tries several values of k and evaluates how well each clustering separates the colours inside the blob. The best clustering is selected based on how balanced and meaningful the colour groups are
4. Refine and clean up each colour cluster using morphology
5. **Extract new connected components** from each cleaned cluster, and sorted horizontally to reconstruct the captcha sequence

Model 2: CNN



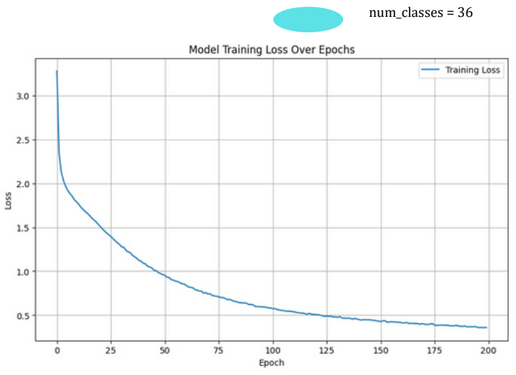


Hyperparamter Validation

Over 200 epochs (for convergence), we experimented with the following sets of hyperparameters, validating on the test set, across three different levels of preprocessing on the test set (None, Black lines removed, Enhanced constrast):

- 'learning_rate': [0.001, 0.0005, 0.0001],
- 'batch_size': [32, 64, 128],
- 'dropout_rates': [(0.2, 0.2, 0.3), (0.3, 0.3, 0.4), (0.2, 0.3, 0.5)],
- 'dense_units': [128, 256, 512]

We found that a learning rate of 0.001, batch size 32, and dropout rate 0.2, 0.2, 0.3 at bock 1, 2, and 3 respectively gave the best (smallest) Edit Distance between actual and predicted sequences (~1.9), a 75% character-level accuracy on training data, and 65% character-level accuracy on the test data.



Evaluation

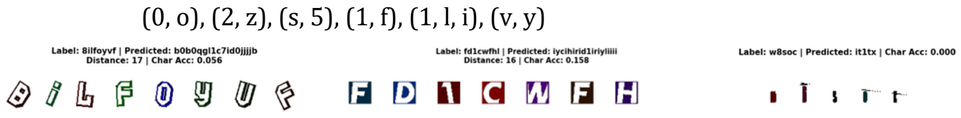
Due to the fact that segmentation may fail (cannot peek into the length of true labels in test set), we evaluate sequence-wise as well as character-wise. The following figures are performance on test data only.

CAPTCHA MODEL BENCHMARK COMPARISON

Metric	With Preprocessing	Base Model	Improvement
Exact Match Accuracy (String-level)	0.3410 (34.10%)	0.3272 (32.72%)	+0.0138 (+1.38%)
Character-Level Accuracy	0.7559 (75.59%)	0.7474 (74.74%)	+0.0085 (+0.85%)
Normalized Edit Distance (Median)	0.2000 (20.00%)	0.2000 (20.00%)	+0.0000 (+0.00%)
Levenshtein Similarity Score	0.7536 (75.36%)	0.7443 (74.43%)	+0.0093 (+0.93%)
Precision (Character-level)	0.7730 (77.30%)	0.7635 (76.35%)	+0.0095 (+0.95%)
Recall (Character-level)	0.7907 (79.07%)	0.7827 (78.27%)	+0.0080 (+0.80%)
F1 Score (Character-level)	0.7854 (78.54%)	0.7779 (77.79%)	+0.0075 (+0.75%)

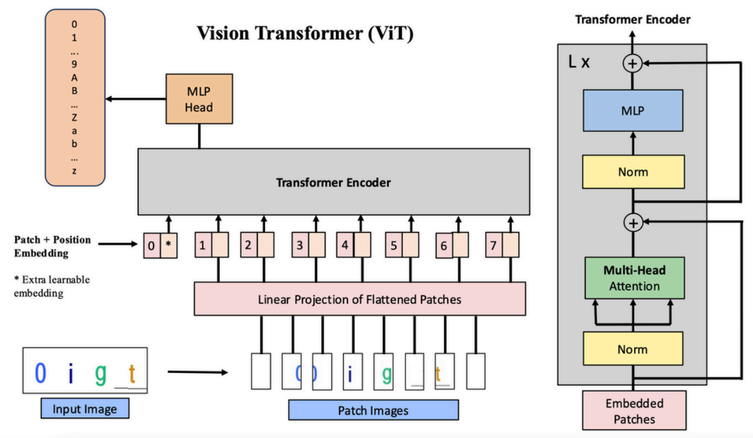
Confusion matrix

common miscategorisations:
(0, o), (2, z), (s, 5), (1, f), (1, l, i), (v, y)



Other Exploration: Using ViT

The Vision Transformer (ViT) reads a CAPTCHA image, then first split into small patches, each patch is linearly embedded into a vector, and learnable positional encodings are added so the model knows where each patch came from. These patch tokens are then processed by stacked transformer encoder layers with multi-head self-attention, allowing every patch to “look at” every other patch.



This architecture **was not selected** because its fixed patch tokenization is brittle for CAPTCHAs with variable length, spacing, warping, and noise: a single patch size cannot reliably isolate one character per token, so small patches fragment glyphs while larger patches fuse multiple characters, producing ambiguous attention and misaligned per-position logits that lead to wrong character order in the decoded sequence. This sensitivity makes training unstable across fonts and layouts, yields strong memorization but poor validation sequence accuracy, and requires costly retuning of patch size, stride, and sequence length for each dataset variant. Instead, architectures that maintain continuous convolutional features and use CTC or attention decoders provide more robust character-level alignment without depending on delicate patch granularity.

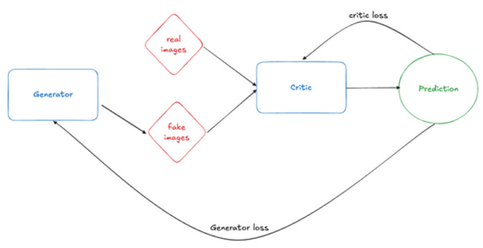
BONUS: CAPTCHA GAN

We designed a WGAN-GP to generate synthetic CAPTCHA characters. The model consists of two convolutional neural networks: a **Generator** and a **Critic**.

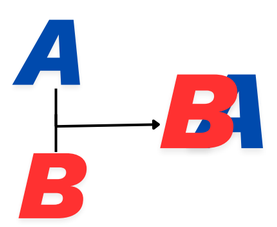
Generator converts a 100-dimensional noise vector to an 80×80 grayscale image through a series of transposed convolutional layers with batch normalization and LeakyReLU activations.

Critic outputs a score representing how realistic an image appears. Its architecture uses stacked convolutional layers followed by a linear output.

GAN Architecture



Composition of 2 images



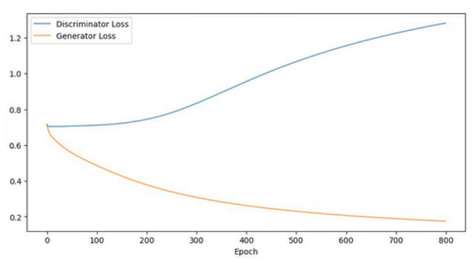
Training of WGAN-GP optimizes the Wasserstein distance between real and generated data distributions. The Critic is updated five times for every Generator step to ensure accurate gradient feedback. A gradient penalty term enforces Lipschitz continuity, stabilizing learning. Both networks use Adam optimization.

To simulate the cluttered appearance of CAPTCHA text, we introduce **C-GAN training**. After an initial warmup phase on plain images, the training data is augmented by randomly blending multiple real characters using blob-shaped masks. This forces the GAN to handle occlusion and overlapping, producing characters that are visually realistic and diverse.

The resulting model can generate varied CAPTCHA characters suitable for improving robustness in CAPTCHA recognition systems.

Generated CAPTCHA characters

GAN Training Losses



Individual Contributions:

Bryan Ho: Preprocessing and GAN
Justin Cheah: Segmentation and evaluation
Wang Hejin: Preprocessing and CRNN
Ng Chun Man: Preprocessing and Vision Transformer
Park Youngseo: CNN and evaluation